

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2223

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Tokyo

Peter Eades Tadao Takaoka (Eds.)

Algorithms and Computation

12th International Symposium, ISAAC 2001
Christchurch, New Zealand, December 19-21, 2001
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Peter Eades
University of Sydney, Basser Department of Computer Science
Madsen Building F09, Sydney, NSW 2006, Australia
E-mail: peter@cs.usyd.edu.au

Tadao Takaoka
University of Canterbury, Department of Computer Science
Private Bag 4800, Christchurch, New Zealand
E-mail: tad@cosc.canterbury.ac.nz

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algorithms and computation : 12th international symposium ; proceedings /
ISAAC 2001, Christchurch, New Zealand, December 19 - 21, 2001. Peter Eades ;
Tadao Takaoka (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong
Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2223)
ISBN 3-540-42985-9

CR Subject Classification (1998):F.2, F.1, C.2, G.2-3, I.3.5, E.1

ISSN 0302-9743

ISBN 3-540-42985-9 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN 10840949 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of the Twelfth Annual International Symposium on Algorithms and Computation (ISAAC 2001), held in Christchurch, New Zealand, 19-21 December 2001. In the past, it has been held in Tokyo (1990), Taipei (1991), Nagoya (1992), Hong Kong (1994), Cairnes (1995), Osaka (1996), Singapore (1997), Taejon, Korea (1998), Madras (1999), and Taipei (2000). Since the third in Nagoya, the proceedings have been published in Springer-Verlag's Lecture Notes in Computer Science series.

Although the symposium rotates mainly in the Asia/Pacific region, the refereeing process was conducted internationally, and researchers gathered from all over the world. We received 124 submissions from 32 countries, of which 62 papers were accepted for presentation at the symposium. The papers of three invited speakers are also included. Submission was conducted electronically by CyberChair generously offered by the University of Twente. Each paper was reviewed by three or four referees with the assistance of external referees, whose names are listed overleaf. Discussions on how to select papers were also conducted electronically over more than a week. Due to the large number of submitted papers, the reviewing process was quite challenging for the program committee. There were many acceptable papers which we could not accommodate into the time frame of the three-day symposium. The best paper award was given to "Computing the Quartet Distance between Evolutionary Trees in Time $O(n \log^2 n)$ " by Gerth Stølting Brodal, Rolf Fagerberg, and Christian Nørgaard Storm Pedersen. We hope all accepted papers will eventually appear in scientific journals in a more polished form.

In the era of the Internet, we tend to think we can exchange ideas with other researchers instantly without needing to meet. This way we tend to be isolated unknowingly. Because of the Internet, however, the need for meeting other people face to face is ever increasing. To do research, it is still best to see our peers directly, and get ideas from the very inventors. We hope ISAAC 2001 made a good forum for researchers in algorithms and computation to meet, being held in New Zealand, a beautiful small nation of 3.8 million people. It is roughly at the center of the water hemisphere, meaning that it is farthest from the rest of the world. People can escape from their daily businesses, meet people, and refresh their thinking.

We thank all the authors who submitted papers and contributed to the high quality of the symposium. We thank all the organizing committee members, program committee members, and external referees who sacrificed their time for the symposium. We thank all the sponsors who made the symposium financially viable. Our special thanks go to Adrian White, who installed CyberChair, Shane Saunders, who maintained it, and all our colleagues and students, who spent many hours preparing for this event.

December 2001

Peter Eades
Tadao Takaoka

Organization

Program Committee

Prosenjit Bose,	Carleton University, Canada
Richard Brent,	Oxford University, UK
Maxime Crochemore,	Univ. de Marne la Vallée, France
Peter Eades,	Univ. Sydney, Australia, co-chair
Xin He,	SUNY Buffalo, USA
Michael Houle,	IBM Tokyo Research Lab., Japan
Tsan-sheng Hsu,	Academia Sinica, Taiwan
Kazuo Iwama,	Kyoto University, Japan
Tao Jiang,	UC Riverside, USA
Tak-wah Lam,	Univ. Hong Kong, Hong Kong
Hon Wai Leong,	Nat. Univ. Singapore, Singapore
Akira Maruoka,	Tohoku University, Japan
Ernst W. Mayr,	TU München, Germany
Mitsunori Ogihara,	Univ. Rochester, USA
Kunsoo Park,	Seoul National University, Korea
Frank Ruskey,	Univ. Victoria, Canada
Igor Shparlinski,	Macquarie University, Australia
Mike Steel,	Univ. Canterbury, New Zealand
Hisao Tamaki,	Meiji University, Japan
Tadao Takaoka,	Univ. Canterbury, NZ, co-chair
Shmuel Zaks,	Technion, Israel

Organizing Committee, all Univ. Canterbury

Tim Bell,	Treasurer
Yoichi Hayashi,	Publicity
Mike Steel,	Local Arrangement
Tadao Takaoka,	Chair
Tim Wright,	Technical Support

Referees

Kazuyuki Amano	Seokhee Hong	Hadas Shachnai
Masami Amano	Nigel Horspool	Mordecai Shalom
Chuan Heng Ang	Sun-yuan Hsieh	Tetsuo Shibuya
Feng Bao	Jing Huang	Jia Shu
Thomas Bayer	Hiro Ito	Mihaela Sighireanu
Markus Blaeser	David Juedes	Alan Sprague
Beate Bollig	Sung-Kwon Kim	Subhash Suri
Rainer E. Burkard	Ines Klimann	Eiji Takimoto
Weidong Cai	Ming-Tat Ko	Akihisa Tamura
W.T. Chan	Yi-Zhi Lao	H.F. Ting
Ee-Chien Chang	Thierry Lecroq	Takeshi Tokuyama
K.W. Chong	Xuemin Lin	Godfried Toussaint
Kyung-Yong Chwa	Bruce Litow	Peter Ullrich
Artur Czumaj	Chor-Ping Low	Rakesh Verma
Vladimir Deineko	Chi-Jen Lu	Ulrich Voll
William Duckworth	Ioan Macarie	Klaus Wagner
Scott Effler	Jean Mairesse	Da-Wei Wang
John Ellis	Kazuhisa Makino	Huaxiong Wang
Jens Ernst	Kenshi Matsuo	Todd Wareham
Patricia Evans	Elvira Mayordomo	Tatsuya Watanabe
Hivert Florent	Sylvain Michelin	John Watrous
Lance Fortnow	Michael Miller	Mark Weston
Satoshi Fujita	Michal Mruk	Sue Whitesides
P.Y. Fung	Yoan J. Pinzon	W.H. Wong
Dan Gordon	Liwen Qian	David Wood
Roberto Grossi	Martin Raab	Shao-Fa Yang
Qian Oing Gu	Kenneth W. Regan	S.M. Yiu
Kiyoharu Hamaguchi	Alex Rosa	Xinguo Yu
Patrick Healy	Kunihiko Sadakane	Marius Zimand
Edith Hemaspaandra	Marie-France Sagot	Min Zou
Volker Heun	Joe Sawada	
Klaus Holzapfel	Mark Scharbrodt	

VIII Organization

Sponsors

University of Canterbury
Lincoln University
Allied Telesyn Research, Ltd.
JADE
C-fACS
Hitachi Naka Techno Center
Information Processing Society of Japan
New Zealand Computer Society
University of Canterbury Student Chapter of the ACM

Table of Contents

Invited Talk 1

Chain Reconfiguration: The Ins and Outs, Ups and Downs of Moving Polygons and Polygonal Linkages	1
<i>Sue Whitesides</i>	

Combinatorial Generation and Optimization (I)

Application of M-Convex Submodular Flow Problem to Mathematical Economics	14
<i>Kazuo Murota and Akihisa Tamura</i>	
A Polynomial Time Approximation Scheme for Minimizing Total Completion Time of Unbounded Batch Scheduling	26
<i>Xiaotie Deng, Haodi Feng, Pixing Zhang, and Hong Zhu</i>	
A Polynomial Time Approximation Scheme for the Multi-vehicle Scheduling Problem on a Path with Release and Handling Times	36
<i>Yoshiyuki Karuno and Hiroshi Nagamochi</i>	
Semi-normal Schedulings: Improvement on Goemans' Algorithm	48
<i>Jianer Chen and Jingui Huang</i>	

Parallel and Distributed Algorithms (I)

Balanced Scheduling toward Loss-Free Packet Queuing and Delay Fairness	61
<i>Hisashi Koga</i>	
Broadcasting with Universal Lists Revisited: Using Competitive Analysis	74
<i>Jae-Hoon Kim and Kyung-Yong Chwa</i>	
On Adaptive Fault Diagnosis for Multiprocessor Systems	86
<i>Kumiko Nomura, Toshinori Yamada, and Shuichi Ueno</i>	
On-Line Multicasting in All-Optical Networks	99
<i>Kenta Hashimoto, Toshinori Yamada, and Shuichi Ueno</i>	

Graph Drawing and Algorithms (I)

Enumerating Floorplans with n Rooms	107
<i>Shin-ichi Nakano</i>	
On Min-Max Cycle Bases	116
<i>Giulia Galbiati</i>	
On the Minimum Local-Vertex-Connectivity Augmentation in Graphs	124
<i>Hiroshi Nagamochi and Toshimasa Ishii</i>	
Counting Graph Isomorphisms among Chordal Graphs with Restricted Clique Number	136
<i>Takayuki Nagoya</i>	

Computational Geometry (I)

Quantum Algorithms for Intersection and Proximity Problems	148
<i>Kunihiko Sadakane, Norito Sugawara, and Takeshi Tokuyama</i>	
BUSHWHACK: An Approximation Algorithm for Minimal Paths through Pseudo-Euclidean Spaces	160
<i>Zheng Sun and John Reif</i>	
Approximation of Minimum Triangulation for Polyhedron with Bounded Degrees	172
<i>Francis Y. L. Chin and Stanley P. Y. Fung</i>	
Tree-Approximations for the Weighted Cost-Distance Problem (Extended Abstract)	185
<i>Christian Schindelhauer and Birgitta Weber</i>	

Computational Complexity and Cryptology

Necessary and Sufficient Numbers of Cards for Sharing Secret Keys on Hierarchical Groups (Extended Abstract)	196
<i>Takaaki Mizuki and Takao Nishizeki</i>	
Generic Algorithms and Key Agreement Protocols Based on Group Actions	208
<i>Akihiro Yamamura and Kaoru Kurosawa</i>	
Baire Category and Nowhere Differentiability for Feasible Real Functions	219
<i>Josef M. Breutzmann, David W. Juedes, and Jack H. Lutz</i>	
Translation among CNFs, Characteristic Models and Ordered Binary Decision Diagrams	231
<i>Takashi Horiyama and Toshihide Ibaraki</i>	

Automata and Formal Languages

On Removing the Pushdown Stack in Reachability Constructions	244
<i>Oscar H. Ibarra and Zhe Dang</i>	
A New Recognition Algorithm for Extended Regular Expressions	257
<i>Hiroaki Yamamoto</i>	
Polynomial-Time Algorithms for the Equivalence for One-Way Quantum Finite Automata	268
<i>Takeshi Koshihara</i>	
An Index for the Data Size to Extract Decomposable Structures in LAD	279
<i>Hiroataka Ono, Mutsunori Yagiura, and Toshihide Ibaraki</i>	

Invited Talk 2

Parameterized Complexity: The Main Ideas and Some Research Frontiers	291
<i>Michael R. Fellows</i>	

Graph Drawing and Algorithms (II)

Tight Bounds on Maximal and Maximum Matchings	308
<i>Therese Biedl, Erik D. Demaine, Christian A. Duncan, Rudolf Fleischer, and Stephen G. Kobourov</i>	
Recognition and Orientation Algorithms for P_4 -Comparability Graphs	320
<i>Stavros D. Nikolopoulos and Leonidas Palios</i>	
Efficient Algorithms for k -Terminal Cuts on Planar Graphs	332
<i>Danny Z. Chen and Xiaodong Wu</i>	
Polynomial Time Algorithms for Edge-Connectivity Augmentation of Hamiltonian Paths	345
<i>Anna Galluccio and Guido Proietti</i>	

Combinatorial Generation and Optimization (II)

Algorithms for Pattern Involvement in Permutations	355
<i>Michael H. Albert, Robert E. L. Aldred, Mike D. Atkinson, and Derek A. Holton</i>	
A Fast Algorithm for Enumerating Bipartite Perfect Matchings	367
<i>Takeaki Uno</i>	

XII Table of Contents

On-Line Scheduling a Batch Processing System to Minimize Total Weighted Job Completion Time	380
<i>Bo Chen, Xiaotie Deng, and Wenan Zang</i>	

On the Complexity of Train Assignment Problems	390
<i>Thomas Erlebach, Martin Gantenbein, Daniel Hürlimann, Gabriele Neyer, Aris Pagourtzis, Paolo Penna, Konrad Schlude, Kathleen Steinhöfel, David Scot Taylor, and Peter Widmayer</i>	

Computational Biology and String Matching (I)

A Combinatorial Toolbox for Protein Sequence Design and Landscape Analysis in the Grand Canonical Model	403
<i>James Aspnes, Julia Hartling, Ming-Yang Kao, Junhyong Kim, and Gauri Shah</i>	

Complexity of Comparing Hidden Markov Models	416
<i>Rune B. Lyngsø and Christian N. S. Pedersen</i>	

DNA Self-Assembly For Constructing 3D Boxes (Extended Abstract)	429
<i>Ming-Yang Kao and Vijay Ramachandran</i>	

Exact Solutions for CLOSEST STRING and Related Problems	441
<i>Jens Gramm, Rolf Niedermeier, and Peter Rossmanith</i>	

Computational Geometry (II)

Topological Peeling and Implementation	454
<i>Danny Z. Chen, Shuang Luan, and Jinhui Xu</i>	

Image Segmentation with Monotonicity and Smoothness Constraints	467
<i>Danny Z. Chen, Jie Wang, and Xiaodong Wu</i>	

Optimization Algorithms for Sweeping a Polygonal Region with Mobile Guards	480
<i>Jae-Ha Lee, Sang-Min Park, and Kyung-Yong Chwa</i>	

Approximation of a Geometric Set Covering Problem	493
<i>Sofia Kovaleva and Frits C. R. Spijksma</i>	

Invited Talk 3

Shortest Path Algorithms: Engineering Aspects	502
<i>Andrew V. Goldberg</i>	

Graph Drawing and Algorithms (III)

Efficient Algorithms for Weighted Colorings of Series-Parallel Graphs 514

Xiao Zhou and Takao Nishizeki

Go with the Winners Algorithms for Cliques in Random Graphs 525

Marcus Peinado

Complexity of Partial Covers of Graphs 537

Jiří Fiala and Jan Kratochvíl

On Game-Theoretic Models of Networks 550

*Hans L. Bodlaender, Michael J. Dinneen, and Bakhadyr Khoussainov***Parallel and Distributed Algorithms (II)**

The Complexity of Some Basic Problems for Dynamic Process Graphs 562

Andreas Jakoby and Maciej Liśkiewicz

Delay Optimizations in Quorum Consensus 575

*Xuemin Lin*Randomized Shared Queues Applied
to Distributed Optimization Algorithms 587*Hyunyoung Lee and Jennifer L. Welch*

Multiprocess Time Queue 599

*Andrej Brodnik and Johan Karlsson***Computational Geometry (III)**

Labeling Points with Weights 610

Sheung-Hung Poon, Chan-Su Shin, Tycho Strijk, and Alexander Wolff

Small Convex Quadrangulations of Point Sets 623

*David Bremner, Ferran Hurtado,
Suneeta Ramaswami, and Vera Sacristán*How to Color a Checkerboard with a Given Distribution –
Matrix Rounding Achieving Low 2×2 -Discrepancy 636*Tetsuo Asano and Takeshi Tokuyama*

Labeling Subway Lines 649

*Mari Ángeles Garrido, Claudia Iturriaga, Alberto Márquez,
José Ramón Portillo, Pedro Reyes, and Alexander Wolff***Randomized and Approximation Algorithms**

Complexity Study on Two Clustering Problems (Extended Abstract) 660

Louxin Zhang and Song Zhu

XIV Table of Contents

A Modified Greedy Algorithm for the Set Cover Problem with Weights 1 and 2	670
<i>Toshihiro Fujito and Tsuyoshi Okumura</i>	

A Unified Framework for Approximating Multiway Partition Problems (Extended Abstract)	682
<i>Liang Zhao, Hiroshi Nagamochi, and Toshihide Ibaraki</i>	

On-Line Algorithms for Cardinality Constrained Bin Packing Problems ...	695
<i>Luitpold Babel, Bo Chen, Hans Kellerer, and Vladimir Kotov</i>	

Computational Biology and String Matching (II)

Suffix Vector: A Space-Efficient Suffix Tree Representation	707
<i>Krisztián Monostori, Arkady Zaslavsky, and István Vajk</i>	

Fragmentary Pattern Matching: Complexity, Algorithms and Applications for Analyzing Classic Literary Works	719
<i>Hideaki Hori, Shinichi Shimozone, Masayuki Takeda, and Ayumi Shinohara</i>	

Computing the Quartet Distance between Evolutionary Trees in Time $O(n \log^2 n)$	731
<i>Gerth Stølting Brodal, Rolf Fagerberg, and Christian N. S. Pedersen</i>	

Algorithms and Data Structures

The <i>Cent-dian</i> Path Problem on Tree Networks	743
<i>Ronald I. Becker, Yen-I Chiang, Isabella Lari, and Andrea Scozzari</i>	

Approximate Hotlink Assignment	756
<i>Evangelos Kranakis, Danny Krizanc, and Sunil Shende</i>	

Efficient Algorithms for Two Generalized 2-Median Problems on Trees	768
<i>Shan-Chyun Ku, Chi-Jen Lu, Bing-Feng Wang, and Tzu-Chin Lin</i>	

Author Index	779

Chain Reconfiguration

The Ins and Outs, Ups and Downs of Moving Polygons and Polygonal Linkages

Sue Whitesides*

School of Computer Science, McGill University
Montreal, Canada H3A 2A7
sue@cs.mcgill.ca

Abstract. A *polygonal linkage* or *chain* is a sequence of segments of fixed lengths, free to turn about their endpoints, which act as joints. This paper reviews some results in chain reconfiguration and highlights several open problems¹

We consider a sequence of closed straight line segments $[A_0, A_1]$, $[A_1, A_2]$, $\dots$ $[A_{n-1}, A_n]$ of fixed lengths $l_1, l_2, \dots, l_n$, respectively, imagining that these line segments are mechanical objects such as rods, and that their endpoints are joints about which these rods are free to turn. We ask how and whether such a *chain* can be moved from one given configuration to another under various assumptions or “rules of the game”. The chain may be confined to the plane throughout its motions; it may be supposed to start and finish in the plane, with motion into 3D allowed for intermediate configurations; its motions in arbitrary dimensional space may be considered. The chain may consist of an open or closed sequence of segments. The links may be allowed or forbidden to cross over or to pass through one another. All of these models are of interest to us.

When the chain consists of a closed sequence of links, we say that the chain is polygonal, or that it is a *polygon*. Consequently, it is natural to use both the language of geometry and mechanics when describing chains. The terms *node*, *vertex* and *joint* are used interchangeably, as are the terms *rod*, *link*, *edge*, and *segment*. The term “polygon” may refer either to a planar object or to a cyclic sequence of links in arbitrary dimension. In case the links are not allowed to intersect except at shared endpoints, we say that the polygon must remain *simple*, i.e., it is not allowed to intersect itself either at rest or during motion.

Polygonal chains are interesting for several reasons. First, there are aesthetic reasons. These very basic objects exhibit surprising behaviors and pose challenging, easily stated questions that arouse our natural curiosity as mathematicians, algorithm designers, and problem solvers. Second, chains can model physical objects such as robot arms and molecules. Here, a word of caution is in order. In

* Research supported by FCAR and NSERC.

¹ A preliminary version of this paper was presented to AWOCA '92, the 12th Australasian Workshop on Combinatorial Algorithms, Lembang, Indonesia, July 14-17, hosted by the Bandung Institute of Technology.

a “real-world” context, our geometric models are often gross simplifications of complex systems. Mechanical robot arms have mass and inertia, they vibrate, their joints are not universal joints, and they don’t have an arbitrary number n of links. In 3D, mechanical links cannot pass through one another, although in 2D, allowing links to pass over one another can model so-called $2\frac{1}{2}$ -D situations where long links remain parallel to the plane, joined by short connections not parallel to the plane. For molecules, preferred configurations (“conformations” being the technical term in the chemistry and physics literature) depend on much more than geometry, low energy conformations being the preferred ones. The energy depends on contributions from bonds (modeled by links) which may stretch and dihedral angles (angles between the two planes determined by three consecutive bonds) which may deform. Then there are the chemical interactions between individual pairs of atoms that are not connected by a bond and that may be far from one another in the graph theoretic sense. While it seems unrealistic to suppose that results about purely geometric models are likely to find immediate and wide-spread application in other fields, it seems equally unwise to suppose that geometric studies cannot be relevant or useful. We do not survey application areas, or even potential application areas here, but mention a few pointers: for connections with molecular modelling, see for example [12,13,14,33,34]; for connections with algorithmic motion planning, see for example [24]; for connections with manufacturing, see for example [28]. In later sections, we mention some results on chains that have connections with knot theory [4] and rigidity theory [7,35].

This survey is a personal account, inevitably biased and incomplete. The intent is to focus on the developments in chain reconfiguration in the last several years, highlighting some interesting open problems. For an earlier survey, see [37].

My introduction to the subject began during 1981-82, a year which I spent visiting the Computer Science Department at Cornell University. There, John Hopcroft was working on robotics problems and suggested I read a preliminary version of Schwarz and Sharir, Piano Movers II [34]. Somewhat daunted by the length, and the fact that it was algebraic geometry, I proposed a simple problem as an alternative way to start our discussions, a problem that we later began calling the “ruler folding” problem, or the “carpenter’s ruler” problem. This terminology is now used to refer to a variety of chain reconfiguration problems; the original problem (see [15,17]) was the following.

Ruler Folding:

given: a sequence of n positive integer lengths, and a positive integer k ;

question: Can a sequence of links of these lengths, hinged at their endpoints, be folded so that they occupy a segment of length at most k ? Here, each joint is to be completely straight, or completely folded.

This problem, and the corresponding optimization problem of finding the minimum folding length, make excellent undergraduate student exercises. While the carpenter’s ruler is an easy-to-grasp object of study, to determine its properties raises in a simple setting a variety of issues in algorithm analysis and design. Clearly the answer may be determined by trying all the 2^{n-1} ways of folding the

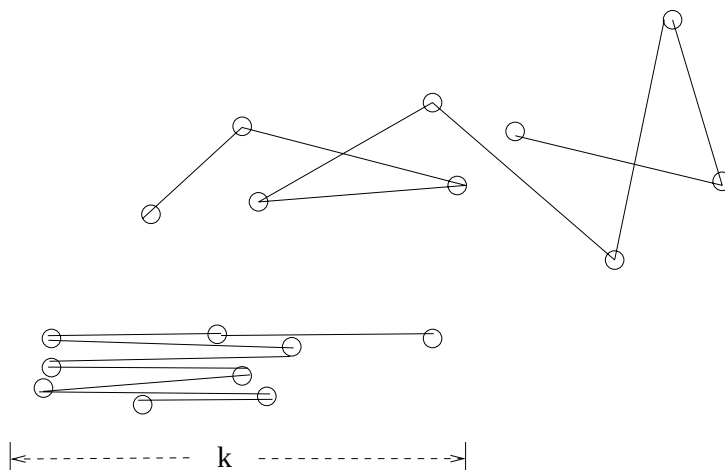


Fig. 1. Ruler Folding

ruler. However, the problem turns out to be NP-complete, by an easy reduction from the Set Partition problem. On the other hand, a simple greedy strategy gives an approximation algorithm to within a factor of 2 for the corresponding optimization version (finding the minimum folding length), and the problem can be solved in $O(n)$ time by dynamic programming when the lengths of the links are bounded above. There is more than one natural way to do this, and designing a second way having seen a first way makes an excellent exercise for students, as does analysing the running time on a Turing machine model of computation when link lengths are unbounded integers.

While the optimization version of the problem can be solved by creating a table or series of tables, some subproblems of a problem instance may not be solved optimally in *any* optimal solution of that instance. Hence the table solution is not, strictly speaking, dynamic programming as it is sometimes described [8]. Finding such examples makes a nice exercise, as does finding examples that show that, for n a power of 2, a simple divide-and-conquer approach doesn't work.

The dynamic programming solutions can be regarded as *fixed parameter tractability* results, in the sense of Downey and Fellows [9]. The idea is to confine the exponential growth in running time to a parameter of the problem that is likely to remain small for typical applications. For Ruler Folding, the dynamic programming solutions allow us to “blame” exponential running time growth on long links; if no links are very long, the solution grows linearly with the length of the input string. To sketch how this goes, let M denote the length of the longest link, let x denote the number of bits in the input data, and let n denote the number of links in the ruler. One dynamic programming method builds a series of roughly M tables, trying all possible folding lengths in the range M to $2M$. Each table has $O(n)$ rows (one for each joint) and $O(M)$ columns (one for

each possible integer coordinate for a joint in the range 0 to the folding length being tried). Computing each table entry takes $O(lgM)$ bit operations, giving an overall running time of $O(xM^2lgM)$. Note that the integer M is no greater than 2^x , since M must be given by at most x bits. Hence the running time is indeed exponential in x . Now consider all rulers that have n links but no link longer than 1000, say. For this restricted set of very reasonable instances, there is a running time upper bound that is linear in the length of the input x (and linear in the number n of links). Thus the exponential behaviour of the algorithm is the “fault” of the possibility of very long links in the general case.

The NP-completeness of Ruler Folding has some consequences for the complexity of related problems, such as determining whether or not a polygonal chain (allowed to cross or not) can be moved from one configuration to another in an environment containing polygonal obstacles. Here, one may design an instance containing a narrow gap such that the chain must folded, or nearly so, into length at most some given amount in order to fit through a narrow passage (see [15]). Furthermore, the NP-completeness of Ruler Folding can be used to show the hardness of the placement of graphs, even trees, having edges of specified lengths so that certain vertices are placed at certain points (see [39,37,38]). This type of problem is sometimes stated in terms of the realizability of distance matrices, where the entries in the matrix give the desired Euclidean distances between certain pairs of vertices in the graph.

In our first investigations of movement properties of chains, we required one end of the chain to be fixed to the plane, and we allowed links to cross over one another as this so-called “arm” moved. Determining what points of the plane can be reached by the opposite end of the chain makes a nice easy exercise for students, as does proving that any point that can be reached at all can be reached by a configuration of the arm that has at most two non-straight joints. This can be seen by thinking in terms of the polar coordinates of the point to be reached relative to an origin at the fixed point of the arm: first find a way to achieve the correct distance between the free end and the fixed end, and then rotate the arm about the origin to move the tip to the desired point. Textbooks that have included some of these problems as exercises include [21,27].

Problems for the reconfiguration of a chain (or an *arm*, as we call a chain with the location of one end fixed) get decidedly more interesting in the presence of obstacles or other constraints. Our first effort to deal with obstacles in the “workspace” of a robot “arm” was to consider the problem of moving an arm confined to a closed disk from one given configuration to another. Links were allowed to cross, and we found a polynomial time algorithm for determining if the desired configuration could be reached from the initial one, and, when this was possible, for designing a specific motion. Here we made a distinction between the running time for computing the motion, and the length of the description of the motion, which we gave in terms of simple motion primitives. These primitives were described, for example, in terms of rotating a link about a fixed endpoint while other joints were fixed in position or joint angles were frozen and dragged along. The basic idea was simple: find a way to move the arm to a “canonical”

position in which all joints at and beyond some joint A_i are placed on the boundary of the circle, regard the initial portion of the chain as a kind of leash whose length can be adjusted by adding or removing links, and then, by adjusting the length of the leash, rotate the remainder of the chain around the circle. A sequence of links in the tail can then, one hopes, be formed and lifted up to reach points in the disk. Kantabutra and Kosaraju [20] pushed this technique and improved the running time of our algorithm.

We studied chains and arms in other confining regions such as convex polygons. We were not able to answer the reconfiguration question even for a triangle, a problem that remains open.

Problem (chain in polygon): Given two configurations of a chain confined to a convex polygonal region, with edge crossings allowed, determine whether the chain can be moved inside the polygon from one configuration to the other.

Kantabutra then went on to explore the use of this general strategy inside other shapes, in particular, a square [18,19]. Here, the links do not rotate around the boundary so conveniently as is the case for the circle. He was able to obtain reconfiguration and reachability results for arms and chains satisfying a bound on the link lengths in terms of the length of a side of the square.

Inspired by Kantabutra's success with squares, the case of arms and chains confined inside triangles seemed interesting to try again. In 1991, Peter Eades and I tried an even simpler version of this case: an arm confined to a wedge. Here, we asked whether and how an arm could be straightened, with links allowed to cross, and were able to solve this problem when the internal wedge angle is $\pi/2$ or greater (the proof, which remains unpublished, was given at AWOCA '92). The case of the acute wedge still remains unsolved. Perhaps the problem is NP-hard. The version of the problem in which links are not allowed to cross may also be interesting.

Problem (arm in a wedge): Given a chain with one extreme end fixed to the plane, and confined to move inside a wedge whose vertex angle is less than $\pi/2$, design an algorithm to decide whether the arm can be straightened, and to move it to such a position when this is possible.

In March 1991, Bill Lenhart came to visit McGill, and we became fascinated with a different problem, which we came to call "turning a polygon inside-out". We decided to drop the idea of exploring the motion of chains or arms confined inside polygonal environments. Instead, we would get rid of the obstacles, but pin down both the endpoints. We quickly noticed that the motions of a chain with both endpoints fixed down correspond to the motions of a closed polygonal linkage.

When this linkage takes the form of a *simple* polygon, it has a natural orientation, as traversing the boundary in the clockwise sense visits the vertices either in increasing or decreasing order of their indices. Obviously, a triangle whose vertices are indexed cannot be moved in the plane so that its orientation changes (disallowing flips out of the plane). Hence the inside-out problem: given a simple polygon lying in the plane, when can it be moved in the plane to its

mirror image? More generally, given two configurations of a polygonal linkage in the plane, when can the linkage be moved from the one configuration to the other? We called two configurations *equivalent* if it is possible to move the linkage between the two configurations, and asked for the number of equivalence classes of polygonal linkages in the plane. The answer turned out to be pretty. The number of classes is either two or one, depending on whether the second and third largest link lengths sum to more than the length of the longest link. Thus a triangle has two equivalence classes, as its longest side has length less than the sum of the lengths of the other two sides. Furthermore, as is the case for the triangle, each configuration in one class has a mirror image in the other class. We designed algorithms for reconfiguring when possible, and eventually noticed that our reconfiguration strategy in 2D always worked in dimension 3 and above, so we obtained a little bonus: polygonal linkages have one equivalence class of configurations in dimension 3 and above.

One of our discussions about unconfined chains led us to some interesting reconfiguration problems for chains *not* allowed to intersect themselves. And what about 3D? We had a simple plan: project the 3D chain to 2D, and reconfigure the shadow in 2D to guide the reconfiguration of the 3D object. There were some difficulties with this, however. Suppose you tie a knot in your shoe lace, and attach a long knitting needle to each end of the lace. You won't be able to undo the "knot", even though it's not a knot in the mathematical sense. It's easy to imagine that a chain of links could be configured like such a knot, with very long links attached at the ends, to give a configuration of an open chain of links in 3D that cannot be straightened. Worse, we realized that even if a chain in 3D had a simple projection onto a plane, we didn't see how to straighten a simple chain in the plane. This led us to several chain straightening and polygon convexifying problems for linkages in the plane, with links not allowed to cross. My colleague Godfried Toussaint at McGill was very enthusiastic about these problems and suggested that we try to convexify *star-shaped* polygons, that is, polygons containing a non-empty "kernel" of points that can "see" all the points in the polygon.

Encouraged by Godfried's enthusiasm, we began describing these problems at every opportunity, beginning with Bill's seminar at McGill in March 1991, and later in August 1991 in our talk at the Canadian Computational Geometry Conference (CCCG '91) in Vancouver, for example, and at my AWOCA '92 sessions, where the chain straightening problem was on the handout of problems given in the problem session, and again in our talk at CCCG '92. We also described these problems in our 1993 McGill technical report. When our journal article based on turning a polygon inside-out ([25]) finally appeared in 1995, we mentioned at the very end that these problems still had not been solved.

Bill and I were not aware in 1991 that this kind of problem had been posed by topologists in the 1970's (see the discussion of the history of the problem in [7]). However, as far as we know, our 1993 McGill University technical report [25] was the first written, publicly accessible statement of these problems. In the computational geometry community, the chain-straightening problem was again

independently rediscovered by Joe Mitchell in 1992, in the context of a tube manufacturability problem. Joe was active in generating possibilities of chains that might not be straightenable in the plane, and discussed these with a number of geometers.

In the spirit of Paul Erdős, I offered a prize of one bottle of Bintang, the local beer, for AWOCA '92 participants who solved the open problems about linkages on the handout. One of the participants who took this offer seriously was Heiko Schröder, who was then at the University of Queensland, where I visited later, in December 1992. In fact, I rented Heiko's apartment, as he was on sabbatical, travelling in Europe. We started corresponding via the fax machine in his apartment about the star-shaped polygon problem. He would propose an idea and fax it to me, with pictures. I would fax back a reply. The difference in time zones made life interesting. Sometimes I would hear the fax machine grinding away at 3 a.m. Brisbane time, and leap out of bed to see what Heiko's latest idea was for winning a Bintang. Always there was some little problem. We tried hard to do a proof by induction: straighten some joint and continue on a simple polygon with fewer vertices. We tried drawing rays through every other vertex, hoping to move some pair of rays apart so that the 2-link sub-chain contained in the wedge bounded by the rays would straighten. All these efforts led to technical difficulties, such as degenerate cases to handle, or problems keeping the chain simple or the polygon star-shaped.

Eventually, Heiko proposed moving the vertices on the rays simultaneously outward along their rays at constant speed. This was quite a novel idea, since it moved an unbounded number of vertices at the same time and had a distinctly different flavor than that of chain reconfiguration algorithms whose motion primitives were localized. Surely such an expansive motion would convexify the linkage, and it had the intuitive appeal that the algorithm sort of "inflated" the polygon. We outlined a formal proof, whose details we never completed. Still, what we had seemed fairly convincing, and I promised Heiko a Bintang.

In 1992-93, Mark van Kreveld of the University of Utrecht, The Netherlands, came to McGill for a postdoc. He sportingly took a look at some chain reconfiguration problems in the plane, with links allowed to cross; in particular, he looked at problems for chains confined to polygons and/or the wedge problem. When these seemingly simple problems proved surprisingly difficult, he called on a heavy-duty weapon, Jack Snoeyink, then of the University of British Columbia. Mark and Jack proposed a new, related problem, that of folding a chain whose links all have the same length, an *equilateral chain*. If one could fold the entire chain onto a single link, one could then hope to move this link around in a containing environment to a position where it could be unfolded to give the desired configuration for the whole chain. Thus, instead of using a completely straightened configuration as an intermediate configuration between initial and desired final configurations, one would use a completely folded intermediate configuration, which seemed sensible in the context of a containing environment. Studying a polygonal environment consisting of an equilateral triangle of side length 1, we found a surprising alternation property for the foldability of equilateral chains

of link-length $l < 1$. Whether or not *every* configuration of such a chain can be folded to a single link is a property that changes *three* times as l increases from 0 to 1. For very small links with l close to 0, every configuration can be folded to a segment of length l , and for chains with l close to 1, not all configurations can be folded. Surprisingly, as l increases from 0 to 1, the foldability of equilateral n -link chains changes from always foldable, to not always foldable, to always foldable, and finally back to not always foldable. See [22] and for another example of alternation, [30].

Recalling Kantabutra's success with reconfiguring chains in squares, my student Naixan Pei and I decided to push farther the strategy of moving the chain to the boundary, and rotating it around the boundary to a position that would enable one end to reach out to touch a desired point. We were able to generalize some of Kantabutra's results to the case of *convex obtuse polygons*. These are convex polygons, not necessarily regular, such that each internal vertex angle is equal to or greater than $\Pi/2$. See [29,30,31,32].

In 1998, Anna Lubiw of the University of Waterloo and I led a small workshop on "Wrapping and Folding" (or, alternatively, "Unwrapping and Unfolding") at McGill's Bellairs Research Institute in Barbados. She and her collaborators, including Joe O'Rourke and Erik Demaine, who was her doctoral student, had obtained some interesting results having a flavor of origami. Her view was that chain reconfiguration was a kind of origami for a lower dimensional object, a line instead of a piece of paper. At this workshop, we revisited chain reconfiguration problems, this time insisting that links not be allowed to cross.

Our first result [1], produced by all the workshop participants in a real group effort, was inspired by a chain configuration that Joe Mitchell had proposed as being possibly unstraightenable in the plane. It seemed to us that this particular example could, however, be straightened, so instead, we explored the possibility that a *tree linkage* based on Joe's chain pattern could not be straightened. Here, straightening a tree means to choose some node as a root and then to move the linkage so that all root-leaf paths form essentially straight lines emanating from the root. Indeed, we found that there are tree linkages that cannot be straightened in the plane and that can exhibit exponentially many equivalence classes of configurations [1]. Here are some questions arising from this work.

Problems (tree linkages): What is the complexity of deciding whether or not a tree configuration can be straightened? Design an algorithm to do this when possible. Can every configuration of a tree linkage whose links all have length 1 be straightened in the plane?

Another idea that our 1998 Folding and Wrapping workshop explored was a line of thought suggested by Godfried Toussaint: suppose the configuration of a chain initially lies in the plane, but that to straighten it without edge crossings, we allow ourselves to lift it into 3D. Eventually, the workshop designed an algorithm that convexifies planar polygons, with intersections forbidden, by lifting one link after another to a subchain that forms a convex arch and that lies in a plane parallel to the original plane, joined to it by a connecting link at each end of the arch. We came to call this the St. Louis arch algorithm, since

the idea of storing the partial solution out of the way of the unprocessed links reminded us of the huge arch that towers over the city of St. Louis, Missouri. For this and other chain straightening and convexification results obtained by this workshop, see [2].

One of the workshop participants, Ileana Streinu, suggested an intriguing approach to the polygon convexification problem. As long as we were willing to use 3D to move an initially flat polygonal linkage to a convex shape, why not simply “flip out” the pockets? A *pocket* of a polygon is a connected component of the convex hull of the polygon with the polygon itself removed. The pockets are thus polygons whose boundaries consist of one convex hull edge $[v_i, v_j]$ together with one of the two chains of edges of the polygon between v_i and v_j . To “flip out” a pocket, one would rotate the pocket about the convex hull edge, and return it to the plane outside the original convex hull. Clearly the entire polygonal linkage would not intersect itself during or after this flipping motion, since the links in the pocket would land outside the convex hull of the remaining links. There are some difficulties to work out. If all the pockets are flipped at the same time, they may intersect one another. Also, the polygon formed by flipping out a pocket is not in general convex, so the process must be continued; it is not even clear it terminates. We eventually found that the process *does* terminate after a finite number of flips, but that this number cannot be bounded by n . Worse yet, we eventually found that Ileana had independently rediscovered a question posed by Erdős in the 1930’s and answered by several people in the mean time.

Some of the workshop participants revisited the problem of convexifying a star-shaped polygon and worked out the details and special cases [11]. The workshop also revisited the knitting needles example that Bill and I had suggested as evidence that chains cannot always be straightened in 3D; a concrete example was made and a simple proof of its nonstraightenability was given (see [2]).

The 1998 workshop served to kindle a lot of interest on reconfiguration problems for chains whose links are not allowed to cross. One of the participants, Joe O’Rourke, together with his student R. Cocan, proved that every polygonal linkage in dimension $D > 3$ can be convexified [6]. For chains whose links are allowed to cross in dimension $D > 2$, Bill Lenhart and I had proved this as a by-product of the our techniques for turning a polygon inside-out in the plane.

A subset of participants, together with Michael Soss, a student of Toussaint, found a convexification procedure for monotone polygons in the plane, whose links are not allowed to cross [3].

Toussaint has written a history of Erdős’ pocket-flipping problem and its solutions and has given a proof combining elements of various ones of these solutions [36]. He has also led his own workshops on various aspects of motion planning for polygonal linkages, which became the topic of Michael Soss’s Ph.D. thesis at McGill.

A fascinating recent development in the area is the following. One of the 1998 workshop participants, Eric Demaine, together with Robert Connelly and Günter Rote has answered the planar chain straightening problem in the affirmative: every *simple* configuration of an open chain (or a closed polygon) in the plane

can be straightened (or convexified) in the plane while keeping the linkage *simple* during the entire motion. See [7]. Their proof uses techniques from rigidity theory and linear programming, combined. They inflate the polygon by moving vertices so that the distance between non-adjacent vertices never decreases. Meanwhile, another one of the 1998 workshop participants, Ileana Streinu, has come up with an elegant, more “concrete” proof method [35], based on the notion of pseudo-triangulations. The flavour is discrete, combinatorial, and mechanical.

Problems (non-crossing planar chain straightening and polygon convexifying): With edges not allowed to cross, are there more simple ways to straighten a chain or convexify a polygon in the plane? Here, one may consider special classes of polygons, as done in [11,3].

Of course, tastes vary about what constitutes a “simple method”. Note that there is a distinction to be made between the running time of an algorithm that computes the description of a motion, and number of “mechanical steps” one makes in physically carrying out a motion, or the length of the description of the motion. Then there is the question of practical implementability, both electronic and mechanical. Another consideration is whether to allow more than a constant number of joints to be active at the same time, where “active” refers to a change in the angle between two adjacent links or a change in the dihedral angle between the two planes determined three consecutive links.

Problems (3D non-crossing chain straightening and polygon convexification): In 3D, with edges not allowed to cross, when can an open or closed chain be straightened or convexified? What is the complexity of this problem? Are there interesting, nontrivial special situations for which a convexifying strategy be given? (One of the papers from the 1998 workshop [2] gives some easy examples.)

Problems (toleranced reconfiguration): What can be said about reconfiguring chains with a clearance constraint? For example, suppose that one must move a chain from one configuration to another while respecting a safety zone of some fixed radius around each link?

Finally, as mentioned earlier, chains and polygons are special cases of tree-like linkages, which themselves are special cases of graph-like linkages. For trees, even in the plane with edges allowed to cross, it is NP-hard to decide if a tree-linkage can be positioned so that its leaves are located at given points in the plane. See [37] and, for algorithms for placing trees see [38,39].

Problems (tree-like linkages): What can be said about the configurations and motions of tree-like (and more generally, graph-like) linkages?

In view of the piece-wise linear knot theoretic flavor of these problems, and their connection with rigidity theory, the subject of linkage reconfiguration offers much fertile ground to be explored.

References

1. T. Biedl, E. Demaine, M. Demaine, S. Lazard, A. Lubiw, J. O'Rourke, S. Robbins, I. Streinu, G. Toussaint and S. Whitesides. On reconfiguring tree linkages: trees can lock. Accepted in *Discrete Applied Math.*, Feb. 2001, to appear; conference abstract in Proc. of the 10th Canadian Conf. on Computational Geometry CCCG '98, McGill University, Montreal, Canada, Aug. 10-12, 1998, pp. 4-5. [8](#)
2. T. Biedl, E. Demaine, M. Demaine, S. Lazard, A. Lubiw, J. O'Rourke, M. Overmars, S. Robbins, I. Streinu, G. Toussaint, and S. Whitesides. Locked and unlocked polygonal chains in 3D. Accepted in *Discrete and Computation Geom.*, May, 2001, to appear; conference abstract in Proc. of the 10th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA), Baltimore MD, USA, Jan. 1999, pp. 866-867. [9](#), [10](#)
3. T. Biedl, E. Demaine, S. Lazard, S. Robbins, and M. Soss. Convexifying monotone polygons. Proc. of the 10th Annual International Symp. on Algorithms and Computation (ISAAC'99), Chennai, India, Dec. 16-18, 1999, Springer-Verlag Lecture Notes in Computer Science, pp. 415-424. [9](#), [10](#)
4. J. Cantarella and H. Johnston. Nontrivial embeddings of polygonal intervals and unknots in 3-space. *J. of Knot Theory and its Ramifications*, vol. 7 (8), pp. 1027-1039, 1998. [2](#)
5. A. Cauchy. Sur les polygones et les polyèdres, seconde mémoire. *Journal Ecole Polytechnique*, vol. 16 (9); pp. 26-38, 1813.
6. R. Cocan and J. O'Rourke. Polygonal chains cannot lock in 4D. Proc. 11th Canadian Conf. on Computational Geometry (CCCG), 1999. [9](#)
7. R. Connelly, E. Demaine, G. Rote. Straightening polygonal arcs and convexifying polygonal cycles. Proc. of the 41st IEEE Symp. on Foundations of Computer Sciences (FOCS), 2000, pp. 432-442. [2](#), [6](#), [10](#)
8. T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. Undergraduate textbook, MIT Press and McGraw Hill, 1990. [3](#)
9. R. G. Downey and M. R. Fellows. *Parameterized complexity*. Springer-Verlag, New York, 1999. [3](#)
10. Paul Erdős. Problem no. 3763. *American Mathematical Monthly*, vol. 42, p. 627, 1935.
11. H. Everett, S. Lazard, S. Robbins, H. Schröder and S. Whitesides. Convexifying star-shaped polygons. Proc. of the 10th Canadian Conf. on Computational Geometry CCCG '98, McGill University, Montreal, Canada, Aug. 10-12, 1998, pp. 2-3. [9](#), [10](#)
12. P. Finn, D. Halperin, L. Kavraki; J-C. Latombe; R. Motwani; C. Shelton, and S. Venkatasubramanian. Geometric manipulation of flexible ligands. *Applied Computational Geometry*, Springer-Verlag, pp. 67-78, 1996. [2](#)
13. Aviezri Frankel. Complexity of protein folding. *Bulletin of Mathematical Biology*, vol. 55 (6), pp. 1199-1210, 1993. [2](#)
14. Maxim Frank-Kamenetskii. Unravelling DNA. Addison-Wesley, 1997. [2](#)
15. J. Hopcroft, D. Joseph, and S. Whitesides. On the movement of robotic arms in 2-dimensional bounded regions. *SIAM J. on Computing* vol. 14, May 1985, pp. 315-333. [2](#), [4](#)
16. J. Hopcroft, D. Joseph, and S. Whitesides. Movement problems for 2-dimensional linkages. *SIAM J. on Computing* vol. 13, Aug. 1984, pp. 610-629.
17. J. Hopcroft, D. Joseph and S. Whitesides. On the movement of robot arms in two-dimensional bounded regions. Proc. of the IEEE 23rd Annual Symp. on the Foundations of Computer Science (FOCS), Chicago IL, USA, Nov. 3-5, 1982, pp. 280-289. [2](#)

18. V. Kantabutra. Motions of a short-linked robot arm in a square. *Discrete Comput. Geom.* vol. 7, 1992, pp. 69-76. 5
19. Vitit Kantabutra. Reaching a point with an unanchored robot arm in a square. *Int. J. of Computational Geometry and Applications* vol. 7 (6), pp. 539-550, 1997. 5
20. V. Kantabutra and R. Kosaraju. New algorithms for multilink robot arms. *J. Comput. System Sci.*, vol. 32, pp. 136-153, 1986. 5
21. Dexter Kozen. The Design and Analysis of Algorithms. Graduate textbook, Springer-Verlag, 1992. 4
22. M. van Kreveld, J. Snoeyink and S. Whitesides. Folding rulers inside triangles. *Discrete and Computational Geometry* vol. 15, 1996, pp. 265-285; conference abstract in Proc. of the 5th Canadian Conf. on Computational Geometry, Queen's U., Kingston, Canada, Aug. 5-10, 1993, pp. 1-6. 8
23. J. Kitcher. Coordinated Motion Planning of Planar Linkages. Ph.D. thesis, John Hopkins U., 1992.
24. Jean-Claude Latombe. Robot Motion Planning. Kluwer Academic Publishers, Boston, 1991. 2
25. W. Lenhart and S. Whitesides. Reconfiguring closed polygonal chains in Euclidean d-space. *Discrete and Computational Geometry*, vol. 13, 1995, pp. 123-140; conference abstracts in Proc. of the 3rd Canadian Conf. on Computational Geometry, Vancouver, Canada, Aug. 6-10, 1991, pp. 66-69 ("Turning a Polygon Inside-out"), and in Proc. of the 4th Canadian Conf. on Computational Geometry, St. John's, Newfoundland, Canada, Aug. 10-14, 1992, pp. 198-203 ("Reconfiguring with Linetracking Motions"); see also Reconfiguring Simple Polygons, technical report, McGill University, School of Computer Science SOCS-93.3, 1993. 6
26. A. Lubiw and J. O'Rourke. When can a polygon fold to a polytope? Technical Report 048, Dept. of Computer Science, Smith College, June 1996.
27. Joseph O'Rourke. Chapter 8.6, Computational Geometry in C. Cambridge University Press, 1998. 4
28. J. O'Rourke. Folding and unfolding in computational geometry. Proc. Japan Conf. Discrete Comput. Geom., Dec. 1998, LNCS vol. 1763, pp. 258-266, 1999. 2
29. Naixun Pei. On the Reconfiguration and Reachability of Chains. Ph.D. thesis, School of Computer Science, McGill U., 1996. 8
30. N. Pei and S. Whitesides. On folding rulers in regular polygons. Proc. of the 9th Canadian Conf. on Computational Geometry CCCG '97, Queen's University, Kingston, Ontario, Canada, Aug. 11-14, 1997, pp. 11-16. 8
31. N. Pei and S. Whitesides. On the reachable regions of chains. Proc. of the 8th Canadian Conf. on Computational Geometry CCCG '96, Carleton University, Ottawa, Ontario, Canada, Aug. 12-15, 1996, pp. 161-166. 8
32. S. Whitesides and N. Pei. On the reconfiguration of chains. Computing and Combinatorics, Proc. of the 2nd Annual International Conf., COCOON '96, Hong Kong, June 17-19, 1996, J-Y Cai and C-K Wong, eds., Springer-Verlag Lecture Notes in Computer Science LNCS vol. 1090, pp. 381-390. 8
33. Micha Sharir. Algorithmic motion planning. J. E. Goodman and J. O'Rourke, eds., Handbook of Discrete and Computational Geometry, chapter 40, pp. 733-754, CRC Press, Boca Raton FL, 1997. 2
34. J. Schwartz and M. Sharir. On the "piano mover's" problem, II. General techniques for computing topological properties of real algebraic manifolds. *Advances in Applied Math.* vol. 4, pp. 298-351, 1983. 2

35. Ileana Streinu. A combinatorial approach to planar non-colliding robot arm motion planning. Proc. of the 41st IEEE Symp. on Foundations of Computer Sciences (FOCS), 2000, pp. 443-453. [2](#), [10](#)
36. Godfried Toussaint. The Erdős-Nagy theorem and its ramifications. Proc. 11th Canadian Conf. on Computational Geometry, Vancouver, Aug. 1999. [9](#)
37. Sue Whitesides. Algorithmic issues in the geometry of planar linkage movement. *Australian Computer Journal*, vol. 24 (2), pp. 42-50, 1992. [2](#), [4](#), [10](#)
38. S. Whitesides and R. Zhao. Algorithmic and complexity results for drawing Euclidean trees. Advanced Visual Interfaces, Proc. of the International Workshop AVI '92, Rome, Italy, May 25-29, 1992, T. Catarci, M. F. Costabile, and S. Levialdi, eds.. World Scientific Series in Computer Science vol. 36, 1992, pp. 395-410. [4](#), [10](#)
39. Rongyao Zhao. Placements of Euclidean Trees. Ph. D. thesis, School of Computer Science, McGill U., 1990. [4](#), [10](#)

Application of M-Convex Submodular Flow Problem to Mathematical Economics

Kazuo Murota and Akihisa Tamura

RIMS, Kyoto University, Kyoto 606-8502, Japan

{murota,tamura}@kurims.kyoto-u.ac.jp

<http://www.kurims.kyoto-u.ac.jp/>

Abstract. This paper shows an application of the M-convex submodular flow problem to an economic model in which producers and consumers trade various indivisible commodities through a perfectly divisible commodity, money. We give an efficient algorithm to decide whether a competitive equilibrium exists or not, when cost functions of the producers are $M^\natural$ -convex and utility functions of the consumers are $M^\natural$ -concave and quasilinear in money. The algorithm consists of two phases: the first phase computes productions and consumptions in an equilibrium by solving an M-convex submodular flow problem and the second finds an equilibrium price vector by solving a shortest path problem.

1 Introduction

“Discrete convex analysis,” recently proposed by Murota [4,5], is a unified framework of discrete optimization with reference to existing studies on submodular functions, generalized polymatroids, valuated matroids and convex analysis. In discrete convex analysis, the concepts of M-/ $M^\natural$ -convex functions play a central role and the M-convex submodular flow problem is introduced as an extension of the minimum cost flow problem and the submodular flow problem. The M-convex submodular flow problem is a general framework which can be solved in polynomial time. The optimality criterion for the M-convex submodular flow problem is equivalent to the Fenchel-type min-max duality theorem.

The present work addresses a computational aspect of competitive equilibria in an economy with indivisible commodities by applying the M-convex submodular flow problem. We deal with an economic model in which producers and consumers trade various indivisible commodities through a perfectly divisible commodity, money. The producers have $M^\natural$ -convex cost functions and the consumers have $M^\natural$ -concave utility functions quasilinear in money.

Our contribution is an efficient algorithm for finding an equilibrium. The algorithm consists of two phases: the first phase computes productions and consumptions in an equilibrium by solving an M-convex submodular flow problem and the second finds an equilibrium price vector by solving a shortest path problem. Both the smallest and the largest equilibrium price vectors can be computed.

2 M-Convexity

We review several definitions and known results on M-/M^h-convex functions.

We denote by $\mathbf{R}$, $\mathbf{R}_+$, $\mathbf{Z}$ and $\mathbf{Z}_+$ the sets of reals, nonnegative reals, integers and nonnegative integers, respectively. Let V be a finite set. We define the positive support and negative support of $z = (z(v) : v \in V) \in \mathbf{Z}^V$ by

$$\text{supp}^+(z) = \{v \in V \mid z(v) > 0\} \quad \text{and} \quad \text{supp}^-(z) = \{v \in V \mid z(v) < 0\}.$$

For $S \subseteq V$, we denote by χ_S the characteristic vector of S defined by $\chi_S(v) = 1$ if $v \in S$; otherwise 0, and write simply χ_u instead of $\chi_{\{u\}}$ for $u \in V$. For $p \in \mathbf{R}^V$ and $f : \mathbf{Z}^V \rightarrow \mathbf{R} \cup \{\pm\infty\}$, we define functions $\langle p, x \rangle$ and $f[p](x)$ by

$$\langle p, x \rangle = \sum_{v \in V} p(v)x(v) \quad \text{and} \quad f[p](x) = f(x) + \langle p, x \rangle \quad (x \in \mathbf{Z}^V),$$

and the sets of minimizers and maximizers of f and the effective domain of f by

$$\begin{aligned} \arg \min f &= \{x \in \mathbf{Z}^V \mid f(x) \leq f(y) \quad (\forall y \in \mathbf{Z}^V)\}, \\ \arg \max f &= \{x \in \mathbf{Z}^V \mid f(x) \geq f(y) \quad (\forall y \in \mathbf{Z}^V)\}, \\ \text{dom } f &= \{x \in \mathbf{Z}^V \mid -\infty < f(x) < +\infty\}. \end{aligned}$$

For each $x \in \text{dom } f$, the set defined by

$$\partial_{\mathbf{R}} f(x) = \{p \in \mathbf{R}^V \mid f(y) - f(x) \geq \langle p, y - x \rangle \quad (\forall y \in \mathbf{Z}^V)\}$$

is the *subdifferential* of f at x , and an element p of $\partial_{\mathbf{R}} f(x)$ is a *subgradient* of f at x . From the above definitions, we see

$$p \in \partial_{\mathbf{R}} f(x) \iff x \in \arg \min f[-p]. \quad (1)$$

We also define $\partial'_{\mathbf{R}}$ (concave version of $\partial_{\mathbf{R}}$) by $\partial'_{\mathbf{R}} f(x) = -\partial_{\mathbf{R}}(-f)(x)$.

A function $f : \mathbf{Z}^V \rightarrow \mathbf{R} \cup \{+\infty\}$ with $\text{dom } f \neq \emptyset$ is called *M-convex* [5] if it satisfies

(M-EXC) for $x, y \in \text{dom } f$ and $u \in \text{supp}^+(x - y)$, there exists $v \in \text{supp}^-(x - y)$ such that

$$f(x) + f(y) \geq f(x - \chi_u + \chi_v) + f(y + \chi_u - \chi_v).$$

We note that (M-EXC) is also represented as: for $x, y \in \text{dom } f$,

$$f(x) + f(y) \geq \max_{u \in \text{supp}^+(x-y)} \min_{v \in \text{supp}^-(x-y)} [f(x - \chi_u + \chi_v) + f(y + \chi_u - \chi_v)],$$

where the maximum and the minimum over an empty set are $-\infty$ and $+\infty$, respectively. From (M-EXC), the effective domain of an M-convex function lies on a hyperplane $\{x \in \mathbf{R}^V \mid x(V) = \text{constant}\}$, where $x(V) = \sum_{v \in V} x(v)$.

The concept of M^h-convexity is a variant of M-convexity. Let 0 denote a new element not in V and define $\hat{V} = \{0\} \cup V$. A function $f : \mathbf{Z}^{\hat{V}} \rightarrow \mathbf{R} \cup \{+\infty\}$

with $\text{dom } f \neq \emptyset$ is called $M^\natural$ -convex [8] if it is expressed in terms of an M-convex function $\hat{f} : \mathbf{Z}^V \rightarrow \mathbf{R} \cup \{+\infty\}$ as

$$f(x) = \hat{f}(x_0, x) \quad \text{with } x_0 = -x(V).$$

Namely, an $M^\natural$ -convex function is a function obtained as the projection of an M-convex function. Conversely, an $M^\natural$ -convex function f determines the corresponding M-convex function $\hat{f}$ by

$$\hat{f}(x_0, x) = \begin{cases} f(x) & \text{if } x_0 = -x(V) \\ +\infty & \text{otherwise} \end{cases}$$

up to a translation of $\text{dom } f$ in the direction of 0. An $M^\natural$ -convex function can also be defined by using an exchange property.

Theorem 1 ([8]). *A function $f : \mathbf{Z}^V \rightarrow \mathbf{R} \cup \{+\infty\}$ with $\text{dom } f \neq \emptyset$ is $M^\natural$ -convex if and only if it satisfies*

($M^\natural$ -EXC) *for $x, y \in \text{dom } f$,*

$$f(x) + f(y) \geq \max_{u \in \text{supp}^+(x-y)} \min_{v \in \text{supp}^-(x-y) \cup \{0\}} [f(x - \chi_u + \chi_v) + f(y + \chi_u - \chi_v)]$$

where we assume χ_0 is the zero vector on V .

The minimizers of an $M^\natural$ -convex function has a nice characterization which can be checked efficiently.

Theorem 2 ([4,5]). *For an $M^\natural$ -convex function f and $x \in \text{dom } f$,*

$$f(x) \leq f(y) \quad (\forall y \in \mathbf{Z}^V) \iff f(x) \leq f(x - \chi_u + \chi_v) \quad (\forall u, v \in \{0\} \cup V).$$

We next describe the M-convex submodular flow problem introduced in [6]. An instance of the problem consists of a directed network $N = (V, A, \gamma, \underline{c}, \bar{c})$ and an M-convex function $f : \mathbf{Z}^V \rightarrow \mathbf{R} \cup \{+\infty\}$, where V is the vertex-set, A is the arc-set, $\gamma : A \rightarrow \mathbf{R}$ is the cost function, $\underline{c} : A \rightarrow \mathbf{Z} \cup \{-\infty\}$ and $\bar{c} : A \rightarrow \mathbf{Z} \cup \{+\infty\}$ are functions defining lower and upper capacities of arcs. For each vertex v , let δ^+v and δ^-v denote the sets of leaving arcs and entering arcs of v , respectively. A flow ξ in N is a function $\xi : A \rightarrow \mathbf{Z}$, where it should be noted that we consider integer-valued flow only. A function $\partial\xi : V \rightarrow \mathbf{Z}$ derived from ξ by

$$\partial\xi(v) = \sum \{\xi(a) \mid a \in \delta^+v\} - \sum \{\xi(a) \mid a \in \delta^-v\} \quad (v \in V)$$

is called the *boundary* of ξ . The *M-convex submodular flow problem (MSFP)* is an optimization problem formulated by

$$\begin{aligned} \text{Minimize} \quad & \Gamma(\xi) = \sum_{a \in A} \gamma(a)\xi(a) + f(\partial\xi) \\ \text{subject to} \quad & \underline{c}(a) \leq \xi(a) \leq \bar{c}(a) \quad (a \in A), \\ & \partial\xi \in \text{dom } f, \\ & \xi \in \mathbf{Z}^A. \end{aligned}$$

An optimal flow can be characterized by a potential $p : V \rightarrow \mathbf{R}$ as below.

Theorem 3 ([6]). Let $N = (V, A, \gamma, \underline{c}, \bar{c})$ and f be an instance of the MSFP. For any feasible flow $\xi : A \rightarrow \mathbf{Z}$, the following conditions are equivalent:

- (OPT) ξ is an optimal flow,
- (POT) there exists a potential $p : V \rightarrow \mathbf{R}$ such that
 - (i) $\gamma(a) + p(\partial^+ a) - p(\partial^- a) > 0 \implies \xi(a) = \underline{c}(a)$,
 - $\gamma(a) + p(\partial^+ a) - p(\partial^- a) < 0 \implies \xi(a) = \bar{c}(a)$,
 - (ii) $\partial \xi \in \arg \min f[-p]$.

Murota [6] and Iwata and Shigeno [3] gave algorithms for the MSFP. Those algorithms find an optimal flow and an optimal potential, and the latter is a polynomial time algorithm in $|V|$ and $\log_2 C$, where C is a certain number satisfying $C \leq \max_{a \in A} |\gamma(a)| + 2 \max_{x \in \text{dom } f} |f(x)|$.

3 The Model

The present work studies an economy with a finite set L of producers, a finite set H of consumers, a finite set K of indivisible commodities and a perfectly divisible commodity, namely money. *Productions* of producers and *consumptions* of consumers are integer-valued vectors in $\mathbf{Z}^K$ representing the numbers of indivisible commodities that they consume and produce. Here producers' inputs are represented by negative numbers and their outputs by positive numbers, and conversely, consumers' inputs are represented by positive numbers and their outputs by negative numbers. In the model, for a given price vector $p = (p(k) : k \in K) \in \mathbf{R}^K$ of commodities, each producer independently schedules a production in order to maximize his/her profit, and each consumer independently schedules a consumption to maximize his/her utility under his/her budget constraint, and all agents exchange commodities by buying or selling those through money.

We assume that producer l 's profit is described by his/her *cost function* $C_l : \mathbf{Z}^K \rightarrow \mathbf{R} \cup \{+\infty\}$ whose value is expressed in units of money. That is, l 's *profit function* $\pi_l : \mathbf{R}^K \rightarrow \mathbf{R}$ is defined by

$$\pi_l(p) = \max_{y \in \mathbf{Z}^K} \{ \langle p, y \rangle - C_l(y) \} \quad (p \in \mathbf{R}^K).$$

Producer l 's *supply function (correspondence)* $S_l : \mathbf{R}^K \rightarrow 2^{\mathbf{Z}^K}$ represents the set of all productions which attain the maximum of l 's profit for a given price vector, that is,

$$S_l(p) = \arg \max_{y \in \mathbf{Z}^K} \{ \langle p, y \rangle - C_l(y) \} \quad (p \in \mathbf{R}^K).$$

Each consumer $h \in H$ has an initial endowment of indivisible commodities and money which is represented by a vector $(x_h^\circ, m_h^\circ) \in \mathbf{Z}_+^K \times \mathbf{R}_+$, where $x_h^\circ(k)$ denotes the number of commodity $k \in K$ and m_h° the amount of money in his/her initial endowment. In the model, each consumer h shares in the profits

of the producers and θ_{lh} denotes the share of the profit of producer l owned by consumer h . The numbers θ_{lh} are nonnegative and $\sum_{h \in H} \theta_{lh} = 1$ for each $l \in L$. Thus, consumer h gains an income expressed by a function $\beta_h : \mathbf{R}^K \rightarrow \mathbf{R}$ defined by

$$\beta_h(p) = \langle p, x_h^\circ \rangle + m_h^\circ + \sum_{l \in L} \theta_{lh} \pi_l(p) \quad (p \in \mathbf{R}^K).$$

We assume that each consumer's utility is quasilinear in money. That is, consumer h 's utility is represented by a quasilinear utility function $\bar{U}_h : \mathbf{Z}^K \times \mathbf{R} \rightarrow \mathbf{R} \cup \{-\infty\}$ defined by

$$\bar{U}_h(x, m) = U_h(x) + m \quad ((x, m) \in \mathbf{Z}^K \times \mathbf{R})$$

where $U_h : \mathbf{Z}^K \rightarrow \mathbf{R} \cup \{-\infty\}$ whose value is expressed in units of money. It is natural to assume that $\text{dom } U_h$ is bounded because none can consume an infinite number of indivisible commodities. We further assume that the amount of money m_h° in h 's initial endowment is sufficiently large for any $h \in H$. Since consumer h 's schedule maximizes $\bar{U}_h$ under the budget constraint, h 's behavior is formulated in terms of an optimization problem

$$\text{Maximize } U_h(x) + m \quad \text{subject to } \langle p, x \rangle + m \leq \beta_h(p).$$

Since $\text{dom } U_h$ is bounded and m_h° is large, we can take $m = \beta_h(p) - \langle p, x \rangle$ to reduce the above problem to an unconstrained optimization problem

$$\text{Maximize } U_h(x) - \langle p, x \rangle.$$

Thus, we can define h 's *demand function (correspondence)* $D_h : \mathbf{R}^K \rightarrow 2^{\mathbf{Z}^K}$ by

$$D_h(p) = \arg \max_{x \in \mathbf{Z}^K} \{U_h(x) - \langle p, x \rangle\} \quad (p \in \mathbf{R}^K).$$

A tuple $((x_h \mid h \in H), (y_l \mid l \in L), p)$, where $x_h \in \mathbf{Z}^K$, $y_l \in \mathbf{Z}^K$ and $p \in \mathbf{R}^K$, is called an *equilibrium* or a *competitive equilibrium* if the following conditions hold:

$$x_h \in D_h(p) \quad (h \in H), \tag{2}$$

$$y_l \in S_l(p) \quad (l \in L), \tag{3}$$

$$\sum_{h \in H} x_h = \sum_{h \in H} x_h^\circ + \sum_{l \in L} y_l, \tag{4}$$

$$p \geq 0. \tag{5}$$

That is, each agent achieves what he/she wishes to achieve, the balance of supply and demand holds and an equilibrium price vector is nonnegative.

Since a utility function is usually assumed to be concave in mathematical economics, a “discrete concave function” is natural in our context. Here we briefly introduce nice features of an $\mathbf{M}^1$ -concave function, which is the negative

of an $M^\sharp$ -convex function, from the point of view of mathematical economics. A utility function generally has decreasing marginal returns, which is equivalent to submodularity in the discrete case. An $M^\sharp$ -concave function U is submodular [10], that is, $U(x) + U(y) \geq U(x \vee y) + U(x \wedge y)$ for $x, y \in \text{dom } U$, where vectors $x \vee y$ and $x \wedge y$ are defined by $(x \vee y)(i) = \max\{x(i), y(i)\}$ and $(x \wedge y)(i) = \min\{x(i), y(i)\}$ for $i \in K$. Moreover, the $M^\sharp$ -concavity is characterized [11] by the gross substitutes condition and the single improvement condition which are fundamental in mathematical economics.

We return to equilibria of economic models with indivisible commodities. A function $U : \mathbf{Z}^K \rightarrow \mathbf{R} \cup \{-\infty\}$ is said to be *monotone nondecreasing* if $x \leq y \Rightarrow U(x) \leq U(y)$ for any $x, y \in \text{dom } U$. Theorems 4 and 5 stated explicitly in [7] are implied by the results in [1].

Theorem 4 ([1,7]). *In an exchange economy, where $L = \emptyset$, if each U_h is monotone nondecreasing and $M^\sharp$ -concave, then there exists an equilibrium $((x_h \mid h \in H), p)$ for any initial total endowment $x^\circ \in \sum_{h \in H} \text{dom } U_h$, where the summation means the Minkowski sum.*

Theorem 5 ([1,7]). *Suppose that each C_l is $M^\sharp$ -convex and that each U_h is $M^\sharp$ -concave in our model. If the continuous model obtained by regarding all indivisible commodities as divisible has an equilibrium for a given initial total endowment, then there exists an equilibrium $((x_h \mid h \in H), (y_l \mid l \in L), p)$ of indivisible commodities, where cost functions and utility functions in the continuous model are the convex extensions of C_l and concave extensions of U_h , respectively.*

The equilibrium price vectors form a well-behaved polyhedron, $L^\sharp$ -convex polyhedron. A polyhedron $P \subseteq \mathbf{R}^K$ is called an $L^\sharp$ -convex polyhedron [2,9] if

$$p, q \in P \implies (p - \alpha \mathbf{1}) \vee q, p \wedge (q + \alpha \mathbf{1}) \in P \quad (0 \leq \forall \alpha \in \mathbf{R}). \quad (6)$$

Theorem 6 ([7]). *Suppose that each C_l is $M^\sharp$ -convex and each U_h is $M^\sharp$ -concave in our model and that there exists an equilibrium for a given initial total endowment x° . Then the set $P^*(x^\circ)$ of all the equilibrium price vectors is an $L^\sharp$ -convex polyhedron. This means in particular ($\alpha = 0$ in (6)) that $p, q \in P^*(x^\circ) \implies p \vee q, p \wedge q \in P^*(x^\circ)$.*

In order to characterize equilibria, we adopt the aggregate cost function $\Psi : \mathbf{Z}^K \rightarrow \mathbf{R} \cup \{\pm\infty\}$ of the market defined by

$$\Psi(z) = \inf \left\{ \sum_{l \in L} C_l(y_l) - \sum_{h \in H} U_h(x_h) \mid \sum_{h \in H} x_h - \sum_{l \in L} y_l = z \right\} \quad (z \in \mathbf{Z}^K). \quad (7)$$

Then we obtain the following characterization of equilibria.

Lemma 1. *Given an initial total endowment $x^\circ = \sum_{h \in H} x_h^\circ$, the following statements hold.*

- (a) *There exists an equilibrium if and only if $(-\partial_{\mathbf{R}} \Psi(x^\circ)) \cap \mathbf{R}_+^K \neq \emptyset$.*

- (b) A price vector $p \in \mathbf{R}^K$ is an equilibrium price vector if and only if $p \in (-\partial_{\mathbf{R}}\Psi(x^\circ)) \cap \mathbf{R}_+^K$.
- (c) If $((x_h \mid h \in H), (y_l \mid l \in L))$ satisfies (2), (3) and (4) for some p (not necessarily nonnegative), then

$$-\partial_{\mathbf{R}}\Psi(x^\circ) = \left(\bigcap_{h \in H} \partial'_{\mathbf{R}} U_h(x_h) \right) \cap \left(\bigcap_{l \in L} \partial_{\mathbf{R}} C_l(y_l) \right),$$

and if, in addition, $(-\partial_{\mathbf{R}}\Psi(x^\circ)) \cap \mathbf{R}_+^K \neq \emptyset$, then $((x_h \mid h \in H), (y_l \mid l \in L), p')$ is an equilibrium for any $p' \in (-\partial_{\mathbf{R}}\Psi(x^\circ)) \cap \mathbf{R}_+^K$.

4 Computation of Equilibria

This section is the main part of the paper. We show how to calculate an equilibrium in the economy in which each C_l is $M^\natural$ -convex and each U_h is $M^\natural$ -concave. We first formulate the problem of finding an equilibrium as the MSFP. Solution of the problem yields consumptions and productions satisfying (2), (3), (4) as well as a price vector which, however, may not be nonnegative. This is the first phase of our algorithm. The second phase finds an equilibrium price vector by solving the shortest path problem. Our algorithm finds an equilibrium if one exists; otherwise either the MSFP or the shortest path problem has no solution. We can also modify the second phase to find the smallest or the largest equilibrium price vector.

For any vector $z \in \mathbf{R}^K$, we denote by $\hat{z}$ the vector $(-z(K), z) \in \mathbf{R}^{\{0\} \cup K}$ whose 0-th component is the negative of the sum of the others. For $M^\natural$ -convex cost function $C_l : \mathbf{Z}^K \rightarrow \mathbf{R} \cup \{+\infty\}$, we define the corresponding M -convex function $\hat{C}_l : \mathbf{Z}^{\{0\} \cup K} \rightarrow \mathbf{R} \cup \{+\infty\}$ by

$$\hat{C}_l(y_0, y) = \begin{cases} C_l(y) & \text{if } y_0 = -y(K) \\ +\infty & \text{otherwise} \end{cases} \quad (l \in L).$$

We also define the $M^\natural$ -concave function $\hat{U}_h : \mathbf{Z}^{\{0\} \cup K} \rightarrow \mathbf{R} \cup \{-\infty\}$ associated with $U_h : \mathbf{Z}^K \rightarrow \mathbf{R} \cup \{-\infty\}$ by

$$\hat{U}_h(x_0, x) = \begin{cases} U_h(x) & \text{if } x_0 = -x(K) \\ -\infty & \text{otherwise} \end{cases} \quad (h \in H).$$

In the same way as (7), we consider the aggregate cost function

$$\hat{\Psi}(\tilde{z}) = \inf \left\{ \sum_{l \in L} \hat{C}_l(\tilde{y}_l) - \sum_{h \in H} \hat{U}_h(\tilde{x}_h) \mid \sum_{h \in H} \tilde{x}_h - \sum_{l \in L} \tilde{y}_l = \tilde{z} \right\} \quad (\tilde{z} \in \mathbf{Z}^{\{0\} \cup K}).$$

The function $\hat{\Psi}$ is the integer infimal convolution of M -convex functions $\hat{C}_l$ ($l \in L$) and $-\hat{U}_h$ ($h \in H$). It is known that the integer infimal convolution of M -convex functions is also M -convex and can be evaluated by solving an instance of the MSFP [4]. We will demonstrate how $\hat{\Psi}(\hat{x}^\circ)$ is evaluated.

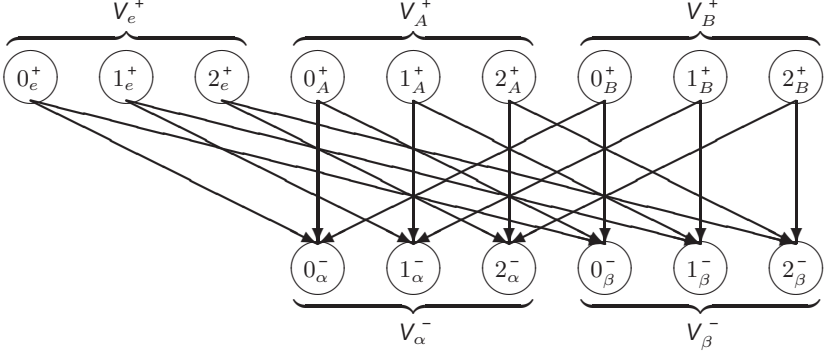


Fig. 1. Graph for the evaluation of M-convex function $\hat{\Psi}(\hat{x}^\circ)$

The instance of the MSFP for the evaluation of $\hat{\Psi}(\hat{x}^\circ)$ is defined as follows. Consider a directed bipartite graph $G = (V^+, V^-, A)$ whose vertex-partition V^+, V^- and arc-set A are defined by

$$V^+ = V_e^+ \cup \left(\bigcup_{l \in L} V_l^+ \right), \quad V^- = \bigcup_{h \in H} V_h^-, \quad V_e^+ = \{k_e^+ \mid k \in \{0\} \cup K\},$$

$$V_l^+ = \{k_l^+ \mid k \in \{0\} \cup K\} \quad (l \in L), \quad V_h^- = \{k_h^- \mid k \in \{0\} \cup K\} \quad (h \in H),$$

$$A = \{(k_l^+, k_h^-) \mid l \in L, h \in H, k \in \{0\} \cup K\} \cup \{(k_e^+, k_h^-) \mid h \in H, k \in \{0\} \cup K\}.$$

Note that V_e^+ , V_l^+ and V_h^- are copies of $\{0\} \cup K$, respectively. In Figure 1, we draw the graph G for the case where $H = \{\alpha, \beta\}$, $L = \{A, B\}$ and $K = \{1, 2\}$. For each arc $a \in A$, we put $\underline{c}(a) = -\infty$, $\bar{c}(a) = +\infty$ and $\gamma(a) = 0$. By using the indicator function $\delta_e : \mathbf{Z}^{V_e^+} \rightarrow \mathbf{R} \cup \{+\infty\}$ of $\{\hat{x}^\circ\}$ defined by

$$\delta_e(\tilde{w}) = \begin{cases} 0 & \text{if } \tilde{w} = \hat{x}^\circ \\ +\infty & \text{otherwise,} \end{cases}$$

we define a function $f : \mathbf{Z}^{V^+ \cup V^-} \rightarrow \mathbf{R} \cup \{+\infty\}$ by

$$f(\tilde{w}, (\tilde{y}_l \mid l \in L), (\tilde{x}_h \mid h \in H)) = \delta_e(\tilde{w}) + \sum_{l \in L} \hat{C}_l(\tilde{y}_l) - \sum_{h \in H} \hat{U}_h(-\tilde{x}_h),$$

where $\tilde{w} \in \mathbf{Z}^{V_e^+}$, $\tilde{y}_l \in \mathbf{Z}^{V_l^+}$ ($l \in L$) and $\tilde{x}_h \in \mathbf{Z}^{V_h^-}$ ($h \in H$). The function f is M-convex, since $\hat{C}_l$ ($l \in L$) are M-convex and $\hat{U}_h$ ($h \in H$) are M-concave. This is the instance of the MSFP that we use for the computation of $\hat{\Psi}(\hat{x}^\circ)$.

An optimal flow and an optimal potential of the MSFP have the following nice properties on which the evaluation of $\hat{\Psi}(\hat{x}^\circ)$ relies.

Lemma 2. Assume that the above instance of MSFP has an optimal flow $\xi \in \mathbf{Z}^A$ and an optimal potential $\bar{p} \in \mathbf{R}^{V^+ \cup V^-}$. Let

$$\tilde{x}_h^* = -\partial\xi|_{V_h^-} \quad (h \in H), \quad \tilde{y}_l^* = \partial\xi|_{V_l^+} \quad (l \in L), \quad \tilde{w}^* = \partial\xi|_{V_e^+}, \quad \tilde{p} = \bar{p}|_{V_e^+},$$

where $\tilde{x}_h^* = -\partial\xi|_{V_h^-}$ means the restriction of $-\partial\xi$ to V_h^- , that is, $\tilde{x}_h^*(k) = -\partial\xi(k)$ ($k \in V_h^-$), etc., and we regard $\tilde{x}_h^*$, $\tilde{y}_l^*$, $\tilde{w}^*$ and $\tilde{p}$ as vectors on $\{0\} \cup K$. Then the following hold.

- (a) $\tilde{w}^* = \hat{x}^\circ$, $\tilde{x}_h^*(0) = -\tilde{x}_h^*(K)$ ($h \in H$) and $\tilde{y}_l^*(0) = -\tilde{y}_l^*(K)$ ($l \in L$).
- (b) $\hat{x}^\circ + \sum_{l \in L} \tilde{y}_l^* = \sum_{h \in H} \tilde{x}_h^*$.
- (c) $\bar{p}(k_h^-) = \bar{p}(k_l^+) = \bar{p}(k_e^+)$ for any $k \in \{0\} \cup K$, $h \in H$ and $l \in L$.
- (d) $\hat{x}^\circ \in \arg \min \hat{\Psi}[\bar{p}]$.
- (e) $\tilde{x}_h^* \in \arg \max \hat{U}_h[-\bar{p}]$ ($h \in H$) and $\tilde{y}_l^* \in \arg \min \hat{C}_l[-\bar{p}]$ ($l \in L$).

Proof. (a): This follows directly from the definitions of $\hat{C}_l$, $\hat{U}_h$ and δ_e .

(b): From the definition of G , for each $k \in \{0\} \cup K$, we have

$$\begin{aligned} \tilde{w}^*(k) + \sum_{l \in L} \tilde{y}_l^*(k) &= \sum_{h \in H} \xi(k_e^+, k_h^-) + \sum_{l \in L} \sum_{h \in H} \xi(k_l^+, k_h^-) \\ &= \sum_{h \in H} \left[\xi(k_e^+, k_h^-) + \sum_{l \in L} \xi(k_l^+, k_h^-) \right] = \sum_{h \in H} \tilde{x}_h^*(k). \end{aligned}$$

Then the assertion follows from $\tilde{w}^* = \hat{x}^\circ$.

(c): Since $\gamma(a) = 0$, $\underline{c}(a) = -\infty$ and $\bar{c}(a) = +\infty$ for any $a \in A$, condition (i) of (POT) of Theorem 3 implies

$$\bar{p}(\partial^+ a) - \bar{p}(\partial^- a) = 0 \quad (a \in A).$$

This and the structure of G show the assertion.

(e): Assertion (c) implies

$$\begin{aligned} \min f[-\bar{p}] &= \min \left\{ \delta_e[-\bar{p}](\tilde{w}) + \sum_{l \in L} \hat{C}_l[-\bar{p}](\tilde{y}_l) - \sum_{h \in H} \hat{U}_h[-\bar{p}](-\tilde{x}_h) \right\} \\ &= \sum_{l \in L} \min_{\tilde{y}} \hat{C}_l[-\bar{p}](\tilde{y}) - \sum_{h \in H} \max_{\tilde{x}} \hat{U}_h[-\bar{p}](\tilde{x}) - \langle \tilde{p}, \hat{x}^\circ \rangle \end{aligned} \quad (8)$$

$$= \min \hat{\Psi}[\bar{p}] - \langle \tilde{p}, \hat{x}^\circ \rangle. \quad (9)$$

We also have

$$f[-\bar{p}](\partial\xi) = \sum_{l \in L} \hat{C}_l[-\bar{p}](\tilde{y}_l^*) - \sum_{h \in H} \hat{U}_h[-\bar{p}](\tilde{x}_h^*) - \langle \tilde{p}, \hat{x}^\circ \rangle. \quad (10)$$

Condition (ii) of (POT) of Theorem 3, (8) and (10) yield (e).

(d): Condition (ii) of (POT) of Theorem 3 and (9) say

$$f[-\bar{p}](\partial\xi) = \min \hat{\Psi}[\bar{p}] - \langle \tilde{p}, \hat{x}^\circ \rangle.$$

On the other hand, assertion (b) guarantees

$$f(\partial\xi) = \min \left\{ \sum_{l \in L} \hat{C}_l(\tilde{y}_l) - \sum_{h \in H} \hat{U}_h(\tilde{x}_h) \mid \sum_{h \in H} \tilde{x}_h - \sum_{l \in L} \tilde{y}_l = \hat{x}^\circ \right\} = \hat{\Psi}(\hat{x}^\circ).$$

The above equalities show

$$\hat{\Psi}[\tilde{p}](\hat{x}^\circ) = f(\partial\xi) + \langle \tilde{p}, \hat{x}^\circ \rangle = f[-\tilde{p}](\partial\xi) + \langle \tilde{p}, \partial\xi \rangle + \langle \tilde{p}, \hat{x}^\circ \rangle = \min \hat{\Psi}[\tilde{p}],$$

where $\langle \tilde{p}, \partial\xi \rangle = 0$ by (b) and (c).

The following theorem, which is a consequence of Lemma 2, guarantees that consumptions and productions satisfying (2), (3), (4) can be computed by solving the above instance of the MSFP, and that if the optimal potential is nonnegative then it serves as an equilibrium price vector.

Theorem 7. *Assume that the above instance of MSFP has an optimal flow $\xi \in \mathbf{Z}^A$ and an optimal potential $\bar{p} \in \mathbf{R}^{V^+ \cup V^-}$ with $\bar{p}(0_e^+) = 0$. Let*

$$x_h^* = -\partial\xi|_{V_h^- \setminus \{0_h^-\}} \quad (h \in H), \quad y_l^* = \partial\xi|_{V_l^+ \setminus \{0_l^+\}} \quad (l \in L), \quad p = \bar{p}|_{V_e^+ \setminus \{0_e^+\}},$$

where we regard x_h^* , y_l^* and p as vectors on K . Then the following statements hold.

- (a) $x^\circ \in \arg \min \Psi[p]$.
- (b) $x^\circ + \sum_{l \in L} y_l^* = \sum_{h \in H} x_h^*$.
- (c) $x_h^* \in \arg \max U_h[-p] \quad (h \in H)$.
- (d) $y_l^* \in \arg \min C_l[-p] \quad (l \in L)$.

Therefore, $((x_h^* \mid h \in H), (y_l^* \mid l \in L), p)$ is an equilibrium if $p \geq 0$. Moreover, if there exists an equilibrium at all, then $((x_h^* \mid h \in H), (y_l^* \mid l \in L))$ is the consumptions and productions of some equilibrium.

Any algorithm for the MSFP will find a tuple $((x_h^* \mid h \in H), (y_l^* \mid l \in L), p)$, which gives an equilibrium for the initial total endowment x° if the optimal potential p happens to be nonnegative. We go on to show that the set of all nonnegative optimal potentials, or equivalently, the set of all equilibrium price vectors, can be expressed by a certain linear inequality system. Thus, the existence of an equilibrium price vector can be checked by solving a linear programming problem which is reduced to the dual of a single-source shortest path problem.

In order to give a linear inequality description of the set of equilibrium price vectors, we give a necessary and sufficient condition for (a) of Theorem 7. Lemma 1 and (1) yield the equivalence that

$$x^\circ \in \arg \min \Psi[q] \Leftrightarrow \begin{cases} x_h^* \in \arg \max U_h[-q] & (h \in H) \\ y_l^* \in \arg \min C_l[-q] & (l \in L). \end{cases} \quad (11)$$

By Theorem 2, we have $y_l^* \in \arg \min C_l[-q]$ if and only if

$$\begin{cases} C_l(y_l^*) - C_l(y_l^* - \chi_j) \leq q(j) \leq C_l(y_l^* + \chi_j) - C_l(y_l^*) & (j \in K) \\ q(j) - q(i) \leq C_l(y_l^* - \chi_i + \chi_j) - C_l(y_l^*) & (i, j \in K, i \neq j) \end{cases} \quad (12)$$

and $x_h^* \in \arg \max U_h[-q]$ if and only if

$$\begin{cases} U_h(x_h^* + \chi_j) - U_h(x_h^*) \leq q(j) \leq U_h(x_h^*) - U_h(x_h^* - \chi_j) & (j \in K) \\ q(j) - q(i) \leq U_h(x_h^*) - U_h(x_h^* + \chi_i - \chi_j) & (i, j \in K, i \neq j). \end{cases} \quad (13)$$

By (11), (12) and (13), we obtain

$$x^\circ \in \arg \min \Psi[q] \Leftrightarrow \begin{cases} l(j) \leq q(j) \leq u(j) & (j \in K) \\ q(j) - q(i) \leq u(i, j) & (i, j \in K, i \neq j), \end{cases} \quad (14)$$

where

$$l(j) = \max \left\{ \max_{h \in H} \{U_h(x_h^* + \chi_j) - U_h(x_h^*)\}, \max_{l \in L} \{C_l(y_l^*) - C_l(y_l^* - \chi_j)\} \right\}, \quad (15)$$

$$u(j) = \min \left\{ \min_{h \in H} \{U_h(x_h^*) - U_h(x_h^* - \chi_j)\}, \min_{l \in L} \{C_l(y_l^* + \chi_j) - C_l(y_l^*)\} \right\}, \quad (16)$$

$$u(i, j) = \min \left\{ \min_{h \in H} \{U_h(x_h^*) - U_h(x_h^* + \chi_i - \chi_j)\}, \min_{l \in L} \{C_l(y_l^* - \chi_i + \chi_j) - C_l(y_l^*)\} \right\}. \quad (17)$$

We note that $l(j) < +\infty$, $u(j) > -\infty$ and $u(i, j) > -\infty$ hold for any $i, j \in K$ ($i \neq j$).

We recall that $P^*(x^\circ)$ denotes the set of all equilibrium price vectors for the initial total endowment x° . We have $P^*(x^\circ) = (-\partial_{\mathbf{R}} \Psi(x^\circ)) \cap \mathbf{R}_+^K$ by (b) of Lemma 1. By the above argument, $P^*(x^\circ)$ can be described as follows by using $((x_h^* \mid h \in H), (y_l^* \mid l \in L))$.

Theorem 8. *The set $P^*(x^\circ)$ of all equilibrium price vectors is a polyhedron described by*

$$P^*(x^\circ) = \left\{ q \in \mathbf{R}^K \mid \begin{array}{l} \max\{0, l(j)\} \leq q(j) \leq u(j) \quad (j \in K) \\ q(j) - q(i) \leq u(i, j) \quad (i, j \in K, i \neq j) \end{array} \right\}, \quad (18)$$

where $l(j)$, $u(j)$ and $u(i, j)$ are defined in (15), (16) and (17) by $((x_h^* \mid h \in H), (y_l^* \mid l \in L))$.

Theorem 8 guarantees that nonemptiness of $P^*(x^\circ)$ can be checked by linear programming. In particular, the largest equilibrium price vector, if any, can be found by solving a linear programming problem:

$$\begin{array}{ll} \text{Maximize} & \sum_{k \in K} q(k) \\ \text{subject to} & \max\{0, l(j)\} \leq q(j) \leq u(j) \quad (j \in K) \\ & q(j) - q(i) \leq u(i, j) \quad (i, j \in K, i \neq j), \end{array} \quad (19)$$

because the largest vector in $P^*(x^\circ)$ maximizes the sum of all components of a vector. Analogously, the smallest equilibrium price vector can be found by solving another linear programming problem:

$$\begin{array}{ll} \text{Minimize} & \sum_{k \in K} q(k) \\ \text{subject to} & \max\{0, l(j)\} \leq q(j) \leq u(j) \quad (j \in K) \\ & q(j) - q(i) \leq u(i, j) \quad (i, j \in K, i \neq j). \end{array} \quad (20)$$

Both (19) and (20) can be easily reduced to the dual of a single-source shortest path problem.

Theorem 9. *There exists an equilibrium price vector if and only if problem (19) (as well as (20)) is feasible. In particular, both the smallest and the largest equilibrium price vectors, if any, can be found by solving the shortest path problem.*

Theorems 7 and 9 are summarized in the following algorithm and theorem.

algorithm CALCULATE-EQUILIBRIUM($C_l(l \in L), U_h(h \in H), x^\circ$)

input: M^h -convex cost functions C_l of producers $l \in L$;
 M^h -concave utility functions U_h of consumers $h \in H$;
initial total endowment x° ;

Step0: construct the instance of the MSFP ;

Step1: solve the MSFP $[(x_h^* \mid h \in H), (y_l^* \mid l \in L), p]$ is computed] ;
if the instance is infeasible **then** stop [there is no equilibrium] ;

Step2: solve the problem (19) [p^* is computed] ;
if (19) is infeasible **then** there is no equilibrium ;
else $((x_h^* \mid h \in H), (y_l^* \mid l \in L), p^*)$ is an equilibrium with largest p^* .

Theorem 10. *The existence of a competitive equilibrium in our economic model can be checked in polynomial time by CALCULATE-EQUILIBRIUM. Furthermore, the smallest equilibrium price vector can be computed by modifying Step 2.*

References

1. V. Danilov, G. Koshevoy, and K. Murota, Discrete convexity and equilibria in economies with indivisible goods and money, *Math. Social Sci.* **41** (2001) 251–273. [19](#)
2. S. Fujishige and K. Murota, Notes on L-/M-convex functions and the separation theorems, *Math. Programming* **88** (2000) 129–146. [19](#)
3. S. Iwata and M. Shigeno, Conjugate scaling algorithm for Fenchel-type duality in discrete convex optimization, *SIAM J. Optim.*, to appear. [17](#)
4. K. Murota, Convexity and Steinitz’s exchange property, *Adv. Math.* **124** (1996) 272–311. [14](#), [16](#), [20](#)
5. K. Murota, Discrete convex analysis, *Math. Programming* **83** (1998) 313–371. [14](#), [15](#), [16](#)
6. K. Murota, Submodular flow problem with a nonseparable cost function, *Combinatorica* **19** (1999) 87–109. [16](#), [17](#)
7. K. Murota, *Discrete Convex Analysis* (in Japanese) (Kyoritsu-Shuppan, Tokyo) to appear. [19](#)
8. K. Murota and A. Shioura, M-convex function on generalized polymatroid, *Math. Oper. Res.* **24** (1999) 95–105. [16](#)
9. K. Murota and A. Shioura, Extension of M-convexity and L-convexity to polyhedral convex functions, *Adv. in Appl. Math.* **25** (2000) 352–427. [19](#)
10. K. Murota and A. Shioura, Relationship of M-/L-convex functions with discrete convex functions by Miller and by Favati–Tardella, *Discrete Appl. Math.* to appear. [19](#)
11. K. Murota and A. Tamura, New characterizations of M-convex functions and their applications to economic equilibrium models with indivisibilities, *Discrete Appl. Math.*, to appear. [19](#)

A Polynomial Time Approximation Scheme for Minimizing Total Completion Time of Unbounded Batch Scheduling^{*}

Xiaotie Deng¹, Haodi Feng¹, Pixing Zhang², and Hong Zhu²

¹ Department of Computer Science, City University of Hong Kong, Hong Kong
csdeng@cityu.edu.hk
fenghd@cs.cityu.edu.hk

² Department of Computer Science, Fudan University, China

Abstract. We study the unbounded batch machine scheduling of n jobs to minimize the total completion time. A batch machine can handle up to $B \geq n$ jobs simultaneously. Each job is characterized by a release(arrival) time and a processing time. Jobs processed in the same batch have the same completion time.(i.e., their common starting time plus the processing time of the longest job in the batch.) For batch processing, non-preemptive scheduling is usually required and we focus on this case. In this paper, we establish a polynomial time approximation scheme for it.

1 Introduction

We study the problem of jobs scheduling in a batch processing system without preemption. More precisely, we are given a set of jobs $\mathcal{J} = \{J_1, \dots, J_n\}$ and a *batch machine*. Each job, J_i , is associated with a release(arrival) time r_i which specifies when the job becomes available, a processing time p_i which specifies the minimum time needed to process the job by the machine. The batch machine can process up to $B \geq n$ jobs simultaneously, and we call such batch machine unbounded batch processing machine. Jobs processed in the same batch have the same start time and the same completion time(we denote by C_i the completion time of job J_i). This type of batching processing machine is motivated by the *burn-in* model for the problem of scheduling burn-in operations in the manufacturing of VLSI circuits, see, e.g., Lee et al. [7]. For this model, the processing time of a batch is the largest processing time of any job in the batch and when a batch is being processed, no preemption is permitted. Our goal is to find a schedule for the jobs so that the total completion time (equivalently, average completion time), $\sum_{i=1}^n C_i$, is minimized. following notations of Graham et al. [4], we may denote this problem as $1|r_j, B = +\infty|\sum_{i=1}^n C_i$.

^{*} This research was supported in part by an RGC CERG grant (Project #: 9040597) of the Research Grants Council of Hong Kong Special Administrative Region, China, and an SRG grant of City University of Hong Kong (Project #: 7001040).

Brucker et al. [2] made a thorough discussion of the scheduling problem on the batch machine with various constraints and objective functions. For the objective of minimizing the total completion time, the discussion was restricted for the case that the job arrival times all are zero. They designed a polynomial time algorithm for unbounded batch processing via dynamic programming approach. Deng and Zhang[3] proved that it is NP-hard to get the optimal schedule for the weighted case for the unbounded batch processing machine. That is, $1|r_j, B = +\infty|\sum_{i=1}^n w_i C_i$ is NP-hard. They also gave polynomial time algorithms for some special case.

For the bounded case $B < n$, the best known result is a 2-approximation algorithm for the case when all jobs arrive at time zero ($1|B < n|\sum_{i=1}^n C_i$) by Hochbaum and Landy[5].

For decades, effort has been made in search of PTASs for many classical scheduling problems (such as $1|r_j|\sum C_j$, $P|r_j|\sum w_j C_j$ etc.). The breakthrough only comes recently in the seminal work of Afrati et al.[1]. The major ideas are a novel combination of time stretching, geometric rounding, and dynamic programming techniques. The outcome is amazingly powerful. PTASs are obtained for several classical scheduling problems via their approach. Our work is another successful application of their general ideas.

Even though the general framework in our study follows that of Afrati, et al., [1], special properties of batch machine, especially with unbounded batch processing power, make it quite nontrivial in the detailed analysis. For example, the summation of jobs processed in each interval is no longer bounded by any constant for our case. In addition, dealing with tiny jobs becomes very different from their methods. Scheduling jobs within a block cannot trivially follow their method and demands special treatment.

In Section 2, we will outline the framework of the general approach and reduce the problem to one of scheduling jobs with a constant number of arrivals under a deadline. In Section 3, we focus on the special case of scheduling a set of jobs with constant number of distinct release times within a relaxed deadline. We end the paper with conclusion and discussion in Section 4.

2 Outline of the Framework

To obtain a PTAS, for any given $\epsilon > 0$, we are to find a $1 + \epsilon$ -optimal solution by our algorithm. To simplify the description and proofs, we will use $\delta > 0$ in the following discussion. Its value will be determined by the desired ϵ .

Lemma 1,2,3,4 and the following dynamic programming equation form the main framework of the general approach of Afrati, et al., [1]. For completeness, we present the lemmas here with a short proof for each. To implement the dynamic programming equation, our problem shows quite different properties from the classical schedule problems considered in [1]. We will explain the details in the following context.

Along the approach of Afrati et.al.[1], we round the release times of the input in the first step.

Lemma 1 [1] *With $1 + \delta$ loss, we can enforce $r_j \geq \delta p_j$ for all job j .*

Proof. In some optimal schedule, multiply every job's(batch's) completion time by $1 + \delta$ and increase start times to match. It follows that we can increase release dates to enforce $r_j \geq \delta p_j$, and at the same time, to obtain a $(1 + \delta)$ -optimal schedule. $\square$

The second idea is geometric rounding.

Lemma 2 [1] *With $1 + \delta$ loss, we can assume that all release times are integer powers of $1 + \delta$.*

Proof. First multiply every release time by $1 + \delta$; Then *decrease* each release time to the next *lower* integer power of $1 + \delta$. For an optimal schedule, we put off some batches to satisfy this property. Similar to the proof of Lemma 1, the loss is no more than $1 + \delta$. $\square$

For an arbitrary integer x , we define $R_x = (1 + \delta)^x$. After the above two preprocessing steps, we can assume all release times are in the form of R_x for some integer x . We partition the time interval $(0, \infty)$ into disjoint intervals of the form $I_x = [R_x, R_{x+1})$. For convenience, we will use I_x to refer to both the interval and the size $(R_{x+1} - R_x)$ of the interval. We often use the fact that $I_x = \delta R_x$. By rounding jobs' release times, we can restrict the number of time intervals that any scheduled batch could cross.

Lemma 3 [1] *Each scheduled batch crosses at most $s = \log_{1+\delta}(1 + \frac{1}{\delta})$ intervals.*

Proof. Suppose a batch contains job j as the longest job. Its processing starts within interval $I_x = [R_x, R_{x+1})$. Since, $R_x \geq r_j \geq \delta p_j$ (Lemma 2), we have $I_x = \delta R_x \geq \delta^2 p_j$. Assume that job j crosses s different intervals, then we have $\sum_{i=x}^{x+s-1} I_i \geq p_j$. Following computing, we have $s \geq \lceil \log_{1+\delta}(1 + \frac{1}{\delta}) \rceil$. $\square$

Now we are ready to describe our algorithm. In the algorithm, we divide the time horizon by blocks. Each block includes $s = \lceil \log_{1+\delta}(1 + \frac{1}{\delta}) \rceil$ consecutive time intervals. Let's denote the blocks in the order of time horizon: $B_1 < B_2 < B_3 < \dots$. During the algorithm, we handle the jobs one block by another. It is possible that there is interaction between two consecutive blocks since some batch from an earlier block can cross into the current block. However by the choice of the block size and Lemma 3, no batch crosses an entire block. For the batches that across blocks, we deal with them by restricting the possible finishing times.

Lemma 4 [1] *If a batch j starts at the previous block and finishes in the interval $I_{x(j)}$ of the current block, then we can enforce the batch finish at $R_{x(j)} + i \cdot \delta I_{x(j)}$ with $1 + \delta + \delta^2$ loss.*

Proof. Assume that $C_j \in (R_x(j) + i\delta I_x(j), R_x(j) + (i+1)\delta I_x(j)]$. The rounded completion time is $C'_j = R_x(j) + (i+1)\delta I_x(j)$ with a cost of at most $\delta I_x(j) = \delta(1+\delta)I_{x-1}$, that is, $\delta(1+\delta)$ times the length of the last interval in a block. Therefore, after processing all this kind of batches, the loss is at most $\delta(1+\delta)$. $\square$

For the rounded finish time, we call them the frontiers. From above analysis, we only consider $\lceil \frac{s}{\delta} \rceil$ different possible finishing times of crossing batch for each block. In the algorithm, we use F_i to denote all such possible locations for block B_i .

Now we can process the jobs according to different blocks by dynamic programming. The dynamic programming equation is computed from forth to back.

The Dynamic Programming table entry $O(i, F, U)$ stores the minimum total completion time achievable by starting the set U of jobs before the end of block B_i and the last batch which crosses block B_i finishes at F of block B_{i+1} . Of course there are $\lceil \frac{s}{\delta} \rceil$ different possible choices for F . Given the table entries for some i , the values for $i+1$ can be computed as follows. Let $W(i, F_1, F_2, S)$ be the minimum completion time achievable by scheduling the set of jobs S in block B_i . For simplicity, we let it denote the value of the schedule that the first release time(F_1) is zero and all jobs in S must be scheduled and finished between F_1 and F_2 , where F_1 is the incoming frontier from block B_{i-1} and F_2 the outgoing frontier to block B_{i+1} . We obtain the following[1]

$$O(i+1, F, U) = \min_{F' \in F_i, V \subseteq U} \{O(i, F', V) + F'|U \setminus V| + W(i+1, F', F, U \setminus V)\} \quad (1)$$

where $F \in F_{i+1}$.

For this equation we find that it is possible that there could be exponentially many U for a given i , and for a given U , there could be exponentially many V (subset of U). Fortunately, by the following four lemmas we can reduce the number of U and V to polynomial.

Definition 1 For some time t , if a job's(batch's) length $p_i \leq \delta^2 t$, then we call it tiny with respect to time t .

For the tiny jobs we have the following two lemmas.

Lemma 5 With $1+\delta$ loss, we can start a batch for all the unscheduled released tiny jobs at the beginning of processing one new block.

Proof. In some optimal schedule of above rounded problem, let us assume that a set of released jobs T are tiny regard to the start time t of the current block and they still are not processed before t . Their completion time C_{opt}^T must be no smaller than $t|T|$. If we start a tiny batch at t for them and put off the other batches, then the completion time for the batch T is at most $(1+\delta^2)t|T| \leq (1+\delta^2)C_{opt}^T$. Since the starting of such a batch can put off the later batches, (say B'_x s) by at most $\delta^2 t \leq \delta^2 R_x = \delta I_x = \delta(1+\delta)I_{x-1}$, (where $t \in I_x$) that is $\delta(1+\delta)$ times the length of one interval in one block before B'_x s.

Let I denote the sum of all these intervals before B'_x s. (notice that in one block there is at most one of such intervals) After processing all the tiny jobs, B'_x s can be delayed by at most $(\delta(1+\delta))I$, which is no more than $\delta(1+\delta)$ times the completion times of these B'_x s. , therefore, the total loss is at most $1+\delta+\delta^2$. $\square$

This lemma tells us that once some jobs are tiny with regard to some time t , we can start a batch for them at the start time of following block. By the following lemma we find that each job becomes tiny after waiting constant number of intervals.

Lemma 6 *A job becomes tiny after waiting at most $3\lceil\log_{1+\delta}(\frac{1}{\delta})\rceil$ intervals, that is no more than 3 blocks.*

Proof. By Lemma 1 we know that for any job j , $p_j \leq \frac{r_j}{\delta}$. (Let $r_j = R_x$.) If $\delta^2 R_{x+k} > \frac{R_x}{\delta}$, then job j must be tiny with regard to R_{x+k} . By computing, $k \geq 3\lceil\log_{1+\delta} \frac{1}{\delta}\rceil$, and since $\log_{1+\delta} \frac{1}{\delta} \leq \log_{1+\delta}(1+\frac{1}{\delta})$, thus completes the proof. $\square$

We know that a job wait at most 3 blocks before they can be processed without any delay. Therefore any set U or V must include all the jobs released 3 blocks before their corresponding blocks. Subsequently, for each fixed block i , (or U) the number of U (or V) depends on the set of jobs released within 3 blocks. Furthermore, according to the following lemma, this number can be reduced to polynomial.

Lemma 7 *In any optimal schedule, the maximal job length of every batch is smaller than the minimum job length of jobs that are available at the starting time of the batch but not processed in the batch.*

Proof. If some optimal schedule does not obey the rule, we assume the batch is j . We can move the shorter jobs scheduled later to the batch j . The result is that we get another schedule with less total completion time. This is a contradiction $\square$

Lemma 8 *For the subset U (or V) of above equation, we need to consider at most n^{3s} different cases, where s is defined in Lemma 3.*

Proof. According to above lemma, once a job j is processed, all the jobs with release time no later than that of j and processing time no larger than that of j must be processed. Therefore, for each release time, there are at most n different choices. For each block, there are at most s distinct release times. Regarding 3 blocks in total, there are at most n^{3s} different choices. $\square$

Another difficulty which prevents us from implementing the Dynamic Programming equation comes from the possibly large number of blocks. But by Lemma 5 and Lemma 6, any job will be processed after waiting at most 3 blocks. Therefore, we actually only need handle no more than $3n$ blocks by ignoring the empty blocks within which no jobs need to be processed. With above analysis, the following corollary follows.

Corollary 1 *The dynamic programming equation will process at most $3n$ blocks.*

The remaining work is to compute $W(i+1, F', F, U - V)$. In next section, we will give detailed description on how to get a $(1 + \delta + \delta^2)$ -optimal schedule for $W(i+1, F', (1 + \delta + \delta^2)F, U')$, where $U' = U \setminus V$. Of course, we can also get a PTAS for our problem from the relaxed computation of $W(i+1, F', F, U')$.

3 Scheduling Jobs within a Block

Now let us describe briefly the sub-problem: given constant number of distinct release times (say $0 = r_1 < \dots < r_s$), each associated with a set of jobs $P_1, \dots, P_s$ and given a deadline D , s.t. all jobs must be finished before the deadline. The objective function is to minimize the total completion time.

We will deal with this problem in the following three subsections. First we get a $(1 + \delta)$ -optimal schedule algorithm of two release times. Then we extend the algorithm to constant number of release times. Finally we give a $(1 + \delta)$ -approximation algorithm for constant number of release times with relaxed deadline.

3.1 The $1 + \delta$ Approximation Algorithm for Two Distinct Release Times

In this subsection, we consider the special case of two distinct release times. Let P_1 be the set of jobs arriving at time zero, P_2 the set of jobs arriving at time r . The job system is denoted by $((P_1, 0); (P_2, r))$ and the optimal total completion time is denoted by $OPT((P_1, 0); (P_2, r))$. We denote by P^{k_1} the set of first k_1 shortest jobs in P_1 , where the jobs are ordered by $p^{1_1} \leq \dots \leq p^{k_1}$.

Let $N = \max(n, \lceil \frac{1}{\delta} \rceil)$. We will use N this way throughout the following subsections: we divide the time interval $[0, r]$ into N^3 equal sub-intervals and force each batch starting between $[0, r]$ to start at time $i \times r \frac{1}{N^3}$ for some $0 \leq i \leq N^3$.

Lemma 9 *With $1 + \delta$ loss, we can force a batch start at $i \frac{r}{N^3}$ if the batch starts within $((i-1) \frac{r}{N^3}, i \frac{r}{N^3}]$.*

Proof. By the rounding of the batches' start times, we know that the second batch will be delayed at most $\frac{r}{N^3}$, the i th batch will be delayed at most $(i-1) \frac{r}{N^3}$. In summary, the cost caused by the delay is at most $\frac{r}{N}$. Since there are some jobs scheduled no earlier than r , the $OPT((P_1, 0); (P_2, r)) > r$. The cost increased is bounded by $\frac{1}{N} OPT((P_1, 0); (P_2, r)) \geq \frac{1}{\delta} OPT((P_1, 0); (P_2, r))$. For other jobs scheduled after r , we can compute their optimal schedule[2] and combine it with the former part. This completes the proof. $\square$

By Lemma 7, we know that if a job is scheduled, all the smaller jobs released must be scheduled. This property reduces the possible states greatly.

Now let us give the dynamic programming equation. Let $M_{k_1} = OPT_0((P_1 - P^{k_1}) \cup P_2)$ denotes both the optimal schedule and its value of jobs in $(P_1 - P^{k_1}) \cup P_2$ which all arriving at time zero. Assuming all jobs in S have been finished by time t , let $F(S, t)$ denote the value of the optimal schedule of the remaining jobs with the earliest batch's start time t .

The initialization is for $i = 0, \dots, |P_1|$ and $j = 0, \dots, N^3$,

$$F(P^{i_1}, \max(j \frac{r}{N^3} + p^{i_1}, r)) = \max(j \frac{r}{N^3} + p^{i_1}, r)(n - i) + M_{i_1}. \quad (2)$$

For others

$$F(S, t) = +\infty \quad (3)$$

The recursion is for $i_1 = 0, \dots, |P_1| - 1$ and $j = N^3 - 1, \dots, 1, 0$:

$$F(P^{i_1}, j \frac{r}{N^3}) = \min_{i_1+1 \leq l_1 \leq |P_1|} \{ |l_1 - i_1| (p^{l_1} + j \frac{r}{N^3}) + F(P^{l_1}, \min\{\max\{j \frac{r}{N^3} + p^{l_1}, r\}, (j + \lceil \frac{p^{l_1}}{N^3} \rceil) \frac{r}{N^3} \}) \}. \quad (4)$$

To find the optimal solution to the rounded start time problem, we compute $F(0, 0)$. By backtracking, we can find the optimal schedule. The algorithm complexity is $O(n^3 \times n \times n) = O(n^5)$.

3.2 The $1 + \delta$ Approximation Algorithm for Constant Number of Distinct Release Times

Now let us extend above idea to the case for constant number of distinct release times. Assume that there are s distinct release times. Let P_i be the set of jobs arriving at time r_i , P^{k_i} the set of k_i shortest jobs in P_i and p^{k_i} the length of the longest job in P^{k_i} .

Similar to the special case of two release times, we have the following Lemma.

Lemma 10 *With $1 + \delta$ loss, we can force each batch start at $i \frac{r_s}{N^3}$ if it starts within $((i - 1) \frac{r_s}{N^3}, i \frac{r_s}{N^3}]$ (for some $i = 0, 1, \dots, N^3$).*

Now let us give the dynamic programming equation. Let

$M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}} = OPT_0((\bigcup_{i=1}^{s-1} P_i \setminus \bigcup_{i=1}^{s-1} P^{k_i}) \cup P_s)$ denote both the optimal schedule and its value of jobs $(\bigcup_{i=1}^{s-1} P_i \setminus \bigcup_{i=1}^{s-1} P^{k_i}) \cup P_s$ with the assumption that

all of them arrive at time zero. $F(S, t)$ is defined the same as in the proceeding subsection. For a collection of sets $\mathcal{A} = (A_1, A_2, \dots, A_m)$, $(a_1, a_2, \dots, a_m)$ is called a representative system of $\mathcal{A}$ if $a_i \in A_i (1 \leq i \leq m)$. For each $1 \leq i \leq s - 1$, denote by A_i the set of $\{0, 1, \dots, |P_i|\}$ and $\mathcal{A} = (A_1, A_2, \dots, A_{s-1})$; denote by B_i the set of $\{k_i + 1, \dots, |P_i|\}$ and $\mathcal{B} = (B_1, \dots, B_{s-1})$. We use RSC for the set of representative systems of $\mathcal{C}$ under without ambiguity.

The initialization is for all possible representative systems $(k_1, \dots, k_{s-1})$ of $\mathcal{A}$, $e = 1, \dots, s-1$ and $j = 0, 1, \dots, N^3$ (if $j \frac{r_s}{N^3} \in [r_x, r_{x+1})$, force $k_{x+1} = \dots = k_{s-1} = 0$):

$$\begin{aligned} & F\left(\bigcup_{i=1}^{s-1} P^{k_i}, \max\left(j \frac{r_s}{N^3} + p^{k_e}, r_s\right)\right) \\ &= (|P_s| + \sum_{l=1}^{s-1} (|P_l| - k_l)) (\max\{p^{k_e} + j \frac{r_s}{N^3}, r_s\}) + M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}} \end{aligned} \quad (5)$$

The recursion is for all $j = N^3 - 1, \dots, 1, 0$ and all possible representative systems $(k_1, \dots, k_{s-1})$ of $\mathcal{A}$ (if $j \frac{r_s}{N^3} \in [r_x, r_{x+1})$, force $k_{x+1} = \dots = k_{s-1} = 0$):

$$\begin{aligned} & F\left(\bigcup_{i=1}^{s-1} P^{k_i}, j \frac{r_s}{N^3}\right) \\ &= \min_{(\gamma_1, \dots, \gamma_{s-1}) \in RSB} \left\{ \sum_{i=1}^{s-1} |\gamma_i - k_i| \left(\max_{1 \leq i \leq s-1} \{p^{\gamma_i}\} + j \frac{r_s}{N^3} \right) \right. \\ & \quad \left. + F\left(\bigcup_{i=1}^{s-1} P^{\gamma_i}, \min\left\{\max\left\{j \frac{r_s}{N^3} + \max_{1 \leq i \leq s-1} \{p^{\gamma_i}\}, r_s\right\}, \left(j + \left\lceil \frac{\max_{1 \leq i \leq s-1} \{p^{\gamma_i}\}}{\frac{r_s}{N^3}} \right\rceil\right) \frac{r_s}{N^3} \right\}\right) \right\}. \end{aligned} \quad (6)$$

For others

$$F(S, t) = +\infty \quad (7)$$

We can get the optimal solution for the rounded start time problem by computing $F(0, 0)$. The corresponding schedule is found by backtracking. By Lemma 7, we know that there are $O(n^s)$ different choices for $\bigcup_{i=1}^{s-1} P^{k_i}$ and $\bigcup_{i=1}^{s-1} P^{\gamma_i}$. In total there are $O(n^{2s})$ different choices. The algorithm's time complexity is $O(N^3 \times n^s \times n^s) = O(N^{2s+3})$.

3.3 Schedule the Jobs within a Relaxed Deadline

Now let us consider the case with deadline. Assume that the deadline is D . The difficulty here is that it is not straightforward to finish the initialization step. We have to make a deeper analysis.

In the first step, we judge if it is feasible to schedule all the jobs within the deadline D . By the work of Lee and Uzsoy [6], we can get the minimal makespan schedule in $O(n^2)$ time. If the optimal makespan is larger than the deadline, we set $W(i+1, F', F, U') = +\infty$.

Otherwise, we divide the time axle $[0, r_s]$ into N^3 equal intervals and force all batches which starts within $[0, r_s)$ must start at time $i \frac{r_s}{N^3}$ for some i with $0 \leq i \leq N^3$.

Since for each of the possible start time between $[0, r_s]$, it is possible that there are $O(n)$ different batches which may cross time interval $[0, r_s)$ into $[r_s, D]$ and delay the start time of batches in $M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}}$. We enumerate all possible start times for them. There are roughly $O(n \times N^3)$ different possible start times for $M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}}$ in total.

For each possible start time r'_s of $M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}}$, we set the initialization value as following:

if the makespan of $OPT_0(((P_1, \dots, P_{s-1}) - (P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}})) \cup P_s)$ is smaller than $D + \frac{r_s}{N} - r'_s$, set:

$$F(P_{\ell=1, \dots, s-1}^{k_\ell}, r'_s) = (|P_s| + \sum_{l=1}^{s-1} (|P_l| - k_l))r'_s + M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}}; \quad (8)$$

if the makespan of the schedule is larger than $D + \frac{r_s}{N} - r'_s$, we divide $[r'_s, D + \frac{r_s}{N}]$ into N^3 intervals and do dynamic programming for the remaining jobs. In the initialization step, we find that there can be at most one batch starting after $D + \frac{r_s}{N}$. Since if there is one batch starting after $D + \frac{r_s}{N}$, then the sum of the length of the batches start between r'_s and $D + \frac{r_s}{N}$ must be no less than $\frac{N-1}{N}(D + \frac{r_s}{N} - r'_s)$ due to the fact that the possible spaces between these batches are produced only by rounding the starting times and therefore the sum of these spaces can be no more than $\frac{1}{N}(D + \frac{r_s}{N} - r'_s)$. Therefore, the length of the longest batch between r'_s and $D + \frac{r_s}{N}$ is no less than $\frac{1}{n} \frac{N-1}{N}(D + \frac{r_s}{N} - r'_s)$ and so are the batches start after $D + \frac{r_s}{N}$. If there are two batches start after $D + \frac{r_s}{N}$, then the completion time of the last batch will be no less than $D + \frac{r_s}{N} + \frac{2}{n} \frac{N-1}{N}(D + \frac{r_s}{N} - r'_s) > D + \frac{r_s}{N} + \frac{D + \frac{r_s}{N} - r'_s}{N}$, which implies an infeasible solution. In the initialization step, we only need consider the case that at most one batch can start after $D + \frac{r_s}{N}$. If the completion time of this batch is larger than $D + \frac{r_s}{N} + \frac{D + \frac{r_s}{N} - r'_s}{N}$, we set its completion time as ∞ . Once we have finished the initialization step, the remaining thing is the same as in the case without the deadline. The complexity of the algorithm still is $O(N^{2s})$.

Lemma 11 *We get a $(1 + \frac{1}{N} + \frac{1}{N^2})$ -optimal schedule for $W(i+1, F', (1 + \frac{1}{N} + \frac{1}{N^2})F, U')$.*

Proof. Let us analyze the cost caused by rounding the start times. If the optimal total completion time is greater than D , then the cost caused by such rounding is at most $\frac{\frac{r_s}{N} + \frac{D + \frac{r_s}{N} - r'_s}{N}}{D} \leq \frac{1}{N} + \frac{1}{N^2}$. If the optimal total completion time is smaller than D , then the makespan of the optimal schedule must be smaller than D . Correspondingly, the makespan of $M_{P^{k_1}, P^{k_2}, \dots, P^{k_{s-1}}}$ should be less than $D + \frac{r_s}{N} - r'_s$. The cost caused by above rounding is at most $N^2 \times \frac{r_s}{N^3} < \frac{1}{N} OPT$. This completes proof. $\square$

Theorem 1 *We can find a $(1 + \delta + \delta^2)$ -optimal schedule for $W(i+1, F', (1 + \frac{1}{N} + \frac{1}{N^2})F, U')$.*

4 Conclusion

After computing all the blocks, the loss is at most $(1 + \delta + \delta^2)$ by stretching the repeated part of $W(i, F^i, F^{i+1}, V^i)$ and $W(i+1, F^{i+1}, F^{i+2}, V^{i+1})$ for all $i \geq 1$. Finally we have the following conclusion by choosing appropriate δ .

Theorem 2 *We can find a $(1 + \epsilon)$ -optimal solution to $1|r_j, B > n| \sum C_j$ in the time of $O(n^{5 \log_{1+\epsilon}(1+\frac{1}{\epsilon})})$.*

Thus, we have got a PTAS for the total completion time for unbounded batch machine. The major open problem left in this field is for minimization of the total completion time for the bounded batch machine ($B < n$). The only known non-trivial result is the 2-approximation algorithm for a system of jobs all with release time zero [5]. In addition, it would be interesting to know if approximation algorithms can be extended to the weighted case. Notice that both our result and that of Hochbaum and Landy work are for the unweighted case.

References

1. Foto Afrati, Evripidis Bampis, Chandra Chekuri, David Karger, Claire Kenyon, Sanjeev Khanna, Ioannis Milis, Maurice Queyranne, Martin Skutella, Cliff Stein, and Maxim Sviridenko, Approximation Schemes for Minimizing Average Weighted Completion Time with Release Dates, the proceeding of the 40th Annual Symposium on Foundations of Computer Science, Oct. 1999, New York, pp 32-43. [27](#), [28](#), [29](#)
2. P. Brucker, A. Gladky, H. Hoogeveen, M. Kovalyov, C. Potts, T. Tautenhahn, and S. van de Velde, Scheduling a batching machine, *Journal of Scheduling*, **1**, pp.31-54, (1998). [27](#), [31](#)
3. X.Deng, Y.Zhang, Minimizing Mean Response Time in Batch Processing Systme, COCOON'99, Volume 1627 of Lecture Notes in Computer Science, pages 231-240, Tokyo, Japan, July 1999. Springer-verlag. [27](#)
4. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. Rinnooy Kan, Optimization and approximation in deterministic sequencing and scheduling. *Annals of Discrete Math.* 5(1979). [26](#)
5. D. S. Hochbaum and D. Landy, Scheduling semiconductor burn-in operations to minimize total flowtime, *Operations Research* **45** (1997), 874-885. [27](#), [35](#)
6. C. Y. Lee and R. Uzsoy, Minimizing makespan on single batch processing machine with dynamic job arrivals, Research Report, Dept. Industrial and System Engineering, Univ. Florida, Jan, 1996. [33](#)
7. C. Y. Lee, R. Uzsoy and L. A. Martin Vega. Efficient algorithms for scheduling semiconductor burn-in operations. *Operations Research*, 40:764-775, 1992. [26](#)

A Polynomial Time Approximation Scheme for the Multi-vehicle Scheduling Problem on a Path with Release and Handling Times

Yoshiyuki Karuno¹ and Hiroshi Nagamochi²

¹ Kyoto Institute of Technology
Matsugasaki, Sakyo, Kyoto 606-8585, Japan
`karuno@ipc.kit.ac.jp`

² Toyohashi University of Technology
Tempaku-cho, Toyohashi 441-8580, Japan
`naga@ics.tut.ac.jp`

Abstract. In this paper, we consider a scheduling problem of vehicles on a path. Let $G = (V, E)$ be a path, where $V = \{v_1, v_2, \dots, v_n\}$ is its set of n vertices and $E = \{\{v_j, v_{j+1}\} \mid j = 1, 2, \dots, n-1\}$ is its set of edges. There are m vehicles ($1 \leq m \leq n$). The travel times $w(v_j, v_{j+1})$ ($= w(v_{j+1}, v_j)$) are associated with edges $\{v_j, v_{j+1}\} \in E$. Each job j which is located at each vertex $v_j \in V$ has release time r_j and handling time h_j . Any job must be processed by exactly one vehicle. The problem asks to find an optimal schedule of m vehicles that minimizes the maximum completion time of all the jobs. The problem is known to be NP-hard for any fixed $m \geq 2$. In this paper, we present a polynomial time approximation scheme $\{A_\epsilon\}$ to the problem with a fixed m . Our algorithm can be extended to the case where G is a tree so that a polynomial time approximation scheme is obtained if m and the number of leaves in G are fixed.

1 Introduction

In this paper, we consider a scheduling problem of vehicles on a path with release and handling times. The scheduling problem of vehicles, such as AGVs (automated guided vehicles), handling robots, buses, trucks and so forth, on a given road network is an important topic encountered in various applications. In particular, in FMS (flexible manufacturing system) environment, scheduling of the movement of AGVs, which carry materials and products between machining centers, has a vital effect on the system efficiency.

The single-vehicle scheduling problem (VSP, for short) contains the traveling salesman problem (TSP) and the delivery man problem (DMP) [1] as its special cases. In the TSP, a salesman (a vehicle) visits each of n customers (jobs) situated at different locations on a given network before returning to the initial location. The objective is to minimize the tour length. In the DMP, the same scenario is considered but the objective is to minimize the total completion time of all the jobs. The VSP usually takes into account the time constraints of jobs (i.e.,

release, handling and/or due times), and therefore other important objective functions, such as the tour time, the maximum completion time of jobs, the maximum lateness from the due times and so forth, are also considered. Since path and tree are important network topologies from both practical and graph theoretical views, VSPs on these networks have been studied in several papers, e.g., Psaraftis, Solomon, Magnanti and Kim [12], Tsitsiklis [13], Averbakh and Berman [1], Karuno, Nagamochi and Ibaraki [5,6,7], Nagamochi, Mochizuki and Ibaraki [10,11].

The multi-vehicle scheduling problem (MVSP, for short), which is a more general problem than the VSP, on a path to be discussed here is called PATH-MVSP, and the number of vehicles is denoted by m ($1 \leq m \leq n$). Problem PATH-MVSP asks to find an optimal schedule of m vehicles (i.e., their optimal sequences of jobs) that minimizes the maximum completion time of all the jobs. Note that the objective is equivalent to minimizing the maximum workload of all the vehicles. The PATH-MVSP is NP-hard for any fixed $m \geq 2$, since it contains the PARTITION (e.g., see Garey and Johnson [2]) as its special case. The PATH-MVSP with $m = 1$ (i.e., the VSP on a path) was proved by Tsitsiklis [13] to be NP-hard if the initial location of a vehicle is specified. The PATH-MVSP with $m = 1$ is 2-approximable due to the results by Psaraftis et al. [12], and it was shown to be 1.5-approximable by Karuno et al. [7] if the initial and goal locations of a vehicle are specified as one end of the path. Recently, Karuno and Nagamochi [8] first presented an $O(mn^2)$ time 2-approximation algorithm to the PATH-MVSP with an arbitrary m , where edge weights are assumed to be symmetric (i.e., $w(v_j, v_{j+1}) = w(v_{j+1}, v_j)$).

In this paper, for the PATH-MVSP with a fixed number m of vehicles and symmetric edge weights, we present a *polynomial time approximation scheme*, i.e., a family of algorithms $\{A_\varepsilon\}$ with the following property: for any $\varepsilon > 0$, A_ε delivers a schedule with its maximum completion time at most $(1 + \varepsilon)$ times the optimal. The running time is bounded by a polynomial in n , but by an exponential in $1/\varepsilon$ (in this paper, we assume that m is a fixed number). Our approximation scheme A_ε is based on approximation of the problem by rounding given release times, and the fact that any schedule with λ gaps consists of $(\lambda + 1)$ gapless schedules on subpaths of a given path, where an edge is called a *gap* in a schedule if it is traversed by none of the vehicles. The scheme is a two-fold dynamic programming, one for computing an optimal gapless schedule to the problem with rounded release times, and the other for finding the best schedule to the original problem by combining several gapless schedules over all choices of gaps on the path. Rounding given release times has been used to design some polynomial time approximation schemes for solving scheduling problems subject to release times, e.g., see Hall and Shmoys [4], Hall [3], Kovalyov and Werner [9]. In particular, one of our dynamic programmings follows Kovalyov and Werner's approach [9] (which dealt with problem $F2/r_j/C_{max}$, i.e., the two-machine flowshop scheduling problem with release times to minimize the maximum completion time).

The remainder of this paper is organized as follows. In Section 2, we provide the mathematical description of the PATH-MVSP. In Section 3, we explain basic properties of the PATH-MVSP with rounded release times. In Section 4, we discuss a polynomial time approximation scheme for finding an optimal gapless schedule, and in Section 5, we present a polynomial time approximation scheme $\{A_\varepsilon\}$ to the original problem, such that, for any $\varepsilon > 0$, algorithm A_ε delivers a schedule with its maximum completion time at most $(1 + \varepsilon)$ times the optimal attained by general schedules. We also mention that our algorithm can be extended to the case where G is a tree, showing that a polynomial time approximation scheme exists for the problem if m and the number of leaves in G are fixed. Finally, in Section 6, we give some concluding remarks.

2 Multi-vehicle Scheduling Problem on a Path

2.1 Problem Description

Problem PATH-MVSP is formulated as follows. Let $G = (V, E)$ be a path network, where $V = \{v_1, v_2, \dots, v_n\}$ is its set of n vertices and $E = \{\{v_j, v_{j+1}\} \mid j = 1, 2, \dots, n-1\}$ is its set of edges. In this paper, we assume that vertex v_1 is the *left end* of G , and v_n the *right end* of it. There is a job j at each vertex $v_j \in V$. The job set is denoted by $J = \{j \mid j = 1, 2, \dots, n\}$. There are m vehicles on G ($1 \leq m \leq n$), which are assumed to be identical. Each job must be processed by exactly one vehicle.

The *travel time* of a vehicle is $w(v_j, v_{j+1}) \geq 0$ to traverse $\{v_j, v_{j+1}\} \in E$ from v_j to v_{j+1} , and is $w(v_{j+1}, v_j) \geq 0$ to traverse it in the opposite direction. Edge weight $w(v_j, v_{j+1})$ for $\{v_j, v_{j+1}\} \in E$ is called *symmetric* if $w(v_j, v_{j+1}) = w(v_{j+1}, v_j)$ holds. In this paper, we assume that all edge weights are symmetric. The travel time for a vehicle to move from vertex v_i to vertex v_j on G is the sum of edge weights belonging to the unique path from v_i to v_j . Each job $j \in J$ has its release time $r_j \geq 0$ and handling time $h_j \geq 0$: That is, a vehicle cannot start processing job j before time r_j , and it takes h_j time units to process job j (no interruption of the processing is allowed). A vehicle at vertex v_j may wait until time r_j to process job j , or move to other vertices without processing job j if it is more advantageous (in this case, the vehicle must come back to v_j later to process job j , or another vehicle must come to v_j to process it). An instance of the problem PATH-MVSP is denoted by $(G(= (V, E)), r, h, w, m)$.

A motion schedule of the m vehicles is completely specified by m sequences of jobs $\pi^{[p]} = (j_1^{[p]}, j_2^{[p]}, \dots, j_{n_p}^{[p]})$, $p = 1, 2, \dots, m$, where n_p is the number of jobs to be processed by vehicle p (hence, it holds that $\sum_{p=1}^m n_p = n$), and $j_i^{[p]}$ is its i -th job; i.e., vehicle p is initially situated at vertex $v_{j_1^{[p]}}$, starts processing job $j_1^{[p]}$ at time $\max\{0, r_{j_1^{[p]}}\}$, and takes $h_{j_1^{[p]}}$ time units to process it. After completing job $j_1^{[p]}$, the vehicle immediately moves to $v_{j_2^{[p]}}$ along the unique path from $v_{j_1^{[p]}}$ to $v_{j_2^{[p]}}$, taking travel time of the path (i.e., $w(v_{j_1^{[p]}}, v_{j_1^{[p]}+1}) + \dots + w(v_{j_2^{[p]}-1}, v_{j_2^{[p]}})$ or $w(v_{j_1^{[p]}}, v_{j_1^{[p]}-1}) + \dots + w(v_{j_2^{[p]}+1}, v_{j_2^{[p]}})$), and processes job $j_2^{[p]}$ after waiting until

time $r_{j_2^{[p]}}$ if necessary, and so on, until it completes the last job $j_{n_p}^{[p]}$. A schedule is denoted by a set of m sequences of jobs $\pi = \{\pi^{[1]}, \pi^{[2]}, \dots, \pi^{[m]}\}$. The completion time of vehicle p (i.e., the workload of it) is defined as the completion time of its last job $j_{n_p}^{[p]}$, which is denoted by $C(\pi^{[p]})$. The objective is to find a π that minimizes the maximum completion time of all the jobs, i.e.,

$$C_{max}(\pi) = \max_{1 \leq p \leq m} C(\pi^{[p]}). \quad (1)$$

In this paper, we denote by π^* an optimal schedule and by C_{max}^* the minimum of the maximum completion time $C_{max}(\pi^*)$.

2.2 Subpath and Subinstance

Let $V(i, j) = \{v_i, v_{i+1}, \dots, v_j\} (\subseteq V)$, where $i \leq j$. Define $G(i, j) = (V(i, j), E(i, j))$ be a subpath of a given path $G = (V, E)$ induced by $V(i, j)$ and $E(i, j) = \{\{v_{j'}, v_{j'+1}\} \mid j' = i, i+1, \dots, j-1\} (\subseteq E)$, and $J(i, j) = \{i, i+1, \dots, j\} (\subseteq J)$ the corresponding subset of jobs to the subpath $G(i, j)$. This definition states that $G(1, n) = G$ and $J(1, n) = J$.

Next consider the scheduling problem of $p (\leq m)$ vehicles on $G(i, j)$. This is a *subinstance* of the original instance (G, r, h, w, m) . We denote this subinstance by $(G(i, j), r, h, w, p)$; i.e., scheduling p vehicles on subpath $G(i, j) = (V(i, j), E(i, j))$ of the given path G with release times $r_{j'}$ and handling times $h_{j'}$ for $j' \in J(i, j)$ and with edge weights $w(v_{j'}, v_{j'+1}) (= w(v_{j'+1}, v_{j'}))$ for $\{v_{j'}, v_{j'+1}\} \in E(i, j)$ (hence, the original instance is denoted by (G, r, h, w, m) as well as $(G(1, n), r, h, w, m)$).

2.3 Zone Schedule and Gapless Schedule

For a schedule π , assume that a vehicle *covers* a subpath $G(i, j) = (V(i, j), E(i, j))$: That is, all jobs processed by the vehicle are on $G(i, j)$ and two jobs i and j located at the end vertices of $G(i, j)$ have to be processed by it. But, there may be some jobs j' ($i < j' < j$) processed by other vehicles. Then, the subpath $G(i, j)$ for the vehicle is referred to as its *zone*.

A feasible schedule π using m' vehicles ($m' \leq m$) is referred to as a *zone schedule* if any two zones in π do not intersect and thus there are $m' - 1$ edges which are not traversed by any vehicle. Such an edge that is not traversed by any vehicle is called a *gap*. A schedule π is called *gapless* if each edge $\{v_j, v_{j+1}\} \in E$ is traversed at least once (from v_j to v_{j+1} or from v_{j+1} to v_j) by some vehicle. Define

$$Z = W + H, \quad (2)$$

where $W = \sum_{j=1}^{n-1} w(v_j, v_{j+1})$ and $H = \sum_{j=1}^n h_j$.

3 Rounding Given Release Times

In this section, we provide a dynamic programming approach to the PATH-MVSP with rounded release times, based on Kovalyov and Werner's notion [9]. This will become a basis of the proposed polynomial time approximation scheme in Section 5.

3.1 Basic Properties

The optimal schedule for the PATH-MVSP with a single vehicle (i.e., $m = 1$) is trivial if release times of all jobs are equal: That is, the single vehicle simply moves from v_1 to v_n or from v_n to v_1 , processing jobs one by one. Such a schedule for a vehicle is called a *1-way* schedule. Let $\pi^{[1]} = (1, 2, \dots, n)$ and $\mu^{[1]} = (n, n-1, \dots, 1)$ be these 1-way schedules. Their completion times can be computed as $C(\pi^{[1]}) = h_1 + \sum_{j=2}^n \{w(v_{j-1}, v_j) + h_j\}$ and $C(\mu^{[1]}) = h_n + \sum_{j=1}^{n-1} \{w(v_{j+1}, v_j) + h_j\}$, respectively, where $C(\pi^{[1]}) = C(\mu^{[1]})$ holds by the symmetry of edge weights.

We first restrict our attention to the PATH-MVSP that has a fixed number of distinct release times, for which we can enjoy the next lemma due to Hall [3].

Lemma 1. [3] *Given a polynomial time approximation scheme for the restricted version of the PATH-MVSP in which there is a fixed (but arbitrary) number of distinct release times, then there is a polynomial time approximation scheme for the PATH-MVSP.*

Proof. Let $r^* = \max_{j \in J} r_j$ and $\Delta = \varepsilon r^*/2$, where $\Delta \leq \varepsilon C_{max}^*/2$ holds since $r^* \leq C_{max}^*$. We consider the following algorithm for the unrestricted version of the PATH-MVSP. Round each release time r_j down to the nearest multiple of Δ , i.e.,

$$r'_j = \Delta \lfloor r_j / \Delta \rfloor, \quad j = 1, 2, \dots, n, \quad (3)$$

where $\lfloor x \rfloor$ is the largest integer no greater than x . Consequently, the number of distinct release times r'_j , denoted by K , is bounded as follows:

$$K \leq 1 + r^*/\Delta \leq 1 + 2/\varepsilon. \quad (4)$$

Apply an approximation scheme to obtain a $(1 + \varepsilon/2)$ -approximation solution π for the problem with release times r'_j . Postpone each job's starting time σ_j in the π to $\sigma_j + \Delta$ so that the schedule remains feasible for the original problem. Let C'_{max} be the optimal objective value for the problem with release times r'_j . Since the rounded problem is less constrained than the original one, we obtain $C'_{max} \leq C_{max}^*$, and the maximum completion time of the final schedule is bounded by $(1 + \varepsilon/2)C'_{max} + \Delta \leq (1 + \varepsilon)C_{max}^*$. $\square$

Let $\rho_1 < \rho_2 < \dots < \rho_K$ be the K distinct release times. We set $\rho_{K+1} = \infty$ for notational convenience. As in the proof of Lemma 1, for a given schedule π ,

we denote the starting time of job j by σ_j . We refer to the jobs j processed by vehicle p with $\rho_k \leq \sigma_j < \rho_{k+1}$ as the k -th *interval set* of the vehicle, denoted by $J_{p,k}$. Note that only jobs with $r_j \leq \rho_k$ can belong to the k -th interval set. Since there is no other release time between ρ_k and ρ_{k+1} , an optimal schedule to process the jobs in the k -th interval set by a single vehicle is assumed to be a 1-way schedule.

3.2 Dynamic Programming

In this section, we provide a dynamic programming algorithm for the PATH-MVSP with K distinct release times assuming that all handling times and all edge weights are *integers* (we discuss in Section 4 how to round given handling times and edge weights into integers). Now Z in (2) is an integer.

For each $p = 1, 2, \dots, m$ and $k = 1, 2, \dots, K$, we consider a 1-way subschedule $\pi_k^{[p]}$ and its reversal $\mu_k^{[p]}$, each of which processes the jobs in the k -th interval set by the p -th vehicle, and denote their completion times by $C_{p,k}$ ($= C(\pi_k^{[p]}) = C(\mu_k^{[p]})$). For the PATH-MVSP with K distinct release times, there are at most K^n assignments of n jobs to K interval sets, and at most m^n assignments of n jobs to m vehicles; we call each of the $K^n m^n$ ways of assigning n jobs to K interval sets and m vehicles as a *job assignment*. Moreover, for each interval set of a vehicle, there are two 1-way subschedules. Choosing the best schedule among all these possible cases would take $\Omega(2^K K^n m^n)$ time and space.

Fixed a job assignment, an optimal schedule of each vehicle p obeying the job assignment can be easily constructed by concatenating 1-way subschedules for all interval sets (recall that there are only two 1-way subschedules for each interval set of vehicle p), where we only need to compute the least time to resume the first job in the $(k+1)$ -interval set after finishing the last job in the k -interval set for each $k = 1, 2, \dots, K-1$. During the concatenation, if the starting time of the last job in the k -interval becomes equal to or larger than ρ_{k+1} due to the finishing time of the previous job, then we can abort the construction for the job assignment since it violates the definition of interval sets; we call such a job assignment *violating*. Notice that, in this computation for constructing an optimal schedule of vehicle p , we only need to know the completion times $C_{p,k}$ of two 1-way subschedules to process all jobs in each interval set $J_{p,k}$, and the first and last jobs to be processed in the interval set $J_{p,k}$ (the set of jobs in each interval set is not necessarily stored as long as $C_{p,k}$ for all p, k are stored and the n jobs are assumed to be correctly assigned according to a job assignment).

To facilitate the above computation, we define a table $X^{(n)}$ as the following $3Km$ -tuple X :

$$\begin{aligned} X = & (C_{11}, C_{12}, \dots, C_{1K}; C_{21}, C_{22}, \dots, C_{2K}; \dots; C_{m1}, C_{m2}, \dots, C_{mK}; \\ & L_{11}, L_{12}, \dots, L_{1K}; L_{21}, L_{22}, \dots, L_{2K}; \dots; L_{m1}, L_{m2}, \dots, L_{mK}; \\ & R_{11}, R_{12}, \dots, R_{1K}; R_{21}, R_{22}, \dots, R_{2K}; \dots; R_{m1}, R_{m2}, \dots, R_{mK}), \end{aligned}$$

where $L_{p,k}$ (resp., $R_{p,k}$) denotes the index of the left (resp., right) end vertex of the zone of $\pi_k^{[p]}$ (which is also the zone of $\mu_k^{[p]}$), implying that $L_{p,k}$ and $R_{p,k}$

are respectively the first and last jobs to be processed during $\pi_k^{[p]}$ while they are respectively the last and first jobs during $\mu_k^{[p]}$. We here assume that a table $X^{(n)}$ is constructed from a job assignment. We remark that different job assignments may produce the same table $X^{(n)}$ (this reduces the number of different tables to at most $Z^K n^K n^K$ by $C_{p,k} \leq Z$, $L_{p,k} \leq n$ and $R_{p,k} \leq n$).

From such a table $X^{(n)}$, an optimal schedule of each vehicle p obeying the job assignment which produces the table can be constructed by concatenating 1-way subschedules by choosing one of $\pi_k^{[p]}$ and $\mu_k^{[p]}$ for each k , taking $O(2^K K)$ time to find the best way of concatenation.

Now consider how to compute the set $\{X^{(n)}\}$ of such tables. Let $\mathcal{X}^{(j)}$ denote the set of tables $X^{(j)}$ defined for the jobs in $J(1, j)$. Suppose that the set $\mathcal{X}^{(j-1)}$ of tables has been computed. We add the j -th job to the k -th interval set of vehicle p for all p and k with $r_j \leq \rho_k$, and update each $X \in \mathcal{X}^{(j-1)}$ into a table $X_{p,k}$ in $\mathcal{X}^{(j)}$ by computing

$$C_{p,k} := C_{p,k} + w(v_{R_{p,k}}, v_j) + h_j, \quad R_{p,k} := j, \quad L_{p,k} := j \text{ (if } L_{p,k} = 0), \quad (5)$$

where we let $w(v_{R_{p,k}}, v_j) = 0$ if $R_{p,k} = 0$. We suppose that $w(u, v)$ for all $u, v \in V$ have been computed, and hence $X_{p,k}$ for each p, k can be updated in $O(1)$ time.

We are now ready to describe an algorithm for computing $\mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \dots, \mathcal{X}^{(n)}$ by a dynamic programming and for constructing an optimal schedule from $\mathcal{X}^{(n)}$.

Algorithm A'

Step 1 (Initialization): Set $X^{(0)}$ to be $\{(0, 0, \dots, 0; \dots; 0, 0, \dots, 0)\}$, a $3Km$ -tuple with zero entries.

Step 2 (Generation of $\mathcal{X}^{(1)}, \mathcal{X}^{(2)}, \dots, \mathcal{X}^{(n)}$): For $j = 1, 2, \dots, n$, perform the following computations:

(i) Initialize $\mathcal{X}^{(j)}$ by $\mathcal{X}^{(j)} := \emptyset$.

(ii) For each table $X \in \mathcal{X}^{(j-1)}$, $k = 1, 2, \dots, K$ and $p = 1, 2, \dots, m$,

we add job j to the k -th interval set of the p -th vehicle in the X and
update X into $X_{p,k}$ according to (5);
 $\mathcal{X}^{(j)} := \mathcal{X}^{(j)} \cup \{X_{p,k}\}$ if $X_{p,k} \notin \mathcal{X}^{(j)}$.

Step 3 (Determination of an optimal schedule): For each table $X \in \mathcal{X}^{(n)}$, compute the maximum completion time of the corresponding schedule, where we discard table X if it turned out to be violating during the computation. (Recall the completion time of each vehicle can be obtained in $O(2^K K)$ time, and hence the maximum completion time in $O(2^K Km)$ time for each table.) A table $X \in \mathcal{X}^{(n)}$ with the minimum of maximum completion times corresponds to an optimal schedule, which can be obtained by backtracking.

The time complexity of this algorithm is evaluated as follows. As already observed, $|\mathcal{X}^{(j-1)}| \leq (n^2 Z)^{Km}$, $j = 2, \dots, n+1$. In Step 2(ii), we need to test whether $\mathcal{X}$ already contains a table which is identical with the table $X_{p,k}$ just generated. By preparing a complete set of all possible tables of $3Km$ -tuples,

where each table has a flag which indicates if the table belongs to the current $\mathcal{X}$, we can answer this query in $O(1)$ time and $O((n^2Z)^{Km})$ space. Hence Step 2 takes $O(Km(n^2Z)^{Km})$ time. Step 2 repeats n times, and Step 3 takes $O(2^K Km(n^2Z)^{Km})$ time for computing the minimum of the maximum completion times from a table $X \in \mathcal{X}^{(n)}$. Therefore, the time complexity of algorithm A' is at most $O(Km(n + 2^K)(n^2Z)^{Km})$. Note that this is a pseudopolynomial time algorithm if the number of vehicles m and the number of release times K are constants.

4 An Approximation Scheme for Finding an Optimal Gapless Schedule

Let $C_{max}(\pi_g^*)$ be the minimum of the maximum completion time of a gapless schedule, i.e., the maximum completion time of an optimal gapless schedule π_g^* in (G, r, h, w, m) . The following lower bound on the minimum of the maximum completion time $C_{max}(\pi_g^*)$ is immediately obtained:

$$LB = \frac{W + H}{m} \left(= \frac{Z}{m} \right). \quad (6)$$

Consider the PATH-MVSP with K distinct release times, where handling times and arbitrary edge weights may not be integers. We define

$$\delta = (\varepsilon Z)/4mn^2, \quad (7)$$

and replace the given handling times h_j and edge weights $w(v_j, v_{j+1})$ by scaled handling times and scaled edge weights

$$h'_j = \lfloor h_j/\delta \rfloor \quad \text{and} \quad w'(v_j, v_{j+1}) = \lfloor w(v_j, v_{j+1})/\delta \rfloor. \quad (8)$$

Note that $LB \leq C_{max}(\pi_g^*)$ implies $2\delta n^2 \leq (\varepsilon/2)C_{max}(\pi_g^*)$. Suppose that we have found an optimal gapless schedule π'_g and its maximum completion time $C'_{max}(\pi'_g)$ for the problem with scaled handling times and scaled edge weights. Let $C_{max}(\pi'_g)$ denote the maximum completion time of this schedule with respect to the original handling times and the original edge weights. Let π_g^* be an optimal gapless schedule to the original problem, and let $C'_{max}(\pi_g^*)$ denote the maximum completion time of the schedule π_g^* with respect to the scaled handling times and scaled edge weights. Making use of the inequalities

$$\delta h'_j \leq h_j < \delta(h'_j + 1) \quad \text{and} \quad \delta w'(v_j, v_{j+1}) \leq w(v_j, v_{j+1}) < \delta(w'(v_j, v_{j+1}) + 1), \quad (9)$$

and noticing that, for any schedule, if h_j and $w(v_j, v_{j+1})$ are increased by some value β , then the maximum completion time of the schedule increases at most $2\beta n^2$ (since each vertex or each edge is visited at most n times by a vehicle in a schedule), we obtain

$$\begin{aligned} C_{max}(\pi'_g) &< \delta C'_{max}(\pi'_g) + 2\delta n^2 \leq \delta C'_{max}(\pi_g^*) + 2\delta n^2 \\ &\leq C_{max}(\pi_g^*) + 2\delta n^2 \leq (1 + \varepsilon/2)C_{max}(\pi_g^*). \end{aligned}$$

Thus, any exact algorithm for the problem with scaled handling times h'_j and scaled edge weights $w'(v_j, v_{j+1})$ is a $(1 + \varepsilon/2)$ -approximation algorithm for the problem with original handling times and original edge weights.

We now define a family of algorithms $\{A'_\varepsilon\}$ as follows: A'_ε is the algorithm A' applied to the problem with scaled release times r'_j (see (3)), scaled handling times h'_j and scaled edge weights $w'(v_j, v_{j+1})$ (see (8)). The value Δ (see the proof of Lemma 1) is added to each job's starting time in the final schedule to make it feasible for the original problem. We have the following theorem:

Theorem 1. *The family of algorithms $\{A'_\varepsilon\}$ is a polynomial time approximation scheme to the PATH-MVSP for finding an optimal gapless schedule with a time complexity of*

$$O((1 + 2/\varepsilon)m(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)}).$$

Proof. Algorithm A' applied to the problem with scaled handling times h'_j , scaled edge weights $w'(v_j, v_{j+1})$ and K distinct release times is a $(1 + \varepsilon/2)$ -approximation algorithm for the problem with original handling times and original edge weights. From Lemma 1, algorithm A'_ε delivers a schedule which maximum completion time is at most $(1 + \varepsilon)C_{\max}(\pi_g^*)$.

Next we examine the time complexity of A'_ε . We define $U = \sum_{j=1}^{n-1} w'(v_j, v_{j+1}) + \sum_{j=1}^n h'_j$. Obviously, $U \leq Z/\delta = 4mn^2/\varepsilon$. Since $K \leq 1 + 2/\varepsilon$ (see (4)), the running time of algorithm A'_ε is $O((1 + 2/\varepsilon)m(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)})$. $\square$

5 An Approximation Scheme for General Schedules

Unfortunately, the optimal schedule π^* for a problem instance (G, r, h, w, m) is not always a gapless schedule, and hence $LB = (W + H)/m$ (see (6)) cannot be used as a lower bound on the minimum of the maximum completion time $C_{\max}^*$ attained by general schedules. Thus, a $(1 + \varepsilon)$ -approximation algorithm presented below conducts an efficient computation equivalent to taking into account all configurations of gaps on G which are possible to be incurred by the optimal schedule. A schedule consists of several gapless schedules for subinstances of G . For gaps $e'_1, e'_2, \dots, e'_{\lambda'} \in E$, each of maximal subpaths $G_1, G_2, \dots, G_{\lambda'+1}$ of G induced by non-gap edges will be served by a gapless schedule. To compute an approximate solution to a given instance (G, r, h, w, m) , we first consider a configuration of gaps on G that minimizes the maximum of the maximum completion times obtained by algorithm A'_ε on subpaths $G_1, G_2, \dots, G_{\lambda'+1}$. For a subinstance $(G(i, j), r, h, w, p)$, let $C_\varepsilon(G(i, j), r, h, w, p)$ denote the maximum completion time of a schedule computed by algorithm A'_ε , which is at most $(1 + \varepsilon)$ times of the optimal to the instance. For given jobs $i, j \in J$ ($i \leq j$), a number p of vehicles and an upper bound λ ($< p$) on the number of gaps, we denote by $Q(i, j, p, \lambda)$ the minimum of the maximum $C_\varepsilon(G_t, r, h, w, p_t)$ of instances (G_t, r, h, w, p_t) , $t = 1, 2, \dots, \lambda' + 1$ over all possible gaps $e'_1, e'_2, \dots, e'_{\lambda'} \in E$, where $\lambda' \leq \lambda$ and $\sum_{1 \leq t \leq \lambda' + 1} p_t = p$.

$$\begin{aligned} Q(i, j, p, 0) &= C_\varepsilon(G(i, j), r, h, w, p), \\ Q(1, j, p, \lambda) &= \min\{Q(1, j, p, 0), \\ &\quad \min_{1 \leq p' \leq p-1} \min_{1 \leq j' \leq j-1} \{\max\{Q(1, j', p-p', \lambda-1), Q(j'+1, j, p', 0)\}\}\}. \end{aligned}$$

Algorithm A_ε

Output: A schedule π_ε with $C_{max}(\pi_\varepsilon) \leq (1 + \varepsilon) \cdot C_{max}^*$

```

Step 1: for  $p = 1, 2, \dots, m$  do
    for  $i = 1, 2, \dots, n - p + 1$  do
        for  $j = i + p - 1, i + 1, \dots, n$  do
             $Q(i, j, p, 0) := C_\varepsilon(G(i, j), r, h, w, p)$ 
            by calling algorithm  $A'_\varepsilon$  for  $(G(i, j), r, h, w, p)$ 
        end; /* for */
    end; /* for */
end; /* for */

Step 2: for  $p = 2, 3, \dots, m$  do
    for  $\lambda = 1, 2, \dots, p - 1$  do
        for  $j = p, p + 1, \dots, n$  do
             $Q(1, j, p, \lambda) := \min \left[ Q(1, j, p, 0), \min_{1 \leq p' \leq p-1} \min_{1 \leq j' \leq j-1} \left\{ \right.$ 
 $\left. \max \{ Q(1, j', p - p', \lambda - 1), Q(j' + 1, j, p', 0) \} \right\} \right]$ 
        end; /* for */
    end; /* for */
end; /* for */

Step 3: Compute the configuration of gaps that achieves the  $Q(1, n, m, m - 1)$ 
For each subinstance  $G(i, j)$  incurred by the configuration, we
    compute a schedule  $\pi_{(i,j)}$  in Theorem 1;
Let  $\pi_\varepsilon$  be the schedule consisting of these schedules  $\pi_{(i,j)}$ .

```

Step 1 calls algorithm A'_ε $O(mn^2)$ times, and hence it requires $O((1 + 2/\varepsilon)m^2n^2(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)})$ time from Theorem 1. The value of $Q(1, j, p, \lambda)$ for each j, p and λ at Step 2 can be computed in $O(mn)$ time, and thus this step requires $O(m^3n^2)$ time. In Step 3, we can trace the configuration of gaps

that achieves the $Q(1, n, m, m - 1)$ in the same time complexity (by storing the indices that attain the minimum in the formula in Step 2). Therefore, the running time of algorithm A_ε is $O((1 + 2/\varepsilon)m^2n^2(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)})$.

Theorem 2. *The family of algorithms $\{A_\varepsilon\}$ is a polynomial time approximation scheme to the PATH-MVSP for finding an optimal schedule with a time complexity of*

$$O((1 + 2/\varepsilon)m^2n^2(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)}).$$

Proof. In the above, we have seen that algorithm A_ε can obtain $Q(1, n, m, m - 1)$ and a configuration of gaps that achieves $Q(1, n, m, m - 1)$ in $O((1 + 2/\varepsilon)m^2n^2(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)})$ time. For each subpath $G(i, j)$ incurred by the configuration, a schedule $\pi_{(i, j)}$ in Theorem 1 can be computed in at most $O((1 + 2/\varepsilon)m(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)})$ time. By Theorem 1 $\pi_{(i, j)}$ is a $(1 + \varepsilon)$ -approximation to the subinstance, satisfying $C_{\max}(\pi_{(i, j)}) \leq Q(1, n, m, m - 1)$. Therefore, the schedule which consists of these schedules $\pi_{(i, j)}$ is a $(1 + \varepsilon)$ -approximation to the original problem $(G(1, n), r, h, w, m)$. $\square$

Before closing this section, we remark that our approach to the PATH-MVSP can be applied to the MVSP in a tree G . Given release times and handling times on jobs (each located at a vertex in G) and symmetric weights on edges, the problem asks to find an optimal schedule of m vehicles to process all the n jobs. Let ℓ be the number of leaves in G . For a set J' of jobs with the same release time, we see that an optimal schedule to process J' is given by visiting all the jobs in J' along the minimal subtree $T_{J'}$ containing J' (hence the completion time is bounded by $2Z$). Thus such a schedule can be reconstructed in $O(n)$ time from the first and last jobs to be processed in J' and the set of leaves of $T_{J'}$. To obtain our dynamic programming in Section 3 in the tree case, we need to have a table X of $(\ell + 1)Km$ -tuple which consists of the completion time C and the set of leaves in a subtree, where $K (\leq 1 + 2/\varepsilon)$ is the number of distinct rounded release times. An update of each table takes $O(Kmn(2Zn^\ell)^{Km})$ time. Thus the dynamic programming computes an optimal solution to the problem with integer handling times and edge weights in $O(Km(n^2 + 2^K)(2Zn^\ell)^{km})$ time. From a similar discussion in Section 4, by scaling handling times and edge weights by $\delta = (2Z\varepsilon)/8mn^2$, we obtain a $(1 + \varepsilon)$ -approximation algorithm with time complexity $O((1 + 2/\varepsilon)m(n^2 + 2^{1+2/\varepsilon})(8mn^{2+\ell}/\varepsilon)^{m(1+2/\varepsilon)})$ to the problem of finding an optimal gapless schedule. To find the best configuration of gaps in a tree G , we need to check at most $\sum_{p=1}^m \binom{n-1}{p-1} \binom{m-1}{p-1} = O((nm)^m)$ cases. Therefore, we obtain a $(1 + \varepsilon)$ -approximation algorithm with time complexity $O((1 + 2/\varepsilon)n^m m^{m+1}(n^2 + 2^{1+2/\varepsilon})(8mn^{2+\ell}/\varepsilon)^{m(1+2/\varepsilon)})$ to the MVSP in trees, which is polynomial if the numbers of vehicles and leaves in G are fixed.

6 Concluding Remarks

In this paper, we discussed a scheduling problem of vehicles on a path with release and handling times, PATH-MVSP. The problem asks to find an optimal schedule of m vehicles serving n jobs that minimizes the maximum completion

time of all the jobs. The PATH-MVSP is NP-hard for any fixed $m \geq 2$. In this paper, we presented a polynomial time approximation scheme to the problem, i.e., a family of algorithms $\{A_\varepsilon\}$. For any $\varepsilon > 0$, algorithm A_ε delivers a schedule with the maximum completion time at most $(1 + \varepsilon)$ times the optimal in $O((1 + 2/\varepsilon)m^2n^2(n + 2^{1+2/\varepsilon})(4mn^4/\varepsilon)^{m(1+2/\varepsilon)})$ time. Our approximation scheme $\{A_\varepsilon\}$ is based on approximation of the problem by rounding given release times, and the fact that any schedule consists of some gapless schedules on subpaths of a given path. We also observed that our algorithm can be extended to the case where G is a tree, showing the MVSP in trees admits a polynomial time approximation scheme as long as the numbers of vehicles and leaves in G are constant. In the future research, different polynomial time approximation schemes with less space complexity may be considered.

References

1. Averbakh, I. and Berman, O.: Sales-delivery man problems on treelike networks, *Networks* **25** (1995) 45-58. [36](#), [37](#)
2. Garey, M. R. and Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco (1979). [37](#)
3. Hall, L. A.: A polynomial approximation scheme for a constrained flow-shop scheduling problem, *Mathematics of Operations Research* **19** (1994) 68-85. [37](#), [40](#)
4. Hall, L. A. and Shmoys, D. B.: Approximation schemes for constrained scheduling problems, *Proceedings of IEEE 30th Annual Symposium on Foundations of Computer Science* (1989) 134-139. [37](#)
5. Karuno, Y., Nagamochi, H. and Ibaraki, T.: Vehicle scheduling on a tree to minimize maximum lateness, *Journal of the Operations Research Society of Japan* **39** (1996) 345-355. [37](#)
6. Karuno, Y., Nagamochi, H. and Ibaraki, T.: Vehicle scheduling on a tree with release and handling times, *Annals of Operations Research* **69** (1997) 193-207. [37](#)
7. Karuno, Y., Nagamochi, H. and Ibaraki, T.: A 1.5-approximation for single-vehicle scheduling problem on a line with release and handling times, *Proceedings of ISCIE/ASME 1998 Japan-U. S. A. Symposium on Flexible Automation* **3** (1998) 1363-1366. [37](#)
8. Karuno, Y. and Nagamochi, H.: A 2-approximation algorithm for the multi-vehicle scheduling problem on a path with release and handling times, *Proceedings of 9th Annual European Symposium on Algorithms, ESA2001* (Aarhus, Denmark, August 28-31, 2001), *Lecture Notes in Computer Science*, Springer-Verlag (2001) in press. [37](#)
9. Kovalyov, M. Y. and Werner, F.: A polynomial approximation scheme for problem $F2/r_j/C_{max}$, *Operations Research Letters* **20** (1997) 75-79. [37](#), [40](#)
10. Nagamochi, H., Mochizuki, K. and Ibaraki, T.: Complexity of the single vehicle scheduling problem on graphs, *Information Systems and Operations Research* **35** (1997) 256-276. [37](#)
11. Nagamochi, H., Mochizuki, K. and Ibaraki, T.: Solving the single-vehicle scheduling problems for all home locations under depth-first routing on a tree, *IEICE Transactions: Fundamentals* **E84-A** (2001) 1135-1143. [37](#)
12. Psaraftis, H., Solomon, M., Magnanti, T. and Kim, T.: Routing and scheduling on a shoreline with release times, *Management Science* **36** (1990) 212-223. [37](#)

13. Tsitsiklis, J. N.: Special cases of traveling salesman and repairman problems with time windows, *Networks* **22** (1992) 263-282. [37](#)

Semi-normal Schedulings: Improvement on Goemans' Algorithm^{*}

Jianer Chen^{1,2} and Jingui Huang²

¹ Department of Computer Science, Texas A&M University
College Station, TX 77843-3112, USA
chen@cs.tamu.edu

² College of Information Engineering, Central-South University
ChangSha, Hunan 410083, P. R. China hjg@hunnu.edu.cn

Abstract. Theoretical study of the multiprocessor job scheduling problem has made significant progress recently, which, however, seems not yet to imply practical algorithms. This paper offers new observations and introduces new techniques for the multiprocessor job scheduling problem $P_3|fix|C_{\max}$. The concept of *semi-normal schedulings* is introduced and a very simple linear time algorithm for constructing semi-normal schedulings is developed. Thorough analysis is provided in the study of semi-normal schedulings, which enables us to conclude that the proposed algorithm is an approximation algorithm of ratio 9/8 for the $P_3|fix|C_{\max}$ problem. This improves the previous best (practical) ratio 7/6 by Goemans. Our techniques are also useful for multiprocessor job scheduling problems on systems with more than three processors.

1 Introduction

An assumption made in classical scheduling theory is that each job is executed by a single processor. With the advances in parallel processing, this assumption may no longer be valid for job systems. For example, in semiconductor circuit design workforce planning, a design project is processed by a group of people. The project contains n jobs, and each job is handled by a specific subgroup of the people working simultaneously on the job. Each person may belong to several different subgroups but can work on at most one job at a time. Many other applications of this kind of *multiprocessor job scheduling* model have been discovered [5,15,16]. A typical multiprocessor job scheduling problem is the $P_k|fix|C_{\max}$ problem, in which the system has k processors, and each job is assigned to a fixed processor set. The objective is to schedule a given set of jobs so that the makespan (i.e., the system final finishing time) is minimized.

Feasibility and approximability of $P_k|fix|C_{\max}$, in particular, of $P_3|fix|C_{\max}$, have been studied extensively. Hoogeveen et al. [13] showed that $P_3|fix|C_{\max}$ is NP-hard in the strong sense thus it does not have a fully polynomial time approximation scheme unless $P = NP$ (see also [2,3]). Blazewicz et al. [2] developed

^{*} This work is supported in part by USA NSF Grant CCR-0000206, China NSFC Grant No.69928201, and the Changjiang Scholar Reward Project.

a polynomial time approximation algorithm of ratio $4/3$ for $P_3|fix|C_{\max}$, which was improved later by Dell'Olmo et al. [8] to ratio $5/4$. Goemans [10] further improved the results with a polynomial time approximation algorithm of ratio $7/6$ for $P_3|fix|C_{\max}$. More recently, Amoura et al. [1] developed a polynomial time approximation scheme for $P_k|fix|C_{\max}$ for every fixed integer k . Polynomial time approximation schemes for a generalized version of $P_k|fix|C_{\max}$ have also been developed recently [7, 14].

The approximation schemes [1, 7, 14] for $P_k|fix|C_{\max}$ are of great theoretical significance. However, most of these algorithms are based on extensive enumerations of certain kinds of schedulings together with either dynamic programming or linear programming techniques. This makes the algorithms practically slow and difficult to implement. Chen and Miranda [7] have called for more practically efficient and easy-implementable approximation algorithms for the multiprocessor job scheduling problem for systems with small number of processors.

The current paper is a response to the call by [7]. We study the $P_3|fix|C_{\max}$ problem. All practical algorithms for $P_3|fix|C_{\max}$ are more or less based on the concept of *normal schedulings* in which all jobs of the same mode must be executed consecutively. It is known [8] that the best normal scheduling has makespan at most $5/4$ of the optimal makespan. Goemans [10] generalized the concept of normal scheduling by allowing splitting one 1-processor job set of the same mode, and showed that the best schedulings under this generalization have makespan at most $7/6$ of the optimal makespan. This is currently the best practical algorithm for the $P_3|fix|C_{\max}$ problem.

We further generalize Goemans' technique by allowing all possible splittings of 1-processor job sets. In other words we study schedulings in which only 2-processor jobs of the same mode are required to be executed consecutively. We call such schedulings "semi-normal" schedulings. We first show that the problem of constructing optimal semi-normal schedulings can be reduced to the classical partition problem, and develop a very simple linear time algorithm that constructs a nearly optimal semi-normal scheduling. Thorough analysis and detailed discussion are provided in the study of semi-normal schedulings, which enables us to conclude that the proposed algorithm is an approximation algorithm of ratio $9/8$ for the $P_3|fix|C_{\max}$ problem. This improves the previous best ratio $7/6$ by Goemans [10]. Our techniques are also useful for multiprocessor job scheduling problems on systems with more than three processors.

2 Basic Definitions and Simple Facts

Suppose that the system has three processors p_1 , p_2 , and p_3 . An instance of the $P_3|fix|C_{\max}$ problem is a set of jobs: $J = \{j_1, j_2, \dots, j_n\}$, where each job j_i is described by a pair $j_i = (Q_i, \tau_i)$, Q_i is a subset of $\{p_1, p_2, p_3\}$ indicating the processor set required to execute the job j_i , and τ_i is the parallel processing time of the job j_i executed on the processor set Q_i . The processor set Q_i is called the *processing mode* (or simply *mode*) of the job j_i .

A *scheduling* S of the job set J is an assignment of each job j_i in J with a starting time to be executed on the processor set Q_i in the 3-processor system such that no processor is used for execution of more than one job at any moment. Preemption is not allowed. The *makespan* of the scheduling S is the latest finish time of a job in J under the scheduling S . An *optimal scheduling* of the job set J is a scheduling whose makespan is the minimum over all schedulings of J . The makespan of an optimal scheduling for the job set J is denoted by $Opt(J)$. An *approximation algorithm* A for $P_3|fix|C_{\max}$ is an algorithm that for each given instance J of $P_3|fix|C_{\max}$ constructs a scheduling for J . The *approximation ratio* of the algorithm A is (bounded by) r if for any instance J , the makespan of the scheduling for J constructed by A is bounded by $r \cdot Opt(J)$.

We consider approximation algorithms for $P_3|fix|C_{\max}$. Since we can always schedule jobs of mode $\{p_1, p_2, p_3\}$ before other jobs without increasing the approximation ratio of an algorithm for $P_3|fix|C_{\max}$, we can assume, without loss of generality, that an instance of $P_3|fix|C_{\max}$ contains no jobs of mode $\{p_1, p_2, p_3\}$. Thus, there are at most $2^3 - 2 = 6$ processing modes for jobs in an instance (the mode $\emptyset$ is also naturally excluded). A job is a *1-job* if its mode consists of a single processor, and a *2-job* if its mode consists of two processors.

Group the jobs in J into six subsets in terms of their modes: $J = \{J_1, J_2, J_3, J_{12}, J_{13}, J_{23}\}$, here for each index $y \in \{1, 2, 3\}$, J_y is the set of all 1-jobs of mode $\{p_y\}$ in J , and for each index pair $\{y, z\}$ in $\{1, 2, 3\}$, J_{yz} is the set of all 2-jobs of mode $\{p_y, p_z\}$ in J . For a job set J' , we will denote by $|J'|$ the sum of the total processing times required by the jobs in J' .

Definition 1. Let $T_1 = |J_1| + |J_{12}| + |J_{13}|$, $T_2 = |J_2| + |J_{12}| + |J_{23}|$, and $T_3 = |J_3| + |J_{13}| + |J_{23}|$. Define $K_J = \max\{T_1, T_2, T_3, |J_{12}| + |J_{13}| + |J_{23}|\}$.

Lemma 1. The value K_J is a lower bound for $Opt(J)$, i.e., $Opt(J) \geq K_J$.

Lemma 2. If $|J_{xz}| \leq K_J/8$ for an index pair $\{x, z\}$, or $|J_y| \leq |J_{xz}| + K_J/8$ for a permutation $\{x, y, z\}$ of the indices $\{1, 2, 3\}$, then at least one of the schedulings in Figure 1 has makespan bounded by $(9/8)Opt(J)$.

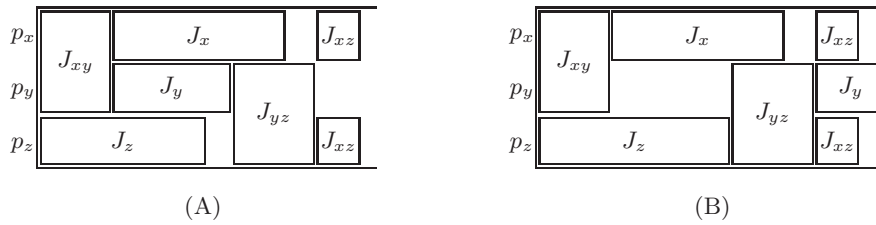


Fig. 1. (A) Scheduling when $|J_{xz}|$ is small; (B) Scheduling when $|J_y|$ is not too large

Proof. Suppose $|J_{xz}| \leq K_J/8$. Consider the scheduling in Figure 1(A). It is easy to see that before the job set J_{xz} starts, there is at least one processor, as p_y in Figure 1(A), which has never had idle time. Therefore, the makespan of the scheduling is not larger than $\max\{T_1, T_2, T_3\} + |J_{xz}|$, which is bounded by $K_J + K_J/8 \leq (9/8)Opt(J)$.

Now suppose $|J_y| \leq |J_{xz}| + K_J/8$. Consider the scheduling in Figure 1(B). It is easy to see that the job subset J_{xz} finishes at time no later than K_J . Now if we let job set J_y start at the time where J_{xz} starts, then since $|J_y| \leq |J_{xz}| + K_J/8$, the job set J_y will finish at time no later than $K_J + K_J/8$, which is bounded by $(9/8)Opt(J)$. $\square$

Therefore, we will concentrate on the following kind of instances.

Definition 2. An instance J of $P_3|fix|C_{\max}$ is *nontrivial* if for every permutation $\{x, y, z\}$ of $\{1, 2, 3\}$, $|J_y| > |J_{xz}| + K_J/8$, and for every index pair $\{x, z\} \subseteq \{1, 2, 3\}$, $|J_{xz}| > K_J/8$.

In particular, if J is nontrivial, then $|J_y| > K_J/4$ for all indices $y \in \{1, 2, 3\}$.

3 Semi-normal Scheduling

Following [8], we call a scheduling for an instance J of $P_3|fix|C_{\max}$ a *normal scheduling* if all jobs of the same mode in J are executed consecutively. It is known [8] that the best normal scheduling for J has makespan bounded by $(5/4)Opt(J)$. We propose a natural extension of the normal schedulings.

Definition 3. A scheduling for an instance J of the $P_3|fix|C_{\max}$ problem is *semi-normal* if for each index pair $\{y, z\} \subseteq \{1, 2, 3\}$, all 2-jobs of mode $\{p_y, p_z\}$ are executed consecutively.

We discuss how a semi-normal scheduling is constructed based on partitions of 1-jobs. Suppose we have somehow partitioned the 1-job set J_y into J'_y and J''_y , and the 1-job set J_z into J'_z and J''_z . Hence the job set J is now partitioned into: $J = \{J_x, J'_y, J''_y, J'_z, J''_z, J_{xy}, J_{xz}, J_{yz}\}$. A scheduling algorithm **Scheduler**(y, z) based on this partition is given in Figure 2, where for job set J_x , denote by s_x and f_x the starting and finishing times of J_x , respectively, under the scheduling. Similarly we define $s'_y, f'_y, s''_y, f''_y, s'_z, f'_z, s''_z, f''_z, s_{xy}, f_{xy}, s_{xz}, f_{xz}, s_{yz},$ and f_{yz} . An intuitive illustration of the scheduling is also given in Figure 2.

Lemma 3. The makespan of the scheduling S_{yz} in algorithm **Scheduler**(y, z) is equal to $\max\{Opt(J), |J'_z| + |J_{yz}| + |J'_y|, |J_{xy}| + |J''_y| + |J_{yz}| + |J''_z| + |J_{xz}|\}$.

Proof. Let t_0 be the makespan of the scheduling S_{yz} . We have $t_0 = \max\{f'_y, f_{xz}\}$.

Suppose $t_0 = f'_y$. If $f''_y \geq f'_z$, then $t_0 = |J_{xy}| + |J''_y| + |J_{yz}| + |J'_y| = T_y \leq Opt(J)$, while if $f''_y \leq f'_z$, then $t_0 = |J'_z| + |J_{yz}| + |J'_y|$.

Now suppose $t_0 = f_{xz}$. If $f_x \geq f''_z$ then $t_0 = |J_{xy}| + |J_x| + |J_{xz}| = T_x \leq Opt(J)$. For $f_x \leq f''_z$, if $f''_y \geq f'_z$ then $t_0 = |J_{xy}| + |J''_y| + |J_{yz}| + |J''_z| + |J_{xz}|$; while if $f''_y \leq f'_z$, then $t_0 = |J'_z| + |J_{yz}| + |J''_y| + |J_{xz}| = T_z \leq Opt(J)$.

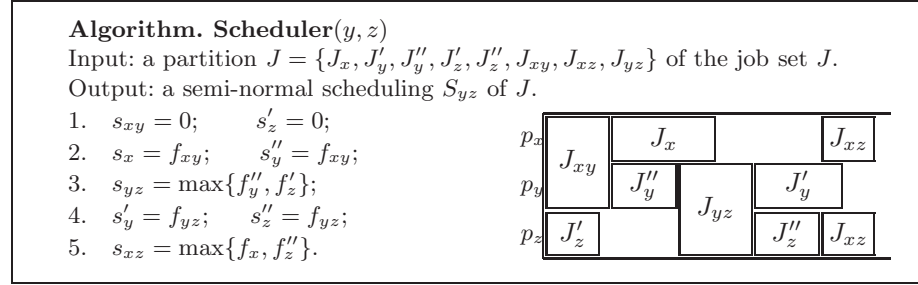


Fig. 2. Scheduling S_{yz} based on partitions of 1-job sets J_y and J_z

Thus, $t_0 \leq \max\{Opt(J), |J'_z| + |J_{yz}| + |J'_y|, |J_{xy}| + |J''_y| + |J_{yz}| + |J''_z| + |J_{xz}|\}$.

Consider the other direction. We obviously have $t_0 \geq Opt(J)$. Moreover, note that in the scheduling S_{yz} , $\{J'_z, J_{yz}, J'_y\}$ and $\{J_{xy}, J''_y, J_{yz}, J''_z, J_{xz}\}$ are the ordered sequences of job sets in which a job set will not start before the job set before it finishes. This implies directly that the makespan t_0 of S_{yz} is not smaller than $|J'_z| + |J_{yz}| + |J'_y|$ and $|J_{xy}| + |J''_y| + |J_{yz}| + |J''_z| + |J_{xz}|$. $\square$

By Lemma 3, to reduce the makespan of S_{yz} , we should construct a partition of J_y and J_z to minimize the value $\max\{|J'_y| + |J'_z|, |J''_y| + |J''_z| + (|J_{xy}| + |J_{xz}|)\}$.

Definition 4. Let J be an instance for $P_3|fix|C_{\max}$. For each $\{y, z\} \subseteq \{1, 2, 3\}$, let L_{yz} be the list that consists of the item $\tau_{xy+xz} = |J_{xy}| + |J_{xz}|$ and the processing times of the jobs in $J_y \cup J_z$. Let L_{yz}^D be the sublist of L_{yz} consisting of τ_{xy+xz} and the items larger than $K_J/4$ in L_{yz} .

Therefore, we should partition the list L_{yz} into two boxes B'_{yz} and B''_{yz} so that the box size difference is minimized. To simplify the discussion, we will say “a job j is in a list L (e.g., L_{yz} or L_{yz}^D)” if the processing time of the job j is in the list L . This should not introduce any confusion.

Remark 3.1. For any J , each of the sets J_y and J_z has at most three jobs in the list L_{yz}^D — otherwise we would have either $|J_y| > K_J$ or $|J_z| > K_J$.

Remark 3.2. If the instance J is nontrivial, then $|J_{yz}| > K_J/8$ for all index pairs $\{y, z\}$. Thus, each job subset J_y can have at most two jobs in the list L_{yz}^D . In fact, if J_y has three jobs in L_{yz}^D , then $|J_y| > (3/4)K_J$. This, together with $\tau_{xy+yz} = |J_{xy}| + |J_{yz}| > K_J/4$, would give $T_y = |J_y| + |J_{xy}| + |J_{yz}| > K_J$.

We apply a variation of the classical *Graham List Scheduling* algorithm [11], to partition the list L_{yz} into two boxes B'_{yz} and B''_{yz} , as shown in Figure 3.

Remark 3.3. By Remark 3.1, the list L_{yz}^D consists of at most seven items. Thus, step 1 of **Partition**(y, z) takes constant time, and the algorithm **Partition**(y, z) runs in linear time.

Definition 5. Let B'_{yz} and B''_{yz} be the boxes constructed by the algorithm **Partition**(y, z). The item that is last added to the larger box among B'_{yz} and B''_{yz}

Algorithm. Partition(y, z)Input: the list L_{yz} Output: a partition (B'_{yz}, B''_{yz}) of the list L_{yz}

1. start with a partition of the list L_{yz}^D into the two boxes B'_{yz} and B''_{yz} so that the item $\tau_{xy+xz} = |J_{xy}| + |J_{xz}|$ is in the box B''_{yz} and the difference of the box sizes is minimized;
2. **for** each item τ in $L_{yz} - L_{yz}^D$ **do**
 if $|B'_{yz}| \leq |B''_{yz}|$ **then** add τ to B'_{yz} **else** add τ to B''_{yz} ;
3. $J'_y = J_y \cap B'_{yz}$; $J''_y = J_y \cap B''_{yz}$; $J'_z = J_z \cap B'_{yz}$; $J''_z = J_z \cap B''_{yz}$.

Fig. 3. Partition of the job sets J_y and J_z

is called the *covering item*, and the difference $||B'_{yz}| - |B''_{yz}||$ is called the *box difference*, denoted by d_{yz} .

Remark 3.4. Steps 1 and 2 of the algorithm **Partition**(y, z) guarantees that the covering item is not smaller than the box difference d_{yz} .

Lemma 4. Let (B'_{yz}, B''_{yz}) be the partition of the list L_{yz} from **Partition**(y, z). Then the makespan of the semi-normal scheduling S_{yz} based on this partition by **Scheduler**(y, z) is equal to $\max\{Opt(J), |B'_{yz}| + |J_{yz}|, |B''_{yz}| + |J_{yz}|\}$, which is bounded by $Opt(J) + d_{yz}/2$.

Proof. The first conclusion is obvious. Moreover, we have

$$\begin{aligned}
& \max\{|B'_{yz}| + |J_{yz}|, |B''_{yz}| + |J_{yz}|\} = \max\{|B'_{yz}|, |B''_{yz}|\} + |J_{yz}| \\
&= (|B'_{yz}| + |B''_{yz}|)/2 + ||B'_{yz}| - |B''_{yz}||/2 + |J_{yz}| \\
&\leq (|J'_y| + |J'_z| + |J''_y| + |J''_z| + |J_{xy}| + |J_{xz}|)/2 + d_{yz}/2 + |J_{yz}| \\
&= (|J_y| + |J_z| + |J_{xy}| + |J_{xz}| + 2|J_{yz}|)/2 + d_{yz}/2 \\
&= (T_y + T_z)/2 + d_{yz}/2 \\
&\leq Opt(J) + d_{yz}/2 \quad \square
\end{aligned}$$

Corollary 1. If the covering item is not in the list L_{yz}^D , then the makespan of the scheduling S_{yz} constructed by **Scheduler**(y, z) is bounded by $(9/8)Opt(J)$.

Proof. By the definition of the list L_{yz}^D , if the covering item is not in L_{yz}^D , then it is bounded by $K_J/4$. By Remark 3.4, the box difference d_{yz} is also bounded by $K_J/4$. Now since $K_J \leq Opt(J)$, the corollary follows directly from Lemma 4. $\square$

Our main algorithm, **SemiNormal**, for the $P_3|fix|C_{\max}$ problem constructs a semi-normal scheduling for a given instance based on the algorithms **Scheduler** and **Partition**. The algorithm is given in Figure 4. The algorithm obviously runs in linear time. The rest of this paper is to show that the approximation ratio of the algorithm **SemiNormal** is bounded by 9/8.

Algorithm. SemiNormalInput: an instance J of the $P_3|fix|C_{\max}$ problem.Output: a semi-normal scheduling for J .

1. **for** each index pair $\{y, z\} \subseteq \{1, 2, 3\}$ **do**
 - call **Partition**(y, z) to partition the list L_{yz} into (B'_{yz}, B''_{yz}) ;
 - call **Scheduler**(y, z) to construct a scheduling S_{yz} for J on the partition;
2. output the scheduling of the minimum makespan constructed in step 1.

Fig. 4. Main algorithm for the $P_3|fix|C_{\max}$ problem

4 On Job Sets with No Small 1-Jobs

In this section, we study the approximation ratio of our algorithm **SemiNormal** on instances of $P_3|fix|C_{\max}$ with a special structure.

Definition 6. An instance J is *with no small 1-jobs* if all 1-jobs in J have processing time $> K_J/4$.

A scheduling S_J of makespan t_0 for an instance J of $P_3|fix|C_{\max}$ can be naturally divided into disjoint *job blocks*, as follows. A *2-job block* of mode $\{p_x, p_y\}$ consists of the two processors p_x and p_y and a maximal time interval $[t, t'] \subseteq [0, t_0]$ in S_J such that from time t to t' , the processors p_x and p_y are executing jobs of mode $\{p_x, p_y\}$, and a *1-job block* of mode $\{p_x\}$ consists of the processor p_x and a maximal time interval $[t, t'] \subseteq [0, t_0]$ such that at any time during $[t, t']$, the processor p_x is either executing a job of mode $\{p_x\}$ or is idle. Note that in a 2-job block, the two related processors remain busy during the entire time interval, while in a 1-job block, the related processor may become idle for a part or the entire time interval. Since no two 2-jobs of different modes can be executed in parallel, no two 2-job blocks in S_J can have their time intervals overlap. Thus, the 2-job blocks in S_J can be given in a list $H(S_J)$, ordered increasingly by their starting execution times. The list $H(S_J)$ will be called the *2-job block list* for S_J . We will use the subscript to indicate the mode of a block. For example, F_{yz} will be a 2-job block of mode $\{p_y, p_z\}$.

Lemma 5. *If two consecutive 2-job blocks in the 2-job block list $H(S_J)$ have the same mode, then they can be merged into a single job block without increasing the makespan of the scheduling.*

Lemma 6. *Let S_J be a scheduling for a nontrivial instance J with no small 1-jobs. If there are more than two 2-job blocks of mode $\{p_x, p_y\}$ in the 2-job block list $H(S_J)$, then we can merge two 2-job blocks of mode $\{p_x, p_y\}$ into a single one without increasing the makespan.*

Lemma 7. *Let S_J be a scheduling for a nontrivial instance J with no small 1-jobs. If the 2-job block list $H(S_J)$ contains three consecutive 2-job blocks of the form $\{F_{xy}, F_{yz}, F'_{xy}\}$, then two 2-job blocks in S_J can be merged into one without increasing the makespan.*

Lemma 8. *Let J be an instance with no small 1-jobs. Then the scheduling constructed by the algorithm **SemiNormal** for J has the minimum makespan over all semi-normal schedulings for J .*

Proof. Let S_J be a semi-normal scheduling for J such that the makespan t_0 of S_J is the minimum over all semi-normal schedulings for J . Let $H(S_J) = \{F_{xy}, F_{yz}, F_{xz}\}$ be the 2-job block list for S_J . Without loss of generality, we assume the job block F_{xy} starts at time 0 (otherwise, we simply swap F_{xy} with the 1-job blocks of mode $\{p_x\}$ and of mode $\{p_y\}$ before F_{xy} in S_J). Similarly, we assume the job block F_{xz} ends at time t_0 .

Under these assumptions, there are exactly one 1-job block F_x of mode $\{p_x\}$, at most two 1-job blocks F'_y and F''_y of mode $\{p_y\}$, and two 1-job blocks F'_z and F''_z of mode $\{p_z\}$, as illustrated in Figure 2, where each 1-job set is extended to include the neighboring idle time in the processor to form a 1-job block. By Lemmas 3 and 4, the makespan of S_J is equal to $t_0 = \max\{Opt(J), |\bar{B}'_{yz}| + |J_{yz}|, |\bar{B}''_{yz}| + |J_{yz}|\}$, where $(\bar{B}'_{yz}, \bar{B}''_{yz})$ is a partition of the list L_{yz} .

When we apply the algorithm **SemiNormal** to the instance J , in step 1 on the same index pair $\{y, z\}$, the algorithm **SemiNormal** calls the algorithm **Partition**(y, z) to partition the list L_{yz} into (B'_{yz}, B''_{yz}) , then constructs a scheduling S_{yz} based on this partition. By Lemma 4, the makespan of the scheduling S_{yz} is equal to $\max\{Opt(J), |B'_{yz}| + |J_{yz}|, |B''_{yz}| + |J_{yz}|\}$. Since J is an instance with no small 1-jobs, the lists L_{yz} and L_{yz}^D are identical. Thus, the partition (B'_{yz}, B''_{yz}) of the list L_{yz} constructed by **Partition**(y, z) minimizes the box difference $||B'_{yz}| - |B''_{yz}||$ (see step 1 of the algorithm **Partition**(y, z) in Figure 3), that is, $\max\{|B'_{yz}|, |B''_{yz}|\} \leq \max\{|\bar{B}'_{yz}|, |\bar{B}''_{yz}|\}$. Thus, the makespan of the scheduling S_{yz} is not larger than t_0 . The lemma follows since the algorithm **SemiNormal** picks the best semi-normal scheduling over all S_{yz} . $\square$

Corollary 2. *Let J be an instance with no small 1-jobs. If J is not nontrivial, then the semi-normal scheduling constructed by **SemiNormal** for J has makespan bounded by $(9/8)Opt(J)$.*

Thus, we only need to concentrate on nontrivial instances.

Lemma 9. *Let J be a nontrivial instance with no small 1-jobs in which the job subset J_z has two jobs. Let d_{xz} and d_{yz} be the box differences of the partitions of the lists L_{xz} and L_{yz} , respectively, constructed by the algorithm **Partition**. Then we have either $d_{xz} \leq \max\{(T_x + T_z)/5, K_J/4\}$ or $d_{yz} \leq \max\{(T_y + T_z)/5, K_J/4\}$.*

Lemma 10. *Let J be an instance with no small 1-jobs. Then the makespan of the semi-normal scheduling for J constructed by the algorithm **SemiNormal** is bounded by $(9/8)Opt(J)$.*

Proof. By Corollary 2, we can assume that the instance J is nontrivial. By Lemma 8, it suffices to show that in any case there is always a semi-normal scheduling for the instance J whose makespan is bounded by $(9/8)Opt(J)$.

Let S_{opt} be an optimal scheduling of makespan $Opt(J)$ for the job set J . Let $H(S_{opt})$ be the 2-job block list for S_{opt} . If S_{opt} is semi-normal, then we are done.

So suppose S_{opt} is not semi-normal. By Lemmas 5, 6, and 7, we can assume that (1) no two consecutive 2-job blocks in $H(S_{opt})$ are of the same mode; (2) each 2-processor mode has at most 2 job blocks in $H(S_{opt})$; and (3) in $H(S_{opt})$ there are no three consecutive 2-job blocks of the form $(F_{xy}, F_{yz}, F'_{xy})$. Under all these assumptions, the list $H(S_{opt})$ must be of one of the following three forms: $(F_{xy}, F_{yz}, F_{xz}, F'_{xy})$, $(F_{xy}, F_{yz}, F_{xz}, F'_{xy}, F'_{yz})$, and $(F_{xy}, F_{yz}, F_{xz}, F'_{xy}, F'_{yz}, F'_{xz})$.

In case $H(S_{opt}) = (F_{xy}, F_{yz}, F_{xz}, F'_{xy}, F'_{yz}, F'_{xz})$, we can, similar as before, assume that F_{xy} starts at time 0 and that F'_{xz} ends at time $Opt(J)$. The scheduling S_{opt} should have the configuration shown in Figure 5(A). The situations in which $H(S_{opt})$ has less than six 2-job blocks can also be represented by this configuration by properly setting certain blocks to have length 0.

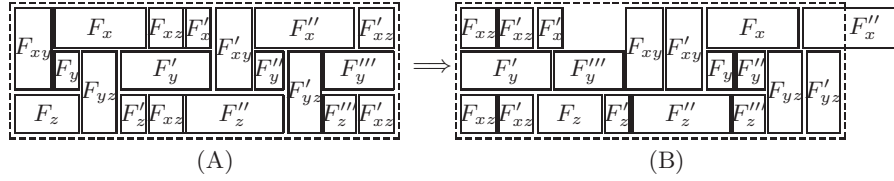


Fig. 5. Rearrangement for the structure $\{F_{xy}, F_{yz}, F_{xz}, F'_{xy}, F'_{yz}, F'_{xz}\}$

If one of F_z and F'_z is empty, then we can easily merge F_{xy} and F'_{xy} into a single one without increasing the makespan. Therefore, we can assume that F_z and F'_z are not empty. Since J is a nontrivial instance with no small 1-jobs, J_z has at most two jobs. Thus, the 1-job blocks F'_z and F''_z must be empty.

Case 1. $|F_{xy}| + |F_x| + |F'_y| \leq |F_z| + |F_{yz}| + |F'_{xz}| + Opt(J)/8$.

In this case, we rearrange the job blocks as shown in Figure 5(B). In this rearrangement, processors p_y and p_z still halt at time $Opt(J)$. To see this, note that after pushing F_{xz} and F'_{xz} to the beginning and F_{yz} and F'_{yz} to the end of the dashed area, the space left in processor p_z is just enough for F_z , F'_z , F''_z , and F'''_z . Moreover, from Figure 5(A), we have $|F'_y| \geq |F_{xz}| + |F'_x|$ and $|F'_y| \geq |F'_{xz}|$. Therefore, there is no “gap” between F'_y and F_{xy} in the rearrangement in Figure 5(B). It follows immediately that there is just enough space left in processor p_y for F_y and F'_y . Thus, the makespan of this rearrangement is

$$|F'_y| + |F'_y| + |F_{xy}| + |F'_{xy}| + |F_x| + |F'_x|$$

which, under the assumption of this case, is not larger than

$$|F'_y| + |F_z| + |F_{yz}| + |F'_{xy}| + |F'_{xz}| + |F'_x| + Opt(J)/8$$

Since $|F'_y| + |F_z| + |F_{yz}| + |F'_{xy}| + |F'_{xz}| + |F'_x|$ is just the makespan $Opt(J)$ of S_{opt} in Figure 5(A), the makespan of the rearrangement in Figure 5(B) is bounded by $(9/8)Opt(J)$. Moreover, the resulting scheduling is semi-normal.

Case 2. $|F_{xy}| + |F_x| + |F'_y| > |F_z| + |F_{yz}| + |F'_{xz}| + Opt(J)/8$.

According to Figure 5(A) and the assumption of the case, we have

$$\begin{aligned}
 \text{Opt}(J) &= |F_{xy}| + |F_x| + |F_{xz}| + |F_z''| + |F_{yz}'| + |F_y'''| \\
 &> |F_{xz}| + |F_z''| + |F_{yz}'| + |F_z| + |F_{yz}| + |F_{xz}'| + \text{Opt}(J)/8 \\
 &\geq T_z + \text{Opt}(J)/8
 \end{aligned}$$

The second inequality is because the processor p_z is entirely idle in the 1-job blocks F_z' and F_z''' . Thus $T_z < (7/8)\text{Opt}(J)$.

Since both F_z' and F_z'' are nonempty, and J is with no small 1-jobs, the job subset J_z has exactly 2 jobs. By Lemma 9, the box differences d_{xz} and d_{yz} of the partitions of the lists L_{xz} and L_{yz} constructed by the algorithm **Partition** satisfy either $d_{xz} \leq \max\{(T_x + T_z)/5, K_J/4\}$ or $d_{yz} \leq \max\{(T_y + T_z)/5, K_J/4\}$. If d_{xz} or d_{yz} is bounded by $K_J/4$, then the lemma follows from Lemma 4. Thus, assume $d_{xz} \leq (T_x + T_z)/5$ (the proof is similar for the case $d_{yz} \leq (T_y + T_z)/5$). By Lemma 4, the semi-normal scheduling S_{xz} constructed by the algorithm **Scheduler** based on the partition (B'_{xz}, B''_{xz}) given by the algorithm **Partition**(x, z) has makespan equal to $\max\{\text{Opt}(J), |B'_{xz}| + |J_{xz}|, |B''_{xz}| + |J_{xz}|\}$. We have

$$\begin{aligned}
 &\max\{|B'_{xz}| + |J_{xz}|, |B''_{xz}| + |J_{xz}|\} = \max\{|B'_{xz}|, |B''_{xz}|\} + |J_{xz}| \\
 &= (|B'_{xz}| + |B''_{xz}|)/2 + d_{xz}/2 + |J_{xz}| \\
 &= (|J_x| + |J_z| + |J_{xy}| + |J_{yz}| + 2|J_{xz}|)/2 + d_{xz}/2 \\
 &= (T_x + T_z)/2 + d_{xz}/2 \leq (T_x + T_z)/2 + (T_x + T_z)/10 \\
 &= (3/5)(T_x + T_z) \leq (3/5)(\text{Opt}(J) + (7/8)\text{Opt}(J)) \\
 &= (9/8)\text{Opt}(J)
 \end{aligned}$$

Thus, the semi-normal scheduling S_{xz} constructed by the algorithm **Scheduler** on index pair $\{x, z\}$ has makespan bounded by $(9/8)\text{Opt}(J)$. $\square$

5 Concluding Analysis and Final Remarks

We are finally able to conclude the approximation ratio of our main algorithm **SemiNormal** on the $P_3|fix|C_{\max}$ problem.

Theorem 1. *The approximation ratio of the algorithm **SemiNormal** for the $P_3|fix|C_{\max}$ problem is bounded by $9/8$.*

Proof. Let J be an instance for the $P_3|fix|C_{\max}$ problem and let $\bar{J}$ be the job set J with all 1-jobs of processing time bounded by $K_J/4$ removed. Thus, $\bar{J}$ is an instance with no small 1-jobs.

By Lemma 8, on an index pair $\{y, z\}$, based on the partition $(\bar{B}'_{yz}, \bar{B}''_{yz})$ of the list L_{yz}^D given by the algorithm **Partition**, the algorithm **Scheduler** constructs the semi-normal scheduling $\bar{S}_{yz}$ for the instance $\bar{J}$ whose makespan is the minimum over all semi-normal schedulings of $\bar{J}$. By Lemma 4 and Lemma 10, the makespan of $\bar{S}_{yz}$ is equal to $\max\{\text{Opt}(\bar{J}), |\bar{B}'_{yz}| + |J_{yz}|, |\bar{B}''_{yz}| + |J_{yz}|\}$ and is

bounded by $(9/8)Opt(\bar{J})$ (note that the job sets J and $\bar{J}$ have the same set of 2-jobs).

Applying the algorithm **Partition** to the list L_{yz} results in a partition (B'_{yz}, B''_{yz}) of L_{yz} . Applying the algorithm **Scheduler** on this partition gives a scheduling S_{yz} of the job set J , whose makespan is equal to $\max\{Opt(J), |B'_{yz}| + |J_{yz}|, |B''_{yz}| + |J_{yz}|\}$. If the covering item of the partition (B'_{yz}, B''_{yz}) is not in the list L_{yz}^D , then by Corollary 1, the scheduling S_{yz} has makespan bounded by $(9/8)Opt(J)$.

On the other hand, suppose the covering item of the partition (B'_{yz}, B''_{yz}) is in the list L_{yz}^D . By the algorithm **Partition**, the partition (B'_{yz}, B''_{yz}) is constructed by starting with the partition $(\bar{B}'_{yz}, \bar{B}''_{yz})$ of the list L_{yz}^D then applying Graham List Scheduling on the items in $L_{yz} - L_{yz}^D$. Thus, if the covering item of (B'_{yz}, B''_{yz}) is in L_{yz}^D , then all items in $L_{yz} - L_{yz}^D$ are in the smaller box among B'_{yz} and B''_{yz} , and we have

$$\max\{|B'_{yz}|, |B''_{yz}|\} = \max\{|\bar{B}'_{yz}|, |\bar{B}''_{yz}|\}$$

Thus, the makespan of the scheduling S_{yz} for J is bounded by

$$\begin{aligned} \max\{Opt(J), |\bar{B}'_{yz}| + |J_{yz}|, |\bar{B}''_{yz}| + |J_{yz}|\} &\leq \max\{Opt(J), (9/8)Opt(\bar{J})\} \\ &\leq (9/8)Opt(J) \end{aligned}$$

The last inequality is because $\bar{J}$ is a subset of J so $Opt(\bar{J}) \leq Opt(J)$.

The theorem now follows since the algorithm **SemiNormal** picks the best scheduling S_{yz} among all index pairs $\{y, z\}$. $\square$

We close this paper by a number of remarks.

Goemans' algorithm allows splitting one 1-job set J_y , while our algorithm allows splitting two 1-job sets J_y and J_z . It is easy to see that splitting further 1-job set will not get any improvement. In fact, our semi-normal scheduling only requires that the 2-jobs of the same mode be executed consecutively, regardless how the 1-jobs are split. Therefore, to achieve further improvement, we must consider splitting 2-job sets.

The makespan of semi-normal schedulings cannot be arbitrarily close to the optimal makespan. In fact, by Lemma 4 (and also see the proof of Lemma 8), an optimal semi-normal scheduling for J can be obtained by an optimal partition of the list L_{yz} for some index pair $\{y, z\}$. It is well-known that the optimal list partition problem has a fully polynomial time approximation scheme [9]. Thus, if the makespan of semi-normal schedulings is arbitrarily close to the optimal makespan, then the problem $P_3|fix|C_{\max}$ would also have a fully polynomial time approximation scheme, contradicting the fact that $P_3|fix|C_{\max}$ is NP-hard in the strong sense [13].

Despite the above fact, the approximation ratio of our algorithm is much better than $9/8$ in most cases, in particular for instances that do not have very large 1-jobs. For example, if the processing time of 1-jobs in an instance J is bounded by $K_J/10$, then from Lemma 2 and Lemma 4, it is not difficult to see that the scheduling constructed by our algorithm has makespan bounded by $(21/20)Opt(J)$.

Approximation algorithms for multiprocessor job scheduling problems based on normal schedulings and their variations are in general very simple and seem to achieve very good approximation ratio. Besides the previous success [8,10] and our current paper on the $P_3|fix|C_{\max}$ problem, we have recently extended our techniques successfully to achieve improved approximation ratio for systems with more than three processors [12]. For example, for the $P_k|fix|C_{\max}$ problem for $k = 4$ and 5, our algorithms based on normal schedulings achieve approximation ratios 1.5 and 2, respectively, improving the previous best results [6]. It should be interesting to further explore the potential of this method.

References

1. A. K. AMOURA, E. BAMPIS, C. KENYON, AND Y. MANOUSSAKIS, Scheduling independent multiprocessor tasks, *Proc. 5th Ann. European Symposium on Algorithms, Lecture Notes in Computer Science 1284*, (1997), pp. 1-12. 49
2. J. BLAZEWICZ, P. DELL'OLMO, M. DROZDOWSKI, AND M. SPERANZA, Scheduling multiprocessor tasks on the three dedicated processors, *Information Processing Letters 41*, (1992), pp. 275-280. 48
3. J. BLAZEWICZ, P. DELL'OLMO, M. DROZDOWSKI, AND M. SPERANZA, Corrigendum to "Scheduling multiprocessor tasks on the three dedicated processors, *Information Processing Letters 41*, (1992), pp. 275-280." *Information Processing Letters 49*, (1994), pp. 269-270. 48
4. J. BLAZEWICZ, M. DROZDOWSKI, AND J. WEGLARZ, Scheduling multiprocessor tasks to minimize scheduling length, *IEEE Transactions on Computers 35*, (1986), pp. 389-393.
5. J. BLAZEWICZ, W. DROZDOWSKI, AND J. WEGLARZ, Scheduling multiprocessor tasks – a survey, *International Journal of Microcomputer Applications 13*, (1994), pp. 89-97. 48
6. J. CHEN AND C.-Y. LEE, General multiprocessor task scheduling, *Naval Research Logistics 46*, (1999), pp. 57-74. 59
7. J. CHEN AND A. MIRANDA, A polynomial time approximation scheme for general multiprocessor job scheduling, *SIAM J. Comput. 31*, (2001), pp. 1-17. 49
8. P. DELL'OLMO, M. G. SPERANZA, ZS. TUZA, Efficiency and effectiveness of normal schedules on three dedicated processors, *Discrete Mathematics 164*, (1997), pp. 67-79. 49, 51, 59
9. M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979. 58
10. M. X. GOEMANS, An approximation algorithm for scheduling on three dedicated machines, *Discrete Applied Mathematics 61*, (1995), pp. 49-59. 49, 59
11. R. L. GRAHAM, Bounds for certain multiprocessing anomalies, *Bell System Technical Journal 45*, (1966), pp. 1563-1581. 52
12. J. HUANG, J. CHEN, AND S. CHEN, A simple linear time approximation algorithm for multi-processor job scheduling on four processors, *Lecture Notes in Computer Science 1969 (ISAAC'00)*, (2000), pp. 60-71. 59
13. J. A. HOOGEVEEN, S. L. VAN DE VELDE, AND B. VELTMAN, Complexity of scheduling multiprocessor tasks with prespecified processor allocations, *Discrete Applied Mathematics 55*, (1994), pp. 259-272. 48, 58

14. K. JANSEN AND L. PORKOLAB, General multiprocessor task scheduling: approximate solutions in linear time, *Lecture Notes in Computer Science 1663* (WADS'99), (1999), pp. 110-121. [49](#)
15. H. KRAWCZYK AND M. KUBALE, An approximation algorithm for diagnostic test scheduling in multicomputer systems, *IEEE Transactions on Computers* 34, (1985), pp. 869-872. [48](#)
16. C.-Y. LEE, L. LEI, AND M. PINEDO, Current trends in deterministic scheduling, *Annals of Operations Research* 70, (1997), pp. 1-42. [48](#)

Balanced Scheduling toward Loss-Free Packet Queuing and Delay Fairness

Hisashi Koga

¹ Fujitsu Laboratories Ltd.
Kawasaki 211-8588, Japan

² Department of Information Science, University of Tokyo
Tokyo 113-0033, Japan
koga@flab.fujitsu.co.jp

Abstract. Packet losses in the current networks take place because of buffer shortage in a router. This paper studies how many buffers should be prepared in a router to eliminate packet losses in the context that an on-line scheduling algorithm in the router must decide the order of transmitting packets among m queues each of which corresponds to a single traffic stream. To exclude packet losses with a small amount of buffers, the maximum queue length must be kept low over the whole scheduling period. This new on-line problem is named the *balanced scheduling problem* (BSP). By competitive analysis, we evaluate the power of on-line algorithms regarding to the prevention of packet losses. The BSP accompanies tasks with negative costs. Solving an on-line problem which admits tasks with negative costs is our main theoretical contribution. We prove a simple greedy algorithm is $\Theta(\log m)$ -competitive and nearly optimal, while the *ROUND ROBIN* scheduling cannot break the trivial upper bound of m -competitiveness. Finally, this paper examines another balancing problem whose objective is to balance the delay among the m traffic streams.

1 Introduction

Network communication represented by the Internet has been expanded to public steadily during the past decade. However, the current Internet is inadequate for commercial use because of its best-effort natures which admits packet losses, when the network links are congested. In the current best-effort network the end source host has to retransmit the discarded packets like TCP protocol to recover the lost information when a packet loss occurs. Unfortunately this solution has the disadvantage that additional traffic may make the congestion worse. For this reason, it is marvelous if one can construct a network environment where the network itself guarantees no packet loss in the first place.

In general, packet losses are caused when buffers in a network router run short because of sudden burst traffic. Therefore, there are two means to prevent packet losses: (I): To restrict the amount of the total traffic flowing into the router and/or (II): To prepare many buffers in the router.

The former approach is called the admission control in the research area of QoS (quality of service) networks. As for the latter approach that is the

theme of this paper, avoiding packet losses is possible if the router is ideally given infinite buffers. This paper studies the amount of buffers which should be given to a router to eliminate packet losses in the context that m traffic streams flowing into a router R shares the same output port and a scheduling algorithm in R must decide the order of transmitting packets among m FIFO queues each of which is responsible for exactly one traffic stream (Fig. 1). At each time unit t , N_t packets arrive at R . Here N_t depends on t and $N_t \geq 0$. To which traffic stream a packet belongs is identified by a label attached to the packet. According to this label, the packet is once stored into the corresponding FIFO queue. As for the output, R chooses one non-empty queue and outputs the packet at its head per time unit. Therefore the phenomenon that $N_t > 1$ implies some burst traffic is breaking out. This paper assumes a simple fair Complete Buffer Partitioning (CBP [8]) scheme which fairly allocates the same number of buffers to the m queues statically, i.e. reassigning buffers dynamically among the queues is disallowed. It is very easy to implement the CBP scheme.

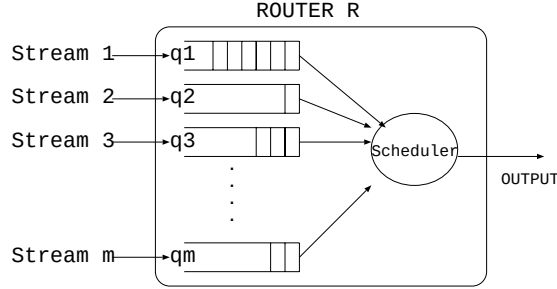


Fig. 1. Scheduling in a router

In the above setting, the number of buffers enough to exclude packet losses is equal to m times the maximal queue length where the maximum is taken over the whole period during which the scheduling algorithm in R serves all the packets. This quantity depends on the scheduling policy, so that we can judge the power of scheduling algorithms in terms of the prevention of packet losses from the maximal queue length over the whole scheduling period. For this reason, this paper investigates a new on-line scheduling problem whose purpose is to minimize the maximal queue length over the whole scheduling period. We name this fresh problem the on-line *balanced scheduling problem* (BSP).

We evaluate the power of on-line algorithms for the BSP using competitive analysis [9] which compares the performance of an on-line algorithm to that of the optimal off-line algorithm. Concretely, we examine how many times on-line algorithms must prepare as many buffers as the optimal off-line algorithm opt which knows the entire packet arrival sequence in advance so as to eliminate packet losses. Let $L_A(\sigma)$ be the maximal queue length over the whole scheduling

period when a scheduling algorithm A serves a packet arrival sequence σ . An on-line algorithm A is called c -competitive if $L_A(\sigma) \leq c \times L_{opt}(\sigma)$ for any σ .

First in Sect. 2, the balanced scheduling problem is formally defined. In Sect. 3, the lower bounds of the competitiveness of on-line algorithms are investigated. We show that any deterministic/randomized on-line algorithm cannot exceed the competitiveness of $\Omega(\log m)$. Specifically the popular algorithm *ROUND ROBIN* is not better than the trivial upper bound of m -competitiveness. In Sect. 4, the greedy algorithm named *GREEDY* which always selects the longest queue at each time unit is studied. We show that *GREEDY* is a nearly optimal on-line algorithm and achieves the competitiveness of $O(\log m)$. Thus, *GREEDY* is by far superior to *ROUND ROBIN* regarding to the prevention of packet losses.

We would like to emphasize that the basic essence of the BSP to balance some objective function among traffic streams is getting more and more important in QoS networks. While the BSP aims at balancing the queue length, balancing other objective functions, say delay, is also worth pursuing from the practical viewpoint. For example, a promising QoS model named *Proportional Delay Differentiation Service* [6] requires that the weighted delay per packet should be balanced among traffic streams so that each traffic stream may receive a different level of service proportional to its importance. Motivated by this work, in Sect. 5, we consider an on-line problem whose purpose is to decrease the maximum sum-of-delays incurred in a single queue. This problem is named the *delay balanced scheduling problem* (DBSP). For the DBSP, no deterministic on-line algorithm is better than $\Omega(\log m)$ -competitive.

Here, we remark this paper does not argue that *ROUND ROBIN* is useless, because *ROUND ROBIN* achieves the throughput fairness among the streams which *GREEDY* does not. For practical use, *GREEDY* should be adopted after the length of the longest queue goes beyond some threshold value and another scheduling algorithm like *ROUND ROBIN* should be used till then.

1.1 Related Work

Theoretically the BSP is related to the on-line load balancing problem initiated by Graham [7]. In the load balancing problem, given m servers, we must assign each incoming task to one of the m servers in such a way that the maximum load on the servers is minimized. Each task arrives one by one and holds its own positive load vector of length m whose coordinates indicate the increase in load when it runs on the corresponding server. Many variants of on-line load balancing problems have been considered so far. In the identical machines model [7] all the coordinates of a load vector are the same. In the restricted machines model [4], each task can be handled only by a subset of the servers, though all the coordinates of a load vector are the same. The natural greedy algorithm becomes $(2 - \frac{1}{m})$ -competitive in the identical machines model [7] and $\Theta(\log m)$ -competitive in the restricted machines model [4] respectively. The temporal tasks model [2][3] assumes that tasks have a limited duration and disappear after their duration. The greedy algorithm becomes $\Theta(m^{\frac{2}{3}})$ -competitive [2] in this model.

The BSP differs greatly from the traditional on-line load balancing problem in that load of all the servers must be balanced by selecting departing packets.

By the procedure illustrated in Fig. 2, the BSP is transformed to a new on-line load balancing problem that must face two difficulties that all the previous models do not have:

- The coordinates of a load vector of a task may take a negative value.
- A subset of servers on which a individual task can run depends on the behavior of the scheduling algorithm.

Especially, handling the former difficulty is a large contribution of this paper, because tasks with negative costs usually make competitive analysis infeasible. Roughly speaking, the transformed BSP seems to be the extension of the restricted machines model that admits tasks with negative costs. Because *GREEDY* is $\Theta(\log m)$ -competitive for the BSP, the complexity of the BSP does not differ from that of the restricted machines model of the on-line load balancing problem interestingly, despite tasks with negative costs are introduced. By contrast, *GREEDY* achieves a quite smaller upper bound in the BSP than in the temporal tasks model, though both problems admit tasks to leave servers. We infer the reason for this difference is that the scheduler decides aggressively the finish time of tasks in the BSP unlike the temporal tasks model.

1. A packet arrival in the BSP is mapped to a task of size 1 which only a single specific server can process.
2. A packet output from R in the BSP is mapped to a task of size -1 which only non-idol servers can process.

Fig. 2. Transforming Procedure of the BSP

2 Problem Statement

2.1 Balanced Scheduling Problem

The balanced scheduling problem (BSP) is formally defined as follows. We are given m FIFO queues $q_1, q_2, \dots, q_m$ in a router R and a sequence of packet arrivals at R . Initially at time 0, all the m queues are empty. At each time $t > 0$, N_t packets expressed by m -tuples $(N_t^1, N_t^2, \dots, N_t^m)$ first arrive at R , where each N_t^i is a non-negative integer and $N_t = \sum_{i=1}^m N_t^i$. The packets that have just arrived are stored into the m queues such that N_t^i packets go into q_i for $1 \leq i \leq m$. Then a scheduling algorithm A operating in R selects one non-empty queue and outputs a packet at its head unless all the queues are empty. We assume that at least one queue is not empty until the end of the whole scheduling period. This assumption does not lose generality, because, if there is a time when all the queues are empty, we can partition the packet arrival sequence into multiple subsequences for each of which the BSP is separately solved.

Let $l_A^i(t)$ be the length of q_i at time t in the running of the scheduling algorithm A after arriving packets are stored into the corresponding queue. Since

the length of a queue before A outputs a packet from R may differ from that after A outputs it within the same time instance t , we distinguish the time after the output of the packet by attaching a superscript a to t like t^a specially when necessary. Since the maximum instantaneous queue length must be considered to avoid packet losses, we normally pay attention to the length of the queues before A outputs a packet, i.e. $l_A^i(t)$, not $l_A^i(t^a)$. The notation of t^a is used only for the analysis of algorithms. The maximal queue length at time t by A is defined as

$$l_A(t) = \max_{1 \leq i \leq m} \{l_A^i(t)\}.$$

Let σ be a sequence of packet arrivals and $|\sigma|$ be the time of the last arrival. Then, the maximal queue length over the whole scheduling period for σ by A is defined as

$$L_A(\sigma) = \max_{0 \leq t \leq |\sigma|} l_A(t). \quad (1)$$

The purpose of the BSP is to find a scheduling algorithm A that reduces $L_A(\sigma)$.

Let us describe the total number of packets stored in the m queues at time t as $C(t)$. Note that $C(t)$ does not depend on the scheduling algorithm, because the number of packets that have left R before t is independent of the scheduling algorithm. For any on-line algorithm A , $l_A(t) \leq C(t)$ trivially. For the optimal off-line algorithm opt , $l_{opt}(t) \geq \frac{C(t)}{m}$ because $C(t)$ packets are distributed among the m queues. Therefore it holds that $l_A(t) \leq m \cdot l_{opt}(t)$ for any t . Thus, the obvious upper bound for the BSP is derived as Theorem 1.

Theorem 1. *Any on-line algorithm for the BSP is m -competitive at worst.*

2.2 Delay Balanced Scheduling Problem

To describe the DBSP, we need to explain what the term "delay" means. Suppose that a packet p arrives at R at time t_1 and leaves R at time t_2 . Then the delay of the packet, d_p , is $t_2 - t_1$. Let P_i be the set of all the packets assigned to q_i over the whole scheduling period. Then the total delay D_A^i of a queue q_i incurred by the scheduling algorithm A is defined as $\sum_{p \in P_i} d_p$. The purpose of the DBSP is to construct a scheduling algorithm A which minimizes $\max_{1 \leq i \leq m} D_A^i$.

In analyzing algorithms for the DBSP, we suppose a packet p incurs a delay of 1 per time unit over its duration, rather than it incurs a delay of d_p all at once at the end of its duration. Then the total delay of q_i up to time t by A , $D_A^i(t)$, is defined as follows.

$$D_A^i(1) = 0.$$

$$D_A^i(t+1) = \begin{cases} D_A^i(t) + l_A^i(t) - 1. & \text{If } q_i \text{ is selected by } A \text{ at time } t. \\ D_A^i(t) + l_A^i(t). & \text{If } q_i \text{ is not selected by } A \text{ at time } t. \end{cases}$$

The function $D_A^i(t)$ is equal to D_A^i at the end of the scheduling period. If we define $D_A(t)$ as $\max_{1 \leq i \leq m} D_A^i(t)$, $D_A(t)$ is equal to $\max_{1 \leq i \leq m} D_A^i$ at the end of the scheduling period similarly. In Sect. 5, we deal with the function $D_A(t)$.

3 The Lower Bound

3.1 Lower Bounds for General On-line Algorithms

At the beginning, we obtain lower bounds for general on-line algorithms. A technique similar to the lower bound technique [4] for the restricted machines model of the on-line load balancing problem is exploited. In the proof, an adversary constructs a packet arrival sequence σ which annoys on-line algorithms.

Theorem 2. *Let A be any deterministic on-line algorithm for the BSP. Then A is not better than $(1 + \lfloor \log_2 m \rfloor)$ -competitive.*

Proof. Let j be the largest integer satisfying $2^j \leq m$, i.e. $j = \lfloor \log_2 m \rfloor$. The adversary constructs σ by dividing it into $j + 1$ phases. For $1 \leq k \leq j$, the k -th phase starts at time $1 + \sum_{r=1}^{k-1} 2^{j-r}$ and lasts for 2^{j-k} time units. The final $(j + 1)$ -th phase starts at time $1 + \sum_{r=1}^j 2^{j-r}$ and continues only for one time unit. For example, the first phase starts at time 1 and finishes at time 2^{j-1} , the second phase starts at time $1 + 2^{j-1}$ and finishes at time $2^{j-1} + 2^{j-2}$ and so on. How to construct σ in the k -th phase is shown below in detail.

- Step 1: 2^{j-k+1} packets arrive at R when the k -th phase starts, so that exactly one new packet is assigned to each of the 2^{j-k+1} longest queues in A 's running. Ties are broken arbitrarily. Note that the adversary predicts accurately the lengths of all the m queues since A is deterministic.
- Step 2: No more packet arrives during the k -th phase after Step 1.

σ has a property that the number of leaving packets in the k -th phase is equal to the number of arriving packets in the $(k + 1)$ -th phase for $1 \leq k \leq j$. From now on, we show that $L_A(\sigma)$ reaches $1 + \lfloor \log_2 m \rfloor$ while $L_{opt}(\sigma) = 1$. Let T_k be the time when the k -th phase begins.

As for opt , in the k -th phase, opt selects the 2^{j-k} queues to which a packet is assigned in the $(k + 1)$ -th phase exactly once. This scheduling keeps the invariant that, at the beginning of every phase, exactly 2^j queues hold just one packet and the rest of the queues are empty. That is, for any t , $l_{opt}(t) = 1$. Hence, $L_{opt}(\sigma) = 1$.

As for A , by induction on the index of phases k , we prove that the 2^{j-k+1} longest queues have a length of k at time T_k . The base case is trivial, since 2^j queues have a length of 1 at time 1 from the structure of σ . Assume that the 2^{j-k+1} longest queues have a length of k at time T_k in A 's running. 2^{j-k} packets leave R in the k -th phase because its duration is 2^{j-k} . Thus, at least $2^{j-k+1} - 2^{j-k} = 2^{j-k}$ queues still have a length of k when the k -th phase terminates. Because the 2^{j-k} longest queues increase their lengths by 1 at the beginning of the next phase, it follows that the 2^{j-k} longest queues have a length of $k + 1$ at T_{k+1} , which completes the proof of the induction step. Thus it holds that $l_A(T_k) = k$ for $1 \leq k \leq j + 1$. Therefore, $L_A(\sigma) = 1 + \lfloor \log_2 m \rfloor = (1 + \lfloor \log_2 m \rfloor) \cdot L_{opt}(\sigma)$. $\square$

The proof technique in Theorem 2 enables us to derive a randomized lower bound also. The proof is omitted here.

Theorem 3. *Let A be any randomized on-line algorithm for the BSP. Then A is not better than $(1 + \frac{\lfloor \log_2 m \rfloor}{2})$ -competitive against the oblivious adversary.*

3.2 Lower Bound for ROUND ROBIN

Next we examine the lower bound of a popular algorithm *ROUND ROBIN*. Unfortunately *ROUND ROBIN* cannot exceed the trivial upper bound of m .

Algorithm ROUND ROBIN: Initially the algorithm may select any non-empty queue. On condition that the algorithm selects a queue q_i at time t , the queue selected at time $t + 1$ is the queue $q_{(r+i) \bmod m}$. Here r is the minimum positive integer satisfying the condition that $q_{(r+i) \bmod m}$ is not empty.

Theorem 4. *ROUND ROBIN is not better than m -competitive.*

Proof. Without loss of generality, we assume that *ROUND ROBIN* selects q_1 initially. Again an adversary passes a bad packet arrival sequence σ to *ROUND ROBIN*. σ is constructed in the next way.

- Step1: At time 1, m^2 packets arrive at R such that m packets are assigned to each q_i for $1 \leq i \leq m$.
- Step2: At time $km + 1$ for $1 \leq k \leq m$, m new packets are assigned to q_m .

The analysis advances by dividing the scheduling period into $m + 1$ phases. The k -th phase begins at time $(k - 1)m + 1$ and finishes at time km for $1 \leq k \leq m + 1$. Note that the number of packets that leave R in the k -th phase is equal to the number of arriving packets in the $(k + 1)$ -th phase for $1 \leq k \leq m$.

The optimal off-line algorithm opt chooses q_m all the time. This assures that the length of q_m equals 0 at the end of each phase and that it increases to m at the beginning of the k -th phase for $k > 2$. The lengths of all the other queues take a constant value of m all the time. Hence it holds that $l_{opt}(t) = m$ for an arbitrary time t . Thus, $L_{opt}(\sigma) = m$.

On the other hand, *ROUND ROBIN* selects all the m queues once in each phase except the last $(m + 1)$ -th phase, since all the queues have at least one packet at the beginning of the k -th phase ($k \leq m$). Since the algorithm initially selects q_1 , q_m is always the longest queue in the running of *ROUND ROBIN*. The length of q_m at the beginning of the k -th phase ($k \leq m + 1$) is calculated as:

$$l_{ROUND\ ROBIN}((k - 1)m + 1) = mk - (k - 1) = (m - 1)k + 1.$$

This value reaches to the maximum when $m + 1$ is substituted for k . Hence, $L_{ROUND\ ROBIN}(\sigma) = (m - 1)(m + 1) - 1 = m^2 = m \cdot L_{opt}(\sigma)$. $\square$

4 The Upper Bound

This section analyzes the performance of a specific algorithm *GREEDY*. Since simple greedy policies are analyzed in many load-balancing problems [1][2][4][7], measuring the performance of *GREEDY* enables us to estimate the relative difficulty of the BSP to other problems.

Algorithm GREEDY: At any time, *GREEDY* selects the longest queue. Ties are broken arbitrary.

Theorem 5. *GREEDY is $(3 + \lceil \log_2 m \rceil)$ -competitive.*

Theorem 5 together with Theorem 2 claims that *GREEDY* is a nearly optimal on-line algorithm. We extend the proof technique for the restricted machines model of the on-line load balancing problem [4].

Proof. We introduce a function named *gap* which maps packets in a FIFO queue in *GREEDY*'s running to some integers. Let σ be an arbitrary packet arrival sequence. Suppose p is the r -th packet from the top (i.e. the output port) of a FIFO queue q_i at time t in *GREEDY*'s running. Then the gap of the packet p at time t , denoted by $\text{gap}(p, t)$ is defined as $r - l_{opt}^i(t)$. Intuitively the function *gap* expresses the height of packets in a FIFO queue in *GREEDY*'s running relative to the length of the corresponding queue in *opt*.

According to the value of *gap*, we partition FIFO queues in *GREEDY*'s running into layers as follows. Denote $L_{opt}(\sigma)$ by l . See Fig. 3(A). The k -th layer of a queue q_i at time t consists of packets p stored in q_i at t in *GREEDY*'s running such that $(k-1)l+1 \leq \text{gap}(p, t) \leq kl$. The number of packets contained in the k -th layer of q_i is expressed as $W_k^i(t)$. The next property about $W_k^i(t)$ is crucial in the analysis of *GREEDY*.

Lemma 1. *For any $k \geq 1$, $W_k^i(t) = l$ if $W_{k+1}^i(t) > 0$.*

Proof. Because $W_{k+1}^i(t) > 0$, $l_G^i(t) > l_{opt}^i(t) + kl$. Hence, the number of packets in q_i whose gaps are greater than or equal to $(k-1)l+1$ but less than or equal to kl is exactly l in *GREEDY*'s running. $\square$

Corollary 1. *For any $k \geq 1$, $W_k^i(t) \geq W_{k+1}^i(t)$.*

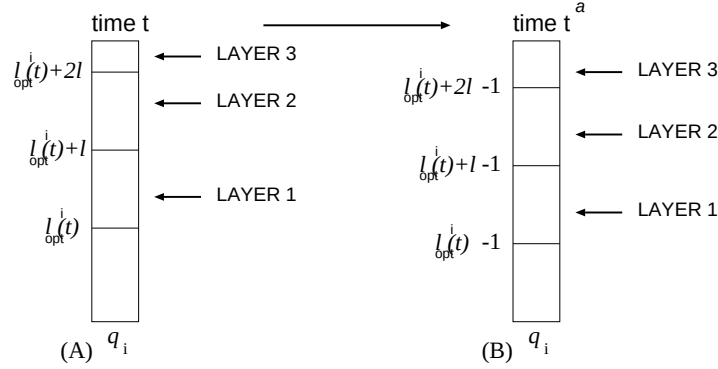


Fig. 3. Partition of a queue into layers

Furthermore, the following notations are required to proceed the proof.

- $R_k^i(t) = \sum_{j>k} W_j^i(t)$.
- $W_k(t) = \sum_{i=1}^m W_k^i(t)$.

$$- R_k(t) = \sum_{j>k} W_j(t).$$

$R_k^i(t)$ is equal to the total number of packets in q_i contained in the layers strictly higher than the k -th layer. $W_k(t)$ presents the total number of packets contained in the k -th layer over all the m queues, while $R_k(t)$ presents the total number of packets contained in the layers strictly higher than the k -th layer over all the m queues. Therefore, it holds, for any $k \geq 1$, that

$$R_{k+1}(t) = R_k(t) - W_{k+1}(t). \quad (2)$$

Note that $W_k(t^a) = W_k(t+1)$ and $R_k(t^a) = R_k(t+1)$ because the number of packets in each layer of a certain queue q_i is not affected even if the same number of packets arrive at q_i at the beginning of time $t+1$ both in *opt* and in *GREEDY*. By contrast $W_k(t^a)$ ($R_k(t^a)$) may be different from $W_k(t)$ ($R_k(t)$) respectively depending on the queues selected by the two algorithms at time t .

Our strategy is to compare the simultaneous running of the two algorithms *GREEDY* and *opt* on an arbitrary packet arrival sequence σ and to prove the next relation is maintained all the time for all $k \geq 1$.

$$W_k(t) \geq R_k(t). \quad (3)$$

Assume that (3) is correct. Then, from (2) and (3), we have $R_{k+1}(t) = R_k(t) - W_{k+1}(t) \leq R_k(t) - R_{k+1}(t)$. Thus,

$$R_{k+1}(t) \leq \frac{1}{2} R_k(t).$$

Then, by applying this inequality $\lceil \log_2 m \rceil$ times, the next inequality is derived. Note that $R_1(t) \leq ml$, because $L_{opt}(\sigma) = l$.

$$R_{\lceil \log_2 m \rceil + 1}(t) \leq \left(\frac{1}{2}\right)^{\lceil \log_2 m \rceil} R_1(t) \leq \left(\frac{1}{2}\right)^{\log_2 m} R_1(t) = \frac{1}{m} R_1(t) \leq \frac{ml}{m} = l.$$

Therefore the number of packets included in the layers strictly higher than the $(\lceil \log_2 m \rceil + 1)$ -th layer is at most l . As the result, the length of the longest queue in *GREEDY*'s running at time t is bounded from above as follows.

$$l_G(t) \leq l_{opt}(t) + (\lceil \log_2 m \rceil + 1)l + l \leq (3 + \lceil \log_2 m \rceil)l. \quad (4)$$

Since (4) holds without regard to time t , the proof of Theorem 5 is complete.

From now on we verify (3) for all $k \geq 1$ and for any t . Pick up an arbitrary positive integer as k . The proof makes use of the induction on time t . As for the base case, (3) is trivial at time 1 before packets are output, since all the m queues have the same number of packets both in *opt* and in *GREEDY*.

In the next, suppose that $W_k(t) \geq R_k(t)$ at t before the two algorithms select queues from which a packet is output. It suffices to show that

$$W_k(t^a) \geq R_k(t^a),$$

because $W_k(t+1) = W_k(t^a)$ and $R_k(t+1) = R_k(t^a)$. Assume q_i is the queue selected by *opt* and q_j is the one selected by *GREEDY* at t . If q_i is identical

with q_j , it is obvious that $W_k(t^a) = W_k(t) \geq R_k(t) = R_k(t^a)$. Let us suppose $q_i \neq q_j$ hereafter. Because $W_k^h(t) = W_k^h(t^a)$ and $R_k^h(t) = R_k^h(t^a)$ for any queue q_h except q_i and q_j , we may concentrate on how q_i and q_j behave only.

First consider the behavior of q_i . If $l_G^i(t) < l_{opt}^i(t)$, $l_G^i(t^a) \leq l_{opt}^i(t^a)$. So that

$$W_k^i(t) = W_k^i(t^a) = 0 \text{ and } R_k^i(t) = R_k^i(t^a) = 0. \quad (5)$$

If $l_G^i(t) \geq l_{opt}^i(t)$, the next arithmetic expressions are obtained from the definition of $W_k^i(t)$ and $R_k^i(t)$, where Y is defined as $\lfloor \frac{l_G^i(t) - l_{opt}^i(t)}{l} \rfloor + 1$. See Fig. 3(B).

$$W_k^i(t^a) = \begin{cases} W_k^i(t), & \text{if } k \neq Y \\ W_k^i(t) + 1, & \text{if } k = Y \end{cases} \quad (6)$$

$$R_k^i(t^a) = \begin{cases} R_k^i(t) + 1, & \text{if } k < Y \\ R_k^i(t), & \text{if } k \geq Y \end{cases} \quad (7)$$

Next consider the behavior of q_j . If $l_G^j(t) \leq l_{opt}^j(t)$, $l_G^j(t^a) < l_{opt}^j(t^a)$. Thus,

$$W_k^j(t) = W_k^j(t^a) = 0 \text{ and } R_k^j(t) = R_k^j(t^a) = 0. \quad (8)$$

In case $l_G^j(t) > l_{opt}^j(t)$, let X be $\lceil \frac{l_G^j(t) - l_{opt}^j(t)}{l} \rceil$. From the definition of $W_k^j(t)$ and $R_k^j(t)$, the next arithmetic formulas are obtained.

$$W_k^j(t^a) = \begin{cases} W_k^j(t), & \text{if } k \neq X \\ W_k^j(t) - 1, & \text{if } k = X \end{cases} \quad (9)$$

$$R_k^j(t^a) = \begin{cases} R_k^j(t) - 1, & \text{if } k < X \\ R_k^j(t), & \text{if } k \geq X \end{cases} \quad (10)$$

From the equations from (5) to (10) and the assumption that $W_k(t) \geq R_k(t)$ at t , at least either of the next two conditions must be satisfied to break the relation (3) at t^a : (TYPE I) $W_k^j(t^a) = W_k^j(t) - 1$, or (TYPE II) $R_k^i(t^a) = R_k^i(t) + 1$. From now on, we show that (3) is preserved even if either of the above conditions takes place.

(TYPE I): Suppose that $W_k^j(t^a) = W_k^j(t) - 1$. From (9), $l_G^j(t) > l_{opt}^j(t)$ and k must be X . Hence $l_G^j(t) \leq l_{opt}^j(t) + lX$. We show the $(X+2)$ -th layers of all the m queues are empty at t^a by contradiction. Since $l_G^j(t) \leq l_{opt}^j(t) + lX$, $l_G^j(t^a) = l_G^j(t) - 1 \leq l_{opt}^j(t^a) + lX$ and the $(X+2)$ -th layer of q_j is empty at t^a . Assume there exists a queue $q_h (\neq q_j)$ whose $(X+2)$ -th layer contains some packets at t^a . Since q_h is not selected by *GREEDY* at t , we have

$$\begin{aligned} l_G^h(t) = l_G^h(t^a) &\geq (X+1)l + 1 \quad (\text{since } q_h \text{'s } (X+2)\text{-th layer is not empty}) \\ &> l_{opt}^j(t) + lX \geq l_G^j(t). \end{aligned}$$

This contradicts with the fact that q_j is selected by *GREEDY* at t . Thus the $(X + 2)$ -th layers of all the m queues must be empty at t^a . Thus, by applying Corollary 1, $W_X(t^a) \geq W_{X+1}(t^a) = R_X(t^a)$, which shows (3) holds for this case.

(TYPE II): Suppose that $R_k^i(t^a) = R_k^i(t) + 1$. Since $R_k^i(t^a) \geq 1$, we have $l_G^i(t) = l_G^i(t^a) \geq kl + 1$. Since *GREEDY* selects not q_i but q_j , we have

$$l_G^j(t) \geq l_G^i(t) \geq kl + 1 \geq l_{opt}^j(t) + (k - 1)l + 1.$$

Especially if $l_{opt}^j(t) + (k - 1)l + 1 \leq l_G^j(t) \leq l_{opt}^j(t) + kl$, we can show that the $(k + 2)$ -th layers of all the m queues are empty at t^a exactly in the same way as the previous paragraph. Thus, from Corollary 1, $W_k(t^a) \geq W_{k+1}(t^a) = R_k(t^a)$.

By contrast, if $l_G^j(t) > l_{opt}^j(t) + kl$, we have $R_k^j(t) > 0$. Hence, $R_k^j(t^a) = R_k^j(t) - 1$ after *GREEDY* outputs a packet at t from q_j . By comparing this with (8) and with (10), we have $l_G^j(t) > l_{opt}^j(t)$ and $k < X$.

$$\begin{aligned} R_k(t^a) &= R_k^i(t^a) + R_k^j(t^a) + \sum_{h \neq i, j} R_k^h(t^a) \\ &= (R_k^i(t) + 1) + (R_k^j(t) - 1) + \sum_{h \neq i, j} R_k^h(t) = R_k(t). \end{aligned} \quad (11)$$

About $W_k(t)$, it follows that $W_k^i(t^a) \geq W_k^i(t)$ from (6) and that $W_k^j(t^a) = W_k^j(t)$ from (9) as $k \neq X$. Hence,

$$W_k(t^a) \geq W_k(t). \quad (12)$$

From (11) and (12), it follows that $W_k(t^a) \geq W_k(t) \geq R_k(t) = R_k(t^a)$. Thus, we have proved (3) for all the possible cases, the entire proof of Theorem 5 ends. $\square$

5 The Lower Bound for the DBSP

Delay is another important performance measure for QoS networks. The purpose of the DBSP is to balance the total delay per queue. Here the total delay of a queue q_i is defined as the sum of delays of all the packets assigned to q_i . This section studies the deterministic lower bound of the competitiveness for the DBSP. Interestingly the DBSP contains the BSP as a subproblem and the lower bound for the BSP in Theorem 2 applies to the DBSP almost as it is.

The next lemma compares any deterministic on-line algorithm with some off-line algorithm *off* which may not be necessarily optimal.

Lemma 2. *Consider an arbitrary deterministic on-line algorithm A for the DBSP. Suppose that there exists a time t such that $l_A(t) = X$ and $l_{off}(t) = Y$, where *off* is a certain off-line algorithm in whose execution there are at least two non-empty queues at t before *off* selects a queue. Then A is not better than $\frac{X-1}{Y}$ -competitive.*

The condition on *off* simply guarantees $l_{off}(t^a)$ grows still Y and is not crucial.

Proof. Let q_i be the longest queue in A 's running and let q_j be the longest queue in off 's running at time t . In case multiple queues are the longest at t simultaneously in off 's running, q_j is chosen so that $D_{off}^j(t)$ may be larger than any other candidates. Since $l_A^i(t) = X$, $l_A^i(t^a) \geq X - 1$. On the other hand, since $l_{off}^j(t) = Y$, we have $l_{off}^j(t^a) = Y$ provided that off does not select q_j at t .

Suppose that exactly one packet is assigned to q_i per time unit and that no packet is assigned to the rest of the queues after t . Since any scheduling algorithm cannot output more than one packet at each time unit, the length of q_i in A is always greater than $X - 1$ after t . On the other hand, off keeps on outputting the packet that have just arrived immediately and keeps the length of the longest queue q_j to Y after t . Hence for any $t' > t$, we have

$$D_A(t') \geq D_A^i(t') \geq D_A^i(t) + (X - 1)(t' - t). \quad (13)$$

As for off , q_j always becomes the longest after t . Moreover, $D_{off}^j(t)$ is larger than any other queues that have the same length as q_j at t . Hence, for sufficiently large values of $t' > t$, it follows that

$$D_{off}(t') = D_{off}^j(t) + Y(t' - t). \quad (14)$$

The lower bound of competitiveness for the DBSP is obtained by dividing $D_A(t')$ by $D_{off}(t')$ as follows. $\frac{D_A(t')}{D_{off}(t')} \geq \frac{D_A^i(t) + (X-1)(t'-t)}{D_{off}^j(t) + Y(t'-t)}$. This value comes close to $\frac{X-1}{Y}$ as t' goes to ∞ . $\square$

By applying the bad request sequence σ for the BSP in Sect. 3 to the DBSP, the lower bound for the DBSP is derived as Theorem 6.

Theorem 6. *No deterministic on-line algorithm A for the DBSP is better than $\lfloor \log_2 m \rfloor$ -competitive.*

Proof. Let opt be the optimal off-line algorithm for the BSP in the proof of Theorem 2. By using opt as the base algorithm of off in Lemma 2, we can prove no deterministic on-line algorithm is better than $\frac{(1 + \lfloor \log_2 m \rfloor) - 1}{1} = \lfloor \log_2 m \rfloor$ -competitive for the DBSP. $\square$

6 Conclusion

This paper investigates the balanced scheduling problem (BSP) to evaluate the power of scheduling algorithms in a router in terms of prevention of packet losses. The BSP is a fresh on-line load balancing problem that faces a new difficulty that tasks can have negative costs.

There are many open problems regarding to the balanced scheduling problem. One important open problem is to find the optimal off-line algorithm. This enables us to compute the actual number of buffers in order for *GREEDY* to eliminate packet losses, since this paper has made clear the competitiveness of *GREEDY*. The problems below are also worth pursuing.

- Extending the BSP to dynamic buffer allocation policy. In this model, a traffic stream with higher priority can also use the buffer memories prepared for the streams with lower priority. To form this model, the BSP must be combined with the hierarchical server topology studied by [5].
- Changing the amount of buffers assigned to each queue.
- Discovering a competitive on-line algorithm for the DBSP.

Acknowledgment

I would like to thank Susanne Albers for helpful comments on the early version of this paper.

References

1. J. Aspens, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *Journal of ACM*, (44):486–504, 1997. 67
2. Y. Azar, A. Z. Broder, and A. R. Karlin. On-line load balancing. In *Proceedings of 33rd Symposium on Foundations of Computer Science*, pages 218–225, 1992. 63, 67
3. Y. Azar, B. Kalyanasundaram, S. Plotkin, K. R. Pruhs, and O. Waarts. On-line load balancing of temporary tasks. *Journal of Algorithms*, 22:93–110, 1997. 63
4. Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proceedings of 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992. 63, 66, 67, 68
5. A. Bar-Noy, A. Freund, and J. Naor. On-line load balancing in a hierarchical server topology. In *Proceedings of the Seventh Annual European Symposium on Algorithm*, pages 77–88, 1999. 73
6. C. Dovrolis, D. Stiladis, and P. Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. In *Proceedings of ACM SIGCOMM’99*, 1999. 63
7. R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966. 63, 67
8. A. M. Lin and J. A. Silvester. Priority queuing strategies and buffer allocation protocols for traffic control at an ATM integrated broadband switching system. *IEEE Journal on Selected Areas in Communications*, 9:1524–1536, 1991. 62
9. R. E. Tarjan and D. D. Sleator. Amortized efficiency of list update and paging rules. *Communication of the ACM*, 28:202–208, 1985. 62

Broadcasting with Universal Lists Revisited: Using Competitive Analysis

Jae-Hoon Kim and Kyung-Yong Chwa

Department of Electrical Engineering & Computer Science,
Korea Advanced Institute of Science and Technology, Taejeon 305-701, Korea
{jhoon,kychwa}@jupiter.kaist.ac.kr

Abstract. We study a variant of broadcasting: each node has a pre-determined ordered list of neighbors (regardless of the node, called the source, from which the message is originated to be transmitted to all nodes in a network) and transmits a received message to neighbors in order of the list. This problem was introduced in [3]. We propose a new measure of the efficiency of a broadcasting scheme, which is obtained from the *competitive analysis* [4,7], and we design new broadcasting schemes for lines, complete k -ary trees, grids, complete graphs, and hypercubes.

1 Introduction

Broadcasting is the process of transmitting a message held in one node, called the source, to every node in a network. In this process, each node which has already received the message transmits it to one of neighbors in a unit of time (, called a step).

A broadcasting algorithm determines the order of message transmissions to neighbors at every node, which can be viewed as an assignment of an ordered list of neighbors to every node. In classical broadcasting [5], the list assigned to each node may depend on the source. To execute broadcasting from any possible source, each node needs to have large local memory enough to store a lot of lists corresponding to different sources and needs to know the source to choose the corresponding list for each particular broadcasting. This requires substantial local memory at each node and increases the number of message bits circulated in a network since messages should contain the names of the sources which they are originated from.

We define a *broadcasting scheme* as a function assigning to every node a single ordered list(, called universal list,) of its neighbors regardless of the source such that after transmissions in order of the list at each node, all nodes receive the source message. (Here, we use the term of broadcasting scheme to be distinguished from the classical broadcasting algorithm.) Also we refer to as the *optimal (classical) broadcasting scheme* the broadcasting algorithm that can complete a broadcasting at every source in the minimum step under the classical model.

The problem of broadcasting with universal lists was introduced in [3]. Diks and Pelc considered two models: *adaptive* and *nonadaptive*. In the adaptive

model, each node knows which neighbors the obtained messages came from and can skip those neighbors in its list. But in the nonadaptive model, each node does not know the neighbors from which it receives the messages and may retransmit the messages to those neighbors. Thus each node obviously sends the source message to neighbors in order of its list. In some applications such as radio communication, nodes may not know from where messages come [1,2]. In this paper, we concentrate only on the nonadaptive model.

Given a graph G and a source s in G , we define $OPT(G, s)$ to be the number of steps in which the optimal broadcasting scheme, denoted by OPT , completes a broadcasting at s in G . Let $OPT(G)$ be the maximum of $OPT(G, s)$ over all sources s . Also $B_\sigma(G, s)$ is defined to be the number of steps used by a scheme σ to complete a broadcasting at s in G . Let $B(G, s)$ be the minimum of $B_\sigma(G, s)$ over all schemes σ and $B(G)$ be the maximum of $B(G, s)$ over all sources s . Then $B(G)$ is called the *broadcasting time* of G (for the nonadaptive model). In [3], they were able to establish $B(G)$ precisely for lines, rings, and grids, and gave an upper bound of $B(G)$ for trees, tori, and complete graphs. By definition, $B(G) \leq \max_s B_\sigma(G, s)$ for any broadcasting scheme σ . Thus they designed broadcasting schemes which give the equalities for lines, rings, grids, and the upper bounds for trees, tori, complete graphs, respectively.

In this paper, we propose a new measure of the efficiency of a broadcasting scheme using a *competitive analysis* [4,7]: A broadcasting scheme is in the absence of complete information, i.e., ignorant of the source. The performance of a broadcasting scheme is compared with that of the optimal broadcasting scheme. A broadcasting scheme σ in G is said to be *c-competitive* if there exists a constant d such that

$$B_\sigma(G, s) \leq c \cdot OPT(G, s) + d,$$

for all possible sources s . The infimum over all c such that σ is c -competitive, equivalently, $\max_s B_\sigma(G, s) / OPT(G, s)$, is called the *competitive ratio* of the broadcasting scheme σ in G and denoted by $\mathcal{C}(G, \sigma)$. Also we define the competitiveness of broadcasting (with universal lists) in G as the competitive ratio of the best possible broadcasting scheme, that is, $\min_\sigma \mathcal{C}(G, \sigma)$, where σ ranges over all broadcasting schemes in G .

Let L_n be the line of n (even) nodes. In [3], they proved that $B(L_n) = \frac{3}{2}(n-1) - \frac{1}{2}$. In Figure 1, it is shown the broadcasting scheme σ'_0 to give the equality. (The arrows represent the direction to the node which each node first passes a message to.) But the scheme σ'_0 has the competitive ratio of at least two since $B_{\sigma'_0}(L_n, c) = n-1 = 2OPT(L_n, c) - 1$. We consider another broadcasting scheme σ_0 in Figure 1. For any node s in L_n , it is easily shown that $B_{\sigma_0}(L_n, s) \leq \frac{3}{2}OPT(L_n, s)$. Also we can show that the scheme σ_0 is best possible, that is, the competitiveness of broadcasting in L_n is $\frac{3}{2}$. Given any broadcasting scheme σ . The worst case time $\max_s B_\sigma(L_n, s)$ is given at the left or right end node, and w.l.o.g., assume the maximum is taken at the right end node r . Since $B_\sigma(L_n, r) \geq B(L_n)$ and $OPT(L_n, r) = n-1$, $B_\sigma(L_n, r) \geq \frac{3}{2}OPT(L_n, r) - \frac{1}{2}$. Thus the competitive ratio of the scheme σ_0 is best possible over all broadcasting schemes

in L_n . This example gives a motivation to investigate the new measure and to design new broadcasting schemes with better performances *w.r.t.* the measure.

For any symmetric graph G , $OPT(G, s)$ are identical for all sources s , that is, $OPT(G, s) = OPT(G), \forall s$. Thus if a broadcasting scheme σ is c -competitive, then an upper bound of $B(G)$ can be given by $c \cdot OPT(G)$.

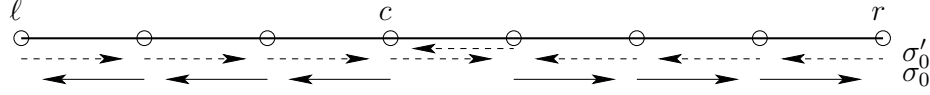


Fig. 1. Broadcasting schemes σ'_0 and σ_0 in the line

2 Trees

In this section, we begin with proving an upper bound for the competitiveness of broadcasting in trees. Following this, we give a lower bound for the competitiveness.

Let T be a tree and u, v be nodes in T . Then the *distance* from u to v , $d(u, v)$, is the least length of a path from u to v . The *eccentricity* of u is the maximum of its distances to other nodes. In the tree T , the diameter and radius are the maximum and minimum of the node eccentricities, respectively. The *center* of T is the subgraph induced by the nodes of minimum eccentricity, called *central nodes*. It is well known that the center of a tree is one node or one edge [8].

The set of nodes u in T for which $OPT(T, u)$ is minimum is called the *broadcasting center* of T , briefly, *b_center*, and denoted by $BC(T)$. The number $OPT(T, w)$ for a node w in $BC(T)$ is called the *broadcasting number* of T , denoted by $b(T)$. In fact, $BC(T)$ consists of a star with at least two nodes [6]. Cockayne et.al. [6] designed a linear time algorithm to find the *b_center* $BC(T)$ of T and showed that for any node v in T , $OPT(T, v)$ can be given from the broadcasting number $b(T)$.

Actually, in OPT, each node can receive only a single list regardless of the source but it can skip which neighbors the received messages came from. Specifically, for two adjacent nodes u and v in T , $T(u, v)$ and $T(v, u)$ denote the subtrees obtained by deleting the edge (u, v) from T that contain u and v , respectively. In OPT, wherever the source message is, each node u has a list $L_u = (u_1, \dots, u_k)$ such that

$$OPT(T(u_1, u), u_1) \geq OPT(T(u_2, u), u_2) \geq \dots \geq OPT(T(u_k, u), u_k), \quad (1)$$

and skips the node from which it receives a message.

Let u be the center node of the star $BC(T)$ and have the list $L_u = (u_1, \dots, u_k)$ defined by (1). Choose the smallest index j such that $j + OPT(T(u_j, u), u_j) = b(T)$. Then, $BC(T) = \{u, u_1, \dots, u_j\}$. See [6] for details.

Lemma 1. [6] *Let v be a node of a tree T which is not in the b -center of T and let the shortest distance from v to a node w in the b -center of T be k . Then $OPT(T, v) = k + OPT(T, w) = k + OPT(T(w, w'), w) = k + b(T)$, where w' is the node adjacent to w on the path $\mathcal{P}$ from v to w .*

Lemma 1 says that by OPT with a source v , the source message M is transmitted to w on $\mathcal{P}$ with no delay and then it is propagated in $T(w, w')$, the total time for which transfers are executed dominates the optimal broadcasting time.

2.1 Upper Bound

To give an upper bound for the competitiveness of broadcasting in trees, we investigate the competitive ratio of a broadcasting scheme σ_1 , described as follows: In σ_1 , each node u initially has the list $L_u = (u_1, \dots, u_k)$ defined by (1). Then, except a central node c , each node u modifies L_u by moving, in front of the list, its parent in T_c , the tree T rooted at c . Let $\tilde{\ell}_1(u)$ and $\ell_1(u)$ denote the first element of the initial list L_u and of the modified list of σ_1 , respectively, for every node u in T . Then either $\tilde{\ell}_1(u) = \ell_1(u)$ or $\tilde{\ell}_1(u)$ is the second element of the list of σ_1 .

In [3], it was shown that $B_{\sigma_1}(T) \leq \frac{3}{2} OPT(T)$. Here we give an upper bound of the competitive ratio of σ_1 .

Theorem 1. *Let T be a tree. For the broadcasting scheme σ_1 of T ,*

$$B_{\sigma_1}(T, s) \leq OPT(T, s) + r, \quad \text{for all sources } s,$$

where r is the radius of T .

Proof. Let s be a source in T and c be the central node of T in σ_1 . Assume s is not in the b -center of T . By Lemma 1, there is a node e in the b -center of T closest to s such that $OPT(T, s) = d(s, e) + OPT(T, e)$. Let $\mathcal{P} = x_0 x_1 \dots x_k$ be the path from s to e , where $x_0 = s$ and $x_k = e$. (It may be that $x_1 = e$ or $x_{k-1} = s$.) For $i = 1, \dots, k-1$, T_i denotes $T(x_i, x_{i-1}) \cap T(x_i, x_{i+1})$, and T_0 and T_k denote $T(x_0, x_1)$ and $T(x_k, x_{k-1})$, respectively. Then we can see that the tree T is decomposed of the subtrees T_i , $i = 0, \dots, k$. According to the location of c , there are three cases.

Case 1: c is in T_0 . While M is passed from x_0 to x_k , each node x_i in $\mathcal{P}$ has one delay, that is, the retransmission to x_{i-1} , since $c \in T_0$. For each $i = 1, \dots, k$, after M arrives at x_i , it is propagated in T_i . We consider T_i as the tree rooted at x_i . Since c is in $T(x_{i-1}, x_i)$, in the propagation of M in T_i , each node has only the retransmission to its parent in T_i , guaranteeing the broadcasting time $OPT(T_i, x_i)$. Thus all nodes in T_i are informed in $OPT(T, s) + d(x_0, x_i) + d_i$ steps, where d_i is the depth of T_i . Let $\bar{d}$ be the maximum of values $d(x_0, x_i) + d_i$, $i = 1, \dots, k$. Then all nodes in $\cup_{1 \leq i \leq k} T_i$ are informed in $OPT(T, s) + \bar{d}$.

Next, we consider the propagation of M in T_0 . Assume that $s \neq c$. Let $\mathcal{P}' = y_0 \dots y_h$ be the path from s to c , where $y_0 = s$ and $y_h = c$. Then T_0

is also decomposed of the subtrees T'_i , $i = 0, \dots, h$, where $T'_i = T(y_i, y_{i-1}) \cap T(y_i, y_{i+1})$, $i = 1, \dots, h-1$, $T'_0 = T(y_0, x_1) \cap T(y_0, y_1)$, and $T'_h = T(y_h, y_{h-1})$. Fix any node y_j , $j \neq h$. Since both c and e are out of T'_j , $\tilde{\ell}_1(y_j) = y_{j-1}$ and $\ell_1(y_j) = y_{j+1}$. Thus, before M is transferred to T'_j , it can be delayed for two steps at y_j . Similarly as before, all other nodes in T'_i , $i = 0, \dots, h-1$, and all nodes in T'_h have only one delay of M . Thus all nodes in T_0 are informed in $OPT(T, s) + \max\{\tilde{d}_1 + 1, \tilde{d}_2\}$, where $\tilde{d}_1$ is the maximum of depths of T'_i , $i = 1, \dots, h$, and $\tilde{d}_2$ the depth of T'_0 . If $s = c$, then all nodes in T_0 are informed in $OPT(T, s) + \tilde{d}_2$.

Consequently, the broadcasting is completed in $OPT(T, s) + r$ since $\bar{d}$ and $\max\{\tilde{d}_1 + 1, \tilde{d}_2\}$ are less than or equal to r .

The other cases in which c is in T_k or c is not in $T_0 \cup T_k$ have similar analyses, and when s is in the b_center , it is also similar. $\square$

Corollary 1. *The broadcasting scheme σ_1 is 2-competitive.*

In the following subsection, we will show that for k -ary trees, the competitive ratio of the broadcasting scheme σ_1 is $1 + \frac{1}{k}$, which matches the upper bound obtained in Theorem 1. Thus it states that the upper bound is tight.

2.2 Complete K -ary Trees

Here we are concerned with a special class of trees, namely, complete k -ary trees. For complete k -ary trees, the competitive ratio of the broadcasting scheme σ_1 is given, and another broadcasting scheme σ_2 which gives a better competitive ratio is proposed.

Proposition 1. *Let T_k be a complete k -ary tree. Then,*

$$\mathcal{C}(T_k, \sigma_1) = 1 + \frac{1}{k}.$$

Now, we describe another broadcasting scheme σ_2 , which has a better competitive ratio. Consider T_x , the tree T_k rooted at x . Let $D(\gamma) = \{v \in T_x : d(x, v) \geq \gamma\}$. The broadcasting scheme σ_2 is given as follows: Initially, each node has the same list as in σ_1 . Next, each node in $D(\gamma)$ moves the first node in its list to the last. Then, we choose $\gamma = \lfloor \alpha \rfloor$ such that

$$\alpha = \frac{k^2 d}{k^2 + 1 - 1/d},$$

where d is the depth of T_x . Here α is derived from the condition under which the ratio of the broadcasting time of σ_2 to the optimal broadcasting time at one neighbor of x is equal to that at any leaf of T_x . The details are included in the full paper.

Proposition 2. *Let T_k be a complete k -ary tree and d the depth of T_k . Then it is satisfied that*

$$\mathcal{C}(T_k, \sigma_2) \leq 1 + \frac{k}{k^2 + 1 - 1/d}.$$

2.3 Lower Bound

Let L be the set of all leaf nodes of T . We define a positive integer λ to be the distance between $BC(T)$ and L , that is, the minimum distance between u and v for all $u \in BC(T)$ and $v \in L$. Here the lower bound for the competitiveness of broadcasting in trees is given.

Theorem 2. *For any broadcasting scheme σ of a tree T , there is a node s_0 such that*

$$B_\sigma(T, s_0) \geq OPT(T, s_0) + \lambda.$$

Proof. Fix a broadcasting scheme σ . Let c be the center node of the star $BC(T)$. Choose a node v_1 which are last informed by σ from the source c . Then it is a leaf node of T . Let $\mathcal{P}_1$ be the path from v_1 to c and c_1 the node adjacent to c on $\mathcal{P}_1$. Assume c_1 is not in $BC(T)$. (Similarly, we can prove for the case c_1 is in $BC(T)$.) The σ can also be viewed as a broadcasting scheme in $T(c, c_1)$ and let v_2 be a node informed by σ from c in $T(c, c_1)$ at $B_\sigma(T(c, c_1), c)$ -th step. Then $\mathcal{P}_2$ denotes the path from c to v_2 (does not contain c) and c_2 the node adjacent to c on $\mathcal{P}_2$.

Consider the path $\mathcal{P}$ which consists of $\mathcal{P}_1$ and $\mathcal{P}_2$. We assign a label *Up* or *Down* to each node in $\mathcal{P}_1$ and $\mathcal{P}_2$. If a node v in $\mathcal{P}_1$ or $\mathcal{P}_2$, except c , has its parent in T_c as the first node of its list in σ , the label *Up* is assigned to v , where T_c is the tree T rooted at c . Otherwise, the label *Down* to v . If c_2 is prior to c_1 in the list of c in σ , *Up* is assigned to c and otherwise, *Down* to c . Let U_i and D_i be the set of nodes labeled by *Up* and *Down*, respectively, on the path $\mathcal{P}_i$, for $i = 1, 2$. If $|U_2| + |D_1| \geq |U_1| + |D_2|$, the adversary chooses v_1 as the source, and otherwise, she does v_2 . For the former case, until the source message M reaches v_2 by being transferred to c and propagated in $T(c, c_1)$, it is delayed for at least $|U_2| + |D_1| \geq \frac{1}{2}d(v_1, v_2) \geq \lambda$. Thus we get

$$\begin{aligned} B_\sigma(T, v_1) &\geq d(v_1, c) + B_\sigma(T(c, c_1), c) + \lambda \\ &\geq d(v_1, c) + OPT(T(c, c_1), c) + \lambda \geq OPT(T, v_1) + \lambda. \end{aligned}$$

In the later case, we can also get

$$\begin{aligned} B_\sigma(T, v_2) &\geq d(v_2, c) + B_\sigma(T, c) + \lambda \\ &\geq d(v_2, c) + OPT(T, c) + \lambda \geq OPT(T, v_2) + \lambda. \end{aligned}$$

□

Consider the line L_n of n (even) nodes. Then $\lambda = n/2 - 1$. As shown in the previous section, for the broadcasting scheme σ_0 , $B_{\sigma_0}(L_n, s) \leq OPT(L_n, s) + \lambda$, for all sources s . It says that the given bound in Theorem 2 is tight.

Let λ' be the maximum distance between u and v for all $u \in BC(T)$ and $v \in L$, that is, the maximum of all eccentricities of u in $BC(T)$.

Corollary 2. *Let T be a tree and D be the maximum degree of nodes in T . Then no broadcasting scheme of T can be better than $(1 + \frac{\lambda}{\lambda'D+1})$ -competitive.*

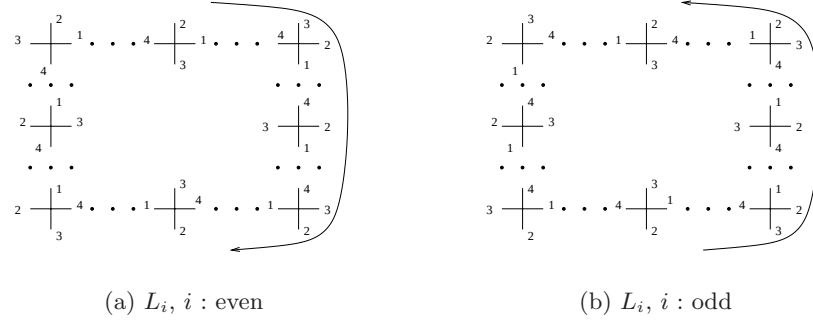


Fig. 2. Lists of nodes in a nondegenerate layer L_i in σ_3

3 Grids

In this section, we investigate the competitiveness of broadcasting in grids. First, it is shown that the competitive ratio of the broadcasting scheme given in [3], denoted by σ_3 , can be relatively large, and then a new broadcasting algorithm σ_4 is described, which gives an upper bound for the competitiveness. We will also provide a lower bound.

3.1 Broadcasting Scheme in [3]

Let $G_{m,n}$ be a grid with m rows and n columns, for $m \leq n$. Each node of $G_{m,n}$ is labeled with a coordinate (x, y) such that $1 \leq x \leq m$, $1 \leq y \leq n$, and $(1, 1)$ is the lower left-corner of $G_{m,n}$. Here we describe the broadcasting scheme σ_3 of $G_{m,n}$ given in [3]. The grid $G_{m,n}$ is partitioned into layers $L_1, \dots, L_{\lceil m/2 \rceil}$ defined as follows:

$$L_i = \{(x, y) : (x \in \{i, m-i+1\} \text{ and } i \leq y \leq n-i+1) \text{ or} \\ (i \leq x \leq m-i+1 \text{ and } y \in \{i, n-i+1\})\}.$$

A layer is called degenerate if all its x -coordinates are equal. Figure 2 and 3 show the lists of nodes in σ_3 . In [3], they showed that $B_{\sigma_3}(G_{m,n}) = m + n - 1 = \text{OPT}(G_{m,n}) + 1$. But we show that the broadcasting scheme σ_3 has a large competitive ratio.

Theorem 3. *Let $G_{m,n}$ be an $m \times n$ grid ($m \leq n$). For the broadcasting scheme σ_3 of $G_{m,n}$,*

$$B_{\sigma_3}(G_{m,n}, s) \leq \text{OPT}(G_{m,n}, s) + \lceil \frac{n}{2} \rceil, \quad \text{for all sources } s,$$

and there is a node s_0 such that

$$B_{\sigma_3}(G_{m,n}, s_0) \geq \text{OPT}(G_{m,n}, s_0) + \lceil \frac{m}{2} \rceil - 2.$$

By Theorem 3, for a grid $G_{n,n}$, the competitive ratio of σ_3 is (asymptotically) at least $\frac{3}{2}$. In the next subsection, we provide a new broadcasting scheme which is 1-competitive, that is, whose broadcasting time has only constant difference from the optimal for all sources.

3.2 New Broadcasting Scheme

We propose another broadcasting scheme σ_4 of $G_{m,n}$ with a better competitive ratio. First, choose integers α and β such that $m-1 = \alpha \cdot 4 + \beta$, where $0 \leq \beta < 4$ and $m \geq 6$. Let $\gamma = \lceil \frac{\alpha+\beta}{2} \rceil$. Then we consider horizontal lines H_{k+1} each of which consists of nodes with x -coordinate of $\nu_{k+1} = 1 + \gamma + k\alpha$, $k = 0, 1, 2, 3$, respectively, called *horizontal highways*. Also each vertical line consisting of nodes on a column is called a *vertical highway*.

Now, we describe a broadcasting scheme σ_4 . Given a node $u = (x, y)$ in $G_{m,n}$. First, we consider the node u such that $3 \leq y \leq n-2$. If u is on the horizontal highway H_1 or H_3 , then it first transmits a message into the node $(x, y+1)$, and has the list given in (1) (y : odd) or (2) (y : even) in Figure 4 (a). If u is on H_2 or H_4 , then it first transmits a message into $(x, y-1)$, and has the list given in (3) (y : odd) or (4) (y : even) in Figure 4 (a). However, if n is even, we have exceptional nodes χ_1 and χ_2 given by $(\nu_1, n-2)$ and $(\nu_4, 3)$, respectively. The nodes χ_1 and χ_2 have the list given in (4) and (2), respectively, in Figure 4 (b). If u is not on any horizontal highway and is on a vertical highway of odd column, i.e. y is odd, then it first sends a message to $(x+1, y)$, and has the list given in (1) ($\nu_2 < x < \nu_3$) or (2) (*o.w.*) in Figure 4 (b). If u is not on any horizontal highway and is on a vertical highway of even column, it first to $(x-1, y)$, and the list given in (3) ($\nu_2 < x < \nu_3$) or (4) (*o.w.*) in Figure 4 (b). Finally, if u is on the vertical highway of k -th column, $k = 1, 2, n-1, n$, then it has the list given in (1) (y : odd), (3) ($y = 2$), or (4) ($y = n-1$, n : odd) in Figure 4 (b).

Theorem 4. Let $G_{m,n}$ be an $m \times n$ grid ($n \geq m \geq 6$). For the broadcasting scheme σ_4 of $G_{m,n}$,

$$B_{\sigma_4}(G_{m,n}, s) \leq OPT(G_{m,n}, s) + 4, \quad \text{for all sources } s.$$

Proof. Let $s = (x, y)$ be a source. Assume m and n are even and $\frac{m}{2} + 1 \leq x \leq m$ and $1 \leq y \leq \frac{n}{2}$. Then we can see that $OPT(G_{m,n}, s) = n + x - y - 1$, which is the length of a shortest path from s to $f = (1, n)$. We define the regions $\mathcal{R}_k$ and $\mathcal{R}'_k$, $k = 0, \dots, 4$ in $G_{m,n}$ as follows: $\mathcal{R}_k = \{(v, w) : \nu_k \leq v \leq \nu_{k+1}, y < w \leq n\}$ and $\mathcal{R}'_k = \{(v, w) : \nu_k \leq v \leq \nu_{k+1}, 1 \leq w \leq y\}$, where $\nu_0 = 1$ and $\nu_5 = m$.

Suppose that s is on a vertical highway of odd column, that is, y is odd.

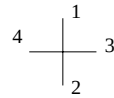
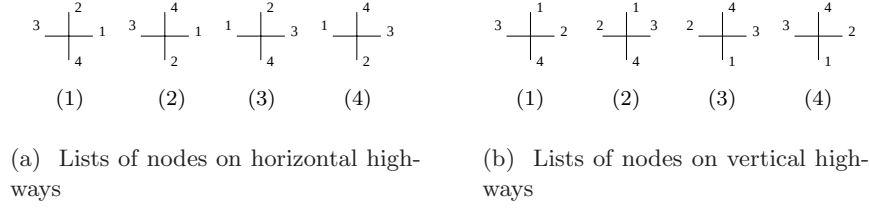
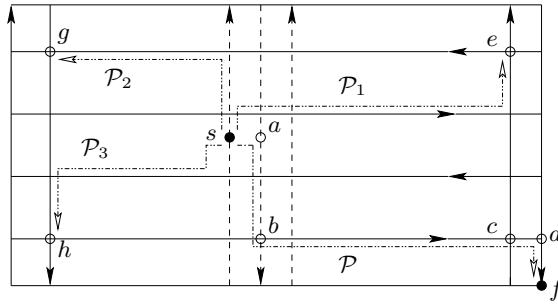


Fig. 3. List of nodes in the degenerate layer in σ_3

**Fig. 4.** Lists of nodes in σ_4

Case 1: $x < \nu_3$. Let $a = (x, y+1)$, $b = (\nu_0, y+1)$, $c = (\nu_0, n-1)$, and $d = (\nu_0, n)$. We consider the path $\mathcal{P}$ from s to f defined by $\mathcal{P} = s a b c d f$. (See Figure 5.) Imagine the source message M transferred through $\mathcal{P}$ by σ_4 . After one transfer to $(x+1, y)$ at s , M is sent to a , and it is transmitted to b with one delay on H_2 through the vertical highway of $(y+1)$ -th column. Then, M is moved to c with one delay at χ_1 through H_1 .

Let t be the number of steps after which M reaches c from s . Then $t = d(s, c) + 3$. We also consider the path $\mathcal{P}_1 = s (\nu_3, y) (\nu_3, n-1) e$, where $e = (\nu_4, n-1)$. After the source message M departs from s , it is transferred to e with no delay through $\mathcal{P}_1$. Since $d(s, e) \leq d(s, c)$, M can reach e in $t-3$ steps. During this transfer, all nodes on vertical highways of odd columns in $\mathcal{R}_3$ and of even columns in $\mathcal{R}_2$ except n -th column are informed. With two additional steps at $(\nu_3, n-1)$, that is, in $t-1$ steps, all nodes on the n -th column in $\mathcal{R}_2$ are informed. In fact, all nodes in $\mathcal{R}_2 \cup \mathcal{R}_3$ are informed in $t-1$ steps. Let $\mathcal{P}_2$ and $\mathcal{P}_3$ be the paths $s (\nu_4, y) g$ and $s (x, y-1) (\nu_2, y-1) (\nu_2, 2) h$, respectively, where $g = (\nu_4, 2)$ and $h = (\nu_1, 2)$. After M starts to move from s , it reaches g with two delays on H_3 and at χ_2 through $\mathcal{P}_2$ and h with two delays at s through $\mathcal{P}_3$. Since $d(s, g) \leq d(s, c) - 3$ and $d(s, h) \leq d(s, c) - 3$, we can see that M is transmitted to g and h in $t-4$ steps. Also during the transfer from s to h , all nodes on vertical highways of even columns in $\mathcal{R}'_1$ and of odd columns in $\mathcal{R}'_2$

**Fig. 5.** Broadcasting scheme σ_4 of $G_{m,n}$

except the first column are informed. With two additional steps at $(\nu_2, 2)$, that is, in $t - 2$ steps, all nodes on the first column in $\mathcal{R}'_2$ are informed, and so all nodes in $\mathcal{R}'_1 \cup \mathcal{R}'_2$ are informed in $t - 2$ steps.

Imagine the situation of message transmissions after γ steps, that is, at $t + \gamma$. Let $\mathcal{R}_{11}$ and $\mathcal{R}'_{31}$ be the set of nodes (v, w) with $\nu_1 \leq v \leq \nu_1 + \lceil \frac{\alpha}{2} \rceil$ and $\nu_3 \leq v \leq \nu_3 + \lceil \frac{\alpha}{2} \rceil$, respectively, and let $\mathcal{R}_{12} = \mathcal{R}_1 \setminus \mathcal{R}_{11}$ and $\mathcal{R}'_{32} = \mathcal{R}'_3 \setminus \mathcal{R}'_{31}$. Then all nodes on vertical highways of odd columns in $\mathcal{R}_{11}$ and $\mathcal{R}_{31}$ and of even columns in $\mathcal{R}_{12}$ and $\mathcal{R}_{32}$ are informed, because $\lceil \frac{\alpha}{2} \rceil \leq \gamma$. Also all nodes on vertical highways of odd columns in $\mathcal{R}_4$ and $\mathcal{R}'_4$ and of even columns in $\mathcal{R}'_0$ and $\mathcal{R}_0$ except n -th column in $\mathcal{R}_0$ are informed. Thus we can see that after two additional steps, that is, at $t + \gamma + 2$, all nodes in $G_{m,n}$ except on the n -th column in $\mathcal{R}_0$ are informed. Consider the nodes on n -th column. They are on the path $\mathcal{P}$. The source message M has reached c in t steps and then after one delay at c , it is transferred to f with no delay through $\mathcal{P}$. Therefore they are informed in $t + \gamma + 2$ steps. Consequently, the broadcasting is completed in $d(s, f) + 4$ steps, that is, $OPT(G_{m,n}, s) + 4$.

The case that $\nu_3 \leq x \leq m$ is omitted for the lack of space and the case in which s is on a vertical highway of even column has a similar analysis. Also in each case in which m or n is odd, the analysis is slightly different but the arguments are very similar. The details are included in the full paper. $\square$

3.3 Lower Bound

Here we give a lower bound of the competitiveness of broadcasting in grids.

Theorem 5. *For any broadcasting scheme σ of an $m \times n$ grid $G_{m,n}$, there is a node s_0 such that*

$$B_\sigma(G_{m,n}, s_0) \geq OPT(G_{m,n}, s_0) + 2.$$

4 Complete Graphs

Let G be a symmetric graph and σ be a broadcasting scheme in G . If the competitive ratio of σ is given by c , then $\max_s B_\sigma(G, s) = c \cdot OPT(G)$ since $OPT(G, s) = OPT(G), \forall s$. Also since $B(G) \leq \max_s B_\sigma(G, s)$, it is obtained that $B(G) \leq c \cdot OPT(G)$.

For a complete graph K_n of n nodes, it was shown in [3] that $B(K_n) \leq \lceil \log n \rceil + 2\lfloor \sqrt{\log n} \rfloor$. In this section, we improve the upper bound of $B(K_n)$ by proving that $\max_s B_{\sigma_5}(K_n, s) = \log n + 2 \log \log n + 3$, for a broadcasting scheme σ_5 .

First, we construct a multi-level broadcasting tree $\mathcal{T}$ with a specific node x as a root, which contains a minimum broadcasting tree in each level. (Here a minimum broadcasting tree with the root r refers to a tree obtained by an optimal broadcasting at the source r .) Initially, we pick one node as x and $\log n - 1$ of other nodes from which a minimum broadcasting tree, denoted by

$\mathcal{T}_x^1$, is constructed with the root x . That is, if the source message is given at x , then on $\mathcal{T}_x^1$, the broadcasting for the $\log n$ nodes is completed in $\log \log n$ steps. This tree $\mathcal{T}_x^1$ is called the level-1 subtree of $\mathcal{T}$ and all nodes in it are said to be of level-1 in $\mathcal{T}$. For each node u of level-1, except x , we choose $\log n - 1$ other nodes of K_n and construct a minimum broadcasting tree of these nodes with u as a root, denoted by $\mathcal{T}_u^2$. It is said that all nodes in these subtrees are of level-2 in $\mathcal{T}$. Continuing this construction, we get a multi-level tree $\mathcal{T}$ with $\frac{\log n}{\log \log n}$ levels. Then we see that each node of level- i belongs to a minimum broadcasting tree with, as a root, a node of level- $(i-1)$ and it is itself a root of a minimum broadcasting tree consisting of nodes of level- $(i+1)$. In the next lemma, we show that this tree $\mathcal{T}$ contains nodes of K_n more than $\frac{n}{2}$.

Lemma 2. *The multi-level tree $\mathcal{T}$ constructed in the above, has at least $\frac{n}{2}$ nodes ($n \geq 8$).*

Here, a broadcasting scheme σ_5 of K_n is described. By Lemma 2, we can assign a node of $\mathcal{T}$ to each node in $K_n \setminus \mathcal{T}$ and the node assigned a node u in $\mathcal{T}$ is called the twin node of u .

In σ_5 , a node u belonging to a level- i subtree $\mathcal{T}_v^i$ first passes a message M on $\mathcal{T}_v^i$, and then on $\mathcal{T}_u^{i+1}$. After finishing the transmissions on $\mathcal{T}_v^i$ and $\mathcal{T}_u^{i+1}$, a nonleaf node u transmits M to its twin node if there it is, and a leaf node u first sends M to the root x of $\mathcal{T}$ and then to its twin. Note the leaf node u has no transmissions on $\mathcal{T}_v^i$.

Theorem 6. *Let K_n be a complete graph of n nodes ($n \geq 8$). For the broadcasting scheme σ_5 of K_n ,*

$$\max_s B_{\sigma_5}(K_n, s) = OPT(K_n) + 2 \log \log n + 3.$$

5 Hypercubes

In this section, we give a simple broadcasting scheme σ_6 in hypercubes. The hypercube of dimension n is given by $H_n = \{v = (v_1, \dots, v_n) : v_i = 0 \text{ or } 1\}$. Let $H_{n-1}^0 = \{v : v \in H_n \text{ and } v_n = 0\}$ and $H_{n-1}^1 = \{v : v \in H_n \text{ and } v_n = 1\}$. Then H_{n-1}^0 and H_{n-1}^1 are hypercubes of dimension $n-1$. Consider a binomial subtree $\mathcal{T}$ of H_{n-1}^0 obtained by OPT with the source $0 = (0, \dots, 0)$, that is, from the source 0 , the broadcasting is completed on $\mathcal{T}$ in $n-1$ steps. Then leaf nodes of $\mathcal{T}$ form a hypercube of dimension $n-2$ which has a binomial subtree $\mathcal{T}'$ with the root $1' = (1, \dots, 1, 0)$.

Now, we describe a broadcasting scheme σ_6 of H_n . For each node $v = (w, 0)$ in H_{n-1}^0 , if v is a nonleaf node of $\mathcal{T}$, it first passes a message M through $\mathcal{T}$ like OPT with source 0 and after finishing the transmissions on $\mathcal{T}$, it sends M to the node $(w, 1)$ in H_{n-1}^1 . If v is a leaf node of $\mathcal{T}$, it has only two transmissions, the first to its parent in $\mathcal{T}'$ and the second to $(w, 1)$. For each node v in H_{n-1}^1 , if v has a list $L_v = (v_1, \dots, v_k)$ in σ_6 , then $\bar{v}$ in H_{n-1}^0 has the list $L_{\bar{v}} = (\bar{v}_1, \dots, \bar{v}_k)$.

Theorem 7. *Let H_n be a hypercube of dimension n . For the broadcasting scheme σ_6 of H_n ,*

$$\max_s B_{\sigma_6}(H_n, s) = 2 \cdot \text{OPT}(H_n).$$

6 Concluding Remarks

We proposed a new analysis for the problem of broadcasting with universal lists, and the competitivenesses of broadcasting in specific graphs, i.e., trees, grids, complete graphs, and hypercubes were investigated. For trees and grids, there is a gap between the lower and upper bound. For complete graphs and hypercubes, the competitive ratio of the broadcasting scheme gives an upper bound of the broadcasting time. It was given as open problems in [3] to find the broadcasting time for complete graphs and other important graphs: hypercubes, CCC, de Bruijn graphs, etc. We gave answers for complete graphs and hypercubes. For hypercubes, we tried to design a broadcasting scheme using the binary encoding of the nodes, but failed to find a better scheme than the simple broadcasting scheme.

References

1. M. Chlebus, L. Gasieniec, A. Ostlin, and J. M. Robson. Deterministic broadcasting in radio networks. In *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP'00)*, pages 717–728, 2000. 75
2. M. Chrobak, L. Gasieniec, and W. Rytter. Fast broadcasting and gossiping in radio networks. In *Proc. 41st Annual IEEE Conference on Foundations of Computer Science (FOCS'00)*, pages 575–581, 2000. 75
3. K. Diks and A. Pelc. Broadcasting with universal lists. *Networks*, 27:183–196, 1996. 74, 75, 77, 80, 83, 85
4. Amos Fiat and Gerhard J. Woeginger. *Online Algorithms: The State of the Art*. Springer-Verlag, 1998. 74, 75
5. S. M. Hedetniemi, S. T. Hedetniemi, and A. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988. 74
6. E. J. Cockayne P. J. Slater and S. T. Hedetniemi. Information dissemination in trees. *SIAM J. Comput.*, 10:692–701, 1981. 76, 77
7. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985. 74, 75
8. Douglas B. West. *Introduction to Graph Theory*. Prentice-Hall, 1996. 76

On Adaptive Fault Diagnosis for Multiprocessor Systems

Kumiko Nomura, Toshinori Yamada, and Shuichi Ueno

Dept. of Communications and Integrated Systems,
Graduate School of Science and Engineering
Tokyo Institute of Technology, Tokyo 152-8552, Japan
{nomura,yamada,ueno}@lab.ss.titech.ac.jp

Abstract. We first consider adaptive serial diagnosis for multiprocessor systems. We present an adaptive diagnosis algorithm using $N + t - 1$ tests, which is the smallest possible number, for an N -processor system modeled by a $(2t - 1)$ -connected graph with at most t faulty processors. We also present an adaptive diagnosis algorithm using minimum number of tests for a system modeled by cube-connected cycles. We consider adaptive parallel diagnosis as well. We show that for adaptive parallel diagnosis of an N -processor system modeled by a hypercube, three testing rounds are necessary and sufficient if the number of faulty processors is at most $\log N - \lceil \log(\log N - \lceil \log \log N \rceil + 4) \rceil + 2$. We also show that three testing rounds are necessary and sufficient for adaptive parallel diagnosis of a system modeled by cube-connected cycles of dimension greater than three.

1 Introduction

The system diagnosis has been extensively studied in the literature in connection with fault-tolerant multiprocessor systems. An original graph-theoretical model for system diagnosis was introduced in a classic paper by Preparata, Metze, and Chien [16]. In this model, each processor is either faulty or fault-free. The fault-status of a processor does not change during the diagnosis. The processors can test each other only along communication links. A testing processor evaluates a tested processor as either faulty or fault-free. The evaluation is accurate if the testing processor is fault-free, while the evaluation is unreliable if the testing processor is faulty. The system diagnosis is to identify all faulty processors based on test results.

A system is t -diagnosable if all faulty processors can always be identified provided that the number of faulty processors does not exceed t . It is well-known that a system with N processors is t -diagnosable only if $t < N/2$ and each processor is connected with at least t distinct other processors by communication links [16]. A complete characterization of t -diagnosable system was shown by Hakimi and Amin [9]. The original model is nonadaptive in the sense that all tests must be determined in advance. It can be shown that each processor must be tested by at least t distinct other processors in nonadaptive diagnosis if as many

as t processors may be faulty. It follows that at least tN tests are necessary for nonadaptive diagnosis of an N -processor system with at most t faulty processors.

In adaptive diagnosis introduced by Nakajima [15], tests can be determined dynamically depending on previous test results. Blecher [7] and Wu [17] showed that $N + t - 1$ tests are sufficient for adaptive diagnosis of an N -processor system with at most t faulty processors if the system is modeled by a complete graph and $t < N/2$. Moreover, Blecher [7] showed that $N + t - 1$ is also the lower bound for the number of tests in the worst case. The adaptive diagnosis of some practical systems modeled by sparse graphs has been considered in the literature [4,5,6,8,12,13,14]. Among others, Kranakis, Pelc, and Spatharis [14] showed adaptive diagnosis algorithms using minimum number of tests in the worst case for systems modeled by trees, cycles, and tori. Björklund [6] showed an adaptive diagnosis algorithm for an N -processor system modeled by a hypercube with at most t faulty processors. The algorithm uses $N + t - 1$ tests if $t = \log N$, and $N + t$ tests if $t < \log N$.

This paper shows an adaptive diagnosis algorithm using minimum number of tests for systems modeled by cube-connected cycles. We also show an adaptive diagnosis algorithm using $N + t - 1$ tests for an N -processor system modeled by a $(2t - 1)$ -connected graph with at most t faulty processors. This is an extension of a previous result on systems modeled by complete graphs in the sense that an N -vertex complete graph K_N is $(2t - 1)$ -connected if $t < N/2$. Notice that our algorithm uses $N + t - 1$ tests for an N -processor system modeled by a hypercube with at most t faulty processors if $t \leq (\log N + 1)/2$, since an N -vertex hypercube is $\log N$ -connected.

The adaptive parallel diagnosis has been considered as well in the literature [1,2,3,6,11,13]. In adaptive parallel diagnosis, each processor may participate in at most one test, either as a testing or tested processor, in each testing round. Beigrl, Hurwood, and Kahale [1] showed that for adaptive parallel diagnosis of an N -processor system modeled by K_N with at most t faulty processors, 4 testing rounds are necessary and sufficient if $2\sqrt{2N} \leq t \leq 0.03N$, 5 testing rounds are necessary if $t \geq 0.49N$, and 10 testing rounds are sufficient if $t < N/2$. Since at least $N + t - 1$ tests are necessary for adaptive parallel diagnosis of an N -processor system with at most t faulty processors and there are at most $N/2$ tests in each testing round, $\lceil (N + t - 1)/(N/2) \rceil$, which is 3 if $t \geq 2$, is a general lower bound for the number of testing rounds [2]. Björklund [6] showed that 4 testing rounds are sufficient for adaptive parallel diagnosis of an N -processor system modeled by a hypercube with at most $\log N$ faulty processors. It is still open whether 3 testing rounds are sufficient for such systems, as mentioned in [6].

We partially answer the question above by showing that for adaptive parallel diagnosis of an N -processor system modeled by a hypercube, 3 testing rounds are necessary and sufficient if the number of faulty processors is at most $\log N - \lceil \log(\log N - \lceil \log \log N \rceil + 4) \rceil + 2$. We also show that 3 testing rounds are necessary and sufficient for adaptive parallel diagnosis of systems modeled by cube-connected cycles of dimension greater than 3.

2 Preliminaries

A multiprocessor system is modeled by a graph in which the vertices represent processors and edges represent communication links. Each vertex is either faulty or fault-free. A pair of adjacent vertices can test each other. A test performed by u on v is represented by an ordered pair $\langle u, v \rangle$. The outcome of a test $\langle u, v \rangle$ is 1(0) if u evaluates v as faulty(fault-free). The outcome is accurate if u is fault-free, while the outcome is unreliable if u is faulty. A graph is t -diagnosable if all faulty vertices can always be identified from test results provided that the number of faulty vertices is not more than t . If an N -vertex graph G is t -diagnosable then $t < N/2$ and the minimum degree of a vertex is at least t [16].

We denote the vertex set and edge set of a graph G by $V(G)$ and $E(G)$, respectively. For $S \subseteq V(G)$, $G - S$ is the graph obtained from G by deleting the vertices in S . For a positive integer k , a graph G is said to be k -connected if $G - S$ is connected for any $S \subseteq V(G)$ with $|S| \leq k - 1$. A graph is said to be k' -connected for any integer $k' \leq 0$ for convenience. We denote a cycle, path and complete graph with N vertices by C_N , P_N , and K_N , respectively. C_N is called an even cycle if N is even, and odd cycle otherwise. The product of graphs G and H is a graph $G \times H$ with vertex set $V(G) \times V(H)$, in which (u, v) is adjacent to (u', v') if and only if either $u = u'$ and $(v, v') \in E(H)$ or $v = v'$ and $(u, u') \in E(G)$.

An n -dimensional cube $Q(n)$ is recursively defined as follows: $Q(1) = P_2$; $Q(n) = Q(n-1) \times P_2$. It follows that $Q(n) = Q(p) \times Q(q)$ for any positive integers p and q such that $p + q = n$. $Q(n)$ has 2^n vertices, and the degree of a vertex is n .

The n -dimensional cube-connected cycles(CCC) is constructed from $Q(n)$ by replacing each vertex of $Q(n)$ with C_n in CCC. For any positive integer k , $[k]$ denotes $\{0, 1, \dots, k-1\}$. For any positive integer n and $\mathbf{x} = x_{n-1}x_{n-2} \cdots x_0 \in [2]^n$ and $i \in [n]$, let $\chi_i(\mathbf{x}) = x_{n-1} \cdots x_{i+1} \overline{x_i} x_{i-1} \cdots x_0$, where $\overline{x_i} = 1 - x_i$, that is the complement of x_i . The n -dimensional CCC, denoted by $CCC(n)$, is the graph defined as follows:

$$\begin{aligned} V(CCC(n)) &= [2]^n \times [n]; \\ E(CCC(n)) &= \{([\mathbf{x}, i], [\chi_i(\mathbf{x}), i]) : i \in [n]\} \cup \{([\mathbf{x}, i], [\mathbf{x}, j]) : j = (i \pm 1) \bmod n\}. \end{aligned}$$

$CCC(n)$ has $n2^n$ vertices, and the degree of a vertex is 3.

3 Adaptive Diagnosis

In nonadaptive diagnosis, all tests are scheduled in advance. It is known that at least tN tests are necessary for nonadaptive diagnosis of an N -vertex graph with at most t faulty vertices [16].

In adaptive diagnosis, tests can be determined dynamically depending on previous test results. The following theorem shows a general lower bound for the number of tests necessary to adaptively diagnose a graph.

```

function Expand( $G, H_0, F_0, t$ )
begin
   $H \leftarrow H_0; F' \leftarrow F_0;$ 
  while  $H \cup F' \neq V(G)$  and  $|F'| < t$  do
    begin
      Select any  $v \in V(G) - (H \cup F')$  s.t.  $(u, v) \in E(G)$  for some  $u \in H$ ;
      if outcome of test  $\langle u, v \rangle$  is 0 then  $H \leftarrow H \cup \{v\}$ 
      else  $F' \leftarrow F' \cup \{v\};$ 
    end
  return( $F'$ )
end

```

Fig. 1. Function Expand

Theorem I [7] *If G is an N -vertex graph with at most t faulty vertices then $N + t - 1$ tests are necessary to adaptively diagnose G in the worst case.*

The following theorem shows upper bounds for the number of tests sufficient to adaptively diagnose hypercubes.

Theorem II [6] *$Q(n)$ is adaptively t -diagnosable using at most $N - t + 1$ tests if $t = n$, and using at most $N + t$ tests if $t < n$, where $N = 2^n$ is the number of vertices in $Q(n)$.*

3.1 $(2t - 1)$ -Connected Graphs

In this section, we prove the following theorem.

Theorem 1. *Let G be an N -vertex graph and t be a positive integer. If G is $(2t - 1)$ -connected and $t < N/2$ then G is adaptively t -diagnosable using at most $N + t - 1$ tests.*

Since K_N is $(N - 1)$ -connected and $Q(n)$ is n -connected, we have the following corollaries:

Corollary I [7,17] *K_N is adaptively t -diagnosable using at most $N + t - 1$ tests if $t < N/2$. $\square$*

Corollary 1. *$Q(n)$ is adaptively t -diagnosable using at most $N + t - 1$ tests if $t \leq (n + 1)/2$ and $n \geq 2$, where $N = 2^n$ is the number of vertices in $Q(n)$. $\square$*

3.1.1 Proof of Theorem 1

We need a preliminary result.

Lemma 1. *Let G be a t -connected graph, and F be a set of all faulty vertices with $|F| \leq t$. If $H_0 \subseteq V(G) - F$, $H_0 \neq \emptyset$, and $F_0 \subseteq F$ then Function Expand shown in Fig. 1 identifies F using at most $|V(G)| - |H_0 \cup F_0|$ tests.*

Proof. We prove the lemma by a series of claims.

Claim 1. *If $H \cup F' \neq V(G)$, $H \neq \emptyset$, and $|F'| < t$ then there is a vertex $v \in V(G) - (H \cup F')$ such that $(u, v) \in E(G)$ for some vertex $u \in H$.*

Proof (of Claim 1). Since $|F'| < t$ and G is t -connected, $G - F'$ is connected. Since $V(G) - (H \cup F') \neq \emptyset$ and $H \neq \emptyset$, there is a vertex $v \in V(G) - (H \cup F')$ such that $(u, v) \in E(G)$ for some vertex $u \in H$. $\square$

The following claim is obvious.

Claim 2. $H \subseteq V(G) - F$, $H \neq \emptyset$, and $F' \subseteq F$. $\square$

Claim 3. *If $H \cup F' = V(G)$ or $|F'| = t$ then $F = F'$.*

Proof (of Claim 3). If $F \neq F'$ then we conclude by Claim 2 that $H \cup F' \neq V(G)$ and $|F'| < |F| \leq t$, which is a contradiction. Hence, $F = F'$. $\square$

By Claims 1 and 3, Function Expand identifies F . Since each vertex of $V(G) - (H_0 \cup F_0)$ is tested at most once, Function Expand uses at most $|V(G)| - |H_0 \cup F_0|$ tests. $\square$

Now we are ready to prove Theorem 1. Let G be a $(2t - 1)$ -connected graph, and F be a set of all faulty vertices with $|F| \leq t$. We prove the theorem by induction on t .

Since we can identify $F = \emptyset$ correctly with no test, the theorem holds for $t = 0$.

Let t be a positive integer. For inductive step, assume that the theorem holds for any non-negative integer $t' < t$. Select any $v \in V(G)$. Let $u_1, u_2, \dots, u_k$ be the vertices adjacent to v . We perform a sequence of tests $\langle u_1, v \rangle, \langle u_2, v \rangle, \dots, \langle u_k, v \rangle$, and add u_i to T_j if the outcome of test $\langle u_i, v \rangle$ is j ($j = 0, 1$) until either of the following two events occurs: (i) $|T_0| = t$; (ii) $|T_1| = |T_0| + 1$. It should be noted that $k \geq 2t - 1$ because G is $(2t - 1)$ -connected. Thus, either of (i) and (ii) always occurs. It is easy to see the following:

Claim 4. $T_1 \subseteq F$ if v is fault-free, and $T_0 \cup \{v\} \subseteq F$ otherwise. $\square$

We distinguish two cases.

(i) $|T_0| = t$: Since $|T_0 \cup \{v\}| = t + 1$ and $|F| \leq t$, v is fault-free and $T_1 \subseteq F$ by Claim 4. Hence, by Lemma 1, $\text{Expand}(G, \{v\}, T_1, t)$ identifies F . The total number of tests performed is at most $|T_0| + |T_1| + (N - |T_1| - 1) = N + t - 1$.

(ii) $|T_1| = |T_0| + 1$: Let $s = |T_1|$ and $G' = G - (T_0 \cup T_1 \cup \{v\})$. By Claim 4, there exists at least s faulty vertices in $T_0 \cup T_1 \cup \{v\}$, and so G' has at most $t - s$ faulty vertices. It should be noted that $|T_0 \cup T_1 \cup \{v\}| = 2s$. Since G is $(2t - 1)$ -connected and $|V(G)| = N \geq 2t + 1$, G' is $(2(t - s) - 1)$ -connected and $|V(G')| = N - 2s \geq 2(t - s) + 1$. Thus, by inductive hypothesis, we can identify $F \cap V(G')$ using at most $(N - 2s) + (t - s) - 1 = N + t - 3s - 1$ tests. Let $H' = V(G') - F$. We further distinguish two cases.

(ii)-(a) $H' \cap \{u_1, u_2, \dots, u_k\} \neq \emptyset$: Let $u \in H' \cap \{u_1, u_2, \dots, u_k\}$. If the outcome of test $\langle u, v \rangle$ is 0 then v is fault-free, and so $T_1 \subseteq F$ by Claim 4. Thus, by Lemma 1, $\text{Expand}(G, H' \cup \{v\}, (F \cap V(G')) \cup T_1, t)$ identifies F . The total number

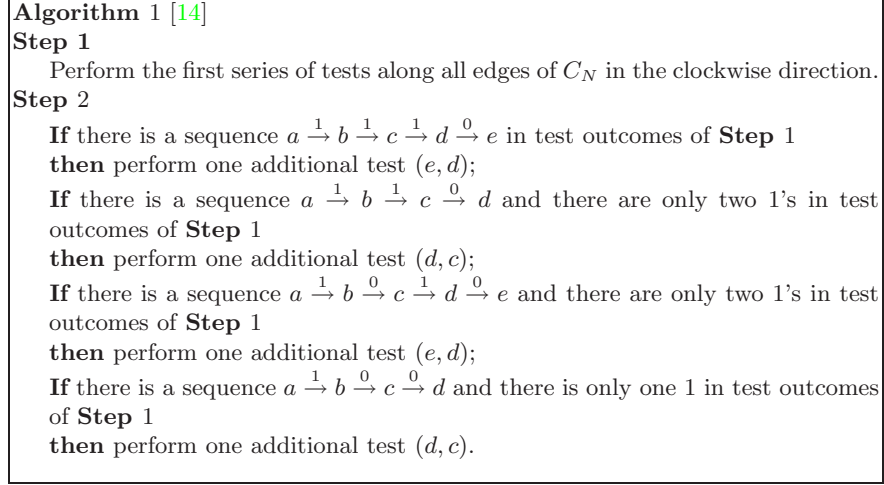


Fig. 2. Algorithm 1

of tests performed is at most $(2s-1) + (N+t-3s-1) + 1 + (s-1) = N+t-2$. If the outcome of test $\langle u, v \rangle$ is 1 then v is faulty, and so $T_0 \cup \{v\} \subseteq F$ by Claim 4. Thus, by Lemma 1, $\text{Expand}(G, H', (F \cap V(G')) \cup T_0 \cup \{v\}, t)$ identifies F . The total number of tests performed is at most $(2s-1) + (N+t-3s-1) + 1 + s = N+t-1$.

(ii)-(b) $H' \cap \{u_1, u_2, \dots, u_k\} = \emptyset$: Since $2t-1 \leq |T_0| + |T_1| + |F \cap V(G')| \leq t + s - 1$, we have $s \geq t$. On the other hand, $|T_0| \leq t-1$, and so we have $s = |T_1| = |T_0| + 1 \leq t$. Thus, we conclude that $s = t$. Since $|T_0 \cup \{v\}| = |T_1| = t$, we have by Claim 4 that $F = T_0 \cup \{v\}$ or $F = T_1$. Notice that $H' = V(G')$ and $F \cap V(G') = \emptyset$. Thus, $T_0 \cup T_1 = \{u_1, u_2, \dots, u_k\}$. Since G is $(2t-1)$ -connected and $|T_0| = t-1 \leq 2(t-1)$, $G - T_0$ is connected, and so there exists some vertex $w \in T_1$ such that $(x, w) \in E(G)$ for some $x \in V(G') = H'$. If the outcome of test $\langle x, w \rangle$ is 0 then w is fault-free, and so we conclude that $F = T_0 \cup \{v\}$. If the outcome of test $\langle x, w \rangle$ is 1 then w is faulty, and so we conclude that $F = T_1$. Hence, we can identify F using at most $(2t-1) + (N-2t-1) + 1 \leq N+t-1$ tests.

3.2 Cycles

We will use the following results on cycles proved in [14].

Theorem III [14] *Algorithm 1 shown in Fig. 2 adaptively diagnoses C_N using at most $N+1$ test if the number of faults is at most 2 and $N \geq 5$.*

3.3 CCC's

Theorem 2. *$CCC(n)$ is adaptively 3-diagnosable using at most $N+2$ tests if $n \geq 4$, where $N = n2^n$ is the number of vertices in $CCC(n)$.*

Proof. Suppose $n \geq 4$ and $F \subseteq V(CCC(n))$ is a set of all faulty vertices with $|F| \leq 3$.

Let $p = \lfloor n/2 \rfloor$ and $q = n - p (= \lceil n/2 \rceil)$. For any $k \in [2^{n-1}]$, set $m_k = 4p$ if $k < 2^{n-2}$, and $m_k = 4q$ otherwise. Notice that $m_k \geq 8$ since $n \geq 4$. For any $k \in [2^{n-1}]$ and any $i \in [m_k]$, define $v_{k,i}$ as follows: If $k < 2^{n-2}$ then

$$v_{k,i} = \begin{cases} [\mathbf{b}_1 \cdot 0 \cdot \mathbf{b}_0 \cdot 0, i] & \text{if } i < p, \\ [\mathbf{b}_1 \cdot 1 \cdot \mathbf{b}_0 \cdot 0, 2p - 1 - i] & \text{if } p \leq i < 2p, \\ [\mathbf{b}_1 \cdot 1 \cdot \mathbf{b}_0 \cdot 1, i - 2p] & \text{if } 2p \leq i < 3p, \\ [\mathbf{b}_1 \cdot 0 \cdot \mathbf{b}_0 \cdot 1, 4p - 1 - i] & \text{if } 3p \leq i, \end{cases}$$

where $\mathbf{b}_1 \in [2]^q$ and $\mathbf{b}_0 \in [2]^{p-2}$ are the q most and $p-2$ least significant bits of the $(n-2)$ -bit binary representation of k , respectively, and $\mathbf{a} \cdot \mathbf{b}$ denotes the concatenation of $\mathbf{a}$ and $\mathbf{b}$; If $k \geq 2^{n-2}$ then

$$v_{k,i} = \begin{cases} [0 \cdot \mathbf{b}'_1 \cdot 0 \cdot \mathbf{b}'_0, i + p] & \text{if } i < q, \\ [1 \cdot \mathbf{b}'_1 \cdot 0 \cdot \mathbf{b}'_0, n - 1 + q - i] & \text{if } q \leq i < 2q, \\ [1 \cdot \mathbf{b}'_1 \cdot 1 \cdot \mathbf{b}'_0, i + p - 2q] & \text{if } 2q \leq i < 3q, \\ [0 \cdot \mathbf{b}'_1 \cdot 1 \cdot \mathbf{b}'_0, n - 1 + 3q - i] & \text{if } 3q \leq i, \end{cases}$$

where $\mathbf{b}'_1 \in [2]^{q-2}$ and $\mathbf{b}'_0 \in [2]^p$ are the $q-2$ most and p least significant bits of the $(n-2)$ -bit binary representation of $k - 2^{n-2}$, respectively. Define that if $k < 2^{n-2}$ then

$$\overline{v_{k,i}} = \begin{cases} [\mathbf{x}, n - 1] & \text{if } j = 0, \\ [\mathbf{x}, p] & \text{if } j = p - 1, \\ [\chi_j(\mathbf{x}), j] & \text{otherwise,} \end{cases}$$

and if $k \geq 2^{n-2}$ then

$$\overline{v_{k,i}} = \begin{cases} [\mathbf{x}, p - 1] & \text{if } j = p, \\ [\mathbf{x}, 0] & \text{if } j = n - 1, \\ [\chi_j(\mathbf{x}), j] & \text{otherwise,} \end{cases}$$

where $v_{k,i} = [\mathbf{x}, j]$. For any $k \in [2^{n-1}]$, let $V_k = \{v_{k,i} : i \in [m_k]\}$. It is easy to see the following claims:

Claim 5. $(V_0, \dots, V_{2^{n-1}-1})$ is a partition of $V(CCC(n))$. $\square$

Claim 6. $(v_{k,i}, \overline{v_{k,i}}) \in E(CCC(n))$ and $\overline{v_{k,i}} \in V(CCC(n)) - V_k$ for any $k \in [2^{n-1}]$. $\square$

Claim 7. The subgraph of $CCC(n)$ induced by V_k is isomorphic to a cycle C_{m_k} for any $k \in [2^{n-1}]$. In particular, $(v_{k,i}, v_{k,(i \pm 1) \bmod m_k}) \in E(CCC(n))$. $\square$

Let $E_k = \{(v_{k,i}, v_{k,(i+1) \bmod m_k}) : i \in [m_k]\}$ for any $k \in [2^{n-1}]$ and any $i \in [m_k]$. For each $k \in [2^{n-1}]$, perform test $\langle v_{k,i}, v_{k,(i+1) \bmod m_k} \rangle$ in order of $i = 0, 1, \dots, m_k - 1$ until the outcome of test $\langle v_{k,i}, v_{k,(i+1) \bmod m_k} \rangle$ is 1 for some i or we have m_k tests. Let $X = \{(v_{k,i}, v_{k,(i+1) \bmod m_k}) : \text{the outcome of test } \langle v_{k,i}, v_{k,(i+1) \bmod m_k} \rangle \text{ is } 1\}$. Then, it is easy to see the followings:

Claim 8. $|E_k \cap X| \leq 1$ for any $k \in [2^{n-1}]$. $\square$

Claim 9. If $E_k \cap X = \emptyset$ then every vertex of V_k is fault-free. $\square$

Claim 10. If $(v_{k,i}, v_{k,(i+1) \bmod m_k}) \in X$, at least one of $v_{k,i}$ and $v_{k,(i+1) \bmod m_k}$ is faulty. $\square$

We have $|X| \leq |F| \leq 3$ by Claims 8 and 10. There are four cases.

(i) $|X| = 3$: Let Y denote the set of vertices incident with an edge in X . Every vertex of $V(CCC(n)) - Y$ is fault-free since Y has three faulty vertices by Claims 8 and 10. If $(v_{k,i}, v_{k,(i+1) \bmod m_k}) \in X$ then one of $v_{k,i}$ and $v_{k,(i+1) \bmod m_k}$ is faulty and the other is fault-free by Claim 10. If $i \geq 1$ then $v_{k,i}$ is fault-free, for otherwise $|F| \geq |\{v_{k,0}, v_{k,1}, \dots, v_{k,i}\}| + |X - (v_{k,i}, v_{k,(i+1) \bmod m_k})| \geq 4$, which is a contradiction. Since $v_{k,i}$ is fault-free, $v_{k,(i+1) \bmod m_k}$ is faulty. If $i = 0$ then test $v_{k,0}$ by $v_{k,n-1} \in V(CCC(n)) - Y$. If the outcome of test $\langle v_{k,n-1}, v_{k,0} \rangle$ is 1 then $v_{k,0}$ is faulty, and otherwise $v_{k,1}$ is faulty. Hence, we can identify F using at most $N = n \times 2^n$ tests.

(ii) $|X| = 2$: If $E_k \cap X = \emptyset$ then every vertex of V_k is fault-free by Claim 9. Thus, we can diagnose V_k with m_k tests. If $E_k \cap X \neq \emptyset$ then $|E_k \cap X| = 1$ by Claim 8, and so $|X - E_k| = 1$. It follows that V_k has at most two faulty vertices by Claim 10. Thus, from Claim 7 and the fact that $m_k \geq 8$, we can diagnose all vertices of V_k by applying Algorithm 1 for C_{m_k} . Notice that if $(v_{k,i}, v_{k,(i+1) \bmod m_k}) \in E_k \cap X$ then it suffices for Algorithm 1 to perform at most $(m_k - i)$ additional tests in order to diagnose V_k , since the outcome of $i + 1$ tests $\langle v_{k,i}, v_{k,(i+1) \bmod m_k} \rangle (j \in [i + 1])$ can be used to diagnose V_k . Thus, we can diagnose V_k with at most $m_k + 1$ tests. Since $|\{k : E_k \cap X\}| = |X| = 2$, we can identify F with at most $\sum_{i \in [2^{n-1}]} m_k + |X| = N + 2$ tests.

(iii) $|X| = 1$: Let $u_{k,i} \in X$ for some $k \in [2^{n-1}]$ and $i \in [m_k]$. Then, every vertex of $V(CCC(n)) - V_k$ is fault-free. We further distinguish three cases.

(iii)-(a) $i = 0$ or $i = 1$: We can identify F by testing $v_{k,j}$ by $\overline{v_{k,j}}$ for every $j \in [m_k]$, since $v_{k,j} \in V(CCC(n)) - V_k$ by Claim 6. The total number of tests performed is at most $N - m_k + (i + 1) + m_k \leq N + 2$.

(iii)-(b) $2 \leq i \leq m_k - 2$: Perform test $\langle \overline{v_{k,j}}, v_{k,j} \rangle$ in order of $j = 0, 1, \dots$ until the outcome of test $\langle \overline{v_{k,j}}, v_{k,j} \rangle$ is 0 for some $j = l$. Notice that $\overline{v_{k,i}}$ is fault-free since $\overline{v_{k,i}} \in V(CCC(n)) - V_k$ by Claim 6. Thus,

$$\begin{cases} v_{k,0}, v_{k,1}, v_{k,2} \in F & \text{if } l = 3, \\ v_{k,0}, v_{k,1}, v_{k,i+1} \in F & \text{if } l = 2, \\ v_{k,0}, v_{k,i+1} \in F & \text{if } l = 1, \text{ and} \\ v_{k,i+1} \in F & \text{if } l = 0. \end{cases}$$

If $l \leq 1$ then we test $v_{k,j}$ by $\overline{v_{k,j}}$ for every integer j , $i + 2 \leq j \leq m_k - 1$. If the outcome of test $\langle \overline{v_{k,j}}, v_{k,j} \rangle$ is 1 then $v_{k,j}$ is faulty. Hence, we can identify F using at most $N - m_k + (i + 1) + 2 + (m_k - i - 2) \leq N + 1$ tests.

(iii)-(c) $i = m_k - 1$: In this case, $v_{k,0}$ is faulty and $v_{k,j}$ is fault-free for any integer j , $3 \leq j \leq m_k - 1$. Thus, if the outcome of test $\langle \overline{v_{k,1}}, v_{k,1} \rangle$ is 0, then $F = \{v_{k,0}\}$; If the outcome of test $\langle \overline{v_{k,1}}, v_{k,1} \rangle$ is 1 and the outcome of test

$\langle \overline{v_{k,2}}, v_{k,2} \rangle$ is 0, then $F = \{v_{k,0}, v_{k,1}\}$; If the outcome of test $\langle \overline{v_{k,1}}, v_{k,1} \rangle$ is 1 and the outcome of test $\langle \overline{v_{k,2}}, v_{k,2} \rangle$ is 1, then $F = \{v_{k,0}, v_{k,1}, v_{k,2}\}$. Hence, we can identify F using at most $N + 2$ tests.

(iv) $|X| = 0$: By Claim 9, we can identify $F = \emptyset$ using N tests.

By (i), (ii), (iii), and (iv), we can diagnose $CCC(n)$ using at most $N + 2$ tests. $\square$

4 Adaptive Parallel Diagnosis

In adaptive parallel diagnosis, several tests may be performed simultaneously in a testing round, but each vertex can participate in at most one test. That is, the tests in a testing round are a directed matching on the vertices. Since at least $N + t - 1$ tests are necessary for adaptive parallel diagnosis of an N -vertex graph with at most t faulty vertices and there are at most $N/2$ tests in each testing round, $\lceil (N + t - 1)/(N/2) \rceil$ is a general lower bound for the number of testing rounds. Thus we have the following.

Theorem IV [2] *If G is a graph with at most t faulty vertices then 3 testing rounds are necessary to adaptively diagnose G provided that $t \geq 2$.*

4.1 Even Cycles

The following theorem will be used in the next section.

Theorem 3. *An even cycle C_N can be adaptively diagnosed in 3 testing rounds if the number of faults is not more than 2 and $N \geq 6$.*

Proof. In **Step** 1 of Algorithm 1 shown in Figure 2, all tests can be performed in two rounds, since N is even. In **Step** 2, just one test is performed, and this can be done in a testing round. Thus we have the theorem.

4.2 Hypercubes

The following theorem is shown in [6].

Theorem V [6] *$Q(n)$ can be adaptively diagnosed in 4 testing rounds if the number of faults is not more than n and $n \geq 3$.*

We prove the following theorem.

Theorem 4. *$Q(n)$ can be adaptively diagnosed in 3 testing rounds if the number of faults is not more than $n - \lceil \log(n - \lceil \log n \rceil + 4) \rceil + 2$ and $n \geq 4$.*

4.2.1 Proof of Theorem 4 Let $t = n - \lceil \log(n - \lceil \log n \rceil + 4) \rceil + 2$. $Q(n)$ is represented as $Q(n - t + 2) \times Q(t - 2)$. Notice that $t \geq 3$ since $n \geq 4$. We need a few technical lemmas.

Lemma 2. $|V(Q(n - t + 2))| > t$.

Proof. $|V(Q(n - t + 2))| = 2^{n-t+2} = 2^{\lceil \log(n - \lceil \log n \rceil + 4) \rceil} \geq n - \lceil \log n \rceil + 4 > t$. $\square$

Lemma 3. For any $S \subseteq V(Q(n))$ with $|S| \leq n$, each vertex in S has a distinct adjacent vertex in $V(Q(n)) - S$.

Proof. We prove the lemma by induction on n . The case when $n = 1$ is trivial. Assume that the lemma holds if $n = k$. Let S be a set of vertices of $Q(k + 1)$ with $|S| \leq k + 1$. Since $Q(k + 1) = Q(k) \times P_2$, $Q(k + 1)$ can be decomposed into two disjoint copies of $Q(k)$, say $Q_1(k)$ and $Q_2(k)$. We distinguish two cases.

(i) $S \subseteq V(Q_1(k))$: The vertices in $Q_2(k)$ corresponding to the vertices in S are the desired vertices.

(ii) $S \cap V(Q_1(k)) \neq \emptyset$ and $S \cap V(Q_2(k)) \neq \emptyset$: Let $S_i = S \cap V(Q_i(k))$ ($i = 1, 2$). Since $|S_i| \leq k$ ($i = 1, 2$), S_i has a desired set of vertices in $Q_i(k)$ by the inductive hypothesis. $\square$

Now we are ready to describe our algorithm. Our algorithm works in two steps. It is well-known that $Q(n)$ has a Hamilton cycle. In the first step, we perform in two testing rounds all tests along a Hamilton cycle in all copies of $Q(n - t + 2)$ in the clockwise direction. A copy of $Q(n - t + 2)$ is said to be fault-free if it has no faulty vertex, and faulty otherwise. The following is immediate from Lemma 2.

Lemma 4. A copy of $Q(n - t + 2)$ is faulty if and only if the tests along a Hamilton cycle have an outcome of 1. $\square$

Let $\mathcal{F}$ be the set of all faulty copies of $Q(n - t + 2)$.

The second step of our algorithm is distinguished in three cases depending on $|\mathcal{F}|$.

If $|\mathcal{F}| = t$ then each faulty copy of $Q(n - t + 2)$ has just one faulty vertex, which we can identify from the test results in the first step.

If $|\mathcal{F}| = t - 1$ then each faulty copy of $Q(n - t + 2)$ has at most two faulty vertices, which we can identify in one more testing round by Theorem 3.

If $|\mathcal{F}| \leq t - 2$ then for each faulty copy Q_F of $Q(n - t + 2)$, there is a distinct fault-free copy Q_H of $Q(n - t + 2)$ in which each vertex v_H is adjacent to the corresponding vertex v_F in Q_F by Lemma 3. By performing the tests $\langle v_H, v_F \rangle$ for all faulty copies of $Q(n - t + 2)$ in one testing round, we can identify all the faults.

Our algorithm is summarized in Fig. 3.

4.3 CCC's

The following theorem is proved based on adaptive serial diagnosis for CCC's in Section 3.3.

Algorithm 2 Step 1 Perform in 2 testing rounds all tests along a Hamilton cycle in all copies of $Q(n - t + 2)$ in the clockwise direction. Let $\mathcal{F}$ be the set of all faulty copies of $Q(n - t + 2)$. Step 2 If $\mathcal{F} = t$ then identify the faults; If $\mathcal{F} = t - 1$ then perform tests in one more testing round according to Step 2 of Algorithm 1, and identify the faults; If $\mathcal{F} \leq t - 2$ then diagnose all vertices in all faulty copies of $Q(n - t + 2)$ by corresponding vertices in distinct fault-free copies of $Q(n - t + 2)$ in one more testing round.

Fig. 3. Algorithm 2

Theorem 5. $CCC(n)$ can be adaptively diagnosed in 3 testing rounds if the number of faults is not more than 3 and $n \geq 4$.

Proof. Let $(V_0, V_1, \dots, V_{2^{n-1}-1})$ be a partition of $V(CCC(n))$ defined in the proof of Theorem 2. Our algorithm works in two steps. By Claim 7, every block $V_k (k \in [2^{n-1}])$ is isomorphic to C_{m_k} . In the first step, we perform in two testing rounds all tests along a cycle C_{m_k} in all block V_k in the clockwise direction. A block V_k is said to be fault-free if it has no faulty vertex, and faulty otherwise. Since every block V_k has $4\lceil n/2 \rceil \geq 4$ vertices, we have the following.

Lemma 5. V_k is faulty if and only if the tests along a cycle C_{m_k} have an outcome of 1. $\square$

Let $\mathcal{F}$ be the set of all faulty blocks. $|\mathcal{F}| \leq 3$ by the assumption. The second step of our algorithm is distinguished in four cases depending on $|\mathcal{F}|$.

- (i) $|\mathcal{F}| = 3$: Each block $V_k \in \mathcal{F}$ has only one faulty vertex since there are at most three faulty vertices. Thus faulty vertices can be identified from the test results in the first step.
- (ii) $|\mathcal{F}| = 2$: Each block $V_k \in \mathcal{F}$ has at most 2 faulty vertices, which we can identify in one more testing round by Theorem 3.
- (iii) $|\mathcal{F}| = 1$: It is easy to see from Claim 6 that each vertex v_F in the block $V_F \in \mathcal{F}$, there exists a distinct vertex $\overline{v_F} \in V(CCC(n)) - V_F$ adjacent with v_F . We perform tests $\langle \overline{v_F}, v_F \rangle$ for all vertices v_F in V_F in one testing round.
- (iv) $|\mathcal{F}| = 0$: From the test results in the first step, we know that there is no fault.

5 Concluding Remarks

1. We can prove that 4 testing rounds are necessary and sufficient to adaptively diagnose an odd cycle C_N if the number of faulty vertices is at most 2 and $N \geq 5$.

2. We can prove that $CCC(3)$ is also adaptively 3-diagnosable using at most $N+2$ tests. The proof is similar to that of Theorem 2. Notice that $CCC(2)$ is just C_8 . We can show that $CCC(3)$ can be adaptively diagnosed in 4 testing rounds if the number of faults is at most 3. It is open if 3 testing rounds are sufficient for $CCC(3)$.
3. $Q(3)$ can be adaptively diagnosed in 3 testing rounds if the number of faults is at most 3, as mentioned in [13]. Notice that $Q(2)$ is just C_4 . We can prove that $Q(n)$ can be adaptively diagnosed in 3 testing rounds if the number of faults is not more than $n - \lceil \log(n - \lceil \log n \rceil + 3) \rceil + 2$ and $n \geq 3$. The proof is similar to that of Theorem 4 but more complicated. It is still open whether 3 testing rounds are sufficient to adaptively diagnose $Q(n)$ with at most t faulty vertices even if $n - \lceil \log(n - \lceil \log n \rceil + 3) \rceil + 3 \leq t \leq n$. A similar approach based on the decomposition of $Q(n)$ into subcubes can be found in [13], in which it is shown that $Q(n)$ is adaptively n -diagnosable using $N + 3n/2$ tests if $n \geq 3$, and $Q(n)$ is adaptively diagnosable in 11 testing rounds if the number of faulty vertices is not more than n and $n \geq 3$.
4. We can prove that a d -dimensional torus can be adaptively diagnosed in 3 testing rounds if the number of faulty vertices is at most $2d$ and the number of vertices in the side is even. We can also show that a d -dimensional mesh can be adaptively diagnosed in 3 testing rounds if the number of faulty vertices is at most d . The details will appear in the forthcoming full version of the paper.

Acknowledgments

The adaptive diagnosis algorithms in Section 3 are a joint work with Ken Hayase. The research is a part of CAD21 Project at TIT.

References

1. R. Beigel, W. Hurwood, and N. Kahale. Fault diagnosis in a flash. *Proc. of 36th FOCS*, pages 571–580, 1995. 87
2. R. Beigel, S.R. Kosaraju, and G.F. Sullivan. Locating faults in a constant number of testing rounds. *Proc. of 1st SPAA*, pages 189–198, 1989. 87, 94
3. R. Beigel, G. Margulis, and D.A. Spielman. Fault diagnosis in a small constant number of parallel rounds. *Proc. of 5th SPAA*, pages 21–29, 1993. 87
4. R.P. Bianchini, Jr., and R. Buskens. An adaptive distributed system level-diagnosis algorithm and its implementation. *Proc. of 21st Int. Symp. on Fault Tolerant Computing*, pages 222–229, 1991. 87
5. R.P. Bianchini, Jr., K. Goodwin, and D.S. Nydick. Practical application and implementation of distributed system-level diagnosis theory. *Proc. of 20th Int. Symp. on Fault Tolerant Computing*, pages 332–339, 1984. 87
6. A. Bjöklund. Optimal adaptive fault diagnosis of hypercubes. *Proc. of SWAT, LNCS 1851*, pages 527–534, 2000. 87, 89, 94
7. P.M. Blecher. On a logical problem. *Discrete Mathematics*, 43:107–110, 1983. 87, 89

8. C. Feng, L.N. Bhuyan, and F. Lombardi. Adaptive system-level diagnosis for hypercube multiprocessors. *IEEE Trans. Computers*, 45:1157–1170, 1996. [87](#)
9. S.L. Hakimi and A.T. Amin. Characterization of connection assignment of diagnosable systems. *IEEE Trans. Computers*, 23:86–88, 1974. [86](#)
10. S.L. Hakimi and K. Nakajima. On adaptive system diagnosis. *IEEE Trans. Computers*, 33:234–240, 1984.
11. S.L. Hakimi, M. Otsuka, E.F. Schmeichel, and G.F. Sullivan. A parallel fault identification algorithm. *Journal of Algorithm*, 11:231–241, 1990. [87](#)
12. S.L. Hakimi and E.F. Schmeichel. An adaptive algorithm for system level diagnosis. *Journal of Algorithms*, 5:524–530, 1984. [87](#)
13. E. Kranakis and A. Pelc. Better adaptive diagnosis of hypercubes. *IEEE Trans. Computers*, 49:1013–1020, 2000. [87](#), [97](#)
14. E. Kranakis, A. Pelc, and A. Spatharis. Optimal adaptive fault diagnosis for simple multiprocessor systems. *Networks*, 34:206–214, 1999. [87](#), [91](#)
15. K. Nakajima. A new approach to system diagnosis. *Proc. 19th allerton Conf. Commun. Contr. and Computing*, pages 697–706, 1981. [87](#)
16. F.P. Preparata, G. Metze, and R.T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Trans. Electronic Computers*, 16:848–854, 1967. [86](#), [88](#)
17. Pei Yuan Wu. Partial solution to problem 81-6. *Journal of Algorithms*, 12:379–380, 1982. [87](#), [89](#)

On-Line Multicasting in All-Optical Networks

Kenta Hashimoto, Toshinori Yamada, and Shuichi Ueno

Department of Communications and Integrated Systems,
Graduate School of Science and Engineering,
Tokyo Institute of Technology, Tokyo 152-8552, Japan
{hashi,yamada,ueno}@lab.ss.titech.ac.jp

Abstract. We consider the routing for a special type of communication requests, called a multicast, consisting of a fixed source and a multiset of destinations in a wavelength division multiplexing all optical network. We prove a min-max equality that the minimum number of wavelengths necessary for routing a multicast is equal to the maximum of the average number of paths that share a link in a cut of the network. Based on the min-max equality above, we propose an on-line algorithm for routing a multicast, and show that the competitive ratio of our algorithm is equal to the ratio of the degree of the source to the link connectivity of the network. We also show that $4/3$ is a lower bound for the competitive ratio of an on-line algorithm for routing a multicast.

1 Introduction

A WDM (Wavelength Division Multiplexing) all-optical network consists of routing nodes interconnected by point-to-point unidirectional fiber-optic links, which support a certain number of wavelengths. The same wavelength on two input ports cannot be routed to a same output port due to the interference. A fundamental problem for WDM all-optical networks is the optical routing, which assigns a path and a wavelength for each communication request in such a way that no two paths that traverse a common link are assigned the same wavelength by using as few wavelengths as possible. This paper considers the on-line optical routing for a special collection of communication requests called a multicast.

A WDM all-optical network is modeled as a symmetric digraph (directed graph) G with vertex set $V(G)$ and arc (directed edge) set $A(G)$ such that if $(u, v) \in A(G)$ then $(v, u) \in A(G)$, where the vertices represent the routing nodes and each arc represents a point-to-point unidirectional fiber-optic link connecting a pair of routing nodes.

Let $P(x, y)$ denote a dipath (directed path) in G from the vertex x to y which consists of consecutive arcs beginning at x and ending at y . A request is an ordered pair of vertices (x, y) in G corresponding to a message to be sent from x to y , and an instance I is a collection (multiset) of requests. A routing for an instance I is a collection of dipaths $R = \{P(x, y) | (x, y) \in I\}$.

Given a symmetric digraph G , an instance I , and a routing R for I , $\omega(G, I, R)$ is the minimum number of wavelengths that can be assigned to the dipaths in R ,

so that no two dipaths sharing an arc have the same wavelength. Let $\omega(G, I)$ denote the smallest $\omega(G, I, R)$ over all routings R for I . The load of an arc $\alpha \in A(G)$ in R , denoted by $\pi(G, I, R, \alpha)$, is the number of dipaths in R containing α . Let $\pi(G, I, R)$ denote the largest $\pi(G, I, R, \alpha)$ over all arcs $\alpha \in A(G)$, and $\pi(G, I)$ denote the smallest $\pi(G, I, R)$ over all routings R for I . It is known that computing $\omega(G, I)$ and $\pi(G, I)$ is NP-hard in general [2]. It is not difficult to see that $\omega(G, I) \geq \pi(G, I)$ for an instance I in a symmetric digraph G and that the inequality can be strict in general [2].

Beauquier, Hell, and Perennes [3] proved that for a multicast I in a symmetric digraph G , $\omega(G, I) = \pi(G, I)$ and both $\omega(G, I)$ and $\pi(G, I)$ can be computed in polynomial time. An instance I is called a multicast if I is of the form $\{(x, y) | y \in Y\}$ for a fixed vertex $x \in V(G)$, called the source, and a collection Y of vertices in $V(G)$, called the destinations.

This paper shows a min-max equality on $\omega(G, I)$ for a multicast I in a symmetric digraph G by means of the cut in G . For a digraph G and a nonempty proper subset $S \subset V(G)$, a cut $(S, \overline{S})$ is the set of arcs beginning in S and ending in $\overline{S}$, where $\overline{S} = V(G) - S$. For a multicast $I = \{(x, y) | y \in Y\}$ and a cut $(X, \overline{X})$ with $x \in X \subset V(G)$, let $\mu(G, I, X)$ denote $\lceil |Y \cap \overline{X}| / |(X, \overline{X})| \rceil$, and $\mu(G, I)$ denote the largest $\mu(G, I, X)$ over all cuts $(X, \overline{X})$ with $x \in X \subset V(G)$. Notice that $\mu(G, I, X)$ is a lower bound on the average load of an arc in $(X, \overline{X})$ for any routing for I . We prove a min-max equality that $\omega(G, I) = \mu(G, I)$, which is used as a basis for on-line multicasting. Let $\delta(x)$ denote the outdegree of x and $\lambda(x)$ denote $\min\{|(X, \overline{X})| | x \in X \subset V(G)\}$. Notice that $\delta(x) \geq \lambda(x)$. If I is a broadcast, that is $I = \{(x, y) | y \in V(G) - x\}$ and $\delta(x) = \lambda(x)$ then our min-max equality implies that $\omega(G, I) = \lceil |V(G) - 1| / \delta(x) \rceil$, which is essentially Theorem 3.1 in [4] proved by Bermond, Gargano, Perennes, Rescigno, and Vaccaro.

Given a symmetric digraph G and a sequence of requests (x_i, y_i) , an on-line algorithm assigns a dipath $P(x_i, y_i)$ and a wavelength to $P(x_i, y_i)$, so that no two dipaths sharing an arc are assigned the same wavelength. The performance measure for an on-line algorithm is the competitive ratio defined as the worst-case ratio over all request sequences between the number of wavelengths used by the on-line algorithm and the optimal number of wavelengths necessary on the same sequence. Bartal and Leonardi [1] showed on-line algorithms with competitive ratio of $O(\log N)$ for any instances in N -vertex digraphs associated with meshes, trees, and trees of rings, where the digraph associated with a graph H is the symmetric digraph obtained when each edge e of H is replaced by two oppositely oriented arcs with the same ends as e . They also proved a matching lower bound of $\Omega(\log N)$ for digraphs associated with meshes, and a lower bound of $\Omega(\log N / \log \log N)$ for digraphs associated with trees and trees of rings [1].

We show here an on-line algorithm for a multicast $I = \{(x, y) | y \in Y\}$ in a symmetric digraph G . We prove that the competitive ratio of our algorithm is $\lceil \delta(x) / \lambda(x) \rceil$. It follows that if $\delta(x) = O(1)$ then the competitive ratio of our algorithm is $O(1)$. Moreover, if $\delta(x) = \lambda(x)$ then our algorithm is optimal. We also show a complementary result that if $\delta(x) > \lambda(x)$ then there is no optimal

on-line algorithm. Moreover, we show that the competitive ratio of any on-line algorithm is at least $4/3$. We also consider the dynamic multicasting.

2 Off-Line Multicasting

We prove in this section the following min-max equality, which will be used in the subsequent sections.

Theorem 1. $\omega(G, I) = \mu(G, I)$ for a multicast I in a symmetric digraph G .

2.1 Proof of Theorem 1

Let G be a symmetric digraph and $I = \{(x, y) | y \in Y\}$ be a multicast in G .

Proof of $\omega(G, I) \geq \mu(G, I)$. It is well-known and easily verified that

$$\omega(G, I) \geq \pi(G, I). \quad (1)$$

Since $\mu(G, I, X)$ is a lower bound on the average load of an arc in a cut $(X, \overline{X})$ with $x \in X \subset V(G)$ for any routing R for I , we have

$$\pi(G, I, R) \geq \mu(G, I, X)$$

for any routing R for I and any cut $(X, \overline{X})$ with $x \in X \subset V(G)$. Thus, it follows that

$$\pi(G, I) \geq \mu(G, I). \quad (2)$$

Combining (1) and (2), we have

$$\omega(G, I) \geq \mu(G, I).$$

Proof of $\omega(G, I) \leq \mu(G, I)$. It is proved in [3] that for a multicast $I = \{(x, y) | y \in Y\}$ in a symmetric digraph G we have

$$\omega(G, I) = \pi(G, I), \quad (3)$$

by using flow networks derived from G .

In a flow network, we denote by $c(u, v)$ the capacity of an arc (u, v) , and by $c(T, \overline{T})$ the capacity of a cut $(T, \overline{T})$. Although Y is a collection (multiset) in general, we assume without loss of generality that Y is just a set, as mentioned in [3].

In order to compute $\pi(G, I)$ the following flow network F_p is introduced in [3]. Let s and t be two new vertices which will be the source and sink in F_p ,

respectively. The flow network F_p is defined as follows:

$$\begin{aligned} V(F_p) &= \{s, t\} \cup V(G) \\ A(F_p) &= \{(s, x)\} \cup A(G) \cup \left(\bigcup_{y \in Y} \{(y, t)\}\right) \\ c(s, x) &= \infty \\ c(u, v) &= p \text{ for all } (u, v) \in A(G) \\ c(y, t) &= 1 \text{ for all } y \in Y. \end{aligned}$$

The following theorem is immediate from the definitions.

Theorem I [3] $\pi(G, I) \leq p$ if and only if F_p has a flow of value $|Y|$.

By (3) and Theorem I above, it suffices to show that $F_{\mu(G, I)}$ has a flow of value $|Y|$. We prove this by showing that any cut in $F_{\mu(G, I)}$ separating s and t has capacity at least $|Y|$. Any cut in $F_{\mu(G, I)}$ separating s and t can be represented as $(S \cup \{s\}, \bar{S} \cup \{t\})$ for a subset S of $V(G)$ and $\bar{S} = V(G) - S$. It is easy to see that

$$c(S \cup \{s\}, \bar{S} \cup \{t\}) = \begin{cases} |Y \cap S| + \mu(G, I) \cdot |(S, \bar{S})| & \text{if } x \in S \\ \infty & \text{if } x \in \bar{S} \end{cases}$$

where $(S, \bar{S})$ is a cut in G . It follows that we may assume that $x \in S$. Then we have

$$\begin{aligned} c(S \cup \{s\}, \bar{S} \cup \{t\}) &= |Y \cap S| + \mu(G, I) \cdot |(S, \bar{S})| \\ &= |Y \cap S| + \max \left\{ \left\lceil \frac{|Y \cap \bar{X}|}{|(X, \bar{X})|} \right\rceil \mid x \in X \subset V(G) \right\} \cdot |(S, \bar{S})| \\ &\geq |Y \cap S| + \left\lceil \frac{|Y \cap \bar{S}|}{|(S, \bar{S})|} \right\rceil \cdot |(S, \bar{S})| \\ &\geq |Y \cap S| + \frac{|Y \cap \bar{S}|}{|(S, \bar{S})|} \cdot |(S, \bar{S})| \\ &= |Y \cap S| + |Y \cap \bar{S}| = |Y|, \end{aligned}$$

as desired.

3 On-Line Multicasting

3.1 Upper Bounds

Let G be a symmetric digraph, and $(x, y_1), (x, y_2), \dots, (x, y_j), \dots$ be a sequence of multicast requests in G . Let I_j denote the collection $\{(x, y_1), (x, y_2), \dots, (x, y_j)\}$, and Y_j denote the collection $\{y_1, y_2, \dots, y_j\}$. We assume without loss of generality that x is not a cut-vertex in G . We also assume that the wavelengths are labeled with positive integers. Our on-line algorithm is based on the following classic theorem due to Edmonds [5]. For a vertex u of a digraph G , u -arborescence $H(u)$ in G is an acyclic spanning subdigraph of G such that for every vertex $v \in V(G)$ there is exactly one dipath in $H(u)$ from u to v .

Theorem II [5] *For a digraph G and a vertex $u \in V(G)$, the maximum number of arc-disjoint u -arborescences in G is equal to $\lambda(u)$.*

Let $\mathcal{H} = \{H_1(x), H_2(x), \dots, H_{\lambda(x)}(x)\}$ be a set of arc-disjoint x -arborescences in G . For each request, our on-line algorithm, called ARB, assigns a dipath in an x -arborescence in $\mathcal{H}$. Given a request (x, y_j) , ARB finds an x -arborescence $H_k(x)$ such that the number of dipaths in $H_k(x)$ assigned to the existing requests is minimal, assigns the unique dipath $P(x, y_j)$ in $H_k(x)$, and assigns the lowest available wavelength to $P(x, y_j)$.

Theorem 2. *The competitive ratio of ARB is $\lceil \delta(x)/\lambda(x) \rceil$.*

Proof. From Theorem 1, we have that for any j ,

$$\begin{aligned} \omega(G, I_j) &= \mu(G, I_j) \\ &= \max \left\{ \left\lceil \frac{|Y_j \cap \overline{X}|}{|(X, \overline{X})|} \right\rceil \mid x \in X \subset V(G) \right\} \\ &\geq \left\lceil \frac{|Y_j \cap (V(G) - \{x\})|}{|(\{x\}, V(G) - \{x\})|} \right\rceil \\ &= \left\lceil \frac{|Y_j|}{\delta(x)} \right\rceil \\ &\geq \frac{|Y_j|}{\delta(x)}. \end{aligned}$$

Let $\omega(G, I_j, \text{ALG})$ denote the number of wavelengths used by an on-line algorithm ALG for I_j . We have that

$$\begin{aligned} \omega(G, I_j, \text{ARB}) &= \left\lceil \frac{|Y_j|}{\lambda(x)} \right\rceil \\ &\leq \left\lceil \frac{\omega(G, I_j) \cdot \delta(x)}{\lambda(x)} \right\rceil \\ &\leq \left\lceil \frac{\delta(x)}{\lambda(x)} \right\rceil \cdot \omega(G, I_j), \end{aligned}$$

as desired. $\square$

The following corollaries are immediate. An on-line algorithm ALG is said to be optimal for G if $\omega(G, I_j, \text{ALG}) = \omega(G, I_j)$ for any j .

Corollary 1. *If $\delta(x)$ is $O(1)$ then the competitive ratio of ARB is $O(1)$.*

Corollary 2. *If $\delta(x) = \lambda(x)$ then ARB is optimal for G .*

Corollary 3. *ARB is optimal for digraphs associated with trees, cycles, tori, hypercubes, and cube-connected cycles.*

3.2 Lower Bounds

The following is a complementary result to Corollary 2.

Theorem 3. *If $\delta(x) > \lambda(x)$ then there is no on-line algorithm optimal for G .*

Proof. We prove the theorem by contradiction. Let G be a symmetric digraph, and x be a vertex in G with $\delta(x) > \lambda(x)$. Assume that there is an on-line algorithm ALG optimal for G . Let $(X, \overline{X})$ be a cut in G such that $x \in X \subset V(G)$ and $|(X, \overline{X})| = \lambda(x)$, and v be a vertex in $\overline{X}$. We denote the arcs with tail x by $(x, u_1), (x, u_2), \dots, (x, u_{\delta(x)})$. We consider the following sequence of requests:

$$(x, u_1), (x, u_2), \dots, (x, u_{\delta(x)}), \underbrace{(x, v), (x, v), \dots, (x, v)}_{\lambda(x)+1}.$$

Since ALG is optimal for G , ALG assigns for the requests (x, u_i) arc-disjoint dipaths $P(x, u_i)$ and the same wavelength, say w , to the dipaths $P(x, u_i)$ ($1 \leq i \leq \delta(x)$). Notice that each arc (x, u_i) is contained in the dipaths assigned wavelength w ($1 \leq i \leq \delta(x)$). Since $|(X, \overline{X})| = \lambda(x)$, ALG uses at least two more wavelengths different from w for the last $\lambda(x) + 1$ requests of (x, v) . Thus, ALG uses at least 3 wavelengths for the request sequence.

On the other hand, we have the following off-line algorithm. There is a set $\mathcal{A}$ of $\lambda(x)$ arc-disjoint x -arborescences in G by Theorem II. For each of $\lambda(x)$ requests of (x, v) , we assign a dipath in distinct x -arborescence in $\mathcal{A}$, and assign the same wavelength, say w , to the dipaths. Since $\delta(x) > \lambda(x)$, there exists some u_i ($1 \leq i \leq \delta(x)$) such that no dipaths above pass through u_i . Since x is not a cut-vertex, there is a dipath $P(u_i, v)$ that does not pass through x . For the remaining request of (x, v) , we assign a dipath consisting of arc (x, u_i) and $P(u_i, v)$, and assign a wavelength different from w , say w' , to the dipaths. Then we can assign a dipath consisting of an arc (x, u_j) with wavelength w' for every requests (x, u_j) ($j \neq i$), and arc (x, u_i) with wavelength w for request (x, u_i) . In total, we use only 2 wavelengths for the request sequence, a contradiction. Thus we have the theorem. $\square$

By corollary 2 and Theorem 3 above, we have the following corollary.

Corollary 4. *There is an on-line algorithm optimal for G if and only if $\delta(x) = \lambda(x)$.*

We can show a general lower bound as follows. Let M be a mesh with $V(M) = \{0, 1, 2\}^2$. The vertices ij and $i'j'$ are adjacent if and only if $|i - i'| + |j - j'| = 1$. Let G_M be the digraph associated with M .

Theorem 4. *The competitive ratio of any on-line algorithm for G_M is at least $4/3$.*

Proof. Let $u_1 = 01, u_2 = 10, u_3 = 12, u_4 = 21, v = 00$, and $x = 11$. Let ALG be any on-line algorithm for G_M . For any positive integer l , we consider the

following sequence of $4l$ requests I_{4l} :

$$\underbrace{(x, u_1), \dots, (x, u_1)}_l, \underbrace{(x, u_2), \dots, (x, u_2)}_l, \underbrace{(x, u_3), \dots, (x, u_3)}_l, \underbrace{(x, u_4), \dots, (x, u_4)}_l. \quad (4)$$

If $\omega(G_M, I_{4l}, \text{ALG}) \geq 4l/3$ then we are done, because $\omega(G_M, I_{4l}) = l$ as easily seen, and we have

$$\omega(G_M, I_{4l}, \text{ALG}) \geq \frac{4}{3}l = \frac{4}{3}\omega(G_M, I_{4l}).$$

If $\omega(G_M, I_{4l}, \text{ALG}) < 4l/3$ then we consider the following sequence of additional $4l$ requests I'_{4l} :

$$\underbrace{(x, v), (x, v), \dots, (x, v)}_{4l}. \quad (5)$$

Suppose that ALG uses $l+i$ ($0 \leq i < l/3$) wavelengths for the sequence (4), and let $W = \{w_1, w_2, \dots, w_{l+i}\}$ be the set of wavelengths used for the sequence (4). Since the outdegree of x is 4, the maximum number of requests for which we can assign wavelengths in W is $4(l+i)$. Since the number of requests in the sequence (4) is $4l$, ALG can use the wavelengths in W for at most $4(l+i) - 4l = 4i$ requests in the sequence (5). Since the indegree of v is 2, ALG needs at least $(4l - 4i)/2 = 2l - 2i$ additional wavelengths not in W for the sequence (5). Thus, ALG uses at least $(l+i) + (2l - 2i) = 3l - i$ wavelengths for the concatenation of the sequences (4) and (5). Since $i < l/3$, we have

$$\omega(G_M, I_{4l} \cup I'_{4l}, \text{ALG}) \geq 3l - i > 3l - \frac{1}{3}l = \frac{8}{3}l.$$

On the other hand, it is easy to see that $\omega(G_M, I_{4l} \cup I'_{4l}) = 2l$. Thus we have

$$\omega(G_M, I_{4l} \cup I'_{4l}, \text{ALG}) > \frac{4}{3}\omega(G_M, I_{4l} \cup I'_{4l}),$$

as desired. $\square$

Notice that $\omega(G_M, I, \text{ARB}) \leq 2\omega(G_M, I)$ for any multicast I .

Our general upper bound for the competitive ratio is $\lceil \delta(x)/\lambda(x) \rceil$, and general lower bound is $4/3$. It is an interesting open problem to close the gap between upper and lower bounds above.

4 Dynamic Multicasting

Given a symmetric digraph G and a sequence of request arrivals and terminations for a multicast $I = \{(x, y) | y \in Y\}$, a dynamic algorithm assigns a dipath $P(x, y_i)$ and a wavelength to $P(x, y_i)$, so that no two dipaths sharing an arc are assigned the same wavelength if a request (x, y_i) arrives, and deletes $P(x, y_i)$ together with the wavelength assigned if a request (x, y_i) terminates.

Let I_j denote a collection of the existing requests just after j th request arrival or termination in the sequence. We denote by $\omega(G, x, L, \text{ALG}, I_j)$ the number of wavelengths used by a dynamic algorithm ALG for I_j provided that $\mu(G, I_j) \leq L$ for any j . Let $\omega(G, x, L, \text{ALG})$ denote $\max_j \omega(G, x, L, \text{ALG}, I_j)$ and $\omega(G, x, L)$ denote the smallest $\omega(G, x, L, \text{ALG})$ over all dynamic algorithms ALG. Notice that $\omega(G, x, L) \geq L$.

Our dynamic algorithm ARB' is obtained from ARB by just adding an operation that when an existing request terminates, ARB' deletes the dipath assigned for the request together with wavelength assigned. The following results are immediate from the corresponding results in the previous section.

Theorem 5.

$$\omega(G, x, L, \text{ARB}') \leq \left\lceil \frac{L \cdot \delta(x)}{\lambda(x)} \right\rceil.$$

Corollary 5. If $\delta(x) = O(1)$ then $\omega(G, x, L, \text{ARB}') = O(L)$.

Theorem 6. $\omega(G, x, L) = L$ if and only if $\delta(x) = \lambda(x)$.

Theorem 7.

$$\omega(G_M, x, L) \geq \frac{4}{3}L.$$

It should be noted that the performance of dynamic optical routing is considerably less than that of on-line optical routing in general, as mentioned in [6]. Our results indicate that the performance of dynamic multicasting is comparable to that of on-line multicasting.

References

1. Y. Bartal and S. Leonardi. On-line routing in all-optical networks. *Theoretical Computer Science*, 221:19–39, 1999. 100
2. B. Beauquier, J-C. Bermond, L. Gargano, P. Hell, S. Perennes, and U. Vaccaro. Graph problems arising from wavelength-routing in all-optical networks. *2nd workshop on Optics and Computer Science*, April 1997. 100
3. B. Beauquier, P. Hell, and S. Perennes. Optimal wavelength-routed multicasting. *Discrete Applied Mathematics*, 84:15–20, 1998. 100, 101, 102
4. J-C. Bermond, L. Gargano, S. Perennes, A.A. Rescigno, and U. Vaccar. Efficient collective communication in optical networks. *Theoretical Computer Science*, 233:165–189, 2000. 100
5. J. Edmonds. Edge-disjoint branchings. In *Combinatorial Algorithms*, pages 91–96. Algorithmics Press, 1972. 102, 103
6. G. Sasaki. WDM networks with limited or no wavelength conversion for worst case traffic. In *Optical WDM Networks -Principles and Practice-*, pages 127–145. Kluwer Academic Publishers, 2000. 106

Enumerating Floorplans with n Rooms

Shin-ichi Nakano

Gunma University
Kiryu 376-8515, Japan
Fax: +81-277-30-1812
nakano@cs.gunma-u.ac.jp

Abstract. A plane drawing of a graph is called a floorplan if every face (including the outer face) is a rectangle. A based floorplan is a floorplan with a designated base line segment on the outer face. In this paper we give a simple algorithm to generate all based floorplans with at most n faces. The algorithm uses $O(n)$ space and generates such floorplans in $O(1)$ time per floorplan without duplications. The algorithm does not output entire floorplans but the difference from the previous floorplan. By modifying the algorithm we can generate without duplications all based floorplans having exactly n faces in $O(1)$ time per floorplan. Also we can generate without duplications all (non-based) floorplans having exactly n faces in $O(n)$ time per floorplan.

Keyword: Graphs, Plane graphs, Enumeration, Listing

1 Introduction

Generating all graphs with some property without duplications has many applications, including unbiased statistical analysis [6]. A lot of algorithms to solve these problems are already known [1,2,6,7, etc]. See nice textbooks [3,4].

In this paper we wish to generate all “based” floorplans, which will be defined precisely in Section 2. All based floorplans with three rooms are shown in Fig. 1. Such floorplans play an important role in many algorithms, including VLSI floorplanning. By checking all (or some of) based floorplans, we can find the best (or possibly nice) floorplan with respect to some given property. Also we can have a catalog of floorplans.



Fig. 1. Based floorplans with three rooms

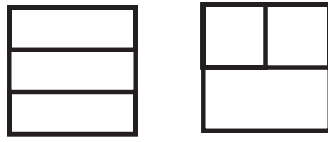


Fig. 2. Floorplans with three rooms

To solve these all-graph-generating problems some types of algorithms are known.

Classical method algorithms [3, p57] first generate all the graphs with given property allowing duplications, but output only if the graph has not been output yet. Thus this method requires quite a huge space to store a list of graphs that have already been output. Furthermore, checking whether each graph has already been output requires a lot of time.

Orderly method algorithms [3, p57] need not to store the list, since they output a graph only if it is a “canonical” representative of each isomorphism class.

Reverse search method algorithms [1] also need not to store the list. The idea is to implicitly define a connected graph H such that the vertices of H correspond to the graphs with the given property, and the edges of H correspond to some relation between the graphs. By traversing an implicitly defined spanning tree of H , one can find all the vertices of H , which correspond to all the graphs with the given property.

The main idea of our algorithm is that for some problems [5] we can define a tree (not a general graph) as the graph H of reverse search method. Thus our algorithm does not need to find a spanning tree of H , since H itself is a tree. With some other ideas we give the following simple but efficient algorithms.

Our first algorithm generates all based floorplans with “at most” n rooms. A *based floorplan* is a floorplan with a designated base line segment on the outer face. For instance, there are six based floorplans with exactly three rooms, as shown in Fig. 1. The base line segments on the outer face are depicted by thick lines. However, there are only two (non-based) floorplans with exactly three rooms. See Fig. 2. The algorithm uses $O(n)$ space and runs in $O(f(n))$ time, where $f(n)$ is the number of nonisomorphic based floorplans with at most n rooms. The algorithm generates floorplans without duplications. So the algorithm generates each floorplan in $O(1)$ time on average. The algorithm does not output entire floorplans but the difference from the previous floorplans.

By modifying our first algorithm we can generate without duplications all based floorplans having “exactly” n rooms in $O(1)$ time per floorplan. The algorithm uses $O(n)$ space. Also we can generate all (non-based) floorplans having exactly n rooms in $O(n)$ time (on average) per floorplan. The algorithm also uses $O(n)$ space.

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 shows a tree structure among based floorplans. Section 4 presents our

first algorithm. By modifying the algorithm we give two more algorithms in Section 5. Finally Section 6 is a conclusion.

2 Preliminaries

In this section we give some definitions.

Let G be a connected graph. A *tree* is a connected graph with no cycle. A *rooted tree* is a tree with one vertex r chosen as its *root*. For each vertex v in a tree, let $P(v)$ be the unique path from v to r . If $P(v)$ has exactly k edges then we say the *depth* of v is k . The *parent* of $v \neq r$ is its neighbor on $P(v)$, and the *ancestors* of $v \neq r$ are the vertices on $P(v)$ except v . The parent of r and the ancestors of r are not defined. We say if v is the parent of u then u is a *child* of v , and if v is an ancestor of u then u is a *descendant* of v . A *leaf* is a vertex having no child.

A drawing of a graph is *plane* if it has no two edges intersect geometrically except at a vertex to which they are both incident. A plane drawing divides the plane into connected regions called *faces*. The unbounded face is called *the outer face*, and other faces are called *inner faces*. We regard *the contour* of a face as the clockwise cycle formed by the line segments on the boundary of the face. Two faces F_1 and F_2 are *ns-adjacent* if they share a horizontal line segment. Two faces F_1 and F_2 are *ew-adjacent* if they share a vertical line segment.

A *floorplan* is a plane drawing in which every face (including the outer face) is a rectangle. In this paper we only consider floorplans which has no vertex shared by four (or more) rectangles. A *based floorplan* is a floorplan with one designated bottom line segment on the contour of the outer face. The designated bottom line segment is called *the base*, and we always draw the base as the lowermost line segment of the drawing. For examples, based floorplans with three faces are shown in Fig. 1, in which each base is depicted by a thick line. If two floorplans P_1 and P_2 have a one-to-one correspondence between faces preserving ns- and ew-adjacency, then we say P_1 and P_2 are isomorphic. If two based floorplans P_1 and P_2 have a one-to-one correspondence between faces preserving ns- and ew-adjacency, and in which each base corresponding to the other, then we say P_1 and P_2 are isomorphic.

3 The Sweeping Sequence and the Genealogical Tree

Let S_n be the set of all non-isomorphic based floorplans having at most n faces. In this section we explain a tree structure among the floorplans in S_n .

Let R_1 be the floorplan having exactly one inner face. Assume R is a based floorplan in S_n except R_1 . Let F be the inner face of R having the upper-left corner of the outer rectangle of R . We call such a face *the first face* of the based floorplan R . “First faces” of based floorplans are shaded in Fig. 3–6. The first face F is *upward removable* if R has a vertical line segment with upper end v where v is the lower-right corner of F . See Fig. 3 (a). Otherwise, R has a horizontal line segment with left end v , and the first face F is *leftward removable*.

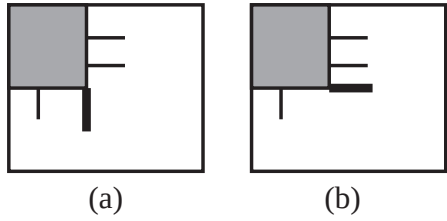


Fig. 3. (a) An upward removable face and (b) a leftward removable face

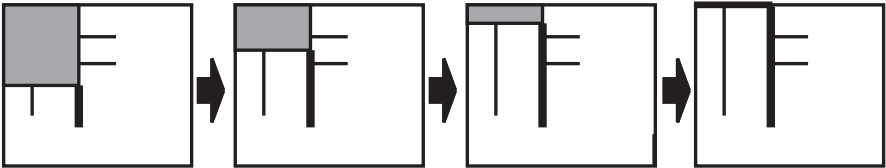


Fig. 4. Removing an upward removable face

See Fig. 3 (b). Since R is not R_1 , for any R the first face is either upward removable or leftward removable. If F is upward removable then by continually shrinking the first face into the uppermost horizontal line of R with preserving the width of F , and enlarging the faces below F , as shown in Fig. 4, eventually we have a floorplan with one less faces. Similarly, if F is leftward removable then by continually shrinking the first face into the leftmost line of R with preserving the height of F , eventually we have a floorplan with one less faces. If we remove the first face from R then the resulting floorplan is again a based floorplan in S_n with one less faces. We denote such floorplan as $P(R)$. Thus we can define the based floorplan $P(R)$ in S_n for each R in S_n except R_1 . We say R is a *child floorplan* of $P(R)$.

Given a floorplan R in S_n , by repeatedly removing the first face, we can have the unique sequence $R, P(R), P(P(R)), \dots$ of floorplans in S_n which eventually ends with R_1 , which is the floorplan having exactly one inner face. See an example in Fig. 5, in which the first faces are shaded.

By merging those sequences we can have the *genealogical tree* T_n of S_n such that the vertices of T_n correspond to the floorplans in S_n , and each edge corresponds to each relation between some R and $P(R)$. For instance, T_4 is shown

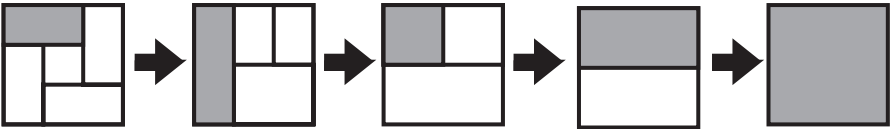
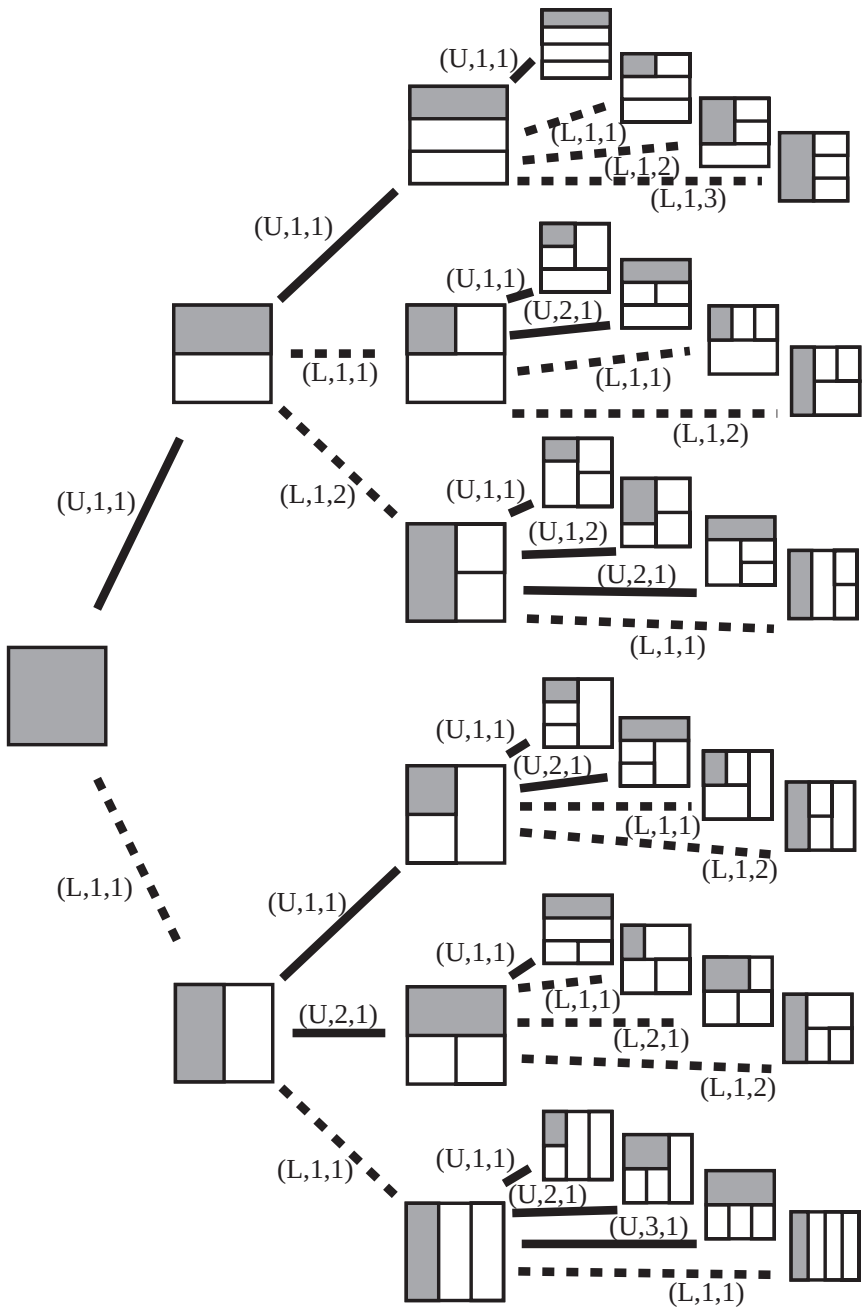


Fig. 5. The removing sequence

Fig. 6. Genealogical tree T_4

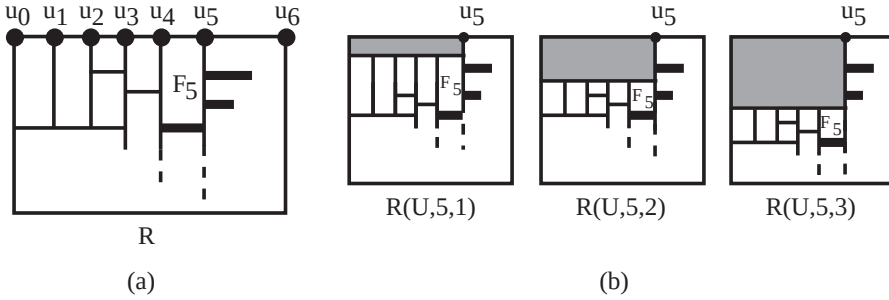


Fig. 7. (a) A floorplan R and (b) some child floorplans of R

in Fig. 6, in which the first faces are shaded, each edge corresponds to upward removing is depicted by a solid line, and each edge corresponds to leftward removing is depicted by a dotted line. We call the vertex in T_n corresponding to R_1 the root of T_n .

4 Algorithm

Given S_n we can construct T_n by the definition, possibly with huge space and much running time. However, how can we construct T_n efficiently only given an integer n ? Our idea is by reversing the removing procedure as follows.

Given a based floorplan R in S_n with at most $n - 1$ faces, we wish to find all child floorplans of R .

We need some definitions here. Assume R is a floorplan in S_n . Let P_N be the uppermost horizontal line segment of R , and vertices $u_0, u_1, \dots, u_x$ are vertices on P_N each of which is an upper end of a vertical line segment. Assume $u_0, u_1, \dots, u_x$ appear on P_N from left to right in this order. See an example in Fig. 7(a). Let F_i be the inner face of R with upper-right corner u_i , for $1 \leq i \leq x$. We can observe that if F_i has k neighbor faces to the right, the R has exactly k child floorplans R_c such that the first face of R_c is upward removable, the first face has i neighbor faces to the bottom. We denote by $R(U, s, e)$ the child floorplan of R such that (1) the first face of $R(U, s, e)$ is upward removable, and (2) the first face of $R(U, s, e)$ has s neighbor faces to the bottom and e neighbor face to the right. For instance, face F_5 of floorplan R in Fig. 7(a) has three neighbor faces to the right, therefore R has exactly three child floorplans $R(U, 5, 1)$, $R(U, 5, 2)$, $R(U, 5, 3)$ in each of which the first face has five neighbor faces to the bottom. Similarly we denote by $R(L, s, e)$ the child floorplan of R such that (1) the first face of $R(L, s, e)$ is leftward removable, and (2) the first face of $R(L, s, e)$ has s neighbor faces to the bottom and e neighbor face to the right. In Fig. 6 the labels (U, s, e) and (L, s, e) are shown.

Thus, given a based floorplan R in S_n with at most $n - 1$ faces, we can find all child floorplans $R(U, 1, 1), \dots$ of R in S_n . If R has k child floorplans then we can find them in $O(k)$ time, since if we have $R(U, s, e)$ then we can have each of

$R(U, s + 1, e)$ and $R(U, s, e + 1)$ in $O(1)$ time, respectively. This is an intuitive reason why our algorithm generates each floorplan in $O(1)$ time on average.

And recursively repeating this process from the root of T_n corresponding to R_1 we can traverse T_n without constructing whole T_n . During the traversal of T_n , we assign a label either (U, s, e) or (L, s, e) to each edge connecting R and $P(R)$ in T_n , as shown in Fig. 6. Each label denotes how to generate a child floorplan of R , and each sequence of labels on a path starting from the root specifies a floorplan in S_n . For instance $(U, 1, 1)$, $(U, 1, 1)$, $(U, 1, 1)$ specify the uppermost floorplan in Fig. 6. During our algorithm we will maintain these labels only on the path from the root to the “current” vertex, because those are enough information to generate the “current” floorplan. To generate next floorplan, we need to maintain some more information only for the floorplans on the “current” path, which has length at most n . This is an intuitive reason why our algorithm uses only $O(n)$ space, while the number of floorplans may not be bounded by a polynomial in n .

Our algorithm is as follows.

Procedure find-all-child-floorplans(R)

begin

- 1 Output R { Output the difference from the previous tree. }
- 2 **if** R has exactly n faces **then return**
- 3 Let $F_1, F_2, \dots, F_x$ are the inner face of R sharing the uppermost horizontal line segment of R , and assume that they appear from left to right in this order.
- 4 **for** $i = 1$ **to** x
- 5 Assume F_i has $e(i)$ neighbors to the right.
- 6 **for** $j = 1$ **to** $e(i)$
- 7 **find-all-child-floorplans**($R(U, i, j)$)
- 8 Let $F_1^\square, F_2^\square, \dots, F_y^\square$ are the inner face of R sharing the leftmost vertical line segment of R , and assume that they appear from top to bottom in this order.
- 9 **for** $i = 1$ **to** y
- 10 Assume $F_i^\square$ has $s(i)$ neighbors to the bottom.
- 11 **for** $j = 1$ **to** $s(i)$
- 12 **find-all-child-floorplans**($R(L, j, i)$)
- end**

Algorithm find-all-floorplans(n)

begin

- 1 **find-all-child-floorplans**(R_1)
- end**

Theorem 1. *The algorithm uses $O(n)$ space and runs in $O(f(n))$ time, where $f(n)$ is the number of non-isomorphic based floorplan with at most n faces.*

Proof. Given a based floorplan R , we can find all k child floorplans in $O(k)$ time. For other part, our algorithm needs only a constant time of computations

for each edge of the tree. Thus the algorithm runs in $O(f(n))$ time. For each recursive call we need a constant number of space, and the depth of recursive call is bounded by $n - 1$. Thus the algorithm uses $O(n)$ space. $\square$

5 Modification of the Algorithm

Then we consider our second problem.

Let $S_{=n}$ be the set of non-isomorphic based floorplans having exactly n faces. We wish to generate all floorplans in $S_{=n}$ without duplications. Clearly all such floorplans are in S_n but with other floorplans. How can we output only floorplans in $S_{=n}$, furthermore efficiently? We have the following lemma.

Lemma 1. *Let $g(n)$ be the number of floorplans in $S_{=n}$. Then S_n has at most $2 \cdot g(n)$ floorplans.*

Proof. Each floorplan R with $n - 1$ or less faces has at least two child floorplans $R(U, 1, 1)$ and $R(L, 1, 1)$. By the definition of the genealogical tree each vertex with depth $n - 1$ in T_n is a leaf, and the number of those vertices is $g(n)$. Thus T_n has at most $2 \cdot g(n)$ vertices, so S_n has at most $2 \cdot g(n)$ floorplans.

Modifying our first algorithm so that it output only based floorplans having exactly n faces, which corresponds to leaves of T_n , we can have the following lemma.

Lemma 2. *The modified algorithm uses $O(n)$ space and output based floorplans having exactly n faces $O(1)$ time per floorplan without duplications.*

We modify the algorithm further so that it output all (non-based) floorplans having exactly n faces, as follows.

At each leaf v of the genealogical tree T_n , the floorplan R corresponding to v is checked whether the removing sequence of R with the base is the lexicographically first one among the four based floorplans each of which is derived from R by rotating R and then choosing the bottom line segment as the base, and only if R has the lexicographically first one R is output. Thus we can output only the canonical representative of each isomorphism class.

Theorem 2. *The modified algorithm uses $O(n)$ space and runs in $O(n \cdot h(n))$ time, where $h(n)$ is the number of non-isomorphic (non-based) floorplans having exactly n faces.*

Proof. Given a based floorplan R , we can find the removing sequence in $O(n)$ time. For each floorplan corresponding to a leaf of T_n , we construct four removing sequences, and find the lexicographically first one in $O(4n)$ time, and for each output floorplan our tree contains at most four isomorphic ones corresponding to the four choices of the base. Thus the algorithm runs in $O(n \cdot h(n))$ time. The algorithm clearly uses $O(n)$ space. $\square$

6 Conclusion

In this paper we have given three simple algorithms to generate all graphs with some property. Our algorithms first define a genealogical tree such that each vertex corresponds to each graph with the given property, then output each graph without duplications by traversing the tree.

To find other all-something-generating problems to which our method can be applied is remained as an open problem.

References

1. D. Avis, *Generating rooted triangulations without repetitions*, Algorithmica, 16, (1996) 618-632. 107, 108
2. T. Beyer and S. M. Hedetniemi, *Constant time generation of rooted trees*, SIAM J. Comput., 9, (1980) 706-712. 107
3. L. A. Goldberg, *Efficient algorithms for listing combinatorial structures*, Cambridge University Press, New York, (1993). 107, 108
4. D. L. Kreher and D. R. Stinson, *Combinatorial algorithms*, CRC Press, Boca Raton, (1998). 107
5. Z. Li and S. Nakano, *Efficient Generation of Plane Triangulations without Repetitions*, Proc. of ICALP 2001, LNCS 2076, (2001) 433-443. 108
6. B. D. McKay, *Isomorph-free exhaustive generation*, J. of Algorithms, 26, (1998) 306-324. 107
7. R. A. Wright, B. Richmond, A. Odlyzko and B. D. McKay, *Constant time generation of free trees*, SIAM J. Comput., 15, (1986) 540-548. 107

On Min-Max Cycle Bases

Giulia Galbiati

Dipartimento di Informatica e Sistemistica, Università degli Studi di Pavia
Via Ferrata 1, 27100 Pavia, Italy
galbiati@unipv.it
<http://dimat.unipv.it/~galbiati>

Abstract. An undirected biconnected graph G with non negative weights on the edges is given. In the *cycle space* associated with G , a subspace of the vector space of G , we define as weight of a basis the maximum among the weights of the cycles of the basis. The problem we consider is that of finding a basis of minimum weight for the cycle space. It is easy to see that if we do not put additional constraints on the basis, then the problem is easy and there are fast algorithms for solving it. On the other hand if we require the basis to be fundamental, i.e. to consist of the set of all fundamental cycles of G with respect to the chords of a spanning tree of G , then we show that the problem is NP-hard and cannot be approximated within $2 - \epsilon, \forall \epsilon > 0$, even with uniform weights, unless $P=NP$. We also show that the problem remains NP-hard when restricted to the class of complete graphs; in this case it cannot be approximated within $13/11 - \epsilon, \forall \epsilon > 0$, unless $P=NP$; it is instead approximable within 2 in general, and within $3/2$ if the triangle inequality holds.

1 Introduction

Let $G = (V, E)$ be an undirected graph having m edges and n vertices. It is known that there is associated with G a vector space over $GF(2)$, of dimension m , consisting of all subsets of E , including the empty set. An important subspace of this vector space is the *cycle space*, consisting of all circuits (including the null circuit) and all unions of edge-disjoint circuits of G . If p denotes the number of connected components of G then the dimension of this subspace is known to be $m - n + p$, called the *nullity* or *cyclomatic number* of G .

In this paper we are interested in finding *cycle bases*, i.e. bases for the cycle space, of an undirected biconnected graph G . When G is connected there are special cycle bases that can be derived from the spanning trees of G , which we call *fundamental cycle bases*. If T is a spanning tree of G having branches denoted by $b_1, b_2, \dots, b_{n-1}$ and chords denoted by $c_1, c_2, \dots, c_{m-n+1}$ then the set of cycles obtained by inserting into T each chord, which is called the set of fundamental cycles with respect to T , is a fundamental cycle basis.

Cycle bases have been used in graph analysis, to examine the cyclic structure of a graph, in theory of algorithms, to analyze algorithms, and in other theoretical contests. Cycle bases have had practical applications instead in electrical

networks, since the time of Kirchoff, and in biology; fundamental cycle bases have been used in the frequency analysis of computer program [4], and also by organic chemists interested in the coding of ring compounds [6].

If the edges in G have non negative weights and the weight of a basis is defined as the sum of the weights of its cycles, then the problem of finding cycle bases of minimum weight has been extensively studied both for general bases and for fundamental bases. For general bases the problem is easy and [3] gives the first polynomial time algorithm for its solution; in [2] instead it is shown that for fundamental bases the problem is NP-hard and a number of polynomial time heuristic algorithms, which yield approximate solutions, are given together with a discussion of their performances.

In this paper we address the problem of finding cycle bases and fundamental cycle bases of minimum weight in a weighted undirected biconnected graph, using a different measure of the weight of a basis, i.e. the maximum among the weights of the cycles in the basis. This measure is new, and interesting both in a theoretical contest and in the practical context of electrical and communication networks. In particular, a simple example can come from the study of electrical networks, where the use of the new measure for a fundamental cycle basis allows to simplify the auxiliary algebraic operations needed to solve the network graph.

We show that for general bases the problem is easy and there are fast algorithms for solving it. On the other hand if we require the basis to be fundamental, then we show that the problem is NP-hard and cannot be approximated within $2 - \epsilon, \forall \epsilon > 0$, even with uniform weights, unless $P=NP$. We also show that the problem remains NP-hard when restricted to the class of complete weighted graphs; in this case it cannot be approximated within $\frac{13}{11} - \epsilon, \forall \epsilon > 0$, unless $P=NP$; it is instead approximable within 2 in general and within $3/2$ if the triangle inequality holds.

Various interesting problems remain open; they are highlighted in the various sections.

2 Definition and Results

Throughout this paper all graphs G are finite, undirected, without loops or multiple edges; moreover, since the cycle space of a graph is the direct sum of the cycle spaces of its 2-connected components, we assume G to be biconnected. If weights on the edges of G are given, they are non-negative integer numbers.

The *nullity*, or *cyclomatic number*, $\nu(G)$ of G is therefore equal to $m - n + 1$, where m and n denote the numbers of edges and vertices of G .

As explained in the introduction the weight of a cycle basis is defined to be the maximum among the weights of its cycles.

Let $B = \{b_1, \dots, b_{m-n+1}\}$ be a cycle basis for G and let $C \subseteq B$. Denote by G_C the subgraph of G consisting of the cycles in C .

In [5] two useful characterizations are given for fundamental cycle bases.

Theorem 1. *A cycle basis B of G is fundamental if and only if B contains no cycle which consists entirely of edges belonging to other cycles of B .*

Theorem 2. *A cycle basis B of G is fundamental if and only if $\nu(G_C) = |C|$ for every $C \subseteq B$.*

3 Generating Optimum Cycle Bases

In this section we address the problem of finding a cycle basis B of minimum weight in a weighted graph G . Theorem 3 contains the main result. If for any two vertices of G there is a unique shortest path joining them, then the following algorithm solves the problem; otherwise one may use standard perturbation techniques (see for instance [1], page 8) in order to guarantee uniqueness.

Algorithm 1.

- (1) Find the shortest path $P(x, y)$ between each pair of vertices x, y .
- (2) For each vertex v and edge $\{x, y\}$ in graph G , create the cycle $C(v, x, y) = P(v, x) + P(v, y) + \{x, y\}$, and calculate its weight. Degenerate cases in which $P(v, x)$ and $P(v, y)$ have vertices other than v in common can be omitted.
- (3) Order the cycles by increasing weights.
- (4) Use the greedy algorithm to find from this reduced set of cycles an optimum cycle basis.

Theorem 3. *Algorithm 1 finds a cycle basis of minimum weight in a weighted graph G if any two vertices of G are joined by a unique shortest path.*

Proof. Since a vector space is also a matroid it follows that the cycle space of graph G is a matroid. It is known that in a matroid the greedy algorithm finds a basis that simultaneously minimizes the sum of the weights of its elements and the maximum among the weights of its elements. Using Theorem 4 in [3] we may deduce that, if all shortest paths in G are unique, then the reduced set of cycles used by Algorithm 1 contains all the cycles appearing in any basis that minimizes the sum of the weights of its elements. This reduced set of cycles, being a subset of a vector space, is also a matroid; it follows easily that the greedy algorithm in step (4) finds a basis for the cycle space of G that minimizes the maximum among the weights of its elements. $\square$

4 Generating Optimum Fundamental Cycle Bases

In this section we address the interesting problem of finding a fundamental cycle basis B of minimum weight in a weighted graph G . In the first subsection we consider the general case, in the following one we devote our attention to the special case of complete graphs. We will obtain the various results by exhibiting three reductions, of increasing complexity, from a well-known NP-complete problem, the Satisfiability problem.

4.1 In General Graphs

The first result, given by the next theorem, shows that the problem is NP-hard even when restricted to graphs having uniform weights.

Theorem 4. *The problem of finding a fundamental cycle basis B of minimum weight in a graph G is NP – hard.*

Proof. We prove the theorem by exhibiting a reduction of the Satisfiability problem to the recognition form of our problem. Given an instance I of Satisfiability, i.e. a CNF formula F on a set U of boolean variables, we define an instance for the recognition form of the problem, i.e. a graph G and an integer k , such that I is satisfiable iff there exists in G a fundamental cycle basis of weight at most k .

Let I be a collection $C = C_1, \dots, C_h$ of h disjunctive clauses of literals, where a literal is a variable or a negated variable in $U = \{u_1, \dots, u_n\}$.

First we define a graph G having arcs with weights equal to 1 or to a large integer M to be defined later and we prove the result for this graph; then we observe that the result is not affected if we replace each arc having weight M with a chain of M arcs of unitary weight.

We start the construction of G from the graph $G^\square$ given in Fig. 1 where the only weights indicated are those equal to M

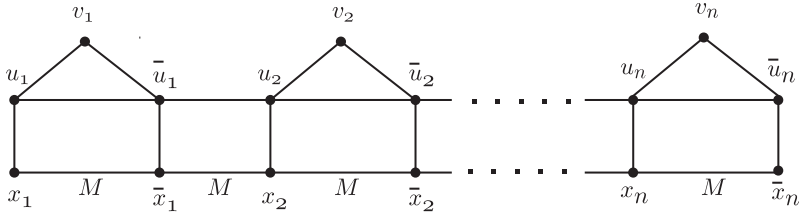


Fig. 1. Graph G'

Then, in order to obtain G from G' , for each clause C_i we add to G' two vertices c_i and c^i and the edge $\{c_i, c^i\}$ with weight equal to M ; moreover if C_i contains the variable u_j or its negation we add the edge $\{c^i, v_j\}$; finally if C_i contains the variable u_j (resp. $\bar{u}_j$) we add the edge $\{c_i, u_j\}$ (resp. $\{c_i, \bar{u}_j\}$) (See Fig.2).

We complete the reduction by setting k equal to $M + 3$.

Now if I is satisfiable there exists a truth assignment for U that satisfies each clause; we show that we can find a spanning tree T of G having a fundamental set of cycles of weight at most $M + 3$. We start the construction of tree T from the tree T' consisting of the edges $\{u_j, \bar{u}_j\}, \{u_j, x_j\}, \{\bar{u}_j, \bar{x}_j\}$, for all $j = 1, \dots, n$ and of the edges $\{\bar{u}_j, u_{j+1}\}$, for all $j = 1, \dots, n - 1$. Then in order to obtain T we add to T' , for each variable u_j set to true (resp. false) in the assignment, the

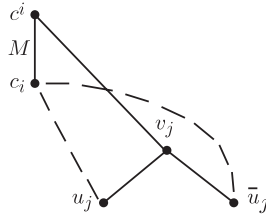


Fig. 2. Additional edges and vertices for clause C_i

edge $\{v_j, u_j\}$ (resp. $\{v_j, \bar{u}_j\}$); moreover for each clause C_i we choose a literal that satisfies the clause and if the chosen literal is variable u_j (resp. negated variable $\bar{u}_j$) we also add the edges $\{c^i, v_j\}$ and $\{c_i, u_j\}$ (resp. $\{c^i, v_j\}$ and $\{c_i, \bar{u}_j\}$). It is easy to verify that, if M is chosen to satisfy the inequality $2n + 3 \leq M + 3$, the set of fundamental cycles with respect to T has cycles of weight at most $M + 3$.

Conversely, suppose that there exists in G a fundamental cycle basis of weight at most $M + 3$, with $M = 2n$. Observe that all cycles that are fundamental cycles with respect to the chords of T' (these chords have weight equal to M) must belong to the basis; moreover for each clause C_i the edge $\{c_i, c^i\}$ must belong to a cycle in the basis that goes through a vertex v_j , for some $j = 1, \dots, n$; call this cycle A_j (resp. $\bar{A}_j$) if it goes also through vertex u_j (resp. $\bar{u}_j$). It is crucial to notice that all cycles of the basis containing the edges $\{c_i, c^i\}$, for all $i = 1, \dots, h$, cannot contain both A_j and $\bar{A}_j$, for some index j , otherwise Theorem 2 would be violated: the A_j and $\bar{A}_j$ plus the cycle that goes through the vertices $\{u_j, \bar{u}_j, x_j, \bar{x}_j\}$ would represent a set S of cycles such that $\nu(G_S) = |S| + 1$, since the additional cycle through the vertices $\{u_j, v_j, \bar{u}_j\}$ would be generated. Now it is easy to conclude that all the A_j or $\bar{A}_j$ containing the edges $\{c_i, c^i\}$, for each $i = 1, \dots, h$, allow to identify a truth assignment for U that satisfy all clauses in I .

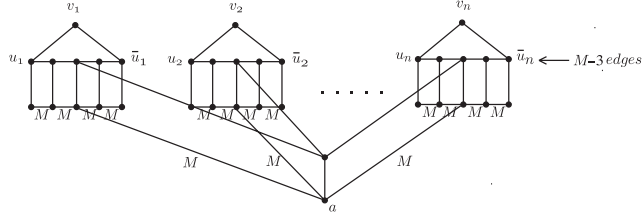
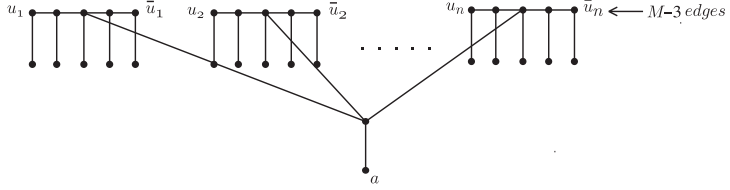
The conclusion follows if, as already said, each arc having weight equal to M is replaced with a chain of M arcs of unitary weight. $\square$

The following theorem proves a non-approximability result for our problem, again for graphs with uniform weights.

Theorem 5. *The problem of finding a fundamental cycle basis B of minimum weight in a graph G cannot be approximated within $2 - \epsilon$, $\forall \epsilon > 0$, unless $P=NP$.*

Proof. We prove the theorem by giving a more sophisticated reduction from Satisfiability to the optimization form of our problem, which exhibits a gap. More precisely, we show that yes-instances of Satisfiability are mapped into instances that have an optimum solution of weight at most $M + 3$, whereas no-instances are mapped into instances that have an optimum solution of weight at least $2M$. From this we will be able to conclude that the problem cannot be approximated within $2 - \epsilon$, $\forall \epsilon > 0$, unless $P=NP$.

The reduction constructs a graph G from a starting graph in a way identical to the one used in Theorem 4; here the starting graph is not G' but the graph G''

Fig. 3. Graph G'' Fig. 4. Tree T''

given in Fig. 3; the addition of edges and vertices to G'' is identical to that done to G' .

It is easy to see that yes-instances of Satisfiability are mapped into instances that exhibit a fundamental cycle basis of weight at most $M + 3$ and hence into instances whose optimum weight is at most $M + 3$. Such a basis is the set of fundamental cycles with respect to the tree T built, by the procedure used in Theorem 4, starting from the tree T'' illustrated in Fig. 4. The only necessary requirement for M is to satisfy $M - 3 \geq 2$.

We now show that if G had a fundamental cycle basis where all cycles have weight less than $2M$, then it would be possible to satisfy instance I . In fact, in such a case, all cycles that are fundamental with respect to those chords of T'' which have weight equal to M and no vertex a as an endpoint should belong to the basis; we group these cycles naturally in n groups, called $B_1, \dots, B_n$. Observe now that the cycles of the basis that include the edges $\{c_i, c^i\}$, for all $i = 1, \dots, h$, could not include both edges $\{v_j, u_j\}$ and $\{v_j, \bar{u}_j\}$ for some j , otherwise Theorem 2 would be violated, because of the cycles in B_j ; and this would be sufficient to identify a true assignment satisfying instance I . Hence no-instances are mapped into instances whose optimum weight is at least $2M$. At this point we may conclude that the problem cannot be approximated within $\frac{2M}{M+3} - \epsilon'$, $\forall \epsilon' > 0$, unless $P=NP$. It follows that $\forall \epsilon > 0$, if we choose ϵ' to be less than or equal to $\frac{\epsilon}{2}$ and M is chosen in such a way that $\frac{6}{M+3} \leq \frac{\epsilon}{2}$, then the inequality $\frac{2}{1+3/M} - \epsilon' \geq 2 - \epsilon$ becomes true and the conclusion follows. $\square$

It is an open problem to decide whether approximation within 2 is possible in the uniform case; one also wonders if even stronger non-approximability results hold in the weighted case.

4.2 In Complete Graphs

Of course the problem of finding an optimum fundamental cycle basis in a complete graph with uniform weights is easy, since each star gives an optimum solution. Hence it makes sense to consider only the weighted case. The next two theorems show that the case of complete weighted graphs is just as interesting as the case of uniform general graphs.

Theorem 6. *The problem of finding a fundamental cycle basis B of minimum weight in a complete weighted graph G is NP – Hard.*

Proof. This proof is based on a third reduction from Satisfiability. This time the starting graph G''' , from which graph G is built, is illustrated in Fig.5; as usual only weights different from 1 are indicated in the figure.

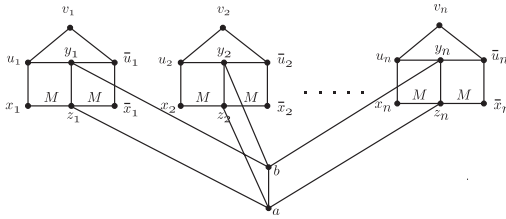


Fig. 5. Graph G'''

Graph G is built from G''' by first adding edges and vertices as specified in the proofs of the preceding theorems and as illustrated in Fig. 2, with the same weights on the edges, then by completing the resulting graph with edges of weights that we now specify. Precisely, the weights assigned are set equal to $y = 4$ for all edges $\{u_j, z_j\}$, $\{x_j, y_j\}$, $\{y_j, \bar{x}_j\}$, $\{z_j, \bar{u}_j\}$, $\{b, z_j\}$, $\{a, y_j\}$, $j = 1, \dots, n$, and for all edges $\{c_i, v_j\}$ if clause C_i contains variable u_j or its negation, and all edges $\{c^i, u_j\}$ (resp. $\{c^i, \bar{u}_j\}$) if C_i contains the variable u_j (resp. $\bar{u}_j$); the remaining edges receive weights set equal to $z = 3$.

Following the lines of the preceding proofs it is now not difficult to conclude that instance I of Satisfiability is satisfiable iff there exists in G a fundamental cycle basis of weight at most $M+3$, with $M \geq 8$, since the inequality $z+8 \leq M+3$ must hold true. $\square$

Theorem 7. *The problem of finding a fundamental cycle basis B of minimum weight in a complete weighted graph G cannot be approximated within $\frac{13}{11} - \epsilon$, $\forall \epsilon > 0$, unless $P=NP$.*

Proof. We show that the reduction given in the preceding theorem, when seen as a reduction from Satisfiability to the optimization form of our problem, exhibits a gap. More precisely, yes-instances of Satisfiability are mapped into instances that have an optimum solution of weight at most $M + 3$, with $M \geq 8$, whereas no-instances are mapped into instances that have an optimum solution of weight at least $M + 5$. From this we are able to conclude that the problem cannot be approximated within $\frac{M+5}{M+3} - \epsilon, \forall \epsilon > 0$, unless $P=NP$, and the conclusion follows if we set $M = 8$. $\square$

Finally we are able to give some simple approximability results. It remains an open problem to find similar results for non-complete graphs.

Theorem 8. *The problem of finding a fundamental cycle basis B of minimum weight in a complete weighted graph G can be approximated within 2 in general and within $\frac{3}{2}$ if the weights satisfy the triangle inequality.*

Proof. Let T be a spanning tree of G having minimum diameter D^* . Let W be the weight of the set of fundamental cycles with respect to T ; of course $W \leq D^* + w$, where w denotes the maximum weight of an edge of G . Let W^* denote the weight of an optimum fundamental cycle basis for G and let D denote the diameter of the spanning tree of G whose set of fundamental cycles is the optimum basis. If G is complete, of course $W^* \geq D$; hence $W \leq D^* + w \leq D + w \leq W^* + w$. Since $w \leq W^*$, and $w \leq \frac{1}{2}W^*$ if the triangle inequality holds, the conclusion follows. $\square$

Acknowledgment

I wish to express my heartfelt thanks to Francesco Maffioli and Edoardo Amaldi for drawing my attention to these problems, for generously devoting time to discussing various aspects of their inner structure and for carefully examining earlier solutions and preliminary versions of this paper.

References

1. Bley A., Grotchel M. and Wessaly R., Design of Broadband Virtual Private Networks: Model and Heuristics for the B-Win, Pre-print, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 1998. 118
2. Deo N., Prabhu G. M. and Krishamoorthy M. S., Algorithms for Generating Fundamental Cycles in a Graph, ACM Trans. Math. Software, 8 (1982), pp 26-42 117
3. Horton J. D., A polynomial-time algorithm to find the shortest cycle basis of a graph, SIAM J. Computing, vol. 16, n.2 (1987), pp 358-366 117, 118
4. Knuth D. E., The Art of Computer Programming, vol.1, Addison-Wesley, Reading Mass, (1968), pp 363-368 117
5. Syslo M. M., On Cycle Bases of a Graph, Networks, vol 9 (1979) pp 123-132 117
6. Sussenhout E. Jr, A graph theoretical algorithm for matching chemical structures. J. Chem. Doc. 5,1 (1965), pp 36-43 117

On the Minimum Local-Vertex-Connectivity Augmentation in Graphs

Hiroshi Nagamochi and Toshimasa Ishii

Department of Information and Computer Sciences,
Toyohashi University of Technology,
Toyohashi, Aichi, 441-8580, Japan
{naga,ishii}@ics.tut.ac.jp

Abstract. Given a graph G and target values $r(u, v)$ prescribed for each pair of vertices u and v , we consider the problem of augmenting G by a smallest set F of new edges such that the resulting graph $G + F$ has at least $r(u, v)$ internally disjoint paths between each pair of vertices u and v . We show that the problem is NP-hard even if G is $(k - 1)$ -vertex-connected and $r(u, v) \in \{0, k\}$, $u, v \in V$ holds for a constant $k \geq 2$. We then give a linear time algorithm which delivers a $\frac{3}{2}$ -approximation solution to the problem with a connected graph G and $r(u, v) \in \{0, 2\}$, $u, v \in V$.

1 Introduction

The problem of augmenting a given graph $G = (V, E)$ by adding a smallest set F of new edges such that the augmented graph $(V, E \cup F)$, denoted by $G + F$, meets a connectivity requirement is called *the connectivity augmentation problem*. It has many applications and has been studied extensively (see [1] for a survey). The local vertex-connectivity between two vertices is measured by the maximum number of internally disjoint paths between them. In this paper, we consider *the local-vertex-connectivity augmentation problem* (LVCAP for short) which asks to find a smallest edge set F such that the local vertex-connectivity between every two vertices u and v in $G + F$ is equal to or larger than the target value $r(u, v)$ prescribed for each pair of $u, v \in V$, where the function r from $V \times V$ to the set Z^+ of nonnegative integers is called a *target function*. Not many algorithms have been developed to the LVCAP. To solve the LVCAP with a target function r such that $r(u, v) \in \{0, 2\}$, K. Tsuchiya *et al.* [10] proposed an algorithm which computes an optimal solution in $O(B_n(n + m))$ time, where n and m denote the numbers of vertices and edges in G , respectively and B_t is the Bell number of a t -element set (which is exponential in t). Based on a primal-dual approach, R. Ravi and D. P. Williamson [8] gave a 3-approximation algorithm to the LVCAP with a target function r such that $r(u, v) \in \{0, 1, 2\}$ (where their algorithm remains applicable to the edge-weighted case). In particular, their algorithm delivers a 2-approximation solution if G is connected. T. Jordán [6] proved the NP-hardness of the LVCAP in the case where a given graph $G = (V, E)$ is $(\frac{n}{2})$ -vertex-connected and a target function r satisfies $r(u, v) \in \{0, \frac{n}{2} + 1\}$, $u, v \in V$.

However, as pointed out in [7], the complexity of the problem is not known if target values $r(u, v)$, $u, v \in V$ are independent of n .

We here briefly mention about the complexity results known to other local-connectivity augmentation problems. The local edge-connectivity augmentation problem (LECAP for short) asks to augment a graph (or digraph) G by a smallest set F of new edges such that the local edge-connectivity between each pair of vertices u and v (i.e., the maximum number of edge-disjoint paths from u to v) in $G + F$ becomes equal to or larger than the target value $r(u, v)$. In the case of graphs, A. Frank [2] proved that the problem can be solved in polynomial time. However, for digraphs, he showed that the LVCAP and the LECAP are both NP-hard even if the target function satisfies $r(u, v) \in \{0, 1\}$ for all pairs $u, v \in V$ [2]. So among the local-connectivity augmentation problems, the complexity status is left open only for the LVCAP in graphs with small target values (such as $r(u, v) \in \{0, 2\}$).

In this paper, we first show that the LVCAP in a graph with $r(u, v) \in \{0, k\}$, $u, v \in V$ is NP-hard for any constant $k \geq 2$. We then consider designing an approximation algorithm to the LVCAP for a connected graph G with a target function r such that $r(u, v) \in \{0, 2\}$. By using a graph theoretic approach, we present a $\frac{3}{2}$ -approximation algorithm to the problem.

2 NP-Hardness of LVCAP

In this section, we prove the next result.

Theorem 1. *Given a $(k - 1)$ -vertex-connected graph $G = (V, E)$ and a target function $r : \binom{V}{2} \rightarrow \{0, k\}$ for an integer $k \geq 2$, the problem of testing whether there is a solution F to the LVCAP with size $|F|$ equal to or smaller than a specified value is NP-hard.* $\square$

For a subset $X \subset V$, let $\Gamma_G(X)$ denote the set of vertices in $V - X$ which are adjacent to a vertex in X . For a subset $Y \subset V$, let $G - Y$ denote the graph resulting from G by deleting vertices in Y together with edges incident to them, and we say that Y *separates* two vertices x and y if x and y belong to different components in $G - Y$. For two vertices $u, v \in V$, let $\kappa_G(u, v)$ denote the maximum number of internally disjoint paths between u and v , which is equal to the minimum size of a subset $Y \subseteq V - \{u, v\}$ that separates u and v if u and v are not adjacent. A singleton set $X = \{x\}$ may be written as x .

We prove the NP-hardness of the LVCAP by reducing from the following problem, which is known to be NP-complete in the strong sense [3, p.224].

3-PARTITION

INSTANCE: (A, B, σ) with a set A of $3m$ elements, an integer $B \in \mathbb{Z}^+$, and a size $\sigma(a) \in \mathbb{Z}^+$ for each $a \in A$ such that $B/4 < \sigma(a) < B/2$ and such that $\sum_{a \in A} \sigma(a) = mB$.

QUESTION: Can A be partitioned into m disjoint sets $A_1, A_2, \dots, A_m$ such that, for $1 \leq i \leq m$, $\sum_{a \in A_i} \sigma(a) = B$ (note that each A_i must therefore contain exactly three elements from A)? $\square$

The strong NP-hardness of the 3-PARTITION is shown in [3] by a polynomial transformation from the 3-dimensional matching problem (3DM) which is one of the NP-complete problems in such a way that, given an instance I_{3DM} of the 3DM, $B = \frac{1}{m} \sum_{a \in A} \sigma(a)$ in the resulting instance I_{3PTN} of the 3-PARTITION are bounded from above by a polynomial of the input size of the I_{3DM} . (Thus, any pseudo-polynomial algorithm for the 3-PARTITION implies a polynomial algorithm for the 3DM.) Therefore, to prove the NP-hardness of the LVCAP, it suffices to transform an instance I_{3PTN} of the 3-PARTITION to an instance I_L of the LVCAP so that the time of the transformation and the size of I_L are bounded from above by a polynomial of m and B .

Take an instance $I_{3PTN} = (A, B, \sigma)$, where $A = \{a_1, \dots, a_{3m}\}$, of the 3-PARTITION; $\sigma(a) \geq 3$ for all $a \in A$ is assumed without loss of generality (if necessary we increase each $\sigma(a)$ by 2 and B by 6). From the I_{3PTN} , we construct an instance $I_L = (G = (V, E), r)$ of the LVCAP. The vertex set V of G is given by the union of $5m + 1$ subsets, $S, U_i, 1 \leq i \leq 3m$, and $W_j, Z_j, 1 \leq j \leq m$, which are defined as follows. Let $S = \{s_1, s_2, \dots, s_{k-1}\}$. Associated with each element $a_i \in A$, let $U_i = \{u_i^1, u_i^2, \dots, u_i^{\sigma(a_i)}\}$. Associated with each $j \in \{1, 2, \dots, m\}$, let $W_j = \{w_j^1, \dots, w_j^{k-1}\}$ and $Z_j = \{z_j^1, \dots, z_j^B\}$. The edge set E of G consists of edges (x, y) such that

$$x \in W_j \text{ and } y \in (W_j - x) \cup Z_j \text{ for each } j = 1, 2, \dots, m$$

and edges (x', y') such that

$$x' \in S \text{ and } y' \in (U_1 \cup \dots \cup U_{3m}) \cup (W_1 \cup \dots \cup W_m).$$

Observe that the resulting graph G is $(k - 1)$ -vertex-connected. The target function r is given by $r(x, y) = k$ for all pairs of vertices $x, y \in X$ such that $X \in \{U_1, U_2, \dots, U_{3m}, Z_1, Z_2, \dots, Z_m\}$; $r(x, y) = 0$ otherwise. Clearly I_L can be constructed in time polynomial in m and B . Then the remaining task is to show that I_{3PTN} has the desired partition if and only if I_L has a solution.

We prove that there is a set F of at most mB new edges such that $\kappa_{G+F}(u, v) \geq r(u, v)$ for all $u, v \in V$ if and only if the instance $I_{3PTN} = (A, B, \sigma)$ of the 3-PARTITION has a desired partition.

If part: Let $A_1, A_2, \dots, A_m$ be m disjoint subsets of A such that $\sum_{a \in A_h} \sigma(a) = B$ for $1 \leq h \leq m$. Associated with each $A_h, 1 \leq h \leq m$, we set M_h to be a matching of B new edges (u, z) such that $u \in U_i$ with $a_i \in A_h$ and $z \in Z_h$. Let $F = \cup_{1 \leq h \leq m} M_h$, where $|F| = mB$. It is not difficult to see that $\kappa_{G+F}(u, v) \geq r(u, v)$ for all $u, v \in V$.

Only-if part: Suppose that there is an edge set F such that $|F| \leq mB$ and $\kappa_{G+F}(u, v) \geq r(u, v)$ for all $u, v \in V$. Let $U = \cup_{1 \leq i \leq 3m} U_i$ and $Z = \cup_{1 \leq j \leq m} Z_j$. By $|G_G(v)| = k - 1$ ($v \in U \cup Z$) and $|U \cup Z| = 2mB$, F must be a perfect matching on $U \cup Z$. Then every edge $e \in F$ must join a vertex in U and a vertex in Z , because otherwise if an edge in F joins two vertices $u \in U_i$ and $u' \in U_{i'}$ (where possibly $i = i'$) then u cannot be connected to any other vertex $u'' \in U_i - \{u, u'\}$ (where $|U_i| = \sigma(a_i) \geq 3$) in the graph $G + F - S$, contradicting $\kappa_{G+F}(u, u') \geq k$.

Now we claim that for each U_i all edges in F incident to U_i must be incident to the same Z_j . Assume indirectly that F contains edges (u, z) and (u', z') such that $u, u' \in U_i$, $z \in Z_j$ and $z' \in Z_{j'}$ ($j \neq j'$). In the graph $G + F - S$, $W_j \cup Z_j$ belongs to the same component, whose vertex set consists of $W_j \cup Z_j$ and vertices in U which are adjacent Z_j via edges in F . Similarly for $W_{j'} \cup Z_{j'}$. This implies that u and u' would belong to different components in $G + F - S$, contradicting $\kappa_{G+F}(u, u') \geq k$. This proves the claim. By letting $A_h = \{a_i \mid \text{the vertices in } U_i \text{ are incident to } Z_h\}$, $h = 1, \dots, m$, we obtain a desired partition to the instance I_{3PTN} of 3-PARTITION.

This completes the proof of Theorem 1.

3 A $\frac{3}{2}$ -Approximation Algorithm

3.1 Main Theorem

Let $G = (V, E)$ be a simple undirected graph. A subset F of new edges is called a *solution* to the LVCAP with (G, r) if $\kappa_{G+F}(u, v) \geq r(u, v)$ for all $u, v \in V$ and $G + F$ remains *simple*. In the sequel, we assume without loss of generality that G is connected, but not 2-vertex-connected and $|V| \geq 3$ holds. Hence G has a vertex $v \in V$ such that $G - v$ has more than one component. Such a vertex is called a *cut-vertex*. An edge is called a *bridge* of G if the graph becomes disconnected by removing the edge. We can assume that $r(u, v) = 0$ or 2 for $u, v \in V$. Let $R = \{(u, v) \mid r(u, v) = 2, u, v \in V\}$ be a set of fictitious edges, called *r-edges*.

We first derive some lower bounds on the optimal value $opt(G, r)$ to the LVCAP with (G, r) . A subset $T \subset V$ with $|\Gamma_G(T)| = 1$ is called an *r-tight set* if it satisfies one of the following (i) and (ii).

- (i) $r(u, v) = 2$ holds for some pair of vertices $u \in T$ and $v \in V - T - \Gamma_G(T)$.
- (ii) for $\{s\} = \Gamma_G(T)$ and some vertices $t, u \in T$, $(t, s) \in E$ is a bridge and $r(u, s) = 2$ holds (where possibly $t = u$ or $V - T - \Gamma_G(T) = \emptyset$).

For any r -tight set T of type (i) (resp., of type (ii)), $V - T - \Gamma_G(T)$ (resp., $V - T$) is also an r -tight set of the same type. We call an inclusionwise minimal r -tight set T an *r-leaf set*. Observe that any r -leaf sets are pairwise disjoint. Let $\alpha(G, r)$ denote the number of all r -leaf sets in G . Any solution F to the LVCAP with (G, r) must contain an edge which joins two vertices from an r -leaf set T and the set $V - T$. Therefore we see

$$opt(G, r) \geq \lceil \alpha(G, r)/2 \rceil.$$

For a cut-vertex s in G , we denote by $\mathcal{C}_G(s)$ be the set of all components in $G - s$, and by $\mathcal{C}_G(s, r)$ the set of components in $\mathcal{C}_G(s)$ containing at least one r -leaf set. We partition $\mathcal{C}_G(s, r)$ as follows. We say that two components $H, H' \in \mathcal{C}_G(s, r)$ are *r-linked* if they are contained in the same component in $(G + R) - s$. Consider an inclusionwise maximal subset $\mathcal{H}' \subseteq \mathcal{C}_G(s, r)$ such that

any two components $H, H' \in \mathcal{H}'$ are r -linked. Then $\mathcal{C}_G(s, r)$ is uniquely partitioned into such maximal subsets, say $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_p$. Note that any solution contains at least $|\mathcal{H}_i| - 1$ edges that join the components in $\mathcal{H}_i$ into a single component. Let $\beta(G, r, s) = \sum_{1 \leq i \leq p} (|\mathcal{H}_i| - 1)$ and $\beta(G, r) = \max\{\beta(G, r, s) \mid \text{cut-vertices } s \text{ in } G\}$. Then we have

$$\text{opt}(G, r) \geq \beta(G, r).$$

In this section, we prove the next result.

Theorem 2. *For a connected graph $G = (V, E)$ with $|V| \geq 3$ and a set R of r -edges, there exists a set F of new edges such that $\kappa_{G+F}(u, v) \geq 2$ holds for all $(u, v) \in R$ and $|F| \leq \lceil \alpha(G, r)/2 \rceil + \frac{1}{2}\beta(G, r) (\leq \frac{3}{2}\text{opt}(G, r))$ holds. Moreover such an F can be found in $O(|E| + |R|)$ time. $\square$*

3.2 Eliminating Non- r -leaf Sets

A subgraph of G induced by a subset $X \subseteq V$ is denoted by $G[X]$. An induced subgraph $G[X]$ is called a *block* if no two vertices in X are separated by any cut-vertex and X is maximal subject to this property. For any two blocks $G[X]$ and $G[Y]$, $E(G[X]) \cap E(G[Y]) = \emptyset$, $|X \cap Y| \leq 1$, and a vertex in $X \cap Y$ (if any) is a cut-vertex. Observe that $\kappa_G(u, v) \geq 2$ holds for any two vertices u and v in a block $G[X]$ unless $|X| = 2$, i.e., $G[X]$ is a single edge (i.e., a bridge). A *block-cut tree* $\mathcal{T}$ of a connected graph G is obtained by replacing each block of G with a single vertex adjacent to the cut-vertices in the block. More formally, $\mathcal{T}$ is constructed as follows. Let V_c be the set of all cut-vertices of G . Starting with the vertex set V_c but no edges, we create a new vertex v_X associated with each block $G[X]$ of G , joining v_X with all cut-vertices in $V_c \cap X$ via new edges. Let $\mathcal{T} = (\mathcal{V} = V_c \cup V_b, \mathcal{E})$ be the resulting tree, where V_b denotes the set of vertices v_X associated with each block $G[X]$ in G , and $\mathcal{E}$ is the set of new edges. The block-cut tree of G can be obtained in linear time, since all blocks and cut-vertices can be identified by the depth-first search [9].

A subset $T \subset V$ is called a *tight set* if $|\Gamma_G(T)| = 1$ and $V - T - \Gamma_G(T) \neq \emptyset$. An inclusionwise minimal tight set T is called a *leaf set*, which always induces an connected subgraph. Any leaf sets are pairwise disjoint. For a subset $Z \subseteq V$, let $\mathcal{L}_G(Z)$ be the set of all leaf sets T in G such that $T \subseteq Z$. For a subgraph H of G and its vertex set $V(H)$, $\mathcal{L}_G(V(H))$ may be written as $\mathcal{L}_G(H)$. For each leaf set $T \in \mathcal{L}_G(V)$, $G[T \cup \Gamma_G(T)]$ is a block of G . Note that for a leaf set T and an r -leaf set T' , if $T \cap T' \neq \emptyset$ implies $T \subseteq T'$. We now observe the following two properties on leaf sets and r -leaf sets.

Lemma 1. *Let F be a solution to the LVCAP with (G, r) . Then there is a solution F^* with $|F^*| \leq |F|$ such that any edge in F^* joins two vertices from distinct leaf sets in G .*

Proof: Let us assume that F is minimal (i.e., any proper subset of F is not a solution). Clearly F contains no edge joining two vertices from the same leaf

set. Let F contain an edge $e = (u, v) \in F$ such that v does not belong to any leaf set in $\mathcal{L}_G(V)$. Since $F - e$ is not a solution, $\kappa_{G+F}(u, v) \geq r(u, v) = 2 > \kappa_{G+F-e}(u, v) \geq 1$. Thus there is an r -tight set X such that $v \in X$ and $u \in V - X - \Gamma_G(X)$ (note that $G + F$ remains simple). Let X be inclusionwise minimal subject to this property, and let $\{z\} = \Gamma_G(X)$. There is a leaf set T_X in $\mathcal{L}_G(X)$, where $v \notin T_X$. Choose a vertex $t \in T_X$, for which $G[X \cup \{z\}]$ has a path P which connects t and z passing through v (since otherwise there would be an r -tight set $X' \subset X$ with $v \in X'$, contradicting the minimality of X). For a new edge $e' = (u, t)$, we claim that $F' = (F - e) \cup \{e'\}$ is also a solution to (G, r) . Since $G[V - X]$ contains a path P' between z and u , $G + F'$ contains a cycle C consisting of P' , $\{e'\}$ and the above path P . Assume indirectly that there are vertices x and y with $\kappa_{G+F}(x, y) \geq r(x, y) = 2 > \kappa_{G+F'}(x, y) = 1$. Thus, $G + F'$ has a bridge (x, y) (or a cut-vertex w) such that $e = (u, v)$ has connected the two components H_x and H_y in $G + F - (x, y)$ (or in $G + F - w$). This, however, contradicts that cycle C in $G + F'$ contains u and v .

By repeating this, we can obtain a solution F^* in which every edge joins two vertices from distinct leaf sets in $\mathcal{L}_G(V)$. $\square$

In what follows, we assume that G has at least *three* r -leaf sets (otherwise if there are only two r -leaf sets T_1 and T_2 , then by $|V| \geq 3$, we can add a new edge $e = (u, v)$ for some $u \in T_1$ and $v \in T_2$ without creating multiple edges; we easily see that $\{e\}$ is an optimal solution). For a function $r : \binom{V}{2} \rightarrow Z^+$ and a subset $X \subseteq V$, we denote by $r|_X$ the function $f : \binom{X}{2} \rightarrow Z^+$ such that $f(u, v) = r(u, v)$ for all $u, v \in X$.

Lemma 2. *Let $G = (V, E)$ have at least three r -leaf sets, and let $T \in \mathcal{L}_G(V)$ be a leaf set, but not an r -leaf set.*

- (i) *Any solution F' to $(G - T, r|_{V-T})$ is also a solution to (G, r) .*
- (ii) *Conversely there is an optimal solution F to (G, r) such that no edge in F is incident to any vertex in T (hence F is a solution to $(G - T, r|_{V-T})$).*

Proof: (i) Let F' be a solution to $(G - T, r|_{V-T})$. Since T is not an r -leaf set, F' remains to be a solution to (G, r) .

(ii) Let F be an optimal solution to (G, r) . By Lemma 1, every edge in F is assumed to join two vertices from distinct leaf sets without loss of generality. Furthermore, we assume that F is chosen so as to minimize the set F_T of edges in F incident to T . Without loss of generality F_T is denoted by $\{e_i = (u_i, v_i) \mid i = 1, 2, \dots, p\}$ such that $u_i \in T$ and $v_i \in T_i$ hold for some tight sets $\{T_1, T_2, \dots, T_p\} \subseteq \mathcal{L}_G(V) - T$ (where $T_i \neq T_j$ holds for $i \neq j$, since F is a smallest solution). Assume $F_T \neq \emptyset$, from which we shall show that there is a solution F^* such that $|F^*| \leq |F|$ holds and F^* has no edge incident to T (which contradicts the assumption on F).

Let $\{t\} = \Gamma_G(T)$. We construct $F^* = (F - F_T) \cup F_T^*$, where F_T^* is defined as follows.

Case-a: $p \geq 2$. Let $F_T^* = \{e_1^* = (t, v_1)\} \cup \{e_i^* = (v_1, v_i) \mid i = 2, 3, \dots, p\}$, where we discard e_1^* from F^* if $(t, v_1) \in E$.

Case-b: $p = 1$ and $\Gamma_G(T_1) \neq \{t\}$. Let $F_T^* = \{e_1^* = (t, v_1)\}$.

Case-c: $p = 1$ and $\Gamma_G(T_1) = \{t\}$. Let T_a be an r -leaf set disjoint with $T \cup T_1$, and v_a be a vertex in T_a (such T_a exists by the assumption that there are at least three r -leaf sets). Let $F_T^* = \{e_1^* = (v_a, v_1)\}$.

In any of the three cases, we can observe that F^* contains no edge incident to T , and that, for each $i = 1, 2, \dots, p$, $G + F_T^*$ contains a cycle C_i containing t, v_1 and v_i . Moreover, we can assume that no two edges $e \in E$ and $e' \in F^*$ are multiple edges, since multiple edges (if any) can be removed without losing the above properties. We here claim that F^* still satisfies the connectivity requirement. Assume indirectly that there are vertices x and y with $r(x, y) = 2 > \kappa_{G+F^*}(x, y)$. Thus, $G + F^*$ has a bridge (x, y) (or a cut-vertex w) such that an edge $e_j = (u_j, v_j) \in F_T$ has joined the two components G_x and G_y in $G + F - (x, y)$ (or in $G + F - w$). Assume without loss of generality $x, u_j \in V(G_x)$ and $y, v_j \in V(G_y)$.

Since u_j and x belong to the same block $G[T \cup \{t\}]$ and cycle C_j contains $\{t, v_1, v_j\}$, the block $G[T \cup \{t\}]$ must be the single edge (x, y) (or $T \subseteq V(G_x)$ and $t = w$ hold). Since T is not an r -tight set, $x \notin T$, and hence we have $x \in V(G_x) - T$ and $t = w$. However, $X = V(G_x) - T$ is an r -tight set, but F has no edge joining X and $T \cup (V - X - \{w\})$, contradicting that F satisfies the connectivity requirement. $\square$

By checking maximal tight sets which contain no r -tight sets by means of the block-cut tree, we can remove all leaf sets that are not r -leaf sets in linear time. We hereafter assume that each leaf set is an r -leaf set in a given (G, r) . Hence $\alpha(G, r) = |\mathcal{L}_G(V)|$.

3.3 Balancedness

To find an approximation solution to a given instance (G, r) , we first characterize structure of blocks and cut-vertices in G by introducing a notion of balancedness as follows.

A set $\{a_1, a_2, \dots, a_p\}$ of nonnegative integers is called *balanced* if $\max_{1 \leq i \leq p} a_i \leq \frac{1}{2} \sum_{1 \leq i \leq p} a_i$, and is called *critically balanced* if $\max_{1 \leq i \leq p} a_i = \lfloor \frac{1}{2} \sum_{1 \leq i \leq p} a_i \rfloor$. Notice that if $\{a_1, a_2, \dots, a_p\}$ is balanced, but not critically balanced, then for any $a_i, a_j > 0$, set $\{a_1, \dots, a_i - 1, \dots, a_j - 1, \dots, a_p\}$ remains balanced.

A set M of edges is called an *edge cover* to a set V of vertices if each vertex in V is incident to at least one edge in M . We here review some result from the graph realization problem studied in [4].

Lemma 3. *For a set V of n vertices, let $X_1, X_2, \dots, X_p$ be partition of V such that $\{|X_1|, |X_2|, \dots, |X_p|\}$ is balanced, where $|X_1| = \max_{1 \leq i \leq p} |X_i|$ is assumed.*

(i) [4] *For an even n , there is an edge cover $M = \{(s_i, t_i) \mid i = 1, 2, \dots, \frac{n}{2}\}$ to V such that, for each $(s_i, t_i) \in M$, s_i and t_i do not belong to the same set X_j .*

(ii) *For an odd n , there is an edge cover $M = \{(s_i, t_i) \mid i = 1, 2, \dots, \frac{n+1}{2}\}$ to V such that, for each $(s_i, t_i) \in M$, s_i and t_i do not belong to the same set X_j , and a vertex in X_1 receives two edges from M .*

(iii) *Such an edge cover M in (i) and (ii) can be constructed in $O(n)$ time.*

Proof: (i) Denote the vertices in V by $v_1, v_2, \dots, v_n$ so that they appear in the sets $X_1, X_2, \dots, X_p$ in this order. Construct a perfect matching $M = \{(v_i, v_{\frac{n}{2}+i}) \mid i = 1, 2, \dots, \frac{n}{2}\}$. Clearly, any edge in M cannot join two vertices in the same X_i unless $v_{\frac{n}{2}}, v_{\frac{n}{2}+1} \in X_i$. For the X_i containing $v_{\frac{n}{2}}$ and $v_{\frac{n}{2}+1}$ (if any), vertices in $X_i \cap \{v_{\frac{n}{2}+1}, v_{\frac{n}{2}+2}, \dots, v_n\}$ are joined with vertices in X_1 since $|X_1| \geq |X_i|$. Hence, M is the desired edge cover.

(ii) By introducing a new vertex v_0 and setting $X'_1 = X_1 \cup \{v_0\}$, we construct a perfect matching M in (i) for the vertices $v_0, v_1, v_2, \dots, v_n$. Then we merge v_0 and v_1 into v_1 , yielding the desired edge cover.

(iii) Immediate from the construction method in the proofs for (i) and (ii). $\square$

We call an edge cover M in the lemma *a minimal edge cover on V with partition constraint $\{X_1, X_2, \dots, X_p\}$* .

We now return to the problem of finding a solution to a (G, r) . A cut-vertex v in G is called *balanced* if, for $\mathcal{C}_G(v) = \{H_1, H_2, \dots, H_p\}$ (the set of the components in $G - v$), the set $\{\ell_1, \ell_2, \dots, \ell_p\}$ of integers $\ell_i = |\mathcal{L}_G(H_i)|$ is balanced. A balancedness of blocks is defined as follows. For a block $G[X]$, let $v_1, v_2, \dots, v_q$ be the cut-vertices v of G such that $v \in X$. For each v_i , let $C_i \in \mathcal{C}_G(v_i)$ be the component with $V(C_i) \supseteq X - v_i$. The block $G[X]$ is called *balanced* if the set $\{\ell_1, \ell_2, \dots, \ell_q\}$ of integers $\ell_i = |\mathcal{L}_G(V) - \mathcal{L}_G(C_i)|$ is balanced. We here observe the following property.

Lemma 4. *If G is connected and has a cut-vertex, then G has a balanced block or a balanced cut-vertex.*

Proof: To prove this, we use the following fact on a centroidal vertex in a tree T [5], which is defined to be a vertex v that minimizes the size of a maximum component in $T - v$. For a centroidal vertex v^* in a tree T (which is uniquely determined if T has an even number of edges), each component in $T - v^*$ contains at most a half number of vertices from the entire set of vertices in T .

We first consider the block-cut tree $\mathcal{T} = (\mathcal{V} = V_c \cup V_b, \mathcal{E})$ of G . Observe that there is a one-to-one correspondence between the set of leaves in $\mathcal{T}$ and the set of leaf sets in G . To prove the lemma, it suffices to show that the tree $\mathcal{T}$ has a balanced cut-vertex x^* .

We then convert $\mathcal{T}$ into another tree $\mathcal{T}^*$ by attaching $n^* = |\mathcal{V}|$ new vertices to each leaf u in $\mathcal{T}$ so that u becomes a non-leaf adjacent to n^* new leaves in the resulting tree $\mathcal{T}^*$ (if necessary, we attach one extra new leaf to some u to make the number of edges of $\mathcal{T}^*$ even). From the above fact, any centroidal vertex v^* in $\mathcal{T}^*$ is a balanced cut-vertex in $\mathcal{T}$, since n^* is sufficiently large. $\square$

The two cases distinguished by Lemma 4 are treated in the next two subsections, respectively.

3.4 Balanced Block

We here consider the case where G has a balanced block $G[X]$. In this case, an optimal solution to the problem can be found in linear time by the following

procedure. Let $v_1, v_2, \dots, v_p$ be the cut-vertices of G contained in the block $G[X]$, and for each v_i , let $C_i \in \mathcal{C}_G(v_i)$ be the component with $V(C_i) \supseteq X - v_i$. Let $\mathcal{L}_i = \mathcal{L}_G(V) - \mathcal{L}_G(C_i)$, and $\Lambda = \{\mathcal{L}_1, \mathcal{L}_2, \dots, \mathcal{L}_p\}$. By the balancedness, $|\mathcal{L}_i| \leq \frac{1}{2} \sum_{1 \leq h \leq p} |\mathcal{L}_h| = \frac{1}{2} |\mathcal{L}_G(V)|$ holds for $i = 1, 2, \dots, p$.

We first choose a set $W \subseteq V$ of $|\mathcal{L}_G(V)|$ vertices each of which is chosen from distinct leaf set in $\mathcal{L}_G(V)$, and partition W into $X_1, X_2, \dots, X_p$ so that all vertices in X_i belong to some leaf set in $\mathcal{L}_i \in \Lambda$. We then construct a minimal edge cover M on W with partition constraint $\{X_1, X_2, \dots, X_p\}$. By Lemma 3, such an M can be constructed in $O(|W|)$ time.

Lemma 5. *For the above edge cover M on W , $\kappa_{G+M}(u, v) \geq 2$ holds for any pair $u, v \in V$ with $r(u, v) = 2$.*

Proof: It suffices to show that $G + M$ is 2-vertex-connected. By construction of M , for each vertex u in a leaf set $T \in \mathcal{L}_G(V)$, $G + M$ has a cycle C which passes through u and at least two vertices from X . Assume that, for some cut-vertex v in G , $G + M - v$ has two components H_1 and H_2 with $X - v \subseteq V(H_2)$. Since H_1 contains a leaf set $T_1 \in \mathcal{L}_G(V)$, there is a vertex $u' \in T_1$ such that $G + M$ has a cycle which passes through u' and some two vertices $x, x' \in X$. This, however, implies that u and a vertex in $\{x, x'\} - v$ remain connected after removal of v , a contradiction. $\square$

3.5 Balanced Cut-Vertex

We then consider the case where G has a balanced cut-vertex s . Note that each component $H \in \mathcal{C}_G(s)$ contains at least one leaf set, which is an r -leaf set. We partition $\mathcal{C}_G(s)$ into maximal subsets $\mathcal{H}_1, \mathcal{H}_2, \dots, \mathcal{H}_p$ such that, for each $\mathcal{H}_i$, any two components $H, H' \in \mathcal{H}_i$ are r -linked. With these notions, we first observe a sufficient condition for a set of new edges to be a solution to a given (G, r) .

Lemma 6. *A set of F such that $G + F$ remains simple is a solution if there is a cut-vertex s of G such that*

- (i) *For each leaf set $T \in \mathcal{L}_G(V)$, F contains an edge joining T with a leaf set T' such that T and T' are contained in distinct components in $\mathcal{C}_G(s)$.*
- (ii) *Any two r -linked components $H, H' \in \mathcal{C}_G(s)$ contained in the same component in the graph $G + F - s$.*

Proof: Let $(x, y) \in R$ be an r -edge with $\kappa_G(x, y) = 1$, for which we distinguish the following two cases (a) and (b).

(a) (x, y) is a bridge of G . Thus the removal of edge (x, y) from G creates two components G_x and G_y ; $s \in V(G_y)$ is assumed without loss of generality. Then G_x contains at least one leaf set $T \in \mathcal{L}_G(V)$, and the T belongs to a component $H_T \in \mathcal{C}(s)$. By (i), an edge in F joins T with a leaf set T' in some $H_{T'} \in \mathcal{C}_G(s) - H_T$, graph $G + F$ contains a cycle which passes through (x, y) and s , proving $\kappa_{G+F}(x, y) \geq 2$.

(b) x and y are separated by a cut-vertex w . We first claim that $w \neq s$. This is trivial if $\{x, y\} \subseteq V(H) \cup \{s\}$ for some $H \in \mathcal{C}_G(s)$. If $x \in V(H)$ and $y \in V(H')$

for distinct $H, H' \in \mathcal{C}_G(s)$, then H and H' are r -linked and they are contained in the same component in $G + F - s$ by (ii) (hence $w \neq s$).

Let $S_1, S_2, \dots, S_q$ be the components in $G - w$; $x \in S_1$ and $s \in S_q$ are assumed without loss of generality. It suffices to show that x and y remain connected in $G + F - w$, i.e., x and y are connected by a path which does not contain w in $G + F$.

We first consider the case of $y \in S_q$. The tight set S_1 contains a leaf set $T \in \mathcal{L}_G(V)$, which is joined to a leaf set T' in a different component $H' \in \mathcal{C}_G(s) - H$ by an edge in F (by (i)). This implies that $G + F$ contains a path from x to y passing through s but not w .

We next consider the case of $y \notin S_q$. Then $q \geq 3$ holds; $y \in S_2$ is assumed without loss of generality. The tight sets S_1 and S_2 contain leaf sets T_1 and T_2 , respectively in G ; each T_i is joined to a leaf set T'_i in a component $H'_i \in \mathcal{C}_G(s) - H$ by an edge in F (by (i)). This also implies that x and y are connected by a path which does not contain w . $\square$

Based on this lemma, we consider the following algorithm for augmenting a given G .

MATE

Initially set all leaf sets to be *uncovered*, and all components $H \in \mathcal{C}_G(s)$ to be *unscanned*;

$\ell_H := |\mathcal{L}_G(H)|$ for all $H \in \mathcal{C}_G(s)$; /* The set $\{\ell_H \mid H \in \mathcal{C}_G(s)\}$ will remain balanced. */

$F_0 := \emptyset$;

while (i) some $\mathcal{H}_i$ contains at least two unscanned components and
(ii) $\{\ell_H \mid H \in \mathcal{C}_G(s)\}$ is not critically balanced **do**

 Choose a pair of unscanned subsets $H, H' \in \mathcal{H}_i$, letting H and H' be scanned;

 Choose two leaf sets $L \subseteq \mathcal{L}_G(H)$ and $L' \subseteq \mathcal{L}_G(H')$, letting L and L' be covered;

 Add to F_0 a new edge joining L and L' , letting $\ell_H := \ell_H - 1$ and

$\ell_{H'} := \ell_{H'} - 1$;

end /* while */

 /* Each $\mathcal{H}_i$ contains at most one unscanned component H ,
 or $\{\ell_H \mid H \in \mathcal{C}_G(s)\}$ is critically balanced */

For each $H \in \mathcal{C}_G(s)$, choose a vertex from each uncovered leaf set in $\mathcal{L}_G(H)$, and let X_H denote the set of these vertices;

/* $|X_H|$ is equal to the current ℓ_H */

Construct an edge cover M on $W = \cup_{H \in \mathcal{C}_G(s)} X_H$ with partition constraint $\{X_H \mid H \in \mathcal{C}_G(s), \ell_H > 0\}$;

for each $\mathcal{H}_i$ ($i = 1, 2, \dots, p$) **do**

 Let q_i denote the number of components in the induced graph

$(G + (F_0 \cup M))[\cup_{H \in \mathcal{H}_i} V(H)]$;

 Choose a set F_i of $q_i - 1$ edges which connect all components in $(G + (F_0 \cup M))[\cup_{H \in \mathcal{H}_i} V(H)]$

into a single component;

end /* for */

$F := F_0 \cup M \cup F_1 \cup F_2 \cup \dots \cup F_p$. $\square$

We easily see that MATE can be implemented to run in linear time by an appropriate data structure. A set F computed by MATE satisfies the conditions (i) and (ii) in Lemma 6, and is a solution to (G, r) .

Lemma 7. *For a set F output by MATE, $|F| \leq \lceil \alpha(G, r)/2 \rceil + \frac{1}{2}\beta(G, r, s)$ holds.*

Proof: By construction, $|F_0 \cup M| = \lceil \alpha(G, r)/2 \rceil$. It suffices to show that $|F_1 \cup F_2 \cup \dots \cup F_p| \leq \frac{1}{2}\beta(G, r, s)$. We first consider the case where the set $\{\ell_H \mid H \in \mathcal{H}\}$ obtained after the while-loop is not critically balanced. In this case, components in each $\mathcal{H}_i$ (except for at most one unscanned component) are partitioned into pairs of components H and H' which are joined by an edge in F_0 . Thus q_i in MATE is at most $\frac{1}{2}(|\mathcal{H}_i| + 1)$.

We next consider the case where the set $\{\ell_H \mid H \in \mathcal{H}\}$ obtained after the while-loop is critically balanced. In this case, in each $\mathcal{H}_i$, scanned components are partitioned into pairs joined by edges in F_0 and unscanned components are joined to the same component in $\mathcal{C}_G(s)$.

In any case, we have $|F_i| = q_i - 1 \leq \frac{1}{2}(|\mathcal{H}_i| - 1)$. This proves $|F_1 \cup F_2 \cup \dots \cup F_p| \leq \frac{1}{2}\beta(G, r, s)$. $\square$

By summing up the discussions so far, we complete the proof of Theorem 2.

We close this section by constructing an example which shows that the performance guarantee $\frac{3}{2}$ of our algorithm is tight. Let p be a positive integer. Let $G = (V, E)$ be a tree with vertex set $V = U_1 \cup U_2 \cup \{s, w_1, w_2\} \cup Z_1 \cup Z_2$, where $U_i = \{u_1^i, \dots, u_{2p+1}^i\}$ and $Z_i = \{z_1^i, \dots, z_{2p+1}^i\}$ for $i = 1, 2$, and edge set $E = \{(u, s) \mid u \in U_1 \cup U_2\} \cup \{(s, w_1), (s, w_2)\} \cup \{(w_1, z) \mid z \in Z_1\} \cup \{(w_2, z) \mid z \in Z_2\}$. Let $r(x, y)$ be 2 if $\{x, y\} \subset X$ with $X \in \{U_1, U_2, Z_1, Z_2\}$ and be 0 otherwise. Clearly s is the balanced cut-vertex in the (G, r) , from which algorithm MATE computes $F_0 = \{(u_1^1, u_2^1), (u_3^1, u_4^1), \dots, (u_{2p-1}^1, u_{2p}^1)\} \cup \{(u_1^2, u_2^2), (u_3^2, u_4^2), \dots, (u_{2p-1}^2, u_{2p}^2)\}$, $M = \{(u_{2p+1}^1, z_{2p+1}^1), (u_{2p+1}^2, z_{2p+1}^2)\} \cup \{(z_1^1, z_1^2), \dots, (z_{2p}^1, z_{2p}^2)\}$, $F_1 = \{(u_1^1, u_3^1), (u_4^1, u_5^1), \dots, (u_{2p}^1, u_{2p+1}^1)\}$ and $F_2 = \{(u_2^2, u_3^2), (u_4^2, u_5^2), \dots, (u_{2p}^2, u_{2p+1}^2)\}$. The output solution $F = F_0 \cup M \cup F_1 \cup F_2$ has size $|F| = 6p + 2$, while an optimal solution $F^{opt} = \{(u_j^i, z_j^i) \mid j = 1, \dots, 2p + 1, i = 1, 2\}$ has size $|F^{opt}| = 4p + 2$. The ratio $|F|/|F^{opt}| = (3p + 1)/(2p + 1)$ is asymptotically $\frac{3}{2}$.

4 Conclusion

In this paper, we showed that the local-vertex connectivity augmentation problem is NP-hard even if $r(u, v) \in \{0, k\}$, $u, v \in V$ for a fixed k . We then presented a $\frac{3}{2}$ -approximation algorithm for the problem with a connected graph and target values $r(u, v) \in \{0, 2\}$, $u, v \in V$. It is a future work to design approximation algorithms for more general cases of the LVCAP by extending the graph theoretic technique in this paper.

References

1. A. Frank: "Connectivity augmentation problems in network design," in: *Mathematical Programming: State of the Art* 1994 (eds: J. R. Birge and K. G. Murty), The University of Michigan, Ann Arbor, MI, (1994) 34–63. [124](#)
2. A. Frank: "Augmenting graphs to meet edge-connectivity requirements," *SIAM J. Discrete Mathematics*, 5 (1992) 25–53. [125](#)
3. M. R. Gary and D. S. Johnson: *Computers and intractability, a guide to the theory of NP-completeness*, Freeman, San Francisco 1978. [125](#), [126](#)
4. L. S. Hakimi: "On realizability of a set of integers as degrees of the vertices of a linear graph," *J. SIAM*, 10 (1962) 496–506. [130](#)
5. F. Harary: *Graph Theory*, Addison-Wesley, Reading, MA, 1969. [131](#)
6. T. Jordán: *Connectivity augmentation problem in graphs*, PhD thesis, Dept. of Computer Science, Eötvös University, Budapest, Hungary, 1994. [124](#)
7. H. Nagamochi and T. Ibaraki: "Graph connectivity and its augmentation: applications of MA orderings," *Discrete Applied Mathematics* (to appear). [125](#)
8. R. Ravi and D. P. Williamson: "An approximation algorithm for minimum-cost vertex-connectivity problems," *Algorithmica*, 18 (1995) 21–43. [124](#)
9. R. E. Tarjan: "Depth-first search and linear graph algorithms," *SIAM J. Computing*, 1 (1972) 146–160. [128](#)
10. K. Tsuchiya, T. Mashima and T. Watanabe: "Solving the 2-edge or 2-vertex-connectivity augmentation problem for specified sets of vertices," (in Japanese) *IPSJ SIG Notes*, 96-AL-54 (1996) 65–72. [124](#)

Counting Graph Isomorphisms among Chordal Graphs with Restricted Clique Number

Takayuki Nagoya

The University of Electro-Communications, Department of Computer Science
Chofugaoka 1-5-1, Chofu-shi, Tokyo 182, Japan
nagoya-t@np.cs.uec.ac.jp

Abstract. In this paper, we study the following problem: given two graphs G and H and an isomorphism ϕ between an induced subgraph of G and an induced subgraph of H , compute the number of isomorphisms between G and H that doesn't contradict to ϕ . We show that this problem can be solved in $O((k+1)!n^3)$ time when input graphs are restricted to chordal graphs with clique number at most $k+1$. To show this, we first show that the tree model of a chordal graph can be uniquely constructed except for the ordering of children of each node in $O(n^3)$ time. Then, we show that the number of isomorphisms between G and H that doesn't contradict to ϕ can be efficiently computed by use of the tree model.

1 Introduction

The graph isomorphism problem is as follows: given two graphs G and H , determine if there is a renaming of vertices of G that results in H . It is not known if the general graph isomorphism problem can be solved in polynomial time. It remains unknown even for several special classes of graphs. For example, the class of chordal graphs is one of such classes. In fact, testing graph isomorphism for this class is polynomial time equivalent to the general case.

On the other hand, it was shown that the isomorphism problem was able to be solved in polynomial time when restricted to special graph classes, e.g., to graphs of bounded degrees[11], planar graphs[9], trees, interval graphs[4], rooted directed path graphs[3], cographs[5], permutation graphs[7], k -trees (for fixed k)[2],[13], chordal graphs with restricted clique number[10], and graphs of bounded eigenvalue multiplicity[1]. In particular, Boadlaender[2] showed that if input graphs are restricted to partial k -trees, the problem can be solved in $O(k!n^{k+4.5})$ time. Klawe[10] showed that a tree model of a chordal graph can be uniquely constructed except for the ordering of children of each node in polynomial time and if such tree model is given then the problem can be solved in $O(((k+1)!)^2n^3)$ time.

The graph isomorphism counting problem is as follows: given two graphs G and H , count the number of isomorphisms between G and H . The problem can be solved in polynomial time when restricted to labeled trees, planar graphs, interval graphs[6], and partial k -trees[13].

In this paper, we study the problem: given two graphs G and H and an isomorphism ϕ between an induced subgraph of G and an induced subgraph of H , compute the number of isomorphisms between G and H that doesn't contradict to ϕ . Note that this problem is a generalization of the graph isomorphism counting problem. For partial k -trees, it was shown in [13] that this problem can be computed in $O((k+1)!n^{k+4})$ time.

In this paper, we consider the case where input graphs are restricted to chordal graphs with clique number at most $k+1$. Since the tree width of all such graphs is at most k , we see, from the result of [13], that this case can be also solved in $O((k+1)!n^{k+4})$ time. Furthermore, by use of the fact that a chordal graph with clique number at most $k+1$ has a width k tree model and the number of maximal cliques of a chordal graph is at most $O(n)$, we can observe that the algorithm designed in [13] can be modified so as to run in time $O((k+1)!n^5)$.

In this paper, we show that this case can be solved in time at most $O((k+1)!n^3)$. First, we show that a tree model of given chordal graph can be uniquely constructed in $O(n^3)$ time except for the ordering of children of each node. Then we show that this case can be solved efficiently by use of the tree model. Therefore, the graph isomorphism problem and the graph isomorphism counting problem can be solved in time at most $O((k+1)!n^3)$ when input graphs are restricted to chordal graphs with clique number at most $k+1$.

In section 2, we give some notations and a result for chordal graphs. In section 3, we give an algorithm that, for a given chordal graph G , constructs a tree model of G that is unique except for ordering of children of each node. In section 4, we show that, for two chordal graph G, H with clique number at most $k+1$ and an isomorphism ϕ from an induced subgraph of G to an induced subgraph of H , the number of isomorphisms from G to H that doesn't contradict to ϕ can be computed in time at most $O((k+1)!n^3)$.

2 Preliminary

The graphs in this paper are simple, undirected and connected. For a graph G , we denote $V(G)$ the set of vertices of G and denote $E(G)$ the set of edges of G . Let U be a subset of $V(G)$. The subgraph of G induced by U is denoted by $G[U]$ where $V(G[U]) = U, E(G[U]) = \{(x, y) \in E(G) \mid x, y \in U\}$. For a vertex $v \in V(G)$, we define $N_U(v) = \{w \in U \mid (v, w) \in E(G)\}$ and $N_U[v] = N_U(v) \cup \{v\}$. For a subset $W \subseteq V(G)$, we define $N_U(W) = \bigcup_{v \in W} N_U(v)$. For any subset $S \subseteq V(G)$, we say S separates $C \subseteq V(G)$ and $D \subseteq V(G)$ in G if there is no path between C and D in $G[V \setminus S]$. For a rooted tree T and a node i of T , we define the level of i , $\text{level}(i)$, to be the length of the path from the root of T to i .

Definition 2.1: We say a vertex $v \in V(G)$ is *simplicial* if $N(v)$ is a clique in G . We say a linear ordering $v_1, \dots, v_n$ of vertices in $V(G)$ is a *perfect elimination scheme* of G if for any $i (1 \leq i \leq n)$ v_i is a simplicial vertex of $G[\{v_j \mid j = i, i+1, \dots, n\}]$. $\square$

Definition 2.2: A graph is a *chordal graph* if it doesn't contain an induced cycle of length more than or equal to four. $\square$

For a graph G , if G is the intersection graph of a family of subtrees of a tree, then such family called a *tree model* of G . In this paper, a tree model of a graph is defined as follow.

Definition 2.3: A *tree model* of G is a pair (T, X) where T is a tree and $X = \{X_i | i \in V(T)\}$ is a collection of subsets of $V(G)$ satisfying the following conditions.

1. $\bigcap_{i \in V(T)} X_i = V(G)$
2. $v, w \in V(G) [(v, w) \in E(G) \Leftrightarrow \bigcap_{i \in V(T)[v, w]} X_i]$
3. $i, j, k \in V(T) [j \text{ is on the path from } i \text{ to } k \text{ in } T \Rightarrow X_i \cap X_k \subseteq X_j]$

The *width* of a tree model (T, X) is $\max\{|X_i| - 1 | i \in V(T)\}$. □

For a tree model (T, X) of G and a vertex $v \in V(G)$, we define $T(v) = T[\{i \in V(T) | v \in X_i\}]$. Then we easily see from the condition 3 above that each $T(v)$ is a connected subtree of T . Furthermore, we can easily see from the condition 2 above that for two vertices $v, w \in V(G)$, v is adjacent to w if and only if $T(v)$ intersects to $T(w)$, that is, $V(T(v)) \cap V(T(w)) \neq \emptyset$. We also see from the condition 2, that for every $i \in V(T)$, X_i is a clique of G . In other words, we can restate the condition 2 as follows:

2. (a) $v, w \in V(G) [(v, w) \in E(G) \Leftrightarrow \bigcap_{i \in V(T)[v, w]} X_i]$ and
(b) for every $i \in V(T)$, X_i is a clique in G .

Moreover, we see from this fact that, if G has a tree model and its clique number is at most $k + 1$, then any tree model of G is of width at most k . We will use these facts in the later argument.

For a graph G that have a tree model, there exists many tree models of G . Figure 1 is an example of a graph and its tree models. In these tree models, each $T(v)$ is the subtree of the tree induced by those nodes that contain v .

Definition 2.4: A tree model (T, X) is called a *rooted tree model* if T is a rooted tree. □

In the remainder of this paper, whenever we say tree model, we suppose it to be rooted.

The following theorem is well known[8]. In section 3, We will use the theorem to construct a tree model of a given chordal graph.

Theorem 2.5: For every graph G , the following statements are equivalent.

- (i) G is a chordal graph.
- (ii) G has a perfect elimination scheme.
- (iii) G has a tree model. □

By this theorem, chordal graphs can be described as the intersection graphs of the subtrees of a tree. Note that the graph in Fig.1 is a chordal since it has a tree model.

Definition 2.6: Let G and H be graphs. Let ϕ be a bijection from a subset of $V(G)$ to a subset of $V(H)$. Then we denote the domain of ϕ by $Dom(\phi)$

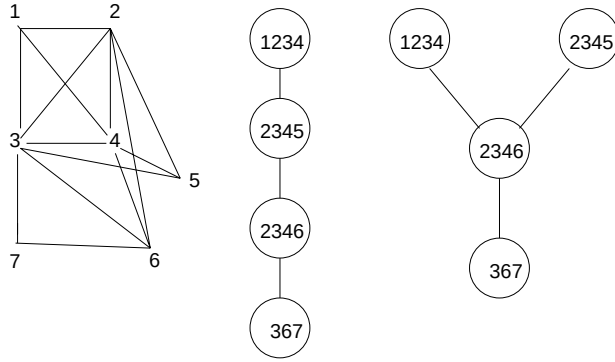


Fig. 1. A graph and its tree models

and denote the image of ϕ by $Im(\phi)$. For each $X \subseteq Dom(\phi)$, we denote $\phi|_X$ the restriction of ϕ whose domain is X . Let ψ be an another bijection from a subset of $V(G)$ to a subset of $V(H)$. Then we say ψ *doesn't contradict* to ϕ if either $Dom(\phi) \cap Dom(\psi) = \emptyset$ and $Im(\phi) \cap Im(\psi) = \emptyset$, or, for all $x \in Dom(\phi) \cap Dom(\psi)$ [$\phi(x) = \psi(x)$] and for all $y \in Im(\phi) \cap Im(\psi)$ [$\phi^{-1}(y) = \psi^{-1}(y)$]. $\square$

3 Constructing a Tree Model

In this section, we give a new algorithm that, for a given chordal graph G , constructs a tree model of G that is unique except for ordering of children of each node.

Definition 3.1: Let G be a chordal graph. Let $S = S_1, S_2, \dots, S_d$ be an ordered partition of $V(G)$. That is, it is a partition of $V(G)$ and the S_i 's are ordered with their indices. Then, the ordered partition is called *simplicial* if for every $i = 1, \dots, d$, S_i is the set of all simplicial vertices of the induced subgraph $G[S_1 \cup \dots \cup S_d]$. $\square$

It is known that a graph has the simplicial partition if and only if it is a chordal graph[3].

For a connected chordal graph G and a simplicial vertex v of G , we can easily see that $G[V \setminus \{v\}]$ is connected. Furthermore, for a connected chordal graph G , it is easy to see that all vertices of G are simplicial if and only if G is a clique. Therefore, for the simplicial partition $S_1, \dots, S_d$ of a connected chordal graph G , S_d is a clique of G .

By definition 3.1, the simplicial partition is unique for every chordal graph. It is easy to see from this fact that, for a chordal graph G and the simplicial partition S of G , every automorphism on G must map S_i to S_i for all i ($1 \leq i \leq d$).

Let $\pi = v_1, \dots, v_n$ be a perfect elimination scheme of a chordal graph G . For an arbitrary $i = 1, \dots, n$, we define $t(i) = \min\{j | v_i \text{ is adjacent to } v_j\}$. Then we can easily see that a vertex v_i is a simplicial vertex of G if either $i \leq t(i)$ or $N[v_i] = N[v_{t(i)}] \cap \{v_{t(i)} \dots v_n\}$. Furthermore, for a chordal graph G , a perfect elimination scheme of G can be computed at most $O(|V(G)| + |E(G)|)$ time [14]. Based on this criterion, we may compute the simplicial partition of a chordal graph G in time at most $O(|V(G)|^3)$.

Given a chordal graph G and the simplicial partition $S = S_1, \dots, S_d$ of G , the following algorithm constructs a tree model (T, X) of G with $X = \{X_i | i \in V(T)\}$ that is unique except for the ordering of children of each node of T . In the following algorithm, r denotes the root of T , and for every $v \in V(G)$, $I_S(v) = i$ denotes the index of S_i to which v belongs. In the algorithm, we use a queue denoted by Q . $\text{Enqueue}(Q, s)$ denotes an operation inserting an element s to Q and $\text{Dequeue}(Q)$ denotes an operation deleting an element from Q .

We illustrate the outline on how TREEMODEL works. First, this algorithm sets $X_r = S_d$ i.e., it puts S_d as a root of tree model. Let $C(r) = \{C_1, C_2 \dots C_m\}$ be a set of all connected components of $G[V \setminus X_r]$. For each C_p , this algorithm constructs a child r_p of r and sets $X_{r_p} = U \cup N_{X_r}(U)$, where U is a subset of C_p . Furthermore, for each connected component of $G[C_p \setminus U]$, the algorithm constructs children of r_p in the same manner as above.

TREEMODEL(G, S)

- (1) Set $V(T) = \{r\}$ and $X_r = S_d$.
- (2) Set $C(r)$ to the set of connected components of $G[V \setminus X_r]$.
- (3) $\text{Enqueue}(Q, r)$
- (4) while $Q \neq \emptyset$ do
 - (4-1) $l \leftarrow \text{Dequeue}(Q)$
 - (4-2) for each $C \in C(l)$ do
 - (4-2-1) Add a new node s to $V(T)$ as a child of l .
 - (4-2-2) $g \leftarrow \max\{I_S(v) | v \in C\}$.
 - (4-2-3) $U \leftarrow \{v \in C | I_S(v) = g\}$
 - (4-2-4) $X_s \leftarrow U \cup N_{X_l}(U)$
 - (4-2-5) Set $C(s)$ to the set of connected components of $G[C \setminus U]$.
 - (4-2-6) $\text{Enqueue}(Q, s)$

In this algorithm, for each node l and each $C \in C(l)$, a new node s is added as a child of l . Then we say s corresponds to C . Note that the number of children of l is equal to the number of connected components in $C(l)$.

We must show that (T, X) constructed by TREEMODEL is a tree model of input graph G . It is obvious that (T, X) satisfies the condition 1 in the definition of tree model. Furthermore, we easily see from the way of constructing (T, X) that, for every $v \in V(G)$, $T(v)$ is a connected subtree of T where $T(v) = T[\{i \in V(T) | v \in X_i\}]$. By this fact, it is easily to see that (T, X) satisfies the condition 3 in the definition of tree model. We leave its proof to the reader. We below show that (T, X) satisfies the condition 2 in the definition of tree model. That is, we show:

2. (a) $v, w \in V(G) \setminus E(G)$
 $i \in V(T) \setminus [v, w \setminus X_i]$ and
 (b) for every $s \in V(T)$, X_s is a clique in G .

Next lemma show that the condition (b) holds.

Lemma 3.2: For every $s \in V(T)$, X_s is a clique. $\square$

Proof: Let s be a node of T . We use an induction on the level h of s in T . Suppose $h = 0$. In this case, s is the root of T and $X_s = S_d$. By the definition of simplicial partition, X_s is a clique.

Suppose $h \geq 1$. Suppose also that, for the parent l of s in T , X_l is a clique in G . We below show that X_s is a clique in G . By the way of constructing (T, X) , there exists a connected component C in $C(l)$ to which s corresponds. Let $g = \max\{I_S(v) \mid v \in C\}$ and let $U = \{v \in C \mid I_S(v) = g\}$. Then, by the way of constructing (T, X) , $X_s = U \cup N_{X_l}(U)$.

Now, we have the following two claims.

CLAIM 1: For every two vertices $u_1, u_2 \in U$, u_1 is adjacent to u_2 . $\square$

CLAIM 2: For every two vertices $u_1, u_2 \in U$, $N_{X_l}(u_1) = N_{X_l}(u_2)$. $\square$

By the CLAIM 1, we have that U is a clique. As X_l is a clique by the induction hypothesis, $N_{X_l}(U)$ is a clique also. Furthermore, by the CLAIM 2, every vertex belong to U is adjacent to every vertex belong to $N_{X_l}(U)$. Therefore, we have that $X_s = U \cup N_{X_l}(U)$ is a clique. This completes the proof. $\square$

Next lemma show that the condition (a) holds.

Lemma 3.3: For every two vertices $x, y \in V(G)$, if x is adjacent to y then there exists a node $i \in V(T)$ such that $x, y \in X_i$. $\square$

Proof: Let x and y be vertices of G such that x is adjacent to y . Without loss of generality, we assume that $I_S(y) \leq I_S(x)$. Let $T(x) = T[\{i \in V(T) \mid x \in X_i\}]$ and let $T(y) = T[\{i \in V(T) \mid y \in X_i\}]$. We easy see from the way of constructing (T, X) that $T(x)$ and $T(y)$ are connected subtrees of T . Let r_x and r_y be the root of $T(x)$ and $T(y)$ respectively. We easy see from $I_S(x) \geq I_S(y)$ and x is adjacent to y that r_y is a descendant of r_x . Let $p(r_x, r_y) = (r_x = l_1, l_2, \dots, l_q = r_y)$ be a path from r_x to r_y in T . we below show that, for every $i (1 \leq i \leq q)$, $x, y \in X_{l_i}$. Note that if this statement holds then $x, y \in X_{l_q}$.

We use an induction on i . For $i = 1$, we have that $x \in X_{l_1}$. Suppose $x \in X_{l_{i-1}}$. We below show that $x \in X_{l_i}$. By the way of constructing (T, X) , there exists a connected component C in $C(l_{i-1})$ such that l_i corresponds to C . Then, we have that y belongs to C . This is because, if y doesn't belong to C then, in the subtree of T whose root is l_i , there is no node to which y belongs. This contradict to the fact that $y \in X_{r_y}$ and r_y is in the subtree of T whose root is l_i . Let $g = \max\{I_S(v) \mid v \in C\}$ and let $U = \{v \in C \mid I_S(v) = g\}$. Then, by the way of constructing (T, X) , $X_{l_i} = U \cup N_{X_{l_{i-1}}}(U)$.

First, we consider the case where $y \in U$. In this case, as $x \in N_{X_{l_{i-1}}}(y)$ $N_{X_{l_{i-1}}}(U)$, we have that $x \in X_{l_i}$.

Next, we consider the case where $y \in U$. Let u be a vertex belong to U . Then as $G[C]$ is connected, there exists a path $p(u, y) = (u = d_0, d_1, \dots, d_h = y)$ in $G[C]$. Furthermore, as y is adjacent to x , $p(u, x) = (u = d_0, d_1, \dots, d_h = y, d_{h+1} = x)$ is a path in G . We easy see that, for every $v \in X_{l_{i-1}}$, $g < I_S(v)$.

Therefore, since $x \in X_{I_{i-1}}$, we have that $I_S(u) = g < I_S(x)$. Moreover, for every j ($1 \leq j \leq h$), $I_S(d_j) \leq g$. This is because, for every j ($1 \leq j \leq h$), $d_j \in C$, and $g = \max\{I_S(v) \mid v \in C\}$. We below show that x is adjacent to u . We can choose f ($1 \leq f \leq h$) such that $I_S(d_f) = \min\{I_S(d_0), \dots, I_S(d_{h+1})\}$. We consider the graph $J = G[a \in V(G) \mid I_S(a) \geq I_S(d_f)]$. Then all vertices in the path $p(u, x)$ are in J . Noting that $J = G[S_{I_S(d_f)} \cup S_{I_S(d_f)-1} \cup \dots \cup S_d]$, we see from the definition of simplicial partition that all vertices in $S_{I_S(d_f)}$ are simplicial in J . Since the vertex d_f belong to $S_{I_S(d_f)}$, we have that d_f is a simplicial in J . This indicates that d_{f-1} is adjacent to d_{f+1} . Therefore, we can get a new path $\hat{p}(u, x) = (u = d_0, d_1, \dots, d_{f-1}, d_{f+1}, \dots, d_{h+1} = x)$. Repeating this argument for the path $\hat{p}(u, x)$, we have that x is adjacent to u . Now, we have that $u \in U$ and $x \in X_{I_{i-1}}$ and x is adjacent to u . This indicates that $x \in N_{X_{I_{i-1}}}(u) \cap N_{X_{I_{i-1}}}(U)$. Therefore, as $X_{I_i} = U \cap N_{X_{I_{i-1}}}(U)$, we have that $x \in X_{I_i}$.

By the above argument, we have that, for every i ($1 \leq i \leq q$), $x \in X_{I_i}$. Therefore, $x, y \in X_{I_q} = X_{r_y}$. This completes the proof. $\square$

By the Lemma 3.2 and the Lemma 3.3, we have that (T, X) constructed by TREEMODEL satisfies the condition 2 of the definition of tree model. Therefore, we have next corollary.

Corollary 3.4: (T, X) constructed by TREEMODEL is a tree model of input graph. $\square$

By the CLAIM 2, in line (4-2-4), $X_S \leftarrow U \cap N_{X_i}(U)$ can be replaced by $X_S \leftarrow U \cap N_{X_i}(u)$ for arbitrary $u \in U$. Therefore, we have next lemma.

Lemma 3.5: The time complexity of TREEMODEL is at most $O(|V(G)|^3)$. $\square$

Let (T, X) be a tree model of G constructed by TREEMODEL. By the way of constructing (T, X) , we have the next proposition.

Proposition 3.6: For every automorphism μ on G , there exists an one to one correspondence τ on $V(T)$ such that, for every $i \in V(T)$, the level of i in T is equal to the level of $\tau(i)$ in T and $\mu(X_i) = X_{\tau(i)}$. $\square$

Let G and H be chordal graphs, and let $(T(G), X(G))$ and $(T(H), X(H))$ be tree models of G and H constructed by TREEMODEL. Then, this proposition indicates that, for every isomorphism μ between G and H , there exists a bijection τ from $V(T(G))$ to $V(T(H))$ such that, for every $i \in V(T(G))$, the level of i in $T(G)$ is equal to the level of $\tau(i)$ in $T(H)$ and $\mu(X_i(G)) = X_{\tau(i)}(H)$. We implicitly use this fact in the next section.

4 Algorithm

In this section, we let G and H be chordal graphs with clique number at most $k + 1$, and let ϕ be an isomorphism between an induced subgraph of G and an induced subgraph of H . Then we show that the number of isomorphisms between G and H that doesn't contradict to ϕ can be computed in $O((k + 1)!n^3)$ time.

Our algorithm first computes simplicial partition $S(G) = S_1(G), S_2(G), \dots, S_d(G)$ and $S(H) = S_1(H), S_2(H), \dots, S_d(H)$ of G and H respectively. And then, the algorithm constructs tree models $(T(G), X(G))$ and $(T(H), X(H))$ that is

unique except for the ordering of children of each node. By using these tree models, the algorithm computes the number of isomorphisms between G and H that doesn't contradict to ϕ . Note that the width of each tree model is at most k .

If either $|V(G)| = |V(H)|$ or $d = \hat{d}$, then there is no isomorphism between G and H . Therefore we assume $|V(G)| = |V(H)| = n$ and $d = \hat{d}$ in the following argument.

Definition 4.1: For two bijections ϕ and ψ that doesn't contradict to each other, we define the bijection $\phi \sim \psi$ from $Dom(\phi) \cap Dom(\psi)$ to $Im(\phi) \cap Im(\psi)$ as follows.

$$\phi \sim \psi \stackrel{def}{=} \begin{cases} \phi(x) & \text{if } x \in Dom(\phi) \\ \psi(x) & \text{if } x \in Dom(\psi) \end{cases}$$

□

Definition 4.2: Let X be a subset of $V(G)$, and let Y be a subset of $V(H)$. Then for every bijection ψ from X to Y , we say ψ is ϕ -isomorphism from $G[X]$ to $H[Y]$ if ψ doesn't contradict to ϕ and ψ is an isomorphism from $G[X]$ to $H[Y]$. We denote $G[X] =^\phi H[Y]$ if and only if there exists a ϕ -isomorphism from $G[X]$ to $H[Y]$. We define $Iso_\phi(G[X], H[Y])$ to be the set of all ϕ -isomorphisms from $G[X]$ to $H[Y]$.

□

Lemma 4.3: Let X and Y be as in Definition 4.2. Moreover, let ψ be a ϕ -isomorphism from $G[X]$ to $H[Y]$. Then the following two statements holds.

(1) For two connected components $C_1, C_2 (C_1 \cap C_2 = \emptyset)$ of $G[V \setminus X]$ and for two connected components $D_1, D_2 (D_1 \cap D_2 = \emptyset)$ of $H[V \setminus Y]$, if $G[X \cup C_1] =^\phi \psi H[Y \cup D_1]$ and $H[X \cup D_1] =^\phi \psi G[Y \cup C_2]$ and $G[X \cup C_2] =^\phi \psi H[Y \cup D_2]$, then $G[X \cup C_1] =^\phi \psi H[Y \cup D_2]$.

(2) For a connected component C of $G[V \setminus X]$ and for two connected components D_1, D_2 of $H[V \setminus Y]$, if $G[X \cup C] =^\phi \psi H[Y \cup D_1]$ and $G[X \cup C] =^\phi \psi H[Y \cup D_2]$, then the number of $\phi \sim \psi$ -isomorphisms between $G[X \cup C]$ and $H[Y \cup D_1]$ is equal to the number of $\phi \sim \psi$ -isomorphisms between $G[X \cup C]$ and $H[Y \cup D_2]$.

□

Definition 4.4: We define $N(G, H, \phi)$ to be the number of ϕ -isomorphisms between G and H .

□

We below consider how to compute $N(G, H, \phi)$.

Definition 4.5: Let s be a node in $T(G)$ (or $T(H)$). We denote $T_s(G)$ (or $T_s(H)$) the maximal subtree of $T(G)$ (or $T(H)$) whose root is s , and we denote $V_s(G) = \bigcup_{f \in V(T_s(G))} X_f$ (or $V_s(H) = \bigcup_{f \in V(T_s(H))} X_f$). If G (or H) is obvious from context, we write T_s and V_s .

□

Definition 4.6: For each $s \in V(T(G))$, each $t \in V(T(H))$, and each $\psi \in Iso_\phi(G[X_s], H[X_t])$, we denote $N(G, H, \phi; s, t, \psi)$ the number of ϕ -isomorphisms between $G[V_s]$ and $H[V_t]$ that doesn't contradict to ψ .

□

As $(T(G), X(G))$ and $(T(H), X(H))$ are uniquely determined except for the ordering of children of each node, every ϕ -isomorphism between G and H must map $X_r(G)$ to $X_{\hat{r}}(H)$. Therefore, the following proposition is satisfied.

Proposition 4.7:

$$N(G, H, \phi) = \sum_{\psi \in Iso_\phi(G[X_r], H[X_{\hat{r}}])} N(G, H, \phi; r, \hat{r}, \psi)$$

□

□

This proposition indicates that $N(G, H, \phi)$ can be computed by using $N(G, H, \phi; r, \hat{r}, \psi)$.

We below show how to compute $N(G, H, \phi; r, \hat{r}, \psi)$. Let s and t be any nodes of $V(T(G))$ and $V(T(H))$ whose level are equal to each other. And, let ψ be an ϕ -isomorphism from $G[X_s]$ to $H[X_t]$. We show that $N(G, H, \phi; s, t, \psi)$ can be computed recursively. If the number of children of s is different from the number of children of t , then we have $N(G, H, \phi; s, t, \psi) = 0$. Therefore, we assume that these are equal. In the following argument, we let $s_1, \dots, s_m$ be all children of s in $T(G)$ and $t_1, \dots, t_m$ be all children of t in $T(H)$. By the way of constructing $(T(G), X(G))$, $G[V_s \setminus X_s]$ has m connected components $C_1, \dots, C_m$. We let, for all p , s_p corresponds to C_p . Similarly, $H[V_t \setminus X_t]$ has m connected components $D_1, \dots, D_m$. We let, for all p , t_p correspond to D_p .

Definition 4.8: We define $N(p, q)$ to be the number of ϕ -isomorphisms between $G[X_{s_p} \setminus C_p]$ and $H[X_{t_q} \setminus D_q]$ that doesn't contradict to ψ . $\square$

Noting that $\psi : X_s \rightarrow X_t$ is fixed, we have the next proposition.

Proposition 4.9: Let μ be a bijection from V_s to V_t . Then the next two statements are equivalent.

1. μ is a ϕ -isomorphism between $G[V_s]$ and $H[V_t]$ that doesn't contradict to ψ .
2. There exists an one to one correspondence τ on $\{1, \dots, m\}$ such that, for each p , $\mu|_{X_{s_p} \setminus C_p}$ is a ϕ -isomorphism between $G[X_{s_p} \setminus C_p]$ and $H[X_{t_{\tau(p)}} \setminus D_{\tau(p)}]$ that doesn't contradict to ψ . $\square$

By this proposition, we have next corollary.

Corollary 4.10: $N(G, H, \phi; s, t, \psi)$ can be computed by:

$$\square \quad \square \quad N(p, \tau(p)).$$

$$\tau \quad 1 \leq p \leq m$$

where the summation is taken over all one to one correspondence τ on $\{1, 2, \dots, m\}$ $\square$

We below show that if, for every p, q , $N(p, q)$ is known then $N(G, H, \phi; s, t, \psi)$ can be efficiently computed by using this corollary.

Definition 4.11: We define $B(G, H, \phi; s, t, \psi)$ to be an edge-weighted bipartite graph (V_B^1, V_B^2, E_B) with an weight function $e_B : E_B \rightarrow \mathbb{Z}^+$. Here:

- $V_B^1 = \{1, \dots, m\}$
- $V_B^2 = \{1, \dots, m\}$
- $(p, q) \in E_B \Leftrightarrow N(p, q) > 0$
- $e_B(p, q) = N(p, q)$ $\square$

If, for every p and q , $N(p, q)$ is known, the bipartite graph $B(G, H, \phi; s, t, \psi)$ can be constructed in time at most $O(m^2)$.

The next corollary follows immediately from corollary 4.10

Corollary 4.12: Let M be the set of all perfect matchings of the bipartite graph $B(G, H, \phi; s, t, \psi)$. Then :

$$N(G, H, \phi; s, t, \psi) = \square \quad \square \quad e_B(p, q) \quad \square$$

$$\sum_{M \in M} \sum_{(p, q) \in M} \sum_M$$

This corollary indicates that in order to compute $N(G, H, \phi; s, t, \psi)$, it is enough to compute the permanent of $B(G, H, \phi; s, t, \psi)$. To compute the permanent, we decompose $B(G, H, \phi; s, t, \psi)$ to connected components. Let $B_1, \dots, B_f$ be all such connected components, and let B_h^1, B_h^2 denote the vertex sets of each B_h . Then the permanent of $B(G, H, \phi; s, t, \psi)$ can be computed as the multiplication of the permanents of all connected components. By Lemma 4.3 (1), all connected components of $B(G, H, \phi; s, t, \psi)$ are complete bipartite graphs. Moreover, by Lemma 4.3 (2), for every connected component B_h , all the weights of edges in B_h are equal. Let a_{B_h} be such weight. The perfect matching of a connected component B_h exists only if $|B_h^1| = |B_h^2|$. And if $|B_h^1| = |B_h^2|$, all one to one correspondence B_h^1 to B_h^2 are perfect matchings. Therefore the permanent of B_h can be computed as:

$$a_{B_h}^{|V_{B_h^1}|} \times |V_{B_h^1}|!$$

Therefore $N(G, H, \phi; s, t, \psi)$ can be computed by the following corollary.

Corollary 4.13:

$$N(G, H, \phi; s, t, \psi) = \begin{cases} \prod_h a_{B_h}^{|V_{B_h^1}|} \times |V_{B_h^1}|! & \text{if } B_h[|V_{B_h^1}| = |V_{B_h^2}|] \\ 0 & \text{otherwise} \end{cases}$$

□

If, for every p and q , we know $N(p, q)$, the bipartite graph $B(G, H, \phi; s, t, \psi)$ can be constructed and we can compute $N(G, H, \phi; s, t, \psi)$ by use of this corollary.

The next lemma indicates $N(p, q)$ can be computed using the information of S_p and t_q . Note that S_p corresponds to C_p and t_q corresponds to D_q .

Lemma 4.14:

$$N(p, q) = \sum_{\xi} N(G, H, \phi; S_p, t_q, \xi)$$

where the summation is taken over all ξ in $Isom_\phi(G[X_{S_p}], H[X_{t_q}])$ that doesn't contradict to ψ . □

Proof: The vertex set $X_s \setminus C_p$ can be partitioned into three vertex sets $X_s \setminus X_{S_p}$, $X_s \cap X_{S_p}$, and C_p . Furthermore, by the definition of tree model, the vertex set $X_s \cap X_{S_p}$ separates $X_s \setminus X_{S_p}$ and C_p in $G[X_s \setminus C_p]$. Similarly, the vertex set $X_t \setminus D_q$ can be partitioned into three vertex sets $X_t \setminus X_{t_q}$, $X_t \cap X_{t_q}$ and D_q , and the vertex set $X_t \cap X_{t_q}$ separates $X_t \setminus X_{t_q}$ and D_q in $H[X_t \setminus D_q]$. Furthermore, $\psi : X_s \rightarrow X_t$ is fixed. Therefore, $N(p, q)$ is equal to the number of ϕ -isomorphisms between $G[(X_s \cap X_{S_p}) \setminus C_p] = G[V_{S_p}]$ and $H[(X_t \cap X_{t_q}) \setminus D_q] = H[V_{t_q}]$ that doesn't contradict to ψ .

As X_{S_p} and X_{t_q} is uniquely determined, every ϕ -isomorphism between $G[V_{S_p}]$ and $H[V_{t_q}]$ that doesn't contradict to ψ map X_{S_p} to X_{t_q} . Thus, the number of ϕ -isomorphisms between $G[V_{S_p}]$ and $H[V_{t_q}]$ that doesn't contradict to ψ is equal to:

$$\sum_{\xi} N(G, H, \phi; S_p, t_q, \xi)$$

where the summation is taken over all ξ in $Isom_\phi(G[X_{S_p}], H[X_{t_q}])$ that doesn't contradict to ψ . This completes the proof. □

As the size of X_{s_p} and X_{t_q} is at most $k + 1$, whether ξ contradicts to ψ or not is solved in time at most $O(k + 1)$. Furthermore, the number of elements of $ISO_\phi(G[X_{s_p}], H[X_{t_q}])$ is at most $(k + 1)!$. Therefore, if, for every $\xi \in ISO_\phi(G[X_{s_p}], H[X_{t_q}])$, $N(G, H, \phi; s_p, t_q, \xi)$ is known, then $N(p, q)$ can be computed in time at most $O((k + 1)(k + 1)!)$.

Summarizing the results so far, we have the following theorem that gives us a recursive formula on $N(G, H, \phi; s, t, \psi)$.

Theorem 4.15: Let s and t be any nodes of $V(T(G))$ and $V(T(H))$ whose level are equal to each other. And, let ψ be a ϕ -isomorphism from $G[X_s]$ to $H[X_t]$. Then $N(G, H, \phi; s, t, \psi)$ can be computed by:

$$\sum_{\tau=1}^m \sum_{1 \leq p \leq m} \sum_{\xi} N(G, H, \phi; s_p, t_{\tau(p)}, \xi).$$

where the outer summation is taken over all one to one correspondence τ on $\{1, 2, \dots, m\}$ and the inner summation is taken over all ξ in $ISO_\phi(G[X_{s_p}], H[X_{t_{\tau(p)}}])$ that doesn't contradict to ψ . $\square$

Given two chordal graphs G, H and the tree models $(T(G), X(G)), (T(H), X(H))$ of G and H , the next algorithm computes $N(G, H, \phi)$. In the algorithm, L denotes the height of $T(G)$ and $T(H)$.

COUNTISO($G, H, (T(G), X(G)), (T(H), X(H))$)

(01) for each $l = L, L - 1, \dots, 0$ do

(02) for each $s \in V(T(G)), \text{level}(s) = l$ do

(03) for each $t \in V(T(H)), \text{level}(t) = l$ do

(04) for each $\psi \in ISO_\phi(G[X_s], H[X_t])$ do

(05) Construct $B(G, H, \phi; s, t, \psi)$.

(06) Decompose $B(G, H, \phi; s, t, \psi)$ to connected components $B_1, \dots, B_f$.

(07) for each B_h do

(08) if $|V_{B_h}^1| = |V_{B_h}^2|$ then

(09) $N_h \leftarrow a_{|V_{B_h}^1|} \times |V_{B_h}^1|!$

(10) else $N_h \leftarrow 0$ $\square$

(11) $N(G, H, \phi; s, t, \psi) \leftarrow \sum_h N_h$

(12) $N(G, H, \phi) \leftarrow \sum_{\psi} N(G, H, \phi; s, t, \psi)$

(13) output $N(G, H, \phi)$

Lemma 4.16: The time complexity of COUNTISO is at most $O((k + 1)(k + 1)!n^3)$. $\square$

The next theorem is the main result in this paper.

Theorem 4.17: Let G and H be chordal graphs with clique number at most $k + 1$, and let ϕ be an isomorphism from an induced subgraph of G to an induced subgraph of H . Then the number of ϕ -isomorphisms can be computed in time at most $O((k + 1)(k + 1)!n^3)$. $\square$

The following two corollary follows immediately from this theorem.

Corollary 4.18: Given two chordal graphs G, H with clique number at most $k + 1$, the number of isomorphisms between G and H can be computed in time at most $O((k + 1)(k + 1)!n^3)$. $\square$

Corollary 4.19: Given two chordal graphs G, H with clique number at most $k + 1$, testing isomorphism between G and H can be done in time at most $O((k + 1)(k + 1)!n^3)$. $\square$

Acknowledgement

The author is grateful to Professor Shigeki Iwata, my major advisor, for his encouragement. The author is also grateful to Professor Seinosuke Toda of Nihon University, my minor advisor, for his helpful discussion.

References

1. L. Babai, D. Y. Grigor'ev, and D. M. Mount, "Isomorphism of graphs with bounded eigenvalue multiplicity", Proc.14th ACM STOC, pp.310-324, 1982. 136
2. H. L. Bodlaender, "Polynomial Algorithms for Graph Isomorphism and Chromatic Index on Partial k -Trees", Journal of Algorithms, vol.11, no., pp.631-643, 1990. 136
3. L. Babel, I. N. Ponomarenko and G. Tinhofer, "The Isomorphism Problem For Directed Path Graphs and For Rooted Directed Path Graphs", Journal of Algorithms, vol.21, no., pp.542-564, 1996. 136, 139
4. K. S. Booth and G. S. Lueker, "A linear-time algorithm for deciding interval graph isomorphism", J. Assoc. Comput. Math. vol.26, pp.183-195, 1979 136
5. D. G. Corneil, H. Lerchs, and L. Stewart, "Complement reducible graphs", Discrete Appl.Math. vol.3, pp.163-174, 1981. 136
6. G. J. Colbourn, K. S. Booth, "Linear time automorphism algorithms for trees, interval graphs and planar graphs", SIAM J. on Comput., pp.203-225, 1981. 136
7. G. J. Colbourn, "On testing isomorphism of permutation graphs", Networks, vol.11, pp.13-21, 1981. 136
8. M. C. Golumbic, "Algorithmic Graph Theory and Perfect Graphs", Academic Press, 1980. 138
9. J. E. Hopcroft and J. K. Wong, "Linear-time algorithm for isomorphism of planar graphs", Proc.6th ACM Symp. Theory of Computing, pp.172-184, 1974. 136
10. M. M. Klawe, D. G. Corneil and A. Proskurwski, "Isomorphism testing in hookup classes", SIAM j. Alg. Disc. Meth., pp.260-274, 1982. 136
11. E. M. Luks, "Isomorphism of graphs of bounded valence can be tested in polynomial time", J. Comput. System. Sci. vol.25, pp.42-65, 1982. 136
12. R. Mathon, "A Note on The Graph Isomorphism Counting Problem," Inf. Process. lett., vol.8, no., pp.131-132, 1979.
13. T. Nagoya, S. Tani, and S. Toda, "A Polynomial-Time Algorithm for Counting Graph Isomorphisms among Partial k -trees," COMP98-24(1998-07). 136, 137
14. D. J. Rose, and R. E. Tarjan, "Algorithmic Aspects of Vertex Elimination," Proc.7th Ann.ACM Symp.Theory of Computing, pp.245-254,1975. 140

Quantum Algorithms for Intersection and Proximity Problems

Kunihiko Sadakane, Norito Sugawara, and Takeshi Tokuyama

Graduate School of Information Sciences, Tohoku University,
Sendai, 980-8577, Japan
`{sada,norito,tokuyama}@dais.is.tohoku.ac.jp`

Abstract. We discuss applications of quantum computation to geometric data processing. Especially, we give efficient algorithms for intersection problems and proximity problems. Our algorithms are based on Brassard et al.'s amplitude amplification method, and analogous to Buhrman et al.'s algorithm for element distinctness. Revealing these applications is useful for classifying geometric problems, and also emphasizing potential usefulness of quantum computation in geometric data processing. Thus, the results will promote research and development of quantum computers and algorithms.

1 Introduction

Quantum computation is one of recent computing paradigms that may give breakthrough against barriers for the standard RAM (random access machine) model computation. Although the computational ability of current quantum computers is far from practical, its potential power is quite attractive. The theory of quantum computing has two major trends: (1) investigating the computational complexity classes to structure new hierarchy by using quantum models (e.g. [4]), and (2) comparing theoretical efficiency of the quantum model with the RAM model. In the latter trend, positive results demonstrating advantage of the quantum computation to the RAM model computation give motivations for development of quantum computers. In the literature, Grover's data search algorithm [9] and Shor's factorization algorithm [15] have been major driving-forces for the recent development of quantum computers.

One major use of commercialized computers is to handle a data set consisting of d -dimensional vectors of real numbers. A set of d -dimensional vectors can be considered to be a set of n points in the d -dimensional Euclidean space. More generally, we can consider geometric data processing (i.e. computational geometry [13]) problems, where the input is a set of n geometric objects in a d -dimensional space.

In the literature of computational geometry, several one-dimensional (or binary) data processing problems are naturally extended to multi-dimensional data processing problems; for example, sorting is extended to many problems including convex hull and Voronoi diagram, and list searching is extended to point

location, nearest neighbor searching and range searching. Analogously, it is expected that quantum algorithms for one-dimensional data processing can be adapted to multi-dimensional data processing. Moreover, the introduction of random sampling method [7,8] has created a new trend in computational geometry, and enormous research results have been produced on the application of probabilistic methods to geometric data processing. In the same way, one may expect that quantum computation (regarded as a kind of biased random sampling method) opens new aspects of computational geometry.

Related to the classification of problems via quantum computation, we want to consider the following question: “What kind of geometric problems can be solved in sublinear time in the quantum model?” In contrast to the parallel computation (PRAM) model where we seek for NC (i.e. polylogarithmic time and polynomial-size work) algorithms, it seems to be valuable to seek for sublinear time algorithms (the size of parallelism is not restricted) for the quantum model.

In a recent paper [14], the authors gave sublinear-time solutions of several geometric optimization problems by using Grover’s data-search algorithm combined with techniques of minimax parametric optimization. It demonstrated that quantum computing is considerably advantageous to RAM model in multi-dimensional data processing. For example, the constant-dimensional linear programming and the minimum enclosing ball problem can be solved in sublinear strongly-polynomial time in the quantum model.

Grover’s data-search algorithm, which is utilized in [14], can be considered as a biased random sampling method [10,11], by which an element satisfying a certain condition is sampled in a high probability. Recently, Brassard et al. [6] developed a new quantum method named *amplitude amplification*. For example, consider a randomized algorithm based on the *birthday trick* [12], in which we sample a subset of a data set, test whether the sampled set satisfies a certain condition, and answer “yes” (usually together with a certificate) if the set succeeds in the test. We iterate sampling until we encounter such a successful sample set, and answer “no” if we encounter none after a given number of trials. The efficiency of the algorithm depends on the number of iterations (i.e. the probability that the sampled subset satisfies the condition when the answer is “yes”) and the time complexity to test a sample. In the amplitude amplification paradigm, we utilize quantum algorithms instead of randomized algorithms (this may reduce the time complexity to test a sample), and also we can increase the probability that the sampled subset satisfies the required condition. Buhrman et al. [5] applied this paradigm to design an efficient algorithm for the *element distinctness* problem that detects whether the data set contains identical pair of data or not.

The aim of the present paper is to apply the framework of amplitude amplification to geometric data processing, and design sublinear-time algorithms analogously to the element distinctness algorithm.

A natural multi-dimensional generalization of the element distinctness problem is the *intersection detection problem*, that detects whether there exists an intersecting pair in a given set of geometric objects. Another generalization is the

proximity problem, that finds the nearest pair of objects. We modify Buhrman et al.’s algorithm to solve intersection detection problems and proximity problems. We do not introduce new quantum operators for solving the problems. Indeed, the idea is quite simple: We apply amplitude amplification, and utilize geometric data structures such as ray shooting and Voronoi diagrams to speed up testing each sample set. As consequences, we obtain an $\tilde{O}(n^{3/4})$ time algorithm for the triangle-point incidence detection problem, an $\tilde{O}(n^{7/8})$ time algorithm for the segment intersection detection problem, an $\tilde{O}(n^{3/4})$ time algorithm for the nearest-pair computation in the plane, and an $\tilde{O}(n^{3/4})$ time algorithm for the Hausdorff distance computation between convex polygons. We also consider some higher dimensional problems.

We remark that our algorithms are mainly of theoretical interest, and our focus is to have a better understanding of the quantum geometric data processing as a continuation of [14]. We should regard that we design algorithms in a special parallel model with the quantum constraints, rather than implementable algorithms on a real quantum computer. In practice, it is probably difficult to implement or simulate the algorithms on a large-scale instance, since they use a polynomial number of quantum bits (typically, $O(\sqrt{n})$ bits). Current quantum computers can use very small number of quantum bits, and increasing it seems to be quite difficult (or expensive). We have trade-off between the number of quantum bits and the time complexity, but it seems that designing sublinear time algorithms is difficult if we only use $O(\log n)$ quantum bits. Moreover, we need to store a data structure for each pure quantum state (i.e., as a parallel model, each processor must have a data space to store a data structure). Hence, compared to quantum algorithms using $O(\log n)$ quantum bits like Grover’s data search algorithm and geometric algorithms given in [14], it is expected to be much harder to develop a quantum computer to execute the algorithms presented in this paper.

2 Preliminaries on Quantum Algorithms

2.1 Grover’s Algorithm

It is well-known that Grover’s database search algorithm can be considered to be a biased (or controlled) sampling method. We have a set S of n data items $p_1, p_2, \dots, p_n \in U$, where U is a universe to which data items belong. We also have a function f from U to $\{1, -1\}$ such that $f(x)$ can be computed for each element x of U independently.

What we want is to find a data item p_i satisfying $f(p_i) = 1$ (we call such data items *target data items* or *target data*). If there is no target data in S , we report “none”.

In the RAM model, a naive way is to check all the member of U in $O(tn)$ time, if we can compute $f(x)$ in $O(t)$ time for each x . If there are k target data in S , random sampling is a well-known method: we randomly choose a sample from S until we find a target data. Unfortunately, this takes $O(tn/k)$ time, and does not effectively work if there are few (or no) target data in S . On the other

hand, in a parallel model such as CRCW PRAM, we can obviously solve the problem in $O(t)$ time by using n processors.

The quantum model is the intermediate between them (note that the description in this paper is simplified so that it is sufficient for solving the problems discussed here): In the quantum model, the value $f(p_i)$ is computed for each i ($i = 1, 2, \dots, n$) independently in parallel. We also compute a *state vector* $v = (v_1, v_2, \dots, v_n)$ in a parallel fashion, so that we can read the contents (e.g., p_i and $f(p_i)$) of the i -th “processor” as an output with a probability $|v_i|^2/|v|^2$. Here we consider vectors in the n -dimensional complex vector space. A major restriction is that we can only apply parallel computations realized as unitary transformations on the state vector. Moreover, the read operation affects on the state vector, and hence, we cannot re-use intermediate state vectors once we read an output. Note that the classical random algorithm can be considered as a special quantum algorithm where no interaction between processors is allowed.

Grover’s algorithm [9] consists of an alternating sequence of the following two kind of unitary operations: (1) an entrywise product operation with $(f(p_1), f(p_2), \dots, f(p_n))$, which is a unitary transformation corresponding to the diagonal matrix a $\text{diag}(f(p_1), f(p_2), \dots, f(p_n))$ and (2) *inversion about average*, which transforms v_i to $2\mu - v_i$, where μ is the average of the amplitudes of v . The algorithm has been improved to the following result [11]:

Theorem 1 (Grover[11]). *If the number k of target data in S satisfies that $k \geq k_0 > 0$, we can read a target data with a probability larger than $1/c$ for a fixed constant c after $O(t \sqrt{n/k_0})$ steps of quantum computation, where t is the time complexity for verifying whether a given data is a target data or not. Moreover, the output target data is randomly sampled from the set of all target data.*

We call the Grover’s sampling method given in the above theorem *quantum sampling* for simplicity.

2.2 Quantum Algorithm for the Element Distinctness Problem

We review a quantum algorithm to solve the element distinctness problem developed by Buhrman et al. [5]. The element distinctness problem is:

Given a set S of n data in a totally ordered set, decide whether there are identical pair of data. Moreover, pick one such pair if exists.

The quantum algorithm of Buhrman et al. is based on the amplitude amplification [6]. In the amplitude amplification paradigm, the computation of $f(x_i)$ can be done by using a quantum algorithm. Here, for simplicity, we give a modified description of it rather than explaining the amplitude amplification itself. Indeed, we regard the algorithm as a quantum version of the subset sampling method (birthday-trick), which is a basic strategy in randomized algorithms. We consider a random sample of size k from S . There are ${}_nC_k$ (the number of combinations of choosing k elements from n elements) such samples. We say a

subset Y of S an *evidence subset* if it contains at least one element containing in some identical pair. The probability that a random sample of size k becomes an evidence set is at least k/n , if S has at least one identical pair; otherwise it is obviously zero.

In the quantum model, we can amplify the random sampling so that the evidence subsets (if exist) have larger amplitudes. Now, assume that S has an identical pair. Then, can obtain an evidence subset with a constant probability in $O(t(n, k) \cdot n/k)$ time by using the quantum sampling, where $t(n, k)$ is the time for deciding a given subset Y of size k in S is an evidence set or not. For deciding whether Y is an evidence set or not, we first sort Y in $O(k \log k)$ time, and check whether x is identical to an element in Y in $O(\log k)$ time for selected elements $x \in S \setminus Y$. In RAM model, we must test for every element of $S \setminus Y$; however, in the quantum model, we can amplify so that the elements x identical to an element in Y is selected with a high probability. In precise, for each element p in S , we first search p in the sorted list in parallel (in precise, quantum-parallel), and set $f(p) = 1$ if p is identical to an element in Y (but not an element in Y); otherwise $f(p) = -1$. Then, we apply the quantum sampling spending $O(\sqrt{n} \log k)$ time. Hence, $t(n, k) = O(\sqrt{n} \log k + k \log k)$, and setting $k = \sqrt{n}$, we can detect an identical pair in $O(n^{3/4} \log n)$ time with a constant probability. This gives a one-sided bounded-error algorithm for the element distinctness problem.

Number of Quantum Bits One unfortunate fact is that we are not permitted to do measurement of the quantum state during the quantum algorithm applied as a subroutine of the amplitude amplification process (except to see the final output). Thus, instead of examining the sample set Y of size k explicitly, we must prepare the quantum state mixing all the pure states corresponding to nC_k samples. This needs $\log(nC_k \times k) = O(k \log n)$ quantum bits, which is too expensive. However, we have a better strategy: First consider a partition of m data into $\lceil m/k \rceil$ subsets each of which contains at most k elements. Instead of randomly selecting a subset with k elements, we select Y from these partitions. This enables to reduce the number of quantum bits to $O(\log n)$ ¹. We need $O(k)$ bits for storing the sorting list for Y , and hence the total space complexity is $O(k + \log n)$, which is $O(\sqrt{n})$ if we choose $k = \sqrt{n}$ to attain $O(n^{3/4} \log n)$ time complexity.

3 Quantum Algorithms for Geometric Problems

A natural geometric generalizations of the element distinctness problem is as follows:

Bicromatic intersection detection problem Given two geometric objects (one is red, and the other is blue), detect whether there is an intersection between them.

¹ This method was communicated by P. Høyer.

Intersection detection problem Given a set of geometric objects, detect whether there is an intersecting pair of objects.

Proximity problem Given a set of geometric objects, answer the nearest distance (under a given metric) between objects.

The difficulty to solve these problems depends on the shape of geometric objects and also the metric; thus, we consider several typical cases.

3.1 Incidence Detection between Points and Triangles

Let P be a set of n points and let T be a set of m geometric objects in the plane. For simplicity, we focus on the case where T is a set of triangles. We say that P and T have an incidence if there exists a triangle $\Delta_i \in T$ and a point $p_j \in P$ satisfying that $p_j \in \Delta_i$. We want to know whether P and T has an incidence or not. This is a special case of the bicromatic intersection detection problem. A very naive algorithm to solve this problem is to test every pair of point and triangle. This needs $O(nm)$ time. In computational geometry (on RAM model), we can first construct the union of triangles to have a planar subdivision, and then construct a *point location data structure*, so that for each query point, we can answer whether the point is inside the union of triangles in $O(\log m)$ time. We spend $O(m \log m)$ time and $O(m)$ space for constructing the data structure. Thus, by querying for each point of P , we can detect whether P and T has an incidence or not in $O((n + m) \log m)$ time.

Let us design an quantum algorithm. A naive idea is to consider all pairs of triangles and points, and set a function f which takes a value 1 on each incident pair, and -1 otherwise. Thus, we can apply Grover's database search algorithm. Unfortunately, there are $O(nm)$ pairs, and hence the search algorithm takes $O(\sqrt{nm})$ time. This only gives a polylogarithmic improvement (if $n = m$) over the RAM model algorithm.

Instead, we apply the amplitude amplification paradigm analogously to the solution for the element distinctness. We take a sample S of $k = \sqrt{m}$ triangles from T , and construct their union and associated point location data structure. The set S is an evidence set if S has a triangle Δ containing a point in P . We can detect whether S is an evidence set or not by finding the location of each element of P by using the data structure. In RAM model, it takes $O(n \log k)$ time; however, by applying the quantum searching, it is reduced to $O(\sqrt{n} \log k)$ time. Hence, adding the preprocessing time, it takes $O((k + \sqrt{n}) \log k)$ time.

If P and T has at least one incidence, the probability that S is an evidence set is at least k/m , and hence we can find an evidence set after taking $O(m/k)$ expected number of sampling trials, and hence it takes $O((m/k)(k + \sqrt{n}) \log k)$ time.

By applying the amplitude amplification, we can increase the amplitude, and find the evidence set after $O(\sqrt{m/k})$ unitary operations, instead of $O(m/k)$ sampling trials. Hence, the time complexity is $O(\sqrt{m/k}(k + \sqrt{n}) \log k)$. Setting $k = \sqrt{n}$, we obtain $O(m^{1/2} n^{1/4} \log n)$ time complexity.

Theorem 2. *Incidence detection between m triangles and n points can be solved in $O(m^{1/2}n^{1/4}\log n)$ time in the quantum model.*

We have the following space-time trade-off: For each sample set Y , we need $O(k)$ quantum states for the searching procedure and also need $O(k)$ space to store the point location data structure. Since we choose $k = \sqrt{n}$, we need $O(\sqrt{n})$ space. If we choose a smaller k , the number of quantum bits is reduced, with a trade off of increase of time complexity to $O(\sqrt{mn/k} \log k)$.

3.2 Segment Intersection Detection Problem in the Plane

Given a set S of n line segments in the plane, we would like to detect whether S has a pair of intersecting segments.

Our quantum algorithm is as follows: We choose a subset Y of size k from S by using quantum sampling, such that with a high probability it contains at least one segment that intersects some segment in S , provided that S has an intersecting pair of segments. Instead of sorting adopted in the element distinctness problem, we prepare a *ray shooting* data structure [1]. Here, given a query ray (halfline) ℓ , we can answer the first segment in Y intersecting ℓ . Thus, given a segment s in S , consider the ray containing s emanated from one of its endpoint, find the first intersecting segment ℓ in Y , and decide s intersects Y if and only if ℓ intersects s .

First, we check whether Y itself has an intersecting pair or not by using a plane sweep algorithm in $O(k \log k)$ time. If it has an intersecting pair, we have done; otherwise, the segments in Y are pair wise non-intersecting. It is known [1] that we can construct a data structure in $O(M)$ time for a set of mutually nonintersecting k segments such that the ray-shooting query can be done in $\tilde{O}(k/\sqrt{M})$ time, if $k < M \leq k^2$. Here $\tilde{O}$ is the big-O notation ignoring polylogarithmic factors. Since we can decide whether Y is an evidence set or not by testing for $\sqrt{n}$ quantum-sampled elements in S , it takes $t(n, k) = \tilde{O}(M + k \sqrt{n/M})$ time. Thus, the overall time for segment intersection detection becomes $O(t(n, k) \sqrt{n/k}) = \tilde{O}(\sqrt{n/k}(M + k \sqrt{n/M}))$. This is optimized if we set $k = n^{1/4}$ and $M = n^{1/2}$. We remark that this is a little disappointing, since the ray shooting with $O(k^2)$ preprocessing time is not very interesting theoretically.

Thus, the time complexity becomes $\tilde{O}(n^{7/8})$. From the nature of a sampling algorithm, the algorithm runs faster if there are many intersecting pairs.

Theorem 3. *Segment intersection problem can be solved in $\tilde{O}(n^{7/8})$ time in the quantum model.*

3.3 Triangle Intersection Detection

Consider a set T of n triangles in the plane, and we want to detect whether T has an intersecting pair of triangles. Two triangles intersect if and only if either they have intersecting pair of edges or a vertex of one of the triangles is inside the other triangle. Thus, by combining the segment intersection detection algorithm and point-triangle incidence algorithm, we have the following:

Theorem 4. *Triangle intersection problem can be solved in $\tilde{O}(n^{7/8})$ time in the quantum model.*

3.4 Nearest Pair Problem

Given a set S of n points in the plane, the nearest pair is a pair of points of S with the minimum distance. We consider the corresponding decision problem: Given a threshold value δ , decide whether there exists a pair of points whose distance is at most δ , and report such a pair if exists.

If we consider a disk of radius $\delta/2$ around each point of S , the decision problem is the intersection detection problem of the set of n disks; hence, we can regard it as a geometric version of the element distinctness problem. Here, instead of sorting, we construct the Voronoi diagram (together with a point-location data structure) of the sample set Y . We set the size k of the sample to be $\sqrt{n}$, and can solve the problem in $O(n^{3/4} \log n)$ time.

If we consider the nearest pair of a set of n points in d -dimensional space where $d \geq 3$, we utilize a nearest-neighbor-search data structure based on the range-searching method. By spending $O(M)$ preprocessing time, we can construct a nearest-neighbor-search data structure of a sample Y (of size k) so that the nearest neighbor of a point S in Y can be computed in $O(k/\tilde{M}^{1/\lceil d/2 \rceil})$ time, where $k < M \leq k^{\lceil d/2 \rceil}$. We set $M = k^{\lceil d/2 \rceil}$, and $k = n^{1/2\lceil d/2 \rceil}$ to obtain $\tilde{O}(n^{1-1/4\lceil d/2 \rceil})$ time complexity for the nearest pair decision problem.

We can solve the nearest-pair problem by using this decision problem $O(\log \Gamma)$ time by using a binary searching if the precision of the input is Γ (i.e., each coordinate value of each point is represented by a quotient number of $\log \Gamma$ bit integers). Moreover, we can design a strongly polynomial time algorithm by applying the following simple strategy: We first randomly pick two points and set the distance between them as the initial value of δ . Thus, the algorithm must report a pair of points whose distance is less than δ , if δ is not the optimal value. Then, we replace δ by the distance of the reported pair. If we examine the algorithm, the reported pair is randomly selected. In precise, the newly selected pair has a distance μ such that the number of point pairs with distances less than μ is smaller than half of those less than δ with a probability at least $1/2$. Therefore, the process stops after $O(\log n)$ iterations with a high probability. Hence, we have the following:

Theorem 5. *The nearest pair of a set of n points in d -dimensional space can be computed in $O(n^{3/4} \log^2 n)$ time for $d = 2$, and in $\tilde{O}(n^{1-1/4\lceil d/2 \rceil})$ time for $d \geq 3$.*

If we replace “nearest” by “farthest” in our algorithm, we obtain the following:

Theorem 6. *The diameter of a set of n points in d -dimensional space can be computed in $O(n^{3/4} \log^2 n)$ time for $d = 2$, and in $\tilde{O}(n^{1-1/4\lceil d/2 \rceil})$ time for $d \geq 3$.*

We can also solve the nearest bicromatic pair problem: Given sets S and T of points, compute the pair $s \in S$ and $t \in T$ minimizing the distance $d(s, t)$. Instead of element distinctness, this is a geometric version of *claw finding*[5], and the solution is analogous.

Corollary 1. *The nearest bicromatic pair problem in d -dimensional space can be computed in $O(n^{3/4} \log^2 n)$ time for $d = 2$, and in $\tilde{O}(n^{1-1/4 \lceil d/2 \rceil})$ time for $d \geq 3$, where n is the total number of points.*

Moreover, we can handle the case where T is a set of non-intersecting segments if $d = 2$, since we can use Voronoi diagrams of segments instead of nearest neighbor data structure.

Corollary 2. *Given a set S of points and T of segments in a plane, the nearest pair of a point and a segment can be computed in $O(n^{3/4} \log^2 n)$ time.*

3.5 Hausdorff Distance Computation

Consider two closed polygonal regions P and Q in the plane. The Hausdorff distance between P and Q is $d(P, Q) = \inf\{\rho \mid P \subset Q + \rho B, Q \subset P + \rho B\}$, where $+$ is the Minkowski sum and B is the closed unit disk. We also define the one-sided Hausdorff distance $d(\langle P, Q \rangle) = \inf\{\rho \mid P \subset Q + \rho B\}$. It is easy to see that $d(P, Q) = \max\{d(\langle P, Q \rangle), d(\langle Q, P \rangle)\}$. Therefore, we concentrate on the problem of computing $d(\langle P, Q \rangle)$.

We focus on the case where Q is a convex polygonal region, and compute $d(\langle P, Q \rangle)$ efficiently in the quantum model, under the condition that the linked list data structure of the boundary $C(Q)$ of Q is given, such that each vertex has pointers to incident edges. We remark that we can modify the argument for the case where we only have a list of edges of $C(Q)$ such that each edge has pointers to its adjacent edges.

Since Q is convex, $Q + \rho B$ is also convex. The boundary $C(Q + \rho B)$ of $Q + \rho B$ is a convex chain consisting of line segments and circular arcs. Let us consider the condition $P \subset Q + \rho B$. Because of the convexity, the condition $P \subset Q + \rho B$ is equivalent to the condition that all the vertices of P are contained in $Q + \rho B$.

Let $V(Q)$ and $E(Q)$ are set of vertices and (open) edges of Q , and let $F(Q) = V(Q) \cup E(Q)$ is the set of *faces* of Q . For $f \in F(Q)$, its Voronoi region is $Vor(f) = \{x \in \mathbb{R}^2 \setminus Q : \text{dist}(x, f) < \text{dist}(x, g) \text{ for } g \in F(Q) \setminus \{f\}\}$, where $\text{dist}(\cdot, \cdot)$ is the Euclidean distance. We can subdivide the plane outside Q into (closures) of Voronoi regions to have the Voronoi diagram of Q . One important observation is that the Voronoi region $Vor(f)$ can be locally computed by using the information of f . This fact holds because Q is a convex polygonal region.

For each vertex p of P , consider the face f of Q such that $p \in Vor(f)$. Without loss of generality, we can assume that such a face is unique for each p . We call f the *dominating face* of p . It is not difficult to see that the following claim holds:

Claim. $P \subset Q + \rho B$ if and only if the distance between every vertex p of P outside Q and its dominating face is at most ρ .

Thus, we can design the following algorithm: We sample a set S of k faces of Q . S is an evidence set if it has a face f such that there exists a vertex p of P such that $\text{dist}(p, f) > \rho$. We consider the Voronoi regions $\text{Vor}(f)$ for each $f \in S$, and subdivide the regions into these Voronoi regions and the complement region $U(S)$. Note that $\text{Vor}(f)$ is not the Voronoi cell of the Voronoi diagram of the sample set but that of Q itself. The complexity of the subdivision is $O(k)$, and we can construct a point-location data structure in $O(k \log k)$ time. For each vertex p of P , we find the region in the above subdivision. If $p \in U(S)$, we do nothing; otherwise, if p is in $\text{Vor}(f)$ for $f \in S$, compute $\text{dist}(p, f)$ and report that S is an evidence set if $\text{dist}(p, f) > \rho$.

In the quantum model, instead of testing all vertices of p , we can find such a point p in $O(\sqrt{n} \log k)$ time if S is an evidence set.

Moreover, if $d(\langle P, Q \rangle) > \rho$, a set S of k faces of Q is an evidence set with a probability at least n/k . Hence in the quantum model, the computation time to decide whether $d(\langle P, Q \rangle) > \rho$ or not is $O(\frac{n}{k}[\sqrt{n} \log k + k \log k])$. Thus, setting $k = \sqrt{n}$, the time complexity becomes $n^{3/4} \log n$. We can find the value $d(\langle P, Q \rangle)$ by using the binary searching, and similarly to the previous section, the number of iterations is $O(\log n)$. Hence, we have the following:

Theorem 7. *If Q is a convex polygonal region, $d(\langle P, Q \rangle)$ can be computed in $O(n^{3/4} \log^2 n)$ time in the quantum model, where n is the total number of edges in the polygonal regions P and Q .*

Corollary 3. *If both P and Q are convex and linked list structures of $C(P)$ and $C(Q)$ are given, the Hausdorff distance between them can be computed in $O(n^{3/4} \log^2 n)$ time in the quantum model.*

It is curious whether the Hausdorff distance between two simple polygonal regions can be efficiently computed or not.

4 Discussion and Concluding Remarks

We have given samples of geometric problems which can be solved efficiently (measured in processing time) in the quantum model. An important problem is to give a nice characterization of problems which can be solved efficiently (in sublinear time) in the quantum model. One observation is that each problem (in precise, its decision problem) discussed in this paper have a short proof; for example, it suffices to show a pair of intersecting segments to prove that there is an intersecting pair in a given set of segments. For the nearest pair problem, consider its decision version: “is there any pair of points whose distance is less than δ ”, then, it suffices to show such a pair to certify that the decision problem is answered “yes”. Each of the proofs has size $O(1)$, compared to the polynomial size proofs of general NP problems. For the Hausdorff distance problem, it seems that a constant size proof seems to be difficult in general; however, as we have shown in this paper, if Q is convex and the linked list structure of $C(Q)$ is given, we can give a proof of size $O(1)$ to certify that the answer to the question “is

$d(\langle P, Q \rangle) > \delta$ is “yes”. The problems discussed in [14] also have this property; for example, the decision problem of the minimum enclosing circle problem has a proof consisting of its base set (i.e. three points forming an acute triangle inscribing the circle or two points determining the diameter of the circle).

On the other hand, it is known that the parity version of the element distinctness problem needs $\Omega(n)$ time to solve in the quantum model, and it seems to be difficult to give a short proof to the parity problem. It seems to be valuable to determine what kind of problems having short proofs can be efficiently solved in the quantum model.

Acknowledgement

The authors gratefully acknowledge to Kazuo Iwama, who informed the reference [5]. Indeed, the information initiated this research. They much appreciate Peter Høyer who gave many professional comments on an earlier version of this paper, and corrected some errors in understanding of the algorithm for the element distinctness. Also they thank Takashi Mihara for his valuable comments.

References

1. P. Agarwal, Range searching, Section 31 of *Handbook of Discrete and Computational Geometry* (1997) 575–598, CRC Press. 154
2. P. Agarwal and M. Sharir, Efficient Algorithms for Geometric Optimization *Research Report, Duke University, 2708-0129* (1997).
3. A. Ambanis, A better bound for quantum algorithms searching an ordered list, *Proc. 40th FOCS* (1999) 352–357.
4. E. Bernstein and U. Vazirani, Quantum complexity theory, *SIAM Journal on Computing* **26**(5) (1997) 1411–1473. 148
5. H. Buhrman, C. Dürr, M. Heiligman, P. Høyer, F. Magniez, M. Santha, and R. de Wolf, Quantum algorithms for element distinctness, quant-ph/0007016, *Proc. IEEE Complexity 2001* 149, 151, 156, 158
6. G. Brassard, P. Høyer, M. Mosca, and A. Tapp. Quantum amplitude amplification and estimation, quant-ph/0005055, to appear in *Quantum Computation and Quantum Information: A Millenium Volume*, AMS Contemporary Series, ed. S. Lomonaco. 149, 151
7. K. Clarkson, New applications of random sampling in computational geometry, *Discrete & Computational Geometry* **2** (1987) 195–222. 149
8. K. Clarkson and P. Shor, Applications of random sampling in computational geometry II, *Discrete & Computational Geometry* **4** (1989) 387–421. 149
9. L. Grover, A faster quantum mechanical algorithm for database search, *Proc. 28th STOC* (1996) 212–219. 148, 151
10. L. Grover, Quantum search on structured problems, *Chaos, Solution, & Frctals* **10** (1999), 1695–1705. 149
11. L. Grover, Rapid sampling through quantum computing, *Proc. 32nd STOC* (2000), 618–626. 149, 151
12. R. Motowani and P. Raghavan, *Randomized Algorithms*, Cambridge Univ. Press, 1995. 149

13. F. Preparata and I. Shamos, *Computational Geometry – An Introduction* (2nd edition), Springer-Verlag, 1988. 148
14. K. Sadakane, N. Sugawara, and T. Tokuyama, Quantum computation in computational geometry, submitted (2001). 149, 150, 158
15. P. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM J. Comput.*, **26(5)** (1997) 1484-1509. 148

BUSHWHACK: An Approximation Algorithm for Minimal Paths through Pseudo-Euclidean Spaces[□]

Zheng Sun and John Reif

Department of Computer Science, Duke University
Durham, NC 27708, USA
{sunz, reif}@cs.duke.edu

Abstract. In this paper we define *piecewise pseudo-Euclidean optimal path problems*, where each region has a distinct cost metric of a class we call pseudo-Euclidean, that allows the path cost to possibly vary within the region in a predictable and efficiently computable way. This pseudo-Euclidean class of costs allows us to model a wide variety of various geographical features. We provide an approximation algorithm named *BUSHWHACK* that efficiently solves these piecewise pseudo-Euclidean optimal path problems. BUSHWHACK uses a previously known technique of dynamically generating a discretization in progress. However, it combines with this technique a “lazy” and best-first path propagation scheme so that fewer edges need to be added into the discretization. We show both analytically and experimentally that BUSHWHACK is more efficient than approximation algorithms based on Dijkstra’s algorithm.

1 Introduction

In many applications, such as robotic motion planning and geographical information systems, there arise *optimal path problems* where each problem is to find a minimal cost path in the plane. One common assumption made by many previous studies on these problems is that, for any s and t , an optimal path from s to t is the straight line segment $\overline{st}$ if $\overline{st}$ lies entirely inside the free space.

In recent years there has been an increasing attention and motivation on path planning problems with various non-Euclidean metrics. If the free space consists of multiple regions and the metric is not the same for all regions, the straight line segment $\overline{st}$ may no longer be an optimal path from s to t , even if $\overline{st}$ lies in the free space. Therefore, many techniques developed in previous motion planning works are no longer valid.

In the weighted region optimal path problem ([5,4,3,1,6,2]), the entire free space is divided into polygonal regions each of which is associated with a unit weight. The cost of a path p is defined to be the weighted sum of the lengths of the segments of p inside each region. Another example is the flow problem ([7]),

* Supported by NSF ITR EIA-0086015, NSF-IRI-9619647, NSF CCR-9725021, SEGR Award NSF-11S-01-94604, Office of Naval Research Contract N00014-99-1-0406.

where inside each region there is a flow defined by a vector, and the cost of path p is the total travel time on p by robot with a fixed maximum velocity.

To solve the weighted region optimal path problem, a number of previous works ([3,1,2]) used a discretization of the problem based on edge subdivision, and Dijkstra’s algorithm to find an optimal path in the graph induced by discretization. Aleksandrov et al [1] proposed a logarithmic discretization that guarantees an ϵ -short approximation, where $m = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ Steiner points are placed on each boundary edge. Their algorithm then applies a “pruned” version of Dijkstra’s algorithm to find an optimal path in the discrete graph in $O(\frac{n}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \log n) \log \frac{1}{\epsilon})$ time, where n is the number of all boundary edges.

The algorithm BUSHWHACK presented here uses a subgraph to render an optimal path in the discrete graph by dynamically adding edges. This technique has been used by a number of prior precedents, but our BUSHWHACK algorithm, by adopting a “lazy” and best-first path propagation scheme, is able to use fewer edges to compute an optimal path in the discrete graph.

Some of the key features of the BUSHWHACK algorithm were used in an approximation algorithm of Reif and Sun [6] introduced to approximately solve the weighted region optimal path problem; that algorithm finds an optimal path in the discrete graph in time $O(nm \log nm)$. The generalized algorithm BUSHWHACK can be used to compute approximate optimal paths in $2\mathcal{D}$ spaces that satisfy a wide range of possible geometric properties. In Section 2 we define a class of spaces that we call *Piecewise Pseudo-Euclidean Space*. We show that the BUSHWHACK algorithm can be applied to an optimal path problem in any space in this class. An immediate application is path planning in an area with various geographical features such as plains (regions with low unit costs), swamps (regions with high unit costs), and rivers and tides (regions with flows).

The focus of this paper is on the generality of the BUSHWHACK algorithm and the characterizations of the spaces to which BUSHWHACK can be applied. However, we feel that it is important to make a brief digression here on its efficiency. For the weighted region optimal path problem, by applying BUSHWHACK to the logarithmic discretization scheme proposed by [2], we can have an algorithm that computes an ϵ -short approximate optimal path in $O(nm \log nm) = O(\frac{n}{\epsilon}(\log \frac{1}{\epsilon} + \log n) \log \frac{1}{\epsilon})$ time. This improves on all other approximation algorithms, including those mentioned above.

2 Preliminaries

A convex polygonal region r is said to be a *pseudo-Euclidean region* if it satisfies the following two properties.

Property 1 *Region r is associated with a cost function $d_r : (\mathcal{R}^2, \mathcal{R}^2) \rightarrow \mathcal{R}^+ \cup \{0\}$ so that, for any two points x and y in r , the cost of the straight line path $\overline{xy}$ is $d_r(x, y)$. $d_r(x, y) = 0$ if and only if $x = y$. The cost function d_r has the property that the path with the least cost, among all paths from x to y that lie completely inside r , is the straight line segment $\overline{xy}$.*

Property 2 *Letting x_0 be a point in region r (including the boundary) and letting $e = \overline{v_0 v_1}$ be an edge of r that is not incident to x_0 , then there are only a small number of local extrema for function $g_{x_0, e} : [0, 1] \rightarrow \mathcal{R}^+$, where $g_{x_0, e}(\lambda) \equiv d_r(x_0, (1 - \lambda)v_0 + \lambda v_1)$. These local extrema can be computed efficiently.*

In the following discussion, we will refer to $d_r(x, y)$ as the *region distance*, or *region cost*, from x to y . A space is said to be a *piecewise pseudo-Euclidean space* if it consists of a finite number of pseudo-Euclidean regions. For an arbitrary path p in the space, if p can be divided into segments $p_1, p_2, \dots, p_m$ so that each $p_i, 1 \leq i \leq m$, lies entirely inside a region r , the cost of p is defined to be the sum of the costs of all segments, each of which is determined by the respective region cost function. In case a segment $\overline{xy}$ lies on a boundary edge e , we define its cost $d_e(x, y)$ to be $\min\{d_r(x, y), d_{r'}(x, y)\}$, where r and r' are the neighboring regions of e . A *piecewise pseudo-Euclidean optimal path problem* is to find an optimal path (i.e., the path with the least cost) from a source point s to a destination point t in a piecewise pseudo-Euclidean space. Here s and t are both vertices of some pseudo-Euclidean regions.

According to Property 1, an optimal path in a piecewise pseudo-Euclidean space is piecewise linear. A segment of a path is said to be “edge-crawling” if it lies on an edge; or “face-crossing” if it cuts through a region. We call a path a “face-crossing” (“edge-crawling”) path if the last segment of the path is “face-crossing” (“edge-crawling”, respectively). Although a piecewise pseudo-Euclidean space may consist of hybrid regions whose cost functions can be completely different, inside each region it is a Euclidean-like space because the shortest path between two points inside the region is still the straight line segment.

At a later point we will explain the importance of Property 2 to our algorithm; we will also show how “efficient” the computation of local extrema needs to be. Although this property may seem to be restrictive, in practice many optimal path planning problems satisfy this property. Examples are the weighted region problem and flow problem mentioned in the previous section.

Here we first introduce some notations that will be used in the rest of the paper. We let S be a polygonal decomposition of the planar space and let V be the set of vertices in S . We use E to denote the set of all boundary edges in S and let $n = |E|$. Without loss of generality, we assume that each region is a triangle. For any path p , we let $d(p)$ denote the cost of p . For two paths p_1 and p_2 , we let $p_1 + p_2$ denote the concatenation of p_1 and p_2 . If $p = p_1 + p_2$, we say p is an *extension* of p_1 . In particular, if $p = p_1 + \overline{v_1 v_2}$, we say p is a *one-segment extension* of p_1 . For any two points x and y , we use $p(x, y)$ to denote a path from x to y and use $p_{opt}(x, y)$ to denote an optimal path from x to y . We define the “distance” from s to t , $d_{opt}(s, t)$, to be the cost of $p_{opt}(s, t)$. At any time during the search of an optimal path from s to t , a point v is said to be *discovered* if and only if $d_{opt}(s, v)$ is determined.

A natural approach to these problems is to discretize the 2D space by introducing Steiner points. For each boundary edge $e \in E$, we add m Steiner points on e for some positive integer m . Let V_s be the set of Steiner points and let $V' = V \cup V_s$. A directed discrete graph $G(V', E')$ is constructed by intercon-

necting points in V' that are on the boundary of the same region. Each edge (x, y) in G is assigned a weight $w(x, y)$ where $w(x, y)$ is defined to be $d_e(x, y)$ if $\overline{xy}$ is on edge e ; or $d_r(x, y)$ if $\overline{xy}$ crosses region r of S .

By constructing G , the original path planning problem in a continuous space is transformed to the problem of finding a minimum path in the discrete graph. The latter problem can be solved by Dijkstra's algorithm. A path in G is called a *discrete path*. An optimal discrete path found from s to t is then used to approximate an optimal path in the original continuous space. The more Steiner points we place on each edge of S , the more accurate the approximation will be. Aleksandrov et al [2] showed that there exists a discretization, with $m = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ Steiner points inserted on each boundary edge, that guarantees an ϵ -short approximation of an optimal path from s to t .

It takes $O(|E'| + |V'| \log |V'|) = O(nm^2 + nm \log nm)$ time to find an optimal discrete path in G using Dijkstra's algorithm. Observe that, when m is large, the dominant part of the time complexity is $O(nm^2) = O(|E'|)$. To reduce the cost of processing edges of G , Aleksandrov et al [2] proposed a "pruned" version of Dijkstra's algorithm. By exploiting the fact that the in-angle and out-angle of an optimal path at a bending point obey "Snell's Law," their algorithm only uses a sparse subgraph $G'(V', E'')$ of G which still yields an optimal discrete path from s to t in G . The number of edges included in subgraph G' , $|E''|$, is $O(n/\epsilon^{\frac{3}{2}} \log \frac{1}{\epsilon}) = O(\sqrt{\epsilon}/\log \frac{1}{\epsilon} \cdot |E'|)$. The total time complexity of the algorithm is therefore reduced from $O(\frac{n}{\epsilon}(\frac{1}{\epsilon} \log \frac{1}{\epsilon} + \log n) \log \frac{1}{\epsilon})$ to $O(\frac{n}{\epsilon}(\frac{1}{\sqrt{\epsilon}} + \log n) \log \frac{1}{\epsilon})$.

Our BUSHWHACK algorithm follows the same discretization approach. However, by maintaining a collection of data structures called *intervals*, on average BUSHWHACK only needs to evaluate for each point v the costs of $O(\log m)$ adjacent edges of v , as we will show at the end of Section 4. The total number of edges accessed by the algorithm is thus $O(nm \log m) = O(\epsilon \cdot |E'|)$. Our BUSHWHACK algorithm can therefore find an optimal discrete path efficiently in $O(nm \log nm) = O(\frac{n}{\epsilon}(\log \frac{1}{\epsilon} + \log n) \log \frac{1}{\epsilon})$ time. More importantly, compared to the "pruned" Dijkstra's algorithms [1,2], BUSHWHACK makes weaker assumptions on the metric inside each region and thus can also be applied to other piecewise pseudo-Euclidean optimal path problems.

As the goal of our algorithm is to find the exact optimal discrete path, in the following discussion, wherever we refer to an "optimal path," we mean an optimal discrete path in G unless specified otherwise. We let $p'_{opt}(s, t)$ denote an optimal discrete path from s to t and let $d'_{opt}(s, t)$ be the cost of $p'_{opt}(s, t)$.

3 Intervals

The BUSHWHACK algorithm works similarly to Dijkstra's algorithm. It keeps a sorted list QLIST of candidate optimal paths. At each step, the candidate optimal path p_{min} with the minimum cost is extracted from QLIST. Consequently, a number of candidate optimal paths are inserted into QLIST that are one-segment extensions of p_{min} . (We call this process "propagation.") The iteration continues until the destination point is reached.

Dijkstra's algorithm can be used to compute an optimal path in an arbitrary weighted graph, while the aforementioned discrete graph G is derived from a piecewise pseudo-Euclidean space with certain geometric properties. Therefore, directly applying Dijkstra's algorithm to G does not fully utilize the underlying geometric properties. In particular, Property 1 implies the following lemma:

Lemma 1 *Any two optimal paths in G with the same source point cannot intersect in the interior of any region.*

To keep track of useful line segments, we introduce a data structure named *interval*. Let r be a region of S and let e, e' be two boundary edges of r . Let v be any discovered point on e that is not incident to e' . Interval $I_{v,e,e^\square}$ is defined to be $I_{v,e,e^\square} = \{v^* \in e' \mid d_r(v, v^*) + d'_{opt}(s, v) \leq d_r(v', v^*) + d'_{opt}(s, v') \forall v' \in \text{PLIST}_e\}$, where PLIST_e is the list that includes all discovered points on e . That is, for any point v^* on e' , $v^* \in I_{v,e,e^\square}$ if and only if path $p'_{opt}(s, v) + \overline{vv^*}$ is the least costly path among all paths from s to v^* that are one-segment extensions of optimal paths from s to discovered points on e . For any edge e and e' that share a region r , we use $\text{ILIST}_{e,e^\square}$ to denote the list of intervals $I_{v,e,e^\square}$ for all $v \in \text{PLIST}_e$.

From this definition of intervals we can conclude that, for any discovered point v on e and any point v^* on e' , $\overline{vv^*}$ can not be part of any optimal path originated from s that enters region r through point v if $v^* \notin I_{v,e,e^\square}$. Therefore, by maintaining $\text{ILIST}_{e,e^\square}$ for each e and e' that share a region, the BUSHWHACK algorithm is able to avoid accessing most of the edges in G . Observe that each interval $I_{v,e,e^\square}$ is a dynamic set of points on e' . It is first created when v is discovered. When more points on e are discovered, PLIST_e will contain more points and thus $I_{v,e,e^\square}$ may also be changed, according to the definition.

Lemma 1 implies that each interval is composed of consecutive points on e' (which leads us to name this data structure “interval”). Further, an interval $I_{v,e,e^\square}$ is located to the left (right) of another interval $I_{v',e,e^\square}$ on e' if and only if v is located to the left (right, respectively) of v' on e . Figure 1.a shows how points on edge e' are partitioned into intervals corresponding to discovered points on e . We claim that the two end points of an interval $I_{v,e,e^\square}$ can be computed efficiently (in $O(\log m)$ time) when it is initially created.

For each $v^* \in I_{v,e,e^\square}$, the face-crossing path $p'_{opt}(s, v) + \overline{vv^*}$ needs to be considered as a candidate optimal path from s to v^* . We call such a path a *direct interval path* associated with $I_{v,e,e^\square}$. One strategy is to insert all these paths into QLIST simultaneously when v is discovered. However, this may not be efficient as both $\text{ILIST}_{e,e^\square}$ and intervals in $\text{ILIST}_{e,e^\square}$ are dynamic data structures. Whenever a new point on e is discovered, a new interval (although possibly empty) will be created and inserted into $\text{ILIST}_{e,e^\square}$. If the new interval is not empty, the ranges of the two neighboring intervals will be adjusted.

As shown in 1.b, even though a point v^* originally is in $I_{v_2,e,e^\square}$, after a new point $v_{new} \in e$ is discovered, v^* may fall into the range of the new interval $I_{v_{new},e,e^\square}$. If this is the case, path $p'_{opt}(s, v_2) + \overline{v_2v^*}$ no longer needs to be considered as an optimal path from s to v^* as it is more costly than $p'_{opt}(s, v_{new}) + \overline{v_{new}v^*}$, according to the definition of “interval.”

A more efficient strategy is to insert direct interval paths in a “lazy” and best-first manner. Interval paths associated with $\text{ILIST}_{e,e^\square}$ are sorted in the increasing order of path cost. A path $p'_{\text{opt}}(s, v) + \overline{vv^*}$ is inserted into QLIST only when the previous path is extracted from the list, and only if v^* is still in $I_{v,e,e^\square}$. This strategy will avoid inserting a path $p'_{\text{opt}}(s, v) + \overline{vv^*}$ into QLIST if v^* is later “switched” to another interval.

To achieve this, we need to sort efficiently the direct interval paths by path cost. Since these paths are all one-segment extensions of $p'_{\text{opt}}(s, v)$, we only need to sort $d_r(v, v^*)$ for all $v^* \in I_{v,e,e^\square}$. According to Property 2, the region distance function from v to points on e' has a constant number of local extrema. Thus, $I_{v,e,e^\square}$ can be divided into a constant number of parts by these local extrema so that the region distance from v to points in each part is monotonically increasing or decreasing. We create a *monotonic interval* for each monotonic part of $I_{v,e,e^\square}$ and replace $I_{v,e,e^\square}$ by these intervals in $\text{ILIST}_{e,e^\square}$. Points in each such interval are already sorted by region distance to v . In the following discussion, we always assume that each interval is monotonic. For the weighted region optimal path problem, each region is a Euclidean space and thus each interval $I_{v,e,e^\square}$ will at most be split into two monotonic intervals by the perpendicular point of v on e' , as illustrated in Figure 2.a. The same is true for a region with a uniform flow, although computing the split point is more complicated as shown in [7].

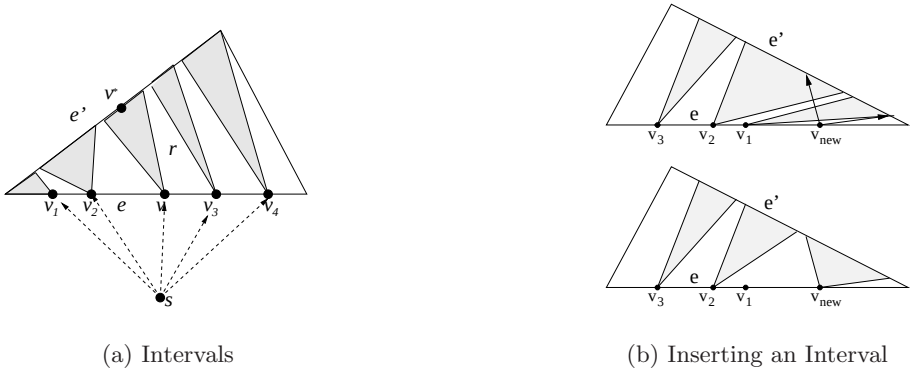


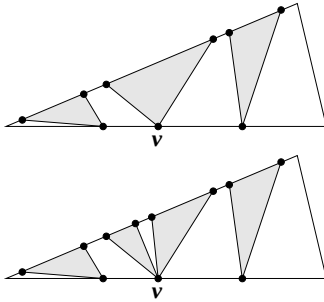
Fig. 1. Intervals and Inserting an Interval

Suppose interval $I_{v,e,e^\square}$ contains points $v_1^*, v_2^*, \dots, v_d^*$ when it is initially created (i.e., when v is discovered), as shown in Figure 2.b. Here $v_1^*, v_2^*, \dots, v_d^*$ are consecutive points on e' and v_1^* and v_d^* are the two end points of the interval. W.L.O.G, we assume $d_r(v, v_1^*) \leq d_r(v, v_d^*)$. As interval $I_{v,e,e^\square}$ is monotonic, we have $d_r(v, v_1^*) \leq d_r(v, v_2^*) \leq \dots \leq d_r(v, v_d^*)$, where r is the region incident to both e and e' . Let $p_1, p_2, \dots, p_d$ be direct interval paths associ-

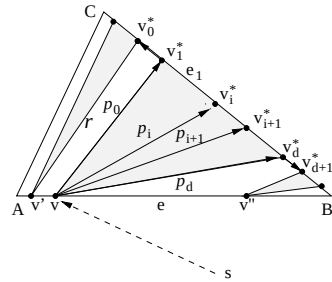
ated with $I_{v,e,e^\square}$, where $p_i = p'_{\text{opt}}(s,v) + \overline{vv_i^*}$ for $1 \leq i \leq d$. For each v_i^* , let $P_i = \{p_{j,i} = p_j + \overline{v_j^*v_i^*} \mid 1 \leq j \leq d\}$. Observe that $p_i \in P_i$ as $p_i = p_i + \overline{v_i^*v_i^*}$. All paths in P_i except p_i are *extended interval paths* that are one-segment extensions of direct interval paths associated with $I_{v,e,e^\square}$. We call both direct interval paths and extended interval paths *interval paths*. P_i is the set of all interval paths associated with $I_{v,e,e^\square}$ that connect s and v_i^* .

We say interval path $p' \in P_i$ is *locally optimal* if $d(p') = \min\{d(p) \mid p \in P_i\}$. For each v_i^* , we want to insert into QLIST only one locally optimal interval path p_i^* that connects s and v_i^* . Initially, interval path $p_1^* = p_1$ is inserted into QLIST. Iteratively, an interval path p_i^* from s to v_i^* is added into QLIST when the interval path p_{i-1}^* from s to v_{i-1}^* is extracted from QLIST. p_i^* is defined to be the less costly path between two paths, p_i and $p_{i-1}^* + \overline{v_{i-1}^*v_i^*}$. That is, the interval path for v_i^* can be constructed by either extending $p'_{\text{opt}}(s,v)$ by line segment $\overline{vv_i^*}$, or extending p_{i-1}^* by line segment $\overline{v_{i-1}^*v_i^*}$, whichever is less costly. The propagation process terminates when all points in $I_{v,e,e^\square}$ are reached by such interval paths. Observe that this process may be terminated before p_d^* is generated and inserted into QLIST. This would occur when another interval $I_{v',e',e'^\square}$ is created that re-adjust the range of $I_{v,e,e^\square}$. We can establish the following theorem (we include the proof in the full version of this paper):

Theorem 1 Each p_i^* is locally optimal.



(a) Splitting an Interval



(b) Propagating Interval Paths

Fig. 2. Operations on Intervals

By the BUSHWHACK algorithm, each interval will generate no more than d locally optimal interval paths, where d is the number of points inside the interval. Thus, for each interval list $\text{ILIST}_{e,e^\square}$, only $m + 2$ interval paths are inserted into QLIST, where m is the number of Steiner points on e' . The total number of interval paths, therefore, is $O(mn)$.

We need to show that this propagation scheme can find an optimal discrete path from s to v^* for any $v^* \in V'$. Following the notations used previously, we let e and e' be two boundary edges of region r . Let $p'_{opt}(s, v^*)$ be an optimal discrete path from s to point $v^* \in e'$ that enters region r through point $v \in e$. As shown in Figure 3, $p'_{opt}(s, v^*)$ can be categorized into one of the following four types: (1) $p'_{opt}(s, v^*) = p'_{opt}(s, v) + \overline{vv^*}$ is a face-crossing path; (2) $p'_{opt}(s, v^*) = p'_{opt}(s, v) + \overline{vu^*} + \overline{u^*v^*}$ is an edge-crawling path where $u^*, v^* \in I_{v,e,e^\square}$; (3) $p'_{opt}(s, v^*) = p'_{opt}(s, v) + \overline{vu^*} + \overline{u^*v^*}$ is an edge-crawling path where $u^* \in I_{v,e,e^\square}$ and $v^* \notin I_{v,e,e^\square}$; and (4) $p'_{opt}(s, v^*) = p'_{opt}(s, v) + \overline{vv^*}$ is an edge-crawling path where v is the joint end point of e' and e .

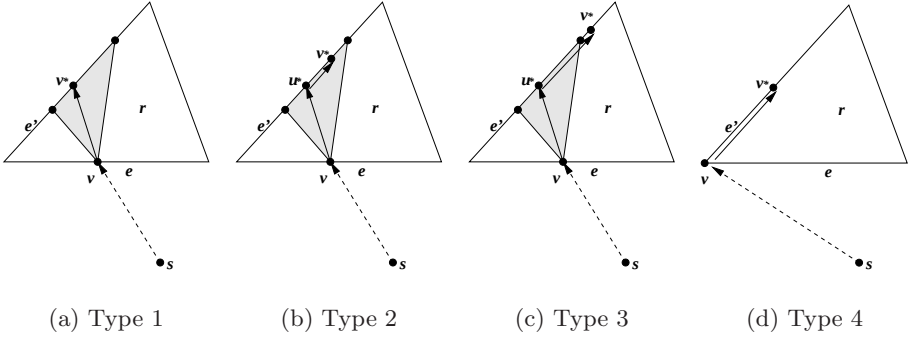


Fig. 3. Four Types of Optimal Paths

It is easy to see that this propagation scheme can find an optimal path $p'_{opt}(s, v^*)$ if $p'_{opt}(s, v^*)$ is of Type 1 or 2, as a Type 1 optimal path is a direct interval path and a Type 2 optimal path is an extended interval path. To capture an optimal path of Type 3, we need to create two other paths for each interval. Let v_0^* be the point next to v_1^* outside interval $I_{v,e,e^\square}$, and let v' be the discovered point on e whose interval $I_{v,e,e^\square}$ contains v_0^* . We insert path $p_0^* = p_1^* + \overline{v_1^*v_0^*}$ into QLIST when $I_{v,e,e^\square}$ is created, if $d'_{opt}(s, v) + d_r(v, v_1^*) + d_e(v_1^*, v_0^*) < d'_{opt}(s, v') + d_r(v', v_0^*)$. Similarly, we will add path $p_{d+1}^* = p_d^* + \overline{v_d^*v_{d+1}^*}$, if necessary, after all interval paths associated with this interval are inserted. Here v_{d+1}^* is the point next to v_d^* outside $I_{v,e,e^\square}$. We call these two paths *non-interval paths* as they are not associated with any interval. These non-interval paths will also be propagated when they are removed from QLIST, as we will show later in this paper. It will also be clear in the next section how our algorithm finds optimal paths of Type 4.

4 Algorithm

The BUSHWHACK algorithm maintains three types of dynamic lists: (a) QLIST, the list of candidate optimal paths sorted by path cost; (b) PLIST_e, the list of discovered points on edge e ; and (c) ILIST_{e,e[□]}, the list of intervals for edge e and e' that are on the boundary of the same region.

As we mentioned previously, all paths can be divided into two categories, interval paths and non-interval paths. All face-crossing paths along with some edge-crawling paths are interval paths. We have shown how two non-interval paths are generated for each interval. This section will explain how other non-interval paths are created by extending these paths.

The main body of the BUSHWHACK algorithm is a loop. Each iteration, the candidate optimal path p_{min} in QLIST with the minimum cost is extracted from the list. Let v be the ending point of path p . If v is not a discovered point (i.e., the distance from s to v is not yet decided), we claim that path p is an optimal path from s to v , and $d(p)$ is the distance from s to v .

```

FindPath(s,t)
  insert path  $p_{opt}^{\square}(s, s)$  into QLIST [1]
  while  $t$  is not reached [2]
    extract the least costly path  $p(s, v)$  from QLIST [3]
    if  $v$  is a not a discovered point [4]
       $p_{opt}^{\square}(s, v) = p(s, v)$ ;  $d_{opt}^{\square}(s, v) = d(p(s, v))$  [5]
      HandleNewDiscovery( $v$ ) [6]
      Propagate( $v, p$ ) [7]

```

Function HandleNewDiscovery(v) creates new intervals for the newly discovered point v , and then inserts into QLIST an interval path associated with each of these intervals.

```

HandleNewDiscovery( $v$ )
  if  $v$  is a Steiner point on an edge  $e$  [1]
    for each region  $r$  incident to  $e$  [2]
      for each edge  $e^{\square}$  of  $r$  that is not  $e$  [3]
        create interval  $I_{v,e,e^{\square}}$  [4]
  else ( $v$  is a vertex in  $S$ ) [5]
    for each edge  $e$  incident to  $v$  [6]
      let  $v_{next}$  be the neighboring Steiner point of  $v$  on  $e$  [7]
      insert path  $p_{new} = p + \overline{vv_{next}}$  into QLIST [8]
      for each region  $r$  incident to  $e$  [9]
        for each edge  $e^{\square}$  of  $r$  that is not  $e$  [10]
          create interval  $I_{v,e,e^{\square}}$  [11]
  for each newly created interval  $I_{v,e,e^{\square}}$  [12]
    split  $I_{v,e,e^{\square}}$  into monotonic intervals  $I_{v,e,e^{\square}}^1, I_{v,e,e^{\square}}^2, \dots, I_{v,e,e^{\square}}^i$  [13]
    for each monotonic interval  $I_{v,e,e^{\square}}^j, 1 \leq j \leq i$  [14]
      add the first non-interval path for  $I_{v,e,e^{\square}}$  if necessary [15]
      add the first interval path for  $I_{v,e,e^{\square}}^j$  [16]

```

Whether or not v is a newly discovered point, Function Propagate(v, p) creates candidate optimal paths by propagating p in a constant number of ways and inserts these paths into QLIST.

```

Propagate( $v, p$ )
  if  $p$  is an interval path associated with  $I_{v^\square, e^\square, e}$  [1]
    if  $p$  is still valid [2]
      if  $v$  is the last point in  $I_{v^\square, e^\square, e^\square}$  [3]
        add the second non-interval path for  $I_{v^\square, e^\square, e^\square}$  if necessary [4]
      else [5]
        add the next interval path for  $I_{v^\square, e^\square, e}$  [6]
    else ( $p$  is an edge-crawling path whose last segment is on edge  $e$ ) [7]
      if  $v$  is not an end point of  $e$  [8]
        let  $v_{prev}$  be the previous point of  $v$  on path  $p$  [9]
        let  $v_{next}$  be the neighboring point of  $v$  that is not between  $v_{prev}$  and  $v$  [10]
        if there has not already been a path that extends to  $v_{next}$  from  $v$  [11]
          insert path  $p_{new} = p + \overline{vv_{next}}$  into QLIST [12]

```

We have explained previously how paths are propagated inside intervals. Observe that the task of handling interval paths is accomplished by the combination of the procedures **Propagate** and **HandleNewDiscovery**. For example, intervals are created in the procedure **HandleNewDiscovery** when a point v is discovered. At the same time, the first interval path associated with each new interval is inserted into QLIST (line 16 of the procedure **HandleNewDiscovery**). The propagation of interval paths for each interval is accomplished in the procedure **Propagate** (line 6). Generating the two non-interval paths for each interval is handled through **HandleNewDiscovery** (line 16) and **Propagate** (line 4).

Each interval also generates two non-interval paths, one when the interval is created (line 16 of the procedure **HandleNewDiscovery**) and the other when the last interval path of that interval is extracted from QLIST (line 4 of the procedure **Propagate**). Another situation that generates non-interval paths is that the newly discovered point v is an end point of an edge e . In this case, a non-interval path is inserted into QLIST that extends $p'_{opt}(s, v)$ to the neighboring Steiner point of v along edge e , as indicated by line 8 of **HandleNewDiscovery**.

All the non-interval paths are edge-crawling paths. According to the procedure **Propagate** (line 8 to line 12), when a non-interval path p from s to v is extracted from QLIST, we may insert an extension of this path into QLIST. Suppose $\overline{v_{prev}v}$ is the last segment of path p . Since p is edge-crawling, v_{prev} is on the same edge e as v . Let v_{next} be the adjacent Steiner point of v on e that is on the other side of v_{prev} . We insert path $p + \overline{vv_{next}}$ into QLIST, if there has not been another path $p' + \overline{vv_{next}}$ inserted into QLIST. The propagation of non-interval paths guarantees that any optimal path of Type 3 or 4 will not be missed by our algorithm.

For each Steiner point $v \in V_s$, there will be at most two non-interval paths from s to v inserted into QLIST, one approaching v from left and one approaching v from right. Similarly, for any vertex $v \in V$, there will be at most $d(v)$ non-interval paths that connect s and v , one from each edge incident to v . Here $d(v)$ is the number of incident boundary edges of v in the original triangular decomposition. Thus, the total number of non-interval paths is bounded by $O(mn)$, and therefore the total number of all paths inserted into QLIST is $O(mn)$.

To show that the BUSHWHACK algorithm is correct, it is sufficient to prove the following theorem: (we include the proof in the full version of this paper):

Theorem 2 *When path $p(s, v)$ is extracted from QLIST, if v is not yet discovered, p is an optimal path from s to v in discrete graph G .*

The complexity of the BUSHWHACK algorithm depends on three factors: (a) the cost of maintaining QLIST which is $O(mn \log mn)$, as at most $O(mn)$ candidate optimal paths are inserted into QLIST; (b) the cost of maintaining all discovered point lists PLIST_e which is $O(nm \log m)$; (c) the cost of maintaining all interval lists $\text{ILIST}_{e, e^\circ}$ which is $O(nm \log m)$. The complexity of the algorithm, therefore, is $O(nm \log nm)$.

In the Introduction section we claimed that, in average, for each Steiner point v BUSHWHACK needs to evaluate the costs of only $O(\log m)$ adjacent edges of v . Even though inside each region only $O(m)$ edges are ever used by candidate optimal paths inserted into QLIST, that is, $O(1)$ edges per Steiner point in the region, BUSHWHACK has to evaluate the costs of additional edges in order to maintain the intervals. To decide the boundary of a new interval I_{v, e, e° , BUSHWHACK needs to take a binary search of $O(\log m)$ steps. At each step, BUSHWHACK has to compare the cost of $\overline{vv^*}$ for some $v^* \in e'$ with the cost of $\overline{v'v^*}$, where v' is one of the two neighboring discovered points of v on e . As a result, $O(\log m)$ edges are evaluated for each Steiner point.

When a new interval I_{v, e, e° is created, we need to divide it into monotonic intervals. To do that, we need to compute the local extrema of the region distance function $g_{v, e}$ from v to points on e' . By Property 2 a pseudo-Euclidean region requires that these local extrema can be computed “efficiently.” But we have not yet specified how efficient the computation needs to be. As only one splitting is performed for each interval, as long as the cost of computing local extrema (and thus the cost of splitting) for each interval does not exceed $O(\log m)$, the total cost of splitting will be bounded by $O(nm \log m)$ and will not affect the total complexity of $O(nm \log nm)$. It should be noted that we only need to find among m values (corresponding to m Steiner points) the values closest to the local extrema. For a weighted region or a flow region, such local extrema can be computed in constant time. In the full version of the paper, we will show that, as long as $g_{v, e}$ itself is a constant degree polynomial function or computing the local extrema of $g_{v, e}$ can be converted to computing the roots of a constant degree polynomial function, the local extrema of $g_{v, e}$ can be computed in $O(\log m)$ time.

5 Preliminary Experimental Results

In this section we report on some preliminary experimental results produced by *IntervalPathFinder*, a Java implementation of the BUSHWHACK algorithm. We compared our algorithm against Dijkstra’s algorithm by running experiments on the same group of artificially generated datasets for both algorithms. All experiments are performed on a Windows 2000 workstation with 256MB memory and a 550MHz Pentium III processor. For simplicity we used uniform discretization and each edge has equal number of Steiner points. The results of these

experiments, as shown in Figure 4, are consistent with our complexity analysis. Although our interval-based BUSHWHACK is slower when m is small due to the high cost of maintaining various complex data structures, its efficiency quickly becomes evident when m is increased to 256 and above.

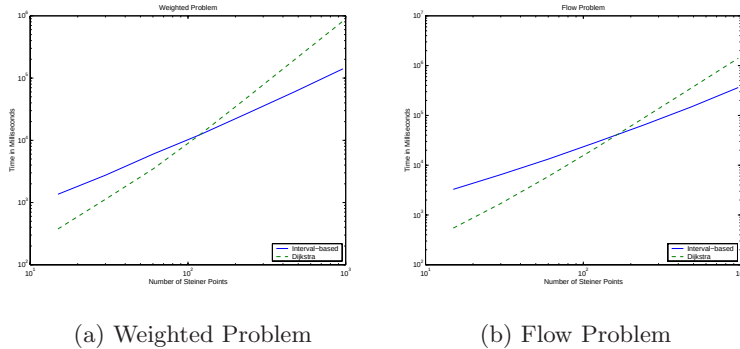


Fig. 4. Experimental Results

References

1. L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack. An ϵ -approximation algorithm for weighted shortest paths on polyhedral surfaces. *Lecture Notes in Computer Science*, 1432:11–22, 1998. 160, 161, 163
2. L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Approximation algorithms for geometric shortest path problems. In *ACM Symposium on Theory of Computing*, 2000. 160, 161, 163
3. M. Lanthier, A. Maheshwari, and J. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *Proceedings of the 13th International Annual Symposium on Computational Geometry*, pages 274–283. 4–6 June 1997. 160, 161
4. C. Mata and J. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions. In *Proceedings of the 13th International Annual Symposium on Computational Geometry*, pages 264–273. June 4–6 1997. 160
5. J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem. In *Proceedings of the 3rd International Annual Symposium on Computational Geometry*, pages 30–38. June 8–10 1987. 160
6. J. Reif and Z. Sun. An efficient approximation algorithm for weighted region shortest path problem. In *Proceedings of the 4th Workshop on Algorithmic Foundations of Robotics*, Mar. 16–18 2000. 160, 161
7. J. Reif and Z. Sun. Movement planning in the presence of flows. *Lecture Notes in Computer Science*, 2125:450–461, 2001. 160, 165

Approximation of Minimum Triangulation for Polyhedron with Bounded Degrees

Francis Y. L. Chin ^{*} and Stanley P. Y. Fung

Department of Computer Science and Information Systems,
The University of Hong Kong, Hong Kong.
{chin,pyfung}@csis.hku.hk

Abstract. Finding minimum triangulations of convex polyhedra is NP-hard. The best approximation algorithms only give a ratio 2 for this problem, and for combinatorial algorithms it is shown to be the best possible asymptotically. In this paper we improve the approximation ratio of finding minimum triangulations for some special classes of 3-dimensional convex polyhedra. (1) For polyhedra without 3-cycles and degree-4 vertices we achieve a tight approximation ratio $3/2$. (2) For polyhedra with vertices of degree-5 or above, we achieve an upper bound $2 - \frac{1}{12}$ on the approximation ratio. (3) For polyhedra with n vertices and vertex degrees bounded by a constant Δ we achieve an asymptotic tight ratio $2 - \Omega(1/\Delta) - \Omega(1/n)$.

1 Introduction

A *triangulation* of a d -dimensional polyhedron is its subdivision into a set of simplices, such that the simplices do not overlap and intersect only at common faces. We are interested in three-dimensional polyhedron triangulations (also called *tetrahedralizations*), which have important applications in computer graphics, finite element analysis, computer-aided design, etc. as well as having fundamental theoretical significance. In particular, we want to find triangulations consisting of a small number of tetrahedra.

The problem of polyhedron triangulation has been studied extensively. Convex polyhedra can always be triangulated, but different triangulations may contain different numbers of tetrahedra, i.e., the *size* of triangulations can be different. It is shown in [2] that finding a minimum triangulation, i.e., a triangulation with the minimum possible size, is NP-hard. There are several algorithms to triangulate a polyhedron, but not specifically addressing the problem of minimum triangulation. For example, the simple *pulling* heuristic [7], which picks a vertex and connects it to all other non-adjacent faces of the polyhedron, gives an approximation ratio 2 for finding minimum triangulations. Though simple, no better triangulation algorithms were known for a long time.

In [3] a new triangulation algorithm is given, by making use of the properties of 3-cycles. A *3-cycle* is a cycle of length three on the surface graph of a polyhedron such that both sides contain some other vertices (i.e. the triangular faces of

[□] This work is supported by RGC grant HKU 7019/00E.

the polyhedron are not regarded as 3-cycles). A 3-cycle separates a polyhedron into two parts. The idea of the algorithm (which we call **CutPull** in this paper) is very simple: we partition the polyhedron along all the 3-cycles into subpolyhedra, each is free of 3-cycles. We then apply the pulling heuristic to each resulting subpolyhedra. It was shown that such an algorithm gives an approximation ratio of $2 - \Omega(\frac{1}{\sqrt{n}})$ where n is the number of vertices of the polyhedron.

Although the above bound seems to be a slight improvement only, it was proved in the same paper that this approximation ratio is the best possible, for algorithms that only consider the combinatorial structure of the polyhedra. This lower bound is proved by utilizing a property of vertex-edge chain structures (VECSs), first introduced in [2]. A VECS of size m consists of the vertices $(a, b, q_0, q_1, \dots, q_{m+1})$, forming the set of triangular faces $\{aq_iq_{i+1}, bq_iq_{i+1} (0 \leq i \leq m)\}$ (Fig. 1(a)). It consists of a chain of degree-4 vertices. An important property of the VECS is [3]: if a polyhedron contains a VECS as a substructure, and the interior edge ab (called the *main diagonal*) is not present in a triangulation of the polyhedron, then in this triangulation at least $2m$ tetrahedra are ‘incident’ to the VECS. On the other hand, if ab is present, $m + 1$ incident tetrahedra may be sufficient for the triangulation.

In view of these results, the following question is raised in that paper: can the approximation ratio be improved when the maximum degree is constant? Another interesting question is that, are there special types of polyhedra that have optimal triangulations or with better approximation ratios using **CutPull**? In this paper we give some results about these problems.

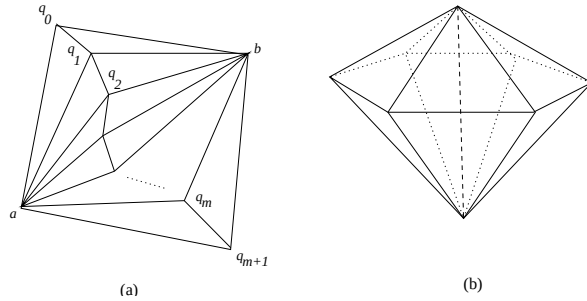


Fig. 1. (a) A VECS of size m . (b) A bipyramid with $n - 2$ vertices in the middle chain; here $n = 8$

The rest of this paper is organized as follows:

- In Sect. 2, we give new bounds on the relationship between the size of minimum triangulation, the maximum vertex degree, and the number of 3-cycles of a polyhedron, improving our previous results.
- In Sect. 3, we show that when a polyhedron has no 3-cycles and no degree-4 vertices, **CutPull** gives an improved approximation ratio, $3/2$ instead of 2 ,

and this bound is tight. For polyhedra with vertices of degree-5 or above, an upper bound $2 - \frac{1}{12}$ on the approximation ratio is proved.

- In Sect. 4, we give an improved analysis of the **CutPull** algorithm for polyhedra having vertex degrees bounded by a constant Δ . The analysis gives an asymptotically tight approximation ratio which is better than that of the general-degree case. In particular, the ratio is $12/7$ for $\Delta = 6$ and $7/4$ for $\Delta = 7$.

2 Preliminaries

Throughout this paper, let P be a convex polyhedron with n vertices, Δ the maximum vertex degree, and k the number of 3-cycles. We only consider polyhedra with vertices in *general position*, i.e., no four vertices are coplanar. Let also t be the size of minimum triangulation of P , and e_m the number of interior edges in a minimum triangulation of P . It is known that $t = e_m + n - 3$ [1]. It is also shown in [3] that e_m is related to Δ under the restriction that the polyhedron has no 3-cycles, by the formula $2e_m(\Delta + 1) \geq n$ which is tight to within a constant multiplicative factor. In this section we first give an alternative but very simple proof to tighten the inequality by almost a factor of 2 (the constant-factor improvement is important when we come to Sect. 4), then extend it to the case with 3-cycles.

Lemma 1. *For a polyhedron P with no 3-cycles and $n > 4$, $e_m\Delta \geq n - 2$, and this is tight.*

Proof. Consider a face $v_0v_1v_2$ in P . We claim that each face must be incident by at least one interior edge. Assume this is not so. Then there is a face $v_0v_1v_2$ of P that has no incident interior edges. It is in some tetrahedron with fourth vertex v_3 , and v_0v_3, v_1v_3, v_2v_3 have to be surface edges of P . Therefore the three triangles $v_0v_1v_3, v_1v_2v_3, v_2v_0v_3$ are either 3-cycles or faces. But 3-cycles are forbidden. If all three triangles are faces, then P is simply a tetrahedron with $n = 4$. Therefore our claim holds. Since there are $2n - 4$ faces, there are at least $2n - 4$ interior edges, but each is counted at most 2Δ times since each of the endpoints can be incident to at most Δ faces. Thus $e_m(2\Delta) \geq 2n - 4$, i.e. $e_m\Delta \geq n - 2$. This bound can be achieved by considering a bipyramid [9] (Fig. 1(b)), in which $e_m = 1, \Delta = n - 2$. $\square$

We can generalize Lemma 1 to polyhedra having k 3-cycles:

Lemma 2. *For a polyhedron with k 3-cycles and $n > 4$, $e_m\Delta \geq n - 2 - 3k$.*

Proof. As in Lemma 1, for each of the $2n - 4$ faces, there should be at least one incident interior edge, unless, among the three bounding edges of the face, at least one is on a 3-cycle. Each 3-cycle can share an edge with at most six faces (on both sides of the three edges). Thus there remain at least $2n - 4 - 6k$ faces having incident interior edges. With the same argument as in Lemma 1, $e_m(2\Delta) \geq 2n - 4 - 6k$, and the result follows. $\square$

We next prove a lemma relating the size of triangulations produced by **CutPull** and the number of 3-cycles of a polyhedron.

Lemma 3. *For a polyhedron with k 3-cycles, the **CutPull** algorithm produces triangulation of size at most $2n - 7 - k$.*

Proof. We prove this by induction on k . When $k = 0$, **CutPull** reduces to pulling, which gives $2n - 4 - \Delta \leq 2n - 7$ tetrahedra, so the claim holds. When $k > 0$, **CutPull** picks a 3-cycle, partitions the polyhedron into two subpolyhedra, with n_1 and n_2 vertices respectively, and having k_1 and k_2 3-cycles respectively. We have $n = n_1 + n_2 - 3, 0 \leq k_1 < k, 0 \leq k_2 < k, k = k_1 + k_2 + 1$. By induction assumption on the two subpolyhedra, their triangulations have sizes at most $2n_1 - 7 - k_1$ and $2n_2 - 7 - k_2$ respectively. So the size of triangulation is at most $(2n_1 - 7 - k_1) + (2n_2 - 7 - k_2) = 2(n + 3) - 14 - (k_1 + k_2) = 2n - 7 - (k_1 + k_2 + 1) = 2n - 7 - k$. Thus the claim holds. $\square$

3 Analysis for a Special Class of Polyhedra

From the results in [2] and [3], it can be seen that the major problems in finding minimum triangulations appear in 3-cycles and VECSSs. In this section we first analyze the special case in which the polyhedra concerned have no 3-cycles and no degree-4 vertices (thus no VECSSs). Note that the non-existence of 3-cycles implies that there are no degree-3 vertices, and thus all vertices have degrees at least five. We show that in this case the approximation ratio of the **CutPull** algorithm is at most $3/2$, better than the general case ratio $2 - \Omega(\frac{1}{\sqrt{n}})$. Moreover this is tight: we construct polyhedra having an approximation ratio no better than $3/2 - \epsilon$ using **CutPull** for any $\epsilon > 0$. We then consider the case when 3-cycles are present.

Empirically, it has been observed that 3-cycles are not very common in polyhedra, in particular those not induced by degree-3 vertices (every degree-3 vertex induces a 3-cycle); but polyhedra usually have some degree-3 and degree-4 vertices. However our results still have the following significance:

- (i) as far as we know this is one of the very few classes of polyhedra that is known to have approximation ratio $2 - \epsilon$ for constant $\epsilon > 0$. For example, ‘stacked polyhedra’ [7] can be triangulated optimally in linear time, or the ‘ k -opt polyhedra’ [8].
- (ii) the existence of 3-cycles and degree-4 vertices can be checked in linear time, in contrast to k -opt polyhedra where no algorithm is known to check whether a polyhedron is k -opt.
- (iii) they may arise as intermediate polyhedra in the processing of other triangulation algorithms, e.g. peeling [6]. If there are few degree-4 vertices, we might peel those degree-4 vertices first and may be left behind a polyhedron with degrees at least five (but note that new degree-4 vertices may appear in peeling).

- (iv) Certain classes of polyhedra, such as prisms, antiprisms, etc. [5] have no 3-cycles and degrees at least five (provided that the coplanar points are perturbed so that the faces are suitably triangulated, and if necessary with simple modification/replication).

We classify all vertices of a polyhedron P into two types (w.r.t. a particular triangulation): a vertex is called *type-I* if some interior edge is directly incident to it. Otherwise it is called *type-II*. For any vertex v , we define the *neighborhood* $N(v)$ of v to be the set of vertices directly connected to v on the surface graph, i.e. $N(v) = \{u | (u, v) \in \text{surface edges of } P\}$. $N(v)$ forms a 3-dimensional polygon. Consider any triangulation of the polygon $N(v)$. (Note that this is slightly different from the definition of ‘dome’ or ‘cap’ [4] [3] in that a triangulation of $N(v)$ may not yield a convex patch of triangular faces.) Triangles with two edges on the polygon $N(v)$ are called ‘*ear*’ triangles, and all others are called ‘*internal*’ triangles (Fig. 2).

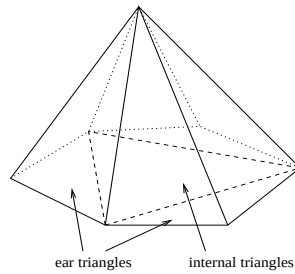


Fig. 2. Ear and internal triangles

We present some observations about type-II vertices in the lemma below, which we shall skip the easy proof:

Lemma 4. Suppose v is a type-II vertex of degree d in a polyhedron P with respect to a triangulation.

- (i) All tetrahedra incident to v form a triangulation of the region bounded by the 3-D polygon $N(v)$ and the faces of P around v . There are $d - 2$ tetrahedra in this part of the triangulation. Their bases triangulate the polygon $N(v)$.
- (ii) For any triangulation of $N(v)$, if $d \geq 5$ and v is not lying on any 3-cycles, there is at least one vertex in $N(v)$ having two or more incident interior edges. The triangulation of $N(v)$ consists of at least two ‘ear’ triangles and at least one ‘internal’ triangle.

Lemma 5. For a polyhedron P having no 3-cycles and with vertex degrees at least five, there are at least $4n/3 - 3$ tetrahedra in any triangulation of P .

Proof. Suppose we have n_1 type-I vertices and n_2 type-II vertices, and $n_1 + n_2 = n$. We consider the two cases, each using a different method to bound the size of triangulation:

Case 1: $n_1 \geq \alpha n$, where α ($0 \leq \alpha \leq 1$) is a constant we will specify later. We want to count the number of interior edge endpoints incident to these vertices (each interior edge having two endpoints). By definition, for each type-I vertex there is at least one interior edge endpoint incident to it. This gives n_1 edge endpoints. In addition, for each of the n_2 type-II vertices, there is at least one type-I vertex in the neighborhood that has two or more edge endpoints incident to it (Lemma 4). But the previous step did not count the extra endpoints (only one endpoint was counted for each type-I vertex). Thus there are at least n_2 additional edge endpoints, if all of them are distinct. Note that these must be interior edges, otherwise there is a 3-cycle. It can be shown that at most two type-II vertices share such an additional endpoint; the worst case is as in Fig. 3 where two type-II vertices (v_1 and v_9) sharing a type-I vertex (v_3) that only has two interior edge endpoints. Thus at least $n_2/2$ edge endpoints are added. Since each interior edge has two endpoints to be counted,

$$e_m \geq \frac{1}{2}(n_1 + \frac{n_2}{2}) = \frac{n_1}{2} + \frac{n_2}{4} = \frac{n + n_1}{4} \geq \frac{(1 + \alpha)n}{4}$$

Thus

$$t = e_m + n - 3 \geq \frac{(1 + \alpha)n}{4} + n - 3 = \frac{(5 + \alpha)n}{4} - 3$$

Case 2: $n_1 < \alpha n$. Then $n_2 > (1 - \alpha)n$. For each type-II vertex v , all tetrahedra incident to it constitute a triangulation of $N(v)$ (Lemma 4)(Fig. 2). Consider any triangulation of the 3-D polygon $N(v)$, with each triangle corresponding to a tetrahedron having v as a vertex. We count the number of tetrahedra incident to the type-II $N(v)$'s. All 'internal' tetrahedra of a type-II vertex v will not be counted by other type-II vertices (since the other three vertices of the tetrahedron are type-I). The 'ear' tetrahedra may be counted twice. For example in Fig. 3 tetrahedra $v_1 v_2 v_3 v_4$ and $v_1 v_3 v_5 v_6$ are 'ear' tetrahedra of a type-II vertex v_1 , but the tetrahedra $v_1 v_2 v_3 v_4$ (resp. $v_1 v_3 v_5 v_6$) may also be counted by v_2 (resp. v_6) if v_2 (resp. v_6) are type-II. It cannot be counted more than twice since the other two vertices of the tetrahedron are type-I. Since $d \geq 5$, there is at least one 'internal' tetrahedron and at least two 'ear' tetrahedra, giving a total of at least two tetrahedra (each 'ear' counted as 0.5 for this vertex to avoid double counting). Thus the total number of tetrahedra incident to these type-II vertices is at least $2n_2 \geq 2(1 - \alpha)n > 2(1 - \alpha)n - 3$.

Considering both cases, the number of tetrahedra is at least $\min(\frac{(5 + \alpha)n}{4} - 3, 2(1 - \alpha)n - 3)$. It is easy to show that this is maximized when $\alpha = 1/3$, thus $t \geq 4n/3 - 3$. $\square$

The above bound is tight (up to a constant additive factor) as shown below:

Lemma 6. *There exist polyhedra, without 3-cycles and degree-4 vertices, whose sizes of minimum triangulation are at most $4n/3 + 8/3$.*

Proof. The construction proceeds as follows.

Step 1. First consider the 9 vertices $v_1, \dots, v_9$ as in Fig. 3; $N(v_1) = \{v_2, v_3, v_4, v_5, v_6\}$, $N(v_9) = \{v_3, v_4, v_5, v_7, v_8\}$. $N(v_1)$ and $N(v_9)$ share the triangle $v_3v_4v_5$. This may be regarded as a (non-convex) polyhedron with v_3v_4 , v_3v_5 being the nonconvex edges. They will become interior edges when we further construct the polyhedron. We now have a (partial) triangulation consisting of six tetrahedra $v_1v_2v_3v_4$, $v_1v_3v_4v_5$, $v_1v_3v_5v_6$, $v_9v_7v_3v_4$, $v_9v_3v_4v_5$ and $v_9v_3v_5v_8$.

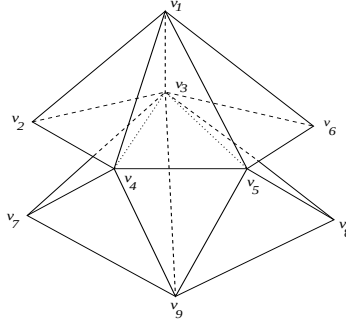


Fig. 3. Step 1 of the construction

Step 2. We patch each of the two non-convex ‘gaps’ by the following structure. For the right-side gap, we add three vertices u_1 , u_2 and u_3 and form the polyhedron as in Fig. 4, producing a new non-convex gap. The u_i ’s are placed carefully so that convexity is maintained (except at the new non-convex gap). The new region can be triangulated by the tetrahedra $v_5v_6u_1u_2$, $v_3v_5v_6u_2$, $u_2u_3v_5v_8$ and $v_3v_5u_2v_8$. The left-side gap is treated similarly. Each application of step 2 replaces the two old gaps by two new ones, using six more vertices and eight more tetrahedra. Note that the new gap and the old gap share a vertex. Step 2 is recursively applied to the new gaps. This step can be applied any number of times. We can keep the maximum degree of the polyhedron constant by alternating which vertex the original gap touches with the new gap.

Step 3. Finally, we patch each of the two gaps by the structure shown in Fig. 5. This structure has 12 vertices (8 of them are new vertices when patched), all are of degree at least five, and need at most $2n - 4 - \Delta = 2(12) - 9 = 15$ additional tetrahedra to triangulate (by pulling). The structure is patched so that the final polyhedron is convex.

It can be seen that this convex polyhedron has no 3-cycles, and all vertices have degrees at least five.

Note that we constructed a triangulation of this polyhedron along with the above steps. This may not be a minimum triangulation, but a minimum triangulation must have same or smaller size. If Step 2 is applied p times, then $n = 9 + 6p + 16$ and $t \leq 6 + 8p + 30$, giving $t \leq 4n/3 + 8/3$. $\square$

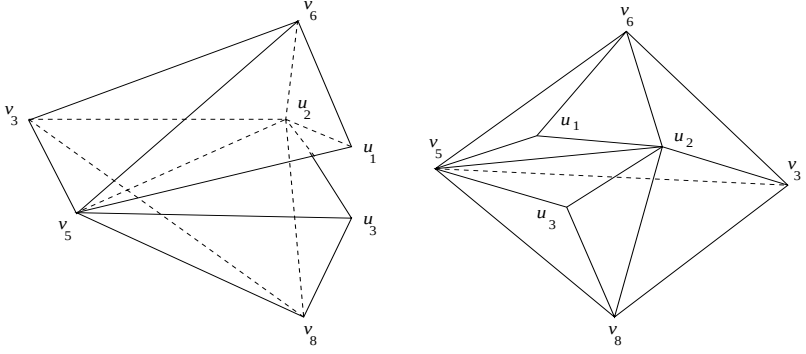


Fig. 4. Step 2, showing 3-D view (left) and view from right (right)

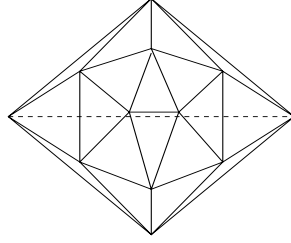


Fig. 5. Final patch for Step 3. The two triangles at back (as indicated by the dashed line) are to be attached to the gap

The tight bound on the size of triangulations gives a tight bound on the approximation ratio of **CutPull**:

Theorem 1. *The approximation ratio of **CutPull** algorithm for polyhedron without 3-cycles and all vertices having degree at least five is at most $3/2$, and this is tight.*

Proof. The approximation ratio follows from Lemmas 3 and 5: $r \leq \frac{2n-7}{4n/3-3} < 3/2$. That the bound is tight follows from the constructed polyhedron in Lemma 6, having constant degrees, no 3-cycles, and $t \leq 4n/3 + 8/3$. Thus for those polyhedra, **CutPull** gives $r \geq \frac{2n-4-\Delta}{4n/3+8/3} = 3/2 - \epsilon$ where $\epsilon = \Theta(1/n)$ tends to 0 as n tends to infinity. $\square$

With the presence of 3-cycles (but still without degree-3 and degree-4 vertices), the argument in Lemma 5 works for vertices not lying on any 3-cycles. Thus if there are n' vertices not lying on 3-cycles, we have $t \geq 4n'/3 - 3$. Note that we have $n' \geq n - 3k$, thus with Lemma 3 we have:

Theorem 2. *For polyhedra with k 3-cycles and having degrees at least five, the approximation ratio $r \leq \frac{2n-7-k}{\max(n-3, 4n/3-4k-3)}$. In particular, it can be shown to be less than $2 - \frac{1}{12} = 1.9166\dots$ for any k .*

4 Analysis for Bounded-Degree Polyhedra

In this section, we turn our attention to another class of convex polyhedra in which the vertex degrees are bounded by a constant. This occurs frequently for randomly-generated polyhedra. We show that in this case the **CutPull** algorithm can be applied with improved approximation ratio, and the ratio is tight up to combinatorial considerations.

4.1 Upper Bound

We first prove the upper bound, using Lemmas 2 and 3:

Theorem 3. *When the maximum degree Δ of a convex polyhedron is bounded by a constant, the **CutPull** algorithm gives an approximation ratio $2 - \frac{2}{\Delta+1} - \Omega(\frac{1}{n})$.*

Proof. Suppose there are k 3-cycles in the polyhedron. By Lemma 3 we have

$$r \leq \frac{2n-7-k}{n-3+e_m} < \frac{2n-1-k}{n+e_m}$$

By Lemma 2, $e_m\Delta + 3k \geq n-2$, thus $e_m\Delta + (\frac{\Delta+1}{2})k \geq n-2$, and

$$k \geq \frac{2}{\Delta+1}n - \frac{4}{\Delta+1} - \frac{2\Delta}{\Delta+1}e_m$$

Substituting into the previous inequality gives

$$\begin{aligned} r &\leq \frac{2n-1 - \frac{2}{\Delta+1}n + \frac{4}{\Delta+1} + \frac{2\Delta}{\Delta+1}e_m}{n+e_m} = \frac{(2 - \frac{2}{\Delta+1})n + (2 - \frac{2}{\Delta+1})e_m - (1 - \frac{4}{\Delta+1})}{n+e_m} \\ &= 2 - \frac{2}{\Delta+1} - \frac{1 - \frac{4}{\Delta+1}}{n+e_m} = 2 - \frac{2}{\Delta+1} - \Omega(\frac{1}{n}) \end{aligned}$$

since $e_m = O(n)$. □

For general (non-constant) Δ , we can actually prove that the approximation ratio is bounded by $r \leq 2 - \Omega(\frac{1}{\Delta}) - \Omega(\frac{\Delta}{n})$. The worst case is when $\Delta = \Theta(\sqrt{n})$ in which the the bound reduces to the $2 - \Omega(\frac{1}{\sqrt{n}})$ given in [3].

4.2 Lower Bound

It is proved in [3] that no combinatorial algorithms can give an approximation ratio better than $2 - O(\frac{1}{\sqrt{n}})$ for the minimum triangulation problem. The proof is based on constructing two polyhedra P1 and P2 with the same combinatorial structure but having different sizes in their minimum triangulations. In this subsection we prove similar results for the constant degree case. This shows that our upper bound in the previous subsection is asymptotically tight when only combinatorial information is considered.

For convenience we first review very briefly the construction for the general-degree case. First, a set of m VECs each of size m are placed as in Fig. 6(a). Wedge W_i has vertices (a_i, b_i, c_i, d_i) with $a_i b_i$ being the main diagonal. All faces $a_i c_i d_i$ lie on the vertical plane $y = -1$ while all faces $b_i c_i d_i$ lie on the horizontal plane $z = 1$. All main diagonals pass through the origin. Vertices $q_i^1, \dots, q_i^m$ are added between c_i and d_i for each wedge. The a_i 's form a convex chain w.r.t. $(0, -1, -\infty)$, and the b_i 's form a convex chain w.r.t. $(\infty, 0, 1)$. We have $n = m^2 + 4m$.

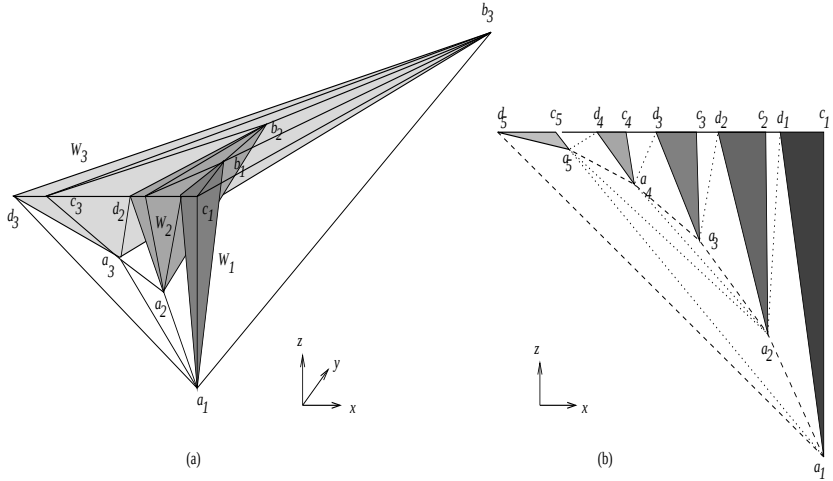


Fig. 6. (a) Construction of P1 and P2, showing 3 wedges. q_i 's are not shown. (b) The wedges with $m = 5$, showing the zig-zag paths on the vertical plane

Second, notice that all vertices lie on two planes, violating the general position assumption. We remove this degeneracy by perturbing the vertices slightly, so that the polyhedron has the following set of edges:

$$\begin{aligned} & a_k c_k, a_k d_k, b_k c_k, b_k d_k (1 \leq k \leq m); \\ & d_k a_{k+1}, d_k c_{k+1}, b_k c_{k+1}, a_k a_{k+1}, b_k b_{k+1} (1 \leq k \leq m-1); \\ & a_1 a_k, b_1 b_k (3 \leq k \leq m); \end{aligned}$$

$$\begin{aligned}
& q_k^i q_k^{i+1}, q_k^1 c_k, q_k^m d_k (1 \leq i \leq m-1, 1 \leq k \leq m); \\
& q_k^i a_k, q_k^i b_k (1 \leq i \leq m, 1 \leq k \leq m); \\
& c_1 b_m, a_1 b_m, a_1 d_m.
\end{aligned}$$

Now the main diagonals all intersect at the origin. In the third step for P1, we ‘push’ the wedges towards each other slightly so that all wedges intersect each other. For P2, we move the wedges slightly apart so that they do not intersect. In this way, P1 will have a large size of triangulation because the wedges are ‘interlocked’ (i.e. penetrating each other), while P2 can have a small triangulation, although the two have the same combinatorial structure.

Our construction for the constant-degree case is very similar to the construction above. The two polyhedra P1 and P2 consists of m wedges each, with each wedge being a VECS of size s , where s is constant. Thus the number of vertices $n = m(s + 4)$. The placement of wedges are almost identical to that as described above. There is only one thing we need to change. In the second step, we perturb the vertices into general position, resulting in edges $a_1 a_k$ and $b_1 b_k$ ($1 \leq k \leq m$). Since we fixed s constant, m is no longer constant and the above set of edges causes the degrees of a_1 and b_1 to be non-constant. To cope for this, our constant-degree construction perturbs the vertices in another way, so that zig-zags paths, e.g. $a_1 a_n a_2 a_{n-1} \dots, b_1 b_n b_2 b_{n-1} \dots$ appears (Fig. 6(b)). Such triangulation is always possible by applying sufficiently small perturbations to the vertices. The maximum degree $\Delta = s + 7$ (attained at e.g. a_2 in Fig. 6(b)) is thus constant.

Theorem 4. *When the maximum degree Δ of a convex polyhedron is bounded by a constant, any algorithm that only considers the combinatorial structure cannot give an approximation ratio better than $2 - O(\frac{1}{\Delta+1}) - O(\frac{1}{n})$.*

Proof. We first show that any minimum triangulation of P1 has at least $(\frac{2\Delta-14}{\Delta-3})n - \Delta$ tetrahedra, while any minimum triangulation of P2 has at most $(\frac{\Delta+1}{\Delta-3})n$ tetrahedra.

As discussed in the introduction, the wedges have the property that they admit triangulation of size either at most $s + 1$ or at least $2s$, depending on the presence of their ‘main diagonal’ in the triangulation. For P1, at most one main diagonal of these m wedges can be present in any triangulation. Thus

$$t_{P1} \geq (m-1)(2s) + (s+1) = (\frac{n}{\Delta-3} - 1)(2\Delta-14) + (\Delta-6) > (\frac{2\Delta-14}{\Delta-3})n - \Delta.$$

For P2, each wedge can be triangulated into $s + 1$ tetrahedra using their main diagonals. Removing these wedges leaves a non-convex region. This can be triangulated into $4(m-1) + 3(m-2) + 2 = 7m - 8$ tetrahedra, using the ‘shielding’ argument same as that in [3]; due to space limitation we do not repeat it here. Note that the a_i ’s and b_i ’s have to be ‘zig-zagged’ in a matching manner for the proof to work. So

$$t_{P2} \leq m(s+1) + 7m - 8 = (\frac{n}{s+4})(s+1+7) - 8 < (\frac{\Delta+1}{\Delta-3})n$$

A combinatorial algorithm cannot distinguish P1 and P2, and always has to give the triangulation of larger size. With the above bounds, we thus have

$$r \geq \frac{(\frac{2\Delta-14}{\Delta-3})n - \Delta}{(\frac{\Delta+1}{\Delta-3})n} = \frac{2\Delta-14}{\Delta+1} - \frac{\Delta(\Delta-3)}{n(\Delta+1)} = 2 - O(\frac{1}{\Delta+1}) - O(\frac{1}{n})$$

□

5 Conclusion

We gave improved approximation ratios for the minimum polyhedron triangulation problem for two special classes of polyhedra: one having no 3-cycles and no degree-4 vertices, and one with constant degrees. For the case without 3-cycles and degree-4 vertices, our algorithm gives a ratio of $3/2$. This seems to be a rather restricted class of polyhedra; can it be optimally triangulated in polynomial time? Can we identify the (more restricted?) class of polyhedra which our algorithm will give the optimal triangulation? Stacked polyhedra are one known type. Can we identify classes of polyhedra that can be triangulated optimally or near-optimally in polynomial time, using perhaps other algorithms? The results may also be generalized to polyhedra having few (but nonzero) degree-4 vertices.

For the constant degree case, we get an asymptotically tight approximation ratio $r = 2 - \Omega(\frac{1}{\Delta}) - \Omega(\frac{1}{n})$, the lower bound being established if only combinatorial structure is considered. It is actually not known whether the constant-degree case is NP-hard (like the general-degree case), and what happens when non-combinatorial information is considered.

Acknowledgements

We wish to thank C. A. Wang for raising the problem of whether polyhedra with degrees at least five have better triangulations, and for many helpful discussions.

References

1. Below A., Brehm U., De Loera J., and Richter-Gebert J., Minimal Simplicial Dissections and Triangulations of Convex 3-Polytopes, *Discrete and Computational Geometry* 24 (2000), 35–48. [174](#)
2. Below A., De Loera J., and Richter-Gebert J., Finding Minimal Triangulations of Convex 3-Polytopes is NP-Hard, *Proceedings of 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (2000), 65–66. [172](#), [173](#), [175](#)
3. Chin F. Y. L., Fung S. P. Y. and Wang C. A., Approximation for Minimum Triangulations of Simplicial Convex 3-Polytopes, *Discrete and Computational Geometry*, to appear. A preliminary version appeared in *Proceedings of 12th Annual ACM-SIAM Symposium on Discrete Algorithms* (2001), 128–137. [172](#), [173](#), [174](#), [175](#), [176](#), [180](#), [181](#), [182](#)

4. Chazelle B. and Palios L., Triangulating a Non-convex Polytope, *Discrete and Computational Geometry* 5 (1990), 505–526. 176
5. Cromwell P. R., *Polyhedra*, Cambridge University Press, 1997. 176
6. Edelsbrunner H., Preparata F. P. and West D. B., Tetrahedrizing Point Sets in Three Dimensions, *Journal of Symbolic Computation* 10 (1990), 335–347. 175
7. Lee C. W., Subdivision and Triangulations of Polytopes, in *Handbook of Discrete and Computational Geometry* (Goodman and O’Rourke eds.), CRC Press, 1997. 172, 175
8. Wang C. A. and Yang B., Optimal Tetrahedralizations of Some Convex Polyhedra, *16th European Workshop on Computational Geometry*, 2000. 175
9. Wang C. A. and Yang B., Minimal Tetrahedralizations of a Class of Polyhedra, *Proceedings of 12th Canadian Conference on Computational Geometry* (2000), 81–90. 174

Tree-Approximations for the Weighted Cost-Distance Problem (Extended Abstract)

Christian Schindelhauer¹ and Birgitta Weber²

¹ Department. of Mathematics and Computer Science and Heinz-Nixdorf-Institute,
Paderborn University
33095 Paderborn, Germany
`schindel@upb.de`

² Institute of Theoretical Computer Science, ETH Zentrum
CH-8092 Zürich, Switzerland
`weberb@inf.ethz.ch`

Abstract. We generalize the Cost-Distance problem: Given a set of n sites in k -dimensional Euclidean space and a weighting over pairs of sites, construct a network that minimizes the cost (i.e. weight) of the network and the weighted distances between all pairs of sites. It turns out that the optimal solution can contain Steiner points as well as cycles. Furthermore, there are instances where crossings optimize the network. We then investigate how trees can approximate the weighted Cost-Distance problem. We show that for any given set of n sites and a non-negative weighting of pairs, provided the sum of the weights is polynomial, one can construct in polynomial time a tree that approximates the optimal network within a factor of $O(\log n)$. Finally, we show that better approximation rates are not possible for trees. We prove this by giving a counter-example. Thus, we show that for this instance that every tree solution differs from the optimal network by a factor $\Omega(\log n)$.

1 Introduction

1.1 Problem and Motivation

Given n terminal points in the Euclidean space we investigate the problem of constructing a network with small cost and short distances. This research is motivated by a number of practical problems arising in network design for real traffic, as well as traffic in communication networks. It is often observed that the cost of networks can be described by a component depending only on the size of the network and by a component growing with the demand of certain connections. Consider a street network: if one minimizes only the network size to cover cost for building and maintenance, the connections between terminals can grow by the diameter of the network. Then, additional costs caused by detours outweigh the fixed costs.

In practice network designers model the demand in a network by a so-called *origin-destination matrix* $w(u, v)$. For sites u, v it describes the traffic starting at u with destination v . We model the cost of the network for each edge by a linear function $c_1 \|e\|_2 + c_2 \sum_{(u,v) \in P(e)} w(u, v) \|e\|_2$ for $c_1, c_2 > 0$, where $\|e\|_2$ denotes the Euclidean length of the edge and $P(e)$ is the set of all pairs (u, v) such that the shortest path between u and v contains e . By summing over all edges we define the *Weighted Cost-Distance* (WCD) of a network N and a weighting w :

$$\text{WCD}_w(N) := \sum_{e \in E(N)} \left(c_1 \|e\|_2 + c_2 \sum_{(u,v) \in P(e)} w(u, v) \|e\|_2 \right). \quad (1)$$

So, for a pair u, v with large weight $w(u, v)$ (frequent traffic) a detour between u and v implies higher costs than between pairs with smaller weight.

There is a trade-off between cost and weighted distance. If we choose $c_2 = 0$ we face the intensively studied *minimum network problem*. If we choose $c_1 = 0$, the optimal solution is a complete network. For sites in general position and positive weights. As we scale the parameter c_1/c_2 from 0 to ∞ , we see a gradual transformation from the Steiner tree to the complete network. We are interested in the structure of the intermediate states.

For simplicity we replace the above definition by the following. Since we only consider $c_2 > 0$, we can set $c_1 = c_2 = 1$ if we simultaneously modify the weighting by $w'(u, v) = \frac{c_1}{c_2} w(u, v)$. This results in the following equivalent version of the Weighted Cost-Distance:

$$\text{WCD}_w(N) := \sum_{e \in E(N)} c(e) + \sum_{u, v \in V(N)} w(u, v) L_N(u, v), \quad (2)$$

where $c(e)$ denotes the cost of an edge and $L_N(u, v)$ the length of the shortest path from u to v in the network N . We use this notation throughout this paper. The corresponding optimization problem is defined as follows.

Definition 1. Let $L_G(u, v)$ denote the minimum length of a path of vertex u to v in graph G .

- Weighted Cost-Distance Network problem (CDN): Given a set of sites V in Euclidean space and a weighting $w : V \times V \mapsto \mathbb{R}^+$, find a network $N = (V, E)$ that optimizes the Cost-Distance $\text{WCD}_w(N)$ (according to equation (2)).
- Weighted Cost-Distance Tree problem (CDT): Given V and $w : V \times V \mapsto \mathbb{R}^+$, find a tree $T = (V, E)$ that optimizes the Cost-Distance $\text{WCD}_w(T)$.

In addition to the sites we allow the use of a non-terminal vertex set, if not explicitly stated otherwise.

1.2 Previous Work

If the weights are set to zero, and no restrictions for the non-terminals are given the Weighted Cost-Distance problem reduces to the *Euclidean Steiner*

Tree problem. It was shown to be NP-hard by Garey, Graham and Johnson [7]. However, in his groundbreaking paper Arora [1] showed that this problem admits a polynomial time approximation scheme.

In [8] the *Balanced Spanning Tree* problem was introduced. Here, the task is to find a tree which optimizes the term

$$\sum_{e \in E(T)} c(e) + \sum_{s \in V} L_T(s, r)$$

for a given root r under a metric c (not necessarily Euclidean). Non-terminal sites are not available.

The authors prove the existence of trees, where the dilation of all vertices' distances from the root is bounded by any $\alpha > 1$ and the trees cost is at most β times the cost of the minimum spanning tree, where $\beta = 1 + \frac{2}{\alpha-1}$. This leads to a constant polynomial time bounded approximation algorithm.

The Balanced Spanning Tree problem is a variant of the Weighted Cost-Distance Network problem, if we allow general metrics and exclude non-terminal vertices. The weighting is limited to $w(r, u) = 1$ and $w(u, v) = 0$ for $u, v \in V \setminus \{r\}$. For this problem in [8] it is shown that a tree is always part of the optimal solution and approximating networks can be pruned to trees. Hence, here the Cost-Distance Network problem reduces to the Cost-Distance Tree problem.

Meyerson et al. [10] generalize this problem by introducing a positive vertex weighting, and by allowing two different metrics for cost and distance: the length metric ℓ and the cost metric c . The Cost-Distance measure is given by

$$\sum_{e \in E(T)} c(e) + \sum_{s \in V} w(s) L_T(s, r)$$

for a root r . They present a polynomial time bounded randomized algorithm that approximates the problem within a factor of $O(\log n)$. Furthermore, they show that the optimal solution is always a tree.

A t -spanner is a connected partial graph of a given graph G such that for all vertices $u, v \in V(G)$ the corresponding shortest path in the t -spanner is at most t times longer than in G . There exist t -spanners in Euclidean space, whose sizes are bounded linearly by the size of the minimum spanning tree [2]. It turns out that these spanning networks already allow us to state constant factor approximation algorithms for the Weighted Cost-Distance Network problem.

Theorem 1 ([2]). *In k -dimensional Euclidean space, for any $t > 1$ there exists a t -spanner with size $O(c(MST))$, which can be computed in time $O(n \log n)$.*

This immediately implies that t -spanners allow constant factor approximation for the CDN-problem.

Corollary 1. *For Euclidean space the Weighted Cost-Distance Network problem can be approximated by a constant factor within time $O(n \log n)$.*

For the two-dimensional Euclidean space we can pin down the constant very accurately by using the result of [9].

Lemma 1. [9] *For $r > 0$, there exists a $(1 + \frac{1}{r}) \frac{2\pi}{3 \cos(\pi/6)}$ -spanner of the complete graph, whose size is at most $2r + 1$ times the costs of the minimal spanning tree.*

Optimizing the choice of t leads to the following result:

Theorem 2. *For the Euclidean plane there exists a polynomial time approximation of the Weighted Cost-Distance Network problem, where we do not allow non-terminal vertices, by a factor of $\frac{2\pi+3+\sqrt{4\pi^2+36\pi+9}}{3\sqrt{3}} \approx 4.23 \dots$*

A complete proof can be found in [11].

Using the results in [3] and [4] one can transfer the t -spanner result of [2] to arbitrary metrics. However the cost is increased by a logarithmic term. Such t -spanners give an approximative solution for CDN:

Corollary 2. *For metric costs and distances the Weighted Cost-Distance Network problem can be approximated in polynomial time within a factor of $O(\log n)$.*

1.3 The Optimal Network Is Not a Tree

For the minimum network problem it is known that introducing non-terminal vertices helps to reduce the network costs (i.e. size) by a constant factor. The optimal choice of such vertices are Steiner points.

Many properties are known for these Steiner networks. First of all minimum networks are trees. Further, in the plane Steiner points have degree three and the angle of neighbored edges is 120° . The number of these non-terminal points is bounded by $n - 2$.

A complete analysis of even small graphs shows that non-terminal sites also allow an improvement of a constant factor for the CDN-problem. Nevertheless, the angles between the adjacent edges may differ from 120° .

In contrast to the Cost-Distance Problems investigated so far, it turns out that the optimal solution is not a tree. We will prove in section 3 that a tree can differ by at least a factor of $\Omega(\log n)$ from the optimal network. Even more surprisingly, non-terminal (quasi-Steiner points) may be involved in cycles and there may be cycles that connect only quasi-Steiner points.

Another interesting observation is that the optimal network may include crossing edges, where the placement of a quasi-Steiner point onto the crossing point does not improve the solution. This reminds of the open problem whether optimal dilation trees contain crossings.

For a more detailed discussion of the topics, addressed in this introductory section we refer to [11]. Examples for crossing and quasi-Steiner points can be seen in figures 1, 2 and 3. In the following section we will prove that the optimal Cost-Distance network can be approximated by a tree within a factor of $O(\log n)$. Furthermore, there is a polynomial time bounded algorithm that computes such a tree, given the weighting and the sites in Euclidean k -dimensional space. In section 3 we prove the optimality of this approximation factor. We finally conclude these results and present some open problems for further research.

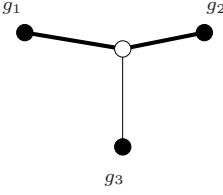


Fig. 1. The optimal WCD-network contains a quasi-Steiner point.
 $w(g_1, g_2) \gg w(g_1, g_3) \wedge$
 $w(g_1, g_2) \gg w(g_1, g_2)$

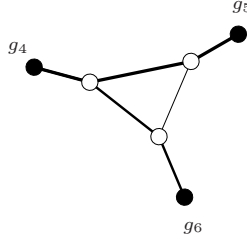


Fig. 2. The optimal WCD-network contains a cycle.
 $w(g_4, g_5) \approx w(g_4, g_6) \wedge$
 $w(g_4, g_5) > w(g_5, g_6)$

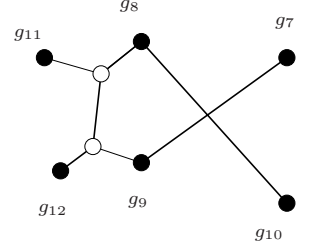


Fig. 3. An instance where a crossing is part of the optimal solution.
 $w(g_8, g_{10}) \approx w(g_7, g_9) \wedge$
 $w(g_8, g_{12}) > w(g_9, g_{11}) \wedge$
 $w(g_9, g_{10}) = w(g_8, g_7) = 0$

2 A Tree-Approximation by Factor of $O(\log n)$

Note that for k -dimensional Euclidean space the quality of the minimum networks differs from the minimum spanning tree only by a constant factor. For the Cost-Distance problem the situation is similar. Therefore we will not use any non-terminals in the following construction.

We use the notion of a *split tree* [2]. A *split tree* is a tree that stems from a hierarchical decomposition of a point set into k -dimensional rectangles of bounded aspect ratio, say in the range $[\frac{1}{3}, 3]$. We start with the smallest possible rectangle, $R_0 = R(V)$, including the point set V . Let r_0 be the root of the split tree. This rectangle R_0 is split into two smaller rectangles R_1 and R_2 . Let $V(R)$ be the subset of vertices in rectangle R . The split tree of R_1 is the split tree for the vertices $V(R_1)$, and similiary for R_2 and $V(R_2)$. These subtrees are connected to the root r_0 .

We will construct a *fair split tree* (FST) where each sub-tree with vertex set V' has a diameter of $O(kd(V'))$, where $d(V') := \max_{u,v \in V'} \|u, v\|_2$. Let $\ell(R)$ be the length of the longest edge of a rectangle R . We will use the following recursive construction given a rectangle R , a root $r \in V(R)$ and a weighting w such that for some $c > 1$: $W := \sum_{u,v} w(u, v) = O(n^c)$.

1. If $\ell(R) \leq \frac{d(V)}{n^c}$, then we choose an arbitrary vertex $r \in R$ and connect all vertices $V(R)$ directly to r .
2. Otherwise, we partition the rectangle R by a hyperplane orthogonal to an edge e with length $\ell(R_0)$. The distance between the hyperplane and the ends of the longest edge is at least $\frac{1}{3}\ell(R_0)$. The exact position depends on the weighting and will be described in the proof of Theorem 3.

The resulting two axis-parallel adjacent rectangles partitioning R are called R_1 and R_2 .

- (a) If r is in R_1 let $r_1 = r$ and take an arbitrary vertex $r_2 \in R_2$ and vice versa if $r \in V(R_1)$. Insert the edge $\{r_1, r_2\}$.

(b) Recursively, proceed with R_1, r_1 and R_2, r_2 .

Note that $d(V) \leq \ell(R_0)$ and observe that after k rounds the length of the longest edge is reduced by at most a factor of $\frac{2}{3}$. So there are only $O(k \log n)$ rounds until the size of the rectangles is bounded by $\frac{\ell(R_0)}{n^\epsilon}$. The length of every path in the resulting tree is bounded by $3k\ell(R_0)$: starting from the vertex of the path closest to the root, following the path downwards in both directions, the lengths of the edges $e_1, e_2, \dots$ and $e'_1, e'_2, \dots$ are upper bounded by $\|e_i\|_2, \|e'_i\|_2 \leq (\frac{2}{3})^{\lfloor i/k \rfloor} d(V)$.

Lemma 2. *Fair split trees have diameter $3kd(V)$ and weight $O(s(MST(V))k \log n)$.*

Proof. We apply the Lemma of [6,5] using the isolation property. If we add non-intersecting cylinders to all edges with radius $r/3$ and distance $r/3$ to the end points, then the cost of the corresponding network is linearly bounded by the cost of the MST. (The isolation property also holds if the cylinder is replaced by other geometric objects). Note that for the edges of each recursion step, we can attach such a cylinder to an edge such that the cylinder is completely in the corresponding rectangle. Since there are at most $O(k \log n)$ recursion steps this implies the claim. $\square$

We have not presented where we place the split. The following Lemma helps us to make a good selection.

Lemma 3. *Given rectangle R_0 and a weighting $w : V \times V \mapsto \mathbb{R}_0^+$. There exists partition of V into rectangles R_1 and R_2 with vertex sets V_1, V_2 such that*

$$\sum_{(u,v) \in V_1 \times V_2 \cup V_2 \times V_1} w(u,v) \leq \frac{3D}{\ell(R_0)}$$

where $D := \sum_{u,v \in V} w(u,v) \|u,v\|_2$.

Proof. Define $p := \left\lfloor \frac{\ell(R_0)}{3\Delta} \right\rfloor$ adjacent parallel rectangles R_i of thickness $\Delta := \frac{D}{W}$, where $W := \sum_{u,v \in V} w(u,v)$. These rectangles have distance of at least $\ell(R_0)/3$ to the left and right end of the longest edge of R_0 . We will partition between a pair R_i and R_{i+1} .

Next consider pair a pair of vertices u, v with $u \in R_i$ and $v \in R_j$. Then, we have $\|u, v\|_2 \geq \Delta \cdot |i - j|$. Measure v_i which is the weight of all connections crossing the right border between R_i and R_{i+1} :

$$v_i = \sum_{j \leq i < k} \sum_{u \in R_j} \sum_{v \in R_k} w(u, v) + w(v, u) .$$

Let $i = I(u)$ denote the index of the rectangle R_i with $u \in R_i$. Note that

$$\sum_i v_i \leq \sum_{u,v \in \bigcup_i R_i} w(u,v) |I(u) - I(v)| \leq \sum_{u,v} \frac{w(u,v) \|u,v\|_2}{\Delta} = W .$$

Hence, for at least one of the rectangles R_i we have $v_i \leq \frac{W}{p} \leq \frac{3C}{\ell(R_0)}$. $\square$

Of course, this split can be found in polynomial time if the number of partitions is not too high. If we use $2p$ rectangles, then a random partition fullfills this property with probability of at least $\frac{1}{2}$. However, the number of sites n is a lower bound of the number of different values v_i . Using this observation one can find an algorithm that always determines such a split in polynomial time, even if Δ is arbitrarily small.

Theorem 3. *Given a set of sites V in k -dimensional Euclidean space and a non-negative weighting w such that the sum of all weights is polynomial in $n = |V|$; there exists a tree with a weighted distance that differs by the optimal Weighted Cost-Distance by at most a factor of $O(k \log n)$. Such a tree has size $O(c(MST(V))k \log n)$ and can be computed in polynomial time.*

Proof. We construct a fair split tree using the partition introduced in Lemma 3. We consider the vertex pair sets $P_1 := V_1^2$, $P_2 := V_2^2$, and $Q := V \times V \setminus (P_1 \cup P_2)$.

It holds for pairs in Q :

$$\sum_{(u,v) \in Q} w(u,v) L_T(u,v) \leq \frac{3D}{\ell(R)} L_T(u,v) \leq \frac{3D}{\ell(R)} 3k\ell(R) \leq 9kD ,$$

where $D := \sum_{u,v} w(u,v) \|u,v\|_2$ is a lower bound for the weighted distance of the optimal network. For the disjoint pair sets P_1 and P_2 we apply this technique recursively for at most $O(ck \log n)$ rounds. As we have already observed, the length of the longest edge of the sub-rectangles is at most $\ell' := \frac{\ell(R_0)}{n^c}$. Then we face partitions $P_1, \dots, P_m$ with partial weight sums $W_1, \dots, W_m$ ($W_i := \sum_{(u,v) \in P_i} w(u,v)$). The sum of all weights $W := \sum_{u,v} w(u,v)$ is bounded by a polynomial $O(n^c)$. Therefore, $\sum_i W_i \leq W = O(n^c)$. The corresponding normalized weighted distances $\Delta_i := \sum_{u,v \in P_i} \frac{w(u,v)}{W_i} \|u,v\|_2$ are bounded by ℓ_i , which is the length of the longest edge of the partition P_i 's rectangle. Note that

$$\begin{aligned} \sum_{i \in P_i} L_T(u,v) w(u,v) &\leq 2 \sum_i \ell_i w(u,v) \leq 2 \sum_i \ell_i W_i \\ &\leq 2\ell' W \leq c' \ell(R_0) \leq c' s(MST(V)) \end{aligned}$$

for a suitable constant c' . This and the recurrency over $O(k \log n)$ rounds imply

$$\begin{aligned} \sum_{u,v} w(u,v) L_T(u,v) &\leq c' s(MST(V)) + c'' k(\log n) D \\ &\leq c''' k(\log n) WCD_w(N) \end{aligned}$$

for a suitable constant c'' and c''' and every network N . $\square$

3 A Lower Bound for Tree-Approximations

Trees cannot approximate the optimal Weighted Cost-Distance graph better than stated in Theorem 2. To show this, we construct a counter-example where

the sites are uniformly distributed and the weighting supports only neighbored sites.

In particular, we consider an $n \times n$ unit square grid G and the following weighting function:

$$w(u, v) = \begin{cases} 1 : \|u, v\|_2 = 1 \\ 0 : \|u, v\|_2 \neq 1 \end{cases}.$$

Clearly, the weighted Cost-Distance of the grid consisting of all positive weighted edges is $O(n^2)$ and since the minimum spanning tree has at least cost $n^2 - 1$, this network is optimal up to a constant factor. We will show that every spanning tree T has weighted distance $\Omega(n^2 \log n)$ even if we allow T to use non-terminal vertices.

Let G_i be the set of vertices with distance $i - 1$ to the convex hull of the grid, i.e. G_1 is the convex hull and G_{i+1} is the convex hull of $G \setminus \bigcup_{j \leq i} G_j$.

Lemma 4. *For every spanning tree T of the grid and for all $i \leq n/2$ there exist two grid neighbors $u, v \in G_i$ such that the connecting path in T has at least length $\frac{n}{2}$.*

Proof. Assume the contrary and consider the upper row of G_i . Note that neighbored vertices (in the grid) are connected by a path which is too short to reach the other half of the grid. Therefore in the upper row the leftmost and the rightmost vertex must be connected by a path, which is completely in the upper half of the rectangle.

For symmetry reasons the analogous property is true for the the left column, the lower row, and the right column. Therefore there exists a cycle that encloses the center of the grid, contradicting the tree property. $\square$

Definition 2 (spanning cut). *A spanning cut splits a tree $T = (V, E)$ by a straight line s into trees $T_1 = (V_1 \cup S_1, E_1)$ and $T_2 = (V_2 \cup S_2, E_2)$. These sub-trees are entirely in the left or right half-space defined by s . All vertices in V_2 (resp. V_1) are orthogonally projected onto s and will be used as non-terminals S_1 in T_1 (resp. S_2 in T_2). All edges in trees T_1 and T_2 are copied from the original tree.*

So, we copy every tree into both half spaces without increasing any edge length, for an example see Fig. 4.

Lemma 5. *For a spanning cut of T in to T_1 and T_2 we have for all $u_1, u_2 \in V(T_1)$ and $v_1, v_2 \in V(T_2)$:*

$$L_G(u_1, u_2) \geq L_{G_1}(u_1, u_2) \text{ and } L_G(v_1, v_2) \geq L_{G_2}(v_1, v_2)$$

Theorem 4. *For every spanning tree T of the $n \times n$ -grid, where $w(u, v) = 1$ if u and v are neighbored vertices and $w(u, v) = 0$ elsewhere, the weighted Cost-Distance is at least $\Omega(n^2 \log n)$, while the optimal Cost-Distance network has cost and weighted distance $O(n^2)$.*

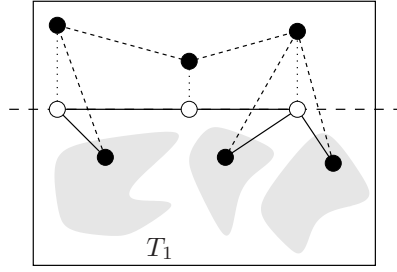


Fig. 4. A spanning cut and the resulting sub-tree in the lower halfspace

Proof. We will split this grid into 16 sub-grids of size $\frac{n}{4} \times \frac{n}{4}$ by 15 spanning cuts (Fig. 5). By Lemma 5 the sum of the weighted distances of the sub-grids is a lower bound for the over-all grid (We also split the weightings into 16 local weightings).

Lemma 4 implies that in every subset G_i there are paths $p_1, \dots, p_{n/2}$ between neighboring vertices with length of at least $n/2$. Furthermore, we can choose these paths such that the spanning cut reduces the lengths of all of them by at least $\frac{n}{4}$, since they reach the other side of the grid.

This way, we can account the length $\frac{n}{4}$ of these $\frac{n}{2}$ paths for this recursion level. This leads to the following recurrency for the weighted distance $W(n)$ of spanning trees of a $n \times n$ -grid:

$$W(n) \geq \frac{n^2}{8} + 16 W(n/4) .$$

Resolving this recurrency proves the claim. $\square$

Applying the algorithm of Section 2 to this instance produces trees structured similar to the U-Layout shown in Fig. 6. Such trees optimize the weighted Cost-Distance of an $n \times n$ grid by a factor of $\Theta(\log n)$.

4 Conclusions and Future Research

As an immediate implication of Theorem 3 we can state the following approximation result:

Corollary 3. *For polynomial weights the Weighted Cost-Distance-Tree problem can be polynomially approximated within a factor of $O(\log n)$.*

There is some hope that the approximation techniques introduced by Arora [1] may lead to a polynomial time approximation scheme. Another follow-up result may be the extension to general metrics. We conjecture that the results of [3] lead to an $O(\log^2 n)$ approximation.

An interesting open question is: if W , the sum of all weights, is super-polynomial, does the upper bound of section 3 also apply? Or can the lower

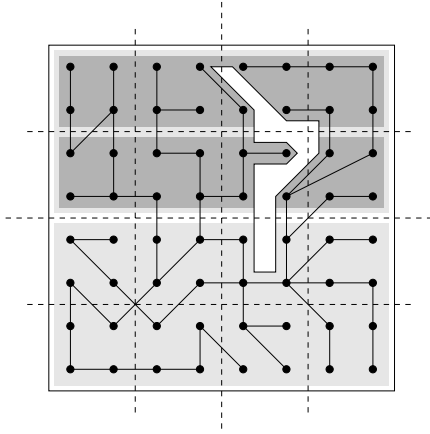


Fig. 5. The white marked p-shaped area induces long paths for a number of neighbored pairs. For the lower bound the grid is tiled into 16 sub-grids

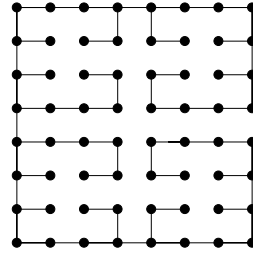


Fig. 6. The U-Layout approximates the Cost-Distance of this instance by a factor of $\Theta(\log n)$

bound factor be increased for such weightings? This mirrors the case in the original setting (Equation (1)) that the fixed costs are sub-polynomial compared to the linear costs.

Another extension of these results may be to consider different metrics for cost and distance as introduced in [10]. They proved a $O(\log n)$ -approximation for the two-metrics Cost-Distance problem with weights only on the root-vertex pairs. We have shown that for pairwise weight trees do not approximate better than $\Theta(\log n)$, while for vertex-root weightings Meyerson et al. [10] showed that a tree is always part of the optimal solution. It is an interesting open question whether trees approximate this Weighted Cost-Distance problem with different metrics within a factor of $O(\log n)$.

References

1. Sanjeev Arora. Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *Journal of the ACM*, 45(5):753–782, 1998. 187, 193
2. S. Arya, G. Das, D. M.Mount, J. S. Slowe, and M. Smid. Euclidean spanners: Short, thin, and lanky. In *Proc. 27th ACM Symp. Theory of Computing*. ACM, 1995. 187, 188, 189
3. Bartal. On approximating arbitrary metrics by tree metrics. In *ACM Symposium on Theory of Computing*, 1998. 188, 193
4. M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, 1998. 188

5. Gautam Das, Paul J. Heffernan, and Giri Narasimhan. Optimally sparse spanners in 3-dimensional euclidean space. In *Symposium on Computational Geometry*, pages 53–62, 1993. 190
6. David Eppstein. Spanning trees and spanners. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, chapter 9, pages 425–461. Elsevier Science Publishing, 2000. 190
7. M. R. Garey, R. L. Graham, and David S. Johnson. Some NP-complete geometric problems. In *ACM Symposium on Theory of Computing*, pages 10–22, 1976. 187
8. S. Khuller, B. Raghavachari, and N. Young. Balancing minimum spanning trees and shortest path trees. In *Algorithmica*, 14, pages 305–321, 1995. 187
9. C. Levcopoulos and A. Lingas. There are planar graphs almost as good as the complete graphs and as short as minimum spanning trees. In *Proc. Internat. Symp. on Optimal Algorithms, volume 401 of LNCS*, pages 9–13. Springer-Verlag, 1989. 187, 188
10. A. Meyerson, K. Munagala, and S. Plotkin. COST-DISTANCE: Two metric network design. In *Proc. 41st Symp. Foundations of Computer Science*. IEEE, 2000. 187, 194
11. Birgitta Weber. Netzwerke optimiert für lokale Kosten-Distanzen. Lübeck University, Diploma Thesis, 2001. 188

Necessary and Sufficient Numbers of Cards for Sharing Secret Keys on Hierarchical Groups (Extended Abstracts)

Takaaki Mizuki^{1,2} and Takao Nishizeki³

¹ Information Synergy Center, Tohoku University,
Aoba-yama, Aoba-ku, Sendai 980-8578, Japan
mizuki@nishizeki.ecei.tohoku.ac.jp

² PRESTO, JST

³ Graduate School of Information Sciences, Tohoku University,
Aoba-yama 05, Aoba-ku, Sendai 980-8579, Japan
nishi@ecei.tohoku.ac.jp

Abstract. Suppose that there are players in two hierarchical groups and a computationally unlimited eavesdropper. Using a random deal of cards, a player in the higher group wishes to send a one-bit message information-theoretically securely either to all the players in her group or to all the players in the two groups. This can be done by the so-called 2-level key set protocol. In this paper we give a necessary and sufficient condition for the 2-level key set protocol to succeed.

1 Introduction

Suppose that there are k (≥ 2) players $P_1, P_2, \dots, P_k$ and a passive eavesdropper, Eve, whose computational power is unlimited. Consider a graph called a *key exchange graph*, in which each vertex i represents a player P_i and each edge (i, j) joining vertices i and j represents a pair of players P_i and P_j sharing a one-bit secret key $r_{ij} \in \{0, 1\}$ that is information-theoretically secure against the eavesdropper Eve. Refer to [6] for the graph-theoretic terminology. A connected graph having no cycle is called a *tree*. If the key exchange graph is a tree, then an arbitrary player can send a one-bit message $m \in \{0, 1\}$ to all the players information-theoretically securely as follows: the player sends the message m to the rest of the players along the tree; when player P_i sends m to player P_j along an edge (i, j) of the tree, P_i computes the exclusive-or $m \oplus r_{ij}$ of m and r_{ij} and sends it to P_j , and P_j obtains m by computing $(m \oplus r_{ij}) \oplus r_{ij}$.

For $k = 2$, Fischer *et al.* give a protocol using a random deal of cards to connect the two players P_1 and P_2 with an edge, that is, to form a tree on the two players [1]. (A random deal of cards will be formally described in Section 2.1.) Fischer and Wright extend this protocol to form a tree for any $k \geq 2$; they formalize a class of protocols called the “key set protocol,” the definition of which will be given in Section 2.2 [2,5]. They also give a sufficient condition on the numbers of cards for the “key set protocol” to always form a tree. Mizuki *et*

al. give a simple necessary and sufficient condition on the numbers of cards for the “key set protocol” to always form a tree [7,9].

On the other hand, Yoshikawa *et al.* consider the following more general problem [10,11]. Suppose that the k players are partitioned into two hierarchical groups, which are represented as V_1 and V_2 , where $V_1 \cup V_2 = \{1, 2, \dots, k\}$ and $V_1 \cap V_2 = \emptyset$. In the hierarchy, the group V_1 is assumed to be higher than the group V_2 . Yoshikawa *et al.* wish to form, as a key exchange graph, a tree T such that the subgraph T_1 of T induced by V_1 is also a tree. Such a tree is called a *2-level tree* (for the hierarchy). Once a 2-level tree T is formed, any player in the higher group V_1 can send a one-bit message m either to all the players in V_1 or to all the players in $V_1 \cup V_2$, because both T_1 and T are connected. Yoshikawa *et al.* modify the “key set protocol” in [2,5] so that their protocol, called a “2-level protocol,” forms a 2-level tree; the formal definition of the “2-level protocol” will be given in Section 2.3. They give a sufficient condition on the numbers of cards for the “2-level protocol” to always form a 2-level tree. However, their condition is not a necessary one, and hence it has been an open problem to obtain a necessary and sufficient condition.

In this paper, we give a necessary and sufficient condition on the numbers of cards for the “2-level protocol” to always form a 2-level tree, and hence close the open problem. Using our necessary and sufficient condition, one can easily know the minimum number of cards needed to form a 2-level tree.

2 Preliminaries

We first formally describe a random deal of cards in Section 2.1, then explain the “key set protocol” in Section 2.2, and finally explain the “2-level protocol” in Section 2.3.

2.1 Random Deal of Cards

In this subsection we formally describe a random deal of cards [4].

Let C be a set of d distinct cards which are numbered from 1 to d . All cards in C are randomly dealt to players $P_1, P_2, \dots, P_k$ and an eavesdropper Eve. We call a set of cards dealt to a player or Eve a *hand*. Let $C_i \subseteq C$ be P_i ’s hand for each $1 \leq i \leq k$, and let $C_e \subseteq C$ be Eve’s hand. We denote this *deal* by $\mathcal{C} = (C_1, C_2, \dots, C_k; C_e)$. Clearly $\{C_1, C_2, \dots, C_k, C_e\}$ is a partition of set C . We write $c_i = |C_i|$ for each $1 \leq i \leq k$ and $c_e = |C_e|$, where $|A|$ denotes the cardinality of a set A . Note that $c_1, c_2, \dots, c_k$ and c_e are the sizes of hands held by $P_1, P_2, \dots, P_k$ and Eve respectively, and that $d = \sum_{i=1}^k c_i + c_e$. We call $\gamma = (c_1, c_2, \dots, c_k; c_e)$ the *signature* of deal $\mathcal{C}$. The set C and the signature γ are public to all the players and even to Eve, but the cards in the hand of a player or Eve are private to herself, as in the case of usual card games.

Using a random deal of cards, a protocol can make several pairs of players share a one-bit secret key, as we will explain in the succeeding subsection. A reasonable situation in which such a protocol is practically required is discussed in [3,5], and also the reason why we deal cards even to Eve is found there.

2.2 Key Set Protocol

In this subsection we explain the “key set protocol” formalized in [2,5].

We first define some terms. A *key set* $K = \{x, y\}$ consists of two cards x and y , one in C_i , the other in C_j with $i \neq j$, say $x \in C_i$ and $y \in C_j$. We say that a key set $K = \{x, y\}$ is *opaque* if $1 \leq i, j \leq k$ and Eve cannot determine whether $x \in C_i$ or $x \in C_j$ with probability greater than $1/2$. Note that both players P_i and P_j know that $x \in C_i$ and $y \in C_j$. If K is an opaque key set, then P_i and P_j can share a one-bit secret key $r_{ij} \in \{0, 1\}$, using the following rule agreed on before starting a protocol: $r_{ij} = 0$ if $x > y$; $r_{ij} = 1$, otherwise. Since Eve cannot determine whether $r_{ij} = 0$ or $r_{ij} = 1$ with probability greater than $1/2$, the secret key r_{ij} is information-theoretically secure. We say that a card x is *discarded* if all the players agree that x has been removed from someone’s hand, that is, $x \notin (\bigcup_{i=1}^k C_i) \cup C_e$. We say that a player P_i *drops out* of the protocol if she no longer participates in the protocol. We denote by V the set of indices i of all the players P_i remaining in the protocol. Note that $V = \{1, 2, \dots, k\}$ before starting a protocol.

The “key set protocol” has the following four steps.

1. Choose a player P_s , $s \in V$, as a *proposer* by a certain procedure.
2. The proposer P_s determines in mind two cards x, y . The cards are randomly picked so that x is in her hand and y is not in her hand, i.e. $x \in C_s$ and $y \in (\bigcup_{i \in V - \{s\}} C_i) \cup C_e$. Then P_s proposes $K = \{x, y\}$ as a key set to all the players. (The key set is proposed just as a set. Actually it is sorted in some order, for example in ascending order, so Eve learns nothing about which card belongs to C_s unless Eve holds y .)
3. If there exists a player P_t holding y , then P_t accepts K . Since K is an opaque key set, P_s and P_t can share a one-bit secret key r_{st} that is information-theoretically secure from Eve. (In this case an edge (s, t) is added to the key exchange graph.) Both cards x and y are discarded. Let P_i be either P_s or P_t that holds the smaller hand; if P_s and P_t hold hands of the same size, let P_i be the proposer P_s . P_i discards all her cards and drops out of the protocol. Set $V := V - \{i\}$. Return to step 1.
4. If there exists no player holding y , that is, Eve holds y , then both cards x and y are discarded. Return to step 1. (In this case no new edge is added to the key exchange graph.)

These steps 1–4 are repeated until either exactly one player remains in the protocol or there are not enough cards left to complete step 2 even if two or more players remain. In the first case the key exchange graph becomes a tree. In the second case the key exchange graph does not become a connected graph and hence does not become a tree.

Considering various procedures for choosing a proposer P_s in step 1, we obtain the class of *key set protocols*.

We say that a key set protocol *works for a signature* γ if the protocol always forms a tree as a key exchange graph for any deal $\mathcal{C}$ having the signature γ

and for any random selection of cards x and y in step 2. Let $k \geq 2$ and $\gamma = (c_1, c_2, \dots, c_k; c_e)$. Without loss of generality one may assume in this subsection that $c_1 \geq c_2 \geq \dots \geq c_k$. Let W be the set of all signatures for each of which there is a key set protocol working, and let L be the set of all signatures for each of which there is no key set protocol working. A simple necessary and sufficient condition for $\gamma \in W$ has been known [2,7,9]. Before mentioning the condition, we give some definitions.

We say that a player P_i is *feasible in γ* if one of the following conditions (1) and (2) holds:

- (1) $c_i \geq 2$; and
- (2) $c_e = 0$, $c_i = 1$ with $i = k$, and $c_{k-1} \geq 2$.

We define a mapping f from the set of all signatures to $\{0, 1, 2, \dots, k\}$, as follows: $f(\gamma) = i$ if P_i is the feasible player in γ with the smallest hand (ties are broken by selecting the player having the largest index); and $f(\gamma) = 0$ if there is no feasible player. We denote $f(\gamma)$ simply by f .

The following Lemma 1 immediately holds.

Lemma 1 ([2,9]) *Let $\gamma \in W$. If $k \geq 2$, then $c_k \geq 1$ and $\sum_{i=1}^k c_i \geq c_e + 2k - 2$. If $k \geq 3$, then $f \geq 1$.*

The following Theorems 2, 3 and 4 provide a necessary and sufficient condition for $\gamma \in W$. In this subsection, let $B = \{i \mid c_i = 2, 1 \leq i \leq k\}$, and let $b = \lfloor |B|/2 \rfloor$.

Theorem 2 ([2]) *Let $k = 2$. Then $\gamma \in W$ if and only if $c_2 \geq 1$ and $c_1 + c_2 \geq c_e + 2$.*

Theorem 3 ([7,9]) *Let $k = 3$. Then $\gamma \in W$ if and only if $c_3 \geq 1$ and $c_1 + c_3 \geq c_e + 3$.*

Theorem 4 ([7,9]) *Let $k \geq 4$, $c_k \geq 1$, and $f \geq 1$. Then $\gamma \in W$ if and only if*

$$\sum_{i=1}^k \max\{c_i - h^+, 0\} \geq \tilde{f}, \quad (1)$$

where

$$\bar{f} = f - \delta, \quad (2)$$

$$\tilde{f} = \bar{f} - 2\epsilon, \quad (3)$$

$$h = c_e - c_k + k - \bar{f}, \quad (4)$$

$$h^+ = h + \epsilon, \quad (5)$$

$$\delta = \begin{cases} 0 & \text{if } f = 1; \\ 1 & \text{if } 2 \leq f \leq k - 1; \\ 2 & \text{if } f = k \text{ and } c_{k-1} \geq c_k + 1; \text{ and} \\ 3 & \text{if } f = k \text{ and } c_{k-1} = c_k, \end{cases} \quad (6)$$

and

$$\epsilon = \begin{cases} \max\{\min\{c_2 - h, b\}, 0\} & \text{if } 5 \leq f \leq k - 1; \\ \max\{\min\{c_2 - h, b - 1\}, 0\} & \text{if } 5 \leq f = k \text{ and } c_e \geq 1; \text{ and} \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

Fischer and Wright give the *SFP* (*smallest feasible player*) *protocol*, which always chooses the feasible player with the smallest hand as a proposer, that is, chooses the proposer P_s as follows:

$$s = \begin{cases} f & \text{if } 1 \leq f \leq k; \\ 1 & \text{if } f = 0. \end{cases}$$

We say that a key set protocol is *optimal* if the protocol works for all signatures in W . Fischer and Wright prove the following Theorem 5.

Theorem 5 ([2,5]) *The SFP protocol is optimal.*

Furthermore, a characterization of optimal key set protocols is given in [7,8].

2.3 2-Level Protocol

In this subsection we explain the “2-level protocol” given in [10,11].

Suppose that there are two hierarchical groups V_1 and V_2 . The “2-level protocol” forms a 2-level tree, whose subgraph induced by V_1 is connected. The “2-level protocol” forms a 2-level tree in which every vertex in V_2 has degree one, that is, every vertex in V_2 is a leaf. The “2-level protocol” is obtained by slightly modifying steps 1 and 3 in the key set protocol, as follows: in step 1, a player in V_1 is always chosen as a proposer P_s ; and in step 3, whenever card y is held by a player P_t in V_2 , P_t drops out of the protocol even if P_t holds the larger hand than P_s . Thus the “2-level protocol” has the following four steps.

1. Choose a player P_s , $s \in V_1$, as a *proposer* by a certain procedure.
2. The proposer P_s randomly determines in mind two cards x, y so that x is in her hand and y is not in her hand. Then P_s proposes $K = \{x, y\}$ as a key set to all the players.
3. If there exists a player P_t holding y , then P_s and P_t can share a one-bit secret key r_{st} . Both cards x and y are discarded.
 - (a) If $t \in V_1$, then let P_i be either P_s or P_t that holds the smaller hand; when P_s and P_t hold hands of the same size, let P_i be the proposer P_s . P_i discards all her cards and drops out of the protocol. Set $V_1 := V_1 - \{i\}$. Return to step 1.
 - (b) If $t \in V_2$, then P_t discards all her cards and drops out of the protocol. Set $V_2 := V_2 - \{t\}$. Return to step 1.
4. If there exists no player holding y , that is, Eve holds y , then both cards x and y are discarded. Return to step 1.

These steps 1–4 are repeated until either exactly one player in V_1 remains in the protocol or there are not enough cards left to complete step 2 even if two or more players remain. In the first case the key exchange graph becomes a 2-level tree, in which every vertex in V_2 has degree one. In the second case the key exchange graph does not become a 2-level tree.

Considering various procedures for choosing a proposer P_s in step 1, we obtain the class of *2-level protocols*.

Without loss of generality one may assume that $V_1 = \{1, 2, \dots, k_1\}$ and $V_2 = \{k_1 + 1, k_1 + 2, \dots, k_1 + k_2\}$ where $k = k_1 + k_2$. One may assume that all the players in V_2 hold at least one card, i.e. $c_i \geq 1$ for all i , $k_1 + 1 \leq i \leq k_1 + k_2$. Once an edge is connected to a player in V_2 during the execution of any 2-level protocol, the player in V_2 necessarily drops out of the protocol. Therefore any player in V_2 does not need two or more cards. More precisely, there is a 2-level protocol which always forms a 2-level tree for $\gamma = (c_1, c_2, \dots, c_{k_1}, c_{k_1+1}, c_{k_1+2}, \dots, c_{k_1+k_2}; c_e)$ if and only if there is a 2-level protocol which always forms a 2-level tree for $\gamma = (c_1, c_2, \dots, c_{k_1}, 1, 1, \dots, 1; c_e)$. We thus use a *2-level signature* $\alpha = (c_1, c_2, \dots, c_{k_1}; k_2; c_e)$ to represent a signature $\gamma = (c_1, c_2, \dots, c_{k_1}, c_{k_1+1}, c_{k_1+2}, \dots, c_{k_1+k_2}; c_e)$. Remember that k_2 is the number of players in V_2 .

We say that a 2-level protocol *works for a 2-level signature* α if the protocol always forms a 2-level tree as a key exchange graph for any deal $\mathcal{C}$ having the 2-level signature α and for any random selection of cards x and y in step 2. Let $k_1 \geq 1$, $k_1 + k_2 \geq 2$, and $\alpha = (c_1, c_2, \dots, c_{k_1}; k_2; c_e)$. One may assume without loss of generality that $c_1 \geq c_2 \geq \dots \geq c_{k_1}$. Let W^2 be the set of all 2-level signatures for each of which there is a 2-level protocol working, and let L^2 be the set of all 2-level signatures for each of which there is no 2-level protocol working.

We say that a player P_i , $i \in V_1$, is *feasible in a 2-level signature* $\alpha = (c_1, c_2, \dots, c_{k_1}; k_2; c_e)$ if one of the following conditions (1), (2) and (3) holds:

- (1) $c_i \geq 2$;
- (2) $k_2 = 0$, $c_e = 0$, $c_i = 1$ with $i = k_1$, and $c_{k_1-1} \geq 2$; and
- (3) $k_1 = k_2 = 1$, $c_e = 0$, and $c_i = 1$ with $i = 1$.

If all players hold at least one card and we choose a feasible player P_s satisfying the condition (1) or (2) above as a proposer, then, after executing steps 1–4, all the players remaining in the protocol will always hold at least one card. If we choose a feasible player P_s satisfying the condition (3) above as a proposer, then, after executing steps 1–4, there is exactly one player remaining in the protocol but she holds no card.

We define a mapping g from the set of all 2-level signatures to $\{0, 1, 2, \dots, k_1\}$, as follows: $g(\alpha) = i$ if P_i is the feasible player in α with the smallest hand (ties are broken by selecting the player having the largest index); and $g(\alpha) = 0$ if there is no feasible player. For example, if $\alpha = (9, 9, 8, 6, 5, 3, 2, 2, 1, 1; 2; 2)$ as illustrated in Figure 1, then $g(\alpha) = 8$. We denote $g(\alpha)$ simply by g .

Yoshikawa *et al.* give a sufficient condition for $\alpha \in W^2$ as in the following Theorem 6.

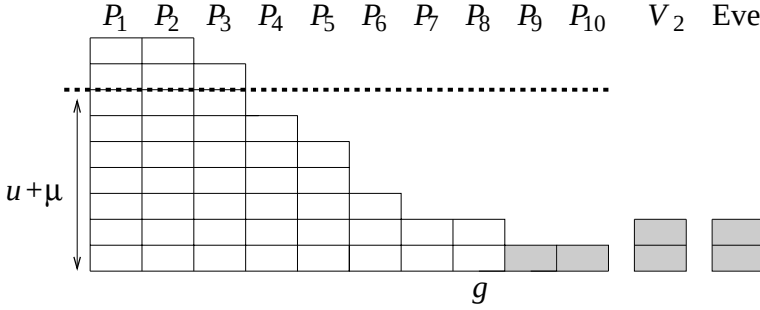


Fig. 1. An illustration of $\alpha = (9, 9, 8, 6, 5, 3, 2, 2, 1, 1; 2; 2)$

Theorem 6 ([10,11]) *Let $k_1 \geq 1$, $k_2 \geq 1$, and $c_{k_1} \geq 1$. If there exists k_0 such that $0 \leq k_0 \leq k_1 - 1$ and $c_{k_1-k_0} \geq c_e + \lfloor \log_2(k_1 - k_0) \rfloor + k_0 + k_2$, then $\alpha \in W^2$.*

They prove Theorem 6 by showing that the 2-level protocol choosing the player P_g as a proposer works for any 2-level signature satisfying the condition in Theorem 6. However, their sufficient condition in Theorem 6 is not a necessary one. For example, the 2-level signature $\alpha = (9, 9, 8, 6, 5, 3, 2, 2, 1, 1; 2; 2)$ above does not satisfy their sufficient condition in Theorem 6, while it is actually in W^2 as we will see in Section 3. Thus it has been an open problem to obtain a necessary and sufficient condition for $\alpha \in W^2$. This paper closes the open problem in Section 3, that is, provides a necessary and sufficient condition for $\alpha \in W^2$. Before giving our condition, we define some terms in the remainder of this subsection.

If a 2-level protocol works for a 2-level signature α , then the key exchange graph must become a 2-level tree for any deal $\mathcal{C}$ having the 2-level signature α and for any random selection of cards x and y in step 2. Hence, whoever has the card y contained in the proposed key set $K = \{x, y\}$, the key exchange graph should become a 2-level tree. The “malicious adversary” determines who holds the card y . Considering a malicious adversary to make it hard for the key exchange graph to become a 2-level tree, we obtain a condition for $\alpha \in W^2$. We use a function $\mathcal{A}$ to represent a malicious adversary, as follows. The inputs to the function $\mathcal{A}(\alpha, s)$ are the current 2-level signature α and the index s of a proposer P_s chosen by the protocol. Its output is either the index t of a player P_t remaining in the protocol or the index e of Eve; $\mathcal{A}(\alpha, s) = t \neq e$ means that player P_t holds card y ; and $\mathcal{A}(\alpha, s) = e$ means that Eve holds card y .

From now on, we denote by $\alpha = (c_1, c_2, \dots, c_{k_1}; k_2; c_e)$ the current 2-level signature, and denote by $\alpha'_{(s, \mathcal{A})} = (c'_1, c'_2, \dots, c'_{k'_1}; k'_2; c'_e)$ the resulting 2-level signature after executing steps 1–4 under the assumption that P_s proposes a key set $K = \{x, y\}$ and $y \in C_{\mathcal{A}(\alpha, s)}$. It should be noted that $c'_e + k'_1 + k'_2 = c_e + k_1 + k_2 - 1$ always holds by the definition of 2-level protocols.

Note that $\alpha \in W^2$ if and only if there exists a proposer P_s such that $\alpha'_{(s, \mathcal{A})} \in W^2$ for any malicious adversary $\mathcal{A}$; for the sake of convenience any

2-level signature $\alpha = (c_1; 0; c_e)$ is assumed to be in W^2 (similarly, we assume that any signature $\gamma = (c_1; c_e)$ is in W). That is,

$$\alpha \in W^2 \iff \exists s \forall \mathcal{A} \quad \alpha'_{(s, \mathcal{A})} \in W^2,$$

in other words,

$$\alpha \in L^2 \iff \forall s \exists \mathcal{A} \quad \alpha'_{(s, \mathcal{A})} \in L^2.$$

It follows from the definition of 2-level protocols that if two players P_i and P_j with $i, j \in V_1$ hold hands of the same size, that is, $c_i = c_j$, then

$$\forall \mathcal{A} \quad \alpha'_{(i, \mathcal{A})} \in W^2 \iff \forall \mathcal{A} \quad \alpha'_{(j, \mathcal{A})} \in W^2.$$

Hence, one may assume without loss of generality that the following two *Assumptions 1* and *2* hold.

(Assumption 1)

If there exist two or more players P_i with $c_i = c_s$ and $i \in V_1$ (including the proposer P_s), then P_s has the largest index among all these players.

(Assumption 2)

If $\mathcal{A}(\alpha, s) = t \neq e$ and there exist two or more players P_i with $c_i = c_t$ and $i \in V_1 - \{s\}$ (including P_t), then P_t has the largest index among all these players.

Under the two assumptions above, $\alpha'_{(s, \mathcal{A})} = (c'_1, c'_2, \dots, c'_{k_1}; k'_2; c'_e)$ satisfies $c'_1 \geq c'_2 \geq \dots \geq c'_{k_1}$ since α satisfies $c_1 \geq c_2 \geq \dots \geq c_{k_1}$. (For key set protocols, we also assume that assumptions similar to Assumptions 1 and 2 hold.)

We now show in the following Lemma 7 that one should not choose a non-feasible player as a proposer.

Lemma 7 *Let $k_1 \geq 1$, $k_2 \geq 1$, and $c_{k_1} \geq 1$. If P_s is not a feasible proposer in α , then there exists a malicious adversary $\mathcal{A}$ such that $\alpha'_{(s, \mathcal{A})} \in L^2$.*

Proof. Assume that the proposer P_s is not feasible in α . Then $c_s = 1$, and either $k_1 \geq 2$, $k_2 \geq 2$ or $c_e \geq 1$ because $k_2 \geq 1$. Therefore, either (i) $k_1 + k_2 \geq 3$ or (ii) $k_1 = k_2 = 1$ and $c_e \geq 1$. Let $\mathcal{A}$ be a malicious adversary such that

$$\begin{cases} \mathcal{A}(\alpha, s) \in V_2 & \text{if } k_1 + k_2 \geq 3; \text{ and} \\ \mathcal{A}(\alpha, s) = e & \text{if } k_1 = k_2 = 1 \text{ and } c_e \geq 1. \end{cases}$$

Then P_s 's hand becomes empty, and hence we have $\alpha'_{(s, \mathcal{A})} = (c'_1, c'_2, \dots, 0; k'_2; c'_e)$. Clearly $\alpha'_{(s, \mathcal{A})} \in L^2$. $\square$

Lemma 7 immediately implies that $g \geq 1$ is a trivial necessary condition for $\alpha \in W^2$ when $k_1 \geq 1$ and $k_2 \geq 1$.

3 Main Results

In this section we give a necessary and sufficient condition for $\alpha \in W^2$.

For the case where $k_2 = 0$, a 2-level protocol can be regarded as a key set protocol. Therefore, for this case, Theorems 2, 3 and 4 immediately provide a necessary and sufficient condition for a 2-level signature α to be in W^2 . One may thus assume that $k_2 \geq 1$.

Our main result is the following Theorem 8. Note that $c_{k_1} \geq 1$ and $g \geq 1$ are trivial necessary conditions for $\alpha \in W^2$. Hereafter we define $B = \{i \mid c_i = 2, 1 \leq i \leq k_1\}$ and $b = \lfloor |B|/2 \rfloor$ for a 2-level signature α .

Theorem 8 *Let $k_1 \geq 1$, $k_2 \geq 1$, $c_{k_1} \geq 1$, and $g \geq 1$. Then*

$$\alpha = (c_1, c_2, \dots, c_{k_1}; k_2; c_e) \in W^2$$

if and only if

$$c_1 - (u + \mu) + \sum_{i=2}^{k_1} \max\{c_i - (u + \mu), 0\} \geq g - 2\mu - 1, \quad (8)$$

where

$$u = c_e + k_1 + k_2 - g \quad (9)$$

and

$$\mu = \max\{\min\{c_3 - u, b\}, 0\}. \quad (10)$$

Note that the third term in the left-hand side of Eq. (8) is defined to be 0 when $k_1 = 1$, and that μ is defined to be 0 when $k_1 \leq 2$.

Consider again $\alpha = (9, 9, 8, 6, 5, 3, 2, 2, 1, 1; 2; 2)$ as an example. The 2-level signature α satisfies $k_1 = 10$, $k_2 = 2$, $c_e = 2$ and $g = 8$. Thus by Eq. (9) $u = 6$. Note that u is equal to the number of shaded rectangles in Figure 1. Since $B = \{7, 8\}$, $b = 1$. Since $c_3 = 8$, $u = 6$ and $b = 1$, we have $\mu = 1$ by Eq. (10). Thus

$$\begin{aligned} c_1 - (u + \mu) + \sum_{i=2}^{k_1} \max\{c_i - (u + \mu), 0\} &= c_1 - 7 + \sum_{i=2}^{10} \max\{c_i - 7, 0\} \\ &= 5 \\ &= g - 2\mu - 1. \end{aligned}$$

Therefore the 2-level signature α satisfies the condition (8) in Theorem 8, and hence $\alpha \in W^2$. Note that the left-hand side of Eq. (8) is equal to the number of cards above the dotted line in Figure 1.

Remember that $g \leq k_1$. It should be noted that Eq. (8) is equivalent to

$$c_1 - (u + \mu) + \sum_{i=2}^{g-2\mu-1} \max\{c_i - (u + \mu), 0\} \geq g - 2\mu - 1, \quad (11)$$

because $c_1 \geq c_2 \geq \dots \geq c_{k_1}$.

It seems at first glance that one can easily prove Theorem 8, because a simple necessary and sufficient condition for a signature γ to be in W has already been

known as in Theorems 2, 3 and 4. However, proving Theorem 8 is a non-trivial task, as we will see in the succeeding section. The main reason is that one cannot choose a player in V_2 as a proposer although one has to make all players in V_2 drop out of the protocol until the protocol terminates.

From Theorem 8 we have the following Corollary 9, which provides a necessary and sufficient condition for $\alpha \in W^2$ under a natural assumption that all players in V_1 hold hands of the same size.

Corollary 9 *Let $k_1 \geq 1$, $k_2 \geq 1$, $c_{k_1} \geq 1$, $g \geq 1$, and $c_1 = c_2 = \cdots = c_{k_1}$. Then $\alpha \in W^2$ if and only if*

$$c_1 \geq \begin{cases} 3 & \text{if } k_1 \geq 4, k_2 = 1 \text{ and } c_e = 0; \\ c_e + k_2 & \text{if } k_1 = 1; \text{ and} \\ c_e + k_2 + 1 & \text{otherwise.} \end{cases} \quad (12)$$

Proof. omitted in this extended abstract.

Theorem 6 obtained by Yoshikawa *et al.* [10,11] implies that a sufficient condition for $\alpha \in W^2$ is $c_1 \geq c_e + k_2 + \lfloor \log_2 k_1 \rfloor$ when $c_1 = c_2 = \cdots = c_{k_1}$. Thus our necessary and sufficient condition in Theorem 8 is much better than the sufficient condition in [10,11].

4 Sketch of Proof of Theorem 8

In this section we give a sketch of a proof of Theorem 8. A complete proof will be given in a journal version.

We wish to prove that $\alpha \in W^2$ if and only if Eq. (8) in Theorem 8 holds. To simplify the notation, we denote by N the left-hand side of Eq. (8), that is,

$$N = c_1 - (u + \mu) + \sum_{i=2}^{k_1} \max\{c_i - (u + \mu), 0\}$$

for a 2-level signature α such that $k_1 \geq 1$, $k_2 \geq 1$, $c_{k_1} \geq 1$ and $g \geq 1$. We shall then prove that $\alpha \in W^2$ if and only if $N \geq g - 2\mu - 1$.

The outline of our proof is as follows. (i) We first transform a 2-level signature α into a signature γ corresponding to α . (ii) We then show that $\alpha \in W^2$ if and only if $\gamma \in W$. (iii) Using the known necessary and sufficient conditions for $\gamma \in W$ (Theorems 2, 3 and 4), we finally show that $\gamma \in W$ if and only if $N \geq g - 2\mu - 1$.

(i) We first transform a 2-level signature $\alpha = (c_1, c_2, \dots, c_{k_1}; k_2; c_e)$ into a signature γ , where γ is either $\sigma(\alpha)$ or $\tau(\alpha)$, as follows. For a 2-level signature α such that $k_1 \geq 1$ and $c_e = 0$, let

$$\sigma(\alpha) = (c_1, c_2, \dots, c_{k_1}; k_2). \quad (13)$$

Thus “ c_e ” for the signature $\sigma(\alpha)$ is equal to k_2 although $c_e = 0$ for the 2-level signature α , and “ k ” for $\sigma(\alpha)$ is equal to k_1 although $k = k_1 + k_2$ for α . For

a 2-level signature α such that $k_1 \geq 1$ and $c_{k_1} \geq 1$, let

$$\tau(\alpha) = (c_1, c_2, \dots, c_{k_1}, \overbrace{1, 1, \dots, 1}^{k_2}, c_e). \quad (14)$$

Thus “ k ” for $\tau(\alpha)$ is equal to $k = k_1 + k_2$. For a 2-level signature α such that $k_1 \geq 1$, we define *Condition A* as follows:

(Condition A)

$$k_2 = 1, c_{k_1} \geq 2 \text{ and } c_e = 0.$$

Note that a 2-level signature α satisfies Condition A if and only if P_{k_1+1} is feasible in the signature $\tau(\alpha)$. If α satisfies Condition A, then let $\gamma = \sigma(\alpha)$; otherwise, let $\gamma = \tau(\alpha)$.

(ii) We can prove that $\alpha \in W^2$ if and only if $\gamma \in W$, using a game-theoretic technique called a “strategy stealing argument,” which is used also in [2].

(iii) We can prove that $\gamma \in W$ if and only if $N \geq g - 2\mu - 1$, distinguishing the following two cases: the case where α satisfies Condition A, and the case where α does not satisfy Condition A.

5 Conclusion

Using a random deal of cards, the 2-level protocol given by Yoshikawa *et al.* makes some pairs of players in two hierarchical groups share secret keys so that any player in the higher group can send a one-bit secret message either to all the players in her group or to all the players in the two groups [10,11]. However, it has been an open problem to characterize the minimum numbers of cards which are required by the 2-level protocol to succeed, that is, to obtain a necessary and sufficient condition for a 2-level protocol to work for a 2-level signature α . In this paper, we close the open problem: we give in Theorem 8 a simple necessary and sufficient condition for a 2-level protocol to work for a 2-level signature α . One can efficiently determine in time $O(k)$ whether a given 2-level signature α satisfies our necessary and sufficient condition or not, where k is the number of players.

The 2-level protocol does not choose any player in the lower group as a proposer. However, one may modify the 2-level protocol so that the protocol may choose a player in the lower group as a proposer. It is an interesting open problem to obtain a necessary and sufficient condition for such a modified protocol to always form a 2-level tree for a signature γ .

In this paper, we consider the case where there are only two groups. Yoshikawa *et al.* [10,11] consider also the situation where there are three or more hierarchical groups, and give a method to distribute secret keys among players in these groups by modifying the 2-level protocol.

References

1. M. J. Fischer, M. S. Paterson, and C. Rackoff, "Secret bit transmission using a random deal of cards," DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, vol. 2, pp. 173–181, 1991. 196
2. M. J. Fischer and R. N. Wright, "An application of game-theoretic techniques to cryptography," DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, vol. 13, pp. 99–118, 1993. 196, 197, 198, 199, 200, 206
3. M. J. Fischer and R. N. Wright, "An efficient protocol for unconditionally secure secret key exchange," Proc. of the 4th Annual Symposium on Discrete Algorithms, pp. 475–483, 1993. 197
4. M. J. Fischer and R. N. Wright, "Bounds on secret key exchange using a random deal of cards," J. Cryptology, vol. 9, pp. 71–99, 1996. 197
5. M. J. Fischer and R. N. Wright, "Multiparty secret key exchange using a random deal of cards," Proc. CRYPTO '91, Lecture Notes in Computer Science, Springer-Verlag, vol. 576, pp. 141–155, 1992. 196, 197, 198, 200
6. F. Harary, "Graph Theory," Addison-Wesley, Reading, Mass., 1969. 196
7. T. Mizuki, "Sharing Unconditionally Secure Secret Keys," Ph.D. Thesis, Tohoku University, Sendai, 2000. Available at <http://www.nishizeki.ecei.tohoku.ac.jp/~mizuki/thesis.ps>. 197, 199, 200
8. T. Mizuki, H. Shizuya, and T. Nishizeki, "Characterization of optimal key set protocols," Proc. IFIP TCS 2000, Lecture Notes in Computer Science, Springer-Verlag, vol. 1872, pp. 273–285, 2000. 200
9. T. Mizuki, H. Shizuya, and T. Nishizeki, "Dealing necessary and sufficient numbers of cards for sharing a one-bit secret key," Proc. EUROCRYPT '99, Lecture Notes in Computer Science, Springer-Verlag, vol. 1592, pp. 389–401, 1999. 197, 199
10. R. Yoshikawa, S. Guo, K. Motegi, and Y. Igarashi, "Construction of secret key exchange spanning trees by random deals of cards on hierarchical structures," IEICE Trans. Fundamentals, vol. E84-A, no. 5, pp. 1110–1119, 2001. 197, 200, 202, 205, 206
11. R. Yoshikawa, S. Guo, K. Motegi, and Y. Igarashi, "Secret key exchange using random deals of cards on hierarchical structures," Proc. ISAAC 2000, Lecture Notes in Computer Science, Springer-Verlag, vol. 1969, pp. 290–301, 2000. 197, 200, 202, 205, 206

Generic Algorithms and Key Agreement Protocols Based on Group Actions

Akihiro Yamamura¹ and Kaoru Kurosawa²

¹ Communications Research Laboratory
4-2-1, Nukui-Kitamachi, Koganei, Tokyo 184-8795, Japan
aki@crl.go.jp

² Tokyo Institute of Technology
2-12-1, O-okayama, Meguro-ku, Tokyo 152-0033, Japan
kurosawa@ss.titich.ac.jp

Abstract. We propose a Diffie-Hellman-like key agreement protocol based on the computational intractability of reversing group action. The concept of a group action generalizes exponentiation and provides an algorithmic problem harder than the discrete logarithm problem. Using the action of the general linear group on the direct product of two cyclic groups, we invent a key agreement protocol secure against an attacker who has power to solve the discrete logarithm problem. We discuss a semantic secure asymmetric encryption scheme as well. Its security is evaluated in terms of a generic algorithm, which is a model of probabilistic algorithms over black box groups (similar to a straight-line program) and does not depend on any specific property of the group representation.

1 Introduction

A *generic algorithm* ([6,10,5,4,8,7]) is a general model for a probabilistic algorithm that finds an answer by querying group operation oracle and some other extra oracles. In such a model, a group is given as a *black box group* [1] and each group element is represented by a binary string of the same length. The Baby-Step-Giant-Step and Pohlig-Hellman are typical generic algorithms. The group operations, multiplication and taking the inverse, are carried out by querying the oracle in the model of generic algorithm. An answer may be a specific group element satisfying the given conditions or some information such as the discrete logarithm. A generic algorithm model is employed to analyze the discrete logarithm problem (DLOG), the Diffie-Hellman problem (DH) ([10,5]) and the cryptosystems ([4,7,8]).

We estimate the probability that a generic algorithm finds a correct answer over a black box group isomorphic to $\mathbb{Z}_p \times \mathbb{Z}_p$ with a limited number of oracle queries. We show that $\Omega(\sqrt[3]{p})$ queries to the group operation and discrete logarithm oracle are required to solve the *multiplicative discrete logarithm problem* (MDLOG) (and $\Omega(\sqrt[3]{p})$ queries for the algorithmic problem called ACTDH problem, which the proposed key agreement protocol is based on) in the generic algorithm model.

2 Key Agreement Protocol

2.1 Group Action

We recall a *group action* and exhibit examples in cryptology. Let S be a group, and X be a nonempty set. Note that S is not necessarily commutative. We say that S *acts on* X if there is a mapping $\sigma : X \times S \rightarrow X$ (the image of $(x, s) \in X \times S$ under σ is denoted by x^s) satisfying $x^{(st)} = (x^s)^t$ and $x^1 = x$ for s, t in S and x in X . The group action is ubiquitous in cryptology. The mechanism of the most popular cryptosystems, Diffie-Hellman and RSA, is explained in terms of the group action.

Example 1 Let p be a prime. Suppose a prime q divides $p - 1$. Take an element $a \in \mathbb{Z}_p^*$ such that $|g| = q$. Then the mapping $\sigma : \langle g \rangle \times \mathbb{Z}_q^* \rightarrow \langle g \rangle$ given by $\sigma(g, s) = g^s$ is an action of $\mathbb{Z}_q^*$ on the cyclic group $\langle g \rangle$. DLOG is characterized as the problem to find s for given g and $\sigma(g, s) = g^s$. This action is used in the Diffie-Hellman protocol [2] and the ElGamal encryption [3].

Example 2 Let p, q be primes. Set $n = pq$. The RSA cryptosystem employs the action σ defined as follows: $\sigma : \mathbb{Z}_{pq} \times (\mathbb{Z}/((p-1)(q-1)))^* \rightarrow \mathbb{Z}_{pq}$ given by $\sigma(m, e) = m^e$. RSA is based on the intractability of finding the e th root, that is, the problem to find m for given e and $\sigma(m, e) = m^e$.

2.2 General Scheme

Suppose that a group S acts on a set X and that the action is always efficiently computable. Let f be an efficiently computable function on X . Note that S is not necessarily commutative. Let S_0 be a subset of S satisfying the following.

Assumption 1: There is an efficiently computable function $' : S_0 \rightarrow S$ such that for all $s, t \in S_0$ we have $s(t') = t(s')$.

Assumption 2: There is no efficient algorithm to compute $f(x^{st^\square})$ when x^s and x^t are given.

We note that Assumption 1 is automatically true when S is commutative. For, we can take the identity mapping as $'$. However, such a function does not always exist in general. The Diffie-Hellman assumption is a special case of Assumption 2, where S is $\mathbb{Z}_q$ and f is the identity mapping. We now present the general scheme of a key agreement protocol.

Initialization: Choose $x \in X$ and publicize it.

Step 1: Alice chooses $s \in S_0$ randomly. She computes x^s and transmits it to Bob.

Step 2: Bob chooses $t \in S_0$ randomly. He computes x^t and transmits it to Alice.

Step 3: Alice computes $(x^t)^{s^\square}$ and $f((x^t)^{s^\square})$. Bob computes $(x^s)^{t^\square}$ and $f((x^s)^{t^\square})$. Then $K = f((x^t)^{s^\square}) = f((x^s)^{t^\square})$ is the common secret key shared by Alice and Bob. Note that we have $(x^t)^{s^\square} = (x^s)^{t^\square}$ by Assumption 1.

2.3 ACTDH Protocol

Let G be a group isomorphic to $C \times C$, where C is a cyclic group whose order is a prime p . We now define an action of $\text{GL}(2, \mathbb{Z}_p)$ on the group $G \times G$ ($= (C \times C) \times (C \times C)$). Suppose that a and b generate G , that is, $G = \langle a, b \rangle$. This implies that $|a| = |b| = p$ and $\langle a \rangle \cap \langle b \rangle = \{1_G\}$ in G , where 1_G denotes the identity element of G . Every element of G is uniquely written as $a^i b^j$ with $i, j \in \mathbb{Z}_p$. For $\mathbf{x} = (a^{i_1} b^{j_1}, a^{i_2} b^{j_2}) \in G \times G$ and $A = \begin{pmatrix} w & x \\ y & z \end{pmatrix} \in \text{GL}(2, \mathbb{Z}_p)$, we define an action of $\text{GL}(2, \mathbb{Z}_p)$ on $G \times G$ by $\mathbf{x}^A = (a^{i_1} b^{j_1}, a^{i_2} b^{j_2})^A = (a^{wi_1 + yi_2} b^{wj_1 + yj_2}, a^{xi_1 + zi_2} b^{xj_1 + zj_2})$. Note that $\mathbf{x}^A$ is efficiently computable as follows. Given the elements $a^{i_1} b^{j_1}$ and $a^{i_2} b^{j_2}$ of G and the matrix A , we compute $(a^{i_1} b^{j_1})^w (a^{i_2} b^{j_2})^y = a^{wi_1 + yi_2} b^{wj_1 + yj_2}$, $(a^{i_1} b^{j_1})^x (a^{i_2} b^{j_2})^z = a^{xi_1 + zi_2} b^{xj_1 + zj_2}$. Although, it is straightforward to see that $\text{GL}(2, \mathbb{Z}_p)$ acts on $G \times G$ in this fashion, we give a clearer explanation as follows. If we identify the element $\mathbf{x} = (a^{i_1} b^{j_1}, a^{i_2} b^{j_2})$ and the matrix $\begin{pmatrix} i_1 & i_2 \\ j_1 & j_2 \end{pmatrix}$, then $\mathbf{x}^A$ is identified with the matrix equation:

$$\begin{pmatrix} i_1 & i_2 \\ j_1 & j_2 \end{pmatrix} \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} wi_1 + yi_2 & xi_1 + zi_2 \\ wj_1 + yj_2 & xj_1 + zj_2 \end{pmatrix}.$$

Since the matrix multiplication is associative, $\text{GL}(2, \mathbb{Z}_p)$ acts on $G \times G$.

We now choose randomly parameters $\alpha, \beta, \gamma, \delta$ in $\mathbb{Z}_p^*$ such that $\alpha\beta\gamma\delta$ is a quadratic nonresidue, and then, define S_0 to be the set of matrices of the form $\begin{pmatrix} i_1\alpha & i_2\gamma \\ i_2\beta & i_1\delta \end{pmatrix}$, where $i_1, i_2 \in \mathbb{Z}_p$ with $(i_1, i_2) \neq (0, 0)$. For $A = \begin{pmatrix} i_1\alpha & i_2\gamma \\ i_2\beta & i_1\delta \end{pmatrix} \in S_0$, we define A' to be the matrix $\begin{pmatrix} i_1\delta & i_2\gamma \\ i_2\beta & i_1\alpha \end{pmatrix}$. Note that $A, A' \in \text{GL}(2, \mathbb{Z}_p)$ since $\alpha\beta\gamma\delta$ is a quadratic nonresidue. If $A = \begin{pmatrix} i_1\alpha & i_2\gamma \\ i_2\beta & i_1\delta \end{pmatrix}$ and $B = \begin{pmatrix} j_1\alpha & j_2\gamma \\ j_2\beta & j_1\delta \end{pmatrix}$, then we have $A, B \in S_0$ and

$$A(B') = \begin{pmatrix} i_1 j_1 \alpha \delta + i_2 j_2 \beta \gamma & (i_1 j_2 + i_2 j_1) \alpha \gamma \\ (i_1 j_2 + i_2 j_1) \beta \delta & i_1 j_1 \alpha \delta + i_2 j_2 \beta \gamma \end{pmatrix} = B(A').$$

Therefore, Assumption 1 in the previous section holds. In the rest of the paper, the matrix $\begin{pmatrix} i_1\alpha & i_2\gamma \\ i_2\beta & i_1\delta \end{pmatrix}$ and $\begin{pmatrix} i_1\delta & i_2\gamma \\ i_2\beta & i_1\alpha \end{pmatrix}$ are denoted by $A(i_1, i_2)$ and $A'(i_1, i_2)$, respectively. Hence, we have

$$A(i_1, i_2)A'(j_1, j_2) = A(j_1, j_2)A'(i_1, i_2) \quad (2.1)$$

for all $i_1, i_2, j_1, j_2 \in \mathbb{Z}_p$. Let $\pi_1 : G \times G \rightarrow G$ be the projection into the first coordinate, that is, $\pi_1(a^{i_1} b^{j_1}, a^{i_2} b^{j_2}) = a^{i_1} b^{j_1}$. We note that $\pi_1((\mathbf{x}^{A(j_1, j_2)})^{A^\square(i_1, i_2)}) = a^{\alpha\delta i_1 j_1 + \beta\gamma i_2 j_2} b^{\beta\delta(i_1 j_2 + i_2 j_1)} = \pi_1((\mathbf{x}^{A(i_1, i_2)})^{A^\square(j_1, j_2)})$.

ACTDH protocol: Let $\mathbf{x} = (a, b)$.

- Step 1:** Alice chooses integers i_1, i_2 randomly. She computes $\mathbf{x}^{A(i_1, i_2)} = (a^{\alpha i_1} b^{\beta i_2}, a^{\gamma i_2} b^{\delta i_1})$ and transmits it to Bob.
- Step 2:** Bob chooses integers j_1, j_2 randomly. He computes $\mathbf{x}^{A(j_1, j_2)} = (a^{\alpha j_1} b^{\beta j_2}, a^{\gamma j_2} b^{\delta j_1})$ and transmits it to Alice.
- Step 3:** Alice obtains $K = a^{\alpha \delta i_1 j_1 + \beta \gamma i_2 j_2} b^{\beta \delta (i_1 j_2 + i_2 j_1)}$ by computing $(a^{\alpha j_1} b^{\beta j_2})^{\delta i_1} = a^{\alpha \delta i_1 j_1} b^{\beta \delta i_1 j_2}$ and $(a^{\gamma j_2} b^{\delta j_1})^{\beta i_2} = a^{\beta \gamma i_2 j_2} b^{\beta \delta i_2 j_1}$ and multiplying them. Note that $K = \pi_1((\mathbf{x}^{A(j_1, j_2)})^{A^\square(i_1, i_2)})$.
- Step 4:** Bob similarly computes $K = \pi_1((\mathbf{x}^{A(i_1, i_2)})^{A^\square(j_1, j_2)})$. Then K is the common key for Alice and Bob.

Example We now provide an example of a direct product of two cyclic groups of prime order. Let q and r be large primes such that $p \mid q - 1$ and $p \mid r - 1$. Set n be qr . Let g_1 be a p th root of unity in $\text{mod } q$, and let g_2 be a p th root of unity in $\text{mod } r$. For some $c_1 \in Z_p^*$ and $c_2 \in Z_p^*$, choose $a \in Z_n$ and $b \in Z_n$ such that $a = g_1 \pmod{q}$, $a = g_2^{c_1} \pmod{r}$, $b = g_1^{c_2} \pmod{q}$, $b = g_2 \pmod{r}$. It is easy to see that if $c_1 c_2 \not\equiv 1 \pmod{p}$, then $Z_n^* = \langle a \rangle \times \langle b \rangle$.

2.4 Security Compared with the Diffie-Hellman Protocol

Breaking the ACTDH protocol is equivalent to solving the following problem. Suppose G is a finite abelian group and that $a, b \in G$. Each of the parameters $\alpha, \beta, \gamma, \delta$ is relatively prime to both $|a|$ and $|b|$. The *action Diffie-Hellman problem* (ACTDH problem) in G for a, b is defined by:

INPUT: $(a, b, a^{\alpha i_1} b^{\beta i_2}, a^{\gamma i_2} b^{\delta i_1}, a^{\alpha j_1} b^{\beta j_2}, a^{\gamma j_2} b^{\delta j_1})$.

OUTPUT: $a^{\alpha \delta i_1 j_1 + \beta \gamma i_2 j_2} b^{\beta \delta (i_1 j_2 + i_2 j_1)}$,

where i_1, i_2, j_1, j_2 are randomly and independently chosen integers.

The following result guarantees that the ACTDH protocol is at least as secure as the DH protocol if the parameters are carefully chosen.

Theorem 1. *Let $\alpha, \beta, \gamma, \delta$ be integers. We suppose each of them is relatively prime to $|G|$. If there exists an efficient algorithm solving the ACTDH problem (with the parameters $\alpha, \beta, \gamma, \delta$) in an abelian group G for all a, b in G , then there exists an efficient algorithm solving the DH problem in G for all a in G .*

Proof. Suppose there is an efficient algorithm solving the ACTDH problem for all a and b . We construct an efficient algorithm solving the DH problem, that is, an algorithm that computes $a^{i_1 j_1}$ for the inputs a^{i_1} and a^{j_1} , where a is an element of G . Let $b = 1$ (the identity element of G). We should note that $\alpha, \beta, \gamma, \delta$ are integers relatively prime to $|a|$ since $|a|$ divides $|G|$. By our assumption, we have an efficient algorithm solving the ACTDH problem for a and b . Let $i_2 = j_2 = 0$. We input $(a^{\alpha i_1} b^{\beta i_2}, a^{\gamma i_2} b^{\delta i_1}) = (a^{\alpha i_1}, 1) = ((a^{i_1})^\alpha, 1)$ and $(a^{\alpha j_1} b^{\beta j_2}, a^{\gamma j_2} b^{\delta j_1}) = (a^{\alpha j_1}, 1) = ((a^{j_1})^\alpha, 1)$ to the algorithm solving the ACTDH problem with respect to a and b . Then we obtain $a^{\alpha \delta i_1 j_1 + \beta \gamma i_2 j_2} b^{\beta \delta (i_1 j_2 + i_2 j_1)} = a^{\alpha \delta i_1 j_1}$. We note that we can compute $(a^{i_1})^\alpha$ and $(a^{j_1})^\alpha$ because we are given a^{i_1}, a^{j_1} and α is a public information. Since both α and δ are relatively prime to $|a|$, we can find the integer m such that $(a^{\alpha \delta})^m = a$. Then $(a^{\alpha \delta i_1 j_1})^m = (a^{\alpha \delta m})^{i_1 j_1} = a^{i_1 j_1}$, and hence, the DH problem is efficiently solved.

Let G be a finite abelian group and a, b be elements in G . We set H to be the subgroup of G generated by a and b . The *multiple discrete logarithm problem* (MDLOG for short) in the group $H = \langle a, b \rangle$ is the algorithmic problem defined by:

INPUT: An element g of H .

OUTPUT: A pair (x, y) of non-negative integers such that $g = a^x b^y$.

Since H is generated by a and b , there exists at least one pair (x, y) of non-negative integers satisfying $g = a^x b^y$. Such a pair is uniquely determined only when H is the internal direct product $\langle a \rangle \times \langle b \rangle$. Clearly the ACTDH problem is reduced to the MDLOG problem, and hence, the ACTDH protocol can be broken if the MDLOG is efficiently solved.

3 Generic Reductions and Security

We discuss the security of the ACTDH protocol from the point of view of the generic model. We show the ACTDH protocol with carefully chosen parameters is securer than the DH protocol in the generic model. To simplify the argument, we consider only the ACTDH protocol over a multiplicative group G isomorphic to $\mathbb{Z}_p \times \mathbb{Z}_p$, where p is a large prime in the rest of the paper. We show that the ACTDH protocol is secure even against the adversary who can solve the DLOG if we impose the condition on the parameters $\alpha, \beta, \gamma, \delta$ as follows.

$$\alpha, \beta, \gamma, \delta, \text{ are relatively prime to } p, \quad (\text{C1})$$

$$\alpha\beta\gamma\delta \text{ is a quadratic nonresidue (mod } p). \quad (\text{C2})$$

Suppose that the conditions (C1) and (C2) are satisfied in the rest of the paper. The condition (C1) is imposed to prevent $\alpha, \beta, \gamma, \delta$ from collapsing elements $a, b \in G$. On the other hand, the condition (C2) seems rather artificial. We explain the condition (C2) in Section 3.3.

3.1 Generic Algorithms

A *generic algorithm* is a general-purpose algorithm, which does not rely on any property of the representation of the group (see [10,5]). A generic algorithm enumerates group elements starting from a given set of elements of G ; starting from a set B of generators of G , it enumerates a sequence $g_1, g_2, \dots, g_m$ of elements of G such that $g_m = g$ and $g_i \in B$ or $g_i = g_j^{-1}$ or $g_i = g_j g_k$ for some $j, k < i$ for each i . In a certain stage, the algorithm finds elements g_i and g_j ($i \neq j$) that represent the identical binary string. Then we can obtain information for the hidden information by solving linear equations obtained from the information on g_i and g_j .

Let σ be a random mapping from $\mathbb{Z}_p$ to a set S of binary strings of size p . The generic algorithm is allowed to query the *group operation oracle* that computes the function add and inv defined by: $add(\sigma(x), \sigma(y)) = \sigma(x + y)$ and

$\text{inv}(\sigma(x)) = \sigma(-x)$ for $x, y \in \mathbb{Z}_p$, without any computational cost. Note that $\text{add}(\sigma(x), \sigma(y)) = \sigma(x+y)$ corresponds to the group multiplication, $g^x g^y = g^{x+y}$, and $\text{inv}(\sigma(x)) = \sigma(-x)$ corresponds inversion of a group element, $(g)^{-1} = g^{-1}$. A generic algorithm for the DLOG in the cyclic group $\mathbb{Z}_p$ takes $(\sigma(1), \sigma(x))$ as an input and outputs x , where $x \in \mathbb{Z}_p$. We note that in [5] the Diffie-Hellman oracle is introduced to study the generic reduction of the DLOG to the DH.

3.2 Generic Reductions

We now investigate the hardness of breaking the ACTDH protocol compared with the DLOG problem in terms of the generic reduction. A generic algorithm for ACTDH problem runs as follows. The group $\mathbb{Z}_p \times \mathbb{Z}_p$ (p is a prime) is encoded by σ into a set S of binary strings. A generic algorithm takes a list $(\sigma(1, 0), \sigma(0, 1), \sigma(\alpha i_1, \beta i_2), \sigma(\gamma i_2, \delta i_1), \sigma(\alpha j_1, \beta j_2), \sigma(\gamma j_2, \delta j_1))$ as an input, computes by calling the group operation oracles and then outputs $\sigma(\alpha \delta i_1 j_1 + \beta \gamma i_2 j_2, \beta \delta (i_1 j_2 + i_2 j_1))$. We note that the input corresponds to group elements $a, b, a^{\alpha i_1} b^{\beta i_2}, a^{\gamma i_2} b^{\delta i_1}, a^{\alpha j_1} b^{\beta j_2}, a^{\gamma j_2} b^{\delta j_1}$, and the output corresponds to the group element $a^{\alpha \delta i_1 j_1 + \beta \gamma i_2 j_2} b^{\beta \delta (i_1 j_2 + i_2 j_1)}$. In addition to the group operation oracles, we allow the generic algorithm calling the discrete logarithm oracle. A *discrete logarithm (DLOG) oracle* for $\mathbb{Z}_p \times \mathbb{Z}_p$ takes the pair $(\sigma(i_1, i_2), \sigma(j_1, j_2))$ as an input and then outputs the integer n such that $n i_1 = j_1 \pmod{p}$ and $n i_2 = j_2 \pmod{p}$ without any computation cost if such n exists. If no n satisfying the equations above exists, then we call such an input *illegal*. We assume the oracle does nothing to illegal inputs and the generic algorithm proceeds.

Theorem 2. *Let $\mathcal{A}$ be a generic algorithm solving the ACTDH problem in the group $\mathbb{Z}_p \times \mathbb{Z}_p$, where p is a prime. The parameters $\alpha, \beta, \gamma, \delta$ satisfy the conditions (C1) and (C2). Suppose $\mathcal{A}$ makes at most R queries to the group operation oracle and at most L queries to the DLOG oracle, respectively. Then the probability θ that $\mathcal{A}$ returns the correct answer is at most $\frac{2L(R+6)(R+5)}{p} + \frac{(R+6)(R+5)}{p} + \frac{4(R+6)}{p^2}$, where the probability is taken over i_1, i_2, j_1, j_2 and a representation σ . The expected number of queries to the DLOG oracle is at least $\frac{p^\theta}{2(R+6)(R+5)} - \frac{1}{2} - \frac{2}{p(R+5)}$.*

We first discuss the consequences of Theorem 2. Let T denote the total running time of $\mathcal{A}$. Since $T \geq L + R$, we have $T \geq L$ and $T \geq R$. Suppose θ is a constant. By Theorem 2, we have $\frac{(2T+1)(T+6)(T+5)}{p} + \frac{4(T+6)}{p^2} \geq \theta$. Therefore, T is in $\Omega(\sqrt[3]{p}) = \Omega(2^{\frac{\log p}{3}})$. This implies that there exists no probabilistic polynomial time algorithm that breaks the ACTDH protocol even if the DLOG oracle is allowed.

Next suppose that the DLOG oracle is not available. The expected number of queries to the group operation oracle for solving the ACTDH problem is derived from Theorem 2 by letting $L = 0$. An upper bound of the success probability θ is $\frac{(R+6)(R+5)}{p} + \frac{4(R+6)}{p^2}$, and hence, the expected number of queries to the group operation oracle is estimated at $\Omega(\sqrt{p})$ if θ is a constant.

We now prove Theorem 2. The following is significant in the proof below and is proved in [9].

Lemma 1 ([9]). *Given a non-zero polynomial F in $\mathbb{Z}_p[X_1, X_2, \dots, X_k]$ (p is a prime) of total degree d , the probability that $F(x_1, x_2, \dots, x_k) = 0$ for independently and randomly chosen elements $x_1, x_2, \dots, x_k$ of $\mathbb{Z}_p$ is at most $\frac{d}{p}$.*

Proof of Theorem 2. We simulate a generic algorithm by polynomials over $\mathbb{Z}_p$. At the beginning, we have six pairs of polynomials $(F_1, H_1) = (1, 0)$, $(F_2, H_2) = (0, 1)$, $(F_3, H_3) = (\alpha X_1, \beta X_2)$, $(F_4, H_4) = (\gamma X_2, \delta X_1)$, $(F_5, H_5) = (\alpha Y_1, \beta Y_2)$, $(F_6, H_6) = (\gamma Y_2, \delta Y_1)$ in the ring $\mathbb{Z}_p[X_1, X_2, Y_1, Y_2]$. Each pair corresponds to the representations (of the group elements) $\sigma(1, 0)$, $\sigma(0, 1)$, $\sigma(\alpha i_1, \beta i_2)$, $\sigma(\gamma i_2, \delta i_1)$, $\sigma(\alpha j_1, \beta j_2)$, $\sigma(\gamma j_2, \delta j_1)$, respectively. We compute polynomials $F_i(X_1, X_2, Y_1, Y_2)$ and $H_i(X_1, X_2, Y_1, Y_2)$ for $i > k, l$ so that the pair (F_i, H_i) of polynomials corresponds to the pair $\sigma(F_i(i_1, i_2, j_1, j_2), H_i(i_1, i_2, j_1, j_2))$ of representations (of the group elements). When the multiplication oracle is called with the inputs corresponding to the pairs (F_k, H_k) and (F_l, H_l) , we compute polynomials F_i and H_i by setting $F_i = F_k + F_l$ and $H_i = H_k + H_l$ where $i > k, l$. Similarly, when the inversion oracle is called with the inputs corresponding to the pair (F_k, H_k) , we compute polynomials $F_i = -F_k$ and $H_i = -H_k$ where $i > k, l$. When the DLOG oracle is called with the inputs corresponding to (F_k, H_k) and (F_l, H_l) , it returns $s \in \mathbb{Z}_p$ such that $sF_k(i_1, i_2, j_1, j_2) = F_l(i_1, i_2, j_1, j_2)$ and $sH_k(i_1, i_2, j_1, j_2) = H_l(i_1, i_2, j_1, j_2)$ if such s exists. In this case, we do not produce polynomials, but we get the information that i_1, i_2, j_1, j_2 satisfy the equations $sF_k = F_l$ and $sH_k = H_l$. We suppose that a generic algorithm has a chance to return the correct answer only when we find non-trivial equations satisfied by i_1, i_2, j_1, j_2 in our simulation of the computation.

When the generic algorithm calls the DLOG oracle for the inputs $\sigma(F_k(i_1, i_2, j_1, j_2), H_k(i_1, i_2, j_1, j_2))$ and $\sigma(F_l(i_1, i_2, j_1, j_2), H_l(i_1, i_2, j_1, j_2))$, there are three possible cases. The first possible case is that the inputs are illegal, that is, the second input is not a power of the first. The second case is that the inputs are legal but the polynomials F_k, H_k, F_l, H_l satisfy the condition $F_k H_l = H_k F_l \pmod{p}$ as a polynomial over $\mathbb{Z}_p$. The third case is that the inputs are legal and $F_k H_l \neq H_k F_l \pmod{p}$. We show that information on i_1, i_2, j_1, j_2 can be derived only in the last case. If the first case occurs, the DLOG oracle does not return anything except for an error message. We have no chance to gain the information on i_1, i_2, j_1, j_2 other than that the second is not a power of the first. We now discuss the second case. Let us suppose that $F_k H_l - F_l H_k = 0 \pmod{p}$. First we note that since F_k, H_k, F_l, H_l are polynomials of at most degree 1 over $\mathbb{Z}_p$, they are units or irreducible polynomials. Since the polynomial ring $\mathbb{Z}_p[X_1, X_2, Y_1, Y_2]$ is a unique factorization domain, we have either $uF_k = F_l$ and $uH_k = H_l$ for some $u \in \mathbb{Z}_p$ or $uF_k = H_k$ and $uF_l = H_l$ for some $u \in \mathbb{Z}_p$. In the case that $uF_k = F_l$ and $uH_k = H_l$, the DLOG oracle returns $u \in \mathbb{Z}_p$ to the inputs $\sigma(F_k, H_k)$ and $\sigma(F_l, H_l)$, but we do not obtain any information on i_1, i_2, j_1, j_2 because the equations $uF_k = F_l$ and $uH_k = H_l$ are satisfied not only by i_1, i_2, j_1, j_2 but also by all $x_1, x_2, y_1, y_2 \in \mathbb{Z}_p$. Next we suppose that $uF_k = H_k$ and $uF_l = H_l$. By the

definition of F_k and H_k , we can write $F_k = c_1 + c_3\alpha X_1 + c_4\gamma X_2 + c_5\alpha Y_1 + c_6\gamma Y_2$ and $H_k = c_2 + c_3\beta X_2 + c_4\delta X_1 + c_5\beta Y_2 + c_6\delta Y_1$ where $c_1, c_2, c_3, c_4, c_5, c_6 \in \mathbb{Z}_p$. Since $uF_k = H_k$, we have $uc_1 = c_2$ and

$$\begin{pmatrix} u\alpha & -\delta \\ \beta & -u\gamma \end{pmatrix} \begin{pmatrix} c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \pmod{p}, \quad \begin{pmatrix} u\alpha & -\delta \\ \beta & -u\gamma \end{pmatrix} \begin{pmatrix} c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \pmod{p}.$$

Because of (C2), the matrix $\begin{pmatrix} u\alpha & -\delta \\ \beta & -u\gamma \end{pmatrix}$ is non-singular. Hence, we have $c_3 = c_4 = c_5 = c_6 = 0 \pmod{p}$ and so both F_k and H_k are constants. It follows that F_l and H_l are constants. Therefore, the oracle call with an input $\sigma(F_k, H_k)$ and $\sigma(F_l, H_l)$ such that $F_k H_l = F_l H_k$ does not provide any information on i_1, i_2, j_1, j_2 . Consequently, we can obtain information on i_1, i_2, j_1, j_2 only when the third case occurs and so we say that a DLOG oracle query is *meaningful* if it is called in the third case, otherwise it is *meaningless*.

We now find an upper bound of the probability that $\mathcal{A}$ returns the correct answer. There are three probable cases for a generic algorithm to return the correct answer. (Case 1) At least one DLOG oracle query is meaningful. (Case 2) All DLOG oracle queries are meaningless and there are (F_k, H_k) and (F_l, H_l) such that $(F_k, H_k) \neq (F_l, H_l)$ as polynomials over $\mathbb{Z}_p$, but $F_k(i_1, i_2, j_1, j_2) = F_l(i_1, i_2, j_1, j_2)$ and $H_k(i_1, i_2, j_1, j_2) = H_l(i_1, i_2, j_1, j_2)$. (Case 3) All DLOG oracle queries are meaningless and we have $(F_k(i_1, i_2, j_1, j_2), H_k(i_1, i_2, j_1, j_2)) = (\alpha\delta i_1 j_1 + \beta\gamma i_2 j_2, \beta\delta(i_1 j_2 + i_2 j_1))$ for some (F_k, H_k) . We find an upper bound on the probability in each of (Case 1), (Case 2) and (Case 3).

(Case 1) The probability that a query to a DLOG oracle is meaningful is bounded by the probability that for some k and l with $F_l H_k - F_k H_l \neq 0$ and some s in $\mathbb{Z}_p$ we have $sF_k(x_1, x_2, y_1, y_2) = F_l(x_1, x_2, y_1, y_2)$ and $sH_k(x_1, x_2, y_1, y_2) = H_l(x_1, x_2, y_1, y_2)$ for randomly chosen x_1, x_2, y_1, y_2 in $\mathbb{Z}_p$. Then the probability is bounded by an upper bound of the probability that for randomly chosen x_1, x_2, y_1, y_2 in $\mathbb{Z}_p$, we have an equation $F_l(x_1, x_2, y_1, y_2)H_k(x_1, x_2, y_1, y_2) = F_k(x_1, x_2, y_1, y_2)H_l(x_1, x_2, y_1, y_2)$

since we have the equations:

$$\begin{aligned} sF_k(x_1, x_2, y_1, y_2)H_k(x_1, x_2, y_1, y_2) &= F_l(x_1, x_2, y_1, y_2)H_k(x_1, x_2, y_1, y_2), \\ sF_k(x_1, x_2, y_1, y_2)H_k(x_1, x_2, y_1, y_2) &= F_k(x_1, x_2, y_1, y_2)H_l(x_1, x_2, y_1, y_2). \end{aligned}$$

The probability that for randomly chosen x_1, x_2, y_1, y_2 from $\mathbb{Z}_p$, we have $F_l(x_1, x_2, y_1, y_2)H_k(x_1, x_2, y_1, y_2) = F_k(x_1, x_2, y_1, y_2)H_l(x_1, x_2, y_1, y_2)$ is bounded by $2/p$ by Lemma 3.1 since the total degree of the polynomials $F_l H_k - F_k H_l$ does not exceed two and $F_l H_k - F_k H_l \neq 0$ as a polynomial. It follows that the probability that at least one DLOG oracle query is meaningful is bounded by $L(R+6)(R+5) \times \frac{2}{p}$. (Case 2) Assume that $(F_k, H_k) \neq (F_l, H_l)$. There are three cases: (i) $F_k \neq F_l$ (we do not care whether $H_k \neq H_l$ or $H_k = H_l$) and (ii) $F_k = F_l$ and $H_k \neq H_l$. In the case (i), the probability that $F_k(i_1, i_2, j_1, j_2) = F_l(i_1, i_2, j_1, j_2)$ for ran-

domly chosen x_1, x_2, y_1, y_2 in $\mathbb{Z}_p$ is at most $\frac{1}{p}$ by Lemma 3.1. Hence, the probability that $F_k(i_1, i_2, j_1, j_2) = F_l(i_1, i_2, j_1, j_2)$ and $H_l(i_1, i_2, j_1, j_2) = H_l(i_1, i_2, j_1, j_2)$ for some k, l and randomly chosen x_1, x_2, y_1, y_2 in $\mathbb{Z}_p$ is $\frac{(R+6)(R+5)}{2} \times \frac{1}{p}$ in the case (i). Similarly the probability in the case (ii) is at most $\frac{(R+6)(R+5)}{2} \times \frac{1}{p}$. Therefore the probability in (Case 2) is at most $\frac{(R+6)(R+5)}{p}$. (Case 3) By Lemma 3.1, an upper bound is $(R+6) \times (\frac{2}{p})^2$ for the probability of the event that for randomly chosen x_1, x_2, y_1, y_2 in $\mathbb{Z}_p$, we have $(F_k(x_1, x_2, y_1, y_2), H_k(x_1, x_2, y_1, y_2)) = (\alpha\delta x_1 y_1 + \beta\gamma x_2 y_2, \beta\delta(x_1 y_2 + x_2 y_1))$ for some (F_k, H_k) because the total degrees of the polynomials $F_k - \alpha\delta X_1 Y_1 + \beta\gamma X_2 Y_2$ and $H_k - \beta\delta(X_1 Y_2 + X_2 Y_1)$ are two. We note that $\alpha\delta \neq 0$, $\beta\gamma \neq 0$ and $\beta\delta \neq 0 \pmod{p}$ by the condition (C1). Consequently, the probability that a generic algorithm outputs the correct answer is at most $L(R+6)(R+5) \times \frac{2}{p} + \frac{(R+6)(R+5)}{p} + (R+6) \times (\frac{2}{p})^2$.

We next discuss the generic reduction of MDLOG to DLOG. The following result indicates that the MDLOG cannot be solved without a large number of queries to the DLOG oracle. For the group $\mathbb{Z}_p \times \mathbb{Z}_p$, where p is a large prime, a generic algorithm for the MDLOG takes $(\sigma(1, 0), \sigma(0, 1), \sigma(x_1, x_2))$ as an input and outputs (x_1, x_2) , where x_1, x_2 are integers with $0 \leq x_1 < p$ and $0 \leq x_2 < p$.

Theorem 3. *Let $\mathcal{A}$ be a generic algorithm that solves the MDLOG in G . Suppose that $\mathcal{A}$ makes at most R queries to the group operation oracle and at most L queries to the DLOG oracle, respectively. Then the probability θ that $\mathcal{A}$ returns the correct answer is at most $\frac{2L(R+3)(R+2)}{p} + \frac{(R+3)(R+2)}{p} + \frac{1}{p^2}$ where the probability is taken over i_1, i_2, j_1, j_2 and a representation σ . The expected number of the DLOG oracle queries is at least $\frac{p\theta}{2(R+3)(R+2)} - \frac{1}{2} - \frac{2}{p(R+2)}$.*

3.3 Improper Parameters

The condition (C2) given in Section 3 is vital. If $\alpha\beta\gamma\delta$ is a quadratic residue $\pmod{p}$, then there exists an attack against the ACTDH protocol by using the DLOG oracle.

Attack against ACTDH protocol with improper parameters:

Suppose that $\alpha\beta\gamma\delta = u^2 \pmod{p}$ for some u . Then the matrix $\begin{pmatrix} u\alpha & -\delta \\ \beta & -u\gamma \end{pmatrix}$ is singular, and hence, the system of equations

$$\begin{pmatrix} u\alpha & -\delta \\ \beta & -u\gamma \end{pmatrix} \begin{pmatrix} s \\ t \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \pmod{p}$$

has a nontrivial solution. Suppose that $(s, t) = (c_3, c_4)$ is a nontrivial solution. We are given group elements $a, b, a^{\alpha i_1} b^{\beta i_2}, a^{\gamma i_2} b^{\delta i_1}$ and so we can compute $(a^{\alpha i_1} b^{\beta i_2})^{c_3} (a^{\gamma i_2} b^{\delta i_1})^{c_4} = a^{c_3 \alpha i_1 + c_4 \gamma i_2} b^{c_3 \beta i_2 + c_4 \delta i_1}$. By the definition of c_3, c_4 , we have $u(c_3 \alpha i_1 + c_4 \gamma i_2) = c_3 \beta i_2 + c_4 \delta i_1$. We have obtained $(ab^u)^{c_3 \alpha i_1 + c_4 \gamma i_2}$. We compute ab^u and then call the DLOG oracle with the inputs $(ab^u)^{c_3 \alpha i_1 + c_4 \gamma i_2}$ and ab^u . The oracle returns $h_1 = c_3 \alpha i_1 + c_4 \gamma i_2$. We then do a similar process

with another (c'_3, c'_4) and obtain $h'_1 = c'_3\alpha i_1 + c'_4\gamma i_2$. Then we may be able to obtain i_1 and i_2 . Likewise the adversary can obtain j_1 and j_2 .

4 Asymmetric Encryption Scheme

The ACTDH protocol is applied to construct a public key cryptosystem as follows. Let G be a group isomorphic to the direct product $C \times C$ where C is the cyclic group of prime order p . Suppose that $G = \langle a, b \rangle$. The action of $\text{GL}(2, \mathbb{Z}_p)$ on $G \times G$ is given in Section 2.3. Recall that $A(i_1, i_2)$ and $A'(i_1, i_2)$ are the matrices $\begin{pmatrix} i_1\alpha & i_2\gamma \\ i_2\beta & i_1\delta \end{pmatrix}$ and $\begin{pmatrix} i_1\delta & i_2\gamma \\ i_2\beta & i_1\alpha \end{pmatrix}$, respectively.

Key generation: Let $\alpha, \beta, \gamma, \delta$ be parameters satisfying (C1) and (C2). Let $\mathbf{x} = (a, b) \in G \times G$. Bob chooses $i_1, i_2 \in \mathbb{Z}_p$ randomly. Then he computes $\mathbf{y} = \mathbf{x}^{A(i_1, i_2)} = (a^{i_1\alpha}b^{i_2\beta}, a^{i_2\gamma}b^{i_1\delta})$, and publicizes the key $(\mathbf{x}, \mathbf{y})$.

Encryption: Alice chooses $j_1, j_2 \in \mathbb{Z}_p$ randomly. Then she encrypts the message $m = a^sb^t \in G$ as $(m\pi_1(\mathbf{y}^{A(j_1, j_2)}), \mathbf{x}^{A(j_1, j_2)})$.

Decryption: Obtaining the ciphertext $(m\pi_1(\mathbf{y}^{A(j_1, j_2)}), \mathbf{x}^{A(j_1, j_2)})$, Bob computes the group element $\pi_1((\mathbf{x}^{A(j_1, j_2)})^{A^\square(i_1, i_2)})$ and its inverse. Then he can obtain m as $m\pi_1(\mathbf{y}^{A(j_1, j_2)})(\pi_1((\mathbf{x}^{A(j_1, j_2)})^{A^\square(i_1, i_2)}))^{-1}$ since the group element $(\mathbf{x}^{A(i_1, i_2)})^{A^\square(j_1, j_2)}$ coincides with $(\mathbf{x}^{A(j_1, j_2)})^{A^\square(i_1, i_2)}$.

Let G be a group isomorphic to the direct product of two cyclic group of order prime p . Suppose $\{a, b\}$ is a set of generators of G . We set $\mathbf{x} = (a, b)$. Suppose $\mathcal{A}$ is a probabilistic polynomial time algorithm and for an input $(\mathbf{x}^{A(i_1, i_2)}, \mathbf{x}^{A(j_1, j_2)}, \mathbf{x}^M)$, where i_1, i_2, j_1, j_2 are chosen randomly and $M \in \text{GL}(2, \mathbb{Z}_p)$, $\mathcal{A}$ outputs 1 if $\mathbf{x}^{A(i_1, i_2)A^\square(j_1, j_2)} = \mathbf{x}^M$ and outputs 0 otherwise, with probability better than $\frac{1}{2} + \frac{1}{n^c}$ for some constant c for large enough n . Then $\mathcal{A}$ is called a *DACTDH algorithm*. The *DACTDH assumption* is to assume that there is no DACTDH algorithm. The DACTDH assumption seems weaker than the DH assumption since there is no trivial application of the DLOG oracle to solve the DACTDH problem while the DDH problem is easily solved using the DLOG oracle.

Theorem 4. *The proposed encryption scheme is not secure in the sense of indistinguishability if and only if there exists a DACTDH algorithm.*

The proof of the theorem above is omitted. The conclusion is that the proposed encryption scheme is semantic secure against passive attacks if and only if DACTDH is intractable.

References

1. L.Babai and E.Szemerédi On the complexity of matrix group problems, IEEE Symp. Found. of Computer Science (1984) 229–240. 208
2. W.Diffie and M. E.Hellman, New directions in cryptography, *IEEE Transactions on Information Theory*, 22 (1976) 644–654. 209

3. T.ElGamal, A public key cryptosystem and a signature scheme based on discrete logarithms, *IEEE Transactions on Information Theory*, 31 (1985) 469–472. 209
4. M.Fischlin, A note on security proofs in the generic model, *Advances in Cryptology (Asiacrypt'00) Lecture Notes in Computer Science*, vol 1976 Springer-Verlag (2000) 458–469. 208
5. U. M.Maurer and S.Wolf, Lower bounds on generic algorithms in groups, *Advances in Cryptology (Eurocrypt'98) Lecture Notes in Computer Science*, vol 1403 Springer-Verlag (1998) 72–84. 208, 212, 213
6. V. I.Nechaev, Complexity of a determinate algorithm for the discrete logarithm, *Math. Notes*, 55 (1994) 165–172. 208
7. C. P.Schnorr, Small generic hardcore subsets for the discrete logarithm: Short secret DL-keys, *Infor. Proc. Letters*, 79 (2001) 93–98. 208
8. C. P.Schnorr and M.Jakobsson, Security of signed ElGamal encryption, *Advances in Cryptology (Asiacrypt'00) Lecture Notes in Computer Science*, vol 1976 Springer-Verlag (2000) 73–89. 208
9. J. T.Schwartz, Fast probabilistic algorithms for verification of polynomial identities, *J. ACM*, 27(4) (1980) 701–717. 214
10. V.Shoup, Lower bounds for discrete logarithms and related problems, *Advances in Cryptology (Eurocrypt'97) Lecture Notes in Computer Science*, vol 1233 Springer-Verlag (1997) 256–266. 208, 212

Baire Category and Nowhere Differentiability for Feasible Real Functions[★]

Josef M. Breutzmann¹, David W. Juedes², and Jack H. Lutz³

¹ Dept. Math/CS/Physics, Wartburg College
Waverly IA 50677 USA
breutzma@wartburg.edu

² School of EE & CS, Ohio University
Athens, OH 45701, USA
juedes@ohiou.edu

³ Dept. of Comp. Sci., Iowa State University
Ames, IA 50011 USA
lutz@cs.iastate.edu

Abstract. A notion of resource-bounded Baire category is developed for the class $P_{\mathcal{C}[0,1]}$ of all polynomial-time computable real-valued functions on the unit interval. The meager subsets of $P_{\mathcal{C}[0,1]}$ are characterized in terms of resource-bounded Banach-Mazur games. This characterization is used to prove that, in the sense of Baire category, almost every function in $P_{\mathcal{C}[0,1]}$ is nowhere differentiable. This is a complexity-theoretic extension of the analogous classical result that Banach proved for the class $\mathcal{C}[0,1]$ in 1931.

1 Introduction

Baire category and Lebesgue measure provide a structural framework to classify the relative sizes of infinite sets in various spaces. In the context of complexity theory, the space that we are most familiar with is the space of all languages, *i.e.*, the Cantor space. Unfortunately, since most sets of languages of interest (P, NP, etc.) inside of the Cantor space are countable, classical versions of category and measure cannot classify the relative sizes of these sets in any nontrivial way. To remedy this situation, computable versions of category were investigated by Mehlhorn [18] and Lisagor [10], and resource-bounded versions of measure and category were developed by Lutz [11,12,13,14], Fenner [6,7], Mayordomo [16,17], Allender and Strauss [1], Strauss [23], and others. Resource-bounded category and measure have been used successfully to examine the structure of complexity classes in a variety of contexts [2,4,24, etc.]. The recent surveys by Lutz [15] and Ambos-Spies and Mayordomo [3] provide an overview of work in this area.

In contrast to classical complexity theory, the complexity theory of real functions [9] works primarily in the space $\mathcal{C}[0,1]$ consisting of all continuous functions

[★] This research was supported in part by National Science Foundation Grants 9157382, 9610461, and 9988483.

over the closed interval $[0, 1]$. As in the Cantor space, all countable subsets of $\mathcal{C}[0, 1]$ are small (meager, measure 0) in the senses of Baire category and Lebesgue measure. Hence, these classical theories cannot classify sets of computable real functions in any nontrivial way. To remedy this situation, we develop a resource-bounded version of Baire category in $\mathcal{C}[0, 1]$ and use it to investigate the distribution of differentiability in $P_{\mathcal{C}[0, 1]}$, the class of all polynomial-time computable, continuous functions over the closed interval $[0, 1]$.

Let $\mathcal{ND} = \{f \in \mathcal{C}[0, 1] \mid f \text{ is nowhere differentiable}\}$, where $\mathcal{C}[0, 1]$ is the space of all continuous functions $f : [0, 1] \rightarrow \mathbb{R}$. In the nineteenth century, Weierstrass [25] exhibited a function $f \in \mathcal{ND}$. Subsequently, many other such functions have been shown to exist [26, etc.]. In 1931 Banach [5] proved that $\mathcal{ND}$ is a *comeager* subset of $\mathcal{C}[0, 1]$ in the sense of Baire category. That is, $\mathcal{C}[0, 1] - \mathcal{ND}$ is meager. Banach's result implies the result of Weierstrass, since $\mathcal{C}[0, 1]$ is not meager. However, Banach's result says more — it says that any subset of $\mathcal{C}[0, 1]$ that is not meager contains a nowhere differentiable function. Hence, the existence of nowhere differentiable functions with various properties can be shown by direct application of the category result.

As mentioned above, $P_{\mathcal{C}[0, 1]}$ is a countable, and hence meager subset of $\mathcal{C}[0, 1]$. Hence, Banach's result cannot be used to demonstrate the existence of a polynomial-time computable real valued function that is nowhere differentiable. Indeed, Banach's result leaves the possibility that no polynomial-time computable function is nowhere differentiable. However, this is not the case. As shown by Ko [9], certain well-known nowhere differentiable functions are, in fact, polynomial-time computable. Indeed, related results for computable functions were shown much earlier in the work by Myhill [19] and Pour-El and Richards [21]. Here we show that $\mathcal{ND}$ is comeager in $P_{\mathcal{C}[0, 1]}$, a result that implies both Banach's original result [5] and Ko's later result [9] for the polynomial-time computable functions.

The paper is structured as follows. In section 2, we give the necessary preliminary notation and definitions from real analysis, Baire category, and the complexity theory of real functions. In section 3, we define a resource-bounded Baire Category for $\mathcal{C}[0, 1]$. The central definition is that of the p -meager sets in $\mathcal{C}[0, 1]$. As we show, every p -meager set is meager in the classical sense. Following the work of Lutz [12, 13], Fenner [6, 7], and Strauss [23], we also show that the class of p -meager sets forms an ideal of “small” sets. That is, the p -meager sets satisfy the following conditions: (i) subsets of p -meager sets are p -meager, (ii) the p -meager sets are closed under finite unions, (iii) the p -meager sets are closed under effective countable unions, (iv) for each function $f \in P_{\mathcal{C}[0, 1]}$, $\{f\}$ is p -meager, and (v) $P_{\mathcal{C}[0, 1]}$ is not p -meager. In addition, we give a characterization of the p -meager sets in terms of resource-bounded Banach-Mazur games in $\mathcal{C}[0, 1]$. In section 4, we use this characterization to prove our main result, namely, that $\mathcal{ND}[0, 1]$ is p -meager. This implies that the set of polynomial-time computable functions that have a derivative at some point $x \in [0, 1]$ is a negligibly small subset of $P_{\mathcal{C}[0, 1]}$. The proofs of all technical results in section 3 are omitted from this extended abstract.

2 Preliminaries

We begin by presenting the necessary notation and definitions from real analysis, Baire category, and complexity theory of real functions. For a more detailed presentation, see Rudin [22], Oxtoby [20], or Ko [9]. To begin, let $\mathcal{C}[0, 1]$ be the set of continuous real valued functions on the compact domain $[0, 1]$. Given any functions f and g in $\mathcal{C}[0, 1]$, the distance between f and g is

$$d(f, g) = \|f - g\| = \sup_{x \in [0, 1]} |f(x) - g(x)|.$$

It is well-known that $\mathcal{C}[0, 1]$ along with the associated distance function d form a complete metric space.

Since $\mathcal{C}[0, 1]$ is a complete metric space, we will sometimes call a function $f \in \mathcal{C}[0, 1]$ a *point*. For $r > 0$, the *neighborhood of radius r* about the function f is the set $N_r(f)$ containing all functions g such that $d(f, g) < r$. Let S be a subset of $\mathcal{C}[0, 1]$. A function f is a *limit point* of S if for every $r > 0$ there exists a $g \neq f$ such that $g \in N_r(f) \cap S$. If every function f that is a limit point of S is contained in S , then S is *closed*.

Given a sequence $f_0, f_1, \dots, f_n, \dots$ of functions in $\mathcal{C}[0, 1]$, the limit of this sequence is defined point-wise. That is, if the sequence $\{f_n(x)\}$ converges for each $x \in [0, 1]$, then the limit of $\{f_n\}$ is the function f defined by $f(x) = \lim_{n \rightarrow \infty} f_n(x)$. Since $\mathcal{C}[0, 1]$ is a compact space, the limit of a sequence of continuous functions is also continuous.

If there is an $r > 0$ such that $N_r(f) \subseteq S$, then the function f is an *interior point* of S . If every function $f \in S$ is an interior point of S , then S is *open*. If every function in $\mathcal{C}[0, 1]$ is contained in S or a limit point of S (or both), then S is *dense* in $\mathcal{C}[0, 1]$.

According to [8], a set S is *nowhere dense* in $\mathcal{C}[0, 1]$ if and only if for each open set O , $O \cap S$ is not dense in O . Equivalently, a set S is nowhere dense if, for every open set O , there exists an open set $O' \subseteq O$ such that $O' \cap S = \emptyset$. A set S is *meager* (a set of first category) in $\mathcal{C}[0, 1]$ if it is a countable union of a family nowhere dense sets. A set S is *nonmeager* (a set of second category) if it is not meager. A set S is *comeager* (residual) if its complement is meager.

Following the work of Ko [9], we use the dyadic rational numbers $\mathbb{D} = \{m \cdot 2^{-n} | m \in \mathbb{Z} \text{ and } n \in \mathbb{N}\}$ as finite approximations to real numbers. Because the dyadic rational numbers are dense in $\mathbb{R}$, it is possible to define the topology of $\mathcal{C}[0, 1]$ in terms piece-wise linear functions with dyadic rational endpoints. A function $f \in \mathcal{C}[0, 1]$ is a *piece-wise linear function with dyadic rational endpoints* if there exist points $a_0 = 0 < a_1 < a_2 \dots < a_n = 1 \in \mathbb{D}$ such that $f(a_i) \in \mathbb{D}$ and for $a_i < x < a_{i+1}$, $f(x) = f(a_i) + \frac{f(a_{i+1}) - f(a_i)}{a_{i+1} - a_i}(x - a_i)$. A *basic open set* O is a set $O = N_d(f)$, where $d \in \mathbb{D} \cup \{\infty\}$, $d > 0$, and f is a piece-wise linear function with dyadic rational endpoints. It is well-known that a set $S \subseteq \mathcal{C}[0, 1]$ is nowhere dense if and only if for every basic open set O there exists a basic open set $O' \subseteq O$ such that $O' \cap S = \emptyset$.

Here we are primarily interested in $P_{\mathcal{C}[0, 1]}$, the set of functions in $\mathcal{C}[0, 1]$ that are feasibly computable. Using Theorem 2.22 of Ko [9, p. 59], we define $P_{\mathcal{C}[0, 1]}$

to be the set of functions $f \in \mathcal{C}[0,1]$ where there exists a sequence of piecewise linear functions $\{f_n\}$ with dyadic rational endpoints and a polynomial-function m such that

- (i) for each $n \in \mathbb{N}$ and $0 \leq i \leq 2^{m(n)}$, $f_n(\frac{i}{2^{m(n)}}) \in \mathbb{D}$,
- (ii) For each n and $0 \leq i < 2^{m(n)}$, $|f_n(\frac{i}{2^{m(n)}}) - f_n(\frac{i+1}{2^{m(n)}})| \leq 2^{-n}$,
- (iii) for each $n \in \mathbb{N}$ and $x \in [0,1]$, $|f_n(x) - f(x)| \leq 2^{-n}$,
- (iv) the polynomial function $m(n) : \mathbb{N} \rightarrow \mathbb{N}$ is computable in time $p(n)$, and the function $\psi : \mathbb{D} \times \mathbb{N} \rightarrow \mathbb{D}$ defined by $\psi(\frac{i}{2^{m(n)}}, n) = f_n(\frac{i}{2^{m(n)}})$ is computable in time $q(m(n) + n)$. Here, both p and q are polynomials.

Finally, we define $DTIME(n^c)_{\mathcal{C}[0,1]}$ to be the set of all functions $f \in P_{\mathcal{C}[0,1]}$ satisfying the above conditions and the condition that $p(n) + q(m(n) + n) = O(n^c)$.

3 Resource-Bounded Baire Category in $\mathcal{C}[0,1]$

Let $\mathcal{B}$ be the set of all basic open sets. Then, a set $X \subseteq \mathcal{C}[0,1]$ is nowhere dense if there exists a function $\alpha : \mathcal{B} \rightarrow \mathcal{B}$ such that for every basic open set $x \in \mathcal{B}$, $\alpha(x) \subseteq x$ and $\alpha(x) \cap X = \emptyset$. Such a function α “testifies” that X is nowhere dense. Intuitively, such a function α takes a basic open set and creates a refinement of that basic open set that misses X . Similarly, a set $X = \bigcup_{i=1}^{\infty} X_i$ is meager if there exists a function $\alpha' : \mathbb{N} \times \mathcal{B} \rightarrow \mathcal{B}$ such that the function $\alpha'_i(x) = \alpha'(i, x)$ testifies that X_i is nowhere dense.

Since each basic open set has a finite binary representation, a natural approach to resource-bounded Baire category on $\mathcal{C}[0,1]$ might be to require that α' be computable in some resource-bound, e.g., X is Δ -meager if there exists a function α' that testifies that X is meager and α' is computable in the resources given by Δ . Unfortunately, this natural approach does not allow for a reasonable notion of category inside of $P_{\mathcal{C}[0,1]}$ because a basic open set’s finite binary representation may need to be exponentially large. To remedy this situation, we instead examine functions that refine segments of basic open sets. We begin by presenting the necessary definitions.

Definition 1. A neighborhood component code is a 6-tuple $\kappa = \langle n, r, a, b, c, d \rangle$ such that $n, a, b \in \mathbb{N}$, $c, d \in \mathbb{Z}$, $r \in \mathbb{Z} \cup \{\infty\}$, and $0 \leq \frac{a}{2^n} < \frac{b}{2^n} \leq 1$. The neighborhood component corresponding to a neighborhood component code $\kappa = \langle n, a, b, c, d, r \rangle$ is the set $N(\kappa)$ consisting of all functions $f \in \mathcal{C}[0,1]$ such that for all $x \in [\frac{a}{2^n}, \frac{b}{2^n}]$, $\varphi_\kappa(x) - 2^r < f(x) < \varphi_\kappa(x) + 2^r$, where $\varphi_\kappa(x) = \frac{d-c}{b-a}(x - \frac{a}{2^n}) + \frac{c}{2^n}$.

Each basic open set can be viewed as a sequence of consistent neighborhood components. This notion of consistency is defined as follows.

Definition 2. A neighborhood component code $\kappa_1 = \langle n_1, r_1, a_1, b_1, c_1, d_1 \rangle$ meets $\kappa_2 = \langle n_2, r_2, a_2, b_2, c_2, d_2 \rangle$ if $n_1 = n_2$, $r_1 = r_2$, $b_1 = a_2$, and $d_1 = c_2$. A neighborhood code on an interval $[a, b]$ is a finite sequence $\boldsymbol{\kappa} = (\kappa_1, \kappa_2, \dots, \kappa_l)$ of neighborhood component codes such that κ_i meets κ_{i+1} for all $1 \leq i < l$, $\frac{a_1}{2^{n_1}} = a$,

and $\frac{b_l}{2^n} = b$, where n is the common first component of all the κ_i . The neighborhood corresponding to a neighborhood code κ on an interval $[a, b]$ is the set $N(\kappa) = \bigcap_{i=1}^n N(\kappa_i)$.

It is easy to see that every basic open set is the neighborhood corresponding to some neighborhood code κ on $[0, 1]$. In order to define the meager sets, we will need a notion of the refinement of a neighborhood.

Definition 3. A refinement of a neighborhood component code $\kappa = \langle n, r, a, b, c, d \rangle$ is a neighborhood code $\kappa = (\kappa_1, \dots, \kappa_l)$ on $[\frac{a}{2^n}, \frac{b}{2^n}]$ such that $N(\kappa) \subseteq N(\kappa)$ and $r_1 < r$.

Now, let $\mathcal{N}_0$ be the set of all neighborhood component codes, and let $\mathcal{N}$ be the set of all neighborhood codes. A *constructor* is a function $\gamma : \mathcal{N}_0 \rightarrow \mathcal{N}$ such that

- (i) $(\forall \kappa \in \mathcal{N}_0) \gamma(\kappa)$ is a refinement of κ , and
- (ii) γ is *consistent* in the sense that if κ_1 meets κ_2 then the right hand component of $\gamma(\kappa_1)$ meets the left hand component of $\gamma(\kappa_2)$.

Given a constructor α , it is natural to extend the application of α from individual neighborhood component codes to full neighborhood codes. Given a constructor α , define $\bar{\alpha} : \mathcal{N} \rightarrow \mathcal{N}$ by $\bar{\alpha}((\kappa_1, \dots, \kappa_l)) = (\alpha(\kappa_1), \dots, \alpha(\kappa_l))$, where $(\kappa_1, \dots, \kappa_l)$ is the vector containing the individual components (in order) of the vectors $\kappa_1, \dots, \kappa_l$.

Such constructors can be used to testify that sets are nowhere dense.

Theorem 1. Let α be a constructor and let $X \subseteq \mathcal{C}[0, 1]$. If it is the case that $N(\bar{\alpha}(\kappa)) \cap X = \emptyset$ for every neighborhood code κ that corresponds to a basic open set, then X is nowhere dense.

Proof. This is immediate from the fact that α is a constructor.

It is not known whether the converse is true, *i.e.*, that every nowhere dense set has a constructor that testifies that it is nowhere dense. Nevertheless, this approach provides a reasonable notion of category in $P_{\mathcal{C}[0, 1]}$.

To define a notion of resource-bounded Baire Category on $P_{\mathcal{C}[0, 1]}$, we apply resource bounds to our constructors. A constructor γ is computable in polynomial time if the function $\hat{\gamma} : \mathcal{N}_0 \times \mathbb{N} \rightarrow \mathcal{N}_0 \cup \{\perp\}$ defined by

$$\hat{\gamma}(\kappa, i) = \begin{cases} \kappa_i & \text{if } 1 \leq i \leq l \\ \perp & \text{otherwise,} \end{cases}$$

where $\gamma(\kappa) = (\kappa_1, \dots, \kappa_l)$, is computable in time polynomial in $|\kappa| + |i|$. Note that we assume that $\kappa = \langle n, r, a, b, c, d \rangle$ is encoded with n and r represented in unary with an additional sign bit for r . It follows that $|\kappa| \geq n + |r|$.

An indexed constructor is a function $\alpha' : \mathbb{N} \times \mathcal{N}_0 \rightarrow \mathcal{N}$ such that $\alpha'(i, \kappa)$ is a constructor for each $i \in \mathbb{N}$. An indexed constructor α' is computable in polynomial time if $\hat{\alpha}'(i, \kappa, j)$ is computable in time bounded by a polynomial in $|i| + |\kappa| + |j|$. We will use indexed constructors to define a notion of meager sets in $P_{\mathcal{C}[0, 1]}$.

Definition 4. A set X is p -meager if $X = \bigcup_{i=1}^{\infty} X_i$ and there exists a polynomial-time computable indexed constructor α' such that $\alpha'(i, \circ)$ testifies that X_i is nowhere dense. A set X is p -comeager if $\overline{X} = \mathcal{C}[0, 1] - X$ is p -meager. A set X is meager in $P_{\mathcal{C}[0, 1]}$ if $X \cap P_{\mathcal{C}[0, 1]}$ is p -meager. A set X is comeager in $P_{\mathcal{C}[0, 1]}$ if $\overline{X}$ is meager in $P_{\mathcal{C}[0, 1]}$.

Example 1. The set $X = \{f \in \mathcal{C}[0, 1] \mid f(1/4) = f(3/4)\}$ is p -meager. Hence, X is meager in $P_{\mathcal{C}[0, 1]}$.

As shown in the previous example, certain simple sets of functions can be shown to be p -meager using Definition 4. In some cases, it is desirable to work with a modified definition that uses a slightly restricted notion of an indexed constructor. We say that a constructor $\alpha : \mathcal{N}_0 \rightarrow \mathcal{N}$ is q -bounded if there exists a polynomial q such that for every $\kappa \in \mathcal{N}_0$, if $\alpha(\kappa) = (\kappa_1, \dots)$ and $\kappa_1 = \langle n, r, a, b, c, d \rangle$ then $n \leq q(|r|)$. An indexed constructor $\alpha' : \mathbb{N} \times \mathcal{N}_0 \rightarrow \mathcal{N}$ is q -bounded if there exists a single polynomial q such that $\alpha'(i, \circ)$ is q -bounded for every i . Notice that it is easy to prove that $X = \bigcup_{i=1}^{\infty} X_i$ is p -meager if and only if there exists a polynomial-time computable q -bounded indexed constructor α' such that $\alpha'(i, \circ)$ testifies that X_i is nowhere dense.

The rationale for using this modified definition of indexed constructors lies in that fact that constructors implicitly define real valued functions. To see this, begin with a basic open set O and iteratively apply some constructor α . If α is computable, the single function in the intersection of the closures of these basic open sets is a computable function. However, if α is computable in polynomial time such a construction may not produce a polynomial-time computable function unless α is q -bounded.

We next give an equivalent definition of the p -meager sets in terms of resource-bounded Banach-Mazur games.

3.1 Resource-Bounded Banach-Mazur Games

It is well-known [20] that Baire category can be characterized in terms of a two person game of perfect information called the Banach-Mazur game. In this context, a Banach-Mazur game is a two person game where the players alternately restrict a set of viable functions. The game begins with $\mathcal{C}[0, 1]$, the set of all continuous functions on $[0, 1]$, and a set $X \subseteq \mathcal{C}[0, 1]$. Player I begins by producing a basic open set B_1 . Player II then produces a basic open set $B_2 \subseteq B_1$ whose radius decreases by at least one half. The game continues forever with player I and player II alternately restricting the resulting basic open set. The result of the game is the single function f contained in the intersection of the closure of the basic open sets produced during each round of the game. Player I wins if $f \in X$. Player II wins if $f \notin X$. A set X is meager if and only if there is a strategy so that player II always wins on X .

Here we characterize the p -meager sets in terms of Banach-Mazur games where the two players are indexed constructors. Let α and β be indexed constructors, and let $N(\kappa)$ be a basic open set. The k th round of the Banach-Mazur game

$[\alpha, \beta; X]$ consists of applying $\overline{\alpha}(k, \circ)$ to κ and then applying $\overline{\beta}(k, \circ)$ to $\overline{\alpha}(k, \kappa)$. The game starts with $\kappa = (\langle 0, \infty, 0, 1, 0, 0 \rangle)$. The neighborhood corresponding to κ is the neighborhood of radius ∞ about the piecewise linear function $f(x) = 0$. This neighborhood contains all of $\mathcal{C}[0, 1]$. Now, define κ_i as follows.

$$\begin{aligned}\kappa_0 &= \kappa = (\langle 0, \infty, 0, 1, 0, 0 \rangle) \\ \kappa_{2i+1} &= \overline{\alpha}(i, \kappa_{2i}), \quad \kappa_{2i+2} = \overline{\beta}(i, \kappa_{2i+1}).\end{aligned}$$

The result of the game $[\alpha, \beta; X]$ is the unique function $f \in \bigcap_{i=0}^{\infty} \overline{N}(\kappa_i)$, where $\overline{N}(\kappa_i)$ is the closure of the neighborhood corresponding to κ_i . Player I wins if $f \in X$, and player II wins if $f \notin X$.

It is straightforward to prove that if both player I and player II are polynomial-time computable indexed constructors that $f \in P_{\mathcal{C}[0,1]}$.

Theorem 2. *Let α and β be polynomial-time computable q -bounded indexed constructors. Then, the unique function f that is the result of the game $[\alpha, \beta; X]$ is an element of $P_{\mathcal{C}[0,1]}$.*

Similarly, if $f \in P_{\mathcal{C}[0,1]}$, then f is the result of some Banach-Mazur game.

Theorem 3. *If $f \in P_{\mathcal{C}[0,1]}$, then there exist polynomial-time computable q -bounded indexed constructors α and β such that f is the result of the game $[\alpha, \beta; X]$.*

If player II (β) to wins the game $[\alpha, \beta; X]$ for all possible α , then this is equivalent to X being a p -meager set.

Theorem 4. *Let $X \subseteq \mathcal{C}[0, 1]$. The following are equivalent.*

- a. X is p -meager.
- b. There exists a polynomial-time q -bounded indexed constructor β such that player II wins the game $[\alpha, \beta; X]$ for all indexed constructors α .

3.2 Basic Results

We end this section with a collection of basic results concerning the p -meager sets. Following previous work on resource-bounded measure and category [12, 7, 23, etc.], we show that the p -meager sets in $P_{\mathcal{C}[0,1]}$ satisfy those conditions expected for a class of small sets, i.e., the p -meager sets are closed under subset, finite union, and appropriate countable union; each singleton $\{f\}$ for $f \in P_{\mathcal{C}[0,1]}$ is p -meager; and $P_{\mathcal{C}[0,1]}$ is not p -meager. We begin by giving a definition of an appropriate countable union of p -meager sets.

Definition 5. *A p -union of p -meager sets is a set X such that there exists a polynomial-time indexed constructor α and a family of sets $\{X_i | i \in \mathbb{N}\}$ such that*

$$(i) \quad X = \bigcup_{i=1}^{\infty} X_i.$$

- (ii) For each i , the indexed constructor α_i defined by $\alpha_i(j, \kappa) = \alpha(\langle i, j \rangle, \kappa)$ testifies that X_i is p -meager.

Theorem 5. *The following conditions concerning the p -meager sets hold.*

- (i) If X is p -meager and $Y \subseteq X$, then Y is p -meager.
- (ii) If X and Y are p -meager, then $X \cup Y$ is p -meager.
- (iii) If X is a p -union of p -meager sets, then X is p -meager.
- (iv) If $f \in P_{\mathcal{C}[0,1]}$, then $\{f\}$ is p -meager.

Theorem 6. (Baire Category Theorem) $P_{\mathcal{C}[0,1]}$ is not p -meager.

4 Nowhere Differentiability in $P_{\mathcal{C}[0,1]}$

We now present a nontrivial application of the theory of resource-bounded Baire category in $P_{\mathcal{C}[0,1]}$. Here we examine the distribution of differentiability in $P_{\mathcal{C}[0,1]}$. As we show in Theorem 7 below, the set of nowhere differentiable functions, $\mathcal{ND}[0,1]$, is p -comeager and hence comeager in $P_{\mathcal{C}[0,1]}$. This result implies the classical result of Banach and existence of nowhere differentiable functions in $P_{\mathcal{C}[0,1]}$. The proof of Theorem 7 requires the following technical lemma.

Lemma 1. *If $\kappa = \langle n, r, a, b, c, d \rangle$ is a neighborhood component code with central segment L and L' is any segment $\overline{P_1 P_2}$ within κ , i.e., $P_1 = (\frac{a}{2^n}, y_1)$ and $P_2 = (\frac{b}{2^n}, y_2)$ with $|y_1 - \frac{c}{2^n}| < 2^r$ and $|y_2 - \frac{d}{2^n}| < 2^r$, then the slopes m and m' of L and L' respectively satisfy $|m - m'| < 2^{n+r+1}$.*

Theorem 7. $\mathcal{ND}[0,1]$ is p -comeager.

Proof. We define a polynomial-time computable clocked constructor γ with which player II can force the result of a Banach-Mazur game to be an element of $\mathcal{ND}[0,1]$. Hence $\overline{\mathcal{ND}[0,1]}$ is p -meager and $\mathcal{ND}[0,1]$ is p -comeager. In our construction, $\gamma(i, \circ)$ does not depend on the parameter i . Hence, we write $\gamma(\kappa)$ for $\gamma(i, \kappa)$.

Given a neighborhood component code $\kappa = \langle n, r, a, b, c, d \rangle$ we define $\gamma(\kappa)$ as follows: first select the least $n' \in \mathbb{N}$ such that $n' > n$, $n' > |r|$, and

$$2^{n'+r} > 8|r| + 4. \quad (1)$$

Second, select the greatest $r' \in \mathbb{Z}$ such that $r' < r$ and

$$2^{n'+r'+1} < |r|. \quad (2)$$

These choices of n' and r' depend only on n and r and can be done consistently (in polynomial-time) across all of $[0,1]$.

The constructor $\gamma(\kappa)$ creates $l = (b - a) \cdot 2^{n'-n}$ subintervals of width $2^{-n'}$. The structure of new neighborhood components within the subintervals depends on the slope $m = \frac{d-c}{b-a}$ of the central segment of κ . There are two cases.

Case 1: $|m| > 2|r| + 1$. In this case, since the slope is already steeper than $|r|$, we attempt to keep the slope of the central segment in each subinterval as close to m as possible. Define $\hat{\gamma}(\kappa, i) = \kappa'_i = \langle n', r', a'_i, b'_i, c'_i, d'_i \rangle$ as follows. Let n' and r' be as given earlier. For $1 \leq i \leq l$, $a'_i = a \cdot 2^{n^\square - n} + (i - 1)$, $b'_i = a'_i + 1$, $c'_1 = c \cdot 2^{n^\square - n}$, and $d'_l = d \cdot 2^{n^\square - n}$. Note that $\frac{a'_i}{2^{n^\square}} = \frac{a}{2^n}$, $\frac{b'_i}{2^{n^\square}} = \frac{b}{2^n}$, $\frac{c'_1}{2^{n^\square}} = \frac{c}{2^n}$, and $\frac{d'_l}{2^{n^\square}} = \frac{d}{2^n}$. For $2 \leq i \leq l$, let $c'_i = d'_{i-1} = \varphi_1(i - 1)$, where $\varphi_1(i) = c'_1 + \lceil m \cdot i \rceil$. Since $n' > |r|$ and $r' < r$, it follows that these subintervals lie within κ . Moreover, the slope of the central segment for each subinterval is

$$m' = \frac{d'_i - c'_i}{b'_i - a'_i} = \varphi_1(i) - \varphi_1(i - 1) = \lceil m \cdot i \rceil - \lceil m(i - 1) \rceil.$$

Since $x \leq \lceil x \rceil < x + 1$, it is easy to show that $m - 1 < m' < m + 1$. It follows that $|m'| > 2|r|$.

By Lemma 1, the slope m'' for any segment L'' within $\kappa'_i = \langle n', r', a'_i, b'_i, c'_i, d'_i \rangle$ will differ from m' , the slope of the central segment by at most $|m'' - m'| < 2^{n^\square + r^\square + 1}$. Since $2^{n^\square + r^\square + 1} < |r|$, we have $|m'' - m'| < |r|$. Since $|m'| > 2|r|$, it follows that $|m''| > |r|$.

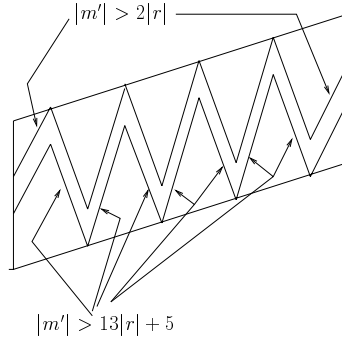


Fig. 1. The sawtooth neighborhood from Case 2

Case 2: As seen in Figure 1, if $|m| \leq 2|r| + 1$, to make the slopes of the refinement steeper than $|r|$ we introduce a sawtooth neighborhood inside of κ so that the absolute value of the slope of the central segment of each component is at least $2|r|$. As before, the original neighborhood component code is broken down into $l = (b - a) \cdot 2^{n^\square - n}$ subintervals of width $2^{-n^\square}$. For each $1 \leq i \leq l$, let $a'_i = a \cdot 2^{n^\square - n} + (i - 1)$ and $b'_i = a'_i + 1$. As before, set $c'_1 = c \cdot 2^{n^\square - n}$ and $d'_l = d \cdot 2^{n^\square - n}$. This provides consistency with the neighboring segments. For $2 \leq i \leq l - 1$, let

$$d'_{i-1} = c'_i = \varphi_2(i) = \begin{cases} c'_1 + \lfloor m(i - 1) + 2^{n^\square + r} - 2^{n^\square + r^\square} \rfloor & \text{if } i \text{ is even} \\ c'_1 + \lceil m(i - 1) - 2^{n^\square + r} + 2^{n^\square + r^\square} \rceil & \text{if } i \text{ is odd.} \end{cases}$$

Notice that this definition places the neighborhood $\gamma(\kappa)$ inside of κ .

Now let's examine the slopes of the central segments in each subinterval. When $2 \leq i \leq l-1$ and i is odd, the slope of the central segment of κ'_i is

$$\begin{aligned} m' &= \frac{d'_i - c'_i}{b'_i - a'_i} = d'_i - c'_i = \lfloor m \cdot i + 2^{n^\square+r} - 2^{n^\square+r^\square} \rfloor - \lceil m(i-1) - 2^{n^\square+r} + 2^{n^\square+r^\square} \rceil \\ &= \lfloor m \cdot i - 2^{n^\square+r^\square} \rfloor - \lceil m(i-1) + 2^{n^\square+r^\square} \rceil + 2^{n^\square+r+1} \end{aligned}$$

The final equality holds because $2^{n^\square+r}$ is an integer whenever $n' > |r|$. Moreover, because $x-1 < \lfloor x \rfloor \leq x$ and $x \leq \lceil x \rceil < x+1$, it is easy to show that $m+2^{n^\square+r+1} - 2^{n^\square+r^\square+1} - 2 < m' \leq m+2^{n^\square+r+1} - 2^{n^\square+r^\square+1}$. Since $2^{n^\square+r} > 8|r|+4$, $2^{n^\square+r^\square+1} < |r|$, and $|m| \leq 2|r|+1$, it follows that $m' > 13|r|+5$. Similarly, we can show that $m' < -13|r|-5$ when $2 \leq i \leq l-1$ and i is even.

When $i = 1$, the slope of the central segment of κ'_1 is

$$m' = \frac{d'_1 - c'_1}{b'_1 - a'_1} = d'_1 - c'_1 = \lfloor m + 2^{n^\square+r} - 2^{n^\square+r^\square} \rfloor = \lfloor m - 2^{n^\square+r^\square} \rfloor + 2^{n^\square+r}.$$

Because $r' < r$, it is easy to show that $m-1+2^{n^\square+r-1} < m' \leq m+2^{n^\square+r}$. Since $|m| \leq 2|r|+1$ and $2^{n^\square+r-1} > 4|r|+2$, it follows that $m' > 2|r|$. Similarly, we can show that $|m'| > 2|r|$ when $i = l$.

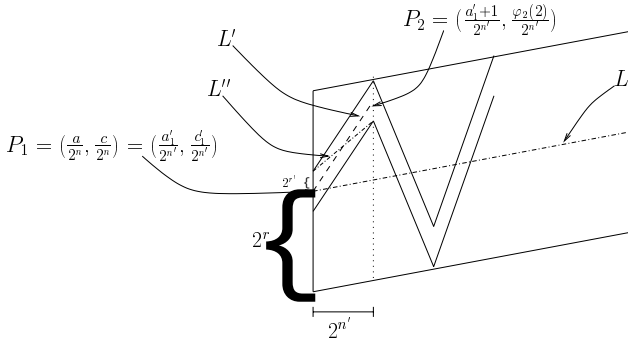


Fig. 2. A closer view of Case 2

Consider a segment L'' with slope m'' inserted into one of the neighborhoods for these l subintervals, e.g., see Figure 2. If we apply Lemma 1 to the neighborhood κ'_i , we have $|m'' - m'| < 2^{n^\square+r+1}$. Since $|m'| > 2|r|$ and $2^{n^\square+r+1} < |r|$, it follows that $|m''| > |r|$.

Thus, in either case, we have that any segment lying “within” a neighborhood component code κ'_i of $\gamma(\langle n, r, a, b, c, d \rangle)$ has slope that exceeds r in absolute value. We now complete the proof through the use of two claims.

Claim. γ is polynomial-time computable.

Proof. Given n, r , we can find n' and r' satisfying (1) and (2) in polynomial-time in the size of n and r . In addition, we can compute the near linear functions $\varphi_1(i)$ and $\varphi_2(i)$ in polynomial-time in the size of the input and $|i|$, and hence we can compute $\kappa_i = \hat{\gamma}(\kappa, i)$ in polynomial time.

Claim. If $f(x) : [0, 1] \rightarrow \mathbb{R}$ is the result of a Banach-Mazur game in which player II uses strategy γ , then $f \in \mathcal{ND}[0, 1]$. Hence, $\overline{\mathcal{ND}[0, 1]}$ is p -meager and $\mathcal{ND}[0, 1]$ is p -comeager.

Proof. Let $x \in [0, 1]$, $\epsilon > 0$, $M > 0$ be given, and let $f(x)$ be the result of the a Banach-Mazur game in which player II used strategy γ . Since $n, |r| \rightarrow \infty$, at some point during the game there must have been a neighborhood code κ given to player II with a component code κ such that x lies in $\kappa'_i = \gamma(\kappa, i) = \langle n', r', a'_i, b'_i, c'_i, d'_i \rangle$ with $2^{-n^\square} < \epsilon$, $|r| > M$, and $\frac{a_i^\square}{2^{n^\square}} \leq x \leq \frac{b_i^\square}{2^{n^\square}}$.

Now, let $P_1 = (\frac{a_i^\square}{2^{n^\square}}, f(\frac{a_i^\square}{2^{n^\square}}))$, $P = (x, f(x))$, and $P_2 = (\frac{b_i^\square}{2^{n^\square}}, f(\frac{b_i^\square}{2^{n^\square}}))$. By the construction of γ , the slope of m of $\overline{P_1 P_2}$ must satisfy $|m| > |r| > M$. By the triangle inequality, if m_1 is the slope of $\overline{P_1 P}$ and m_2 is the slope of $\overline{P P_2}$, one of m_1 or m_2 must satisfy $|m_i| > |r| > M$. Hence, P_1 or P_2 provides a point which yields a difference quotient whose absolute value exceeds M at x . So, f fails to be differentiable at x since M and ϵ were arbitrary. Further, x was arbitrary, and so $f \in \mathcal{ND}[0, 1]$.

Since player II forced f into $\mathcal{ND}[0, 1]$ via γ , $\mathcal{ND}[0, 1]$ is p -comeager. This completes the proof of Theorem 7.

Corollary 1. (*Banach [5]*) $\mathcal{ND}[0, 1]$ is comeager.

Proof. This follows from that fact that every p -meager set is meager.

Corollary 2. (*Ko [9]*) There exists a function $f \in P_{C[0,1]}$ that is nowhere differentiable.

Proof. This follows from Theorems 6 and 7.

References

1. E. Allender and M. Strauss. Measure on small complexity classes with applications for BPP. In *Proceedings of the 35th Annual IEEE Foundations of Computer Science Conference*, pages 807–818, 1994. 219
2. K. Ambos-Spies. Resource-bounded genericity. In *Proceedings of the 10th Conference on Structure in Complexity Theory*, pages 162–181. IEEE Computer Society Press, 1995. 219
3. K. Ambos-Spies and E. Mayordomo. Resource-bounded measure and randomness. In *Complexity, Logic, and Recursion Theory*. Marcel Dekker, Inc., 1997. 219
4. K. Ambos-Spies, H.-C. Neis, and S. A. Terwijn. Genericity and measure for exponential time. *Theoretical Computer Science*, 168:3–19, 1996. 219

5. S. Banach. Über die Baire'sche kategorie gewisser funktionenmengen. *Studia Math.*, 3:174, 1931. [220](#), [229](#)
6. S. A. Fenner. Notions of resource-bounded category and genericity. In *Proceedings of the 6th Annual IEEE Structure in Complexity Theory Conference*, pages 196–212. IEEE Computer Society Press, 1991. [219](#), [220](#)
7. S. A. Fenner. Resource-bounded baire category: A stronger approach. In *Proceedings of the 10th Conference on Structure in Complexity Theory*, pages 162–181. IEEE Computer Society Press, 1995. [219](#), [220](#), [225](#)
8. Kiyosi Itô, editor. *Encyclopedic Dictionary of Mathematics*. MIT Press, Cambridge, Mass., second edition, 1993. [221](#)
9. Ker-I Ko. *Complexity Theory of Real Functions*. Birkhäuser, Boston, 1991. [219](#), [220](#), [221](#), [229](#)
10. L. R. Lisagor. The Banach-Mazur game. *Math. USSR Sbornik*, 38(2):201–216, 1981. [219](#)
11. J. H. Lutz. *Category and Measure in Complexity Classes*. PhD thesis, California Institute of Technology, 1987. [219](#)
12. J. H. Lutz. Resource-bounded Baire category and small circuits in exponential space. In *Proceedings of the 2nd Annual Conference on Structure in Complexity Theory*, pages 87–91. IEEE Computer Society Press, 1987. [219](#), [220](#), [225](#)
13. J. H. Lutz. Category and measure in complexity classes. *SIAM Journal on Computing*, 19(6):1100–1131, 1990. [219](#), [220](#)
14. J. H. Lutz. Almost everywhere high nonuniform complexity. *Journal of Computer and System Sciences*, 44:220–258, 1992. [219](#)
15. J. H. Lutz. The quantitative structure of exponential time. In *Complexity Theory Retrospective II*, pages 225–260. Springer-Verlag, 1997. [219](#)
16. E. Mayordomo. Measuring in PSPACE. In *Proceedings of the 7th International Meeting of Young Computer Scientists*, pages 122–29, 1992. [219](#)
17. E. Mayordomo. *Contributions to the Study of Resource-Bounded Measure*. PhD thesis, Universitat Politècnica de Catalunya, 1994. [219](#)
18. K. Mehlhorn. On the size of sets of computable functions. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 190–196, 1973. [219](#)
19. J. Myhill. A recursive function defined on a compact interval and having a continuous derivative that is not recursive. *Michigan Math. Journal*, 18:97–98, 1971. [220](#)
20. J. C. Oxtoby. *Measure and Category*. Springer-Verlag, Berlin, second edition, 1980. [221](#), [224](#)
21. M. Pour-El and I. Richards. Differentiability properties of computable functions – a summary. *Acta Cybernetica*, 4:123–125, 1978. [220](#)
22. W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, third edition, 1976. [221](#)
23. M. Strauss. Measure on P: Strength of the notion. *Information and Computation*, 136(1):1–23, 1997. [219](#), [220](#), [225](#)
24. Y. Wang. Genericity, randomness, and polynomial-time approximations. *SIAM Journal on Computing*, 28(2):394–408, 1998. [219](#)
25. K. Weierstrass. *Mathematische Werke*, volume II. Mayer and Muller, Berlin, 1894. [220](#)
26. L. Wen. A nowhere differentiable continuous function. *Amer. Math. Monthly*, 107:450–453, 2000. [220](#)

Translation among CNFs, Characteristic Models and Ordered Binary Decision Diagrams

Takashi Horiyama¹ and Toshihide Ibaraki²

¹ Graduate School of Information Science, Nara Institute of Science and Technology
Nara, 630-0101, Japan

horiyama@is.aist-nara.ac.jp

² Graduate School of Informatics
Kyoto University, Kyoto, 606-8501, Japan
ibaraki@i.kyoto-u.ac.jp

Abstract. We consider translation among conjunctive normal forms (CNFs), characteristic models, and ordered binary decision diagrams (OBDDs) of Boolean functions. It is shown in this paper that Horn OBDDs can be translated into their CNFs in polynomial time. As for the opposite direction, the problem can be solved in polynomial time if the ordering of variables in the resulting OBDD is specified as an input. In case that such ordering is not specified and the resulting OBDD must be of minimum size, its decision version becomes NP-complete. Similar results are also obtained for the translation in both directions between characteristic models and OBDDs. We emphasize here that the above results holds on any class of functions having a basis B with $|B| = d$.

1 Introduction

Logical formulae are the traditional means of representing Boolean functions in many fields of computer science. Their flexibility, however, leads to intractability of many problems (e.g., the satisfiability problem of a formula is NP-complete). By restricting the types of propositional clauses such problems can sometimes be solved efficiently. For example, it is common to consider Horn clauses in artificial intelligence [13], and the satisfiability problem of a Horn CNF can be solved in linear time [6]. Nevertheless, many problems still remain to be intractable (e.g., abduction from a knowledge-base is NP-complete even if the knowledge-base is represented by a Horn CNF [15]).

An alternative way of representing a knowledge-base has been proposed; i.e. it uses a subset of its models called characteristic models (see e.g., [9,10,12]). Deduction from a knowledge-base in this model-based approach can be performed in linear time, and abduction is also performed in polynomial time [9]. In addition to these favorable time complexity, empirical evaluation is also performed in the practical sense [10].

Another means of representing Boolean functions is ordered binary decision diagrams (OBDDs) [3,14]. An OBDD is a directed acyclic graph representing a

Boolean function, and can be considered as a variant of a decision tree. By restricting the order of variable appearances and by sharing isomorphic subgraphs, OBDDs have the following useful properties: (i) When an ordering of variables is specified, an OBDD has the unique reduced canonical form for each Boolean function. (ii) Many Boolean functions appearing in practice can be compactly represented. (iii) When an OBDD is given, satisfiability and tautology of the represented function can be easily checked in constant time. (iv) There are efficient algorithms for many other Boolean operations on OBDDs. As a result of these properties, OBDDs are widely used for various practical applications, especially in computer-aided design and verification of digital systems (see e.g., [5]).

CNFs, characteristic models and OBDDs are alternative representations of Boolean functions, and each of them has advantages over the other two [7,9]. Therefore, there is a natural question; how difficult is the translation among these representations? It is known that the translation problems between Horn CNFs and characteristic models, in both directions, are at least as hard as the hypergraph transversal problem (HTR), which is equivalent to the translation between CNFs and DNFs of monotone functions [11].

In this paper, we first consider the translation between Horn CNFs and their OBDDs, which has been studied as one of the fundamental problems. Unfortunately, most works so far have been directed to heuristic algorithms, and there is not much theoretical analysis on its computational complexity. The only result known is that the translation from a general CNF into its OBDD of minimum size is co-NP-hard [16]. We show in this paper that translation from Horn OBDDs into CNFs is solvable in polynomial time.

As for the translation from CNFs into OBDDs, we consider the following two cases: (i) the ordering of variables for the resulting OBDD is specified as an input, and (ii) the ordering can be determined so that the resulting OBDD has the minimum size. In the first case, we show that the problem is solvable in polynomial time (more specifically in output polynomial time). The size of an OBDD largely depends on the ordering of variables, and the resulting size can vary exponentially [3]. Reflecting this aspect, we show that the decision variant of the translation problem is NP-complete. Our result is based on the NP-hardness of the OBDD-minimization problem, which outputs an OBDD with the minimum size for a given monotone OBDD by selecting the ordering of variables [8]. Note that the result of [8] does not directly imply the NP-hardness of the translation from a CNF, since a Horn CNF may require exponentially larger size than its OBDD, and vice versa [7].

We then discuss translation between characteristic models and their OBDDs, which, to our knowledge, has not been discussed in the literature. The following two problems are solvable in output polynomial time; (i) translation from OBDDs into characteristic models and (ii) translation from characteristic models into OBDDs with a specified variable ordering. The translation into OBDDs of minimum sizes is shown to be intractable. Our focus is not only on the class of Horn functions but also on any class of functions having a basis $B = \{b^{(1)}, b^{(2)}, \dots, b^{(d)}\}$ with $|B| = d$.

The rest of this paper is organized as follows. The next section gives definitions and concepts. We discuss translation problems between Horn CNFs and OBDDs in Section 3, and those between characteristic monotone and their OBDDs in Section 4. Section 5 contains the concluding remarks.

2 Preliminaries

2.1 Notations and Basic Concepts

We consider a Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$. An *assignment* is a vector $a \in \{0, 1\}^n$, whose i -th coordinate is denoted by a_i . A *model* of f is a satisfying assignment a of f , and the *theory* $\Sigma(f)$ *representing* f is the set of all models of f . The *size* of $\Sigma(f)$, denoted by $|\Sigma(f)|$, is the number of models in it. Given $a, b \in \{0, 1\}^n$, we denote by $a \leq b$ the usual bitwise (i.e., componentwise) ordering of assignments; $a_i \leq b_i$ for all $i = 1, 2, \dots, n$, where $0 < 1$.

Let $x_1, x_2, \dots, x_n$ be the n variables of f . Negation of a variable x_i is denoted by $\bar{x}_i$. Any Boolean function can be represented by some CNF, which may not be unique. The *size* of a CNF φ , denoted by $|\varphi|$, is the number of literals in φ . We sometimes do not make distinction among a function f , its theory $\Sigma(f)$, and a CNF φ that represents f , unless confusion arises. For example, notation $a \in f$ is used to mean $a \in \Sigma(f)$.

A clause is *Horn* if the number of positive literals in it is at most one, and a CNF is *Horn* if it contains only Horn clauses. A Boolean function f is *Horn* if f can be represented by some Horn CNF. It is known that a theory Σ is Horn if and only if Σ can be represented by some Horn CNF.

Given an assignment $p \in \{0, 1\}^n$, we define $a \leq_p b$ if $(a \oplus_{bit} p) \leq (b \oplus_{bit} p)$ holds, where $\oplus_{bit}$ denotes the bitwise (i.e., componentwise) exclusive-or operation. The *monotone extension* of $a \in \{0, 1\}^n$ with respect to b is $\mathcal{M}_b(a) = \{c \mid a \leq_b c\}$, and the *monotone extension* of f with respect to b is

$$\mathcal{M}_b(f) = \bigcup_{a \in f} \mathcal{M}_b(a).$$

The set of *minimal models* of f with respect to b is defined as

$$\text{min}_b(f) = \{a \in f \mid \text{there exists no } c \in f \text{ satisfying } c <_b a\},$$

where $c <_b a$ denotes that $c \leq_b a$ and $c \neq a$ hold. $\mathcal{M}_b(f)$ can be rewritten as

$$\mathcal{M}_b(f) = \bigcup_{a \in \text{min}_b(f)} \mathcal{M}_b(a).$$

Finally, f is characterized as

$$f = \bigwedge_{b \in \{0, 1\}^n} \mathcal{M}_b(f) = \bigwedge_{b \notin f} \mathcal{M}_b(f),$$

since $f \subseteq \mathcal{M}_b(f)$ and $b \notin f \Leftrightarrow b \notin \mathcal{M}_b(f)$ hold for any $b \in \{0, 1\}^n$ [4].

We may find a small set of assignments $B (\subseteq \{0, 1\}^n - \Sigma(f))$ satisfying $f = \bigwedge_{b \in B} \mathcal{M}_b(f)$. Such B is called a *basis for a Boolean function* f . Furthermore, B is called a *basis for a class of Boolean functions* $\mathcal{C}$ if it is a basis for all functions in $\mathcal{C}$. For a Boolean function $f \in \mathcal{C}$, the *set of characteristic models* $\text{Char}^B(f)$ with respect to B is defined as

$$\text{Char}^B(f) = \bigcup_{b \in B} \min_b(f).$$

It is known that the class of Horn functions $\mathcal{C}_H$ has the basis $B_H = \{b \mid \|b\| \geq n - 1\}$, where $\|b\| = \sum_{i=1}^n b_i$ [12]. The following lemma gives an upper bound on the size (i.e., cardinality) of the set of the minimal models.

Lemma 2.1 [12] *Let $\varphi = t_1 \vee t_2 \vee \dots \vee t_k$ be a DNF of a Boolean function on n variables. Then, for every $b \in \{0, 1\}^n$, $\min_b(f) \leq k$ holds. Moreover, we can construct $\min_b(f)$ in $O(nk^2)$ time.*

We define a *restriction* of f by replacing a variable x_i by a constant $a_i \in \{0, 1\}$, and denote it by $f|_{x_i=a_i}$. Namely, $f|_{x_i=a_i}(x_1, \dots, x_n) = f(x_1, \dots, x_{i-1}, a_i, x_{i+1}, \dots, x_n)$ holds. Restriction may be applied to many variables.

2.2 Ordered Binary Decision Diagrams

An *ordered binary decision diagram* (OBDD) is a directed acyclic graph that represents a Boolean function. It has two sink nodes 0 and 1, called the *0-node* and the *1-node*, respectively (which are together called the *constant nodes*). Other nodes are called *variable nodes*, and each variable node v is labeled by one of the variables $x_1, x_2, \dots, x_n$. Let $\text{var}(v)$ denote the label of node v . Each variable node has exactly two outgoing edges, called a *0-edge* and a *1-edge*, respectively. One of the variable nodes becomes the unique source node, which is called the *root node*. Let $X = \{x_1, x_2, \dots, x_n\}$ denote the set of n variables. A *variable ordering* is a total ordering $(x_{\pi(n)}, x_{\pi(n-1)}, \dots, x_{\pi(1)})$, associated with each OBDD, where π is a permutation $\{1, 2, \dots, n\} \rightarrow \{1, 2, \dots, n\}$. The *level*¹ of a variable $x_{\pi(i)}$, denoted by $\text{level}(x_{\pi(i)})$, is defined to be i . Similarly, the level of a node v , denoted by $\text{level}(v)$, is defined by its label; if node v has label $x_{\pi(i)}$, $\text{level}(v)$ is defined to be i . That is, the root node is in level n and has label $x_{\pi(n)}$, the nodes in level $n - 1$ have label $x_{\pi(n-1)}$ and so on. The level of the constant nodes is defined to be 0. On every path from the root node to a constant node in an OBDD, each variable appears at most once in the decreasing order of their levels.

Every node v of an OBDD also represents a Boolean function f_v , defined by the subgraph consisting of those nodes and edges which are reachable from v . If a node v is a constant node, f_v equals to its label. If a node v is a variable node, f_v is defined as $\overline{\text{var}(v)} f_{0\text{-succ}(v)} \vee \text{var}(v) f_{1\text{-succ}(v)}$ by Shannon's expansion,

¹ This definition of level may be different from its common use.

where $0\text{-succ}(v)$ and $1\text{-succ}(v)$, respectively, denote the nodes pointed by the 0-edge and the 1-edge of node v . The function f represented by an OBDD is the one represented by the root node. Given an assignment a , the value $f(a)$ is determined by following a path from the root node to a constant node by selecting $a_{\text{var}(v)}$ -edge at each variable node v . The value $f(a)$ is given by the label of the final constant node reachable in this manner.

When two nodes u and v in an OBDD represent the same function, and their levels are the same, they are called *equivalent*. A node whose 0-edge and 1-edge both point to the same node is called *redundant*. An OBDD which has no mutually equivalent nodes and no redundant nodes is *reduced*. In the following, we assume that all OBDDs are reduced, unless otherwise stated. The *size* of an OBDD G_f representing f , denoted by $|G_f|$, is the number of nodes in the OBDD. Given a function f and a variable ordering, its reduced OBDD is unique and has the minimum size among all OBDDs with the same variable ordering. The sizes of OBDDs that represent a given Boolean function may vary according to the variable orderings [3].

Given an OBDD that represents f , the OBDDs of $f|_{x_i=0}$ and $f|_{x_i=1}$ can be obtained in $O(|G_f|)$ time [2]. The size of an OBDD does not increase by a restriction. Given two OBDDs G_f and G_g representing f and g respectively, fundamental logic operators, e.g., $f \wedge g$, $f \vee g$, $f \oplus g$ and $f \rightarrow g$, can be applied in $O(|G_f| \cdot |G_g|)$ time [3]. The equivalence condition $f \equiv g$ can be checked in constant time if we use shared binary decision diagrams (SBDDs), in which isomorphic subgraphs are shared among two or more OBDDs [14].

3 Translation between Horn OBDDs and CNFs

3.1 Translating Horn CNFs into OBDDs

We first discuss the translation of a Horn CNF into its OBDD with a specified variable ordering, and show that it can be done in output polynomial time.

Theorem 3.1 *Given a CNF φ_f of a Horn function f on n variables and a variable ordering $\pi = (\pi(n), \pi(n-1), \dots, \pi(1))$, its OBDD G_f with variable ordering π can be obtained in $O(|\varphi_f| \cdot |G_f|^2)$ time.*

Algorithm CONSTRUCT-OBDD and CONSTRUCT-OBDD1 in Fig. 1 construct an OBDD in the depth first-manner. We use a node table T to store the nodes in G_f . A variable node v is stored as a 4-tuple $(x_v, 0\text{-succ}(v), 1\text{-succ}(v), \varphi_v)$, where x_v denotes the label of node v , and φ denotes a CNF representation of the function f_v . The 0-node and the 1-node are stored as $(0, *, *, 0)$ and $(1, *, *, 1)$, respectively. We start the algorithm with storing the constant nodes into T .

Algorithm CONSTRUCT-OBDD1 is the main part of our construction. Before generating a node of φ in Step 3, we check equivalency and redundancy of the node in Steps 1 and 2. In Step 1, we check whether we have already generated a node u whose CNF φ_u is logically equivalent to φ . If the answer is yes, node u is returned as the root node of the OBDD of φ . Otherwise, we decide

Algorithm CONSTRUCT-OBDD**Input:** A Horn CNF φ and a variable ordering $\pi = (\pi(n), \pi(n-1), \dots, \pi(1))$.**Output:** OBDD G_φ with variable ordering π , which represents φ **Step 1 (initialization).** Let $T := \{(0, *, *, 0), (1, *, *, 1)\}$.**Step 2 (main part).** Let $v := \text{CONSTRUCT-OBDD1}(\varphi, \pi)$. Output the graph consisting of those nodes and edges which are reachable from v .**Algorithm CONSTRUCT-OBDD1****Input:** A Horn CNF φ and a variable ordering $\pi = (\pi(n), \pi(n-1), \dots, \pi(1))$.**Output:** The root node of OBDD G_φ with variable ordering π .**Step 1 (equivalency check).** For each node $u = (x_u, 0\text{-succ}(u), 1\text{-succ}(u), \varphi_u)$ in T , check whether $\varphi_u \equiv \varphi$ holds or not. If $\varphi_u \equiv \varphi$ holds, return node u .**Step 2 (redundancy check).** Let $\varphi_0 := \varphi|_{x_{\pi(n)}=0}$; $\varphi_1 := \varphi|_{x_{\pi(n)}=1}$. If $\varphi_0 \equiv \varphi_1$ holds, consider φ_0 and $(\pi(n-1), \dots, \pi(1))$ as a new input, and return to Step 2.**Step 3 (generation of a node).** Let $v_0 := \text{CONSTRUCT-OBDD1}(\varphi_0, (\pi(n-1), \dots, \pi(1)))$; $v_1 := \text{CONSTRUCT-OBDD1}(\varphi_1, (\pi(n-1), \dots, \pi(1)))$. Register node $v = (x_{\pi(n)}, v_0, v_1, \varphi)$ into T , and return node v .**Fig. 1.** Algorithms CONSTRUCT-OBDD and CONSTRUCT-OBDD1 to construct an OBDD from a Horn CNF

to generate a new node v . In Step 2, in order to find the label of the new node, we check whether φ depends on variable $x_{\pi(n)}$. In Step 3, after obtaining the OBDDs of φ_0 and φ_1 , we generate a node of φ and register it into T .

The crucial point of this algorithm is that the equivalent conditions $\varphi_u \equiv \varphi$ in Step 1 and $\varphi_0 \equiv \varphi_1$ in Step 2 can be checked in $O(|\varphi_f|)$ time. This is because, for any Horn CNFs φ_u and φ_v , the equivalence condition $\varphi_u \equiv \varphi_v$ can be checked in $O(|\varphi_u| + |\varphi_v|)$ time. (Note that such check is intractable for general CNFs.) Furthermore, every CNF φ_u stored in T has size at most $|\varphi_f|$ since it is obtained by replacing variables $x_{\pi(n)}, x_{\pi(n-1)}, \dots, x_{\pi(i)}$ by constant values.

Next, we consider the computation time for the construction. Since T contains at most $|G_f|$ nodes, Step 1 in algorithm CONSTRUCT-OBDD1 can be done in $O(|\varphi_f| \cdot |G_f|)$ time. Step 2 can be done in $O(n \cdot |\varphi_f|)$ time, since the step is iterated at most n time. Step 3 says that, for each node in G_f , algorithm CONSTRUCT-OBDD1 is called at most twice. Thus, the entire computation time is $O(|\varphi_f| \cdot |G_f|^2)$.

3.2 Translating Horn OBDDs into CNFs

Now, we consider how to translate Horn OBDDs into their CNFs, and show that it can be done in output polynomial time. The proof is based on the previous results in computational learning theory [1]. In the framework of learning, the

goal is to find a CNF which is logically equivalent to a hidden Boolean function f . A learner can use the following two kinds of queries to the oracle; membership queries and equivalence queries.

Definition 3.1 *Given an assignment $a \in \{0, 1\}^n$, a membership query to the oracle returns $f(a)$.*

Definition 3.2 *Given a CNF φ , an equivalence query to the oracle returns ‘yes’ if $f \equiv \varphi$. Otherwise it returns ‘no’ and a counter example $a \in \{0, 1\}^n$ satisfying $f(a) \neq \varphi(a)$.*

The hidden concept f can be learned in polynomial time if it is Horn.

Lemma 3.1 [1] *Let f be a Horn function on n variables which can be represented by a Horn CNF with m clauses. Then, there exists an algorithm that, given access to the oracle of a hidden Horn function f , outputs a Horn CNF φ_f which is equivalent to f . It runs in $\tilde{O}(m^2n^2)$ time with $O(m^2n)$ membership queries and $O(mn)$ equivalence queries.*

Based on this lemma, we can derive the following result.

Theorem 3.2 *Given an OBDD G_f of a Horn function f on n variables, its CNF representation φ_f can be obtained in $O(m^2n^3 \cdot |G_f|^2)$ time, where m is the minimum number such that there exists a Horn CNF φ_* with m clauses satisfying $\varphi_* \equiv f$.*

Proof: To make use of Lemma 3.1, we consider how the answers to membership queries and equivalence queries can be obtained from the given OBDD G_f . In other words, we treat G_f as the oracle in the learning process. A membership query for an assignment a can be easily answered in $O(n)$ time just by following the path from the root node of G_f to either of the constant nodes.

Next, we consider an equivalence query for the current CNF φ . By observing algorithm HORN1 in [1], we have the following fact; the current CNF φ (and also the resulting CNF φ_f) has at most $m(n+1)$ clauses, and thus has size at most $O(mn^2)$. In order to answer the query, we first construct an OBDD G_φ with the same variable ordering as G_f . For this purpose, we run algorithm CONSTRUCT-OBDD in Theorem 3.1 until it outputs G_φ , or until it exceeds its time bound with respect to φ and G_f (which is $O(|\varphi| \cdot |G_f|^2)$). In the first case, we check the equivalence between G_f and G_φ , which can be done in constant time [14]. If G_φ is not equivalent to G_f , we answer ‘no’ and supplies a counter example $a \in \{0, 1\}^n$ which is a satisfying assignment of $\varphi \oplus f$.

In the second case, we know that $|G_\varphi| > |G_f|$, therefore the answer is ‘no’. Although G_φ is incompletely constructed, it is quite useful for obtaining a counter example. Let G_φ^0 (G_f^0) and G_φ^1 (G_f^1) denote the OBDD pointed by the 0-edge and the 1-edge of the root node of G_φ (G_f), respectively. Since Algorithm CONSTRUCT-OBDD constructs G_φ in the depth-first manner, at the break of the execution, the following two cases are possible:

- (i) OBDD G_φ^0 is under construction, or
- (ii) OBDD G_φ^0 has been completely constructed and OBDD G_φ^1 is under construction.

In case (i), we have $|G_\varphi^0| > |G_f^0|$. Thus, there exists an assignment $a \in \{0, 1\}^n$ satisfying $a_{\pi(n)} = 0$ and $\varphi(a) \neq f(a)$. We can apply the same argument to OBDDs G_φ^0 and G_f^0 recursively, and obtain a counter example $a \in \{0, 1\}^n$ satisfying $\varphi(a) \neq f(a)$. In case (ii), we first check the equivalence condition $\varphi|_{x_{\pi(n)}=0} \equiv f|_{x_{\pi(n)}=0}$. If the condition does not hold, we can obtain a counter example a by checking $\varphi|_{x_{\pi(n)}=0} \oplus f|_{x_{\pi(n)}=0}$. Otherwise, there exists an assignment a satisfying $a_{\pi(n)} = 1$ and $\varphi(a) \neq f(a)$, since we have $|G_\varphi^1| > |G_f^1|$. We can apply the same argument to OBDDs G_φ^1 and G_f^1 . Thus, a counter example is obtained by no more than n equivalence checks of OBDDs and exactly one $\oplus$ operation.

Now, we consider the computation time for the translation. An equivalence query requires $O(mn^2 \cdot |G_f|^2)$ time, since algorithm CONSTRUCT-OBDD requires $O(mn^2 \cdot |G_f|^2)$ time and a counter example is obtained in $O(n \cdot |G_f|^2)$ time. Thus, $O(mn)$ equivalence queries require $O(m^2n^3 \cdot |G_f|^2)$ time. The rest of computation time is minor. $\square$

3.3 Translating Horn CNFs into Minimum OBDDs

We next consider the translation of a Horn CNF into its OBDD of minimum size, and show its intractability by proving the NP-completeness for monotone functions.

Theorem 3.3 *Given a CNF φ_f of a monotone function f on n variables, and a positive integer k , deciding whether there is a variable ordering π that results in an OBDD of f with size at most k is NP-complete.*

Outline of the proof: The problem is obviously in NP since we can guess a variable ordering $(x_{\pi(n)}, x_{\pi(n-1)}, \dots, x_{\pi(1)})$, and construct the resulting unique OBDD by Theorem 3.1.

On the other hand, it is known that, given a monotone OBDD and a positive integer k , deciding whether there is a variable ordering that gives an OBDD of the same function with size at most k is NP-complete [8]. It is proved by a reduction from the Optimal Linear Arrangement Problem (OLA). An instance of OLA is a graph $G_{\text{OLA}} = (V, E)$ and a positive integer K . It asks whether there is a one-to-one mapping $\psi : V \rightarrow \{1, 2, \dots, |V|\}$ satisfying $\sum_{(u,v) \in E} |\psi(u) - \psi(v)| \leq K$.

Since the focus in [8] is on the sizes of the two participating OBDDs, the monotone function used in the reduction does not have a polynomial size CNF with respect to the size of the OLA instance. Therefore the proof in [8] does not immediately prove our theorem. However, by a small modification, we can reduce the given OLA instance to a CNF of a monotone function, whose size is polynomial in the size of OLA. More precisely, an instance of OLA is reduced to the following negative function h^* with $|E|(|V| + 2)$ variables:

$$h^* = h_{|E|} \quad (1)$$

$$h_i = \begin{cases} \text{AMOAMO} & (i = 0) \\ (\overline{y}_i \vee f_i)(\overline{z}_i \vee h_{i-1})(\overline{y}_i \vee \overline{z}_i) & (i \geq 1) \end{cases} \quad (2)$$

$$f_i = (\overline{x}_{(j_1, i)} \vee \overline{x}_{(j_2, i)}) \bigwedge_{k \in \{1, 2, \dots, |V|\}, k \neq j_1, k \neq j_2} \overline{x}_{(k, i)} \quad (e_i = (v_{j_1}, v_{j_2}) \in E) \quad (3)$$

$$\text{AMOAMO} = \bigwedge_{i_1, i_2 \in \{1, 2, \dots, |V|\}, i_1 \neq i_2} (\text{AMO}_{i_1} \vee \text{AMO}_{i_2}) \quad (4)$$

$$\text{AMO}_i = \bigwedge_{w_1, w_2 \in W_i, w_1 \neq w_2} (\overline{w}_1 \vee \overline{w}_2), \quad (5)$$

where $W_i = \{x_{(i,1)}, x_{(i,2)}, \dots, x_{(i,|E|)}\}$. The output of AMO_i is 1 if and only if at most one of the variables in W_i is 1. Similarly, the output of AMOAMO is 1 if and only if at most one of AMO_i 's is 0.

First, we show that h^* can be represented by a negative CNF of size $O(|V| \cdot |E|^2 + |E|^5)$. For convenience, we denote a *clause-size set* of a CNF φ as $\{(c_1, s_1), (c_2, s_2), \dots, (c_k, s_k)\}$, if φ consists of c_1 clauses of size s_1 , c_2 clauses of size s_2 , $\dots$, and c_k clauses of size s_k . The CNF of f_i in Eq. (3) is negative, and its clause-size set is $\{(|V| - 2, 1), (1, 2)\}$. From Eqs. (4) and (5), AMOAMO can be written as

$$\text{AMOAMO} = \bigwedge_{i_1, i_2 \in \{1, 2, \dots, |V|\}, i_1 \neq i_2} \left(\bigwedge_{\substack{w_1, w_2 \in W_{i_1}, w_1 \neq w_2 \\ w_3, w_4 \in W_{i_2}, w_3 \neq w_4}} (\overline{w}_1 \vee \overline{w}_2 \vee \overline{w}_3 \vee \overline{w}_4) \right).$$

Namely, the CNF is negative, and its clause-size set is $\{(O(|V|^2 \cdot |E|^4), 4)\}$. By induction, h_i ($i \geq 1$) can be represented by a negative CNF whose clause-size set is $\left(\bigcup_{k=1}^i \{(|V| - 1, k + 1), (1, k + 2)\}\right) \cup \{(O(|V|^2 \cdot |E|^4), i + 4)\}$. Thus, we can obtain a negative CNF of h^* , whose size is $O(|V|^2 \cdot |E|^5)$.

Two major building blocks in the reduction are *edge functions* f_i ($i = 1, 2, \dots, |E|$) and a *penalty function* AMOAMO. Variables $y_1, y_2, \dots, y_{|E|}, z_1, z_2, \dots, z_{|E|}$ are used to combine f_i 's and AMOAMO into a single function. The underlying idea is the same as [8]:

- (i) The OBDD G_{f_i} of edge function f_i corresponds to edge $e_i = (v_{j_1}, v_{j_2})$ in the given OLA instance G_{OLA} . More precisely, variable ordering $(x_{(k_1, i)}, x_{(k_2, i)}, \dots, x_{(k_{|V|}, i)})$ of G_f corresponds to the mapping $\psi : v_{k_j} \rightarrow k_j$ of G_{OLA} . The size of OBDD G_{f_i} is $|\psi(v_{j_1}) - \psi(v_{j_2})| + |V| + 1$.
- (ii) Node v_k in G_{OLA} has $|E|$ corresponding variables $V_k = \{x_{(k,1)}, x_{(k,2)}, \dots, x_{(k,|E|)}\}$, and no two edge functions depend on the same variables. Therefore, if we have only edge functions, then there exists a trivial ordering which minimizes all of the sizes of their OBDDs.
- (iii) The penalty function AMOAMO force the variables in the OBDD to be grouped. Namely, in the OBDD of minimum size, for any variables x and x'

in V_k (i.e., corresponding to node v_k in G_{OLA}), all variables existing in the levels between $\text{level}(x)$ and $\text{level}(x')$ are also in V_k . Otherwise, the cost of the penalty function becomes large.

We can show that the size of minimum OBDD of h^* is $K^* + 5|V| \cdot |E| - 4|V| - 3|E| + 6$, where K^* is the minimum value of $\sum_{(u,v) \in E} |\psi(u) - \psi(v)|$ (i.e., the cost function of OLA). $\square$

4 Translation between Characteristic Models and OBDDs

4.1 Translating Characteristic Models into OBDDs

In this section, we consider the translation between characteristic models and OBDDs. Our focus is not only on the class of Horn functions but also on any class of functions having a basis $B = \{b^{(1)}, b^{(2)}, \dots, b^{(d)}\}$ with $|B| = d$. Let us first consider the translation of a set of characteristic models into its OBDD with a specified variable ordering. It can be done in output polynomial time.

Theorem 4.1 *Let f be a Boolean function on n variables, which belongs to a class of functions having a basis B with $|B| = d$. Given a set of characteristic models $\text{Char}^B(f)$ and a variable ordering $\pi = (\pi(n), \pi(n-1), \dots, \pi(1))$, its OBDD G_f with variable ordering π can be obtained in $O(nd \cdot |\text{Char}^B(f)|^2 \cdot |G_f|^2)$ time.*

Outline of the proof: We construct OBDD G_f in a manner similar to Theorem 3.1. The difference is that every function g is represented by a d -tuple $t^B(g) = (\min_{b^{(1)}}(g), \min_{b^{(2)}}(g), \dots, \min_{b^{(d)}}(g))$, where $\min_{b^{(i)}}(g)$ is the set of minimal models of g defined in Section 2.1. The initial d -tuple $t^B(f)$ of a given function f is easily obtained from $\text{Char}^B(f)$ in $O(nd \cdot |\text{Char}^B(f)|^2)$ time, since every $\min_{b^{(i)}}(f)$ can be obtained by deleting non-minimal models in $\text{Char}^B(f)$. In Step 2 of Algorithm CONSTRUCT-OBDD1, we use the d -tuple representations of $f_u|_{x_{\pi(n)}=0}$ and $f_u|_{x_{\pi(n)}=1}$. They are obtained by

$$\begin{aligned} \min_{b^{(i)}}(f_u|_{x_{\pi(n)}=c}) &= M_c \\ \min_{b^{(i)}}(f_u|_{x_{\pi(n)}=\bar{c}}) &= M_{\bar{c}} \cup \{a \in M_c \mid \forall a' \in M_{\bar{c}} \ a' \not\leq_{b^{(i)}} a\}, \end{aligned}$$

where $c = b_{\pi(n)}^{(i)} \in \{0, 1\}$, $M_e = \{a \in \{0, 1\}^{n-1} \mid (a, e) \in \min_{b^{(i)}}(f_u)\}$ for $e \in \{0, 1\}$, and (a, e) is a concatenation of a and e . As for the constants 0 and 1, we define $\min_{b^{(i)}}(0) = \emptyset$ and $\min_{b^{(i)}}(1) = \{\top\}$, where $(\top, e) = e$ holds for $e \in \{0, 1\}$. The sizes of $\min_{b^{(i)}}(f_u|_{x_{\pi(n)}=0})$ and $\min_{b^{(i)}}(f_u|_{x_{\pi(n)}=1})$ are at most $|\min_{b^{(i)}}(f_u)|$. By induction, every node v has a d -tuple $t^B(f_v)$ in which $\min_{b^{(i)}}(f_v)$ has size at most $|\min_{b^{(i)}}(f)|$, and thus at most $|\text{Char}^B(f)|$.

Recall that a crucial point of Algorithm CONSTRUCT-OBDD1 is to check the equivalency of arbitrarily given two functions g and g' in polynomial time. As for the modified algorithm, we achieve it by comparing $\min_{b^{(i)}}(g) (\subseteq \{0, 1\}^k)$

and $\min_{b^{(i)}}(g') (\subseteq \{0,1\}^\ell)$ for all $i \in \{1, 2, \dots, d\}$. Here, we have two cases; $k = \ell$, and $k \neq \ell$ (without loss of generality, we assume $k > \ell$). In the first case, we check the equivalency of $\min_{b^{(i)}}(g)$ and $\min_{b^{(i)}}(g')$ for all i 's, since every function has a unique set of minimal models. In the second case, we consider that g' has k variables $x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(k)}$ although g' does not depends on variables $x_{\pi(\ell)+1}, x_{\pi(\ell)+2}, \dots, x_{\pi(k)}$. Then, $g'(x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(k)})$ has the following set of minimal models:

$$M_{k,b^{(i)}}(g') = \left\{ a \in \{0,1\}^k \mid \begin{array}{l} (a_{\pi(1)}, a_{\pi(2)}, \dots, a_{\pi(\ell)}) \in \min_{b^{(i)}}(g') \text{ and} \\ a_{\pi(j)} = b_{\pi(j)}^{(i)} \text{ for } j = \ell + 1, \ell + 2, \dots, k \end{array} \right\}.$$

Similarly to the case when $k = \ell$, the condition $g \equiv g'$ holds if and only if $\min_{b^{(i)}}(g)$ and $M_{k,b^{(i)}}(g')$ are equivalent for all i 's. Therefore, the equivalency of g and g' can be checked in $O(nd \cdot |\text{Char}^B(f)|^2)$ time. The entire computation time is $O(nd \cdot |\text{Char}^B(f)|^2 \cdot |G_f|^2)$. $\square$

4.2 Translating OBDDs into Characteristic Models

Now, we consider the translation of an OBDD into its characteristic set.

Theorem 4.2 *Let f be a Boolean function on n variables, which belongs to a class of functions having a basis B with $|B| = d$. Given an OBDD G_f , its characteristic set $\text{Char}^B(f)$ can be obtained in $O(nd \cdot |\text{Char}^B(f)|^2 \cdot |G_f|)$ time.*

Outline of the proof: By definition, the characteristic models can be obtained by enumerating the minimal models for all elements of the basis B and deleting models appeared more than once. We enumerate the minimal models with respect to $b (\in \{0,1\}^n)$ by traversing OBDD G_f in the depth first manner. Let f_v denote the function represented by node v in G_f . Then, the set of the minimal models of f_v is obtained from $\min_b(f_{0\text{-succ}(v)})$ and $\min_b(f_{1\text{-succ}(v)})$;

$$\begin{aligned} \min_b(f_v) = & \{(a, c) \mid a \in \min_b(f_{c\text{-succ}(v)})\} \\ & \cup \{(a, \bar{c}) \mid a \in \min_b(f_{\bar{c}\text{-succ}(v)}) \text{ and } \forall a' \in f_{c\text{-succ}(v)} a' \not\leq_b a\}, \end{aligned}$$

where $c = b_{\pi(n)}$. By induction, every $\min_b(f_v)$ has size at most $|\min_b(f)|$, and thus at most $|\text{Char}^B(f)|$. By caching all $\min_b(f_v)$'s, $\min_b(f)$ can be obtained in $O(n \cdot |\text{Char}^B(f)|^2 \cdot |G_f|)$ time. $\square$

4.3 Translating Characteristic Models into Minimum OBDDs

We show that the translation of a set of characteristic models into the OBDD of minimum size is intractable.

Theorem 4.3 *Let f be a Boolean function on n variables, which belongs to a class of functions having a basis B with $|B| = d$. Given a set of characteristic models $\text{Char}^B(f)$, deciding whether its minimum OBDD has size at most k is NP-complete.*

Proof: The proof of this theorem is based on that of Theorem 3.3. We consider the dual h^d of function h^* in Theorem 3.3. Since h^* can be represented by a CNF of $O(|V|^2 \cdot |E|^4)$ clauses, h^d can be represented by a DNF of $O(|V|^2 \cdot |E|^4)$ terms. By Lemma 2.1, for every $b \in B$, we can construct $\min_b(h^d)$, whose size is $O(|V|^2 \cdot |E|^4)$. The characteristic set $\text{Char}^B(h^d)$ can be obtained by enumerating $\min_b(h^d)$ for all $b \in B$, which implies $|\text{Char}^B(h^d)|$ is $O(d \cdot |V|^2 \cdot |E|^4)$.

Since h^d is dual to h^* , an OBDD G_{h^d} of h^d is obtained from OBDD G_{h^*} of h^* by negating input variables (i.e., exchanging the roles of 0-edges and 1-edges) and negating output (i.e., exchanging the roles of the 0-node and the 1-node). The size does not change by dualization. Thus, the argument on the size of G_{h^*} also holds for that of G_{h^d} . $\square$

5 Concluding Remarks

In this paper, we considered translation among CNFs, characteristic models, and OBDDs of Boolean functions. We have shown that the translation problems between Horn CNFs and OBDDs, in both directions, are solvable in output polynomial time, while the translation of Horn CNFs into OBDDs of minimum sizes is shown to be intractable. Similar results are also obtained for the characteristic models and OBDDs.

Our results say that any Horn CNF can be translated into characteristic models (and vice versa) via its OBDD representation in polynomial time with respect to the sizes of the CNF, its characteristic models, and the intermediate OBDD. This is interesting since the translation problems between Horn CNFs and characteristic models, in both directions, are known to be at least as hard as the hypergraph transversal problem. We emphasize here that the size of the intermediate OBDD is not always exponentially larger than those of the CNF and the characteristic models.

References

1. D. Angluin, M. Frazier, and L. Pitt, "Learning Conjunctions of Horn Clauses," *Machine Learning*, 9, pp.147–164, 1992. 236, 237
2. K. S. Brace, R. L. Rundell, and R. E. Bryant, "Efficient Implementation of a BDD Package," *Proc. of 27th ACM/IEEE DAC*, pp.40–45, 1990. 235
3. R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, C-35, no.8, pp.677–691, 1986. 231, 232, 235
4. N. H. Bshouty, Exact learning Boolean functions via the monotone theory, *Information and Computation* 123 (1995) 146–153. 233
5. O. Coudert, "Doing Two-Level Logic Minimization 100 Times Faster," *Proc. of 6th ACM/SIAM SODA*, pp.112–118, 1995. 232
6. W. F. Dowling and J. H. Gallier, "Linear Time Algorithms for Testing the Satisfiability of Horn Formula," *J. Logic Programm.*, 3, pp.267–284, 1984. 231
7. T. Horiyama and T. Ibaraki, "Ordered Binary Decision Diagrams as Knowledge-Bases," *Proc. of 10th ISAAC*, LNCS 1741, Springer, pp.83–92, 1999. 232

8. K. Iwama, M. Nozoe, and S. Yajima, "Optimizing OBDDs Is Still Intractable for Monotone Functions," *Proc. of MFCS*, LNCS 1450, pp.625–635, 1998. 232, 238, 239
9. H. A. Kautz, M. J. Kearns, and B. Selman, "Reasoning with Characteristic Models," *Proc. of AAAI-93*, pp.34–39, 1993. 231, 232
10. H. A. Kautz, M. J. Kearns, and B. Selman, "Horn Approximations of Empirical Data," *Artificial Intelligence*, 74, pp.129–245, 1995. 231
11. R. Khardon, "Translating between Horn Representations and their Characteristic Models," *Journal of Artificial Intelligence Research*, 3, pp.349–372, 1995. 232
12. R. Khardon and D. Roth, "Reasoning with Models," *Artificial Intelligence*, 87, pp.187–213, 1996. 231, 234
13. J. McCarthy and P. J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," *Machine Intelligence*, 4, Edinburgh University Press, 1969. 231
14. S. Minato, N. Ishiura and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," *Proc. of 27th ACM/IEEE DAC*, pp.52–57, 1990. 231, 235, 237
15. B. Selman and H. J. Levesque, "Abductive and Default Reasoning: A Computational Core," *Proc. of AAAI-90*, pp.343–348, 1990. 231
16. S. Tani, K. Hamaguchi, and S. Yajima, "The Complexity of the Optimal Variable Ordering Problems of a Shared Binary Decision Diagram", *IEICE Trans. Inf.& Syst.*, vol.E79-D, no.4, pp.271–281, 1996. 232

On Removing the Pushdown Stack in Reachability Constructions[★]

Oscar H. Ibarra¹ and Zhe Dang²

¹ Department of Computer Science, University of California Santa Barbara, CA
93106, USA

`ibarra@cs.ucsb.edu`

² School of Electrical Engineering and Computer Science
Washington State University, Pullman WA 99164, USA

`zdang@eecs.wsu.edu`

Abstract. A discrete pushdown timed automaton is a pushdown machine with integer-valued clocks. It has been shown recently that the binary reachability of a discrete pushdown timed automaton can be accepted by a 2-tape pushdown acceptor with reversal-bounded counters. We improve this result by showing that the stack can be removed from the acceptor, i.e., the binary reachability can be accepted by a 2-tape finite-state acceptor with reversal-bounded counters. We also obtain similar results for more general machine models. Our characterizations can be used to verify certain properties concerning these machines that were not verifiable before using previous techniques. We are also able to formulate a subset of Presburger LTL that is decidable for satisfiability-checking with respect to these machines.

1 Introduction

Developing verification techniques for infinite-state systems is an important ongoing effort, motivated to a large extent by the successes of model-checking techniques for finite-state systems [23]. Unlike for finite-state systems, there is a decidability boundary for infinite-state systems: machines with two counters (i.e., “Minsky machines”) are Turing complete. Therefore, we must seek a balance between the computing power of infinite-state systems and their decidability.

Many infinite-state models have been shown decidable for various model-checking problems. These models include timed automata [1], pushdown automata and pushdown processes [3,14,12], various versions of counter machines [6,9,11,13,22], and various queue machines [2,4,19,20,24].

Pushdown systems are of particular interest, since, in practice, they are related to pushdown processes and, in theory, they are well studied in automata theory. A pushdown machine can be obtained by augmenting a finite-state machine with a pushdown stack. A configuration of a pushdown machine without an input tape (PM), is a string $\alpha = wq$, where w is the stack content and q

[★] Supported in part by NSF grant IRI-9700370

is the state (we assume that the stack alphabet is disjoint from the state set). If M is a PM and S is a set of configurations, define the backward and forward reachability sets of M with respect to S by: $pre^*(M, S) = \{\alpha \mid \text{configuration } \alpha \text{ can reach some configuration in } S\}$ and $post^*(M, S) = \{\alpha \mid \text{configuration } \alpha \text{ is reachable from some configuration in } S\}$. It is known that if S is regular, then $pre^*(M, S)$ and $post^*(M, S)$ are also regular (see, e.g., [3,12,14]). One can also show that the binary reachability of M , $Binary(M) = \{(\alpha, \beta) \mid \beta \text{ is reachable from } \alpha\}$, can be accepted by a 2-tape FA, i.e., a finite-state acceptor with two one-way input tapes. (Note that a 1-tape FA is the usual finite automaton.)

A PM augmented with finitely many real-valued clocks is called a pushdown timed automaton, which is a generalization of a timed automaton [1]. It is discrete if the clocks can only assume nonnegative integer values (definitions are in Section 4). A configuration now includes the clock values written in unary. It is easy to show that in general, the binary reachability of a (discrete) pushdown timed automaton cannot be accepted by a 2-tape FA. Characterizations of the binary reachability of pushdown timed automata with discrete or real-valued clocks have recently been given in [10,8]. In particular, it was shown in [10] (see also [21]) that the binary reachability of a discrete pushdown timed automaton can be accepted by a 2-tape pushdown acceptor augmented with reversal-bounded counters. A counter (which, we assume w.l.o.g., can only store nonnegative integers, since the sign can be remembered in the states) is reversal-bounded if it can be tested for zero and can be incremented or decremented by one, but the number of alternations between nondecreasing mode and nonincreasing mode in any computation is bounded by a given constant; e.g., a counter whose values change according to the pattern 0 1 1 2 3 4 4 3 2 1 0 1 1 0 is 3-reversal, where the reversals are underlined. It follows that the backward and forward reachability sets can be accepted by (1-tape) pushdown acceptors with reversal-bounded counters. These results and the fact that the emptiness problem for multitape pushdown acceptors with reversal-bounded counters is decidable [16] have been used recently to prove the decidability of certain verification problems for infinite-state transition systems [10,17,18,21,22].

In this paper, we improve the above results by showing that the pushdown stack can be *removed* from the acceptors. Specifically, we show that the binary (backward or forward) reachability can be accepted by a 2-tape (1-tape) finite-state acceptor with reversal-bounded counters. In fact, we show that the results hold, even if the discrete pushdown timed automaton is augmented with reversal-bounded counters. Note that equipping the pushdown timed automaton with counters is an important and nontrivial generalization, since it is known that reversal-bounded counters can “verify” Presburger relations on clock values [16].

The results in this paper can be used to verify properties that were not verifiable before using previous techniques. For example, consider the set $W = pre^*(M_1, pre^*(M_2, R_2) \cap R_1)$, where M_1 and M_2 are discrete pushdown timed automata with reversal-bounded counters (with the same states, pushdown alphabet, clock and counter names), and R_1 and R_2 are two sets of configurations accepted by finite-state acceptors augmented with reversal-bounded counters.

We know from [21,10] that $pre^*(M_2, R_2)$ can be accepted by a pushdown acceptor with reversal-bounded counters. Without using our current result, a direct construction of a machine accepting W would require two stacks: one for the machine accepting $pre^*(M_2, R_2)$, and the other is for M_1 itself. This seems to show that the emptiness of W may be undecidable, since a 2-stack machine is equivalent to a Turing machine. However, it follows from our results that W can be accepted by a finite-state acceptor with reversal-bounded counters; hence, the emptiness of W is decidable. As an application, consider the satisfiability-checking (the dual of model-checking) of a property $\Diamond(P_1 \wedge \Diamond P_2)$ concerning a discrete pushdown timed automaton with reversal-bounded counters M , where P_1 and P_2 are Presburger formulas on stack symbol counts and clock and counter values. This problem is reducible to checking the emptiness of $pre^*(M, pre^*(M, P_2) \cap P_1)$, which we now know is decidable.

We also look at discrete timed automata with clocks, reversal-bounded counters, and a read/write worktape (instead of a pushdown stack), but restricted to be *finite-crossing*, i.e., in any computation, the number of times the read/write head crosses the boundary between any two adjacent worktape cells is bounded by a given constant. We show that the binary (backward or forward) reachability set of this machine can also be accepted by a 2-tape (1-tape) finite-state acceptor with reversal-bounded counters. This improves the corresponding results in [18] where the the acceptors needed a finite-crossing read/write tape. Note that without the “finite-crossing” requirement, the model becomes a Turing machine.

We will use the following notation. We use the suffix ‘M’ to indicate that the model has no input tape and ‘A’ when the model has one-way input tape(s). All models are nondeterministic.

1. PM: Pushdown machine
2. PA: Pushdown acceptor
3. PCM: Pushdown machine with reversal-bounded counters
4. PCA: Pushdown acceptor with reversal-bounded counters
5. FM: Finite-state machine
6. FA: Finite-state acceptor
7. CM: Finite-state machine with reversal-bounded counters
8. CA: Finite-state acceptor with reversal-bounded counters
9. WCM: Finite-state machine with a read/write worktape and reversal-bounded counters
10. WCA: Finite-state acceptor with a read/write worktape and reversal-bounded counters
11. k -tape PCA (FA, CA, WCA) is a PCA (FA, CA, WCA) with k input tapes, with one head per tape. A 1-tape PCA (FA, ...) will simply be referred to as a PCA (FA, ...)
12. PTCM (WTCM) is a PCM (WCM) augmented with discrete clocks.

The paper has four sections in addition to this section. Section 2 shows that the binary reachability of a PCM can be accepted by a 2-tape CA and that the backward and forward reachability sets can be accepted by CAs. Section

3 shows that these results hold for finite-crossing WCMs. Section 4 generalizes the results to PCMs and finite-crossing WCMs with “clocks” (i.e., the timed versions of the models). Finally, Section 5 proposes a subset of Presburger LTL whose satisfiability-checking is decidable.

2 PCMs

We first look at the simple case of a PM (pushdown machine without counters). We assume that the pushdown stack has a “bottom” symbol B_0 , and is associated with two kinds of stack operations: $\text{push}(q, Z, q')$, i.e., push symbol Z onto the stack and switch from state q to state q' , and $\text{pop}(q, Z, q')$, i.e., pop the top Z from the stack and switch from state q to state q' . Replacing the top symbol of the stack with another symbol can be implemented by a push followed by a pop.

Let M be a PM. Define predicates push^* and pop^* as follows: $\text{push}^*(q, T, Z, q')$ is true, if there is a sequence of moves of M such that, starting from state q with stack top symbol T , M does not pop this T , and the last move is a push of Z on top of T ending in state q' (Notice that, prior to this last move, the sequence may involve many pushes/pops.) Similarly, $\text{pop}^*(q, Z, T, q')$ is true, if there is a sequence of moves of M such that, starting from state q with stack top symbol T (and Z the symbol directly under T), M does not pop this Z and the result of the moves makes this Z the top of the stack and state q' . We also define $\text{stay}^*(q, T, q')$ to be true if M , starting from state q with stack top symbol T , can reach state q' without performing any stack operations.

Lemma 1. *Given a PM M , we can effectively compute the predicates push^* , pop^* , and stay^* .*

Define $\text{Binary}(M, S, T) = \{(\alpha, \beta) \mid \text{configuration } \alpha \in S \text{ can reach configuration } \beta \in T\}$. When $S = T =$ the set of all configurations, $\text{Binary}(M, S, T)$ will simply be written $\text{Binary}(M)$.

Theorem 1. *$\text{Binary}(M)$ of a PM M can be accepted by a 2-tape FA.*

Corollary 1. *Let M be a PM, and, S and T be sets of configurations of M accepted by FAs. Then $\text{Binary}(M, S, T)$ can be accepted by a 2-tape FA.*

The backward and forward reachability sets of a PM M with respect to a regular set of configurations S is regular [3,14,12]. This result is easily obtained from the corollary above.

Corollary 2. *Let M be a PM and S be a set of configurations of M accepted by an FA A_S . Then $\text{pre}^*(M, S) = \{\alpha \mid \text{configuration } \alpha \text{ can reach some configuration } \beta \text{ in } S\}$ and $\text{post}^*(M, S) = \{\beta \mid \text{configuration } \beta \text{ is reachable from some configuration } \alpha \text{ in } S\}$ can be accepted by FAs.*

We now consider the PCMs. The reversal-bounded counters in the PCMs complicate the constructions, since now we have to incorporate counters into the acceptors of the reachability sets. We need to prove some intermediate results.

Let $\mathbb{N}$ be the set of nonnegative integers and n be a positive integer. A subset S of $\mathbb{N}^n$ is a *linear set* if there exist vectors $\mathbf{v}_0, \mathbf{v}_1, \dots, \mathbf{v}_t$ in $\mathbb{N}^n$ such that

$$S = \{\mathbf{v} \mid \mathbf{v} = \mathbf{v}_0 + a_1\mathbf{v}_1 + \dots + a_t\mathbf{v}_t, \forall 1 \leq i \leq t, a_i \in \mathbb{N}\}.$$

The vectors $\mathbf{v}_0$ (the constant vector) and $\mathbf{v}_1, \dots, \mathbf{v}_t$ (the periods) are called *generators*. A set S is *semilinear* if it is a finite union of linear sets. Semilinear sets are precisely the sets definable by *Presburger formulas* [15].

There is a simple automaton that characterizes semilinear sets. Let M be a nondeterministic finite-state machine (*without* an input tape) with n counters for some $n \geq 1$. The computation of M starts with all the counters zero and the automaton in the start state. An atomic move of M consists of incrementing at most one counter by 1 and changing the state (decrements are not allowed). An n -tuple $\mathbf{v} = (i_1, \dots, i_n) \in \mathbb{N}^n$ is generated by M if M , when started from its initial configuration, halts in an accepting state with $\mathbf{v}$ as the contents of the counters. The set of all n -tuples generated by M is denoted by $G(M)$. We call this machine a C-generator. If the C-generator is augmented with a pushdown stack, the machine is called a PC-generator. Notice that counters in a generator are nondecreasing, i.e., 0-reversal-bounded. We will need the following lemma, which can easily be shown using the results in [16].

Lemma 2. *The following statements are equivalent for $S \subseteq \mathbb{N}^n$: a) S is a semilinear set; b) S can be generated by a C-generator; c) S can be generated by a PC-generator.*

Consider a PCM M with n counters. A configuration of M is now represented as a string $\alpha = wqd_1^{x_1}d_2^{x_2}\dots d_n^{x_n}$, where w is the stack content, q is the state, $d_1, \dots, d_n$ are distinct symbols, and $x_1, x_2, \dots, x_n$ are the values of the counters (thus the counter values are represented in unary). We will show that the binary reachability $\text{Binary}(M)$ can be accepted by a 2-tape CA.

To simplify matters, we convert M to another PCM M' with many more counters than M . Assume M starts from a configuration α and reaches another configuration β . M' operates like M , except that the counters can make at most one reversal. M' simulates M faithfully, except that when a counter c of M makes a reversal from nonincreasing to increasing or c starts to increment before any decrements were made after starting from configuration α , M' suspends the simulation but continues decreasing this counter to zero while simultaneously increasing a new counter c' (starting at zero). When c reaches zero, c' has the old value of c before the simulation was suspended. M' then resumes the simulation with c' taking the role of c . If c' later reverses from nonincreasing to increasing, a new counter c'' is deployed like before. In this way, each counter c of M making r reversals can be replaced by $(r + 1)/2$ counters $c_1, \dots, c_{(r+1)/2}$, where each one makes at most one reversal. Moreover, a configuration α of M translates to a corresponding configuration of M' , where the value of a counter c of M is identified with the value of one of the counters $c_1, \dots, c_{(r+1)/2}$. Clearly, if we can construct a 2-tape CA A' to accept $\text{Binary}(M')$, we can modify A' to a 2-tape CA A to accept $\text{Binary}(M)$.

Let M be a PCM. From the discussion above, we assume that the counters have been normalized, i.e., during the computation from one configuration to another, each counter c behaves as one of the following five patterns:

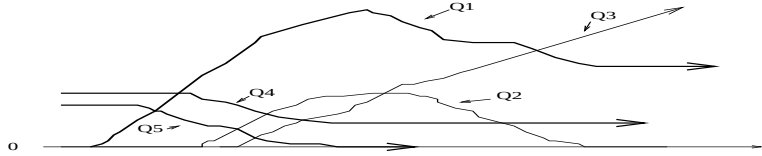


Fig. 1. Behavior patterns for a normalized counter

- c starts at zero, becomes positive, reverses (i.e., decrements), but remains positive.
- c starts at zero, becomes positive, reverses, becomes zero (and remains zero).
- c starts at zero, becomes positive, and does not reverse.
- c starts at a positive value, remains nonincreasing and positive.
- c starts at a positive value, remains nonincreasing, becomes zero (and remains zero).

We do not include the case when a counter remains at zero during the entire computation, since this can be simulated by an increment by 1 followed by a decrement by 1. Call the behaviors above Q_1 , Q_2 , Q_3 , Q_4 and Q_5 , respectively.

Consider a counter c that has behavior Q_1 . During the computation, c makes a mode change at three different instances: when it started at 0, became positive, and when it reversed. We denote these instances by 0 , $+$, rev . Note that c is positive at the end of the computation, since it has behavior Q_1 . In the construction to be described below, c will be simulated by two counters, c^+ and c^- , the first to record increments and the second to record decrements. If c is tested for zero during any segment of the simulation, the simulator assumes that it is zero before the mode changes to $+$ and positive after the mode has changed to $+$. (Note that the simulator knows when the mode changes.) At the end of the entire simulation, the simulator verifies that c is indeed positive by checking that $c^+ - c^-$ is positive.

Similarly, for a counter c with behavior Q_2 , the mode-change instances are: 0 , $+$, rev , $zero$. As in the above case, c will be simulated by two counters c^+ and c^- , and the simulator's action when testing for zero is like in the above case before the mode changes to $zero$. The point when the counter becomes zero (i.e. the mode changes to $zero$) is “guessed” by the simulator. After the mode has changed to $zero$, the simulator assumes that the counter is always zero when it is being tested (and c^+ and c^- will remain the same in the rest of the computation). At the end of the simulation, the simulator verifies that c is zero by checking that $c^+ = c^-$.

For the case of a counter c with behavior Q_3 , the mode-change instances are $0, +$. Like in the case for Q_1 , the simulator assumes the counter is zero before the mode changes to $+$ and positive after the mode has changed to $+$. Then c^+ is exactly c , and c^- will remain zero during the entire simulation. Note that for this case, there is nothing to verify at the end of the simulation.

For the case for Q_4 , the mode-change instance is rev . Counter c stays positive and c^+ will remain zero during the entire computation. The simulator checks that c^- is less than the starting value of c .

For the case of a counter c with behavior Q_5 , the mode-change instances are $rev, zero$. The simulator assumes the counter is positive before the mode changes to $zero$. Notice that the point that c becomes zero can be guessed by the simulator as described in the case for Q_2 . c^+ will remain zero during the entire simulation. Then the simulator checks that c^- is exactly the starting value of c .

When we say that a counter starts with mode m and ends with mode m' in a certain segment of the computation, we mean: 1) The counter is already in mode m , i.e., the mode-change to m has already been executed earlier; 2) If $m' \neq m$, the mode-change to m' occurs during the segment of computation under consideration.

In describing a subcomputation of the machine, we refer to $\langle c, Q_i, m, m' \rangle$ as a mode vector for c , and this means that counter c has behavior Q_i ($i = 1, 2, 3, 4, 5$) and in the subcomputation, c starts with mode m and ends with mode m' . We denote $\langle c, Q_i, m, m' \rangle$ simply as $V(c, Q_i)$, when m and m' are understood.

Let M be a PCM with n counters: $c_1, \dots, c_n$. We associate with each counter c two counters c^+ and c^- . Given $Q_{i_1}, \dots, Q_{i_n}, q, Z, q'$ (each $i_j \in \{1, 2, 3, 4, 5\}$), define a set of $2n$ -tuples of nonnegative integers $\text{push}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, Z, q')$ as follows: $(u_1, \dots, u_n, v_1, \dots, v_n)$ is in $\text{push}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, Z, q')$ if there is a sequence of moves of M such that,

1. The computation starting from state q with stack top symbol T , M does not pop this T , and the last move is a push of Z on top of T ending in state q' (Notice that, prior to this last move, the sequence may involve many pushes/pops.).
2. The computation remains within the specified mode vectors of the counters.
3. For $i = 1, \dots, n$, u_i (v_i) is the number of times counter i is incremented (decremented) by 1. So, for example, for $V(c_1, Q_2, 0, 0)$, $u_1 = 0$ and $v_1 = 0$; for $V(c_1, Q_2, 0, +)$, $u_1 > 0$ and $v_1 = 0$; for $V(c_1, Q_2, 0, rev)$, $u_1 > 0$ and $v_1 > 0$; for $V(c_1, Q_2, rev, rev)$, $u_1 = 0$ and $v_1 \geq 0$; for $V(c_1, Q_2, rev, zero)$, $u_1 = 0$ and $v_1 \geq 0$; for $V(c_1, Q_2, zero, zero)$, $u_1 = 0$ and $v_1 = 0$, etc.

Thus, $\text{push}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, Z, q')$ gives separate counts of the total increments and total decrements for each counter of M during the computation.

Similar to $\text{pop}^*(q, Z, T, q')$ and $\text{stay}^*(q, T, q')$ for a PM, we can define the sets $\text{pop}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), Z, T, q')$ and $\text{stay}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, q')$.

Lemma 3. *We can construct C-generators for $\text{push}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, Z, q')$, $\text{pop}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), Z, T, q')$, and $\text{stay}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, q')$.*

Proof. First we construct a PC-generator B with $2n$ counters which simulates the computation of M starting in state q with its stack top T . During the simulation, B makes sure that items 1 and 2 are satisfied. The simulation halts when M writes Z on the top of symbol T and moves right in state q' . From Lemma 2, B can be converted to an equivalent C-generator for

$$\text{push}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, Z, q').$$

Similarly, we can construct C-generators for

$$\text{pop}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), Z, T, q')$$

and $\text{stay}^*(q, V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}), T, q')$. $\square$

For notational convenience, $(V(c_1, Q_{i_1}), \dots, V(c_n, Q_{i_n}))$ will simply be denoted by V and will be called a global mode vector. Note that there are only a finite number of distinct global mode vectors. We use $A_{\text{push}}(q, V, T, Z, q')$, $A_{\text{pop}}(q, V, Z, T, q')$, and $A_{\text{stay}}(q, V, T, q')$ to denote the C-generators for $\text{push}^*(q, V, T, Z, q')$, $\text{pop}^*(q, V, Z, T, q')$ and $\text{stay}^*(q, V, T, q')$, respectively.

Let V and V' be two global mode vectors. Let $\langle c, Q_i, m, m' \rangle$ be a mode vector for c in V and $\langle c, Q_j, m'', m''' \rangle$ the corresponding mode vector for c in V' . We say that V and V' are compatible with respect to counter c if $Q_i = Q_j$, $m' = m''$, and m''' is a proper mode for Q_i (so, e.g., *rev* and *zero* are not proper for Q_3 ; *zero* is not proper for Q_1). Two global mode vectors V and V' are compatible if they are compatible with respect to every counter c . We are now ready to prove:

Theorem 2. *Binary(M) of a PCM M can be accepted by a 2-tape CA.*

Proof. From definition, $\text{Binary}(M) = \{(\alpha, \beta) \mid \text{configuration } \alpha \text{ can reach configuration } \beta \text{ in } M\}$. We construct a 2-tape CA B to accept $\text{Binary}(M)$. The specifications of all the C-generators $A_{\text{push}}(q, V, T, Z, q')$, $A_{\text{pop}}(q, V, Z, T, q')$, and $A_{\text{stay}}(q, V, T, q')$ are incorporated in the states of B . We describe the operation of B when given configurations α and β on its first and second input tapes, respectively. Let $\alpha = wqd_1^{x_1} \dots d_n^{x_n}$ and $\beta = w'q'd_1^{x'_1} \dots d_n^{x'_n}$. Let $w = Z_1 \dots Z_k$ and $w' = Z'_1 \dots Z'_{k'}$.

B reads the two tapes in parallel and makes sure that the symbol under head 1 is the same as the symbol under head 2. Nondeterministically, B starts to operate in the following way. Assume that both heads are at the m -th ($m \geq 1$) cell with $Z_1 \dots Z_{m-1} = Z'_1 \dots Z'_{m-1}$. There are four cases to consider (nondeterministically chosen):

- Case 1. $m \leq k$ and $m \leq k'$.
- Case 2. $m \leq k$ and $m = k' + 1$.
- Case 3. $m = k + 1$ and $m \leq k'$.

Case 4. $m = k + 1 = k' + 1$.

Consider Case 1. B operates in two phases. In the first phase, B reads the rest of the first input tape and guesses a sequence of pop-generators (when $m = 1$, treat Z_{m-1} as the stack bottom B_0)

$$A_{\text{pop}}(q_0, V_0, Z_{m-1}, Z_m, q_1), \dots, A_{\text{pop}}(q_{k-m}, V_{k-m}, Z_{k-1}, Z_k, q_{k-m+1})$$

such that V_{i+1} and V_i are compatible and each $\text{pop}^*(q_{i+1}, V_{i+1}, Z_{i+m-1}, Z_{i+m}, q_i)$ is not empty for $i = 0, \dots, k - m$, and $q_{k-m+1} = q$. Further, V_{k-m} is consistent with the starting counter values $x_1, \dots, x_n$, e.g., if the behavior of counter c_1 in V_{k-m} is Q2 (c_1 starts from 0), then x_1 must be 0. Note that each counter c in M is associated with two counters c^+ and c^- in the C-generators to keep track of the increments and decrements in counter c . In order to decide the counter values at the beginning of the second phase below, B guesses the value y_i for each counter c_i , and verifies, at the end of phase 1 by using auxiliary counters, that $y_i + \Sigma c_i^- - \Sigma c_i^+ = x_i$ where Σc_i^+ (resp. Σc_i^-) is the total increments (decrements) made to counter c_i for all the pop generators in phase 1. Note that “increments” in each pop generator essentially means “decrements” to y_i , since the pop generators are supposed to change the values of the c_i ’s from x_i ’s to y_i ’s. Doing this ensures that configuration $Z_1 \dots Z_k q d_1^{x_1} \dots d_n^{x_n}$ can reach the intermediate configuration γ (i.e., $Z_1 \dots Z_{m-1} q_0 d_1^{y_1} \dots d_n^{y_n}$) through a sequence of moves that never pops symbol Z_{m-1} .

Now, B starts phase 2, with counter values $y_1, \dots, y_n$ for counters $c_1, \dots, c_n$ in M . B then reads the rest of the second input tape and guesses a sequence of push generators

$$A_{\text{push}}(p_0, U_0, Z'_{m-1}, Z'_m, p_1), \dots, A_{\text{push}}(p_{k^\square-m}, U_{k^\square-m}, Z'_{k^\square-1}, Z'_{k^\square}, p_{k^\square-m+1})$$

such that $p_0 = q_0$ and V_0 and U_0 are compatible (i.e., M continues its computation from the intermediate configuration γ that was reached from the starting configuration α). B also checks that U_i and U_{i+1} are compatible and each $\text{push}^*(p_i, U_i, Z'_{i+m-1},$

$Z'_{i+m}, p_{i+1})$ is not empty for $i = 0, \dots, k' - m$, and $p_{k^\square-m+1} = q'$. In order to verify the intermediate configuration γ can reach configuration β , B needs to check (similar to phase 1) that $y_i - \Sigma c_i^- + \Sigma c_i^+ = x'_i$ where Σc_i^+ (resp. Σc_i^-) is the total increments (decrements) made to counter c_i for all the push generators in phase 2. Finally, B needs to check that the ending counter values $x'_1, \dots, x'_n$ are consistent with the last mode vector $U_{k^\square-m}$. For instance, if counter c_1 has behavior pattern Q4 in $U_{k^\square-m}$, then x'_1 must be positive. B accepts if all the guesses are successful, i.e., α can reach β .

Cases 2 - 4 are handled similarly, where the C-generators $A_{\text{stay}}(q, V, T, q')$ for $\text{stay}^*(q, V, T, q')$ are also used in the construction.

Hence, $\text{Binary}(M)$ can be accepted by a 2-tape CA B . $\square$

As in Corollaries 1 and 2, we have:

Corollary 3. *Let M be a PCM and S and T be sets of configurations of M accepted by CAs. Then $\text{Binary}(M, S, T)$ can be accepted by a 2-tape CA, and $\text{pre}^*(M, S)$ and $\text{post}^*(M, S)$ can be accepted by CAs.*

3 Finite-Crossing WCMs

Let M be a finite-crossing WCM with n counters. A configuration of M is represented as a string $\alpha = w_1 q w_2 d_1^{x_1} d_2^{x_2} \dots d_n^{x_n}$, where $w = w_1 w_2$ is the content of the read/write worktape with the head at the leftmost symbol of w_2 , q is the state, $d_1, \dots, d_n$ are distinct symbols, and $x_1, x_2, \dots, x_n$ are the values of the counters. We can prove the following:

Theorem 3. *Let M be a finite-crossing WCM. Then $\text{Binary}(M)$ can be accepted by a 2-tape CA.*

Corollary 4. *Let M be a finite-crossing WCM and S and T be sets of configurations of M accepted by CAs. Then $\text{Binary}(M, S, T)$ can be accepted by a 2-tape CA, and $\text{pre}^*(M, S)$ and $\text{post}^*(M, S)$ can be accepted by CAs.*

Corollary 5. *Let M be a finite-crossing WCM without counters (i.e., the only memory structure is a finite-crossing read/write worktape) and S and T be regular sets. Then $\text{Binary}(M, S, T)$ can be accepted by a 2-tape FA, and $\text{pre}^*(M, S)$ and $\text{post}^*(M, S)$ can be accepted by FAs.*

4 WCMs and PCMs with Clocks

A timed automaton is a finite-state machine *without* an input tape augmented with finitely many real-valued unbounded clocks [1]. All the clocks progress synchronously with rate 1, except that when a nonempty subset of clocks are reset to 0 at some transition, the other clocks do not progress. A transition between states fires if a clock constraint is satisfied. A clock constraint is a Boolean combination of atomic clock constraints in the following form: $x \# c$, $x - y \# c$ where $\#$ denotes $\leq, \geq, <, >$, or $=$, c is an integer, x, y are clocks. Here we only consider integer-valued clocks, i.e., discrete timed automata. A discrete pushdown timed automaton (finite-crossing worktape timed automaton) is a discrete time automaton with a pushdown stack (finite-crossing read/write tape). We can further generalize these models by augmenting them with reversal-bounded counters, call them PTCM and finite-crossing WTCM, respectively. Thus a PTCM (finite-crossing WTCM) is a PCM (finite-crossing WCM) with clocks. A configuration of a PTCM (finite-crossing WTCM) now contains the values of the clocks. It is known that the binary reachability of a PTCM (finite-crossing WTCM) can be accepted by a 2-tape PCA (finite-crossing WCA) [10,21]. The results in the previous section can be generalized:

Theorem 4. *Let M be a PTCM (or a finite-crossing WTCM). Then $\text{Binary}(M)$ can be accepted by a 2-tape CA.*

Corollary 6. *Let M be a PTCM (or a finite-crossing WTCM) and S and T be sets of configurations of M accepted by CAs. Then $\text{Binary}(M, S, T)$ can be accepted by a 2-tape CA, and $\text{pre}^*(M, S)$ and $\text{post}^*(M, S)$ can be accepted by CAs.*

5 Model-Checking and Satisfiability-Checking

It is important to formulate what kinds of temporal properties are decidable for the machine models discussed in this paper. Given a machine (a PM, PCM, finite-crossing WCM, or its timed version) M and a configuration α , we use α_{c_i} to denote the value of counter c_i in α , $\alpha_{\#_a}$ to denote the number of appearances of symbol a in the stack/tape content in α , $\alpha_{\mathbf{q}}$ to denote the state in α . Let P be a Presburger formula on variables α_{c_i} , $\alpha_{\#_a}$, and $\alpha_{\mathbf{q}}$. Since the solutions of P can be accepted by a deterministic CA [16], it is obvious that the set of configurations satisfying P can be accepted by a deterministic CA. Particularly, if P is a Boolean combination of atomic formulas like $x > k$, $x = k$, where x is a variable (α_{c_i} , $\alpha_{\#_a}$, or $\alpha_{\mathbf{q}}$), and k is an integer, then P is called a *regular* formula. Obviously, the set of configurations satisfying a regular formula A can be accepted by a FA.

Now, we describe a (subset of a) Presburger linear temporal logic $\mathcal{L}$ as follows. This logic is inspired by the recent work of Comon and Cortier [5] on model-checking a special form of counter automata without nested cycles. Formulas in $\mathcal{L}$ are defined as:

$$f ::= P \mid A \mid P \wedge f \mid f \vee f \mid \circ f \mid A U f$$

where P is a Presburger formula, A is a regular formula, $\circ$ and U stand for *next* and *until*, respectively. Formulas in $\mathcal{L}$ are interpreted on (finite) sequences p of configurations of M in a usual way. We use p^i to denote the sequence resulting from the deletion of the first i configurations from p . We use p_i to indicate the i -th element in p . The satisfiability relation $\models$ is recursively defined as follows, for each sequence p and for each formula $f \in \mathcal{L}$ (written $p \models f$):

$p \models P$ if $p_1 \in P$,
 $p \models A$ if $p_1 \in A$,
 $p \models P \wedge f$ if $p \models P$ and $p \models f$,
 $p \models f_1 \vee f_2$ if $p \models f_1$ or $p \models f_2$,
 $p \models \circ f$ if $p^1 \models f$,
 $p \models A U f$ if there exists j (which is not greater than the length of p) such that $p^j \models f$ and $\forall k < j (p^k \models A)$.

We use the convention that $\diamond f$ (eventual) abbreviates $(\text{true} U f)$. The satisfiability-checking problem is to check whether there is an execution p of M satisfying $p \models f$, for a given M and $f \in \mathcal{L}$. The model-checking problem, which is the dual of the satisfiability-checking problem, is to check whether for all execution p of M satisfying $p \models f$, for a given M and $f \in \mathcal{L}$. The results of this paper show that:

Theorem 5. *The satisfiability-checking problem is decidable for $\mathcal{L}$ with respect to the following machine models: PM, PCM, finite-crossing WCM, and their timed versions.*

Proof. (sketch) Given $f \in \mathcal{L}$, we use $[f]$ to denote the set of p such that $p \models f$. For each of the machine models, we will show $[f]$ can be accepted by a CA. There-

fore, the theorem follows by noticing that the satisfiability-checking problem is equivalent to testing the emptiness of the CA, which is decidable.

We will only look at PCM; all the other models can be handled similarly. The proof is based upon an induction on the structure of $\mathcal{L}$. Obviously, $[P]$ and $[A]$ can be accepted by CAs; so can $[f_1 \vee f_2]$ if both $[f_1]$ and $[f_2]$ can. $[P \wedge f]$ can be accepted by a CA, since $[f]$ can be accepted by a CA and $[P]$ can be accepted by a deterministic CA. For the case of $[A \text{ U } f]$, notice that the set $[A \text{ U } f]$ is very similar to $\text{Pre}^*(M, [f])$ – the only difference is that $[A \text{ U } f]$ further requires that each intermediate configuration on the path leading to $[f]$ to be in A . This requirement can be easily fulfilled by slightly modify M , thanks to the fact that A is regular. Therefore, Corollary 3 still applies to show that $[A \text{ U } f]$ can be accepted by a CA. The case for $[\circ f]$ is simpler, since we only look at one move. $\square$

$\mathcal{L}$ is quite powerful. For instance, it can express a property like $\diamond(P_1 \wedge \diamond P_2)$. We should point out that without using the results in this paper, this property cannot be checked. For instance, the timed version of PM was studied in [10]. In that paper, it was shown that $[P_1 \wedge \diamond P_2]$ can be accepted by a PCA – this is bad, since it is not possible to characterize $[\diamond(P_1 \wedge \diamond P_2)]$ from here (a machine accepting $[\diamond(P_1 \wedge \diamond P_2)]$ may need two stacks (i.e., Turing): one stack is for the PM, the other is for $[P_1 \wedge \diamond P_2]$). But now, we have a stronger characterization for $[P_1 \wedge \diamond P_2]$: it can be accepted by a CA. Therefore, the results in this paper give a CA characterization for $[\diamond(P_1 \wedge \diamond P_2)]$.

Since the model-checking problem is the dual of the satisfiability-checking problem, we conclude that

Theorem 6. *The model-checking problem is decidable for $\neg\mathcal{L}$ (taking negation of each formula in $\mathcal{L}$) with respect to the following machine models: PM, PCM, finite-crossing WCM, and their timed versions.*

Acknowledgment

We wish to thank Nicholas Tran for helpful discussions concerning this work.

References

1. R. Alur and D. Dill. “The theory of timed automata,” *TCS*, 126(2):183-236, 1994 [244](#), [245](#), [253](#)
2. P. Abdulla and B. Jonsson. “Verifying programs with unreliable channels,” *Information and Computation*, 127(2): 91-101, 1996 [244](#)
3. A. Bouajjani, J. Esparza, and O. Maler. “Reachability analysis of pushdown automata: application to model-checking,” *CONCUR’97*, **LNCS** 1243, pp. 135-150 [244](#), [245](#), [247](#)
4. G. Cece and A. Finkel. “Programs with Quasi-Stable Channels are Effectively Recognizable,” *CAV’97*, **LNCS** 1254, pp. 304-315 [244](#)

5. H. Comon and V. Cortier. "Flatness is not a weakness," *Proc. Computer Science Logic*, 2000 [254](#)
6. H. Comon and Y. Jurski. "Multiple counters automata, safety analysis and Presburger arithmetic," *CAV'98*, **LNCS** 1427, pp. 268-279 [244](#)
7. H. Comon and Y. Jurski. "Timed automata and the theory of real numbers," *CONCUR'99*, **LNCS** 1664, pp. 242-257
8. Z. Dang. "Binary reachability analysis of timed pushdown automata with dense clocks," *CAV'01*, **LNCS** 2102, pp. 506-517 [245](#)
9. Z. Dang and O. H. Ibarra. "Liveness verification of reversal-bounded counter machines with a free counter," submitted, 2001 [244](#)
10. Z. Dang, O. H. Ibarra, T. Bultan, R. A. Kemmerer and J. Su. "Binary Reachability Analysis of Discrete Pushdown Timed Automata," *CAV'00*, **LNCS** 1855, pp. 69-84 [245](#), [246](#), [253](#), [255](#)
11. Z. Dang, P. San Pietro, and R. A. Kemmerer. "On Presburger Liveness of Discrete Timed Automata," *STACS'01*, **LNCS** 2010, pp. 132-143 [244](#)
12. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. "Efficient Algorithms for Model Checking Pushdown Systems," *CAV'00*, **LNCS** 1855, pp. 232-247 [244](#), [245](#), [247](#)
13. A. Finkel and G. Sutre. "Decidability of Reachability Problems for Classes of Two Counter Automata," *STACS'00*, **LNCS** 1770, pp. 346-357 [244](#)
14. A. Finkel, B. Willems, and P. Wolper. "A direct symbolic approach to model checking pushdown systems," *INFINITY'97* [244](#), [245](#), [247](#)
15. S. Ginsburg and E. Spanier. "Bounded Algol-like languages," *Transactions of American Mathematical Society*, 113, pp. 333-368, 1964. [248](#)
16. O. H. Ibarra. "Reversal-bounded multicounter machines and their decision problems," *J. ACM*, **25** (1978) 116-133 [245](#), [248](#), [254](#)
17. O. H. Ibarra. "Reachability and safety in queue systems with counters and pushdown stack," *Proceedings of the International Conference on Implementation and Application of Automata*, pp. 120-129, 2000. [245](#)
18. O. H. Ibarra, T. Bultan, and J. Su. "Reachability analysis for some models of infinite-state transition systems," *CONCUR'00*, pp. 183-198, 2000. [245](#), [246](#)
19. O. H. Ibarra. "Reachability and safety in queue systems with counters and pushdown stack," *Proceedings of the International Conference on Implementation and Application of Automata*, pp. 120-129, 2000 [244](#)
20. O. H. Ibarra, Z. Dang, and P. San Pietro, "Verification in Loosely Synchronous Queue-Connected Discrete Timed Automata," submitted. 2001 [244](#)
21. O. H. Ibarra and J. Su. "Generalizing the discrete timed automaton," *Proceedings of the International Conference on Implementation and Application of Automata*, 206-215, 2000. [245](#), [246](#), [253](#)
22. O. H. Ibarra, J. Su, T. Bultan, Z. Dang, and R. A. Kemmerer. "Counter Machines: Decidable Properties and Applications to Verification Problems," *MFCS'00*, **LNCS** 1893, pp. 426-435 [244](#), [245](#)
23. K. L. McMillan. "Symbolic model-checking - an approach to the state explosion problem," PhD thesis, Department of Computer Science, Carnegie Mellon University, 1992 [244](#)
24. W. Peng and S. Purushothaman. "Analysis of a Class of Communicating Finite State Machines," *Acta Informatica*, 29(6/7): 499-522, 1992 [244](#)

A New Recognition Algorithm for Extended Regular Expressions[□]

Hiroaki Yamamoto

Department of Information Engineering, Shinshu University,

4-17-1 Wakasato, Nagano-shi, 380-8553 Japan

Fax: +81-26-269-5495

yamamoto@cs.shinshu-u.ac.jp

Abstract. This paper is concerned with the recognition problem for extended regular expressions: given an extended regular expression r of length m and an input string x of length n , determine if $x \in L(r)$, where $L(r)$ denotes the language denoted by r . For this problem, the algorithm based on dynamic programming which runs in $O(mn^3)$ time and $O(mn^2)$ space is widely known. We here introduce a structure called *a modular tree* and present a new automata-based recognition algorithm such that it runs in $O(mn^2 + kn^3)$ time and $O(mn + kn^2)$ space. Here k is a number derived from a modular tree and is less than the number of intersection and complement operators in r . Furthermore, k can be much smaller than m for many extended regular expressions. Thus our algorithm significantly improves the time and space complexities of the classical dynamic programming algorithm.

1 Introduction

This paper is concerned with the recognition problem for extended regular expressions (that is, regular expressions with intersection and complement). Given an extended regular expression r of length m and an input string x of length n , the recognition problem is to determine if $x \in L(r)$, where $L(r)$ denotes the language denoted by r . It is widely known that such a recognition problem can be applied to the pattern matching problem [1,3,5,7,8,9]. The standard recognition algorithm for regular expressions runs in $O(mn)$ time and $O(m)$ space, based on nondeterministic finite automata (NFAs for short) [1,2,4,6]. Myers [7] has improved this algorithm so that it runs in $O(mn/\log n)$ time and space. Thus, for regular expressions, efficient algorithms based on NFAs have been shown, and have been used for the pattern matching problem. However, any efficient algorithm based on finite automata is not known for extended regular expressions. Although extended regular expressions also denote only regular sets, they shorten the length of the expressions needed to describe certain regular sets. It is, therefore, important to design an efficient recognition algorithm for extended

[□] This research has been supported by the Kayamori Foundation of Informational Science Advancement

regular expressions. When we try to translate extended regular expressions to NFAs in the standard way, the number of states multiplicatively increases for intersection and exponentially increases for complement. Since operators can be nested, the number of states can be exponentiated on the order of m times for an expression with m operators. This suggests that any algorithm which uses this translation as one of its step for extended regular expressions is going to be an exponential-time algorithm, and hence another approach has been taken. For example, as seen in [6], the existing algorithm uses a dynamic programming technique. The aim of this paper is to show that we can design an efficient automata-based recognition algorithm for extended regular expressions.

Recently, Yamamoto [10] introduced a new notion of synchronization called input-synchronization, and in [11] he has given a new recognition algorithm for semi-extended regular expressions using new automata called partially input-synchronized alternating finite automata (PISAFAs). This algorithm is faster than the existing one. Thus he has shown that an automata-based technique can be used for semi-extended regular expressions as well as regular expressions for the first time. Although he says that the algorithm runs in $O(mn^2)$ time, it seems to hold only for a subset of semi-extended regular expressions. Our result includes a result refined for semi-regular expressions.

In this paper, we will extend Yamamoto's technique and give an efficient algorithm for extended regular expressions. In addition, we give a more refined analysis for the complexities. Yamamoto's algorithm is based on the linear translation from semi-extended regular expressions to PISAFAs. However, we cannot find out the similar translation for extended regular expressions, because it is difficult to translate complement into a PISAFa. For this reason, we will take advantage of a natural structure called *a modular tree* derived from hierarchical structure of extended regular expressions. Nodes of the modular tree are called *a module*, which denotes a regular expression. The modular tree leads to an inductive construction of NFAs so that we can design an automata-based algorithm. In this approach, the concepts of alternation and input-synchronization such as in [11] explicitly do not appear, but they are implicitly used. Our main result is as follows:

Let r be an extended regular expression of length m over an alphabet Σ , let $\mathcal{T}_r$ be the modular tree of r , and let x be an input string of length n in Σ^* . Then we can design a recognition algorithm which determines if $x \in L(r)$ such that

1. if the height of $\mathcal{T}_r$ is equal to 0, then the algorithm runs in $O(mn)$ time and $O(m)$ space,
2. if the height of $\mathcal{T}_r$ is equal to 1, then the algorithm runs in $O(mn^2)$ time and $O(mn)$ space,
3. if the height of $\mathcal{T}_r$ is ≥ 2 , then the algorithm runs in $O(mn^2 + kn^3)$ time and $O(mn + kn^2)$ space, where $k = CRIT(\mathcal{T}_r)$.

For example, if r is a regular expression, then the height of $\mathcal{T}_r$ is 0. The formal definitions of a modular tree $\mathcal{T}_r$ and $CRIT(\mathcal{T}_r)$ will appear in Section 3.

The value of $CRIT(\mathcal{T}_r)$ is less than the number of extended operators (that is, intersection and complement), and can be much smaller than m for many extended regular expressions. Since the algorithm based on a dynamic programming technique takes $O(mn^3)$ time and $O(mn^2)$ space (see [6]), our algorithm significantly improves the classical dynamic programming algorithm. Thus our result says that automata-theoretic techniques are also applicable for extended regular expressions.

Throughout the paper, as in [1,6,7,8,9,11], we rely on a $\log n$ -bit uniform RAM, that is, each $\log n$ -bit instruction is executed in one unit of time and each $\log n$ -bit register is stored in one unit of space, to evaluate all complexities appearing in this paper.

The paper is organized as follows. In Section 2, we give basic definitions. In Section 3, we discuss an inductive construction of NFAs, and in Section 4 we give a new algorithm.

2 Extended Regular Expressions, Parse Trees and NFAs

We here give some definitions for extended regular expressions.

Definition 1. *Let Σ be an alphabet. The extended regular expressions over Σ are defined as follows.*

1. $\emptyset$, ϵ (the empty string) and a ($a \in \Sigma$) are extended regular expressions that denote the empty set, the set $\{\epsilon\}$ and the set $\{a\}$, respectively.
2. Let r_1 and r_2 be extended regular expressions denoting the sets R_1 and R_2 , respectively. Then $(r_1 \vee r_2)$, $(r_1 r_2)$, (r_1^*) , $(r_1 \wedge r_2)$ and $(\neg r_1)$ are also extended regular expressions that denote the sets $R_1 \cup R_2$, $R_1 R_2$, R_1^* , $R_1 \cap R_2$, and $\overline{R_1}$ ($= \Sigma^* - R_1$), respectively.

Regular expressions are defined by three operators $(r_1 \vee r_2)$, $(r_1 r_2)$ and (r_1^*) , and semi-extended regular expressions are defined by four operators $(r_1 \vee r_2)$, $(r_1 r_2)$, (r_1^*) and $(r_1 \wedge r_2)$. To take advantage of hierarchical structure of extended regular expressions, we introduce their parse trees. Let r be an extended regular expression. Then the parse tree T_r is defined as follows:

1. If $r = \emptyset$ (ϵ , a , respectively), then T_r is a tree consisting of just one node labeled by $\emptyset$ (ϵ , a , respectively).
2. If $r = r_1 \vee r_2$ ($r = r_1 \wedge r_2$, $r = r_1 r_2$, $r = r_1^*$, $r = \neg r_1$, respectively), then T_r is a tree such that its root is labeled by $\vee$ ($\wedge$, $\cdot$, $*$, $\neg$, respectively) and the left subtree and the right subtree of the root are T_{r_1} and T_{r_2} ($*$ and $\neg$ have only T_{r_1}), respectively. The operator “.” means concatenation.

The following theorem is widely known as the linear translation from regular expressions to NFAs (for example, see [6]).

Theorem 1. *Let r be a regular expression of length m . Then we can construct an NFA M such that M has at most $O(m)$ states and accepts the language denoted by r .*

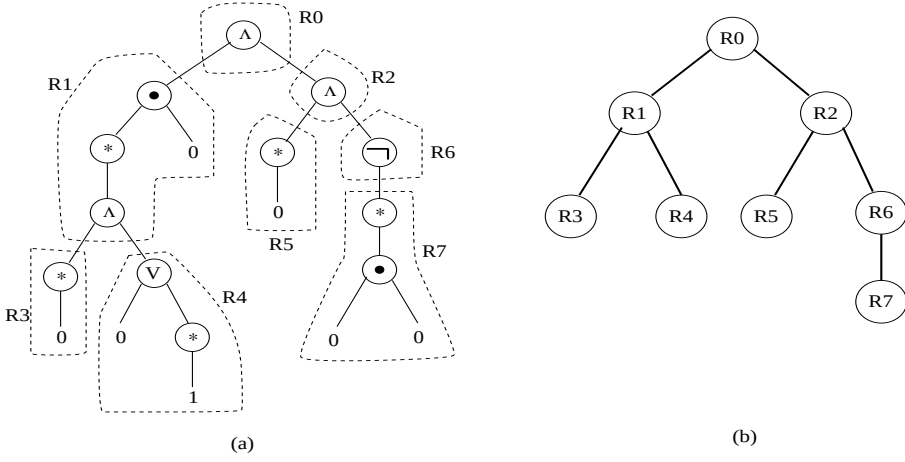


Fig. 1. Parse tree and its partition for $r = ((0^* \wedge (0 \vee 1^*))^* 0) \wedge (0^* \wedge \neg((00)^*))$

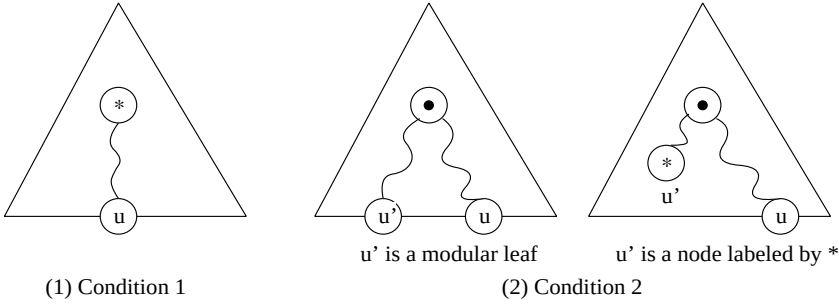


Fig. 2. Conditions in Definition 2

3 Inductive Construction of NFAs

We cannot find out the similar translation to that in [11] for extended regular expressions. Therefore, we design an algorithm by inductively constructing NFAs from hierarchical formation of extended regular expressions. Let r be an extended regular expression over an alphabet Σ and let T_r be the parse tree of r . Then, we partition T_r by nodes labeled with intersection $\wedge$ and complement $\neg$ into subtrees such that (1) the root of each subtree is either a child of a node labeled with $\wedge$ or $\neg$ in T_r or the root of T_r , (2) each subtree does not contain any interior nodes labeled by $\wedge$ or $\neg$, (3) each leaf is labeled by $a \in \Sigma$, $\wedge$ or $\neg$. If it is labeled by $\wedge$ ($\neg$, respectively), then it is called a *universal leaf* (a *negating leaf*, respectively). These leaves are also called a *modular leaf*. We call such a subtree a *module*.

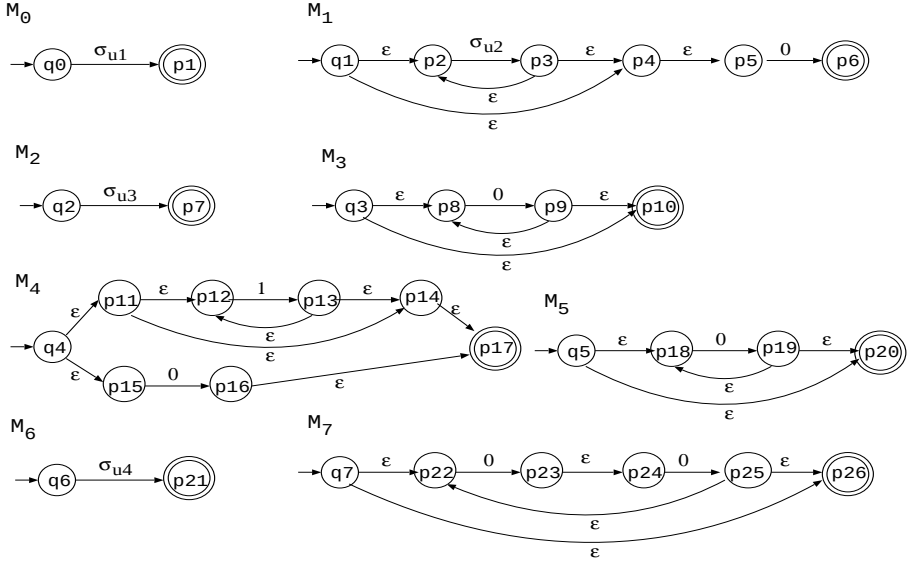


Fig. 3. Augmented NFAs for modules in Fig.1

Let R and R' be modules in the parse tree T_r . If a modular leaf u of R becomes the parent of the root of R' in T_r , then R is called a *parent of R'* , and conversely R' is called a *child of R* or a *child of R at u* . Thus there are two children at each universal leaf, while one child at each negating leaf. If the root of a module R is the root of T_r , then R is called a *root module*. If a module R does not have any children, then R is called a *leaf module*. It is clear that such a parent-child relationship induces a *modular tree* $\mathcal{T}_r = (\mathcal{R}, \mathcal{E})$ such that (1) $\mathcal{R}$ is a set of modules, (2) $(R, R') \in \mathcal{E}$ if and only if R is the parent of R' .

Fig.1(a) shows an example of partition of T_r for $r = ((0^* \wedge (0 \vee 1^*))^*0) \wedge (0^* \wedge (\neg((00)^*)))$, and Fig.1(b) shows the modular tree. In Fig.1, R_0 is the root module, R_3, R_4, R_5 and R_7 are leaf modules.

The height of a modular tree $\mathcal{T}_r$ is defined as follows. For any module R of $\mathcal{T}_r$, if R is the root, then the depth of R is 0; otherwise $h + 1$, where h is the depth of the parent of R . Then the height of $\mathcal{T}_r$ is defined to be the maximum depth over all modules. Now we introduce a parameter which plays a crucial role in the analysis of time and space complexities.

Definition 2. Let R be any module in $\mathcal{T}_r$. Then, we say a modular leaf u of R to be critical if u satisfies at least one of the following two conditions: (1) u has an ancestor labeled by $*$, (2) there is a modular leaf u' or a node u' labeled by $*$ such that u' and u have a common ancestor labeled by concatenation, and u' is on the left-side of u in R . See Fig. 2.

Definition 3. Let $\text{CRIT}(R)$ be the number of critical modular leaves in R , and let $\mathcal{R}' = \mathcal{R} - \{R_0\}$, where $\mathcal{R}$ is the set of modules and R_0 is the root module. Then, $\text{CRIT}(\mathcal{T}_r) = \sum_{R \in \mathcal{R}'} \text{CRIT}(R)$.

For example, $\text{CRIT}(\mathcal{T}_r) = 1$ for Fig.1, because only R_1 has just one critical modular leaf. Clearly, $\text{CRIT}(\mathcal{T}_r)$ is less than the number of extended operators (intersection and complement). Furthermore, if the height of $\mathcal{T}_r$ is 1, then $\text{CRIT}(\mathcal{T}_r) = 0$, because leaf modules do not have any critical modular leaves.

Now, for each module R , we relabel every modular leaf u of R with a new symbol σ_u called a *modular symbol*. By this relabeling, R can be viewed as a regular expression over $\Sigma \cup \{\sigma_u \mid u \text{ is a modular leaf of } R\}$. Then, by Theorem 1, we can construct an NFA M_R for a module R . Let us call this M_R an *augmented NFA* (A-NFA for short). In addition, we call a state q in M_R a *universal state* if there is $\delta(q, \sigma_u) = q'$ for a universal leaf u , and call a state q in M_R a *negating state* if there is $\delta(q, \sigma_u) = q'$ for a negating leaf u . The other states are called an *existential state*. Furthermore, if a module R' is a child of R at u , then A-NFA $M_{R'}$ is said to be *associated with* σ_u . Fig. 3 gives an example for A-NFAs, where q_0 , p_2 and q_2 are universal states, and q_6 is a negating state. M_1 and M_2 are associated with σ_{u_1} , M_3 and M_4 are associated with σ_{u_2} , M_5 and M_6 are associated with σ_{u_3} , and M_7 is associated with σ_{u_4} . It is clear that the following theorem is obtained from Theorem 1.

Theorem 2. Let r be an extended regular expression of length m and let $R_0, \dots, R_l$ be modules produced by partitioning $\mathcal{T}_r$. Then we can construct A-NFAs M_j for each module R_j such that $\sum_{0 \leq j \leq l} \text{STATE}(M_j)$ is at most $O(m)$, where $\text{STATE}(M_j)$ denotes the number of states of M_j .

4 Recognition Algorithm for Extended Regular Expressions

Our algorithm becomes an extension of the algorithm based on NFAs for regular expressions. The main part of the algorithm is the simulation of a set of A-NFAs. This is done using a data structure called a *directed computation graph*. In this section, we first give the definition of a directed computation graph, and then the recognition algorithm.

4.1 A Directed Computation Graph

Let r be an extended regular expression over an alphabet Σ and let $x = a_1 \cdots a_n$ be an input string in Σ^* . We first partition $\mathcal{T}_r$ into modules $R_0, \dots, R_l$ and construct A-NFAs $M_0, \dots, M_l$ for each module as described in Theorem 2. Here R_0 is the root module. After that, to determine if $x \in L(r)$, we simulate the set $\{M_0, \dots, M_l\}$ of A-NFAs on x . This time, each A-NFA M_j satisfies the following properties.

Property 1 For any state q , the number of transitions from q is at most two.

Property 2 For any state q , all the transitions from q are labeled by the same symbol a . If a is either a symbol in Σ or a modular symbol, then the number of transitions is exactly one.

Property 3 The number of final states is exactly one.

To simulate each A-NFA M_j ($0 \leq j \leq l$), we introduce a set of states, called an *existential-element set*. For the root module R_0 , we use just one existential-element set U_0^0 to simulate M_0 . For other modules R_j ($1 \leq j \leq l$), we use at most $n + 1$ existential-element sets U_j^i ($0 \leq i \leq n$) to simulate M_j . A set U_j^i is used to simulate M_j on $a_i \cdots a_n$ using a simple state-transition simulation. Namely, U_j^i always maintains states reachable from the initial state q_j of M_j after M_j has read $a_i \cdots a_{i^0}$ for any $i \leq i' \leq n$. To simulate the set $\{M_0, \dots, M_l\}$, we will construct a *directed computation graph* $\mathcal{G} = (\mathcal{U}, \mathcal{E})$ such that (1) $\mathcal{U}$ is the set of *nodes*, which consists of existential-element sets, and $\mathcal{E}$ is the set of *edges*, which consists of pairs (U, U') of nodes, (2) there is the special node, called a *source node*, which has no incoming edges, (3) Nodes with no outgoing edges are called a *sink node*, (4) let $U_{j_1}^i, U_{j_2}^{i^0}$ and $U_{j_3}^{i^0}$ be nodes of $\mathcal{U}$ for A-NFAs M_{j_1}, M_{j_2} and M_{j_3} , respectively. Then there exist directed edges $(U_{j_1}^i, U_{j_2}^{i^0})$ and $(U_{j_1}^i, U_{j_3}^{i^0})$ in $\mathcal{E}$ if and only if R_{j_2} and R_{j_3} are two children of R_{j_1} at a universal leaf u and M_{j_1} reaches the universal state corresponding to u while processing a_{i^0} , (5) let $U_{j_1}^i$ and $U_{j_2}^{i^0}$ be nodes of $\mathcal{U}$ for A-NFAs M_{j_1} and M_{j_2} , respectively. Then there exists a directed edge $(U_{j_1}^i, U_{j_2}^{i^0})$ in $\mathcal{E}$ if and only if R_{j_2} is the child of R_{j_1} at a negating leaf u and M_{j_1} reaches the negating state corresponding to u while processing a_{i^0} .

4.2 Outline of the Simulation

The simulation starts with $\mathcal{U} = \{U_0^0\}$ and $\mathcal{E} = \emptyset$, where $U_0^0 = \{q_0\}$ and q_0 is the initial state of M_0 . We update a directed computation graph each time reading an input symbol. Suppose that $\mathcal{G} = (\mathcal{U}, \mathcal{E})$ is a directed computation graph obtained after processing $a_1 \cdots a_{i-1}$. This time, note that $\mathcal{G}$ satisfies the property that for any $U_{j_1}^{i_1} \in \mathcal{U}$, $q \in U_{j_1}^{i_1}$ if and only if M_{j_1} can reach the state q from the initial state q_{j_1} of M_{j_1} by $a_{i_1} \cdots a_{i-1}$. Then we will show how $\mathcal{G} = (\mathcal{U}, \mathcal{E})$ is updated by a_i . The computation for a_i consists of two main procedures, *GoTo* and *EClosure*.

The procedure *GoTo* simply computes states reachable from a state in $\mathcal{U}$ by a_i . We first perform the procedure *GoTo*. Next, we perform the procedure *EClosure*. This procedure simulates ϵ -moves of A-NFAs. That is, for any existential-element set $U_{j_1}^{i_1} \in \mathcal{U}$ and any state $q \in U_{j_1}^{i_1}$, we compute all states reachable from q by continuous ϵ -moves. To avoid a redundant computation, *EClosure* performs the computation from sink nodes towards the source node. The computation is classified into three cases according to the kind of a state q as follows: (1) q is existential, (2) q is universal, and (3) q is negating.

If q is an existential state with $\delta(q, \epsilon) = Q'$, then we simply add all the states in Q' to $U_{j_1}^{i_1}$.

If q is a universal state with $\delta(q, \sigma_u) = q'$, then do the following: Let R_{j_1} and R_{j_2} be two children of R_j at the universal leaf u and let q_{j_1} and q_{j_2} be the initial states of M_{j_1} and M_{j_2} , respectively. Then we add two nodes $U_{j_1}^i = \{q_{j_1}\}$ and $U_{j_2}^i = \{q_{j_2}\}$ to $\mathcal{U}$ and add two edges $(U_{j_1}^{i_1}, U_{j_1}^i)$ and $(U_{j_1}^{i_1}, U_{j_2}^i)$ to $\mathcal{E}$. This time, we call a pair $(U_{j_1}^i, U_{j_2}^i)$ a *universal pair*. Furthermore, in the subsequent computation, we check whether or not $U_{j_1}^i$ and $U_{j_2}^i$ have the final states of M_{j_1} and M_{j_2} , respectively, each time reading an input symbol a_{i_2} ($i_2 \geq i$). If they both have a final state, then q' is added to $U_j^{i_1}$. This means that the transition $\delta(q, \sigma_u) = q'$ is possible if and only if M_{j_1} and M_{j_2} accept the same substring $a_i \cdots a_{i_2}$.

If q is a negating state with $\delta(q, \sigma_u) = q'$, then do the following: Let R_{j_1} be the child of R_j at a negating leaf u and let q_{j_1} be the initial state of M_{j_1} . Then we add a node $U_{j_1}^i = \{q_{j_1}\}$ to $\mathcal{U}$, and add an edge $(U_j^{i_1}, U_{j_1}^i)$ to $\mathcal{E}$. This time, we call a node $U_{j_1}^i$ a *negating node*. Furthermore, in subsequent simulation, we check whether or not $U_{j_1}^i$ contains the final state of M_{j_1} at every time. If it does not contain the final state, then we add q' to $U_j^{i_1}$. This means that the transition $\delta(q, \sigma_u) = q'$ is possible if and only if M_{j_1} does not accept a substring $a_i \cdots a_{i_2}$.

After computing all states reachable by ϵ -moves, we finally obtain the updated $\mathcal{G} = (\mathcal{U}, \mathcal{E})$. The above process is repeatedly performed from a_1 through a_n . Let $\mathcal{G}$ be the directed computation graph obtained after processing a_n . In order to determine if x is in $L(r)$, we check whether the source node U_0^0 contains the final state of M_0 or not. If U_0^0 contains it, then our algorithm accepts x ; otherwise rejects x .

4.3 Algorithm in Detail

Now let us give the detail of the algorithm below. Given an extended regular expression r of length m and an input string x of length n , the computation starts with the following ACCEPT.

Algorithm ACCEPT(r, x)

Input: an extended regular expression r , an input string x .

Output: If $x \in L(r)$, then return *YES*; otherwise return *NO*.

Step 1. Partition T_r into modules $R_0, \dots, R_l$.

Step 2. Translate each module R_j ($0 \leq j \leq l$) to an A-NFA M_j .

Step 3. Let M_0 be an A-NFA for the root module R_0 . Then, execute $\text{SIMULATE}(M_0, x, q_0, q_f)$, and if it returns *YES*, then output *YES*; otherwise output *NO*. Here q_0 and q_f is the initial state and the final state of M_0 , respectively.

Function SIMULATE(M, x, q_0, q_f)

Input: An A-NFA M , a string $x = a_1 \cdots a_n$, the initial state q_0 of M and the final state q_f of M .

Output: If M accepts x , then return *YES*; otherwise *NO*.

Comment: This function directly simulates M starting from the state q_0 .

Step 1. Initialization.

1. Set $\mathcal{G} = (\mathcal{U}, \mathcal{E})$, where $\mathcal{U} = \{U_0^0\}$, $U_0^0 := \{q_0\}$, and $\mathcal{E} = \emptyset$.
2. $EClosure(\mathcal{G}, 0)$.

Step 2. For $i = 1$ to n , do the following:

1. $GoTo(\mathcal{G}, a_i)$.
2. $EClosure(\mathcal{G}, i)$.

Step 3. If U_0^0 contains the final state q_f , then return *YES*; otherwise return *NO*.

To perform $EClosure$ efficiently, we introduce the depth for each node U . This can straightforwardly be defined according to the depth of each module as follows. For any node $U \in \mathcal{U}$, let R be the module in $\mathcal{T}_r$ such that the A-NFA for R is simulated by U . Then the depth of U is defined to be just the depth of R . Let h_{max} and h_{min} be the maximum depth and the minimum depth over $\mathcal{U}$, respectively. We can partition $\mathcal{U}$ into some subsets $\mathcal{U}^{h_{min}}, \dots, \mathcal{U}^{h_{max}}$ by the depth of each node U such that $\mathcal{U}^h$ consists of nodes with the depth h . Then the following $EClosure$ simulates ϵ -moves in the order from $\mathcal{U}^{h_{max}}$ to $\mathcal{U}^{h_{min}}$.

Procedure $EClosure(\mathcal{G}, i)$

$\mathcal{G} = (\mathcal{U}, \mathcal{E})$: a directed computation graph;

i : an input position;

Step 1. For $h = h_{max}$ to h_{min} , do the following:

1. $EpsilonMove(\mathcal{U}^h, i)$.
2. $SyncCheck(\mathcal{U}^h)$.
3. $NegCheck(\mathcal{U}^h)$.

Procedure $EpsilonMove(\mathcal{U}', i)$

$\mathcal{U}'$: a subset of $\mathcal{U}$;

i : an input position;

Step 1. For all $U_j^{i_1} \in \mathcal{U}'$ (note that $U_j^{i_1}$ is for an A-NFA M_j), do the following:

1. $U_{old} := \emptyset$.
2. while $U_{old} \neq U_j^{i_1}$ do the following:
 - (a) $U_{old} := U_j^{i_1}$.
 - (b) For all $q \in U_j^{i_1}$, do the following:
 - i. If q is an existential state, then $U_j^{i_1} := U_j^{i_1} \cup \delta(q, \epsilon)$.
 - ii. If q is a universal state with $\delta(q, \sigma_u) = q'$, then do the following:
 Here, note that $U_{j_1}^i$ and $U_{j_2}^i$ are existential-element sets for A-NFAs M_{j_1} and M_{j_2} which are associated with σ_u , respectively. Furthermore, M_{j_1} and M_{j_2} have the initial states q_{j_1} and q_{j_2} , respectively.
 - A. if both $U_{j_1}^i$ and $U_{j_2}^i$ are already in $\mathcal{U}$, then $\mathcal{E} := \mathcal{E} \cup \{(U_j^{i_1}, U_{j_1}^i), (U_j^{i_1}, U_{j_2}^i)\}$, and if they both have a final state, then $U_j^{i_1} := U_j^{i_1} \cup \{q'\}$.

- B. if both $U_{j_1}^i$ and $U_{j_2}^i$ are not in $\mathcal{U}$ yet, then produce two nodes $U_{j_1}^i := \{q_{j_1}\}$ and $U_{j_2}^i := \{q_{j_2}\}$, and then $\mathcal{U} := \mathcal{U} \cup \{U_{j_1}^i, U_{j_2}^i\}$, $\mathcal{E} := \mathcal{E} \cup \{(U_{j_1}^{i_1}, U_{j_1}^i), (U_{j_1}^{i_1}, U_{j_2}^i)\}$. After that, do $EClosure((\{U_{j_1}^i\}, \emptyset), i)$ and $EClosure((\{U_{j_2}^i\}, \emptyset), i)$.
- iii. If q is a negating state with $\delta(q, \sigma_u) = q'$, then do the following:
 Here, note that $U_{j_1}^i$ is an existential-element set for A-NFA M_{j_1} which is associated with σ_u , and this M_{j_1} has the initial state q_{j_1} .
- A. if $U_{j_1}^i$ is already in $\mathcal{U}$, then $\mathcal{E} := \mathcal{E} \cup \{(U_{j_1}^{i_1}, U_{j_1}^i)\}$, and if it does not contain the final state, then $U_{j_1}^{i_1} := U_{j_1}^{i_1} \cup \{q'\}$.
- B. if $U_{j_1}^i$ is not in $\mathcal{U}$ yet, then $U_{j_1}^i := \{q_{j_1}\}$, and then $\mathcal{U} := \mathcal{U} \cup \{U_{j_1}^i\}$, $\mathcal{E} := \mathcal{E} \cup \{(U_{j_1}^{i_1}, U_{j_1}^i)\}$. After that, do $EClosure((\{U_{j_1}^i\}, \emptyset), i)$.

Procedure SyncCheck($\mathcal{U}'$)

$\mathcal{U}'$: a subset of $\mathcal{U}$;

Step 1. For all universal pairs $(U_{j_1}^{i_1}, U_{j_2}^{i_1})$ in $\mathcal{U}'$, if $q_{f_1} \in U_{j_1}^{i_1}$ and $q_{f_2} \in U_{j_2}^{i_1}$, then for each U such that $(U, U_{j_1}^{i_1}) \in \mathcal{E}$, do $U := U \cup \{q\}$. Here, q_{f_1} and q_{f_2} are the final states of M_{j_1} and M_{j_2} which are associated with σ_u , respectively, and q is a state such that $\delta(q', \sigma_u) = q$.

Procedure NegCheck($\mathcal{U}'$)

$\mathcal{U}'$: a subset of $\mathcal{U}$;

Step 1. For all negating nodes $U_{j_1}^{i_1}$ in $\mathcal{U}'$, if $q_{f_1} \notin U_{j_1}^{i_1}$, then for each U such that $(U, U_{j_1}^{i_1}) \in \mathcal{E}$, do $U := U \cup \{q\}$. Here, q_{f_1} is the final state of M_{j_1} which is associated with σ_u , and q is a state such that $\delta(q', \sigma_u) = q$.

Procedure GoTo($\mathcal{G}, a$)

$\mathcal{G} = (\mathcal{U}, \mathcal{E})$: a directed computation graph;

a : an input symbol;

Step 1. For all $U \in \mathcal{U}$, do the following:

1. For all $q \in U$, do the following:
 - (a) If $\delta(q, a) = \{q'\}$, then $U := (U - \{q\}) \cup \{q'\}$,
 - (b) If $\delta(q, a) = \emptyset$, then $U := U - \{q\}$.

Since the translation to A-NFAs can be done in $O(m)$ time using the standard parser, we have the following main theorem.

Theorem 3. *Given an extended regular expression r of length m and an input string x of length n , the algorithm *ACCEPT* correctly determines if $x \in L(r)$, and*

1. *if the height of the modular tree $\mathcal{T}_r$ is 0, then the algorithm runs in $O(mn)$ time and $O(m)$ space,*
2. *if it is ≥ 1 , then the algorithm runs in $O(mn^2 + kn^3)$ time and $O(mn + kn^2)$ space, where $k = \text{CRIT}(\mathcal{T}_r)$.*

As mentioned before, note that if the height of $\mathcal{T}_r$ is 1, then $k = 0$. Hence, in this case, the algorithm runs in $O(mn^2)$ time and $O(mn)$ space.

References

1. A. V. Aho, Algorithms for finding patterns in strings, In J. V. Leeuwen, ed. Handbook of theoretical computer science, Elsevier Science Pub., 1990. [257](#), [259](#)
2. A. Apostolico, Z. Galil ed., Pattern Matching Algorithms, Oxford University Press, 1997. [257](#)
3. R. A. Baeza-Yates and G. H. Gonnet, Fast Text Searching for Regular Expressions or Automaton Searching on Tries, J. ACM, 43,6, 915-936, 1996. [257](#)
4. C. H. Chang, and R. Paige, From regular expressions to DFA's using compressed NFA's, Theoret. Comput. Sci., 178, 1-36, 1997. [257](#)
5. J. R. Knight and E. W. Myers, Super-Pattern matching, Algorithmica, 13, 1-2, 211-243, 1995. [257](#)
6. J. E. Hopcroft and J. D. Ullman, Introduction to automata theory language and computation, Addison Wesley, Reading Mass, 1979. [257](#), [258](#), [259](#)
7. G. Myers, A Four Russians Algorithm for Regular Expression Pattern Matching, J. Assoc. Comput. Mach. 39,4, 430-448, 1992. [257](#), [259](#)
8. E. Myers and W. Miller, Approximate Matching of Regular Expressions, Bull. of Mathematical Biology, 51, 1, 5-37, 1989. [257](#), [259](#)
9. S. Wu, U. Manber and E. Myers, A Sub-Quadratic Algorithm for Approximate Regular Expression Matching, J. of Algorithm, 19, 346-360, 1995. [257](#), [259](#)
10. H. Yamamoto, On the Power of Input-Synchronized Alternating Finite Automata, Proc. COCOON2000, LNCS 1858, 457-466, 2000. [258](#)
11. H. Yamamoto, An Automata-based Recognition Algorithm for Semi-extended Regular Expressions, Proc. MFCS2000, LNCS 1893, 699-708, 2000. [258](#), [259](#), [260](#)

Polynomial-Time Algorithms for the Equivalence for One-Way Quantum Finite Automata

Takeshi Koshihara

Secure Computing Lab., Fujitsu Laboratories Ltd.,
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki 211-8588, Japan
koshihara@acm.org

Abstract. Two quantum finite automata are equivalent if for any string x the two automata accept x with equal probability. This paper gives a polynomial-time algorithm for determining whether two measure-once one-way quantum finite automata are equivalent. The paper also gives a polynomial-time algorithm for determining whether two measure-many one-way quantum finite automata are equivalent.

1 Introduction

A quantum finite automaton (QFA) is a theoretical model for a quantum computer with a finite memory. When restricted to finite memory machines, quantum mechanism in comparison to classical (non-quantum) one has both strengths and weaknesses. It is one of the most important problems to characterize the power of QFAs. Quantum finite automata were introduced independently by Moore and Crutchfield [10] and Kondacs and Watrous [9].

Kondacs and Watrous [9] showed that a two-way QFA could accept more than regular languages. They also restricted the head of a QFA to moving right on each transition and got the one-way QFA model. (Amano and Iwama [2] considered yet another model, called 1.5-way QFAs). During its computation, a one-way QFA performs measurements on its configuration. Since the acceptance capability of a one-way QFA depends on the measurements that the QFA may perform during the computation, two models of one-way QFAs that differ only in the type of measurement that they perform during the computation have been studied. Brodsky and Pippenger called the model introduced by Moore and Crutchfield [10] *measure-once QFA (MO-QFA)* and a similar model for the one introduced by Kondacs and Watrous [9] *measure-many QFA (MM-QFA)*. The main difference between the two models is that a measure-once QFA performs one measurement at the end of computation, while a measure-many QFA performs a measurement after every transition.

Kondacs and Watrous also [9] showed that the languages accepted by one-way MM-QFAs is a proper subset of the regular languages. Brodsky and Pippenger [7] provided a necessary condition for languages to be accepted by one-way MM-QFAs. Ambainis *et al.* [4] gave a characterization for regular languages to be accepted by one-way MM-QFAs. On the other hand, Brodsky and Pippenger [7]

showed that a language is accepted by a one-way MO-QFA (with bounded error) if and only if it is accepted by a group finite automaton. They also pointed out that a non-regular language is accepted by a one-way MO-QFA with unbounded error.

Besides characterization of the power of QFAs, it is also important to consider the decidability of the equivalence for QFAs. Two QFAs are equivalent if for any string x the two QFAs accept x with equal probability. Although Brodsky and Pippenger [7] showed that the equivalence for one-way MO-QFAs is decidable, they did not take notice of its efficiency. On the other hand, it has been still open whether or not the equivalence for one-way MM-QFAs is decidable (see, e.g., [8]).

In this paper, we show the equivalence for one-way MO-QFAs is decidable in polynomial time. Brodsky and Pippenger used the following three techniques to show the decidability of the equivalence for one-way MO-QFAs: (i) transformation from the representation of one-way MO-QFAs to generalized stochastic finite automata [10]; (ii) conversion from generalized stochastic finite automata to stochastic finite automata [12]; (iii) decidability of the equivalence for stochastic finite automata [12]. We use the technique by Tzeng [14] instead of Paz's two techniques (ii) and (iii). Actually, Tzeng showed a polynomial time algorithm to decide the equivalence for stochastic finite automata. We show that a slight observation enables to use Tzeng's algorithm in order to decide the equivalence for generalized finite stochastic automata in polynomial time. Moreover, we show that a one-way MM-QFA is simulated by a *generalized* one-way MO-QFA and it is also represented by a generalized stochastic finite automaton. By showing that the conversion can be done in polynomial time, we show a polynomial-time algorithm to decide the equivalence for one-way MM-QFAs.

2 Preliminaries

2.1 Selective Quantum Operations

In this subsection, we briefly review certain facts from quantum computation and state the definition of selective quantum processes that will be used through this paper. For a more thorough treatment of quantum computing, we refer the reader to the references like [6,11]. The definition of selective quantum operations, we adopt in this paper, is given in [11,15].

Let us consider quantum systems having finite classical state sets. Given a quantum system with a fixed classical state set S , a *pure state* of the system is unit vector in the Hilbert space $\ell_2(S)$. We use the Dirac notation to represent elements of $\ell_2(S)$; for each $s \in S$, $|s\rangle$ represents the unit vector corresponding to the map that takes s to 1 and each $s' \neq s$ to 0. Elements of $\ell_2(S)$ are specified by linear combinations of elements in the orthonormal basis $\{|s\rangle : s \in S\}$.

A *mixed state* is a state that may be described by a distribution on (not necessarily orthogonal) pure states. Intuitively, a mixed state represents the quantum state of a system given that we have limited knowledge of this state. A collection

$\{(p_k, |\psi_k\rangle)\}$ such that $0 \leq p_k$, $\sum_k p_k = 1$, and each $|\psi_k\rangle$ is a pure state is called *mixture*: for each k , the system is in superposition $|\psi_k\rangle$ with probability p_k . For a given mixture $\{(p_k, |\psi_k\rangle)\}$, we associate an $|S| \times |S|$ *density matrix* ρ having operator representation $\rho = \sum_k p_k |\psi_k\rangle\langle\psi_k|$. Necessary and sufficient conditions of a given $|S| \times |S|$ matrix ρ to be a density matrix are (i) ρ must be positive semidefinite, and (ii) ρ must have unit trace.

A *selective quantum operation* is a probabilistic mapping that takes as input a density matrix ρ and outputs a pair $(i, \rho^{(i)})$ with probability p_i , where $\rho^{(i)}$ is a density matrix and i is a classical output that we take to be an integer for simplicity. The output i may be the result of some measurement, although this is not the most general situation (for example, the system may be measured and part of outcome may be discarded). A selective quantum operation $\mathcal{E}$ must be described by a collection $\{A_{i,j} : 0 \leq i \leq m, 1 \leq j \leq l\}$ of $|S| \times |S|$ matrices satisfying the constraint $\sum_{i=0}^m \sum_{j=1}^l A_{i,j}^\dagger A_{i,j} = I$. Given such a collection of matrices, we define a function $p_i : \mathbb{C}^{n \times n} \rightarrow [0, 1]$ and a partial function $E_i : \mathbb{C}^{n \times n} \rightarrow \mathbb{C}^{n \times n}$ as follows:

$$p_i(\rho) = \text{tr} \left(\sum_{j=1}^l A_{i,j} \rho A_{i,j}^\dagger \right)$$

$$E_i(\rho) = \frac{1}{p_i(\rho)} \sum_{j=1}^l A_{i,j} \rho A_{i,j}^\dagger.$$

(In case $p_i(\rho) = 0$, $E_i(\rho)$ is undefined.) Now, on input ρ , the output of $\mathcal{E}$ is defined to be $(i, E_i(\rho))$ with probability $p_i(\rho)$ for each i . We also define a function F_i as $F_i(\rho) = \sum_{j=1}^l A_{i,j} \rho A_{i,j}^\dagger$ for each i . It will simplify matters when calculating unconditional probabilities to consider these functions.

2.2 One-Way Quantum Finite Automata

In this section, we give definitions of two models of one-way quantum finite automata. One is called *measure-once* one-way quantum finite automaton and the other is called *measure-many* one-way quantum finite automaton. Using the representation by selective quantum operations, it is not difficult to consider more general models of one-way quantum finite automata than both models of measure-once ones and measure-many ones. Although it is possibly that the *generic* one-way quantum automata may be more powerful, we focus on the two models of one-way quantum automata in this paper. We give definitions of the two models of one-way quantum finite automata in terms of selective quantum operations.

Measure-Once One-Way QFA and Its Generalization A *measure-once one-way quantum finite automaton* (MO-1QFA) is defined by a quintuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is a finite input alphabet,

δ is a transition function $\delta : Q \times \Gamma \times Q \rightarrow \mathbb{C}$ that represents the probability density amplitude that flows from state q to state q' upon reading symbol σ , the state q_0 is the initial state, and F is the set of accepting states, where $\Gamma = \Sigma \cup \{\#, \$\}$ is the tape alphabet of M and $\#$ and $\$$ are end-markers not in Σ . For all states $q_1, q_2 \in Q$ and symbols $\sigma \in \Gamma$ the function δ must be unitary, thus satisfying the condition

$$\sum_{q' \in Q} \delta^*(q_1, \sigma, q') \delta(q_2, \sigma, q') = \begin{cases} 1 & q_1 = q_2 \\ 0 & q_1 \neq q_2 \end{cases},$$

where δ^* is the complex conjugate of δ . We assume that all inputs are of the form $\#\sigma_1\sigma_2 \cdots \sigma_n\$$. At the end of a computation, M measures its configuration; if it is in an accepting state then it accepts, otherwise it rejects. This definition is equivalent to that of the QFA defined by Moore and Crutchfield.

The configuration of M is a pure state and is represented by an n -dimensional complex unit vector, where $n = |Q|$. This vector is denoted by

$$|\psi\rangle = \sum_{i=1}^n \alpha_i |q_i\rangle$$

where $\{|q_i\rangle\}$ is the set orthonormal basis vectors corresponding to the state of M . The coefficient α_i is the probability density amplitude of M being in state q_i . Since $|\psi\rangle$ is a unit vector, it follows that $\sum_{i=1}^n |\alpha_i|^2 = 1$. Sometimes, it is rather convenient to consider density matrices. Its density matrix is denoted by

$$\rho = |\psi\rangle\langle\psi|.$$

The transition function δ is represented by a set of unitary matrices $\{U_\sigma\}_{\sigma \in \Gamma}$ (or their corresponding selective quantum operations) where U_σ represents the unitary transition of M upon reading symbol σ . If M is in configuration $\rho = |\psi\rangle\langle\psi|$ and reads symbol σ then the new configuration of M is denoted by

$$\rho' = E(\rho) = U_\sigma |\psi\rangle\langle\psi| U_\sigma^\dagger.$$

Measurement is represented by a diagonal zero-one projection matrix $P (= P_{acc})$ where $P_{ii} = [q_i \in F]$. The probability of M accepting string x is defined by

$$p_M(x) = \text{tr}(P \rho_x P^\dagger)$$

where $\rho_x = U(x)|q_0\rangle\langle q_0|U(x)^\dagger$ and $U(x) = U_\$U_{x_n}U_{x_{n-1}} \cdots U_{x_1}U_\#$.

As stated in [10], we sometimes find it useful to relax the unitarity. A *measure-once generalized one-way quantum finite automaton* (MO-g1QFA) is one in which the matrices U_σ are not necessarily unitary.

Measure-Many One-Way QFA A *measure-many one-way quantum finite automaton* (MM-1QFA) is defined by a sextuple $M = (Q, \Sigma, \delta, q_0, Q_{acc}, Q_{rej})$

where Q is a finite set of states, σ is a finite input alphabet, δ is a unitary transition function of the same form as for an MO-1QFA, and the state q_0 is the initial state. The set Q is partitioned into three subsets: Q_{acc} is the set of halting accepting states, Q_{rej} is the set of halting rejecting states, and Q_{non} is the set of non-halting states.

The operation of an MM-1QFA is similar to that of an MO-1QFA except that after every transition M measures its configuration with respect to the three subspaces that correspond to the three subsets Q_{non} , Q_{acc} and Q_{rej} : $S_{non} = \text{span}(\{|q\rangle : q \in Q_{non}\})$, $S_{acc} = \text{span}(\{|q\rangle : q \in Q_{acc}\})$, and $S_{rej} = \text{span}(\{|q\rangle : q \in Q_{rej}\})$, where span is the function that maps a set of vectors to the vector space generated by the vectors in the set. If the configuration is in S_{non} then the computation continues; if the configuration is in S_{acc} then M accepts, otherwise it rejects. After every measurement the superposition collapses into the measured subspace and is renormalized.

The configuration of an MM-1QFA is a mixed state and is represented by a pair in $\{(non, \rho^{(non)}), (acc, \rho^{(acc)}), (rej, \rho^{(rej)})\}$, where $\rho^{(non)}$, $\rho^{(acc)}$ and $\rho^{(rej)}$ are density matrices. The transition function is represented by selective quantum operations only on $\rho^{(non)}$ and measurement is represented by diagonal zero-one projection matrices that project the vector onto the respective subspaces.

Since M can have a non-zero probability of halting partway through the computation, it is useful to keep track of the cumulative accepting and rejecting probabilities. Therefore, in some cases we use the representation of Kondacs and Watrous that represents the state of M as a triple (ρ, p_{acc}, p_{rej}) , where p_{acc} and p_{rej} are the cumulative probabilities of accepting and rejecting. The evolution of M on reading symbol σ is denoted by

$$(F_{non}(\rho), p_{acc} + p_{acc}(\rho), p_{rej} + p_{rej}(\rho)).$$

3 Real-Valued Bilinear Representation

In this paper, we discuss the probability that a string is accepted by a quantum finite automaton. Since we consider MO-1QFAs, MO-g1QFAs and MM-1QFAs, it is convenient to treat them uniformly as in Moore and Crutchfield [10].

A *generalized stochastic function* is a function from strings over an alphabet Σ to real numbers, $f : \Sigma^* \rightarrow \mathbb{R}$, for which there are real-valued vectors π and η and real-valued matrices M_σ for each $\sigma \in \Sigma$ such that f is a bilinear form,

$$f(x) = \eta^T \cdot M_x \cdot \pi$$

where $M_x = M_\$M_{x_n}M_{x_{n-1}} \cdots M_{x_1}M_\#$. We will call such a function *n-dimensional* if π, η and M_σ are *n-dimensional*.

If the components of η are 0 and 1 denoting nonaccepting and accepting states and if π and the rows of M_σ have non-negative entries that sum to 1, then f is a *stochastic function*.

It is well known that complex numbers $c = a + bi$ can be represented by 2×2 real matrices $\mathbf{c} = \begin{pmatrix} a & b \\ -b & a \end{pmatrix}$. In the same way, an $n \times n$ complex matrix can be

simulated by a $2n \times 2n$ real-valued matrix. Moreover, we note that this matrix is unitary if the original matrix is.

Here, we consider a bilinear real-valued representation (rather than quadratic one) of accepting probability for MO-g1QFA $(Q, \Sigma, \delta, q_0, F)$. Let h_i be a set of perpendicular unit vectors spanning S_{acc} . Then, we consider a function f in the following.

$$\begin{aligned} f(x) &= \sum_{i=1}^n \langle h_i | U_x | q_0 \rangle \langle q_0 | U_x^\dagger | h_i \rangle \\ &= \sum_{i=1}^n \langle h_i^* \otimes h_i | U_x^* \otimes U_x | q_0^* \otimes q_0 \rangle \\ &= \langle \sum_{i=1}^n h_i^* \otimes h_i | U_x^* \otimes U_x | q_0^* \otimes q_0 \rangle, \end{aligned}$$

where U_x is the matrix representation corresponding to $\delta(x)$. This has the form $\eta^T \cdot M_x \cdot \pi$ with $\pi = q_0^* \otimes q_0$, $M_\sigma = U_\sigma^* \otimes U_\sigma$ for all $\sigma \in \Gamma$, and $\eta = \sum_{i=0}^n h_i^* \otimes h_i$. Since these are the tensor products of n -dimensional objects, they have n^2 dimensions. Furthermore, using the representation above, we transform η^T , M_σ , and π into $2 \times 2n^2$, $2n^2 \times 2n^2$, and $2n^2 \times 2$ real-valued matrices $\bar{\eta}^T$, $\bar{M}_\sigma$ and $\bar{\pi}$, respectively, and

$$\bar{\eta}^T \cdot \bar{M}_\sigma \cdot \bar{\pi} = f(x) \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Letting η^T and π be the top row of $\bar{\eta}^T$ and the left column of $\bar{\pi}$, respectively, gives the desired real-valued bilinear form.

We call a generalized stochastic function whose representation consists of unitary transition matrices *generalized stochastic function with the unitarity*.

4 Equivalence for Generalized Stochastic Finite Automata

Tzeng showed a polynomial-time algorithm to decide the equivalence for stochastic finite automata [14]. A slight observation enables us to use Tzeng's algorithm in order to decide the equivalence for generalized stochastic finite automata in polynomial time.

A vector is *stochastic* if all its entries are real numbers greater than or equal to zero and sum to 1. A matrix is *stochastic* if all its row vectors are stochastic. A *stochastic finite automaton* (SFA) is defined by a quintuple $M = (Q, \Sigma, \delta, \pi, F)$, where Q is a finite set of states, Σ is a finite input alphabet, δ is a function from Σ to the set of all $n \times n$ stochastic matrices, and $F \subseteq Q$ is a set of final states.

The vector π is called an *initial-state distribution* where the i th component of π indicated the probability of state q_i being the initial state. The value $\delta(x)_{[i,j]}$ is the probability that M moves from state q_i to state q_j after reading symbol $\sigma \in \Sigma$. Let η_F be an n -dimensional row vector such that the i th entry is 1

if $q_i \in F$; the i th entry is 0 otherwise. The *state distribution* induced by string x for M is $D_M(x) = \delta(x)\pi$, where $\delta(x) = \delta(x_n)\delta(x_{n-1}) \cdots \delta(x_1)$. The *accepting probability* is $P_M(x) = (\eta_F)^T D_M(x)$.

A *generalized stochastic finite automaton* (gSFA) is a stochastic finite automaton in which π and δ are not necessarily stochastic.

Let $M_1 = (Q_1, \Sigma, \delta_1, \pi_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, \pi_2, F_2)$ be two gSFAs. Then M_1 and M_2 are said to be equivalent if and only if $P_{M_1}(x) = P_{M_2}(x)$ for any string $x \in \Sigma^*$.

We define, for each string x ,

$$\delta_{M_1 \oplus M_2}(x) = \begin{pmatrix} \delta_1(x) & O \\ O & \delta_2(x) \end{pmatrix}.$$

We also define, for each string x ,

$$D_{M_1 \oplus M_2}(x) = \delta_{M_1 \oplus M_2}(x) \begin{pmatrix} \pi_1 \\ \pi_2 \end{pmatrix}.$$

For two gSFAs M_1 and M_2 , let

$$H(M_1, M_2) = \{D_{M_1 \oplus M_2}(x) : x \in \Sigma^*\}.$$

Since the issue of computing with real numbers is subtle, in what follows we assume that all inputs consist of rational numbers and that each arithmetic operation on rational numbers can be done in constant time.

Now we are ready to consider the equivalence for gSFAs. It is easy to obtain a slight generalization of Tzeng's result in [14].

Theorem 1. *There is a polynomial-time algorithm that takes as input two gSFAs M_1 and M_2 and determines whether M_1 and M_2 are equivalent, where n_1 and n_2 are the number of states in M_1 and M_2 , respectively. Furthermore, if M_1 and M_2 are not equivalent then the algorithm outputs the lexicographically minimum string which is accepted by M_1 and M_2 with different probabilities. This string will always be of length at most $n_1 + n_2 - 1$.*

Proof. (Sketch) Although the assertion is similarly shown as in [14], we show a proof sketch for self-containment and for easy understandability.

First, recall that two gSFAs M_1 and M_2 are equivalent if and only if

$$\forall x \in \Sigma^*, \quad P_{M_1}(x) = P_{M_2}(x).$$

We can reformulate this equation as

$$\forall x \in \Sigma^*, \quad [(\eta_{F_1})^T, -(\eta_{F_2})^T] D_{M_1 \oplus M_2}(x) = 0.$$

We note that it is important that the following lemma in [14] also works for gSFAs.

Lemma 1. *Let M_1 and M_2 be two gSFAs. If V is a basis for $\text{span}(H(M_1, M_2))$ then M_1 and M_2 are equivalent if and only if for all $v \in V$, $[(\eta_{F_1})^T, -(\eta_{F_2})^T]v = 0$.*

Because the dimension of the vector space $\text{span}(H(M_1, M_2))$ is at most $n_1 + n_2$, the number of elements in V is at most $n_1 + n_2$. Thus if we are able to find such a basis in polynomial time then we can solve the equivalence problem for gSFAs in polynomial time.

Without loss of generality, we assume that $\Sigma = \{0, 1\}$. We define binary tree T as follows. Tree T will have a node for every string in Σ^* . The root of T is $\text{node}(\lambda)$, where λ is the empty string. Every $\text{node}(x)$ in T has two children $\text{node}(x0)$ and $\text{node}(x1)$. Let $D_{M_1 \oplus M_2}(x)$ be the $(n_1 + n_2)$ -dimensional vector associated with $\text{node}(x)$. For $\text{node}(x\sigma)$, $\sigma \in \Sigma$, its associated vector $D_{M_1 \oplus M_2}(x\sigma)$ can be calculated by multiplying its parent's associated vector $D_{M_1 \oplus M_2}(x)$ by $\delta_{M_1 \oplus M_2}(\sigma)$.

The method to determine whether M_1 and M_2 are equivalent is to prune tree T . Initially, we set V to be the empty set. We then visit the nodes in T in breadth-first order. At each $\text{node}(x)$, we verify whether its associated vector $D_{M_1 \oplus M_2}(x)$ is linearly independent of V . If it is, we add the vector to V . Otherwise, we prune the subtree rooted at $\text{node}(x)$. We stop traversing tree T when every node in T is either visited or pruned. The vectors in the resulting set V will be linearly independent. Actually, the vectors in V form a basis for $\text{span}(H(M_1, M_2))$. A breadth-first search guarantees that finding the lexicographically minimum string whose accepting probabilities by M_1 and M_2 are different.

The pruning algorithm is shown in Fig.1. We skip the discussion of the validity and the analysis of the time complexity of the algorithm, since the discussion is similar to the one in [14]. $\square$

Input: $M_1 = (Q_1, \{0, 1\}, \delta_1, \pi_1, F_1)$, $M_2 = (Q_2, \{0, 1\}, \delta_2, \pi_2, F_2)$;
 Set V and N to be the empty set;
 $queue \leftarrow \text{node}(\lambda)$;
while $queue$ is not empty **do**
 begin take an element $\text{node}(x)$ from $queue$;
 if $D_{M_1 \sqcup M_2}(x) \notin \text{span}(V)$ **then**
 begin add $\text{node}(x0)$ and $\text{node}(x1)$ to $queue$;
 add vector $D_{M_1 \sqcup M_2}(x)$ to V ;
 add $\text{node}(x)$ to N
 end;
 end;
 if $\forall v \in V, [(\eta_{F_1})^T, -(\eta_{F_2})^T]v = 0$ **then** return(*yes*)
 else return($\text{lex-min}\{x : \text{node}(x) \in N, [(\eta_{F_1})^T, -(\eta_{F_2})^T]D_{M_1 \sqcup M_2}(x) \neq 0\}$);
end;

Fig. 1. Algorithm for the equivalence for gSFAs

5 Polynomial-Time Equivalence for QFAs

In this section, we consider the equivalence for one-way quantum finite automata. The equivalence for MO-1QFAs is easily determined in polynomial time using the bilinearization technique, discussed in Section 3, and the equivalence for gSFAs. On the other hand, we show that the representation of an MM-1QFA is efficiently converted to the one of an MO-g1QFA. Using the similar discussion as the case of MO-1QFAs, we show that the equivalence for MM-1QFAs is also determined in polynomial time.

Theorem 2. *Any MO-1QFA has its representation of generalized stochastic function with the unitarity.*

Proof. Discussion in Section 3 follows the assertion. $\square$

Theorem 3. *Any MM-1QFA has its representation of generalized stochastic function.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, Q_{acc}, Q_{rej})$ be an MM-1QFA. We construct an MO-g1QFA $M' = (Q', \Sigma, \delta', q_0, F)$ as follows.

$$\begin{aligned} Q' &= Q \cup \{q_a : a \in \Sigma \cup \{\#, \$\}\} \setminus Q_{acc}, \\ F &= \{q_a : a \in \Sigma \cup \{\#, \$\}\}. \end{aligned}$$

Each unitary matrix corresponding to δ satisfying

$$U_\sigma |q\rangle = \dots + \alpha_i |q_i\rangle \dots + \alpha_A |q_A\rangle \quad \text{and} \quad q_A \in Q_{acc}$$

is replaced with the following matrix

$$U'_\sigma |q\rangle = \dots + \alpha_i |q_i\rangle \dots + \alpha_A |q_\sigma\rangle,$$

which is not necessarily unitary. We also add the following rules.

$$U'_\sigma |q_\sigma\rangle = |q_\sigma\rangle \quad \text{for all } q_\sigma \in F.$$

We note that the above transformation violates the unitarity. Since each new accepting state in F corresponds the differential of the cumulative probability of accepting with respect to the original MM-1QFA, the accepting probability is preserved. The rest of the proof is similarly shown as Theorem 2. $\square$

Now, we are ready to consider algorithms for equivalence for one-way quantum finite automata. In case of MO-1QFAs, the theorem below immediately follows from Theorem 1, Theorem 2 and the bilinearization technique in Section 3.

Theorem 4. *There is a polynomial-time algorithm that takes as input two MO-1QFAs M_1 and M_2 and determines whether M_1 and M_2 are equivalent, where n_1 and n_2 are the number of states in M_1 and M_2 , respectively. Furthermore, if M_1 and M_2 are not equivalent then the algorithm outputs the lexicographically minimum string which is accepted by M_1 and M_2 with different probabilities. This string will always be of length $O((n_1 + n_2)^2)$.*

In case of MM-1QFAs, since the conversion from MM-1QFAs to MO-g1QFAs, shown in the proof of Theorem 2, can be calculated in polynomial time, we get the following.

Theorem 5. *There is a polynomial-time algorithm that takes as input two MM-1QFAs M_1 and M_2 and determines whether M_1 and M_2 are equivalent, where n_1 and n_2 are the number of states in M_1 and M_2 , respectively. Furthermore, if M_1 and M_2 are not equivalent then the algorithm outputs the lexicographically minimum string which is accepted by M_1 and M_2 with different probabilities. This string will always be of length $O((n_1 + n_2)^2)$.*

6 Conclusion

Although Brodsky and Pippenger had solved the equivalence for MO-1QFAs, the algorithm was not efficient. On the other hand, the decidability of the equivalence for MM-1QFAs had been an open problem. In this paper, for both cases, we gave polynomial-time algorithms for the equivalence. As we have seen, MM-1QFAs (and MO-1QFAs) can be generalized using the representation of selective quantum operations. Thus, we might extend our results to more general model. It is left to consider whether such extension is natural and to characterize the power of 1QFAs of the general model. Furthermore, it is still open whether the equivalence for 2-way QFAs or 1.5-way QFAs, defined in [2], is efficiently decidable or not. The techniques for the equivalence for one-way QFAs seem to be no longer effective to solve them. In [14], the algorithm for the approximate equivalence for stochastic automata was proposed. However, the algorithm is not efficient. Another interesting open problem is an approximate equivalence problem for QFAs which is efficiently decidable.

Acknowledgements

I would like to thank Hirokazu Anai and the anonymous referees for their useful suggestions.

References

1. D. Aharonov, A. Kitaev, and N. Nisan. Quantum circuits with mixed states. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 20–30. ACM Press, 1998.
2. M. Amano and K. Iwama. Undecidability on quantum finite automata. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 368–375. ACM Press, 1999. 268, 277
3. A. Ambainis and R. Freivalds. 1-way quantum finite automata: Strengths, weaknesses and generalizations. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 332–341. IEEE Computer Society Press, 1998.

4. A. Ambainis, A. Kikusts, and M. Valdat. On the class of languages recognizable by 1-way quantum finite automata. In A. Ferreira and H. Reichel, editors, *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2001)*, volume 2010 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 2001. 268
5. E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.
6. A. Berthiaume. Quantum computation. In L. A. Hemaspaandra and A. L. Selman, editors, *Complexity Theory Retrospective II*, pages 23–50. Springer, 1997. 269
7. A. Brodsky and N. Pippenger. Characterizations of 1-way quantum finite automata. Los Alamos Preprint Archive, quant-ph/9903014, 1999. 268, 269
8. J. Gruska. Descriptive complexity issues in quantum computing. *Journal of Automata, Languages and Combinatorics*, 5(3):191–218, 2000. 269
9. A. Kondacs and J. Watrous. On the power of quantum finite state automata. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 66–75. IEEE Computer Society Press, 1997. 268
10. C. Moore and J. P. Crutchfield. Quantum automata and quantum grammars. *Theoretical Computer Science*, 237(1-2):275–306, 2000. 268, 269, 271, 272
11. M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000. 269
12. A. Paz. *Introduction to Probabilistic Automata*. Academic Press, 1971. 269
13. M. O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.
14. W. Tzeng. A polynomial-time algorithm for the equivalence of probabilistic automata. *SIAM Journal on Computing*, 21(2):216–227, 1992. 269, 273, 274, 275, 277
15. J. Watrous. On quantum and classical space-bounded processes with algebraic transition amplitudes. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 341–351. IEEE Computer Society Press, 1999. 269

An Index for the Data Size to Extract Decomposable Structures in LAD

Hiroataka Ono, Mutsunori Yagiura, and Toshihide Ibaraki

Department of Applied Mathematics and Physics, Graduate School of Informatics,
Kyoto University
Kyoto 606-8501, Japan
{htono,yagiura,ibarak}@amp.i.kyoto-u.ac.jp

Abstract. Logical analysis of data (LAD) is one of the methodologies for extracting knowledge as a Boolean function f from a given pair of data sets (T, F) on attributes set S of size n , in which T (resp., F) $\subseteq \{0, 1\}^n$ denotes a set of positive (resp., negative) examples for the phenomenon under consideration. In this paper, we consider the case in which extracted knowledge has a decomposable structure; i.e., f is described as a form $f(x) = g(x[S_0], h(x[S_1]))$ for some $S_0, S_1 \subseteq S$ and Boolean functions g and h , where $x[I]$ denotes the projection of vector x on I . In order to detect meaningful decomposable structures, it is expected that the sizes $|T|$ and $|F|$ must be sufficiently large. In this paper, we provide an index for such indispensable number of examples, based on probabilistic analysis. Using $p = |T|/(|T| + |F|)$ and $q = |F|/(|T| + |F|)$, we claim that there exist many deceptive decomposable structures of (T, F) if $|T| + |F| \leq \sqrt{2^{n-1}/pq}$. The computational results on synthetically generated data sets show that the above index gives a good lower bound on the indispensable data size.

Keywords: logical analysis of data, Boolean functions, decomposable functions, computational learning theory, random graphs, probabilistic analysis

1 Introduction

Extracting knowledge from given data has been studied in such fields as knowledge engineering, data mining, artificial intelligence and database theory (e.g., [4,6,8]). Logical analysis of data (LAD) is one of the methodologies for knowledge discovery. LAD is based on Boolean logic, that is, a given data set is represented as a pair of set T of true vectors (examples that cause the phenomenon to occur) and set F of false vectors (examples not causing the phenomenon to occur), where $T, F \subseteq \{0, 1\}^n$ and $T \cap F = \emptyset$. We denote by $S = \{1, 2, \dots, n\}$ the set of attributes of data. Each vector $x \in \{0, 1\}^n$ consists of n components corresponding to elements in S , i.e., $x = (x_1, x_2, \dots, x_n)$.

LAD tries to find a function $f: \{0, 1\}^n \rightarrow \{0, 1\}$ such that $f(v) = 1$ for all $v \in T$ and $f(w) = 0$ for all $w \in F$. Such a function f , called an *extension*, provides a logical explanation for the phenomenon represented by (T, F) . However,

in general, a consistent extension for (T, F) is not unique and may not necessarily provide structural information of the phenomenon. In other words, the consistency alone may not be sufficient to extract meaningful logical explanation for (T, F) . Therefore, in LAD, we usually consider extensions having some specific features, e.g., positive, Horn, k -DNF, decomposable and so on [10,12]. These features will reveal the logical structure of the phenomenon under consideration.

Even after restricting our attention to those with specific features, there may still exist many extensions if the sizes $|T|$ and $|F|$ are not sufficiently large. In such cases, most of the discovered extensions are deceptive; i.e., they do not represent true explanations of the phenomenon. In this sense, it is important to know what is the “sufficiently large” number of examples in (T, F) . In this paper, we focus on the decomposability as a specific feature of extensions, and propose an index to judge whether the size of the given (T, F) is sufficiently large to extract meaningful decomposable extensions.

Decomposable extensions are defined as follows: An extension f is decomposable if there exist some disjoint subsets S_0 and S_1 of S with $2 \leq |S_1| \leq n-1$ and Boolean functions g and h satisfying $f(x) = g(x[S_0], h(x[S_1]))$, where $x[I]$ denotes the *projection* of vector x on a set of attributes I . The set S_1 represents an intermediate group of attributes, and defines a new “meta-attribute”, which can give a simpler explanation of the phenomenon. This problem of structure identification is in fact one form of knowledge discovery. As an example, let $f(x)$ be a Boolean function that describes whether a certain species is a primate or not; e.g., $f(v) = 1$ for $v = (1100 \dots)$ denotes that the chimpanzee, which has characteristics of viviparous ($v_1 = 1$), vertebrate ($v_2 = 1$), does not fly ($v_3 = 0$), does not have claw ($v_4 = 0$), and so on, is a primate. In the case of the hawk, on the other hand, we shall have $f(0111 \dots) = 0$. In this example, we can group attributes “viviparity” and “vertebrate” as a property of the mammals, and the chimpanzee is a mammal. That is, $f(x)$ can be represented as $g(x[S_0], h(x[S_1]))$, where $S_1 = \{1, 2\}$, $S_0 = S \setminus S_1$, and h describes whether the species is a mammal or not. This “mammal” is a meta-attribute, and we can recognize primates by regarding $S_1 = \{1, 2\}$ as one attribute $h(x[S_1])$. In this sense, finding an attribute set S_1 , which satisfies the above decomposition property, can be understood as finding an essential relation among the original attributes [3,8].

The index proposed in this paper is based on a probabilistic analysis of the event that a randomly generated (T, F) has decomposable extensions. If this probability is high, it indicates the size of the given data set is not sufficient, since a random data set is unlikely to be decomposable if its size is sufficiently large. We claim that a given (T, F) does not give reliable information about decomposability if $|T| + |F| \leq \sqrt{2^{n-1}/pq}$ holds for $p = |T|/(|T| + |F|)$ and $q = |F|/(|T| + |F|)$. That is, the index $\sqrt{2^{n-1}/pq}$ can be used as an index for the number of data vectors needed to ensure that the discovered decomposable extension is not deceptive.

In real world applications, the sizes of given data sets T and F are sometimes quite small compared to $2^n = |\{0, 1\}^n|$. In such cases, the proposed index helps us to judge how reliable the decomposable structures observed in the given data

set are. On the other hand, if the size of the data set is very large, it can be often reduced for efficient computation [9,13] by sampling only a small subset of the data set. The proposed index is also usable to know an appropriate sampling ratio needed to obtain meaningful results.

We then conduct computational experiments of finding decomposable extensions for synthetically generated data sets and real-world data sets, which are not decomposable. The experimental results show that the proposed index in fact provides a good lower bound on the size of data set needed to assure meaningful decomposable extensions.

2 Preliminaries

A *Boolean function*, or a *function* in short, is a mapping $f : \{0, 1\}^n \rightarrow \{0, 1\}$, where $x \in \{0, 1\}^n$ is called a *Boolean vector* (a *vector* in short). $S = \{1, 2, \dots, n\}$ denotes the set of all attributes. If $f(x) = 1$ (resp., 0), then x is called a *true* (resp., *false*) vector of f . The set of all true vectors (resp., false vectors) is denoted by $T(f)$ (resp., $F(f)$).

A *partially defined Boolean function* (pdBf) is defined by a pair of sets (T, F) , where $T \subseteq \{0, 1\}^n$ (resp., $F \subseteq \{0, 1\}^n$) denotes a set of positive (resp., negative) examples. A function f is called an *extension* of a pdBf (T, F) if $T \subseteq T(f)$ and $F \subseteq F(f)$; i.e., if $f(a) = 1$ for all $a \in T$ and $f(b) = 0$ for all $b \in F$.

Evidently, the disjointness of the sets T and F is a necessary and sufficient condition for the existence of an extension, if it is considered in the class of all Boolean functions. It may not be trivial, however, to judge whether a given pdBf has an extension in a certain subclass $\mathcal{C}$ of Boolean functions, such as the class of positive functions, the class of k -DNF functions (DNF functions with at most k literals in each term), and so on [6]. The problem to find an extension in a subclass $\mathcal{C}$ of a given pdBf (T, F) is called the *consistency problem* in computational learning theory [2].

For a subset $S' \subseteq S$, let $\{0, 1\}^{S'}$ denote the vector space defined by an attribute set S' . Given a pair of subsets $S_0, S_1 \subseteq S$, a function f is called $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -*decomposable* if there exist Boolean functions h and g satisfying the following conditions

- (i) $f(x) = g(x[S_0], h(x[S_1]))$ for all $x \in \{0, 1\}^n$,
- (ii) $h : \{0, 1\}^{S_1} \rightarrow \{0, 1\}$,
- (iii) $g : \{0, 1\}^{S'} \rightarrow \{0, 1\}$, where $S' = S_0 \cup \{h\}$.

We also call a pdBf (T, F) $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -*decomposable* if (T, F) has an $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable extension. Decomposability was originally proposed as a more general concept, and other classes of decomposable functions were also considered in [3,10]. In this paper, we restrict our attention to the decomposability in which (S_0, S_1) is a partition (i.e., $S_1 = S \setminus S_0$) satisfying $|S_0| \geq 1$ and $|S_1| \geq 2$, which we call a *nontrivial partition*. This is one of the most fundamental forms of decomposition. We then call a pdBf (T, F) *decomposable*, if there exists a

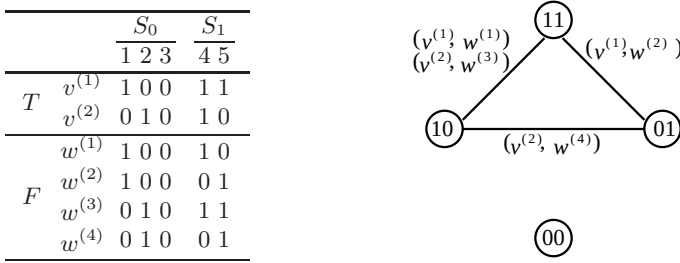


Fig. 1. A pdBf and its conflict graph which is not $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable

nontrivial partition (S_0, S_1) such that (T, F) is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable. Finally, we call the above partition (S_0, S_1) together with f, g and h a *decomposable structure* of (T, F) .

Given a pdBf (T, F) and a pair of subsets S_0 and S_1 , we define its *conflict graph* $G_{(T,F)}(S_0, S_1) = (V, E)$ by¹

$$V = \{x \mid x \in \{0, 1\}^{S_1}\},$$

$$E = \{(v[S_1], w[S_1]) \mid v \in T, w \in F \text{ and } v[S_0] = w[S_0]\}.$$

We call a cycle of length k in the graph a *k-cycle*, and a cycle of odd (resp., even) length an *odd cycle* (resp., *even cycle*). Then the following property holds.

Proposition 1. [3] A pdBf (T, F) is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable if and only if its conflict graph $G_{(T,F)}(S_0, S_1)$ contains no odd cycle; i.e., $G_{(T,F)}(S_0, S_1)$ is bipartite. $\square$

For example, let us consider the pdBf (T, F) given in the truth table in Fig. 1, and its $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability for $S_0 = \{1, 2, 3\}$ and $S_1 = \{4, 5\}$. In the figure, a label of each vertex represents vectors $x[S_1]$ for $x \in \{0, 1\}^5$ and a label on each edge denote the pair of true and false vectors that defines the edge. In this example, the pair $(v^{(1)}, w^{(1)})$ defines the edge $(v^{(1)}[S_1], w^{(1)}[S_1]) = (11, 10)$, and at the same time the pair $(v^{(2)}, w^{(3)})$ defines the same edge $(10, 11)$. Since the corresponding conflict graph in Fig. 1 is not bipartite, pdBf (T, F) is not $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable. On the other hand, pdBf (T, F') with the same T and $F' = F \setminus \{w^{(4)}\}$ is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable, because edge $(10, 01)$ does not exist and $G_{(T,F')}(S_0, S_1)$ is bipartite.

3 Decomposability of Randomly Assigned Functions

For real numbers p and q satisfying $p, q \in [0, 1]$ and $p + q = 1$, $\mathcal{F}_{(p,q)}$ denotes the probability space of the set of functions which take value 1 or 0 for each vector

¹ This definition is slightly different from the original one in [3, 12].

in $\{0, 1\}^n$ with probability p or q , respectively. We denote a function in $\mathcal{F}_{(p,q)}$ by $f_{(p,q)}$. Let U be a random subset of $\{0, 1\}^n$, which satisfies $|U| = l$ for a given l . We define a *randomly assigned pdBf* (T_p, F_q) by $T_p = \{v \in U \mid f_{(p,q)}(v) = 1\}$ and $F_q = \{w \in U \mid f_{(p,q)}(w) = 0\}$; i.e., T_p (resp., F_q) is the set of true (resp., false) vectors constructed from the truth table of $f_{(p,q)}$ by sampling the vectors over U . By definition, each vector in set $U = T_p \cup F_q$ is a member of T_p (resp., F_q) with probability p (resp., q).

In this section, we consider the probability that a randomly assigned pdBf (T_p, F_q) is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable. For the goal, we first consider the probability of an edge to appear in the conflict graph $G_{(T_p, F_q)}(S_0, S_1)$ in Subsection 3.1. In Subsection 3.2, we consider the probability that a (general) random graph is bipartite. In Subsection 3.3, we analyze the probability that the conflict graph of (T_p, F_q) is bipartite (i.e., (T_p, F_q) is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable) from the results in Subsections 3.1 and 3.2.

3.1 Probability of an Edge to Appear in the Conflict Graph

Given a pdBf (T_p, F_q) , let us consider a pair (a, b) with $a, b \in \{0, 1\}^{S_1}$. It becomes an edge of $G_{(T_p, F_q)}(S_0, S_1)$ if and only if there exists a $y \in \{0, 1\}^{S_0}$ such that one of $y \cdot a$ and $y \cdot b$ is in T_p and the other is in F_q , where $y \cdot a$ (resp., $y \cdot b$) denote the concatenation of y and a (resp., y and b). That is, there are $2^{|S_0|}$ pairs of vectors $e_y = (y \cdot a, y \cdot b)$ which can make edge (a, b) present in $G_{(T_p, F_q)}(S_0, S_1)$. Two vectors $v, w \in U = T_p \cup F_q$ are called *complementary* if one of them is in T_p and the other is in F_q . We call $(y \cdot a, y \cdot b)$ a *linked* pair if $y \cdot a$ and $y \cdot b$ are in U and complementary. Assuming $y \cdot a, y \cdot b \in U$, the probability that two vectors $(y \cdot a, y \cdot b)$ is complementary, is $2pq$.

For a pair $e = (a, b)$ and $y \in \{0, 1\}^{S_0}$, let X_{ey} be the indicator random variable defined by

$$X_{ey} = \begin{cases} 1 & (y \cdot a, y \cdot b) \text{ is linked,} \\ 0 & \text{otherwise,} \end{cases} \quad (1)$$

and let

$$X_e = \sum_{y \in \{0, 1\}^{S_0}} X_{ey}. \quad (2)$$

Let $R^* = \Pr(X_e \geq 1)$. That is, R^* gives the probability that $e = (a, b)$ is an edge in the conflict graph $G_{(T_p, F_q)}(S_0, S_1)$. For convenience, let $m = 2^{|S_0|}$ and $M = 2^n$. The probability that l vectors sampled from M vectors (i.e., all vectors in $\{0, 1\}^n$) contain two specific vectors $y \cdot a$ and $y \cdot b$ is $(l(l-1))/(M(M-1))$. Therefore the expectation $\mu = \text{Ex}(X_e)$ of X_e is given by

$$\mu = \text{Ex} \left(\sum_{y \in \{0, 1\}^{S_0}} X_{ey} \right) = \sum_{y \in \{0, 1\}^{S_0}} \text{Ex}(X_{ey}) = m \text{Ex}(X_{ey}) = m \cdot 2pq \cdot \frac{l(l-1)}{M(M-1)} \quad (3)$$

by linearity of expectation. For R^* and μ , we have the following theorem. As the proof is not trivial, we omit it for the sake of space.

Theorem 1. For a randomly assigned pdBf (T_p, F_q) and a partition (S_0, S_1) , $\mu - \mu^2/2 \leq R^* \leq \mu$ holds. $\square$

Both of μ and $\mu - \mu^2/2$ are monotonically increasing in $[0, 1]$, and their values are very close for a small μ (say $\mu \leq 0.1$). These imply that μ is a good approximation of R^* if μ is small.

Claim 1 If μ is small, $\mu \approx R^*$ holds.

In Subsection 3.3, we compute the proposed index by assuming that the conflict graph is a random graph, in which this R^* is the probability that an edge appears in the random graph. As will be shown in Subsection 3.2, the probability that a random graph is bipartite is high if $R^* < 1/N$ holds, where N is the number of vertices of the random graph. In considering the $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability, the number of vertices of the conflict graph is $2^{|S_1|}$ ($|S_1| \geq 2$) and is 4, 8, 16, $\dots$ and so on (i.e., $1/N$ is at most 0.25). In other words, we are interested in R^* not larger than 0.25. In this sense, the condition in the above claim that μ is small is meaningful.

3.2 Appearance of An Odd Cycle in a Random Graph

For a positive integer N and $0 \leq r \leq 1$, let $\mathcal{G}(N, r)$ denote the probability space [1] over the set of random graphs G on the vertex set $V(G) = \{1, 2, \dots, N\}$ and the edge set $E(G)$ determined by independently including each possible edge with probability r . Let Y_{odd} denote the random variable that represents the number of odd cycles in $G \in \mathcal{G}(N, r)$. In this subsection, we investigate the conditions for a random graph $G \in \mathcal{G}(N, r)$ to be bipartite. From the Markov inequality [1], we have

$$\Pr(Y_{\text{odd}} \geq 1) \leq \text{Ex}(Y_{\text{odd}}). \quad (4)$$

Let $N^{\underline{k}}$ denote the k -th falling factorial power of N defined by $N^{\underline{k}} = N(N-1)(N-2) \cdots (N-k+1)$, which represents the number of sequences of k distinct elements in set $\{1, 2, \dots, N\}$. Since a sequence of k vertices represents a k -cycle whose start vertex and direction are specified, the number of potential k -cycle in G is $N^{\underline{k}}/2k$. The probability that all k edges in a k -cycle exist in G is r^k . Therefore the expectation of Y_{odd} is

$$\text{Ex}(Y_{\text{odd}}) = \sum_{\substack{3 \leq k \leq N, \\ k: \text{odd}}} \frac{N^{\underline{k}}}{2k} r^k \leq \sum_{\substack{k \geq 3, \\ k: \text{odd}}} \frac{(Nr)^k}{2k} = \frac{1}{2} \left(\sum_{\substack{k \geq 1, \\ k: \text{odd}}} \frac{(Nr)^k}{k} - Nr \right). \quad (5)$$

Let $z = Nr$. Then if $0 \leq Nr = z < 1$ holds, by the Taylor series of $\ln(1-z)$, the last formula in (5) is equal to

$$U_{\text{odd}}(z) = \frac{1}{2} \left(\frac{1}{2} \ln \left(\frac{1+z}{1-z} \right) - z \right),$$

which is a upper bound on $\text{Ex}(Y_{\text{odd}})$, and hence on $\text{Pr}(Y_{\text{odd}})$.

Function $U_{\text{odd}}(z)$ is monotonically increasing, satisfies $U_{\text{odd}}(0) = 0$ and $U_{\text{odd}}(1) = +\infty$ and is quite small for $z < 0.9$. This means that it is a good upper bound on $\text{Pr}(Y_{\text{odd}} \geq 1)$, especially for small z . Moreover, $U_{\text{odd}}(z_{\text{odd}}^*) = 1$ implies $z_{\text{odd}}^* \in (0.9950, 0.9951)$. Hence, we have

$$Nr \leq 0.9950 \implies \text{Ex}(Y_{\text{odd}}) < 1. \quad (6)$$

This result indicates that the probability that a random graph G is bipartite is large, if Nr is less than 1; e.g., if $Nr \leq 0.9$, the probability is not less than $1 - 0.28610974 = 0.71389026$.

Next, we investigate the value of Nr that satisfies $\text{Ex}(Y_{\text{odd}}) = 1$. Note first that $N^k/N^k = 1 + o(1)$ holds if $k \leq N^{1/2-\varepsilon}$ for any constant $\varepsilon > 0$, where $o(1)$ converges to 0 if N becomes large [14]. In other words, for any constant $c \in [0, 1)$ and $\varepsilon \in (0, 1/2)$, there exists N_0 such that $N^k/N^k \geq c$ holds for $k \leq N^{1/2-\varepsilon}$ and $N \geq N_0$. Hence, for $Nr = 1$, we have

$$\text{Ex}(Y_{\text{odd}}) = \sum_{\substack{3 \leq k \leq N, \\ k:\text{odd}}} \frac{N^k r^k}{2k} \frac{1}{(Nr)^k} \geq \sum_{\substack{3 \leq k \leq \lfloor N^{1/2-\varepsilon} \rfloor, \\ k:\text{odd}}} \frac{c}{2k} = \frac{c}{4} \ln(\lfloor N^{1/2-\varepsilon} \rfloor) + O(1).$$

The last equality is from $\sum_{1 \leq k \leq l} 1/k = \ln l + O(1)$ and $\sum_{1 \leq k \leq l, k:\text{odd}} 1/k - \sum_{1 \leq k \leq l, k:\text{even}} = \ln 2 + O(l^{-1})$. Therefore, if $Nr = 1$, $\text{Ex}(Y_{\text{odd}}) \rightarrow +\infty$ holds as $N \rightarrow +\infty$. In summary, for a sufficiently large N , we have

$$\text{Ex}(Y_{\text{odd}}) = 1 \implies 0.9950 < Nr < 1.$$

Furthermore, we believe that, for large N , $G \in \mathcal{G}(N, 1/N)$ is not bipartite with high probability, since $\text{Ex}(Y_{\text{odd}})$ becomes quite large for large N .

These are summarized in the following claim.

Claim 2 A random graph $G \in \mathcal{G}(N, r)$ tends to be bipartite for large N if $Nr < 1$. On the other hand, if Nr is not less than 1, $G \in \mathcal{G}(N, r)$ tends to be non-bipartite. In this sense, $Nr = 1$ is a threshold point of bipartiteness of $G \in \mathcal{G}(N, r)$.²

3.3 An Index for Decomposability

For a given pdBf (T, F) , we propose an index for the $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability. Let

$$p = |T|/(|T| + |F|), q = |F|/(|T| + |F|), \text{ and } l = |T| + |F|,$$

² Though these results seem similar to the theorem of graph evolution [7], there are some differences. One of the major differences is that our results show the precise value of the threshold point, while the theorem of graph evolution analyzes the asymptotic behavior for the cases of $Nr \rightarrow 0$ and $Nr \rightarrow +\infty$.

and compare (T, F) with (T_p, F_q) which is expected to have the same ratio between the numbers of positive and negative examples.

We regard the conflict graph $G_{(T_p, F_q)}(S_0, S_1)$ as a random graph $\mathcal{G}(N, r)$, where $N = M/m = 2^{|S_1|}$ and $r = \Pr(X_e \geq 1)$ (i.e., the probability of an edge e_i to appear in the conflict graph)³. From the view point of decomposability, we would like to know the value of l for which $Nr = 1$ holds, which is equivalent to $2^{|S_1|} \Pr(X_e \geq 1) = 1$, since $Nr = 1$ is a threshold point of the bipartiteness of random graph as mentioned in Claim 2. As $|S_1| \geq 2$, we have $1/2^{|S_1|} \leq 1/4$. Then, by Claim 1, μ is a good approximation of $\Pr(X_e \geq 1)$. Moreover we can approximate μ by $2pqml^2/M^2$, since M and l are large in general. Hence, we have

$$2^{|S_1|} \Pr(X_e \geq 1) \approx \frac{M}{m} \mu \approx \frac{M}{m} \cdot 2pqm \frac{l^2}{M^2} = pq \frac{l^2}{2^{n-1}}.$$

Therefore

$$l = \sqrt{2^{n-1}/pq}$$

would be a good approximation for the threshold point of the $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability of (T_p, F_q) . These lead to the following claim.

Claim 3 Let (S_0, S_1) be any nontrivial partition of S . If $|T| + |F| \leq \sqrt{2^{n-1}/pq}$ where $p = |T|/(|T| + |F|)$ and $q = |F|/(|T| + |F|)$, $\text{pdBf}(T, F)$ has many deceptive $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable extensions. On the other hand, if $|T| + |F| > \sqrt{2^{n-1}/pq}$, (T, F) tends to have no deceptive $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable extensions. Hence we claim that $\sqrt{2^{n-1}/pq}$ is an index of $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability. $\square$

This index $\sqrt{2^{n-1}/pq}$ is simple and easy to compute. Furthermore, it does not depend on (S_0, S_1) . In the next section, we verify the performance of this index through numerical experiments.

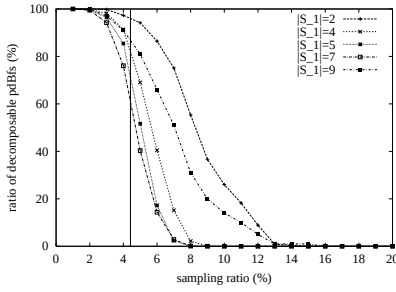
4 Numerical Experiments

We conduct the numerical experiments to check the $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability of (T, F) for any fixed partition (S_0, S_1) , against the sizes of (T, F) using synthetically generated data and a real-world data. We use a simple polynomial time algorithm developed in [12] to judge whether a given data set (T, F) is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable or not for a given (S_0, S_1) . Applying the algorithm to (T, F) for all partitions (S_0, S_1) , we can check the decomposability of (T, F) . In order to avoid the dependency on the selection of (T, F) , we generate 10 instances of (T, F) from each given Boolean function f (which is a randomly constructed, or represents a real-world phenomenon) and take the average of the results.

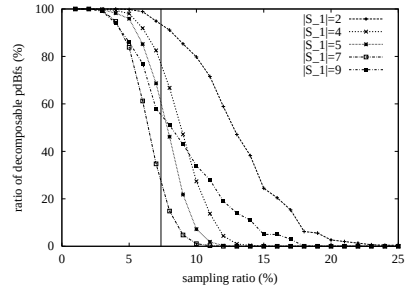
³ This assumption seems rather rough as the possibility of edges to appear in the conflict graph are not independent. However, we believe that they are almost independent based on some probabilistic analyses, though the analyses are omitted due to limitation of space.

4.1 Randomly Generated Data

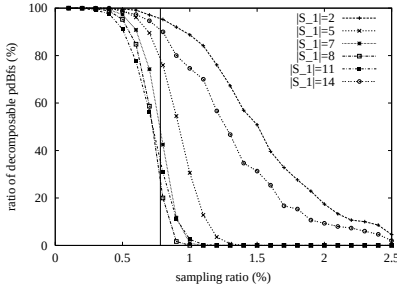
We first generate a randomly assigned function $f_{(p,q)}$ (see Section 3 for the definition of randomly assigned functions). Then, pdBfs (T, F) are generated by randomly choosing $l = |T| + |F|$ vectors from $\{0, 1\}^n$, where their truth assignment are determined by $f_{(p,q)}$. We call $l/2^n \times 100(\%)$ the sampling ratio, and prepare 10 pdBfs (T, F) for each sampling ratio.



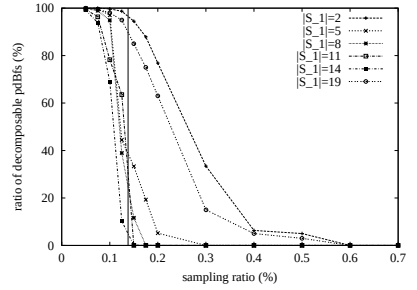
(a) $p = q = 0.5$ and $n = 10$



(b) $p = 0.9, q = 0.1$ and $n = 10$



(c) $p = q = 0.5$ and $n = 15$



(d) $p = q = 0.5$ and $n = 20$

Fig. 2. The ratio of $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable (T, F) for randomly generated data

Fig. 2 shows the ratio of decomposable pdBfs (T, F) for several partitions (S_0, S_1) with different size of $|S_1|$. Figs. 2 (a) and (b) are the results for $n = 10$, and (c) and (d) are the results for $n = 15$ and $n = 20$, respectively. We set $p = q = 0.5$ in Figs. 2 (a), (c) and (d), and set $p = 0.9$ and $q = 0.1$ in Fig. 2 (b). The horizontal axis represents sampling ratio in %, and the vertical axis gives the ratio of $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable pdBfs. The results are classified according to the sizes $|S_1|$, since the sizes of conflict graphs depend on them.

In Fig. 2 (a), the results for $|S_1| = 2, 4, 5, 7, 9$ are shown. (We omitted the results for $|S_1| = 3, 6, 8$ due to legibility.) The vertical line at the sampling ratio 4.419% corresponds to our proposed index $\sqrt{2^{n-1}/pq}/2^n \times 100 = \sqrt{2^9/(0.5 \times 0.5)}/2^{10} \times 100 = 4.419^4$. From this figure, we can observe that if the sampling ratio $(|T| + |F|)/2^n \times 100$ is smaller than the proposed index value, (T, F) is $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable with high probability, while the original Boolean function $f_{(p,q)}^*$ is not $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable for any (S_0, S_1) . Moreover, if the sampling ratio is larger than 4.419%, the ratios of $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable pdBfs rapidly decrease for most of $|S_1|$. That is, the index is a good estimate for the threshold point of the $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability. For partitions (S_0, S_1) with $|S_1| = 2$ and $|S_1| = 9$, the ratio decreases more slowly than other cases.

Fig. 2 (b) is the case where $|T|$ and $|F|$ are asymmetric, and Figs. 2 (c) and (d) are the case where the dimension n is larger. Each vertical line represents the proposed index. In all of them, similar threshold behavior as in Fig. 2 (a) can be observed. This means that the proposed index is a good estimate of the threshold point of decomposability for a wide range of data sets.

4.2 Real-World Data

Next, we apply similar experiment to a real-world data set, Breast Cancer in Wisconsin (BCW for short)⁵. In BCW, each vector was taken from a patient of a breast cancer and each attribute refers to a clinical case, e.g., clump thickness and so on. Patients are classified into two classes, malignant (positive) and benign (negative). Since BCW are not binary, we first binarized it ([5]) and then obtain $(T_{\text{BCW}}, F_{\text{BCW}})$ by the method described in [11]. The obtained $(T_{\text{BCW}}, F_{\text{BCW}})$ is not $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable for any (S_0, S_1) . T_{BCW} and F_{BCW} are sets of vectors in $\{0, 1\}^{11}$ where $|T_{\text{BCW}}| = 173$ and $|F_{\text{BCW}}| = 64$. $|T_{\text{BCW}}| + |F_{\text{BCW}}| = 237$ is equivalent to the sampling ratio 11.6% of 2^{11} . Ten data sets (T, F) for each sampling ratio are obtained by randomly choosing vectors from $T_{\text{BCW}} \cup F_{\text{BCW}}$. The results are shown in Fig. 7 (a). The vertical line at sampling ratio 3.519% corresponds to the proposed index value, where $p = 173/(173 + 64)$ and $q = 64/(173 + 64)$.

For comparison purposes, we show the results for a randomly assigned function $f_{(p,q)}$ with the same parameters n, p and q as BCW. Fig. 3 (b) shows the results of pdBfs (T, F) generated from the $f_{(p,q)}$ with $n = 11$ and $p = 173/(173+64)$ and $q = 64/(173+64)$. The results of Figs. 3 (a) and (b) are close, but the results for $(T_{\text{BCW}}, F_{\text{BCW}})$ are shifted slightly to left compared to the results for $f_{(p,q)}$. Ignoring such slight differences, these results appear to imply that our index is applicable to real-world data.

In summary, all the experimental results in this section show that the proposed index is a good estimate of the threshold point of $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decom-

⁴ Precisely speaking, our index is not exactly 4.419%, but located around 4.419%, because $|T|$ and $|F|$ of pdBfs (T, F) vary slightly depending on the data sets.

⁵ <http://www.ics.uci.edu/%7Emlearn/MLRepository.html>

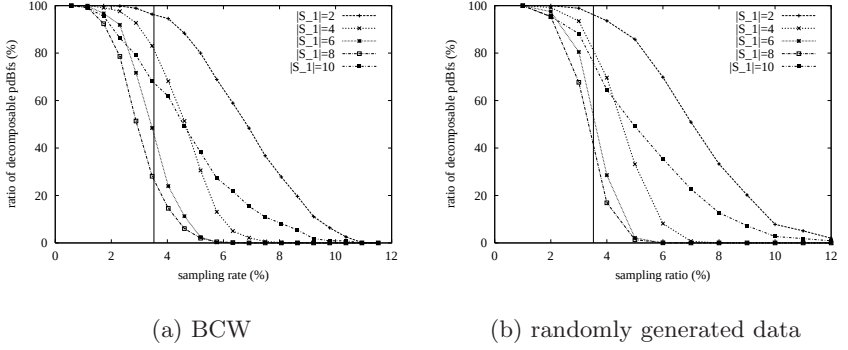


Fig. 3. The ratio of $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable pdBf for BCW and randomly generated data

possibility for any partition (S_0, S_1) satisfying $|S_0| > 1$ and $|S_1| > 2$. However, for (S_0, S_1) satisfying $|S_0| = 1$ or $|S_1| = 2$, threshold behavior is not clear, i.e., the slope is not sharp compared with other cases. Though we omit the details due to limitation of space, one of the reasons for this phenomenon is explained by the fact that these cases may violate the assumption (i) the edges in the conflict graph appear independently (see Subsection 3.3) or (ii) μ is small (see Claim 1 in Subsection 3.1). Even in these cases, our index is still useful, because we can conclude that the discovered $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable extensions are all deceptive if $|T| + |F| \leq \sqrt{2^{n-1}/pq}$. The only problem for small $|S_0|$ or $|S_1|$ is that threshold behavior is no longer visible.

5 Conclusion

In this paper, we proposed an index that tells how many examples should be included in the data set (T, F) so that real $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposability can be detected for the phenomenon under consideration. We claim that $\sqrt{2^{n-1}/pq}$ with $p = |T|/(|T| + |F|)$ and $q = |F|/(|T| + |F|)$ is a good estimate of the threshold point of decomposability, based on probabilistic analysis. That is, the number of deceptive $\mathcal{F}(S_0, \mathcal{F}(S_1))$ -decomposable extensions of (T, F) increases sharply if $|T| + |F| \leq \sqrt{2^{n-1}/pq}$. The computational experiments in Section 4 verified our claim, and indicated that the index is useful in LAD applications to real world data sets.

Acknowledgment

This work was partially supported by the Scientific Grant-in-Aid by the Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

1. N. Alon and J. H. Spencer, *The Probabilistic Method, Second Edition*, (John Wiley & Sons, 2000). 284
2. M. Anthony and N. Biggs, *Computational Learning Theory*, (Cambridge University Press, 1992). 281
3. E. Boros, V. Gurvich, P. L. Hammer, T. Ibaraki and A. Kogan, Decompositions of partially defined Boolean functions, *Discrete Applied Mathematics*, 62 (1995) 51–75. 280, 281, 282
4. E. Boros, P. L. Hammer, T. Ibaraki, A. Kogan, E. Mayoraz and I. Muchnik, An implementation of logical analysis of data, *IEEE Trans. on Knowledge and Data Engineering* 12 (2000) 292–306. 279
5. E. Boros, P. L. Hammer, T. Ibaraki and A. Kogan, Logical analysis of numerical data, *Mathematical Programming*, 79 (1997) 163–190. 288
6. Y. Crama, P. L. Hammer and T. Ibaraki, Cause-effect relationships and partially defined Boolean functions, *Annals of Operations Research*, 16 (1988) 299–325. 279, 281
7. P. Erdős and A. Rényi, On the evolution of random graphs, *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5 (1960) 17–61. 285
8. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, *Advances in Knowledge Discovery and Data Mining*, (AAAI Press, 1996). 279, 280
9. J. Kivinen and H. Mannila, The power of sampling in knowledge discovery, *Proceedings of the 1994 ACM SIGACT-SIGMOD-SIGACT Symposium on Principles of Database Theory (PODS'94)*, (1994) 77–85. 281
10. K. Makino, K. Yano and T. Ibaraki, Positive and Horn decomposability of partially defined Boolean functions, *Discrete Applied Mathematics*, 74 (1997) 251–274. 280, 281
11. S. Mii, Feature Determination Algorithms in the Analysis of Data, Master Thesis, Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, March 2001. 288
12. H. Ono, K. Makino and T. Ibaraki, Logical Analysis of Data with Decomposable Structures, *COCOON2000*, Lecture Notes in Computer Science 1858, (2000) 396–406. 280, 282, 286
13. H. Toivonen, Sampling Large Databases for Association Rules, *Proceedings of 22th International Conference on Very Large Data Bases (VLDB'96)*, (1996) 134–145. 281
14. M. Yagiura, T. Ibaraki and F. Glover, An Ejection Chain Approach for the Generalized Assignment Problem, Technical Report #99013, Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University, 1999. 285

Parameterized Complexity: The Main Ideas and Some Research Frontiers

Michael R. Fellows

School of Electrical Engineering and Computer Science, University of Newcastle
University Drive, Callaghan 2308, Australia
`mfellows@cs.newcastle.edu.au`

Abstract. The purposes of this paper are two: (1) To give an exposition of the main ideas of parameterized complexity, and (2) To discuss some of the current research frontiers and directions.

1 Introduction

Research in the parameterized framework of complexity analysis, and in the corresponding toolkit of algorithm design methods has been expanding rapidly in recent years. This has led to a flurry of recent surveys, all of which are good sources of introductory material [43,38,18,19,3,27,28]. One could also turn to the monograph [17].

With so many introductory surveys available, is not entirely clear what another one can now offer with respect to the basics. Yet I do think that there are a few new things to say, even about the fundamental notions. The first part of this survey attempts to summarize the main ideas of parameterized complexity and put the whole program in perspective.

The second part of the survey is more ephemeral. A few new research directions that I think are particularly important are discussed, together with some new results in these areas. The directions that are highlighted are: (1) connections between parameterized complexity and the complexity of approximation, and (2) the recent emergence of an exciting program of *FPT* optimality that hinges on a particularly compelling open problem.

2 Parameterized Complexity in a Nutshell

The main ideas of parameterized complexity are organized here into three discussions:

- The basic empirical motivation.
- The perspective provided by forms of the Halting Problem.
- The natural relationship of parameterized complexity to heuristics and practical computing strategies.

2.1 Empirical Motivation: Two Forms of Fixed-Parameter Complexity

Most natural computational problems are defined on input consisting of various information. A simple example is provided by the many graph problems that are defined as having input consisting of a graph $G = (V, E)$ and a positive integer k , such as (see [25] for definitions), GRAPH GENUS, BANDWIDTH, MIN CUT LINEAR ARRANGEMENT, INDEPENDENT SET, VERTEX COVER and DOMINATING SET. The last two problems are defined

VERTEX COVER

Input: A graph $G = (V, E)$ and a positive integer k .

Question: Does G have a vertex cover of size at most k ? (A *vertex cover* is a set of vertices $V' \subseteq V$ such that for every edge $uv \in E$, $u \in V'$ or $v \in V'$.)

DOMINATING SET

Input: A graph $G = (V, E)$ and a positive integer k .

Question: Does G have a dominating set of size at most k ? (A *dominating set* is a set of vertices $V' \subseteq V$ such that $\forall u \in V: u \in N[v]$ for some $v \in V'$.)

Although both problems are NP-complete, the input *parameter* k contributes to the complexity of these two problems in two qualitatively different ways.

- 1. After many rounds of improvement involving a variety of clever ideas, the best known algorithm for VERTEX COVER runs in time $O(1.271^k + kn)$ [13]. This algorithm has been implemented and is quite practical for n of unlimited size and k up to around 400 [30,41,21].
- 2. The best known algorithm for DOMINATING SET is *still* just the brute force algorithm of trying all k -subsets. For a graph on n vertices this approach has a running time of $O(n^{k+1})$.

The table below shows the contrast between these two kinds of complexity.

Table 1. The Ratio $\frac{n^{k+1}}{2^{kn}}$ for Various Values of n and k

	$n = 50$	$n = 100$	$n = 150$
$k = 2$	625	2,500	5,625
$k = 3$	15,625	125,000	421,875
$k = 5$	390,625	6,250,000	31,640,625
$k = 10$	1.9×10^{12}	9.8×10^{14}	3.7×10^{16}
$k = 20$	1.8×10^{26}	9.5×10^{31}	2.1×10^{35}

In order to formalize the difference between VERTEX COVER and DOMINATING SET we make the following basic definitions.

Definition 1. A parameterized language L is a subset $L \subseteq \Sigma^* \times \Sigma^*$. If L is a parameterized language and $(x, y) \in L$ then we will refer to x as the main part, and refer to y as the parameter.

A parameter may be non-numerical, and it can also be an aggregate of various kinds of information.

Definition 2. A parameterized language L is multiplicatively fixed-parameter tractable if it can be determined in time $f(k)q(n)$ whether $(x, k) \in L$, where $|x| = n$, $q(n)$ is a polynomial in n , and f is a function (unrestricted). The family of fixed-parameter tractable parameterized languages is denoted FPT .

Definition 3. A parameterized language L is additively fixed-parameter tractable if it can be determined in time $f(k) + q(n, k)$ whether $(x, k) \in L$, where $|x| = n$, $q(n, k)$ is a polynomial in n and k , and f is a function (unrestricted). The family of fixed-parameter tractable parameterized languages is denoted FPT .

As an exercise, the reader might wish to show that a parameterized language is additively fixed-parameter tractable if and only if it is multiplicatively fixed-parameter tractable. This emphasizes how cleanly fixed-parameter tractability isolates the computational difficulty in the complexity contribution of the parameter.

There are many ways that parameters arise naturally, for example:

- *The size of a database query.* Normally the size of the database is huge, but frequently queries are small. If n is the size of a relational database, and k is the size of the query, then answering the query can be solved trivially in time $O(n^k)$. It is known that this problem is unlikely to be FPT [20,40].
- *The nesting depth of a logical expression.* ML compilers work reasonable well. One of the problems the compiler must solve is the checking of the compatibility of type declarations. This problem is complete for deterministic exponential time [31], so the situation appears dire from the standpoint of classical complexity theory. The implementations work well in practice because the ML TYPE CHECKING problem is FPT with a running time of $O(2^k n)$, where n is the size of the program and k is the maximum nesting depth of the type declarations [34]. Since normally $k \leq 10$, the algorithm is clearly practical.
- *The number of sequences in a bio-informatics multiple molecular sequence alignment.* Frequently this parameter is in a range of $k \leq 50$. The problem can be solved in time $O(n^k)$ by dynamic programming. It is currently an open problem whether this problem is FPT for alphabets of fixed size [8].
- *The number of processors in a practical parallel processing system.* This is frequently in the range of $k \leq 64$. Is there a practical and interesting theory of parallel FPT ? Two recent papers that have begun to explore this area (from quite different angles) are [10] and [21].
- *The number of variables in a logical formula, or the number of steps in a deductive procedure.* Some initial studies of applications of parameterized complexity to logic programming and artificial intelligence have recently appeared [42,29], but much remains unexplored. Is it FPT to determine if k steps of resolution are enough to prove a formula unsatisfiable?

- *The number of steps for a motion planning problem.* Where the description of the terrain has size n (which therefore bounds the number of movement options at each step), we can solve this problem in time $O(n^{k+1})$ trivially. Are there significant classes of motion planning problems that are fixed-parameter tractable? Exploration of this topic has hardly begun [16].
- *The number of moves in a game.* The usual computational problem here is to determine if a player has a winning strategy. While most of these kinds of problems are *PSPACE*-complete classically, it is known that some are *FPT* and others are likely not to be *FPT*, when parameterized by the number of moves of a winning strategy [1]. The size n of the input game description usually governs the number of possible moves at any step, so there is a trivial $O(n^k)$ algorithm that just examines the k -step game trees exhaustively. This is potentially a very fruitful area, since games are used to model many different kinds of situations.
- *The size of a substructure.* The complexity class $\#P$ is concerned with whether the number of solutions to a problem (e.g., the number of Hamilton circuits in a graph, or the number of perfect matchings) can be counted in polynomial time. It would be interesting to consider whether *small* substructures can be counted (or generated) in *FPT* time, where the parameter is the size of the substructure (e.g., circuits of length k , or k -matchings). This subject has only just begun to be explored [5,22].
- *A “dual” parameter.* A graph has an independent set of size k if and only if it has a vertex cover of size $n - k$. Many problems have such a natural dual form and it is “almost” a general rule, first noted by Raman, that parametric duals of *NP*-hard problems have complementary parameterized complexity (one is *FPT*, and the other is *W[1]*-hard) [33,6]. For example, $n - k$ DOMINATING SET is *FPT*, as is $n - k$ GRAPH COLORING.
- *The distance from a guaranteed solution.* Mahajan and Raman pointed out that for many problems, solutions with some “intermediate” value (in terms of n) may be guaranteed and that it is then interesting to parameterize above or below the guaranteed value [36]. For a simple (and open) example, by the Four Color Theorem a planar graph must have an independent set of size at least $n/4$. Is it *FPT* to determine if a planar graph has an independent set of size at least $n/4 + k$?
- *The amount of “dirt” in the input or output for a problem.* For example, we might have an application of graph coloring where the input is expected to be 2-colorable, except that due to some imperfections, the input is actually only “nearly” 2-colorable. It would then be of interest to determine whether a graph can be properly colored in such a way that at most k vertices receive a third color. Some results indicate that the problem might be *FPT* [14], but this remains an open problem.
- *The “robustness” of a solution to a problem, or the distance to a solution.* For example, given a solution of the MINIMUM SPANNING TREE problem in an edge-weighted graph, we can ask if the cost of the solution is robust under all

increases in the edge costs, where the parameter is the total amount of cost increases. A number of problems of this sort have recently been considered by Leizhen Cai [9].

- *The distance to an improved solution.* Local search is a mainstay of heuristic algorithm design. The basic idea is that one maintains a *current solution*, and iterates the process of moving to a neighboring “better” solution. A neighboring solution is usually defined as one that is a single step away according to some small edit operation between solutions. The following problem is completely general for these situations, and could potentially provide a valuable subroutine for “speeding up” local search:

k-SPEED UP FOR LOCAL SEARCH

Input: A solution S , k .

Parameter: k

Output: The best solution S' that is within k edit operations of S .

Is it *FPT* to explore the k -change neighborhood for TSP?

- *The goodness of an approximation.* Perhaps the single most important strategy for “coping with *NP*-completeness” [25] is the program of polynomial-time approximation. The goodness of the approximation is an immediately relevant parameter. More about this in §3.

It is obvious that the practical world is full of concrete problems governed by parameters of all kinds that are bounded in small or moderate ranges. If we can design algorithms with running times like $2^k n$ for these problems, then we may have something really useful.

The following definition provides us with a place to put all those problems that are “solvable in polynomial time for fixed k ” without making our central distinction about whether this “fixed k ” is ending up in the exponent or not.

Definition 4. A parameterized language L belongs to the class *XP* (slicewise *P*) if it can be determined in time $f(k)n^{g(k)}$ whether $(x, k) \in L$, where $|x| = n$, α is a constant independent of both n and k , with f and g being unrestricted functions.

Is it possible that $FPT = XP$? This is one of the few structural questions concerning parameterized complexity that currently has an answer [17].

Theorem 1. *FPT is a proper subset of XP.*

2.2 The Halting Problem: A Central Reference Point

The main investigations of computability and efficient computability are tied to three basic forms of the Halting Problem.

1. THE HALTING PROBLEM

Input: A Turing machine M .

Question: If M is started on an empty input tape, will it ever halt?

2. THE POLYNOMIAL-TIME HALTING PROBLEM FOR NONDETERMINISTIC TURING MACHINES

Input: A nondeterministic Turing machine M .

Question: Is it possible for M to reach a halting state in n steps, where n is the length of the description of M ?

3. THE k -STEP HALTING PROBLEM FOR NONDETERMINISTIC TURING MACHINES

Input: A nondeterministic Turing machine M and a positive integer k . (The number of transitions that might be made at any step of the computation is unbounded, and the alphabet size is also unrestricted.)

Parameter: k

Question: Is it possible for M to reach a halting state in at most k steps?

The first form of the HALTING PROBLEM is useful for studying the question:

“Is there any algorithm for my problem?”

The second form of the HALTING PROBLEM has proved useful for nearly 30 years in addressing the question:

“Is there an algorithm for my problem ... like the ones for SORTING and MATRIX MULTIPLICATION?”

The second form of the HALTING PROBLEM is trivially NP -complete, and in fact essentially defines the complexity class NP . For a concrete example of why it is trivially NP -complete, consider the 3-COLORING problem for graphs, and notice how easily it reduces to the P -TIME NDTM HALTING PROBLEM. Given a graph G for which 3-colorability is to be determined, I just create the following nondeterministic algorithm:

Phase 1. (There are n lines of code here if G has n vertices.)

(1.1) Color vertex 1 one of the three colors nondeterministically.

(1.2) Color vertex 2 one of the three colors nondeterministically.

...

(1. n) Color vertex n one of the three colors nondeterministically.

Phase 2. Check to see if the coloring is proper and if so halt. Otherwise go into an infinite loop.

It is easy to see that the above nondeterministic algorithm has the possibility of halting in m steps (for a suitably padded Turing machine description of size m) if and only if the graph G admits a 3-coloring. Reducing any other problem $\Pi \in NP$ to the P -TIME NDTM HALTING PROBLEM is no more difficult than taking an argument that the problem Π belongs to NP and modifying it slightly to be a reduction to this form of the HALTING PROBLEM. It is in this sense that the P -TIME NDTM HALTING PROBLEM is essentially the *defining* problem for NP .

The conjecture that $P \neq NP$ is intuitively very well-founded. The second form of the HALTING PROBLEM would seem to require exponential time because there is seemingly little we can do to analyze unstructured nondeterminism other

than to exhaustively explore the possible computation paths. Apart from 20 years of accumulated habit, this concrete intuition is the fundamental reference point for classical complexity theory.

When the question is:

“Is there an algorithm for my problem ... like the one for VERTEX COVER?”

the third form of the HALTING PROBLEM anchors the discussion. This question will increasingly and inevitably be asked for any NP -hard problem for which small parameter ranges are important in applications. It is trivially solvable in time $O(n^k)$ by exploring the n -branching, depth- k tree of possible computation paths exhaustively, and our intuition here is essentially the same as for the second form of the Halting Problem — that this cannot be improved.

The third form of the Halting Problem defines the parameterized complexity class $W[1]$. Thus $W[1]$ is very strongly analogous to NP , and the conjecture that $FPT \neq W[1]$ is very much as reasonable as the conjecture that $P \neq NP$. The appropriate notion of reduction is as follows.

Definition 5. A parametric transformation from a parameterized language L to a parameterized language L' is an algorithm that computes from input consisting of a pair (x, k) , a pair (x', k') such that:

1. $(x, k) \in L$ if and only if $(x', k') \in L'$,
2. $k' = g(k)$ is a function only of k , and
3. the computation is accomplished in time $f(k)n^\alpha$, where $n = |x|$, α is a constant independent of both n and k , and f is an arbitrary function.

Hardness for $W[1]$ is the working criterion that a parameterized problem is unlikely to be FPT . The k -CLIQUE problem is $W[1]$ -complete, and often provides a convenient starting point for $W[1]$ -hardness demonstrations.

The main degree sequence of parameterized complexity is

$$FPT \subseteq W[1] \subseteq XP$$

There are only the barest beginnings of a structure theory of parametric intractability. Anyone interested in this area should take the recent work of Flum and Grohe as a fundamental reference [23], as well as the few investigations explicated in [17].

2.3 Connections to Practical Computing and Heuristics

What is practical computing, anyway? An amusing and thought-provoking account of this issue has been given by Karsten Weihe in the paper, “On the Differences Between Practical and Applied,” [44].

The crucial question is: *What are the actual inputs that practical computing implementations have to deal with?*

In considering “war stories” of practical computing, such as reported by Weihe, we are quickly forced to give up the idea that the real inputs (for most problems) fill up the definitional spaces of our mathematical modeling. The general rule also is that real inputs are *not random*, but rather have lots of hidden structure, that may not have a familiar name, even if you knew what it was. Weihe describes a problem concerning the train systems of Europe. Consider a bipartite graph $G = (V, E)$ where V is bipartitioned into two sets S (stations) and T (trains), and where an edge represents that a train t stops at a station s . The relevant graphs are huge, on the order of 10,000 vertices. The problem is to compute a minimum number of stations $S' \subseteq S$ such that every train stops at a station in S' . It is easy to see that this is a special case of the HITTING SET problem, and is therefore *NP*-complete. Moreover, it is also $W[1]$ -hard, so the straightforward application of the parameterized complexity program seems to fail as well.

However, the following two reduction rules can be applied to simplify (pre-process) the input to the problem. In describing these rules, let $N(s)$ denote the set of trains that stop at station s , and let $N(t)$ denote the set of stations at which the train t stops.

1. If $N(s) \subseteq N(s')$ then delete s .
2. If $N(t) \subseteq N(t')$ then delete t' .

Applications of these reduction rules cascade, preserving at each step enough information to obtain an optimal solution. Weihe found that, remarkably, these two simple reduction rules were strong enough to “digest” the original, huge input graph into a *problem kernel* consisting of disjoint components of size at most 50 — small enough to allow the problem to then be solved optimally by brute force.

Note that in the same breath, we have here a polynomial-time constant factor approximation algorithm, getting us a solution within a factor of 50 of optimal in, say, $O(n^2)$ time, just by taking *all* the vertices in the kernel components. We will have more to say about this powerful connection between *FPT* kernelization and polynomial-time constant factor approximation in §3.

What can we learn from Weihe’s train problem, and how does it relate to parameterized complexity? First of all, it displays one of the most universally applicable coping strategies for hard problems: *smart pre-processing*. In fact, it would be entirely silly *not* to undertake this sort of pre-processing for an *NP*-hard problem, even if the next phase is simulated annealing or neural nets. In a precise sense, this is *exactly* what fixed-parameter tractability is all about. The following equivalent definition of *FPT* displays this connection [19].

Definition 6. A parameterized language L is kernelizable if there is a parametric transformation of L to itself that satisfies:

1. the running time of the transformation of (x, k) into (x', k') , where $|x| = n$, is bounded a polynomial $q(n, k)$ (so that in fact this is a polynomial-time transformation of L to itself, considered classically, although with the additional structure of a parametric reduction),

2. $k' \leq k$, and
3. $|x'| \leq h(k)$, where h is an arbitrary function.

Lemma 1. *A parameterized language L is fixed-parameter tractable if and only if it is kernelizable.*

Weihe's example looks like an *FPT* kernelization, but what is the parameter? As a thought experiment, let us define $K(G)$ for a bipartite graph G to be the maximum size of a component of G when G is reduced according to the two simple reduction rules above. Then it is clear, although it might seem artificial, that HITTING SET can be solved optimally in *FPT* time for the parameter $K(G)$. We can add this new tractable parameterization of HITTING SET to the already known fact that HITTING SET can be solved optimally in *FPT* for the parameter *treewidth*.

As an illustration of the non-trivial power of non-trivial pre-processing, the reader will easily discover a reduction rule for VERTEX COVER that eliminates all vertices of degree 1. Not so easy is to show that all vertices of degree ≤ 3 can be eliminated, leaving as a kernel a graph of minimum degree 4. This pre-processing routine yields the best known heuristic algorithm for the general VERTEX COVER problem (i.e., no assumption that k is small), and also plays a central role in the best known *FPT* algorithm for VERTEX COVER.

We see in Weihe's train problem an example of a problem where the natural input distribution (graphs of train systems) occupies a limited parameter range, but the relevant parameter is not at all obvious. The inputs to one computational process (e.g., Weihe's train problem) are often the outputs of another process (the building and operating of train systems) that also are governed by computational and other feasibility constraints. We might reasonably adopt the view that the real world of computing involves a vast commerce in hidden structural parameters.

There is one last remark which seems important to make about the connections between parameterized complexity and heuristics. There is an *FPT* algorithm for the BREAKPOINT PHYLOGENY problem, for the natural parameter k taken to be the total cost of the (Steiner) tree that comprises the solution. (The definition of the problem is not important to this discussion.) In practice this is frequently bounded by $k \leq 50$. The running time of this *FPT* algorithm is $f(k) \cdot n^2$ where $f(k) = (k!)^3$ or thereabouts. One might be tempted to say that this is a fine example of "useless *FPT*" that displays the pathology of the definition of *FPT*, where the parameter function f is allowed to be arbitrarily horrible. But in fact, we are only reporting on an *ex post facto* analysis of an algorithm that has already been implemented, that is routinely in use, and that is considered state-of-the-art by the algorithms engineering / heuristics design community [37,12]. Such examples do not seem to be uncommon. Many "heuristic" algorithms currently in use are turning out to be *FPT* algorithms for natural and relevant parameters. Theorists who design *FPT* algorithms should keep in mind that their $f(k)$'s are only the best they are able to prove concerning a worst-case analysis, and that their algorithms may in fact be much more useful

than the parameter function indicates, on realistic inputs, particularly if any nontrivial kernelization is involved.

3 Some Research Frontiers

3.1 The Complexity of Approximation

The emphasis in the vast area of research on polynomial-time approximation algorithms is concentrated on the notions of:

- Polynomial-time constant factor approximation algorithms.
- Polynomial-time approximation schemes.

The connections between the *parameterized complexity* and *polynomial-time approximation* programs are actually very deep and developing rapidly. One of the reasons is that as one considers approximation schemes, there is immediately a parameter staring you in the eye: *the goodness of the approximation*. To illustrate what can happen, the first P -time approximation scheme for the Euclidean TSP due to Arora [4], gave solutions within a factor of $(1 + \epsilon)$ of optimal in time $O(n^{35/\epsilon^3})$. Thus for a 20% error we are looking at a “polynomial-time” algorithm with a running time of $O(n^{4.275})$. The lack of attention to this basic issue in the approximation community is more or less scandalous. The parameter $k = 1/\epsilon$ is one of the most important and obvious in all of the theory of computing.

Can we get the $k = 1/\epsilon$ out of the exponent? is a concrete question that calls out for further clarification for many known P -time approximation schemes. The following definition captures the essential issue.

Definition 7. *An optimization problem Π has an efficient P -time approximation scheme if it can be approximated to a goodness of $(1 + \epsilon)$ of optimal in time $f(k)n^c$ where c is a constant and $k = 1/\epsilon$.*

The following important theorem was first proved by Cristina Bazgan in her Master’s Thesis (independently by Cesati and Trevisan) [7,15].

Theorem 2. *Suppose that Π_{opt} is an optimization problem, and that Π_{param} is the corresponding parameterized problem, where the parameter is the value of an optimal solution. Then Π_{param} is fixed-parameter tractable if Π_{opt} has an efficient PTAS.*

Applying Bazgan’s Theorem is not necessarily difficult — we will sketch here a recent example. Khanna and Motwani introduced three planar logic problems in an interesting effort to give a general explanation of PTAS-approximability. Their suggestion is that “hidden planar structure” in the logic of an optimization problem is what allows PTASs to be developed [32]. They gave examples of optimization problems known to have PTASs, problems having nothing to do with graphs, that could nevertheless be reduced to these planar logic problems. The PTASs for the planar logic problems thus “explain” the PTASs for these other problems. Here is one of their three general planar logic optimization problems.

PLANAR TMIN

Input: A collection of Boolean formulas in sum-of-products form, with all literals positive, where the associated bipartite graph is planar (this graph has a vertex for each formula and a vertex for each variable, and an edge between two such vertices if the variable occurs in the formula).

Output: A truth assignment of minimum weight (i.e., a minimum number of variables set to *true*) that satisfies all the formulas.

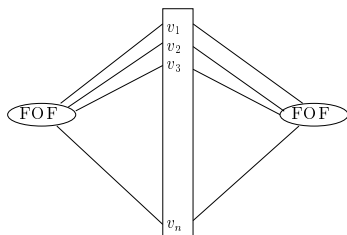
The following theorem is recent joint work with Cai, Juedes and Rosamond.

Theorem 3. *Planar TMIN does not have an EPTAS unless $FPT = W[1]$.*

Proof. We show that CLIQUE is parameterized reducible to PLANAR TMIN with the parameter being the weight of a truth assignment. Since CLIQUE is $W[1]$ -complete, it will follow that the parameterized form of PLANAR TMIN is $W[1]$ -hard.

To begin, let $\langle G, k \rangle$ be an instance of CLIQUE. Assume that G has n vertices. From G and k , we will construct a collection C of FOFs (sum-of-products formulas) over $f(k)$ blocks of n variables. C will contain at most $2f(k)$ FOFs and the incidence graph of C will be planar. Moreover, each minterm in each FOF will contain at most 4 variables. The collection C is constructed so that G has a clique of size k if and only if C has a weight $f(k)$ satisfying assignment with exactly one variable set to true in each block of n variables. Here we have that $f(k) = O(k^4)$.

To maintain planarity in the incidence graph for C , we ensure that each block of n variables appears in at most 2 FOFs. If this condition is maintained, then we can draw each block of n variables as follows.



We describe the construction in two stages. In the first stage, we use k blocks of n variables and a collection C' of $k(k-1)/2 + k$ FOFs. In a weight k satisfying assignment for C' , exactly one variable $v_{i,j}$ in each block of variables $b_i = [v_{i,1}, \dots, v_{i,n}]$ will be set to true. We interpret this event as “vertex j is the i th vertex in the clique of size k .” The $k(k-1)/2 + k$ FOFs are described as follows.

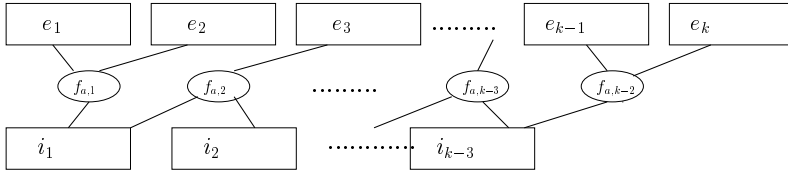
For each $1 \leq i \leq k$, let f_i be the FOF $\bigvee_{j=1}^n v_{i,j}$. This FOF ensures that at least one variable in b_i is set to true. For each pair $1 \leq i < j \leq k$, let $f_{i,j}$ be the FOF $v_{i,u}v_{j,v}$. Each FOF $f_{i,j}$ ensures that there is an edge in G between the i th vertex the clique and the j th vertex in the clique.

It is somewhat straightforward to show that $C' = \{f_1, \dots, f_k, f_{1,2}, \dots, f_{k-1,k}\}$ has a weight k satisfying assignment if and only if G has a clique of size k . To see this, notice that any weight k satisfying assignment for C' must satisfy exactly 1 variable in each block b_i . Each first order formula $f_{i,j}$ ensures that there is an edge between the i th vertex in the potential clique and the j th vertex in the potential clique. Notice also that, since we assume that G does not contain edges of the form (u, u) , the FOF $f_{i,j}$ also ensures that the i th vertex in the potential clique is not the j th vertex in the potential clique. This completes the first stage.

The incidence graph for the collection C' in the first stage is almost certainly not planar. In the second stage, we achieve planarity by removing crossovers in incidence graph for C' . Here we use two types of widgets to remove crossovers while keeping the number of variables per minterm bounded by 4. The first widget A_k consists of $k + k - 3$ blocks of n variables and $k - 2$ FOFs. This widget consists of $k - 3$ internal and k external blocks of variables. Each external block $e_i = [e_{i,1}, \dots, e_{i,n}]$ of variables is connected to exactly one FOF inside the widget. Each internal block $i_j = [i_{j,1}, \dots, i_{j,n}]$ is connected to exactly two FOFs inside the widget. The $k - 2$ FOFs are given as follows. The FOF $f_{a,1}$ is $\bigwedge_{j=1}^n e_{1,j} e_{2,j} i_{1,j}$. For each $2 \leq l \leq k - 3$, the FOF $f_{a,l} = \bigwedge_{j=1}^n i_{l-1,j} e_{l+1,j} i_{l,j}$.

Finally, $f_{a,k-2} = \bigwedge_{j=1}^n i_{k-3,j} e_{k-1,j} e_{k,j}$. These $k - 2$ FOFs ensure that the settings of variables in each block is the same if there is a weight $2k - 3$ satisfying assignment to the $2k - 3$ blocks of n variables.

The widget A_k can be drawn as follows.



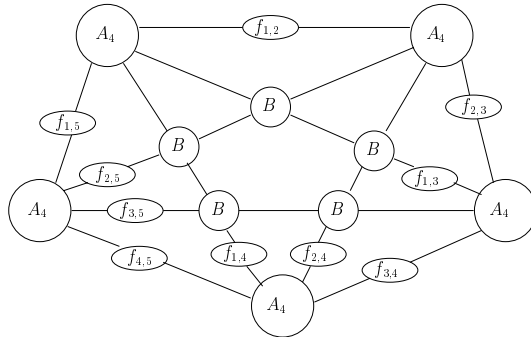
Since each internal block is connected to exactly two FOFs, the incidence graph for this widget can be drawn on the plane without crossing any edges.

The second widget removes crossover edges from the first stage of the construction. In the first stage, crossovers can occur in the incidence graphs because two FOFs may cross from one block to another. To eliminate this, consider each edge i, j in K_k with $i < j$ as a directed edge from i to j . In the construction, we send a copy of block i to block j . At each crossover point from the direction of block $u = [u_1, \dots, u_n]$ and $v = [v_1, \dots, v_n]$, insert a widget B that introduces 2 new blocks of n variables $u_1 = [u_{1,1} \dots u_{1,n}]$ and $v_1 = [v_{1,1} \dots v_{1,n}]$ and a FOF $f_B = \bigwedge_{j=1}^n \bigwedge_{l=1}^n u_j u_{l,j} v_l v_{l,j}$. The FOF f_B ensures that u_1 and v_1 are copies of u and v . Moreover, notice that the incidence graph for the widget B is also planar.

To complete the construction, we replace each of the original k blocks of n variables from the first stage with a copy of the widget A_{k-1} . At each crossover

point in the graph, we introduce a copy of widget B . Finally, for each directed edge between blocks (i, j) , we insert the original FOF $f_{i,j}$ between the last widget B and the destination widget A_{k-1} . Since one of the new blocks of variables created by the widget B is a copy of block i , the effect of the FOF $f_{i,j}$ in this new collections is the same as before.

The following diagram shows the full construction when $k = 5$.



Since each the incidence graph of each widget in this drawing is planar, the entire collection C of first order formulas has a planar incidence graph.

Now, if we assume that there are $c(k) = O(k^4)$ crossover points in standard drawing of K_k , then our collection has $c(k)$ B widgets. Since each B widget introduces 2 new blocks of n variables, this gives $2c(k)$ new blocks. Since we have k A_{k-1} widgets, each of which has $2(k-1) - 3 = 2k - 5$ blocks of n variables, this gives an additional $k(2k-5)$ blocks. So, in total, our construction has $f(k) = 2c(k) + 2k^2 - 5k = O(k^4)$ blocks of n variables. Note also that there are $g(k) = k(k-1)/2 + k(k-2) + c(k) = O(k^4)$ FOFs in the collection C .

As shown in our construction C has a weight $f(k)$ satisfying assignment (i.e., each block has exactly one variable set to true) if and only if the original graph G has a clique of size k . Since the incidence graph of C is planar and each minterm in each FOF contains at most four variables, it follows that this construction is a parameterized reduction as claimed. This completes the proof.

In a similar manner the other two planar logic problems defined by Khanna and Motwani can be shown to be $W[1]$ -hard. PTAS's for these problems are therefore likely never to be very useful, since the goodness of the approximation must apparently be paid for in the exponent of the polynomial running time.

A Second Connection. There is a second strong connection between parameterized complexity and polynomial-time approximation because finding natural, polynomial-time kernelization algorithms for FPT problems yielding small problem kernels (e.g., $|x'| \leq ck$) turns out to be intimately related to polynomial-time approximation algorithms (e.g., to within a factor of c of optimal). This export bridge from the former to the latter was first pointed out in [39]. See also [24]. We do not have the space to pursue this further here, but those who are interested in polynomial-time constant factor approximation algorithms will find this subject well worth exploring.

3.2 The Program of Elucidating FPT Optimality

In the classical framework, restricting the input to a problem can lead to polynomial time complexity, but generally, most hard (NP -complete or worse) problems remain hard when restricted to planar graphs and structures, for example. In the parameterized framework, almost all problems turn out to be FPT when restricted to planar inputs. In fact, for many planar parameterized graph problems, Kloks, Niedermeier and others have recently shown that FPT complexities of the form $c^{\sqrt{k}}n$ can be obtained [2,26]. This immediately raises the question of whether FPT complexities of this form might be achievable for the general unrestricted problems (such as the general parameterized VERTEX COVER problem), or whether lower bounds showing some “optimality” for FPT results might be possible.

In a recent (but flawed) paper Cai and Juedes [11] launched a powerful and exciting program that may resolve many such questions, and provide a lower bound “backstop” against which our efforts to design more efficient FPT algorithms can be measured. At the recent Dagstuhl Workshop on Parameterized Complexity, it was discovered that the proof of one of the key claims in [11] has a fatal error, but the program is still entirely viable if the following conjecture is true.

Conjecture. The following problem is hard for $W[1]$.

k -LOGVC

Input: A graph G on n vertices and a positive integer k .

Parameter: k

Question: Does G have a vertex cover of size at most $k \log n$?

Notice that the known $O(c^k n)$ FPT algorithm for the usual parameterization of VERTEX COVER allows k -LOGVC to be solved in time $O(n^k)$ — it’s almost like inquiring about a new parameterized problem that is the “complexity derivative” of the original FPT problem.

The conjecture would immediately imply that no FPT algorithm for the general VERTEX COVER problem can run in time $O(2^{o(k)} n^c)$ unless $FPT = W[1]$. The conjecture could also be used (potentially) to settle many other similar kinds of questions *via* reductions. For example, we might also ask, since we have achieved an FPT algorithm for PLANAR DOMINATING SET that runs in time $O(2^{O(k^{1/2})} n^c)$ [2] if perhaps an FPT running time of $O(2^{O(k^{1/3})} n^c)$ might be possible. If the conjecture is true, then the answer would be *no*, because (G, k) can be transformed into (G', k') in such a way that:

- (1) G has a k -vertex cover if and only if G' has a dominating set of size $c_1 k^2$
- (2) G' is planar.

Thus if PLANAR DOMINATING SET could be solved in $O(2^{o(\sqrt{k})} n^c)$ time, then the general VERTEX COVER problem could be solved in time $O(2^{o(k)} n^c)$ time, which (by the conjecture) would imply $FPT = W[1]$. The parametric reduction (due to recent work of Cai, Fellows, Juedes and Rosamond) proceeds in several steps: general VERTEX COVER is parametrically reduced to VERTEX COVER FOR GRAPHS OF MAXIMUM DEGREE 3, which is parametrically reduced

to PLANAR INDEPENDENT SET, and this in turn is parametrically reduced to PLANAR DOMINATING SET.

There is clearly a vast potential for this program, but the conjecture above is crucial — as well as elegant and interesting in itself.

References

1. K. Abrahamson, R. Downey and M. Fellows, “Fixed Parameter Tractability and Completeness IV: On Completeness for $W[P]$ and $PSPACE$ Analogs,” *Annals of Pure and Applied Logic* 73 (1995), 235–276. 294
2. J. Alber, H. Fernau and R. Niedermeier, “Parameterized Complexity: Exponential Speed-Up for Planar Graph Problems,” in: Proceedings of ICALP 2001, Crete, Greece, *Lecture Notes in Computer Science* vol. 2076 (2001), 261–272. 304
3. J. Alber, J. Gramm and R. Niedermeier, “Faster Exact Algorithms for Hard Problems: A Parameterized Point of View,” *Discrete Mathematics* 229 (2001), 3–27. 291
4. S. Arora, “Polynomial Time Approximation Schemes for Euclidean TSP and Other Geometric Problems,” In: *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, 1996. 300
5. V. Arvind, “On the Parameterized Complexity of Counting Problems,” *Workshop on Parameterized Complexity*, Madras, India, Dec. 2000. 294
6. V. Arvind, M. R. Fellows, M. Mahajan, V. Raman, S. S. Rao, F. A. Rosamond, C. R. Subramanian, “Parametric Duality and Fixed Parameter Tractability”, manuscript 2001. 294
7. C. Bazgan, “Schémas d’approximation et complexité paramétrée,” Rapport de stage de DEA d’Informatique à Orsay, 1995. 300
8. H. Bodlaender, R. Downey, M. Fellows, M. Hallett and H. T. Wareham, “Parameterized Complexity Analysis in Computational Biology,” *Computer Applications in the Biosciences* 11 (1995), 49–57. 293
9. Leizhen Cai, “The Complexity of Coloring Parameterized Graphs,” to appear in *Discrete Applied Math.* 295
10. M. Cesati and M. Di Ianni, “Parameterized Parallel Complexity,” Technical Report ECCC97-06, University of Trier, 1997. 293
11. Liming Cai and D. Juedes, “Subexponential Parameterized Algorithms Collapse the W -Hierarchy,” in: Proceedings of ICALP 2001, Crete, Greece, LNCS 2076 (2001). 304
12. M. Cosner, R. Jansen, B. Moret, L. Raubeson, L. Wang, T. Warnow and S. Wyman, “A New Fast Heuristic for Computing the Breakpoint Phylogeny and Experimental Phylogenetic Analyses of Real and Synthetic Data,” manuscript, 2000. 299
13. J. Chen, I. A. Kanj and W. Jia, “Vertex Cover: Further Observations and Further Improvements,” *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science* (WG’99), *Lecture Notes in Computer Science* 1665 (1999), 313–324. 292
14. Leizhen Cai and B. Schieber, “A Linear Time Algorithm for Computing the Intersection of All Odd Cycles in a Graph,” *Discrete Applied Math.* 73 (1997), 27–34. 294
15. M. Cesati and L. Trevisan, “On the Efficiency of Polynomial Time Approximation Schemes,” *Information Processing Letters* 64 (1997), 165–171. 300

16. M. Cesati and H. T. Wareham, "Parameterized Complexity Analysis in Robot Motion Planning," *Proceedings 25th IEEE Intl. Conf. on Systems, Man and Cybernetics: Volume 1*, IEEE Press, Los Alamitos, CA (1995), 880-885. 294
17. R. G. Downey and M. R. Fellows, *Parameterized Complexity*, Springer-Verlag, 1998. 291, 295, 297
18. R. Downey and M. Fellows, "Parameterized Complexity After Almost Ten Years: Review and Open Questions," in: *Combinatorics, Computation and Logic, DMTCS'99 and CATS'99*, Australian Computer Science Communications, Springer-Verlag Singapore, vol. 21 (1999), 1-33. 291
19. R. G. Downey, M. R. Fellows and U. Stege, "Parameterized Complexity: A Framework for Systematically Confronting Computational Intractability." In: *Contemporary Trends in Discrete Mathematics*, (R. Graham, J. Kratochvíl, J. Nešetřil and F. Roberts, eds.), Proceedings of the DIMACS-DIMATIA Workshop, Prague, 1997, *AMS-DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 49 (1999), 49-99. 291, 298
20. R. G. Downey, M. Fellows and U. Taylor, "The Parameterized Complexity of Relational Database Queries and an Improved Characterization of $W[1]$," in: *Combinatorics, Complexity and Logic: Proceedings of DMTCS'96*, Springer-Verlag (1997), 194-213. 293
21. F. Dehne, A. Rau-Chaplin, U. Stege and P. Taillon, "Solving Large FPT Problems on Coarse Grained Parallel Machines," manuscript, 2001. 292, 293
22. H. Fernau, "Remarks on Parameterized Enumeration," manuscript, 2001. 294
23. J. Flum and M. Grohe, "Describing Parameterized Complexity Classes," manuscript, 2001. 297
24. M. Fellows, C. McCartin, F. Rosamond and U. Stege, "Trees with Few and Many Leaves," manuscript, full version of the paper: "Coordinatized kernels and catalytic reductions: An improved FPT algorithm for max leaf spanning tree and other problems," *Proceedings of the 20th FST TCS Conference*, New Delhi, India, Lecture Notes in Computer Science vol. 1974, Springer Verlag (2000), 240-251. 303
25. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979. 292, 295
26. G. Gutin and T. Kloks, "Kernels in Planar Digraphs," manuscript, 2001. 304
27. M. Grohe, "Generalized Model-Checking Problems for First-Order Logic," *Proc. STACS 2001*, Springer-Verlag LNCS vol. 2001 (2001), 12-26. 291
28. M. Grohe, "The Parameterized Complexity of Database Queries," *Proc. PODS 2001*, ACM Press (2001), 82-92. 291
29. G. Gottlob, F. Scarcello and M. Sideri, "Fixed Parameter Complexity in AI and Nonmonotonic Reasoning," to appear in *The Artificial Intelligence Journal*. Conference version in: *Proc. of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, vol. 1730 of *Lecture Notes in Artificial Intelligence* (1999), 1-18. 293
30. M. Hallett, G. Gonnet and U. Stege, "Vertex Cover Revisited: A Hybrid Algorithm of Theory and Heuristic," manuscript, 1998. 292
31. F. Henglein and H. G. Mairson, "The Complexity of Type Inference for Higher-Order Typed Lambda Calculi." In *Proc. Symp. on Principles of Programming Languages (POPL)* (1991), 119-130. 293
32. S. Khanna and R. Motwani, "Towards a Syntactic Characterization of PTAS," in: *Proc. STOC 1996*, ACM Press (1996), 329-337. 300
33. S. Khot and V. Raman, 'Parameterized Complexity of Finding Subgraphs with Hereditary properties', *Proceedings of the Sixth Annual International Computing*

- and Combinatorics Conference (COCOON 2000)* July 2000, Sydney, Australia, Lecture Notes in Computer Science, Springer Verlag **1858** (2000) 137-147. **294**
34. O. Lichtenstein and A. Pnueli. "Checking That Finite-State Concurrents Programs Satisfy Their Linear Specification." In: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (1985), 97-107. **293**
 35. P. Moscato, "Controllability, Parameterized Complexity, and the Systematic Design of Evolutionary Algorithms," manuscript, 2001 (<http://www.densis.fee.unicamp.br/~moscato>).
 36. M. Mahajan and V. Raman, "Parameterizing Above Guaranteed Values: MaxSat and MaxCut," *J. Algorithms* 31 (1999), 335-354. **294**
 37. B. Moret, S. Wyman, D. Bader, T. Warnow and M. Yan, "A New Implementation and Detailed Study of Breakpoint Analysis," manuscript, 2000. **299**
 38. R. Niedermeier, "Some Prospects for Efficient Fixed-Parameter Algorithms," *Proc. SOFSEM'98* Springer-Verlag LNCS 1521 (1998), 168-185. **291**
 39. A. Natanzon, R. Shamir and R. Sharan, "A Polynomial-Time Approximation Algorithm for Minimum Fill-In," *Proc. ACM Symposium on the Theory of Computing* (STOC'98), ACM Press (1998), 41-47. **303**
 40. C. Papadimitriou and M. Yannakakis, "On the Complexity of Database Queries," *Proc. ACM Symp. on Principles of Database Systems* (1997), 12-19. **293**
 41. U. Stege, "Resolving Conflicts in Problems in Computational Biochemistry," Ph.D. dissertation, ETH, 2000. **292**
 42. M. Truszczyński, "On Computing Large and Small Stable Models," to appear in *Journal of Logic Programming*. **293**
 43. V. Raman, "Parameterized Complexity," in: *Proceedings of the 7th National Seminar on Theoretical Computer Science*, Chennai, India (1997), 1-18. **291**
 44. K. Weihe, "On the Differences Between Practical and Applied," Dagstuhl Workshop on Experimental Algorithmics, September 2000. **297**

Tight Bounds on Maximal and Maximum Matchings

Therese Biedl¹, Erik D. Demaine¹, Christian A. Duncan²,
Rudolf Fleischer³, and Stephen G. Kobourov⁴

¹ Department of Computer Science, Waterloo University
Waterloo, Canada
{biedl, eddemaine}@uwaterloo.ca

² Department of Computer Science, University of Miami
Miami, FL, USA
duncan@cs.miami.edu

³ Department of Computer Science,
The Hong Kong University of Science and Technology, Hong Kong
rudolf@cs.ust.hk

⁴ Department of Computer Science, University of Arizona
Tucson, AZ, USA
kobourov@cs.arizona.edu

Abstract. In this paper, we study bounds on maximal and maximum matchings in special graph classes, specifically triangulated graphs and graphs with bounded maximum degree. For each class, we give a lower bound on the size of matchings, and prove that it is tight for some graph within the class.

1 Introduction

The problem of finding a maximum matching in a graph has a long and distinguished history beginning with the early work of Petersen [11], König [9], Hall [6], and Tutte [13]. The fastest algorithms to find a maximum matching in an n -vertex m -edge graph takes $O(\sqrt{nm})$ time, for bipartite graphs [7] as well as for general graphs [10].

One intensely studied topic is whether a graph has a perfect matching, i.e., a matching of size $n/2$. This was shown for 3-regular biconnected graphs [11] and for k -regular bipartite graphs [9], and the perfect matching can be found efficiently for these graphs [2,12,4]. Tutte [13] characterized when a graph has a perfect matching, but no algorithm that can find a perfect matching in an arbitrary graph faster than finding a maximum matching is known.

Not as much is known about bounds for graphs that do not have a perfect matching. Recently, Duncan, Goodrich and Kobourov [5] showed that any planar triangulated graph has a matching of size $\frac{n}{12}$ that satisfies additional constraints.

Our research was originally motivated by the question whether the bound of $\frac{n}{12}$ in [5] could be improved by dropping the extra constraints. Thus, we studied the size of maximal and maximum matchings in planar triangulated graphs. (We

included maximal matchings because such matchings can be computed easily in linear time.)

It is known that every triangulated planar graph without separating triangles is 4-connected, hence has a Hamiltonian cycle [14], and hence a matching of size $\lfloor \frac{n}{2} \rfloor$. As we will see, we can generalize this to all triangulated planar graphs by including the number of separating triangles (or more precisely, the number of leaves in the tree of 4-connected components) in the bound on the matching.

Next, we study graphs with small maximum degree. It is known that every 3-regular biconnected graph has a perfect matching [11]. As we will see, we can generalize this to all graphs with maximum degree 3 by including the number of cutvertices (or more precisely, the number of leaves in the tree of 2-connected components) and the number of vertices of smaller degree. The proof for maximal matchings generalizes even further to graphs of maximum degree k .

An overview of our results is given in Table 1. All entries are lower bounds on the size of the matching of a certain type. Also, all bounds are tight for some graph within this class. We typically give two bounds: one bound that depends only on n or m , and one bound that also includes other parameters of the graph.

Table 1. Overview of the results in this paper. Here ℓ_4 denotes the number of leaves in the 4-block tree, ℓ_2 denotes the number of leaves in the 2-block tree, and n_2 denotes the number of vertices of degree 2 (see Section 2 for precise definitions). All bounds in the table are tight

Graph	Matching type	Bound 1	Bound 2
Triangulated planar	Maximal	$\frac{n+4}{6}$	$\frac{n+2}{4} - \frac{\ell_4}{6}$
	Maximum	$\frac{n+4}{3}$	$\frac{n}{2} - \frac{\ell_4}{4} + 1$
Max-deg k	Maximal	$\frac{m}{4k-2}$	$\frac{m}{4k-2}$
Max-deg 3	Maximum	$\frac{n-1}{3}$	$\frac{n}{2} - \frac{\ell_2}{3} - \frac{n_2}{6}$
3-regular	Maximum	$\frac{4n-1}{9}$	$\frac{n}{2} - \frac{\ell_2}{3}$

2 Definitions

Let $G = (V, E)$ be a graph with vertices V and edges E , we denote $|V| = n(G) = n$ and $|E| = m(G) = m$. Denote by n_i the number of vertices of degree i , i.e., with exactly i incident edges. We call G *3-regular* if every vertex has degree 3, and a *max-deg- k graph* if every vertex has degree at most k . G is called *simple* if there are no loops and no multiple edges, and *connected* if for any pair of vertices there exists a path from one vertex to the other. In this paper, we assume that G is simple and connected.

A connected graph G is called *k -connected* if for any set C of at most $k-1$ vertices, the graph that results from deleting the vertices in C is still connected.

A 2-connected graph is also called *biconnected*. If a connected graph is not biconnected, then it must have a vertex v such that $G - v$ is not connected; such a vertex is called a *cutvertex*. If G has cutvertices, then its *biconnected components* are the maximal biconnected subgraphs of the graph. The *2-block tree* is obtained by defining one node for every biconnected component and one node for every cutvertex, and connecting two nodes if and only if one is a cutvertex contained in the biconnected component of the other node. As the name suggests, the 2-block tree is a tree. Let $\ell_2(G)$ denote the number of *leaves* in the 2-block tree; we write ℓ_2 if the graph in question is clear.

A *planar graph* is a graph that can be drawn in the plane without a crossing. Such a planar drawing divides the plane into connected pieces called *faces*. The *degree* of a face is the number of times that a vertex is incident to a boundary of a face. In a simple planar graph with at least three vertices, every face has degree at least 3. A planar graph is called *triangulated* if all faces have degree 3 (i.e., they are a 3-cycle, also called a *triangle*). A triangulated graph has exactly $3n - 6$ edges and is 3-connected.

A *separating triangle* in a planar graph is a triangle that is not the boundary of a face, i.e., a triangle such that there are vertices both inside and outside the triangle. Assume that G is a triangulated graph that is not 4-connected. Then there exist three vertices $\{u, v, w\}$ such that removing them splits G into at least two parts. Since G is triangulated, $\{u, v, w\}$ must form a separating triangle. Hence a triangulated graph is 4-connected if and only if it has no separating triangle.

If G is a triangulated graph that is not 4-connected, then we can split it into its 4-connected components as follows. We say that a separating triangle T_1 is *inside* another separating triangle T_2 if none of the vertices of T_2 is in the outside of T_1 . (Note that some vertices may be in both triangles.) Let $T_1, \dots, T_k$ be those separating triangles that are not inside any other separating triangle. Denote by G^0 the graph that results from G by deleting all vertices that are inside $T_1, \dots, T_k$; then G^0 is a 4-connected graph. For $i = 1, \dots, k$, denote by G_i the graph that results from taking all vertices inside T_i and adding the vertices of T_i . Recursively compute the 4-connected components of $G_1, \dots, G_k$; these are also 4-connected components of G .

Following the construction of 4-connected components, one can obtain a tree to store these components. The root of this tree is G^0 , the 4-connected graph that results from deleting the insides of the separating triangles. This component has one child for each separating triangle that is not inside another separating triangle. The subtrees of these children are computed recursively from $G_1, \dots, G_k$. This so-called *4-block tree* can be computed in $O(n)$ time [8]. Denote by $\ell_4(G)$ the number of leaves of the 4-block tree; we write ℓ_4 if the graph in question is clear.

A *matching* is a set M of edges such that no vertex has two or more incident edges in M . For a given matching M , define V_M to be the *matched vertices*, i.e., the vertices with an incident edge in M , and V_U to be the *unmatched vertices*, i.e., $V - V_M$. A matching is called a *maximal matching* if there is no edge between two

unmatched vertices, i.e., we cannot add an edge to the matching. A matching is called a *maximum matching* if it has the maximum possible cardinality among all matchings. A *perfect matching* is a matching that leaves no unmatched vertices, i.e., a matching with $n/2$ edges.

2.1 Tutte's Theorem and Berge's Generalization

In 1947, Tutte [13] proved a characterization of the existence of a perfect matching. His theorem uses the concept of *odd components* which is explained by the following. Let T be an arbitrary subset of vertices. Removing T from the graph may split the graph into a number of connected components. Some of those may have an even number of vertices, and some may have an odd number of vertices. We denote by $o(T)$ the number of components of $G - T$ that have an odd number of vertices; these are also called the *odd components*.

Tutte proved that a graph has a perfect matching if and only if for any vertex set T , the number of odd components of T is not bigger than $|T|$. Berge showed in 1957 how to extend this theorem to characterize the size of a maximum matching, again using vertex sets T and their odd components.

Lemma 1. [13, 1] *Let G be a graph. For any set $T \subset V$, any matching contains at least $o(T) - |T|$ unmatched vertices. Moreover, there exists a set $T \subset V$ such that any maximum matching of G contains exactly $o(T) - |T|$ unmatched vertices.*

3 Triangulated Graphs

Duncan, Goodrich and Kobourov [5] proved that any planar triangulated graph has a matching with at least $n/12$ edges that do not belong to any separating triangle. It trivially follows that any triangulated planar graph has a matching of size at least $n/12$. We will here obtain better bounds by dropping the condition on separating triangles.

3.1 Maximal Matching for Triangulated Graphs

First we study maximal matchings. We initially give a bound that depends on the number of leaves in the 4-block tree, and then estimate the number of such leaves. We need an easy observation about the relationship between face sizes and vertices in a planar graph.

Lemma 2. *Assume that G is a planar graph with $n \geq 3$ vertices, that has f_3 faces of degree 3 and f_4 faces of degree at least 4. Then $f_3 + 2f_4 \leq 2n - 4$.*

Proof: We have $3f_3 + 4f_4 \leq 2m$, since the left-hand side counts each edge at most twice. Also, $m \leq 3n - 6 - f_4$, because a triangulated graph has $3n - 6$ edges, and there is at least one missing edge for every face of degree at least 4. Combining the two inequalities gives $3f_3 + 4f_4 \leq 6n - 12 - 2f_4$, which after rearranging yields the result. $\square$

Lemma 3. *Any maximal matching of a planar triangulated graph with at least 4 vertices has size at least $\frac{n+2}{4} - \frac{1}{8}\ell_4$, where ℓ_4 is the number of leaves in the 4-block tree of the graph.*

Proof: Let M be an arbitrary maximal matching, and let V_M and V_U be the matched and unmatched vertices. Let G_M be the graph induced by the matched vertices. Since G has at least 4 vertices, it must have at least four matched vertices, so $|V_M| \geq 4$ and G_M has no faces of degree less than 3.

In any face of G_M , there can be at most one unmatched vertex of G , for if there were two or more unmatched vertices, then (because G is triangulated) there must be an edge between them, contradicting the maximality. We split the unmatched vertices into two groups: V_U^3 denotes those that are inside a face of G_M of degree 3, whereas V_U^4 denotes those that are inside a face of G_M of degree at least 4. Note that $|V_U^3| \leq \ell_4$, because if there is a vertex inside a triangular face of G_M , then this triangle contains a vertex inside and also a vertex outside (by $n(G_M) \geq 4$), hence it is a separating triangle in G and contains a leaf of the 4-block tree.

By Lemma 2, we have $|V_U^3| + 2|V_U^4| \leq 2n(G_M) - 4 = 2|V_M| - 4$. Since $|V_U^3| + |V_U^4| + |V_M| = n$, we can reformulate this further as $|V_U^3| + 2|V_U^4| = 2(n - |V_U^3| - |V_U^4|) - 4$. Therefore $3|V_U^3| + 4|V_U^4| \leq 2n - 4$, and

$$|V_U| \leq \frac{1}{4}(3|V_U^3| + 4|V_U^4|) + \frac{1}{4}|V_U^3| \leq \frac{1}{4}(2n - 4) + \frac{1}{4}\ell_4,$$

which implies $|V_M| \geq n - \frac{n-2}{2} - \frac{1}{4}\ell_4 = \frac{n+2}{2} - \frac{1}{4}\ell_4$ as desired. $\square$

Now we need a bound on ℓ_4 . Kant [8] stated that every planar triangulated graph has at most $n - 4$ separating triangles. Since he did not prove this claim, we give a proof here for completeness' sake.

Lemma 4. *Any planar triangulated graph has at most $n - 4$ separating triangles.*

Proof: The proof is by induction. If a graph has a separating triangle, then it must have the three vertices of the triangle and one vertex both inside and outside, so $n \geq 5$, and a graph with 5 vertices can have only one separating triangle. Assume the claim holds for all values up to $n - 1$, $n \geq 6$. Let G be any graph of n vertices, and assume it has a separating triangle $\{u, v, w\}$; otherwise we are done. Let G_i and G_o be the graphs inside and outside $\{u, v, w\}$, respectively. We have $n(G_i) + n(G_o) = n + 3$. Both graphs have fewer vertices than G , and so by induction have at most $n(G_i) - 4$ and $n(G_o) - 4$ separating triangles, respectively. Hence, the number of separating triangles in G is at most $1 + (n(G_i) - 4) + (n(G_o) - 4) = n - 4$. $\square$

This bound is tight, see for example the graph class $\mathcal{H}$ that was defined in [DGK99]. Combining this bound with Lemma 6, we obtain that any planar triangulated graph has a matching of size at least $\frac{n+3}{4}$. However, we can do better by obtaining a bound on the number of leaves in the 4-block tree.

Lemma 5. *Any planar triangulated graph has at most $\frac{2}{3}(n - 2)$ leaves in the 4-block tree.*

Proof: As before, denote by ℓ_4 the number of leaves in the 4-block tree. Let $T_1, \dots, T_{\ell_4}$ be those separating triangles that form the leaves, and let G_L be the graph induced by the vertices in $T_1, \dots, T_{\ell_4}$. For each triangle T_i , there is at least one vertex inside T_i that does not belong to any of the other triangles, and therefore not to G_L . Hence $n(G_L) \leq n - \ell_4$.

Every triangle T_i is a face of G_L (because these triangles are leaves of the 4-block tree). But the number of faces in G_L is at most $2n(G_L) - 4$ since G_L is planar. Hence $\ell_4 \leq 2n(G_L) - 4 \leq 2(n - \ell_4) - 4$, or $3\ell_4 \leq 2n - 4$, which yields the result. $\square$

Combining this bound with the bound of Lemma 3, we obtain that every maximal matching of a triangulated planar graph has size at least $\frac{n+2}{4} - \frac{1}{8} \cdot \frac{2}{3}(n-2) = \frac{n+4}{6}$.

Theorem 1. *Any maximal matching of a triangulated planar graph with at least 4 vertices has size at least $\frac{n+4}{6}$.*

The above bound is tight, i.e., there exists a planar triangulated graph with a maximal matching of size $(n+4)/6$. To see this, take any planar triangulated graph G that has a perfect matching M , and add into each face of G one more vertex connected to the three neighbors. Call the resulting graph G' , and its number of vertices n' . Then $n' = n + 2n - 4$ (because G has $2n - 4$ faces). Also, M is a maximal matching in G' and $|M| = n/2 = (n' + 4)/6$.

3.2 Maximum Matching for Triangulated Graphs

In this section, we provide a bound on the size of a maximum matching in a planar triangulated graph. As we will see, this bound will again depend on the number of leaves in the 4-block tree.

Lemma 6. *Any planar triangulated graph G has a matching of size at least $\min\{\lfloor \frac{n}{2} \rfloor, \frac{n}{2} - \frac{\ell_4}{4} + 1\}$, where ℓ_4 is the number leaves of the 4-block tree of G .*

Proof: Let M be a maximum matching, and let T be a vertex set such that there are $o(T) - |T|$ unmatched vertices, i.e., $|V_U| = o(T) - |T|$ (Lemma 1). The claim holds if $|T| \leq 2$, because then $o(T) \leq 1$ since G is 3-connected, and there is at most one unmatched vertex. The claim also holds if $|T| = 3$, because then there are at most two odd components (the inside and the outside of the separating triangle). So we may assume that $|T| \geq 4$.

Let G_T be the graph that is induced by the vertices of T . Observe that no two odd components can be within the same face of G_T since G is triangulated. Let $o_3(T)$ and $o_4(T)$ be the number of odd components that are inside a face of G_T of degree 3 and degree at least 4, respectively. Note that $o_3(T) \leq \ell_4$, because if there is an odd component inside a triangular face of G_T , then this triangle contains a vertex inside and also a vertex outside (by $|T| \geq 4$), hence it is a separating triangle in G and contains a leaf of the 4-block tree.

By Lemma 2, we know that $o_3(T) + 2o_4(T) \leq 2n(G_T) - 4 = 2|T| - 4$, and

$$\begin{aligned} |V_U| &= o(T) - |T| = o_3(T) + o_4(T) - |T| \\ &= \frac{1}{2}(o_3(T) + 2o_4(T)) + \frac{1}{2}o_3(T) - |T| \\ &\leq \frac{1}{2}(2|T| - 4) + \frac{\ell_4}{2} - |T| = \frac{\ell_4}{2} - 2. \end{aligned}$$

So $|V_M| \geq n - |V_U| \geq n - \frac{\ell_4}{2} + 2$ as desired. $\square$

Combining Lemma 6 with Lemma 5, we obtain the bound on maximum matching in triangulated planar graphs.

Theorem 2. *Every planar triangulated graph with at least 10 vertices has a matching of size at least $\frac{n+4}{3}$.*

Proof: There is nothing to prove for $n \geq 10$ if G has a matching of size $\lfloor \frac{n}{2} \rfloor$. Otherwise, G has a matching of size $\frac{n}{2} - \frac{\ell_4}{4} + 1 \geq \frac{n}{2} - \frac{1}{6}(n - 2) + 1 = \frac{n+4}{3}$. $\square$

The above bound is tight, i.e., there exists a graph for which any matching has at most $(n + 4)/3$ edges. This graph is defined for any $n \equiv 2 \pmod 3$, $n \geq 11$ and shown in Figure 1. It consists of a cycle with $(n - 2)/3$ vertices, two vertices connected to each vertex of the cycle (these parts are shown in black), and one more vertex in every face of the above graph (this part is shown in white).

Let T be the $(n + 4)/3$ black vertices. Since there are no edges between white vertices, graph $G - T$ has $(2n - 4)/3$ isolated vertices, which each form an odd component, so $o(T) - |T| = (2n - 4)/3 - (n + 4)/3 = (n - 8)/3$. Hence in any matching M of the graph, at least $(n - 8)/3$ vertices are unmatched and at most $(2n + 8)/3$ vertices are matched, so $|M| \leq (n + 4)/3$.

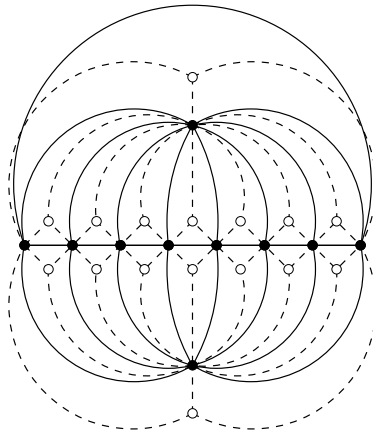


Fig. 1. A planar triangulated graph with a maximum matching of size $\frac{n+4}{3}$

Note also that this graph has $\ell_4 = \frac{2}{3}(n - 2)$ separating triangles which are all leaves of the 4-block tree, so Lemma 5 is tight as well.

4 Graphs with Maximum Degree k

Now we devote our attention to another graph class with a special structure; graph with maximum degree k . We chose this graph class because 3-regular biconnected graphs are known to have a perfect matching, and we tried to generalize this to graphs with bounded maximum degree.

Theorem 3. *Any maximal matching of a max-deg- k graph has size at least $m/(4k - 2)$.*

Proof: Let M be an arbitrary maximal matching, and let V_M and V_U be the matched and unmatched vertices. We split V_U into k sets, $V_U^i, i = 1, \dots, k$, where V_U^i is the set of unmatched vertices with degree i .

Let E_U be the set of edges with at least one endpoint in V_U . Recall that since M is maximal no edge can have both endpoints in V_U . Therefore, E_U is the set of all edges between vertices in V_U and vertices in V_M , and $|E_U| = \sum_{i=1}^k i|V_U^i|$. Since every vertex in V_M is incident to at most k vertices, and at least one of them is also in V_M , we get $|E_U| \leq (k - 1)|V_M|$. Combining, we have

$$\begin{aligned} |V_U| &= \sum_{i=1}^k |V_U^i| \\ &= \sum_{i=1}^k \frac{i}{k} |V_U^i| + \sum_{i=1}^k \frac{k-i}{k} |V_U^i| \\ &= |E_U|/k + \sum_{i=1}^k \frac{k-i}{k} |V_U^i| \\ &\leq (k-1)|V_M|/k + \sum_{i=1}^k \frac{k-i}{k} |V_U^i| \\ &\leq (k-1)|V_M|/k + \sum_{i=1}^k \frac{k-i}{k} n_i \end{aligned}$$

Solving for V_M we get

$$|V_M| = n - |V_U| \geq n - (k-1)|V_M|/k - \sum_{i=1}^k \frac{k-i}{k} n_i$$

and therefore

$$|V_M| \geq \frac{k}{2k-1} \left(n - \sum_{i=1}^k \frac{k-i}{k} n_i \right) = \frac{k}{2k-1} \left(n - \sum_{i=1}^k n_i + \frac{1}{k} \sum_{i=1}^k i n_i \right) = \frac{m}{2k-1},$$

which yields the result. $\square$

This bound is tight, as illustrated in the graph in Figure 2. The bold edges indicate a maximal matching of size $\frac{m}{4k-2}$.

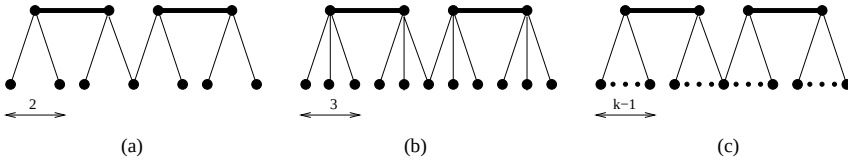


Fig. 2. A max-deg- k graph with a maximal matching of size $m/(4k-2)$ for (a) $k = 3$, (b) $k = 4$, (c) arbitrary k

4.1 Maximum Matching for Max-deg-3 Graphs

We have not succeeded in obtaining a better bound for a *maximum* matching in a graph with maximum degree k , except when $k = 3$.

Lemma 7. *Any max-deg-3 graph G has a matching of size at least $\frac{n}{2} - \frac{\ell_2}{3} - \frac{n_2}{6}$, where ℓ_2 is the number of leaves in the 2-block tree of G and n_2 is the number of vertices of degree 2.*

Proof: Let M be a maximum matching, and let T be a vertex set such that there are $o(T) - |T|$ unmatched vertices (Lemma 1). We define the following three quantities: $o_1(T)$, $o_2(T)$, and $o_3(T)$ are the number of odd components joined to T by one edge, two edges, and at least three edges, respectively. Every odd component joined to T by one edge contains a leaf of the 2-block tree, so $o_1(T) \leq \ell_2$. Every odd component joined to T by two edges must contain at least one vertex of degree 2 (otherwise there would be an odd number of vertices of odd degree), so $o_2(T) \leq n_2$.

The number of edges incident to T is at least $o_1(T) + 2o_2(T) + 3o_3(T)$, but also at most $3|T|$ since G has maximum degree 3. Therefore

$$\begin{aligned} |V_U| = o(T) - |T| &\leq \frac{1}{3}(o_1(T) + 2o_2(T) + 3o_3(T)) + \frac{2}{3}o_1(T) + \frac{1}{3}o_2(T) - |T| \\ &\leq \frac{1}{3}(3|T|) + \frac{2}{3}\ell_2 + \frac{1}{3}n_2 - |T| = \frac{2}{3}\ell_2 + \frac{1}{3}n_2, \end{aligned}$$

and $|V_M| \geq n - \frac{2}{3}\ell_2 - \frac{n_2}{3}$, which proves the claim. $\square$

To obtain a bound that only depends on n , we need to bound ℓ_2 and n_2 .

Lemma 8. *Every max-deg-3 graph has $2\ell_2 + n_2 \leq n + 2$, where ℓ is the number of leaves in the 2-block tree and n_2 is the number of vertices of degree 2.*

Proof: Let G be a connected max-deg-3 graph. If all leaves of the 2-block tree of G contain a vertex of degree 1, then $\ell_2 = n_1$ (because every vertex of degree 1 implies a leaf). A simple counting argument shows that $n_1 \leq n_3 + 2$, and hence $2\ell_2 + n_2 = 2n_1 + n_2 \leq n_1 + n_2 + n_3 + 2 = n + 2$.

If some leaves of the 2-block tree of G do not contain a vertex of degree 1, then obtain a new graph G' by deleting from these leaves all vertices except the cutvertex. Note that G and G' have equally many leaves of the 2-block tree, and G' has at most as many vertices of degree 2 as G . Since the claim holds for G' , we have $2\ell_2(G) + n_2(G) \leq 2\ell_2(G') + n_2(G') \leq n(G') + 2 \leq n(G) + 2$. $\square$

Combining this lemma with Lemma 7, we obtain that the number of unmatched vertices is at most $\frac{2}{3}\ell_2 + \frac{n_2}{3} \leq \frac{n+2}{3}$, hence the maximum matching has size at least $\frac{n-1}{3}$.

Theorem 4. *Every max-deg 3 graph has a matching of size at least $\frac{n-1}{3}$.*

This bound is tight, as can be seen from the graph in Figure 3, for which the maximum matching has size $\frac{n-1}{3}$.

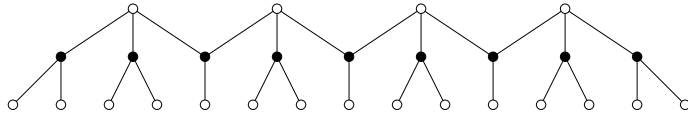


Fig. 3. A max-deg-3 graph for which a maximum matching has size $\frac{n-1}{3}$

One can observe that this graph does not have any vertices of degree 2. However, the factor “ $-\frac{n_2}{6}$ ” in Lemma 7 is tight, as demonstrated by the following example: Consider any 3-regular graph with n vertices and $m = \frac{3}{2}n$ edges. Now split every edge into two, and add a degree-two vertex in the middle. This gives a new graph G' with $n' = n + m = \frac{5}{2}n$ vertices and $n'_2 = \frac{3}{2}n$ vertices of degree 2.

A maximum matching of G' has at most $n' - \frac{n'_2}{2}$ matched vertices, because setting T to be the n original vertices, we obtain $\frac{3}{2}n$ odd components from the added vertices. So the maximum matching has size $\frac{n'}{2} - \frac{n'_2}{4} = n$. Since $\frac{n'}{6} = \frac{n}{4}$, this proves that the bound of Lemma 7 is tight.

Note, however, that the size of the maximum matching for this graph is $\frac{2}{5}n' > \frac{1}{3}n'$. It remains open whether there exists a better bound on the size of the maximum matching if a graph is forced to have vertices of degree 2, for example whether a bound of $\frac{1}{3}n + \frac{1}{5}n_2$ holds for the size of a maximum matching in a graph with maximum degree 3.

4.2 Maximum Matchings for 3-regular Graphs

For 3-regular graphs we can improve the bounds of Theorem 4 even further.

Lemma 9. *Every 3-regular graph has at most $\frac{n+2}{6}$ leaves in the 2-block tree.*

Proof: Let C be a biconnected component that is a leaf in the 2-block tree, and let v be its unique cutvertex. We claim that C has at least 5 vertices, and prove this as follows: Since G is simple, v must have a neighbor $w \neq v$ in C .

Since G is 3-regular, w must have 3 neighbors, which are all in C since w is not a cutvertex. So C has at least 4 vertices. Since all vertices except v in C have odd degree, but v has even degree, C has an odd number of vertices, so C has at least 5 vertices.

Let G_L be the graph that results from G by deleting all vertices that are part of a leaf of the 2-block tree and not a cutvertex. Hence for every leaf we delete at least 4 vertices, so $n(G_L) \leq n - 4\ell_2$. The remaining graph is connected, hence $m(G_L) \geq n(G_L) - 1$. Also, every cutvertex that belonged to a leaf of G has degree 1 in G_L , whereas all other vertices have degree 3, so $2m(G_L) = \ell_2 + 3(n(G_L) - \ell_2)$. Thus we obtain $\ell_2 + 3(n(G_L) - \ell_2) = 2m(G_L) \geq 2n(G_L) - 2$, which implies $n(G_L) \geq 2\ell_2 - 2$, therefore $2\ell_2 \leq n(G_L) + 2 \leq n - 4\ell_2 + 2$ and $\ell_2 \leq \frac{1}{6}(n + 2)$. $\square$

Consequently, the maximum matching of a 3-regular graph has size at least $\frac{n}{2} - \frac{\ell_2}{3} = \frac{n}{2} - \frac{1}{18}(n + 2) = \frac{1}{9}(4n - 1)$.

Theorem 5. *Every 3-regular graph has a matching of size at least $\frac{4n-1}{9}$.*

This bound is also tight, which can be seen by attaching the smallest possible 3-regular graph to every leaf of the graph of Figure 3. The resulting graph (shown in Figure 4) is defined for $n = 16 \bmod 18$. The set of black vertices has size $\frac{n+2}{9}$, and yields $\frac{2n+11}{9}$ odd components. Hence any matching has size at most $\frac{4n-1}{9}$.

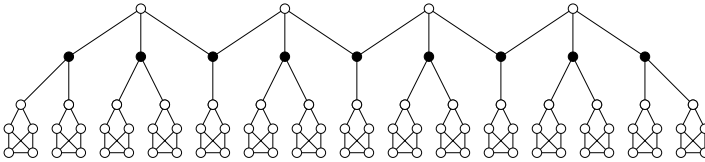


Fig. 4. A simple 3-regular graph with a maximum matching of size $\frac{4n-1}{9}$

5 Conclusion

In this paper, we studied bounds on the size of maximal and maximum matchings in special graphs classes, in particular triangulated planar graphs, graphs with maximum degree k , graphs with maximum degree 3 and 3-regular graphs. We obtain lower bounds on the size of such matchings, and showed that the bounds are tight for some graph within the class.

We leave a number of open problems:

- How quickly can we find matchings that are known to exist? A maximal matching can be found in linear time, but can we find, say, a matching of size $\frac{n}{2} - \frac{\ell_4}{4} + 1$ in a planar triangulated graph in less than $O(m\sqrt{n})$ time?

- What can be said about the size of a maximum matching in a graph with maximum degree k ? Can we obtain a bound better than $m/(4k - 2)$?
- Is there a graph with maximum degree 3 for which a maximum matching has size $\frac{n}{2} - \frac{\ell_2}{3} - \frac{n_2}{6}$, and which has a significant number of vertices of degree 2? Or if not, can we show a better bound?

Acknowledgements

The authors would like to thank Timothy Chan, Martin Demaine, Alastair Farugia, Lars Jacobsen, Ian Munro, Morten Nielsen and Paul Nijjar for useful discussions.

References

1. C. Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci. U. S. A.*, 43:842–844, 1957. 311
2. T. Biedl, P. Bose, E. Demaine, and A. Lubiw. Efficient algorithms for Petersen’s theorem. *J. Algorithms*, 38:110–134, 2001. 308
3. J. Bondy and U. Murty. *Graph Theory and Applications*. American Elsevier Publishing Co., 1976.
4. R. Cole, K. Ost, and S. Schirra. Edge-coloring bipartite multigraphs in $O(E \log D)$ time. Technical Report TR1999-792, Department of Computer Science, New York University, September 1999. 308
5. C. Duncan, M. Goodrich, and S. Kobourov. Planarity-preserving clustering and embedding for large planar graphs. In *Graph Drawing (GD’99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 186–196. Springer-Verlag, 1999. Accepted for publication in *Computational Geometry: Theory and Applications*. 308, 311
6. P. Hall. On representation of subsets. *Journal of the London Mathematical Society*, 10:26–30, 1935. 308
7. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973. 308
8. G. Kant. A more compact visibility representation. *Internat. J. Comput. Geom. Appl.*, 7(3):197–210, 1997. 310, 312
9. D. König. Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre. *Mathematische Annalen*, 77:453–465, 1916. 308
10. S. Micali and V. V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science*, pages 17–27, New York, 1980. Institute of Electrical and Electronics Engineers Inc. (IEEE). 308
11. J. Petersen. Die Theorie der regulären graphs (The theory of regular graphs). *Acta Mathematica*, 15:193–220, 1891. 308, 309
12. A. Schrijver. Bipartite edge coloring in $O(\Delta m)$ time. *SIAM J. Comput.*, 28(3):841–846, 1999. 308
13. W. T. Tutte. The factorization of linear graphs. *Journal of the London Mathematical Society*, 22:107–111, 1947. 308, 311
14. W. T. Tutte. A theorem on planar graphs. *Trans. Amer. Math. Soc.*, 82:99–116, 1956. 309

Recognition and Orientation Algorithms for P_4 -Comparability Graphs

Stavros D. Nikolopoulos and Leonidas Palios

Department of Computer Science, University of Ioannina
GR-45110 Ioannina, Greece
{stavros,palios}@cs.uoi.gr

Abstract. We consider two problems pertaining to P_4 -comparability graphs, namely, the problem of recognizing whether a simple undirected graph is a P_4 -comparability graph and the problem of producing an acyclic P_4 -transitive orientation of a P_4 -comparability graph. These problems have been considered by Hoàng and Reed who described $O(n^4)$ and $O(n^5)$ -time algorithms for their solution respectively, where n is the number of vertices of the given graph. Recently, Raschle and Simon described $O(n + m^2)$ -time algorithms for these problems, where m is the number of edges of the graph.

In this paper, we describe different $O(n + m^2)$ -time algorithms for the recognition and the acyclic P_4 -transitive orientation problems on P_4 -comparability graphs. Instrumental in these algorithms are structural relationships of the P_4 -components of a graph, which we establish and which are interesting in their own right. Our algorithms are simple, use simple data structures, and have the advantage over those of Raschle and Simon in that they are non-recursive, require linear space and admit efficient parallelization.

1 Introduction

Let $G = (V, E)$ be a simple non-trivial undirected graph. An *orientation* of the graph G is an antisymmetric directed graph obtained from G by assigning a direction to each edge of G . An orientation (V, F) of G is called *transitive* if it satisfies the following condition: if abc is a chordless path on 3 vertices in G , then F contains the directed edges $\overrightarrow{ab}$ and $\overrightarrow{bc}$, or $\overleftarrow{ab}$ and $\overleftarrow{bc}$, where $\overrightarrow{uv}$ or $\overleftarrow{vu}$ denotes an edge directed from u to v [4]. An orientation of a graph G is called *P_4 -transitive* if the orientation of every chordless path on 4 vertices of G is transitive; an orientation of such a path $abcd$ is transitive if and only if the path's edges are oriented in one of the following two ways: $\overrightarrow{ab}$, $\overrightarrow{bc}$ and $\overrightarrow{cd}$, or $\overleftarrow{ab}$, $\overleftarrow{bc}$ and $\overleftarrow{cd}$. The term borrows from the fact that a chordless path on 4 vertices is denoted by P_4 .

A graph which admits an acyclic transitive orientation is called a *comparability graph* [3,4]; A graph is a *P_4 -comparability graph* if it admits an acyclic P_4 -transitive orientation [5,6]. In light of these definitions, every comparability graph is a P_4 -comparability graph. Moreover, there exist P_4 -comparability

graphs which are not comparability. The class of the P_4 -comparability graphs (along with the P_4 -indifference, the P_4 -simplicial and the Raspail graphs) was introduced by Hoàng and Reed [6].

Algorithms for many different problems (such as, recognition, coloring, maximum clique, maximum independent set, hamiltonian paths and cycles) on subclasses of perfectly orderable graphs are available in the literature. The comparability graphs in particular have been the focus of much research which culminated into efficient recognition and orientation algorithms [4,7,8,12]. On the other hand, the P_4 -comparability graphs have not received as much attention, despite the fact that the definitions of the comparability and the P_4 -comparability graphs rely on the same principles [1,2,5,6,11].

Our main objective is to study the recognition and acyclic P_4 -transitive orientation problems on the class of P_4 -comparability graphs. These problems have been addressed by Hoàng and Reed who described $O(n^4)$ and $O(n^5)$ -time algorithms respectively [5,6], where n is the number of vertices of G . Recently, newer results on these problems were provided by Raschle and Simon [11]. Their algorithms work along the same lines, but they focus on the P_4 -components of the graph. The time complexity of their algorithms for either problem is $O(n + m^2)$, where m is the number of edges of G , as it is dominated by the time to compute the P_4 -components of G . Raschle and Simon also described recognition and orientation algorithms for P_4 -indifference graphs [11]; their algorithms run within the same time complexity, i.e., $O(n + m^2)$. We note that Hoàng and Reed [5,6] also presented algorithms which solve the recognition problem for P_4 -indifference graphs in $O(n^6)$ time.

In this paper, we present different $O(n + m^2)$ -time recognition and acyclic P_4 -transitive orientation algorithms for P_4 -comparability graphs of n vertices and m edges. Our technique relies on the computation of the P_4 -components of the input graph and takes advantage of structural relationships of these components. Our algorithms are simple, use simple data structures, and have the advantage over those of Raschle and Simon in that they are non-recursive, require linear space and admit efficient parallelization [10].

2 Theoretical Framework

Let $abcd$ be a P_4 of a graph G . The vertices b and c are called *midpoints* and the vertices a and d *endpoints* of the P_4 $abcd$. The edge connecting the midpoints of a P_4 is called the *rib*; the other two edges (which are incident to the endpoints) are called the *wings*. For example, the edge bc is the rib and the edges ab and cd are the wings of the P_4 $abcd$. Two P_4 s are called *adjacent* if they have an edge in common. The transitive closure of the adjacency relation is an equivalence relation on the set of P_4 s of a graph G ; the subgraphs of G spanned by the edges of the P_4 s in the equivalence classes are the P_4 -components of G . With slight abuse of terminology, we consider that an edge which does not belong to any P_4 belongs to a P_4 -component by itself; such a component is called *trivial*. A P_4 -component which is not trivial is called *non-trivial*; clearly a non-

trivial P_4 -component contains at least one P_4 . If the set of midpoints and the set of endpoints of the P_4 s of a non-trivial P_4 -component $\mathcal{C}$ define a partition of the vertex set $V(\mathcal{C})$, then the P_4 -component $\mathcal{C}$ is called *separable*. We can show [9]:

Lemma 1. *Let $G = (V, E)$ be a graph and let $\mathcal{C}$ be a non-trivial P_4 -component of G . Then,*

- (i) *If ρ and ρ' are two P_4 s which both belong to $\mathcal{C}$, then there exists a sequence $\rho, \rho_1, \dots, \rho_k, \rho'$ of adjacent P_4 s in $\mathcal{C}$;*
- (ii) *$\mathcal{C}$ is connected;*

The definition of a P_4 -comparability graph requires that such a graph admit an acyclic P_4 -transitive orientation. However, Hoàng and Reed [6] showed that in order to determine whether a graph is P_4 -comparability one can restrict one's attention to the P_4 -components of the graph. In particular, what they proved ([6], Theorem 3.1) can be paraphrased in terms of the P_4 -components as follows:

Lemma 2. [6] *Let G be a graph such that each of its P_4 -components admits an acyclic P_4 -transitive orientation. Then G is a P_4 -comparability graph.*

Although determining that each of the P_4 -components of a graph admits an acyclic P_4 -transitive orientation suffices to establish that the graph is P_4 -comparability, the directed graph produced by placing the oriented P_4 -components together may contain cycles. However, an acyclic P_4 -transitive orientation of the entire graph can be obtained by inversion of the orientation of some of the P_4 -components. Therefore, if one wishes to compute an acyclic P_4 -transitive orientation of a P_4 -comparability graph, one needs to detect directed cycles (if they exist) formed by edges belonging to more than one P_4 -component and appropriately invert the orientation of one or more of these P_4 -components. Fortunately, one does not need to consider arbitrarily long cycles as shown in the following lemma [6].

Lemma 3. ([6], Lemma 3.5) *If a proper orientation of an interesting graph is cyclic, then it contains a directed triangle.¹*

Given a non-trivial P_4 -component $\mathcal{C}$ of a graph $G = (V, E)$, the set of vertices $V - V(\mathcal{C})$ can be partitioned into three sets:

- (i) R contains the vertices of $V - V(\mathcal{C})$ which are adjacent to some (but not all) of the vertices in $V(\mathcal{C})$,
- (ii) P contains the vertices of $V - V(\mathcal{C})$ which are adjacent to all the vertices in $V(\mathcal{C})$, and
- (iii) Q contains the vertices of $V - V(\mathcal{C})$ which are not adjacent to any of the vertices in $V(\mathcal{C})$.

The adjacency relation is considered in terms of the given graph G .

¹ An orientation is proper if the orientation of every P_4 is transitive. A graph is interesting if the orientation of every P_4 -component is acyclic.

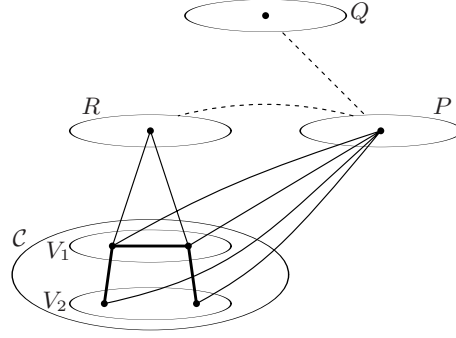


Fig. 1. Partition of the vertex set with respect to a separable P_4 -component $\mathcal{C}$

In [11], Raschle and Simon showed that, given a non-trivial P_4 -component $\mathcal{C}$ and a vertex $v \notin V(\mathcal{C})$, if v is adjacent to the midpoints of a P_4 of $\mathcal{C}$ and is not adjacent to its endpoints, then v does so with respect to every P_4 in $\mathcal{C}$ (that is, v is adjacent to the midpoints and not adjacent to the endpoints of every P_4 in $\mathcal{C}$). This implies that any vertex of G , which does not belong to $\mathcal{C}$ and is adjacent to at least one but not all the vertices in $V(\mathcal{C})$, is adjacent to the midpoints of all the P_4 s in $\mathcal{C}$. Based on that, Raschle and Simon showed that:

Lemma 4. ([11], Corollary 3.3) *Let $\mathcal{C}$ be a non-trivial P_4 -component and $R \neq \emptyset$. Then, $\mathcal{C}$ is separable and every vertex in R is V_1 -universal and V_2 -null². Moreover, no edge between R and Q exists.*

The set V_1 is the set of the midpoints of all the P_4 s in $\mathcal{C}$, whereas the set V_2 is the set of endpoints. Figure 1 shows the partition of the vertices of a graph with respect to a separable P_4 -component $\mathcal{C}$; the dashed segments between R and P and P and Q indicate that there may be edges between pairs of vertices in the corresponding sets. Then, a P_4 with at least one but not all its vertices in $V(\mathcal{C})$ must be a P_4 of one of the following types:

type (1)	vpq_1q_2	where $v \in V(\mathcal{C})$, $p \in P$, $q_1, q_2 \in Q$
type (2)	p_1vp_2q	where $p_1 \in P$, $v \in V(\mathcal{C})$, $p_2 \in P$, $q \in Q$
type (3)	$p_1v_2p_2r$	where $p_1 \in P$, $v_2 \in V_2$, $p_2 \in P$, $r \in R$
type (4)	$v_2pr_1r_2$	where $v_2 \in V_2$, $p \in P$, $r_1, r_2 \in R$
type (5)	rv_1pq	where $r \in R$, $v_1 \in V_1$, $p \in P$, $q \in Q$
type (6)	rv_1pv_2	where $r \in R$, $v_1 \in V_1$, $p \in P$, $v_2 \in V_2$
type (7)	$rv_1v_2v'_2$	where $r \in R$, $v_1 \in V_1$, $v_2, v'_2 \in V_2$
type (8)	$v'_1rv_1v_2$	where $r \in R$, $v_1, v'_1 \in V_1$, $v_2 \in V_2$

Raschle and Simon proved that neither a $P_3 abc$ with $a \in V_1$ and $b, c \in V_2$ nor a $\overline{P}_3 abc$ with $a, b \in V_1$ and $c \in V_2$ exists ([11], Lemma 3.4), which implies that:

² For a set A of vertices, we say that a vertex v is A -universal if v is adjacent to every element of A ; a vertex v is A -null if v is adjacent to no element of A .

Lemma 5. *Let $\mathcal{C}$ be a non-trivial P_4 -component of a graph $G = (V, E)$. Then, no P_4 s of type (7) or (8) with respect to $\mathcal{C}$ exist.*

Additionally, Raschle and Simon proved the following interesting result regarding the P_4 -components.

Lemma 6. ([11], Theorem 3.6) *Two different P_4 -components have different vertex sets.*

Moreover, we can show the following [9]:

Lemma 7. *Let $\mathcal{A}$ and $\mathcal{B}$ be two non-trivial P_4 -components of the graph G . If the component $\mathcal{A}$ contains an edge e both endpoints of which belong to the vertex set $V(\mathcal{B})$ of $\mathcal{B}$, then $V(\mathcal{A}) \subseteq V(\mathcal{B})$.*

Let us consider a non-trivial P_4 -component $\mathcal{C}$ of the graph G such that $V(\mathcal{C}) \subset V$, and let $S_{\mathcal{C}}$ be the set of non-trivial P_4 -components of G which have a vertex in $V(\mathcal{C})$ and a vertex in $V - V(\mathcal{C})$. Then, each component in $S_{\mathcal{C}}$ contains a P_4 of type (1)-(8), and thus, by taking Lemma 5 into account, we can partition the elements of $S_{\mathcal{C}}$ into two sets as follows:

- P_4 -components of type A: the P_4 components, each of which contains at least one P_4 of type (1)-(5) with respect to $\mathcal{C}$;
- P_4 -components of type B: the P_4 -components which contain only P_4 s of type (6) with respect to $\mathcal{C}$.

The following lemmata establish properties of P_4 -components of type A and of type B (the proofs are omitted due to lack of space but can be found in [9]).

Lemma 8. *Let $\mathcal{C}$ be a non-trivial P_4 -component of a P_4 -comparability graph $G = (V, E)$ and suppose that the vertices in $V - V(\mathcal{C})$ have been partitioned into sets R , P , and Q as described earlier in this section. Then, if there exists an edge xv (where $x \in R \cup P$ and $v \in V(\mathcal{C})$) that belongs to a P_4 -component $\mathcal{A}$ of type A, then all the edges, which connect the vertex x to a vertex in $V(\mathcal{C})$, belong to $\mathcal{A}$. Moreover, these edges are all oriented towards x or they are all oriented away from x .*

Lemma 9. *Let $\mathcal{B}$ and $\mathcal{C}$ be two non-trivial P_4 -components of the graph G such that $\mathcal{B}$ is of type B with respect to $\mathcal{C}$. Then,*

- (i) *every edge of $\mathcal{B}$ has exactly one endpoint in $V(\mathcal{C})$;*
- (ii) *if an edge of $\mathcal{B}$ is oriented towards its endpoint that belongs to $V(\mathcal{C})$, then so do all the edges of $\mathcal{B}$;*
- (iii) *the edges of $\mathcal{B}$ incident upon the same vertex v are all oriented either towards v or away from it.*

Lemma 10. *Let $\mathcal{B}$ and $\mathcal{C}$ be two non-trivial P_4 -components of the graph G such that $|V(\mathcal{B})| \geq |V(\mathcal{C})|$ and let $\beta = \sum_{v \in V(\mathcal{C})} d_{\mathcal{B}}(v)$, where $d_{\mathcal{B}}(v)$ denotes the number of edges of $\mathcal{B}$ which are incident upon v . Then, $\mathcal{B}$ is of type B with respect to $\mathcal{C}$ if and only if $\beta = |E(\mathcal{B})|$.*

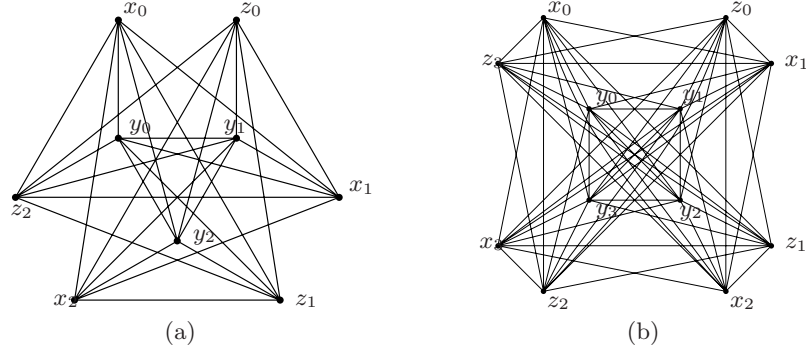


Fig. 2. Graphs that have P_4 -components with cyclic P_4 -transitive orientation

Lemma 11. *Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be three distinct separable P_4 -components such that $\mathcal{A}$ is of type B with respect to $\mathcal{B}$, $\mathcal{B}$ is of type B with respect to $\mathcal{C}$, and $|V(\mathcal{A})| \geq |V(\mathcal{C})|$. Then, if there exists a vertex which is a midpoint of all three components $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, the P_4 -component $\mathcal{A}$ is of type B with respect to $\mathcal{C}$.*

We close this section by showing that the assignment of compatible directions in all the P_4 s of a P_4 -component does not imply that the component is necessarily acyclic. Let k be an integer at least equal to 3, and let $X_k = \{x_i \mid 0 \leq i < k\}$, $Y_k = \{y_i \mid 0 \leq i < k\}$, and $Z_k = \{z_i \mid 0 \leq i < k\}$ be three sets of distinct vertices. We consider the graph $G_k = (V_k, E_k)$ where

$$V_k = X_k \cup Y_k \cup Z_k$$

$$\text{and } E_k = V_k \times V_k - \left(\{x_i y_{i+1} \mid 0 \leq i < k\} \cup \{x_i z_i \mid 0 \leq i < k\} \cup \{y_i z_i \mid 0 \leq i < k\} \right).$$

(The addition in the subscripts is assumed to be done mod k .) Figures 2(a) and 2(b) depict G_3 and G_4 respectively. The graph G_k has the following properties:

- ▷ the only P_4 s of G_k are the paths $x_i y_i y_{i+1} z_i$, $y_{i+1} z_i z_{i+1} x_i$, and $y_{i+1} x_{i+1} x_i z_{i+1}$ for $0 \leq i < k$;
- ▷ G_k has a single non-trivial P_4 -component;
- ▷ the directed edges $y_i y_{i+1}$ ($0 \leq i < k$) form a directed cycle of length k in the non-trivial P_4 -component of G_k ;
- ▷ no directed cycle of length less than k exists in the non-trivial P_4 -component of G_k .

3 Recognition of P_4 -Comparability Graphs

The main idea of the algorithm is to build the P_4 -components of the given graph G by considering all the P_4 s of G ; this is achieved by unioning in a single P_4 -component the P_4 -components of the edges of each such path, while it is

made sure that the edges are compatibly oriented. It is important to note that the orientation of two edges in the same P_4 -component is not free to change relative to each other; either the orientation of all the edges in the component stays the same or it is inverted for all the edges. If no compatible orientation can be found or if the resulting P_4 -components contain directed cycles, then the graph is not a P_4 -comparability graph. The P_4 s are produced by means of BFS-traversals of the graph G starting from each of G 's vertices.

The algorithm is described in more detail below. Initially, each edge of G belongs to a P_4 -component by itself.

Recognition Algorithm.

1. For each vertex v of the graph, we construct the BFS-tree T_v rooted at v and we update the level $level(x)$ ³ and the parent p_x of each vertex x in T_v ; before the construction of each of the BFS-trees, $level(x) = -1$ for each vertex x of the graph. Then, we process the edges of the graph as follows:
 - (i) for each edge $e = uw$ where $level(u) = 1$ and $level(w) = 2$, we check whether there exist edges from w to a vertex in the 3rd level of T_v . If not, then we do nothing. Otherwise, we orient the edges vu , uw , vp_w , and $p_w w$ in a compatible fashion; for example, we orient vu and vp_w away from v , and uw and $p_w w$ away from w (note that if $u = p_w$, we end up processing the edges vu and uw only). If any two of these edges belong to the same P_4 -component and have incompatible orientations, then we conclude that the graph G is not a P_4 -comparability graph. If any two of these edges belong to different P_4 -components, then we union these components into a single component; if the edges do not have compatible orientations, then we invert (during the unioning) the orientation of all the edges of one of the unioned P_4 -components.
 - (ii) for each edge $e = uw$ where $level(u) = i$ and $level(w) = i + 1$ for $i \geq 2$, we consider the edges $p_u u$ and uw . As in the previous case, if the two edges belong to the same P_4 -component and they are not both oriented towards u or away from u , then there is no compatible orientation assignment and the graph is not a P_4 -comparability graph. If the two edges belong to different P_4 -components, we union the corresponding P_4 -components in a single component, while making sure that the edges are oriented in a compatible fashion.
 - (iii) for each edge $e = uw$ where $level(u) = level(w) = 2$, we go through all the vertices of the 1st level of T_v . For each such vertex x , we check whether x is adjacent to u or w . If x is adjacent to u but not to w , then the edges vx , xu , and uw form a P_4 ; we therefore union the corresponding P_4 -components while orienting their edges compatibly. We work similarly for the case where x is adjacent to w but not to u , since the edges vx , xw , and wu form a P_4 .
2. After all the vertices have been processed, we check whether the resulting non-trivial P_4 -components contain directed cycles. This is done by applying topological sorting independently in each of the P_4 components; if the

³ The level of the root of a tree is equal to 0.

topological sorting succeeds then the corresponding component is acyclic, otherwise there is a directed cycle. If any of the P_4 -components contains a cycle, then the graph is not a P_4 -comparability graph.

For each P_4 -component, we maintain a linked list of the records of the edges in the component, and the total number of these edges. Each edge record contains a pointer to the header record of the component to which the edge belongs; in this way, we can determine in constant time the component to which an edge belongs and the component's size. Unioning two P_4 -components is done by updating the edge records of the smallest component and by linking them to the edge list of the largest one, which implies that the union operation takes time linear in the size of the smallest component. As mentioned above, in the process of unioning, we may have to invert the orientation in the edge records that we link, if the current orientations are not compatible.

The correctness of the algorithm follows from the fact that all the P_4 s of the given graph are taken into account (see [9], Lemma 3.1), from the correct orientation assignment on the edges of these paths, and from Lemma 2 in conjunction with Step 2 of the algorithm.

Time and Space Complexity. Computing the BFS-tree T_v of the vertex v of G takes $O(1+m(v)) = O(1+m)$ time, where $m(v)$ is the number of edges in the connected component of G to which v belongs. Processing the tree T_v includes processing the edges and checking for compatible orientation, and unioning P_4 -components. If we ignore P_4 -component unioning, then, each of the Steps 1(i) and 1(ii) takes constant time per processed edge; the parent of a vertex in the tree can be determined in constant time with the use of an auxiliary array, and the P_4 -component of an edge is determined in constant time by means of the pointer to the component head record (these pointers are updated during unioning). The Step 1(iii) of the algorithm takes time $O(\deg(v))$ for each edge in the 2nd level of the tree, where by $\deg(v)$ we denote the degree of the vertex v ; this implies a total of $O(m \deg(v))$ time for the Step 1(iii) for the tree T_v . Now, the time required for all the P_4 -component union operations during the processing of all the BFS-trees is $O(m \log m)$; there cannot be more than $m - 1$ such operations (we start with m P_4 -components and we may end up with only one), and each one of them takes time linear in the size of the smallest of the two components that are unioned. Finally, checking whether a non-trivial P_4 -component is acyclic takes $O(1 + m_i)$, where m_i is the number of edges of the component. Thus, the total time taken by Step 2 is $O(\sum_i (1 + m_i)) = O(m)$, since there are at most m P_4 -components and $\sum_i m_i = m$. Thus, the overall time complexity is $O\left(\sum_v (1 + m + m \deg(v)) + m \log m + m\right) = O(n + m^2)$, since $\sum_v \deg(v) = 2m$.

The space complexity is linear in the size of the graph G ; the information stored in order to help processing each BFS-tree is constant per vertex, and the handling of the P_4 -components requires one record per edge and one record per component. Thus, the space required is $O(n + m)$.

Theorem 1. *It can be decided whether a simple graph on n vertices and m edges is a P_4 -comparability graph in $O(n + m^2)$ time and $O(n + m)$ space.*

Remark. It must be noted that there are simpler ways of producing the P_4 s of a graph in $O(n + m^2)$ time. However, such approaches require $\Theta(n^2)$ space. For example, Raschle and Simon note that a P_4 is uniquely determined by its wings [11]; this implies that the P_4 s can be determined by considering all $\Theta(m^2)$ pairs of edges and by checking if the edges in each such pair are the wings of a P_4 . In order not to exceed the $O(m^2)$ time complexity, the information on whether two vertices are adjacent should be available in constant time, something that necessitates a $\Theta(n^2)$ -space adjacency matrix.

4 Acyclic P_4 -Transitive Orientation

Although each of the P_4 -components of the given graph G which is produced by the recognition algorithm is acyclic, directed cycles may arise when all the P_4 -components are placed together; obviously, these cycles will include edges from more than one P_4 -component. Appropriate inversion of the orientation of some of the components will yield the desired acyclic P_4 -transitive orientation.

Our algorithm to compute the acyclic P_4 -transitive orientation of a P_4 -comparability graph relies on the processing of the P_4 -components of the given graph G and focuses on edges incident upon the vertices of the non-trivial P_4 -component which is currently being processed. It assigns orientations in a greedy fashion, and avoids both the contraction step and the recursive call of the orientation algorithms of Hoàng and Reed [6], and Raschle and Simon [11]. More specifically, the algorithm works as follows:

Orientation Algorithm.

1. We apply the recognition algorithm of the previous section on the given graph G , which produces the P_4 -components of G and an acyclic P_4 -transitive orientation of each component.
2. We sort the non-trivial P_4 -components of G by increasing number of vertices; let $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_h$ be the resulting ordered sequence. We associate with each $\mathcal{C}_i$ a **mark** and a **counter** field which are initialized to 0.
3. For each P_4 -component $\mathcal{C}_i$ ($1 \leq i < h$) in order, we do:
 By going through the vertices in $V(\mathcal{C}_i)$, we collect the edges which are incident upon a vertex in $V(\mathcal{C}_i)$ and belong to a P_4 -component $\mathcal{C}_j$ where $j > i$. Then, for each such edge e , we increment the **counter** field associated with the P_4 -component to which e belongs. Next, we go through the collected edges once more. This time, for such an edge e , we check whether the P_4 -component to which e belongs has its **mark** field equal to 0 and its **counter** field equal to the total number of edges of the component; if yes, then we set the **mark** field of the component to 1, and, in case e is not oriented towards its endpoint in $V(\mathcal{C}_i)$, we flip the component's orientation (by updating a corresponding boolean variable). After that, we set the **counter** field of the component to which e belongs to 0; in this way, the **counter** fields of all the non-trivial P_4 -components are equal to 0 every time a P_4 -component starts getting processed in Step 3.

4. We orient the edges which belong to the trivial P_4 -components: this can be easily done by topologically sorting the vertices of G using only the oriented edges of the non-trivial components, and orienting the remaining edges in accordance with the topological order of their incident vertices.

Note that in Step 3 we process all the non-trivial P_4 -components of the given graph G except for the largest one. This implies that the vertex set $V(\mathcal{C}_i)$ of each P_4 -component $\mathcal{C}_i$ ($1 \leq i < h$) that we process is a proper subset of the vertex set V of G ; if $V(\mathcal{C}_i) = V$, then $V(\mathcal{C}_h) = V$ as well, which implies that $\mathcal{C}_i = \mathcal{C}_h$ (Lemma 6), a contradiction. Thus, the discussion in Section 2 regarding the P_4 -components of type A and type B applies to each such $\mathcal{C}_i$. Moreover, according to Lemma 10, the P_4 -components whose **mark** field is set to 1 in Step 3 are components which are of type B with respect to the currently processed component $\mathcal{C}_i$. Each edge of these components has exactly one endpoint in $V(\mathcal{C}_i)$ (see Lemma 9, statement (i)), so that it is valid to try to orient such an edge towards that endpoint. Furthermore, Lemma 9 (statement (ii)) implies that if such an edge gets oriented towards its endpoint which belongs to $V(\mathcal{C}_i)$, then so do all the edges of the same P_4 -component. In the case that the set R in the partition of the vertices in $V - V(\mathcal{C}_i)$ (as described in Section 2) is empty, there are no P_4 -components of type B with respect to $\mathcal{C}_i$. While processing $\mathcal{C}_i$, our algorithm updates the **counter** fields of the components that contain an edge incident upon a vertex in $V(\mathcal{C}_i)$, finds that none of these components ends up having its **counter** field equal to the number of its edges, and thus does nothing further.

The orientation algorithm does not compute the sets R , P , and Q with respect to the currently processed P_4 -component $\mathcal{C}_i$. These sets can be computed in $O(n)$ time for each $\mathcal{C}_i$ as follows. We use an array with one entry per vertex of the graph G ; we initialize the array entries corresponding to vertices in $V(\mathcal{C}_i)$ to 0 and all the remaining ones to -1. Let v_1 and v_2 be an arbitrary midpoint and an arbitrary endpoint of a P_4 in $\mathcal{C}_i$. We go through the vertices adjacent to v_1 and if the vertex does not belong to $V(\mathcal{C}_i)$, we set the corresponding entry to 1. Next, we go through the vertices adjacent to v_2 ; this time, if the vertex does not belong to $V(\mathcal{C}_i)$, we increment the corresponding entry. Then, the vertices in $\mathcal{C}_i$, R , and Q are the vertices whose corresponding array entries are equal to 0, 1, and -1 respectively, while the remaining vertices belong to P and their corresponding entries are larger than 1; recall that every vertex in $V - V(\mathcal{C}_i)$ which is adjacent to an endpoint of a P_4 of $\mathcal{C}_i$ is also adjacent to any midpoint.

Correctness of the Algorithm. The acyclicity of the directed graph produced by our orientation algorithm relies on the following two lemmata (proofs can be found in [9]).

Lemma 12. *Let $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_h$ be the sequence of the non-trivial P_4 -components of the given graph G ordered by increasing vertex number. Consider the set $S_i = \{\mathcal{C}_j \mid j < i \text{ and } \mathcal{C}_i \text{ is of type B with respect to } \mathcal{C}_j\}$ and suppose that $S_i \neq \emptyset$. If $\hat{i} = \min\{j \mid \mathcal{C}_j \in S_i\}$, then our algorithm orients the edges of the component $\mathcal{C}_i$ towards their endpoint which belongs to $V(\mathcal{C}_{\hat{i}})$.*

Lemma 13. *Let $\mathcal{C}_1, \mathcal{C}_2, \dots, \mathcal{C}_h$ be the non-trivial P_4 -components of a graph G ordered by increasing vertex number and suppose that each component has received an acyclic P_4 -transitive orientation. Consider the set $S_i = \{\mathcal{C}_j \mid j < i \text{ and } \mathcal{C}_i \text{ is of type B with respect to } \mathcal{C}_j\}$. If the edges of each P_4 -component $\mathcal{C}_i$ such that $S_i \neq \emptyset$ get oriented towards their endpoint which belongs to $V(\mathcal{C}_i)$, where $\hat{i} = \min\{j \mid \mathcal{C}_j \in S_i\}$, then the resulting directed subgraph of G spanned by the edges of the $\mathcal{C}_i$ s ($1 \leq i \leq h$) does not contain a directed cycle.*

Theorem 2. *When applied to a P_4 -comparability graph, our orientation algorithm produces an acyclic P_4 -transitive orientation.*

Proof: The application of the recognition algorithm in Step 1 of the orientation algorithm and the fact that thereafter the inversion of the orientation of an edge causes the inversion of the orientation of all the edges in the same P_4 -component imply that the resulting orientation is P_4 -transitive. The proof of the theorem will be complete if we show that it is also acyclic. Since the edges of the trivial P_4 -components do not introduce cycles given that they are oriented according to a topological sorting of the vertices of the graph, it suffices to show that the directed subgraph of G spanned by the edges of the non-trivial P_4 -components of G , which results after the last execution of Step 3, is acyclic. This follows directly from Lemmata 12 and 13. $\square$

Time and Space Complexity. As described in the previous section, Step 1 of the algorithm can be completed in $O(n + m^2)$ time. Step 2 takes $O(m \log m)$ time, since there are $O(m)$ non-trivial P_4 -components. Since the degree of a vertex of the graph does not exceed $n - 1$, the total number of edges processed while processing the P_4 -component $\mathcal{C}_i$ in Step 3 is $O(n |V(\mathcal{C}_i)|)$, where $|V(\mathcal{C}_i)|$ is the cardinality of the vertex set of $\mathcal{C}_i$. This upper bound is $O(n (|E(\mathcal{C}_i)| + 1)) = O(n |E(\mathcal{C}_i)|)$, because the component $\mathcal{C}_i$ is connected (Lemma 1, statement (ii)) and hence $|V(\mathcal{C}_i)| \leq |E(\mathcal{C}_i)| + 1$. The time to process each such edge is $O(1)$, thus implying a total of $O(n |E(\mathcal{C}_i)|)$ time for the execution of Step 3 for the component $\mathcal{C}_i$; since an edge of the graph belongs to one P_4 -component and a component is processed only once, the overall time for all the executions of Step 3 is $O(nm)$. Finally, Step 4 takes $O(n + m)$ time.

Summarizing, the time complexity of the orientation algorithm is $O(n + m^2)$. It is interesting to note that the time complexity is dominated by the time to execute Step 1; the remaining steps take a total of $O(nm)$ time. Therefore, an $o(n + m^2)$ -time algorithm to recognize a P_4 -comparability graph and to compute its P_4 -components will imply an $o(n + m^2)$ -time algorithm for the acyclic P_4 -transitive orientation of a P_4 -comparability graph. The space complexity is linear in the size of the given graph G .

From the above discussion, we obtain the following theorem.

Theorem 3. *Let G be a P_4 -comparability graph on n vertices and m edges. Then, an acyclic P_4 -transitive orientation of G can be computed in $O(n + m^2)$ time and $O(n + m)$ space.*

5 Concluding Remarks

In this paper, we presented an $O(n + m^2)$ -time and linear space algorithm to recognize whether a graph of n vertices and m edges is a P_4 -comparability graph. We also described an algorithm to compute an acyclic P_4 -transitive orientation of a P_4 -comparability graph which runs in $O(n + m^2)$ time and linear space as well. Both algorithms exhibit the currently best time and space complexities to the best of our knowledge, are simple enough to be easily used in practice, are non-recursive, and admit efficient parallelization.

The obvious open question is whether the P_4 -comparability graphs can be recognized and oriented in $o(n + m^2)$ time. Note that a better time complexity for the recognition problem — assuming that the recognition process determines the P_4 -components as well — will imply a better time complexity for our orientation algorithm.

References

1. L. Babel and S. Olariu, A new characterization of P_4 -connected graphs, *Proc. 22nd International Workshop on Graph-theoretic concepts in Computer Science (WG'96)* (F. d'Amore, P. G. Franciosa, and A. Marchetti-Spaccamela, eds.), LNCS 1197, 1996, 17–30. [321](#)
2. C. M. H. de Figueiredo, J. Gimbel, C. P. Mello, and J. L. Szwarcfiter, Even and odd pairs in comparability and in P_4 -comparability graphs, *Discrete Appl. Math.* **91** (1999), 293–297. [321](#)
3. P. C. Gilmore and A. J. Hoffman, A characterization of comparability graphs and of interval graphs, *Canad. J. Math.* **16** (1964), 539–548. [320](#)
4. M. C. Golumbic, *Algorithmic graph theory and perfect graphs*, Academic Press, Inc., New York, 1980. [320](#), [321](#)
5. C. T. Hoàng and B. A. Reed, Some classes of perfectly orderable graphs, *J. Graph Theory* **13** (1989), 445–463. [320](#), [321](#)
6. C. T. Hoàng and B. A. Reed, P_4 -comparability graphs, *Discrete Math.* **74** (1989), 173–200. [320](#), [321](#), [322](#), [328](#)
7. R. M. McConnell and J. Spinrad, Linear-time modular decomposition and efficient transitive orientation of comparability graphs, *Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms* (1994), 536–545. [321](#)
8. R. M. McConnell and J. Spinrad, Linear-time transitive orientation, *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms* (1997), 19–25. [321](#)
9. S. D. Nikolopoulos and L. Palios, Recognition and orientation algorithms for P_4 -comparability graphs, *Technical Report 23-00*, 2000, University of Ioannina, Ioannina, Greece. [322](#), [324](#), [327](#), [329](#)
10. S. D. Nikolopoulos and L. Palios, Parallel algorithms for P_4 -comparability graphs, *Technical Report 20-01*, 2001, University of Ioannina, Ioannina, Greece. [321](#)
11. T. Raschle and K. Simon, On the P_4 -components of graphs, *Discrete Appl. Math.* **100** (2000), 215–235. [321](#), [323](#), [324](#), [328](#)
12. J. P. Spinrad, On comparability and permutation graphs, *SIAM J. on Comput.* **14** (1985), 658–670. [321](#)

Efficient Algorithms for k -Terminal Cuts on Planar Graphs^{*}

Danny Z. Chen and Xiaodong Wu^{**}

Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, IN 46556, USA
{chen,xwu}@cse.nd.edu

Abstract. The minimum k -terminal cut problem is of considerable theoretical interest and arises in several applied areas such as parallel and distributed computing, VLSI circuit design, and networking. In this paper, we present two new approximation and exact algorithms for this problem on an n -vertex undirected weighted planar graph G . For the case when the k terminals are covered by the boundaries of $m > 1$ faces of G , we give a $\min\{O(n^2 \log n \log m), O(m^2 n^{1.5} \log^2 n + kn)\}$ time algorithm with a $(2 - \frac{2}{k})$ -approximation ratio (clearly, $m \leq k$). For the case when all k terminals are covered by the boundary of one face of G , we give an $O(nk^3 + (n \log n)k^2)$ time exact algorithm, or a linear time exact algorithm if $k = 3$, for computing an optimal k -terminal cut. Our algorithms are based on interesting observations and improve the previous algorithms when they are applied to planar graphs.

1 Introduction

Given an n -vertex undirected graph $G = (V, E)$ with non-negative edge weights and a k -vertex subset $T \subset V$ (called the *terminals*), the *minimum k -terminal cut problem* seeks to identify an edge set $C \subset E$ such that there is no path connecting any two distinct terminals of T in the graph $G'' = (V, E - C)$ and the total weight of edges in C is minimized. This problem is a natural generalization of the well-known undirected s - t cut problem, which has been extensively studied since Ford and Fulkerson's work [11]. The minimum k -terminal cut problem arises in various applied areas, such as parallel and distributed computing, VLSI circuit design, and networking [10].

Dahlhaus *et al.* [9] pioneered the k -terminal cut study. They proved [9,10] that, for any fixed $k \geq 3$, the minimum k -terminal cut problem is MAX SNP-hard even if all edge weights are 1, and gave a simple combinatorial *isolation heuristic* to achieve an approximation ratio of $(2 - \frac{2}{k})$. Călinescu and Karloff [6] used a novel geometric relaxation of k -terminal cuts to obtain a $(1.5 - \frac{1}{k})$ -approximation algorithm. This relaxation uses the k -simplex $\mathfrak{S} = \{x \in \mathbb{R}^k :$

^{*} This research was supported in part by the National Science Foundation under Grants CCR-9623585 and CCR-9988468.

^{**} Corresponding author.

$x \geq 0, \sum_i^k x_i = 1\}$, and maps the nodes of the graph to points in $\mathfrak{S}$ such that terminal i is mapped to the i -th vertex of $\mathfrak{S}$ and each edge is mapped to the straight line connecting its endpoints in $\mathfrak{S}$. Based on this embedding, Călinescu and Karloff gave a new linear programming relaxation for k -terminal cuts. Karger *et al.* [18] further explored the geometric relaxation in [6] to obtain an (analytic) bound of $1.3438 - \epsilon_k$, which is less than $(1.5 - \frac{1}{k})$ for all k ; they also gave an optimal rounding algorithm for $k = 3$ with a $\frac{12}{11}$ -approximation ratio. Chopra and Rao [7] and Cunningham [8] developed a polyhedral approach for the minimum k -terminal cut problem. Bertsimas, Teo, and Vohra [4] proposed a nonlinear formulation of the k -terminal cut and related problems, and gave a simple randomized rounding argument yielding the same approximation bounds as [10].

The minimum k -terminal cut problem on planar graphs also attracts much attention due to its considerable practical significance, especially in VLSI circuit partitioning. Dahlhaus *et al.* [10] proved that, if k is not fixed, the minimum k -terminal cut problem on planar graphs is NP-hard even if all edge weights are 1; for any fixed $k \geq 3$, they also gave an exact algorithm for the problem, in $O((4k)^k n^{2k-1} \log n)$ time. Hartvigsen [16] established a close connection between the planar k -terminal cut problem and the well-known Gomory-Hu cut collections, and developed an $O(k 4^k n^{2k-4} \log n)$ time exact algorithm; he also established a connection between the planar k -terminal cut and a particular matroid, resulting in another polynomial time algorithm for fixed k . For trees and 2-trees, there are linear time exact algorithms [7]. For dense unweighted graphs, there is a polynomial time approximation scheme [2, 13].

In this paper, we consider the minimum k -terminal cut problem on an n -vertex planar graph $G = (V, E)$ with the terminals being covered by m faces of G . Our main results are as follows.

- For the case when the k terminals are covered by the boundaries of $m > 1$ faces of G (clearly, $m \leq k$), we present a $\min\{O(n^2 \log n \log m), O(m^2 n^{1.5} \log^2 n + kn)\}$ time algorithm that achieves a $(2 - \frac{2}{k})$ -approximation ratio based on a non-trivial divide-and-conquer strategy.
- For the case when all terminals are covered by one face of G , we explore the connection between the k -terminal cut and the minimum Steiner tree [14] to obtain an $O(nk^3 + (n \log n)k^2)$ time exact algorithm for computing an optimal k -terminal cut for any k . Further, we give a linear time exact algorithm to construct an optimal 3-terminal cut for this case.

To our best knowledge, no previously known approximation algorithm for minimum k -terminal cuts explores the planarity properties in the way we do. For example, unlike [10] (in which a minimum weight *isolating cut* is built for each terminal based on a minimum s - t cut), we efficiently construct a minimum weight *island cut* (to be defined in Section 3) only for each of the m faces involved, in a divide-and-conquer fashion. These island cuts enable us to use shortest paths (instead of minimum s - t cuts) to compute optimal isolating cuts for most terminals. Comparing with the $O(kn^2 \log n)$ time $(2 - \frac{2}{k})$ -approximation algorithms

in [10] (when they are applied to planar graphs), our approximation algorithm runs faster by at least an $O(\frac{k}{\log m})$ factor; in particular, when $m = O(\frac{n^{1/4}}{\sqrt{\log n}})$, our algorithm gives at least an $O(k)$ time improvement over [10]. Note that Călinescu and Karloff [6] and Karger *et al.* [18] have given randomized algorithms for general graphs with better approximation ratios, but these algorithms need to solve linear programming problems with a very large size of variables and constraints, which may restrict their applicability in practice.

Our algorithms actually motivate a study of the *minimum face cover* problem: Given a planar graph with a set of terminals, find a planar embedding of the graph such that all terminals are covered by as few faces as possible. Bienstock and Monma [5] have shown that this problem is NP-complete. Frederickson [12] presented a 2-approximation algorithm in $O(n)$ time if an embedding of the input planar graph is already given; if the embedding is not given, he gave a linear time 4-approximation algorithm for computing the minimum covering faces for the case when all vertices are terminals [12].

Throughout this paper, we assume that the input planar graphs that we are concerned with are already embedded in a plane without any edge crossing, and our algorithms will be dealing with such given graph embeddings. Also, without loss of generality (WLOG), we assume that the input graphs are connected (otherwise, we simply consider each connected component of the graph separately).

We omit the proofs of the lemmas due to the space limit.

2 Terminals on One Face

Let $G = (V, E)$ be an n -vertex undirected planar graph with non-negative edge weights, and $T \subset V$ be the set of k specified terminals for G . In this section, we assume that there is an (embedded) face of G such that all k terminals of T are covered by the boundary of that face. WLOG, let this face be the infinite face of G , denoted by P (other cases can be easily reduced to this case). Subsection 2.1 deals with the basic case when G is a biconnected graph. Subsection 2.2 handles the more general case when G is not biconnected, by reducing it to the biconnected case. Finally, we give a linear time algorithm for the case of $k = 3$.

2.1 When G Is Biconnected

In this subsection, we consider the basic case when the input planar graph G is biconnected [1]. WLOG, one can view the boundary $BD(P)$ of the infinite face P of G as a simple polygon. Let $I = (t_0, t_1, \dots, t_{k-1})$ be the ordered sequence of terminals in T that are visited by a counterclockwise traversal of the boundary $BD(P)$ of P starting at t_0 . We say that each pair (t_i, t_{i+1}) of consecutive terminals in I defines an *interval* I_i on $BD(P)$, where $i = 0, 1, \dots, k-1$ and $t_k = t_0$. As illustrated in Figure 1(a), the solid lined figure is the graph G , $t_0, t_1, \dots, t_4$ are the terminals lying on the boundary of the infinite face of G . The intervals are $(t_0, t_1), (t_1, t_2), (t_2, t_3), (t_3, t_4)$, and (t_4, t_0) .

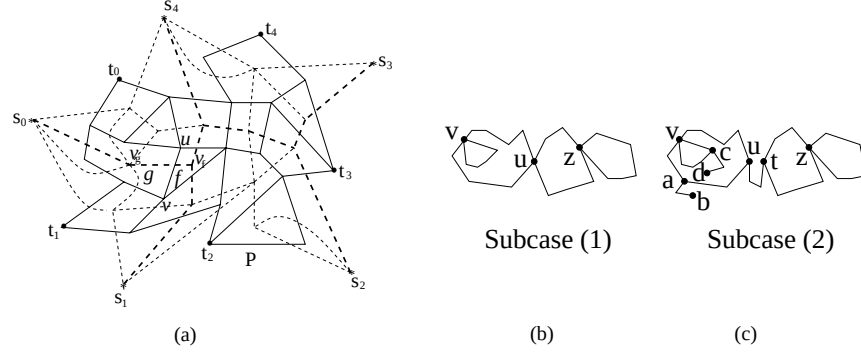


Fig. 1. (a) An augmented dual graph G' (denoted by dashed lines) of a planar graph G (denoted by solid lines). Bold dashed lines represent a Steiner tree that corresponds to a k -terminal cut. (b) and (c) Illustrating the two subcases of non-biconnected planar graphs

We build an augmented dual graph $G' = (V', E')$ of G as follows. As in the usual dual graph, each finite face f of G is associated with a vertex v_f of G' , and each edge (u, v) of G that bounds two finite faces f and g of G (i.e., $P \notin \{f, g\}$) corresponds to an edge $e_{uv} = (v_f, v_g)$ of G' , such that edge e_{uv} of G' has the same weight as edge (u, v) of G . Also, for each interval $I_i = (t_i, t_{i+1})$, we add a vertex s_i to G' , called an *augmented vertex*, such that s_i is located in the infinite face P of G in the plane of embedding. As traversing from t_i to t_{i+1} along $BD(P)$ counterclockwise, for each encountered edge e on P (bounding P and a finite face f of G), an edge (s_i, v_f) is added to G' , such that edge (s_i, v_f) of G' has the same weight as edge e of G . Note that it is easy to construct G' such that G' is a planar graph also embedded in the same plane as G . Figure 1(a) shows the augmented dual graph G' of G , which is the dashed line figure. Let S be the set of k augmented vertices $\{s_0, s_1, \dots, s_{k-1}\}$ in G' , which we treat as the terminals of G' .

Next, we review the *minimum Steiner tree* problem [14]. Given a graph $G'' = (V'', E'')$ with non-negative edge weights and a set S of specified vertices in V'' , a Steiner tree is a connected subgraph $R = (V_r, E_r)$ of G'' such that $S \subseteq V_r$ and $|E_r| = |V_r| - 1$. Actually, R is a tree. The problem is to find a Steiner tree R in G'' for S such that the sum of edge weights of R is minimized.

Our k -terminal cut algorithm for planar graphs with all terminals being covered by one face is based on the following key observations.

Lemma 1. *Every Steiner tree R of the augmented dual graph G' of G with respect to S corresponds to a k -terminal cut C of G for T .*

We say that a k -terminal cut C of G for T is *minimal* if putting any edge $e \in C$ back to G gives rise to a path in the resulted graph $G'' = (V, E - \{C - \{e\}\})$ that connects two distinct terminals of T . Note that there are many different minimal k -terminal cuts of G for T .

Lemma 2. *Every minimal k -terminal cut C of G for T corresponds to a Steiner tree R of the augmented dual graph G' of G for S .*

Thus, we have the following result.

Lemma 3. *A cut C of a biconnected planar G for T corresponding to a minimum Steiner tree of G' with respect to its terminals in S is a minimum k -terminal cut.*

2.2 When G Is Not Biconnected

In this subsection, we assume that the input planar graph G is connected but not biconnected. Our solution for this (general) case is a reduction to the basic case in Subsection 2.1 (i.e., the biconnected graph case). Actually, we consider two subcases of this general case: (1) every edge of G is on at least one cycle in G (see Figure 1(b) for an example), and (2) some edges of G are on no cycles in G (see Figure 1(c) for an example). We proceed by first reducing Subcase (1) to the biconnected graph case, and then reducing Subcase (2) to Subcase (1).

Note that in $O(n)$ time, one can identify all *biconnected components* of G [1] (two distinct edges of G belong to the same biconnected component if and only if there is a cycle in G containing both these edges). It is easy to decide from its biconnected components which subcase holds for G . Note that two distinct biconnected components of G can share at most one common vertex (called an *articulation point*). When we say that we *divide* G at an articulation point v , we separate G into two subgraphs respectively containing the two biconnected components that share v , by splitting v into two vertices v' and v'' , each belonging to one of the two subgraphs. The next two lemmas are useful for Subcase (1).

Lemma 4. *Suppose that Subcase (1) holds for G . If no articulation points of G are terminals of T , then the augmented dual graph G' of G is connected.*

Lemma 5. *Suppose that Subcase (1) holds for G . If an articulation point v of G is a terminal of T , then G can be divided into two subgraphs at v and the original k -terminal cut problem can be reduced to two independent terminal cut problems, one on each subgraph.*

Thus, Subcase (1) can be solved as follows. If Lemma 4 holds for G , then the problem on G can be solved simply as in Subsection 2.1. If Lemma 5 holds for G , then we divide G at every articulation point that is a terminal of T , and apply the result in Subsection 2.1 to each resulted subgraph of G .

We now consider Subcase (2). We call every edge of G that is a biconnected component by itself a *bridge edge*. In Figure 1(c), all edges on the paths from a to b , c to d , and u to t are bridge edges. We reduce Subcase (2) to Subcase (1) by transforming every bridge edge (u, v) with a weight w in G into a new biconnected component, as follows: Replace the one-edge path between u and v by two two-edge paths, such that each such two-edge path has an edge with weight $w/2$ and another with weight $+\infty$. Thus, we have the following result.

Lemma 6. *In $O(n)$ time, the k -terminal cut problem on an n -vertex non-bi-connected planar graph with all terminals lying on the boundary of a certain face can be reduced to a set of instances of the terminal cut problem on a biconnected planar graph.*

2.3 The Algorithms

The best known algorithm for computing a minimum Steiner tree in an embedded planar graph is due to Bern [3]. Given an n -vertex undirected edge-weighted embedded planar graph and a set of k terminals lying on the boundary of a certain face of the graph, the minimum Steiner tree problem can be solved in $O(nk^3 + (n \log n)k^2)$ time. Clearly, the augmented dual graph is also planar and its size is proportional to that of the original planar graph. Hence we have the following result.

Theorem 1. *A minimum k -terminal cut in an n -vertex planar graph with all k terminals lying on the boundary of a certain face can be computed in $O(nk^3 + (n \log n)k^2)$ time.*

Note that the minimum Steiner tree problem on a planar graph with $k = 3$ terminals all lying on the boundary of a certain face of the graph can be solved in linear time by using the single-source shortest path algorithm in [17]. This implies the following result.

Lemma 7. *The minimum 3-terminal cut problem on a planar graph can be solved in linear time if all three terminals lie on the boundary of a certain face of the graph.*

3 Terminals on $m > 1$ Faces

In this section, we consider the more difficult case of the problem when the terminals are covered by the boundaries of $m > 1$ faces of a planar graph (clearly, $m \leq k$). Given an n -vertex undirected planar graph $G = (V, E)$ with non-negative edge weights and a set T of k terminals lying on the boundaries of m faces of G , $\mathcal{F} = \{f_1, f_2, \dots, f_m\}$, we present a $(2 - \frac{2}{k})$ -approximation algorithm with a running time of $\min\{O(n^2 \log n \log m), O(m^2 n^{1.5} \log^2 n + kn)\}$. WLOG, we assume that a subset of terminals, $T_i \subseteq T$, is on $BD(f_i)$ for each face $f_i \in \mathcal{F}$ and the T_i 's are disjoint from each other.

We define two key concepts, called *island cut* and *isolating cut*.

Definition 1. *For a face $f_i \in \mathcal{F}$ and the terminals of $T_i \subseteq T$ on $BD(f_i)$, an island cut for f_i is any set of edges of G whose removal disconnects each terminal $t \in T_i$ from all terminals in $T - T_i$.*

Definition 2. [10] *For a terminal t_i , an isolating cut for t_i is any set of edges of G whose removal disconnects t_i from all terminals in $T - \{t_i\}$.*

Some notations are needed. Given a graph $G = (V, E)$ and a vertex subset $V' \subset V$, let $C = (V', V - V')$ denote the cut of G each of whose edges connects a vertex in V' and a vertex in $V - V'$. We call every vertex that is not in V' but is adjacent to an edge in C an *extended vertex* of V' . Let $EV(V')$ denote the set of all extended vertices of V' . We define a new graph $G_{ex}(V') = (V_{ex}(V'), E_{ex}(V'))$, called an *extended graph* of V' , where $V_{ex}(V') = V' \cup EV(V')$ and $E_{ex}(V') = E[V'] \cup C$, and $E[V']$ is the edge set of the induced subgraph $G[V']$ of G . More specifically, if the cut $C = (V', V - V')$ is a minimum weight island cut for a face $f_i \in \mathcal{F}$ and the set of vertices on $BD(f_i)$, $V(f_i) \subseteq V'$, we call the extended graph $G_{ex}(V')$ an *island graph* for face f_i , simply denoted by G_i . Correspondingly, V_i^e denotes the set of extended vertices.

Note that some of the faces in $\mathcal{F}$ may be adjacent (two faces are said to be *adjacent* if they share a common vertex; otherwise, they are *disjoint*). Since our algorithm requires all faces in $\mathcal{F}$ to be disjoint in order to compute the optimal island cuts, we need to first transform G into a new planar graph $\tilde{G} = (\tilde{V}, \tilde{E})$ such that the T_i 's are on the boundaries of m disjoint faces, as described in Subsection 3.1. In Subsection 3.2, we give our approximation algorithm.

3.1 Generating Disjoint Covering Faces

To generate m disjoint faces that together cover all k terminals of T , we consider three cases based on the size of T_i . For $|T_i| = 1$ and $t \in T_i$, we put two vertices a and b inside the face f_i , and add three edges (t, a) , (t, b) , and (a, b) , each with weight $+\infty$ (see Figure 2(a)). For $|T_i| = 2$, if the two terminals t_0 and t_1 of T_i are adjacent, we put two vertices a and b inside f_i , each connecting to t_0 and t_1 with two edges of weight $+\infty$ (see Figure 2(b)); if t_0 and t_1 are not adjacent as in Figure 2(c), we insert one vertex a in f_i and three edges (t_0, t_1) , (a, t_0) , and (a, t_1) , each with weight $+\infty$. For $k_i = |T_i| \geq 3$, let $I(T_i) = (t_0, t_1, \dots, t_{k_i-1})$ be the ordered sequence of terminals in T_i that are visited by a counterclockwise traversal of $BD(f_i)$ starting at t_0 . For each pair (t_j, t_{j+1}) of consecutive terminals in $I(T_i)$, if t_j and t_{j+1} are adjacent, we put a vertex a_j in the face f_i , and connect a_j to t_j and t_{j+1} each with an edge of weight $+\infty$; otherwise, we connect t_j and t_{j+1} with an edge of weight $+\infty$. Figure 2(d) illustrates this case. For each face $f_i \in \mathcal{F}$ with its terminal set T_i , we perform the operation described above, and generate a new face f'_i for each T_i . Denote the resulted graph by $\tilde{G} = (\tilde{V}, \tilde{E})$. Lemma 8 characterizes some useful properties of the graph $\tilde{G}$.

Lemma 8. (1) $\tilde{G} = (\tilde{V}, \tilde{E})$ is a planar graph and $|\tilde{V}| = O(n)$. (2) Each T_i is on $BD(f'_i)$. (3) The faces in $\mathcal{F}' = \{f'_1, f'_2, \dots, f'_m\}$ are disjoint from each other and together cover all terminals of T . (4) Any optimal cut that separates the terminals on the boundary of each face $f'_i \in \mathcal{F}'$ from all remaining terminals does not cut the boundary of any face in $\mathcal{F}'$. (5) An optimal island cut for f'_i in $\tilde{G}$ is also an optimal island cut for f_i in G .

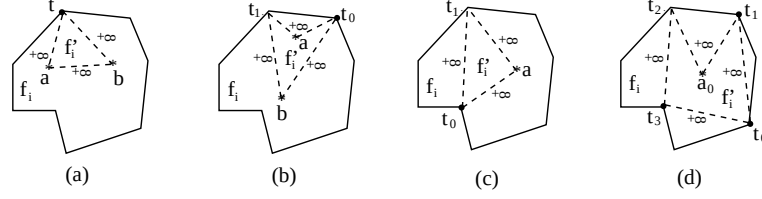


Fig. 2. Making the covering faces disjoint: The dots are terminals, star points represent added vertices, and dashed lines are for added edges. (a) $|T_i| = 1$; (b) $|T_i| = 2$ and the two terminals are adjacent; (c) $|T_i| = 2$ but the two terminals are not adjacent; (d) $|T_i| \geq 3$

3.2 Our Algorithm

We now give a $(2 - \frac{2}{k})$ -approximation algorithm with a $\min\{O(n^2 \log n \log m), O(m^2 n^{1.5} \log^2 n + kn)\}$ running time. Note that the isolating cuts for the k terminals of T together induce a k -terminal cut for G . Thus, as in [10], our goal is to seek a minimum weight isolating cut for each terminal $t_j \in T$. However, unlike [10], when dealing with this problem on planar graphs, we do not need to apply a minimum s - t cut algorithm individually and independently to each terminal to obtain an optimal isolating cut. Instead, we use optimal island cuts and shortest paths in planar graphs to obtain optimal isolating cuts for most terminals on the boundary of each face $f_i \in \mathcal{F}$, and use minimum s - t cuts to obtain optimal isolating cuts for the remaining terminals. Hence, our algorithm consists of three key steps: (1) compute an optimal island cut and an island graph G_i for every face f_i ; (2) use shortest paths in each island graph G_i to obtain optimal isolating cuts for most terminals in T_i (i.e., lying on $BD(f_i)$); (3) use optimal s - t cuts to compute optimal isolating cuts for the remaining terminals of T .

Step (1): Computing optimal island cuts and island graphs

We first transform G into another planar graph $\tilde{G}$ as in Subsection 3.1, and then compute optimal island cuts on $\tilde{G}$. Note that the sought optimal island cuts can be computed by using a minimum s - t cut algorithm for each (disjoint) face $f'_i \in \mathcal{F}'$ in $\tilde{G}$, but that will give a less efficient $O(mn^2 \log n)$ time algorithm. We are able to do better based on the following ideas. We first use a divide-and-conquer procedure to decompose $\tilde{G}$ into m subgraphs $\tilde{G}_1, \tilde{G}_2, \dots, \tilde{G}_m$, each containing exactly one face $f'_i \in \mathcal{F}'$. We then apply a minimum s - t cut algorithm on each subgraph $\tilde{G}_i$ to obtain an optimal island cut and an island graph for f'_i . As shown by our analysis, this approach leads to an $O(n^2 \log n \log m)$ time algorithm.

Our divide-and-conquer procedure decomposes $\tilde{G}$ into m subgraphs $\tilde{G}_i$, $i = 1, 2, \dots, m$, as follows. First, divide $\mathcal{F}'$ into two equal-size subsets $\mathcal{F}'_1 = \{f'_1, f'_2, \dots, f'_{\lfloor \frac{m}{2} \rfloor}\}$ and $\mathcal{F}'_2 = \{f'_{\lfloor \frac{m}{2} \rfloor + 1}, f'_{\lfloor \frac{m}{2} \rfloor + 2}, \dots, f'_m\}$. Introduce two dummy vertices s'_0 and s'_1 , connect s'_0 (resp., s'_1) to every terminal on the boundary of each face $f_a \in \mathcal{F}'_1$ (resp., $f_b \in \mathcal{F}'_2$) by an edge of weight $+\infty$, and compute a minimum s'_0 - s'_1 cut $\tilde{C}_1$ in the resulted graph. The cut $\tilde{C}_1$ induces a partition of the vertex

set $\tilde{V}$ of $\tilde{G}$ into X_1 and X_2 ($X_1 \cup X_2 = \tilde{V}$). The induced subgraph of X_1 (resp., X_2) is denoted by $\tilde{G}[X_1]$ (resp., $\tilde{G}[X_2]$). Based on Lemma 8, each $f'_j \in \mathcal{F}'$ is entirely in either $\tilde{G}[X_1]$ or $\tilde{G}[X_2]$. Next, recursively partition each of $\tilde{G}[X_1]$ and $\tilde{G}[X_2]$, until every induced subgraph contains exactly one face $f \in \mathcal{F}'$. In this way, we obtain a partition of $\tilde{V}$ into $\tilde{V}_1, \tilde{V}_2, \dots, \tilde{V}_m$. Each induced subgraph $\tilde{G}[\tilde{V}_i]$ contains exactly one face $f'_i \in \mathcal{F}'$. Recall that $\tilde{G}_{ex}(\tilde{V}_i)$ denotes the extended graph of $\tilde{V}_i$, the edge set of $\tilde{G}_{ex}(\tilde{V}_i)$ is denoted by $\tilde{E}_{ex}(\tilde{V}_i)$, and $EV(\tilde{V}_i)$ denotes the set of extended vertices of $\tilde{V}_i$. Figure 3(a) shows an embedded planar graph with 7 terminals (denoted by dots) on the boundaries of three faces f_1, f_2 , and f_3 , and illustrates the construction of disjoint covering faces (denoted by heavy dark lines) and the partitioning of $\tilde{G}$.

Based on Lemma 8, the size of $\tilde{G}$ is $O(n)$. Each edge in a cut $(\tilde{V}_i, \tilde{V} - \tilde{V}_i)$ can appear in the extended graphs $\tilde{G}_{ex}(\tilde{V}_i)$, $i = 1, 2, \dots, m$, at most twice. Hence, the next lemma follows.

Lemma 9. *The total size of the m extended graphs $\tilde{G}_{ex}(\tilde{V}_1), \tilde{G}_{ex}(\tilde{V}_2), \dots, \tilde{G}_{ex}(\tilde{V}_m)$ is $O(n)$.*

Next, we compute an optimal island cut $\tilde{L}_i$ in $\tilde{G}$ for each face $f'_i \in \mathcal{F}'$, as follows. For every extended graph $\tilde{G}_{ex}(\tilde{V}_i)$, introduce two dummy vertices s'_0 and s'_1 , connect s'_0 to each terminal on $BD(f'_i)$ by an edge of weight $+\infty$, and s'_1 to each extended vertex in $EV(\tilde{V}_i)$ by an edge of weight $+\infty$, and compute a minimum s'_0 - s'_1 cut in the resulted graph. (Note that the resulted graph is planar.) Lemma 10 shows that $\tilde{L}_i$ thus obtained is an optimal island cut in $\tilde{G}$ for the face f'_i .

Lemma 10. *Our algorithm produces an optimal island cut in $\tilde{G}$ for each f'_i in $\mathcal{F}'$.*

So far, we obtain an optimal island cut $\tilde{L}_i$ for each $f'_i \in \mathcal{F}'$ in $\tilde{G}$. Denote by L_i the corresponding cut of $\tilde{L}_i$ in G for the face f_i . Based on Lemma 8, the following lemma holds.

Lemma 11. *L_i is an optimal island cut for $f_i \in \mathcal{F}$ in G , for every $i = 1, 2, \dots, m$.*

Recall that the island graph G_i is the extended graph of an optimal island cut L_i for face f_i on G . Figure 3(c) shows the island graph G_1 for face f_1 , denoted by solid lines, with extended vertices $v_0, v_1, \dots, v_6$ of L_1 . The next lemma is about the total size of all island graphs.

Lemma 12. *The total size of all island graphs $G_1, G_2, \dots$, and G_m is $O(n)$.*

Step (2): Computing shortest paths

This step is based on a key observation that optimal isolating cuts for most terminals (except for at most one) on $BD(f_i)$ can be computed as shortest paths in an augmented dual graph of G_i for each face $f_i \in \mathcal{F}$.

When the number of terminals, $|T_i|$, on $BD(f_i)$ is more than one, we construct an augmented dual graph G'_i corresponding to the island graph G_i , as follows.

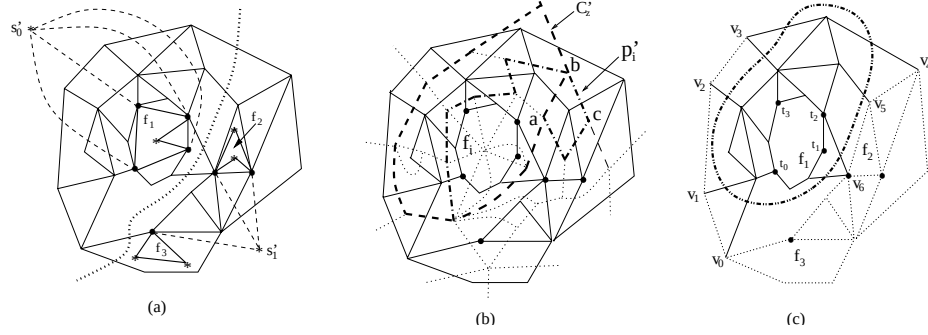


Fig. 3. (a) Illustrating the construction of disjoint covering faces (with heavy dark lines) and the partitioning of G . (b) Illustrating the optimality of the island cuts. (c) The island graph G_1 (with solid-line edges) for f_1

Let F_i be the set of faces of G each of which is bounded by at least one edge of G_i . As in Section 2, build the augmented dual graph G'_i of G_i with respect to the terminals in T_i based on the faces of F_i (in Figure 4(a), G'_1 consists of dashed-line edges). Let $k_i = |T_i|$, $S_i = \{s_0, s_1, \dots, s_{k_i-1}\}$ be the set of augmented vertices in G'_i , and $I'_i = (s_0, s_1, \dots, s_{k_i-1})$ be the sequence of vertices in S_i corresponding to the counterclockwise terminal sequence $(t_0, t_1, \dots, t_{k_i-1})$ of T_i on $BD(f_i)$. We compute a shortest path p_j in G'_i connecting every two consecutive vertices s_j and s_{j+1} in I'_i , $j = 0, 1, \dots, k_i - 1$ (with $s_{k_i} = s_0$). Note that the set C_j of edges in G_i corresponding to the path p_j either forms an isolating cut for $t_j \in T_i$ in G (e.g., t_3 and the s_3 -to- s_0 path p_3 in Figure 4(b)) or it does not (e.g., t_1 and the s_1 -to- s_2 path p_1 in Figure 4(b)). If it does, we say that p_j defines an isolating cut C_j for t_j in G ; else, p_j does not define an isolating cut for t_j in G . Fortunately, as Lemma 13 shows, for each face $f_i \in \mathcal{F}$, there can be at most one such shortest path p_l in G'_i that does not define an isolating cut in G for its corresponding terminal $t_l \in T_i$. WLOG, for each terminal $t_j \in T_i$, let the edge of the path p_j that is adjacent to s_j (resp., s_{j+1}) cross $BD(f_i)$ at a point c_j (resp., c_{j+1}), and R_j be the region on the plane enclosed together by p_j and the path on $BD(f_i)$ from c_j counterclockwise to c_{j+1} .

Lemma 13. *Among all shortest paths in G'_i computed by our algorithm for each face f_i , at most one such path does not define an isolating cut in G for its corresponding terminal on $BD(f_i)$.*

If Lemma 13 holds for a terminal t_l on $BD(f_i)$, we call t_l an *exceptional terminal*. The optimal isolating cuts for all exceptional terminals of G are found as follows.

Step (3): Computing isolating cuts for exceptional terminals

First, we need to identify the exceptional terminal (if any) for each face f_i . Note that if $t_l \in T_i$ is an exceptional terminal, then there is a path in the graph $G_i - C_l = (V_i, E_i - C_l)$ that connects t_l to some extended vertex in G_i .

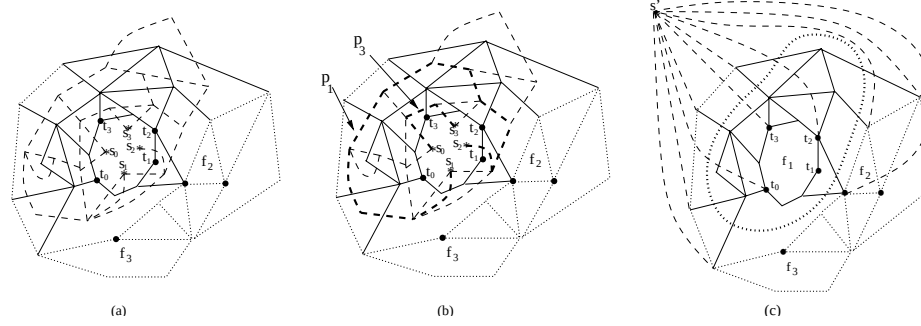


Fig. 4. (a) The augmented dual graph G'_1 of G_1 for the terminals on f_1 (with dashed-line edges). (b) Some (heavy dashed-line) shortest paths p_j in G'_1 . (c) Finding an optimal isolating cut for an exceptional terminal t_1 in T_1

Hence, deciding whether each $t_j \in T_i$ is an exceptional terminal can be done in the same time bound as that for computing the shortest path p_j in G'_i .

Next, assume that $t \in T_i$ on $BD(f_i)$ is an exceptional terminal, and V_i^e is the set of extended vertices in G_i . Add a dummy vertex s' to G_i and connect s' to each vertex in $V_i^e \cup (T_i - \{t\})$ by an edge of weight $+\infty$. Then, compute an optimal $s'-t$ cut C_t in the resulted graph $G_i^{s'}$. Note that if t has a path to any terminal in $T - T_i$, such a path must go through a vertex in V_i^e . Thus, C_t is an isolating cut for t in G . In Figure 4(c), t_1 is the exceptional terminal in T_1 , and the dashed curves connecting the dummy vertex s' are the added dummy edges.

At this point, we have obtained an isolating cut C_j for each terminal $t_j \in T$. We need to argue that every such C_j is an optimal isolating cut for t_j .

Lemma 14. *Our algorithm produces an optimal isolating cut C_j for each terminal $t_j \in T$.*

After obtaining an optimal isolating cut C_j for each terminal $t_j \in T$, we find, as in [10], the cut C_l such that C_l has the maximum cost among all the C_j 's. Let C be the union of all isolating cuts C_j except C_l . Clearly, C is a k -terminal cut for G . As to the approximation ratio of C to the optimal k -terminal cut of G , we can simply apply the same argument as in [10]. Thus, the next lemma follows.

Lemma 15. *Our algorithm constructs a k -terminal cut for an undirected weighted planar graph whose total cost is no more than $(2 - \frac{2}{k})$ times that of the optimal k -terminal cut.*

The best known deterministic $s-t$ cut algorithm is due to Goldberg and Tarjan [15], which takes $O(n^2 \log n)$ time on a sparse graph of $O(n)$ vertices and edges. In Step (1) of our algorithm, we use the minimum $s-t$ cut algorithm to partition $\tilde{G}$ into m subgraphs recursively, which takes $O(n^2 \log n \log m)$ time. Then, computing an optimal island cut in each of the m subgraphs altogether

takes $O(n \log n)$ time due to Lemma 9 and the planarity of every such subgraph. Thus, the total running time of Step (1) is $O(n^2 \log n \log m)$. A shortest path in a planar graph can be computed in linear time [17]. Based on Lemma 12, the total size of all island graphs $G_1, G_2, \dots, G_m$ (which are all planar) is $O(n)$. Step (2) computes a shortest path in the augmented dual graph G'_i of every island graph G_i for each terminal in T_i , and hence this step takes $O(kn)$ time. Lemma 13 bounds the number of exceptional terminals of G by m . For each exceptional terminal, Step (3) finds an optimal s - t cut in its island graph. The total time of Step (3) is $O(\sum_{i=1}^m n_i^2 \log n_i)$, where n_i is the size of G_i . By Lemma 12, we have $O(\sum_{i=1}^m n_i^2 \log n_i) = O(n^2 \log n)$. Clearly, the running time of Step (1) dominates the other steps. Therefore, the total running time of our algorithm is $O(n^2 \log n \log m)$.

The running time of the above algorithm can be improved for a considerable range of values of m . Note that the s - t cut computation is a key procedure of our divide-and-conquer scheme and of Step (3). Also, note that adding two dummy vertices s and t to form an s - t cut problem instance may destroy the planarity of the graph in our constructions. Here, the cut problem that we actually need to handle is that of computing a minimum cut on a planar graph with multiple sources and sinks, with those vertices adjacent to s (resp., t) being the sources (resp., sinks). Miller and Naor [19] designed an $O(m^2 n^{1.5} \log^2 n)$ time algorithm for computing a maximum flow in an n -vertex planar graph with k sources and sinks lying on m faces. Thus, in Step (1), we can apply Miller and Naor's maximum-flow algorithm [19] in our divide-and-conquer scheme to decompose $\tilde{G}$ into m subgraphs recursively (without adding the dummy vertices s and t); this decomposition takes altogether $O(m^2 n^{1.5} \log^2 n)$ time. The total time for computing an optimal island cut in each of the m resulted subgraphs is still $O(n \log n)$ since each such subgraph is a planar one. Step (2) still takes $O(kn)$ time as before. In Step (3), for each exceptional terminal $t \in T_i$, note that the sources and sinks lie on at most 2 faces of G_i . Thus, the total time of Step (3) is $O(\sum_{i=1}^m n_i^{1.5} \log^2 n_i) = O(n^{1.5} \log^2 n)$, where n_i is the size of G_i . Hence, the time complexity of our algorithm is $O(m^2 n^{1.5} \log^2 n + kn)$ by using Miller and Naor's maximum-flow algorithm [19]. Note that for the case when $m = O(\frac{n^{1/4}}{\sqrt{\log n}})$, this version of our algorithm is more efficient than the one using s - t cuts in sparse graphs, and gives at least an $O(k)$ time improvement over Dahlhaus *et al.*'s algorithm [10] when it is applied to planar graphs.

Theorem 2. *In $\min\{O(n^2 \log n \log m), O(m^2 n^{1.5} \log^2 n + kn)\}$ time, our algorithm computes a $(2 - \frac{2}{k})$ -approximate k -terminal cut in an n -vertex undirected weighted planar graph embedded in the plane such that there are m faces of the graph whose boundaries cover all the k terminals.*

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974. 334, 336
2. S. Arora, D. Karger, and M. Karpinski, Polynomial-time Approximation Schemes for Dense Instances of NP-hard Problems, *Proc. 27th ACM STOC*, 1995, 284-293. 333
3. M. Bern, Faster Exact Algorithms for Steiner Trees in Planar Networks, *Networks*, 20(1990), 109-120. 337
4. D. Bertsimas, C. Teo, and R. Vohra, Nonlinear Formulations and Improved Randomized Approximation Algorithms for Multicut Problems, *Proc. 4th Conference on Integer Programming and Combinatorial Optimization*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 920(1995), 29-39. 333
5. D. Bienstock and C. L. Monma, On the Complexity of Covering Vertices by Faces in a Planar Graph, *SIAM J. Computing*, 17(1988), 53-76. 334
6. G. Călinescu and H. Karloff, An Improved Approximation Algorithm for Multiway Cut, *Journal of Computer and System Sciences*, 60(2000), 564-574. 332, 333, 334
7. S. Chopra and M. R. Rao, On the Multiway Cut Polyhedron, *Networks*, 21(1991), 51-89. 333
8. W. H. Cunningham, The Optimal Multiterminal Cut Problem, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 5(1991), 105-120. 333
9. E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, The Complexity of Multiway Cuts, extended abstract, 1983. 332
10. E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, The Complexity of Multiterminal Cuts, *SIAM J. Comp.*, 23(1994), 864-894. 332, 333, 334, 337, 339, 342, 343
11. L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962. 332
12. G. Frederickson, Planar Graph Decomposition and All Pairs Shortest Paths, *J. of the ACM*, 38(1991), 162-204. 334
13. A. Frieze and F. Kannan, The Regularity Lemma and Approximation Schemes for Dense Problems, *Proc. 37th IEEE FOCS*, 1996, 12-20. 333
14. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979. 333, 335
15. A. V. Goldberg and R. E. Tarjan, A New Approach to the Maximum-flow Problem, *J. Assoc. Comput. Mach.*, 35(1988), 921-940. 342
16. D. Hartvigsen, The Planar Multiterminal Cut Problem, *Discrete Applied Mathematics*, 85(1998), 203-222. 333
17. M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian, Faster Shortest-path Algorithms for Planar Graphs, *J. of Computer and System Sciences*, 55(1997), 3-23. 337, 343
18. D. R. Karger, P. Klein, C. Stein, M. Thorup, and N. E. Young, Rounding Algorithms for a Geometric Embedding of Minimum Multiway Cut, *Proc. 31st ACM STOC*, 1999, 668-678. 333, 334
19. G. L. Miller and J. Naor, Flow in Planar Graphs with Multiple Sources and Sinks, *SIAM J. Comput.*, 24(1995), 1002-1017. 343
20. H. Saran and V. V. Vazirani, Finding k -cuts within Twice the Optimal, *Proc. 32nd IEEE FOCS*, 1991, 743-751.

Polynomial Time Algorithms for Edge-Connectivity Augmentation of Hamiltonian Paths^{*}

Anna Galluccio¹ and Guido Proietti^{2,1}

¹ Istituto di Analisi dei Sistemi ed Informatica
CNR, Viale Manzoni 30, 00185 Roma, Italy
galluccio@iasi.rm.cnr.it

<http://www.iasi.rm.cnr.it/iasi/personnel/galluccio.html>

² Dipartimento di Matematica Pura ed Applicata, Università dell'Aquila
Via Vetoio, 67010 L'Aquila, Italy
proietti@univaq.it
<http://w3.dm.univaq.it/~proietti>

Abstract. Given a graph G of n vertices and m edges, and given a spanning subgraph H of G , the problem of finding a minimum weight set of edges of G , denoted as $\text{AUG}_2(H, G)$, to be added to H to make it 2-edge connected, is known to be NP-hard. In this paper, we present polynomial time efficient algorithms for solving the special case of this classic augmentation problem in which the subgraph H is a Hamiltonian path of G . More precisely, we show that if G is unweighted, then $\text{AUG}_2(H, G)$ can be computed in $O(m)$ time and space, while if G is non-negatively weighted, then $\text{AUG}_2(H, G)$ can be computed in $O(m + n \log n)$ time and $O(m)$ space. These results have an interesting application for solving a survivability problem on communication networks.

Keywords: Graph, Hamiltonian path, augmentation, 2-edge connectivity, network survivability.

1 Introduction

Let $G = (V, E)$ be a connected, undirected graph, of n vertices and m edges, where an edge $e = (u, v) \in E$ represents a potential connection between vertices u and v . Let us assume that a non-negative weight is associated with each edge $e \in E$, expressing some cost for activating the edge, and consider the problem of building a network in G which allows all the sites to communicate. For this kind of network, it is generally important to be both *economically attractive*, i.e., it should be as sparse as possible to reduce set-up costs, and *reliable*, i.e., it should remain operational even if individual network components fail.

^{*} This work has been partially supported by the CNR-Agenzia 2000 Program, under Grants No. CNRC00CAB8 and CNRG003EF8, and by the Research Project REACTION, partially funded by the Italian Ministry of University.

As a consequence, the problem of designing networks which combine in the best possible way these two conflicting parameters, i.e., sparseness and reliability, is usually known as the *survivable network design problem* [8]. This topic encompasses a large set of practical applications (e.g., communication and transportation networks design, VLSI layout [12], etc.), as well as theoretical problems (e.g., the Steiner tree [16], the traveling salesman, the minimum cost k -connected subgraph [10], etc.).

The cheapest solution to the problem of designing a communication network in G is that of designing a *minimum weight spanning tree* of G , namely a connected, spanning subgraph of G such that the sum over all the edge weights is minimum. Unfortunately, such a structure will not even survive a single link or site failure. For the case of link failures, which is of interest for this paper, one possibility to solve the problem is that of designing networks with higher edge-connectivity degree. In fact, edge-connectivity $k > 1$ implies the existence of $k - 1$ edge-disjoint paths between any pair of nodes. Thus, a k -edge connected network will survive to the disruption of $k - 1$ links. However, this approach has (at least) two drawbacks: First, computing a minimum weight k -edge connected spanning network of a given graph is NP-hard (although, approximable within a constant ratio [7,11,14]), and second, the communication protocol redundancy grows as k grows.

Fortunately, in practical applications, we can safely assume that a damaged network component can be restored quite quickly. Therefore, the likelihood of having multiple overlapping failures is small. Hence, an alternative strategy to increase the reliability without oversizing the network may be to design it onto two levels: a *primary level* of active links (i.e., the backbone where communication is carried out in the absence of failures), and a *secondary level* of inactive links, a subset of which will switch to active as soon as the network undergoes some link failure. Given their role, links on the secondary level are called *replacement* links.

From a theoretical point of view, the problem of finding a minimum weight set of replacement edges to restore the connectivity after an edge failure, is a classic *edge-connectivity augmentation problem*. In its more general formulation, this problem consists of finding a minimum weight set of edges of a graph G whose addition to a given h -edge connected spanning subgraph H of G increases its edge-connectivity to a prescribed value $k > h > 1$. Such a problem turns out to be NP-hard [3], and thus most of the research in the past focused on the design of approximation algorithms for solving it. In particular, for the special case $k = 2$, which is of interest for this paper, efficient approximation algorithms are known. More precisely, for the weighted case, the best performance ratio is 2 [5,9], while for the unweighted case, Nagamochi and Ibaraki developed a $(51/26 + \epsilon)$ -approximation algorithm, for any constant $\epsilon > 0$ [13]. Analogous versions of augmentation problems for vertex-connectivity and for directed graphs have been widely studied, and we refer the interested reader to the following comprehensive papers [4,10].

Besides designing approximated solutions, researchers have also investigated the problem of characterizing polynomial time solvable cases of the problem. This was first done by Eswaran and Tarjan, which proved that the case $k = 2$ can be solved in polynomial time if G is complete and all edges have weight 1, namely all potential links between sites may be activated at the same cost [3]. Afterwards, Watanabe and Nakamura extended this result to any desired edge-connectivity value [15], and faster algorithms in this scenario have been proposed in [6]. The main purpose of the present paper is to enlarge the set of cases that can be solved in polynomial time. More precisely, we show that if G is unweighted and H is a Hamiltonian path of G , then finding the minimum number of edges of G to be added to H to increase its edge-connectivity to 2 can be solved in optimal $O(m)$ time and space. Moreover, we show that the weighted version of the above problem when G has non-negative weights on its edges, can be solved in $O(m + n \log n)$ time and $O(m)$ space. In the unweighted case, our algorithm is based on a greedy paradigm, while for the weighted case, our algorithm uses a dynamic programming technique.

Our algorithms find practical applications in those scenarios in which 2 overlapping link failures in communication networks has to be afforded, but a 3-edge connected primary level of the network itself is too costly to be maintained. In such a case, we can design a 2-edge connected primary level of the network, and as soon as a link fails, we can efficiently provide an optimal set of replacement links, both in the unweighted and in the weighted case. In this way, the *emergency network*, as obtained by removing from the original network the failed link and by adding the corresponding replacement links, is again 2-edge connected, while at the same time it keeps on to maintain all the old, still working, links.

The paper is organized as follows: in Section 2 we give some basic definitions that will be used throughout the paper; in Section 3 we show how to augment a Hamiltonian path in unweighted graphs, while in Section 4 we study the weighted version of the problem; in Section 5, we apply these results to handle transient edge failures in 2-edge connected networks, and finally, in Section 6, we present conclusions and list some open problems.

2 Basic Definitions

Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. G is said to be *weighted* if there exists a real function $w : E \mapsto \mathbb{R}$, otherwise G is *unweighted*. In this paper, we will be concerned with non-negatively weighted graphs. If multiple edges between pairs of vertices are allowed, then the graph is said to be a *multigraph*. A graph $H = (V(H), E(H))$ is called a *subgraph* of G if $V(H) \subseteq V$ and $E(H) \subseteq E$. If $V(H) = V$, then H is called a *spanning subgraph* of G . The weight of H is defined as $w(H) = \sum_{e \in E(H)} w(e)$.

A *simple path* (or a *path* for short) in G is a subgraph H of G with $V(H) = \{v_0, \dots, v_k | v_i \neq v_j \text{ for } i \neq j\}$ and $E(H) = \{(v_i, v_{i+1}) | 0 \leq i < k\}$, also denoted as $P(v_0, v_k) = \langle v_0, v_1, \dots, v_k \rangle$. A *cycle* is a path whose end vertices v_0 and v_k

coincide. A spanning path of G is called a *Hamiltonian path* of G . Let $\Pi = \langle v_0, v_1, \dots, v_{n-1} \rangle$ be a Hamiltonian path of G . Edges in Π are called *path edges*, while the remaining edges of G are called *cycle edges*. A cycle edge (u, v) covers all the path edges along the path from u to v in Π .

A graph G is *connected* if, for any $u, v \in V$, there exists a path $P(u, v)$ in G . A graph G is said to be *k -edge connected*, where k is a positive integer, if the removal of any $k - 1$ distinct edges from G leaves G connected. Given an h -edge connected spanning subgraph H of a k -edge connected graph G , and a positive integer $h < \lambda \leq k$, finding a λ -*augmentation* of H in G means to select a minimum weight set of edges in $E \setminus E(H)$, denoted as $\text{AUG}_\lambda(H, G)$, such that the spanning subgraph $H' = (V, E(H) \cup \text{AUG}_\lambda(H, G))$ of G is λ -edge connected.

3 Augmenting Hamiltonian Paths in Unweighted Graphs

Let $G = (V, E)$ be a 2-edge connected, unweighted graph, and let Π be a Hamiltonian path of G . Let us start by giving the notion of *path-carving* of G , which is a restriction of the notion of tree-carving given in [11]. A path-carving is a partition of the vertex set V into subsets $V_1, V_2, \dots, V_k$, satisfying the following property: The end vertices of each edge of G belong either to the same subset, or to a pair of consecutive subsets.

We first prove the following:

Theorem 1. *Let G be a 2-edge connected and unweighted graph with n vertices and m edges. Let $\Pi = \langle v_0, v_1, \dots, v_{n-1} \rangle$ be a Hamiltonian path of G . Then, $\text{AUG}_2(\Pi, G)$ can be computed in $O(m)$ time and space.*

Proof. First of all, a set of cycle edges is added to Π , to cover all the path edges. This is done by applying the technique proposed in [11] to cover all the edges of a depth-first search tree. Let us describe how such a technique works for a Hamiltonian path: visit all the vertices of Π one after the other, starting from v_0 , and check whether the edge (v_i, v_{i+1}) in Π is currently covered; if not, add a cycle edge (v_s, v_t) such that $s \leq i$ and v_t is as close as possible to v_{n-1} .

After this first phase, all the edges of Π which caused the insertion of a cycle edge are removed. It is not hard to see that the vertex partition induced by the resulting connected components in Π provides a path-carving in G . Let $V_1, V_2, \dots, V_k$ be such partition. The following holds:

Lemma 1. $|\text{AUG}_2(\Pi, G)| \geq k - 1$.

Proof. Observe that a lower bound on the number of edges of any 2-edge connected spanning subgraph of G is $2(k - 1)$ [11]. In fact, for $i = 1, \dots, k - 1$, at least two edges between V_i and V_{i+1} are needed to guarantee the 2-edge connectivity. Since Π contains just one edge between V_i and V_{i+1} (otherwise we would have a cycle in Π), it follows that any 2-edge connected spanning subgraph of G contains at least $k - 1$ edges which do not belong to Π . From this, the claim follows. $\square$

Since the $k - 1$ cycle edges added to Π increase its edge-connectivity to 2, it follows that this set of edges provides a 2-augmentation of Π in G .

Concerning the time and space complexity, observe that the path-carving can be found in $O(m)$ time and space [11], and from this the thesis follows. $\square$

4 Augmenting Hamiltonian Paths in Weighted Graphs

Let $G = (V, E)$ be a 2-edge connected graph with a non-negative weight function w on the edges, and let $\Pi = \langle v_0, v_1, \dots, v_{n-1} \rangle$ be a Hamiltonian path of G . In the following, a cycle edge (v_i, v_j) , $i < j$, will be considered as a *right* edge for v_i and as a *left* edge for v_j . Moreover, $\mathcal{L}_i$ and $\mathcal{R}_i$ will denote the set of left and right cycle edges of v_i , respectively. Let Π_k denote the restriction of Π to $\langle v_0, v_1, \dots, v_k \rangle$, and let e_k denote the edge (v_{k-1}, v_k) . A *covering* of Π_k is a set of edges in $E \setminus E(\Pi)$ which cover all the edges of Π_k . Figure 1 illustrates the notations used.

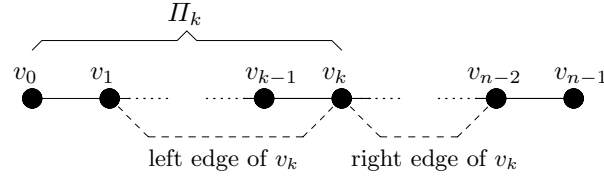


Fig. 1. Subpath Π_k of Π (solid edges), with a left and a right edge of v_k (dashed)

In the next subsections, we first give a high-level description of the algorithm, and we then analyze its correctness and its time and space complexity.

4.1 High-Level Description of the Algorithm

The algorithm consists of $n - 1$ iterations. At the k -th iteration, the algorithm computes a suitable edge $c(k) = (v_s, v_t)$ covering e_k , and such that the set of edges $\text{SOL}(k)$ defined recursively as follows

$$\text{SOL}(k) = \begin{cases} \emptyset & \text{if } k = 0, \\ c(k) \cup \text{SOL}(s) & \text{if } k > 0 \text{ and } c(k) = (v_s, v_t), \end{cases} \quad (1)$$

is a covering of Π_k . As we will prove later, such a covering satisfies the property of being a minimum weight set of edges covering Π_k . For space efficiency reasons, $\text{SOL}(k)$ cannot be stored explicitly throughout the execution of the algorithm. However, as we will see shortly, we only need to maintain its weight $w(\text{SOL}(k))$ to guarantee the algorithm correctness.

To select the edge $c(k)$ properly among all the edges covering e_k , the algorithm maintains a set of *active vertices* $\mathcal{V}_k = \{v_k, \dots, v_{n-1}\}$. With each active vertex $v_j \in \mathcal{V}_k$, two labels are associated:

1. an *edge* $\sigma(v_j)$, belonging to $\mathcal{L}_j$;
2. a *key* $\kappa(v_j)$, containing the weight of a set of edges covering Π_k and using $\sigma(v_j)$.

Throughout the execution of the algorithm, the following invariant is maintained: at the beginning of iteration k , $\sigma(v_j)$ contains an edge (v_s, v_j) of $\mathcal{L}_j$, if any, such that

$$w(\text{SOL}(s)) + w((v_s, v_j)) = \min_{\substack{e = (v_i, v_j) \in \mathcal{L}_j \\ i = 0, \dots, k-2}} \{w(e) + w(\text{SOL}(i))\}. \quad (2)$$

The algorithm starts by initializing $\mathcal{V}_0 := V$, and by letting $w(\text{SOL}(0)) := 0$. Moreover, it sets $\sigma(v_j) := \emptyset$ and $\kappa(v_j) := +\infty$, for each $v_j \in \mathcal{V}_0$.

At the first iteration, the algorithm aims at covering edge e_1 . Hence, it first sets $\mathcal{V}_1 := \mathcal{V}_0 \setminus \{v_0\}$, and then it considers all the right edges of v_0 . For each such edge, say $e = (v_0, v_j)$, it sets $\sigma(v_j) := e$ and $\kappa(v_j) := w(e)$. Then, it looks at the element in $\mathcal{V}_1$ having minimum key, say v_t , and sets $c(1) := \sigma(v_t)$ and $w(\text{SOL}(1)) := \kappa(v_t)$.

The generic k -th iteration may be described as follows:

Step 1: Remove vertex v_{k-1} from $\mathcal{V}_{k-1}$, creating $\mathcal{V}_k$.

(**Comment:** Edges in $\mathcal{L}_{k-1}$ cannot be used to cover e_k and subsequent edges; hence, they are removed from further considerations.)

Step 2: Consider all the edges in $\mathcal{R}_{k-1}$. For each such edge, say $e = (v_{k-1}, v_j)$, let

$$\kappa' = w(e) + w(\text{SOL}(k-1)). \quad (3)$$

If $\kappa' < \kappa(v_j)$, decrease the key of v_j to value κ' , and set $\sigma(v_j) := e$.

(**Comment:** The labels of the active vertices in $\mathcal{V}_k$ are updated. More precisely, at the end of this step, for each v_j in $\mathcal{V}_k$, we have that

$$\kappa(v_j) = \min_{i=0, \dots, k-1} \{w(e) + w(\text{SOL}(i)) \mid e \in \mathcal{R}_i \wedge e \in \mathcal{L}_j\}, \quad (4)$$

and $\sigma(v_j)$ is the left edge of v_j minimizing (4).)

Step 3: Find the minimum key in $\mathcal{V}_k$; let v_t be the corresponding vertex, and let $\sigma(v_t) = (v_s, v_t)$. Then set

$$c(k) := \sigma(v_t) \quad \text{and} \quad w(\text{SOL}(k)) := \kappa(v_t). \quad (5)$$

At the end of the $(n-1)$ -th iteration, the algorithm computes the set of edges $\text{SOL}(n-1)$, as defined in (1), by using the edges obtained in (5). In the next subsection, we shall prove that $\text{SOL}(n-1)$ contains a minimum weight set of edges covering Π in G .

4.2 Analysis of the Algorithm

In order to prove that the algorithm finds a 2-augmentation of Π in G , we first show that $w(\text{SOL}(n-1)) = w(\text{AUG}_2(\Pi, G))$, namely that the weight of the solution found by the algorithm equals the weight of an optimal solution of the 2-augmentation problem.

We have the following:

Lemma 2. $w(\text{SOL}(n-1)) = w(\text{AUG}_2(\Pi, G))$.

Proof. Let $\text{OPT}(k)$ denote a minimum weight set of edges of G covering Π_k , with $\text{OPT}(0) := \emptyset$ and $w(\text{OPT}(0)) := 0$. Notice that $w(\text{OPT}(n-1)) = w(\text{AUG}_2(\Pi, G))$. We shall prove that $w(\text{SOL}(k)) = w(\text{OPT}(k))$, for $k = 0, \dots, n-1$. The proof is by induction on k . For $k = 0$ and $k = 1$, the thesis follows trivially. Assume the thesis is true up to $k-1 < n-1$, i.e., $w(\text{SOL}(i)) = w(\text{OPT}(i))$ for $i = 0, \dots, k-1$. We shall prove that $w(\text{OPT}(k)) = w(\text{SOL}(k))$.

Clearly, $\text{OPT}(k)$ has to contain a cycle edge covering e_k . Therefore, an optimal solution for Π_k has weight

$$\begin{aligned} w(\text{OPT}(k)) &= \min_{\substack{e = (v_i, v_j) \in E \\ i = 0, \dots, k-1}} \{w(e) + w(\text{OPT}(i))\} = \\ &= \min_{j=k, \dots, n-1} \left\{ \min_{i=0, \dots, k-1} \{w(e) + w(\text{OPT}(i)) \mid e \in \mathcal{R}_i \wedge e \in \mathcal{L}_j\} \right\}. \end{aligned} \quad (6)$$

On the other hand, from the algorithm, we have that

$$\begin{aligned} w(\text{SOL}(k)) &= \min_{j=k, \dots, n-1} \kappa(v_j) = \\ &= \min_{j=k, \dots, n-1} \left\{ \min_{i=0, \dots, k-1} \{w(e) + w(\text{SOL}(i)) \mid e \in \mathcal{R}_i \wedge e \in \mathcal{L}_j\} \right\}, \end{aligned} \quad (7)$$

and given that, by assumption, $w(\text{SOL}(i)) = w(\text{OPT}(i))$ for $i = 0, \dots, k-1$, we have that (6) and (7) coincide, and the thesis follows. $\square$

By making use of the above lemma, the following theorem can finally be proved:

Theorem 2. *Let G be a 2-edge connected graph with n vertices, m edges and with non-negative weights on the edges. Let Π be a Hamiltonian path of G . Then, $\text{AUG}_2(\Pi, G)$ can be computed in $O(m + n \log n)$ time and $O(m)$ space.*

Proof. To compute $\text{AUG}_2(\Pi, G)$, we make use of the algorithm presented in the previous section. The correctness of the algorithm derives from the fact that $\text{SOL}(k)$, for any $k = 0, \dots, n-1$, contains a set of edges of G which, by construction, cover Π_k . Hence, $\text{SOL}(n-1)$ consists of a set of edges covering $\Pi_{n-1} = \Pi$, and from Lemma 2, its weight is minimum.

The time complexity follows from the maintenance of sets $\mathcal{V}_k, k = 0, \dots, n-1$, by means of the efficient implementation of priority queues proposed in [2]. In fact, to create $\mathcal{V}_0$ we perform a *MakeQueue* operation, followed by n *Insert* operations (one for each vertex in G). Trivially, $\mathcal{V}_k$ can be obtained from $\mathcal{V}_{k-1}$ by simply removing v_{k-1} , and then we have a total of $n-1$ *Delete* operations. As far as key maintenance is concerned (**Step 2**), notice that $O(m)$ *DecreaseKey* operations take place (since a key may be decreased only when a new right edge is considered, and each right edge is considered at most once). Finally, a total of $n-1$ *FindMin* operations are needed to execute **Step 3** over all the algorithm. Therefore, we obtain a total of $O(m + n \log n)$ time to maintain sets $\mathcal{V}_k, k = 0, \dots, n-1$, since we pay $O(\log n)$ worst-case time for a *Delete* operation, and $O(1)$ worst-case time for all the other operations [2].

The operations in (3) and (5) are clearly performed in $O(1)$ time for each cycle edge. Moreover, $\text{SOL}(n-1)$ can be computed in $O(n)$ time, by making use of (1). Finally, the time complexity for managing sets $\mathcal{R}_i$ and $\mathcal{L}_i, i = 0, \dots, n-1$, is trivially $O(m)$. So, the overall time complexity of the algorithm is $O(m + n \log n)$.

It is easy to see that all the above operations can be performed by using $O(m)$ space, and then the claim follows. $\square$

5 Maintaining 2-Edge Connectivity through Augmentation

The results of the previous sections have an interesting application for solving a survivability problem on networks, that is the problem of adding to a given 2-edge connected network undergoing a transient edge failure, the minimum weight set of edges needed to reestablish the 2-edge connectivity. In this way, extensive (in terms of both computational efforts and set-up costs) network restructuring is avoided.

Let H be a 2-edge connected spanning subgraph of a 3-edge connected graph G . Let $G - e$ denote the graph obtained from G by removing an edge $e \in E$. Given an edge $e \in E(H)$, if $H - e$ is not 2-edge connected, then we say that e is *vital* for H . In the sequel, an edge e removed from H will always be considered as vital for H .

Let $\text{AUG}_2(H - e, G - e)$ be a minimum weight set of edges in $E \setminus E(H - e)$ such that the spanning subgraph $H' = (V, E(H - e) \cup \text{AUG}_2(H - e, G - e))$ of $G - e$ is 2-edge connected. Using the results of the previous sections, we prove that $\text{AUG}_2(H - e, G - e)$ can be computed efficiently both when G is unweighted and when it has non-negative weights on the edges. More precisely:

Theorem 3. *Let G be a 3-edge connected graph with n vertices, m edges and with non-negative weights on the edges. Let H be a 2-edge connected spanning subgraph of G . Then, for any vital edge $e \in E(H)$, we have that the set of edges $\text{AUG}_2(H - e, G - e)$ can be computed in $O(m + n \log n)$ time and $O(m)$ space. The running time can be lowered to $O(m)$ if all edge weights are unitary.*

Proof. After the removal of e from H , every 2-edge connected component in $H - e$ can be computed in $O(m)$ time and space [1]. Let V_i denote the vertex set of the i -th 2-edge connected component in $H - e$, and let $V_1, V_2, \dots, V_k$ be the corresponding vertex partition of V . Let $\Pi = \langle \nu_1, \nu_2, \dots, \nu_k \rangle$ be the path resulting from the contraction of each such vertex set to a single vertex, where vertex ν_i is associated with vertex set V_i . Hence, let $\mathcal{G}$ be the multigraph with vertex set $V(\mathcal{G}) = V(\Pi)$ and edge set

$$E(\mathcal{G}) = E(\Pi) \cup \{(\nu_i, \nu_j) \mid \exists u \in V_i \wedge \exists v \in V_j \text{ such that } (u, v) \in E \setminus E(H - e)\}.$$

It is easy to realize that the algorithms presented in Section 3 and Section 4 can be extended to the case in which parallel edges are allowed. Therefore, since Π is a Hamiltonian path in $\mathcal{G}$, we can apply both Theorem 1 and Theorem 2. It follows that, for any given edge $e \in E(H)$, there exist polynomial time algorithms to compute $\text{AUG}_2(H - e, G - e)$. Their complexity is the same as in Theorem 1 and Theorem 2, respectively. $\square$

6 Conclusions

In this paper we have presented time and space efficient algorithms for solving special cases of the classic problem of finding a minimum weight set of edges that has to be added to a spanning subgraph of a given (either unweighted or non-negatively weighted) graph to make it 2-edge connected. These techniques have been applied to solve efficiently an interesting survivability problem on 2-edge connected networks.

For the weighted case, our algorithm is efficient, but it is still open to establish whether its running time is optimal. Besides that, many interesting problems remain open. Among the others, we mention the extension of our problem to the vertex-connectivity case, which is of interest for managing transient vertex failures in 2-vertex connected networks. Moreover, the results contained in Section 5 should be enlarged to the case in which all the possible edge failures in H are considered, trying to get a faster solution than that obtained by merely applying our algorithms $O(|E(H)|)$ times, one for the failure of each vital edge in H .

We consider the last one as the highest-priority open problem, and we plan to attack it by means of ad-hoc amortization techniques. In fact, from a network management point of view, computing *a priori* the augmentation set associated with every edge in the network is essential to know how the network will react in any possible link failure scenario.

Acknowledgements

The authors are very grateful to Peter Widmayer for helpful discussions on the topic.

References

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison Wesley, (1974). 353
2. G. S. Brodal, Worst-case efficient priority queues, *Proc. 7th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'96)*, ACM/IEEE Computer Society, 52–58. 352
3. K. P. Eswaran and R. E. Tarjan, Augmentation problems, *SIAM Journal on Computing*, **5** (1976) 653–665. 346, 347
4. A. Frank, Augmenting graphs to meet edge-connectivity requirements, *SIAM Journal on Discrete Mathematics*, **5** (1992) 25–53. 346
5. G. N. Frederickson and J. Jájá, On the relationship between the biconnectivity augmentation problems, *SIAM Journal on Computing*, **10** (1981) 270–283. 346
6. H. N. Gabow, Application of a poset representation to edge connectivity and graph rigidity, *Proc. 32nd Ann. IEEE Symp. on Foundations of Computer Science (FOCS'91)*, IEEE Computer Society, 812–821. 347
7. A. Galluccio and G. Proietti, Towards a 4/3-approximation algorithm for biconnectivity, IASI-CNR Technical report R. 506, June 1999. Submitted for publication. 346
8. M. Grötschel, C. L. Monma and M. Stoer, Design of survivable networks, in *Handbooks in OR and MS, Vol. 7*, Elsevier (1995) 617–672. 346
9. S. Khuller and R. Thurimella, Approximations algorithms for graph augmentation, *Journal of Algorithms*, **14** (1993) 214–225. 346
10. S. Khuller, Approximation algorithms for finding highly connected subgraphs, in *Approximation Algorithms for NP-Hard Problems*, Dorit S. Hochbaum Eds., PWS Publishing Company, Boston, MA, 1996. 346
11. S. Khuller and U. Vishkin, Biconnectivity approximations and graph carvings, *Journal of the ACM*, **41**(2) (1994) 214–235. 346, 348, 349
12. R. H. Möhring, F. Wagner and D. Wagner, VLSI network design, in *Handbooks in OR and MS, Vol. 8*, Elsevier (1995) 625–712. 346
13. H. Nagamochi and T. Ibaraki, An approximation for finding a smallest 2-edge connected subgraph containing a specified spanning tree, *Proc. 5th Annual International Computing and Combinatorics Conference (COCOON'99)*, Vol. 1627 of Lecture Notes in Computer Science, Springer, 31–40. 346
14. S. Vempala and A. Vetta, Factor 4/3 approximations for minimum 2-connected subgraphs, *Proc. 3rd Int. Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 2000)*, Vol. 1913 of Lecture Notes in Computer Science, Springer, 262–273. 346
15. T. Watanabe and A. Nakamura, Edge-connectivity augmentation problems, *Journal of Computer and System Sciences*, **35**(1) (1987) 96–144. 347
16. P. Winter, Steiner problem in networks: a survey, *Networks*, **17** (1987) 129–167. 346

Algorithms for Pattern Involvement in Permutations

Michael H. Albert¹, Robert E. L. Aldred²,
Mike D. Atkinson¹, and Derek A. Holton²

¹ Department of Computer Science, University of Otago

² Department of Mathematics and Statistics, University of Otago

Abstract. We consider the problem of developing algorithms for the recognition of a fixed pattern within a permutation. These methods are based upon using a carefully chosen chain or tree of subpatterns to build up the entire pattern. Generally, large improvements over brute force search can be obtained. Even using on-line versions of these methods allow for such improvements, though often not as great as for the full method. Furthermore, by using carefully chosen data structures to fine tune the methods, we establish that any pattern of length 4 can be detected in $O(n \log n)$ time. We also improve the complexity bound for detection of a separable pattern from $O(n^6)$ to $O(n^5 \log n)$.

1 Introduction

The relation of “pattern containment” or “involvement” on finite permutations has become an active area of research in both computer science and combinatorics. In computer science pattern containment restrictions are used to describe classes of permutations that are sortable under various conditions [1,5,6,10]. In combinatorics the focus has been more on enumerating permutations under various pattern containment restrictions [7,8,9,11]. In both these areas it is difficult to gather computational data because of the difficulty of testing for pattern containment.

Formally, we say that two sequences are isomorphic (or order isomorphic) if the permutations required to sort them are the same. For example, the two sequences 3, 4, 7, 1 and 5, 7, 9, 2 are isomorphic. We say that one permutation $\sigma = s_1, \dots, s_m$ is *involved* in another permutation $\tau = t_1, \dots, t_n$ when $t_1, \dots, t_n$ has a subsequence that is isomorphic to $s_1, \dots, s_m$. We write $\sigma \preceq \tau$ to express this.

It appears to be a difficult problem to decide if two given permutations σ, τ whether $\sigma \preceq \tau$ and in this generality the problem is NP-complete [2]. In this paper we study the case that σ is fixed, of length k say, and τ is the input to the problem. This is the situation that arises when we wish to run many “ $\sigma \preceq \tau$ ” tests with τ varying and σ fixed. In practice most pattern containment investigations are of this type. A brute force approach which simply examines all subsequences of τ of length k would have a worst case execution time of $O(n^k)$,

where n is the length of τ . Therefore the problem lies in the complexity class P but of course, for all but small values of k , this execution time will generally be unacceptable.

To the best of our knowledge no previous work has been published on improvements to this upper bound. The best implementation of brute force search that we know of is the program `forbid.c` [4] which uses a tree search approach based on the generating trees defined in [11]; in general however this program does not improve the asymptotics of the worst case.

A few particular cases of the problem have been attacked successfully. There is an $O(n \log \log n)$ algorithm for finding the longest increasing subsequence of a given sequence of length n [3] so this solves the problem for the cases $\sigma = 12 \cdots k$. The permutations 132, 213, 231, 312 can all be handled in linear time by stack sorting algorithms. Also, an algorithm of time complexity $O(n^6)$ was given in [2] for the case of an arbitrary separable permutation.

In this paper we develop general algorithms whose worst case complexity is considerably smaller than $O(n^k)$ and we look at a number of cases where even further improvement is possible.

In the next section we set up a general apparatus for searching for the pattern σ . As will be seen we are prepared to invest a considerable amount of time (exponential in k) in preprocessing σ in the expectation that we shall then be able to solve instances of the “does τ involve σ ” problem much faster than by brute force. We identify an integer $c(\sigma)$ that controls the complexity of our recognition algorithm and report on a statistical study that gives insight into the variation of $c(\sigma)$ as σ varies. This study indicates that the algorithms are never worse than $O(n^{2+k/2} \log n)$ (although that remains unproved) and in some cases are considerably better. We give an example to show that $c(\sigma)$ may be much smaller than the apparent worst case and we briefly discuss how the algorithms solve the associated counting problem. The section ends with a glimpse of an even more general family of algorithms and a minor improvement to the algorithm in [2] for detecting a separable permutation.

Section 3 investigates a special case of our general approach. This special case is particularly suitable for ‘on-line’ algorithms (where τ can be scanned once only). Moreover it avoids the expensive preprocessing stage and is simple enough that upper bounds can be proved analytically. We give examples of infinite families of permutations that can be detected quickly by an on-line algorithm.

In the penultimate section we examine permutations σ of length 4. We refine our general approach and describe $O(n \log n)$ algorithms to recognise whether $\sigma \preceq \tau$.

2 A General Recognition Framework

In this section we develop a general algorithm for testing whether $\sigma \preceq \tau$ and illustrate its power by a number of case studies. Throughout, σ is a *fixed* permutation of length k and τ a variable permutation of length n .

Before giving the technical notation we sketch the general idea behind our approach. In the first stage of the algorithm we shall identify a suitable sequence

$$\sigma_0 \preceq \sigma_1 \preceq \sigma_2 \preceq \dots \preceq \sigma_k = \sigma \quad (1)$$

of subsequences of σ (with σ_i of length i). This sequence will be chosen so that the subsequences θ of τ that are isomorphic to one of the σ_i can be defined and used without storing the whole of θ . This stage of the algorithm may have a cost that is exponential in k but it is independent of the input τ and is not repeated.

The second stage of the algorithm identifies all subsequences of τ that are isomorphic to σ_i , for increasing values of i . Because these subsequences are not stored in their entirety this part of the algorithm is of much lower complexity than naive search.

In order to handle subsequences of both σ and τ we represent them as sets of pairs. If $\sigma = s_1 \dots s_k$ is the permutation that maps i to s_i then σ itself will be represented as the set $\mathcal{S} = \{(i, s_i) \mid 1 \leq i \leq k\}$. Every subset of $\mathcal{S}$ defines a subsequence of σ and vice versa. Subsequences of τ are defined similarly as subsets of the set of pairs $\mathcal{T}$ that defines τ itself.

Let π_1 and π_2 be the projections that map a pair to its first and second component (respectively). With this view of subsequences an isomorphism between a subsequence of σ and a subsequence of τ is simply a bijection β between the two corresponding sets of pairs for which the induced maps

$$p = \pi_1(p, v) \mapsto \pi_1(\beta(p, v))$$

$$v = \pi_2(p, v) \mapsto \pi_2(\beta(p, v))$$

are both order preserving. Note that when an isomorphism exists it is unique.

A sequence such as (1) above is then just a sequence of subsets

$$\emptyset = \mathcal{S}_0 \subseteq \mathcal{S}_1 \subseteq \mathcal{S}_2 \subseteq \dots \subseteq \mathcal{S}_k = \mathcal{S} \quad (2)$$

in which each subset $\mathcal{S}_i$ is obtained from the previous one by adding a new pair (a_i, b_i) . For the moment we shall defer the explanation of how to choose these subsets; once we have described how they are used it will be clear how to choose them optimally. Making this choice is the first step of the algorithm.

Let Σ_i denote the set of subsets of $\mathcal{T}$ that are isomorphic to the set $\mathcal{S}_i$. The second stage of the recognition algorithm is described in Algorithm 1. As it stands this is simply another version of brute force search crippled by the worst case size of Σ_i . To make significant improvements we need a way of handling many elements in Σ_i simultaneously. We shall introduce a concise form of an element $\theta \in \Sigma_i$, denoted by $R(\theta)$, called its *registration*. The key idea is to process all elements with the same registration simultaneously.

Before giving the technical definition of $R(\theta)$ it will be helpful to consider an example. Let us suppose that $\mathcal{S}_3 = \{(2, 4), (5, 3), (3, 6)\}$ and that $\mathcal{S}_4 = \mathcal{S}_3 \cup \{(4, 5)\}$. Suppose also that we have some subsequence $\theta = \{(14, 4), (6, 9), (10, 15)\}$ of τ . Because of the bijection

$$(2, 4) \mapsto (6, 9), (5, 3) \mapsto (14, 4), (3, 6) \mapsto (10, 15)$$

Algorithm 1 Basic form of the recognition algorithm

```

for  $i := 0$  to  $k - 1$  do
  for each  $(p, v) \in \mathcal{T}$  and each  $\theta \in \Sigma_i$  do
    if  $\theta \cup \{(p, v)\}$  is isomorphic to  $\mathcal{S}_{i+1}$  then
      add it to  $\Sigma_{i+1}$ 
    end if
  end for
end for

```

we see that $\mathcal{S}_3 \cong \theta$. The necessary and sufficient condition that this bijection can be extended to an isomorphism between $\mathcal{S}_4$ and $\theta \cup (p, v)$ is $10 < p < 14$ and $9 < v < 15$.

The point of this is that the isomorphism test depends only on 4 items of data rather than the entire 6 items. In this small case the saving is not very dramatic but it is enough to illustrate the general idea. In general, if $\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{(a, b)\}$ we identify in $\mathcal{S}_i$ the two first components which most closely enclose a and the two second components which most closely enclose b . The corresponding items in θ then define the range in which p and v must lie in order that $\theta \cup \{(p, v)\} \cong \mathcal{S}_{i+1}$. Notice that this idea still applies if a (respectively b) is smaller than or greater than any first (respectively second) component; in such a case the range in which p or v must lie is unbounded on one side.

In practice we often need to store considerably more than just these four enclosing components since we must anticipate being able to test isomorphisms with subsets of size greater than $i + 1$. To describe precisely what needs to be stored at each stage we define the *registration type* $r(\mathcal{S}_i)$ of each $\mathcal{S}_i$ as

$$r(\mathcal{S}_i) = (P_i, V_i)$$

where

$$P_i = \{j \in \pi_1(\mathcal{S}_i) \mid \text{either } j + 1 \notin \pi_1(\mathcal{S}_i) \text{ or } j - 1 \notin \pi_1(\mathcal{S}_i)\}$$

and

$$V_i = \{j \in \pi_2(\mathcal{S}_i) \mid \text{either } j + 1 \notin \pi_2(\mathcal{S}_i) \text{ or } j - 1 \notin \pi_2(\mathcal{S}_i)\}$$

Lemma 1. 1. If w, x are the two symbols in $\pi_1(\mathcal{S}_i)$ which most closely enclose a_{i+1} (as $w < a_{i+1} < x$) then $w, x \in P_i$. Similarly, if y, z are the two symbols in $\pi_2(\mathcal{S}_i)$ which most closely enclose b_{i+1} then $y, z \in V_i$.
 2. $P_{i+1} \subseteq P_i \cup \{a_{i+1}\}$ and $V_{i+1} \subseteq V_i \cup \{b_{i+1}\}$.

Proof. For the first part, if $w \notin P_i$ then $w - 1$ and $w + 1 \in \pi_1(\mathcal{S}_i)$. Therefore $w + 1$ and x are a closer enclosing pair for a_{i+1} , a contradiction. The other statements follow similarly. The second part follows from the definition of P_i and V_i .

We shall see later that the sizes of P_i and V_i determine the complexity of a refined version of the algorithm above for recognising whether $\sigma \preceq \tau$. The following example is meant to illustrate that “clever” choices of the sets $\mathcal{S}_i$ can control these values effectively.

Example 2. Suppose that $k = 4m$ and consider the permutation:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & \cdots & 2m-1 & 2m & \cdots & 4m \\ 1 & 2m & 4m-1 & 2m-2 & 4m-3 & \cdots & 2m+3 & 2 & \cdots & 2m+2 \end{pmatrix}$$

(in the second row, the odd values decrease by 2 each time, cyclically, beginning from 1, while the even ones decrease by 2 each time beginning from $2m$). If we simply take the set $\mathcal{S}_i$ to consist of the pairs making up the first i columns of this permutation then although $|P_{2m-1}| = 1$, we have $|V_{2m-1}| = 2m-1$. On the other hand, by simply reordering the pairs as follows:

$$\begin{pmatrix} 1 & 2m+1 & 2 & 2m+2 & 3 & 2m+3 & 4 & 2m+4 & \cdots \\ 1 & 2m+1 & 2m & 4m & 4m-1 & 2m-1 & 2m-2 & 4m-2 & \cdots \end{pmatrix}$$

we obtain an arrangement where $|P_i| \leq 3$ and $|V_i| \leq 4$ for all i .

The registration type $r(\mathcal{S}_i)$ specifies how much of each $\theta \in \mathcal{S}_i$ should be stored by the algorithm. The part that is stored is called the registration of θ and is denoted by $R(\theta)$: $R(\theta)$ is the image of $r(\mathcal{S}_i)$ under the natural maps induced by the order isomorphism between θ and $\mathcal{S}_i$. Let $R_i = \{R(\theta) \mid \theta \in \Sigma_i\}$.

The second stage of the recognition algorithm, incorporating the concept of registration, is specified in Algorithm 2.

Algorithm 2 Recognition algorithm with registration

```

 $\mathcal{R}_i := \emptyset$  { $\mathcal{R}_i$  holds elements of  $R_i$ ; initially none are known}
for  $i := 0$  to  $k-1$  do
  for each  $(p, v) \in \mathcal{T}$  and each  $R(\theta) \in \mathcal{R}_i$  do
    Let  $w, x, y, z$  be defined as in Lemma 1
    Let  $w_\theta, x_\theta, y_\theta, z_\theta$  be the elements of  $R(\theta)$  that correspond to  $w, x, y, z$ 
    if  $w_\theta < p < x_\theta$  and  $y_\theta < v < z_\theta$  then {hence  $\phi = \theta \cup \{(p, v)\} \in \Sigma_{i+1}$ }
      compute  $R(\phi)$  and insert it in  $\mathcal{R}_{i+1}$ 
    end if
  end for
end for

```

Proposition 3. *When the second stage of the algorithm terminates we have $R_i = \mathcal{R}_i$. In particular $\sigma \preceq \tau$ if and only if $|\mathcal{R}_k| > 0$.*

Proof. Notice that Lemma 1 guarantees that $w_\theta, x_\theta, y_\theta, z_\theta$ are present in $R(\theta)$ and that all the symbols needed to compute $R(\phi)$ are available either from $R(\theta)$ itself or from $\{p, v\}$. Note also that the stipulated ranges for p and v are precisely those for which the isomorphism between $\mathcal{S}_i$ and θ can be extended to an isomorphism between $\mathcal{S}_{i+1}$ and ϕ . Therefore each $R(\phi)$ is an element of R_i . Finally, observe that every $R(\phi) \in R_i$ is computed by the algorithm; this follows by induction on i .

Next we discuss the run-time of the algorithm. The outer ‘for’ loop is executed k times but k is independent of n . In a typical iteration of the inner ‘for’ loop we have to consult each of the n pairs of T and each $R(\theta) \in \mathcal{R}_i$. The computation that is done with a typical (p, v) and $R(\theta)$ incurs a cost that depends on how the sets $\mathcal{R}_i$ are stored; by standard data structuring devices we can contain each of the operations that access and update the set $\mathcal{R}_i$ to a time $O(\log |\mathcal{R}_i|)$.

Taking all this into account the total cost of this stage of the algorithm is $O(n \max_i |\mathcal{R}_i| \log |\mathcal{R}_i|)$.

The elements of $\mathcal{R}_i$ are sequences of integers in the range $1..n$ and the length of these sequences is $|P_i| + |V_i|$. It will therefore be advantageous to keep $|P_i| + |V_i|$ as small as possible. To this end we define

$$c(\sigma) = \min_i \max_i (|P_i| + |V_i|)$$

where the minimum is taken over all orderings of the pairs of $\mathcal{S}$. This discussion has proved:

Proposition 4. *If the ordering on $\mathcal{S}$ is chosen so as to minimise $\max_i (|P_i| + |V_i|)$ the second stage of the algorithm requires time $O(n^{1+c(\sigma)} \log n)$.*

It is now evident what the first stage of the algorithm must do: find the ordering on $\mathcal{S}$ that minimises $\max_i (|P_i| + |V_i|)$. The cost of this first stage of the algorithm will be independent of n so it will not contribute to the asymptotic upper estimate of the time complexity. Nevertheless it is not easy to compute the optimal ordering of $\mathcal{S}$; we have found a method based on a shortest path algorithm to do this in $O(2^k)$ steps.

Statistics

We have generated some thousands of random permutations of degrees up to 17 and computed $c(\sigma)$. In all of them we have found that $c(\sigma) \leq 1 + k/2$. The following table summarizes the values of $c(\sigma)$ for samples of random permutations of lengths 8 through 17. In each row, we exhibit the number of permutations observed for each value of $c(\sigma)$ (blanks denoting no observations), and we also provide an example of a particular permutation from the sample which achieved the maximum observed value of $c(\sigma)$. In these examples, two digit numbers 10, 11, ..., are denoted by the letters $A, B, \dots$.

k	$c(\sigma)$									Example
8			157	769	74					52814673
9	1	57	682	260						921745863
10		13	469	515	3					72936A5184
11		1	262	676	61					B52A7481639
12			126	662	212					72A419B538C6
13			48	533	412	7				B4159D6C2A738
14			23	365	582	30				3E68BC41927A5D
15			3	55	201	41				2D7CE5A6913F84B
16			1	38	167	83	1			A4F51C9G7D3B82E6
17				25	145	105	5			6AFDH842G93EC751B

Counting

Our general approach can easily be modified to count the number of occurrences of σ in τ . The only change we have to make is to keep, with every $\chi \in R_i$, an integer t_χ which records the number of θ for which $R(\theta) = \chi$. Then, whenever we detect that $\phi = \theta \cup \{(p, v)\} \in \Sigma_{i+1}$ we increment t_ω by t_χ , where $\omega = R(\phi)$ and $\chi = R(\theta)$.

A more general algorithm

Our general paradigm can be extended so that (2) is replaced by a ‘union’ tree. The leaves of the trees are labelled by singleton pairs and the internal nodes are labelled by subsets of $\mathcal{S}$ which are the disjoint unions of the subsets labelling their subtrees. The recognition algorithm processes the nodes of the tree in any order so long as each node is processed after its subtree nodes. To process a node labelled by a subset $\mathcal{U}$ of $\mathcal{S}$ means to find (and store implicitly by registration) all the subsets of $\mathcal{T}$ isomorphic to $\mathcal{U}$. When we do this we shall have available the corresponding information for the subtrees.

In order to get a comparatively efficient algorithm it will be necessary to have short registrations (as we have previously discussed). But the registration information has to be sufficient that we can successfully recognise the subsets of $\mathcal{T}$ isomorphic to $\mathcal{U}$. It is clearly going to be complicated to examine all possible union trees (although independent of n of course) so we have yet to explore the full potential of this idea. Nevertheless we offer one example of the power of this approach in the following sketch which improves on the algorithm given in [2].

Suppose that σ is any separable permutation. By definition σ may be written as a concatenation $\sigma = \alpha\beta$ where either every entry of α is less than every entry of β or every entry of α is greater than every entry of β ; moreover, α, β are isomorphic to separable permutations. Then $\mathcal{S}$ can be written as a union $\mathcal{L} \cup \mathcal{M}$ where every member of $\pi_1(\mathcal{L})$ is less than every member of $\pi_1(\mathcal{M})$, and where either every member of $\pi_2(\mathcal{L})$ is less than every member of $\pi_2(\mathcal{M})$ (the positive case) or every member of $\pi_2(\mathcal{L})$ is greater than every member of $\pi_2(\mathcal{M})$ (the negative case). The sets $\mathcal{L}$ and $\mathcal{M}$ are structured in a similar fashion and so we have a natural way of defining a union tree for $\mathcal{S}$. The nodes of this tree are positive or negative according to how they were defined.

The registration type of a node $\mathcal{U}$ is a quadruple that we structure as an ordered pair of ordered pairs $((m_1, m_2), (M_1, M_2))$ where m_1 and m_2 are the minimum values in $\pi_1(\mathcal{U})$ and $\pi_2(\mathcal{U})$, and M_1 and M_2 are the maximum values. Thus the registration of a subset of $\mathcal{T}$ isomorphic to $\mathcal{U}$ is the quadruple which corresponds to the registration type under the isomorphism. It follows that each node is associated with at most n^4 registrations.

The central problem is to compute the set of registrations $R(\mathcal{U})$ at a node $\mathcal{U}$ given the registration sets $R(\mathcal{V}), R(\mathcal{W})$ for the child nodes $\mathcal{V}, \mathcal{W}$. For definiteness assume that $\mathcal{U}$ is a positive node (the negative case is similar). A quadruple $((m_1, m_2), (M_1, M_2))$ is easily seen to belong to $R(\mathcal{U})$ if and only if there exist pairs (a, b) and (c, d) for which

$$((m_1, m_2), (a, b)) \in R(\mathcal{V}) \text{ and } ((c, d), (M_1, M_2)) \in R(\mathcal{W}) \text{ and } (a, b) < (c, d)$$

To compute these quadruples we proceed as follows. First, for every (m_1, m_2) we search $R(\mathcal{V})$ and determine the set of all (w, x) for which $((m_1, m_2), (w, x)) \in R(\mathcal{V})$ and we find the set P_{m_1, m_2} of all minimal pairs in this set. Since the pairs of P_{m_1, m_2} are incomparable we can order them increasingly by first component and have the second components decrease. Next, for every (M_1, M_2) we search $R(\mathcal{W})$ and determine the set of all (y, z) for which $((y, z), (M_1, M_2)) \in R(\mathcal{W})$ and we find the set Q_{M_1, M_2} of all maximal pairs in this set. Again, the pairs of Q_{M_1, M_2} are incomparable so we can order them increasingly by first component and have the second components decrease.

Now, for each $((m_1, m_2), (M_1, M_2))$ we have to test whether there exist pairs $(w, x) \in P_{m_1, m_2}$ and $(y, z) \in Q_{M_1, M_2}$ for which $(w, x) < (y, z)$. Since the components are ordered as explained above this test can be made in time $O(n \log n)$. As there are $O(n^4)$ quadruples each node requires time $O(n^5 \log n)$. There are k nodes in all so the total time is still $O(n^5 \log n)$.

3 On-Line Algorithms

In this section we study a simpler version of the algorithm presented in the previous section. This simpler version avoids the preprocessing stage (which was exponential in k) while the second stage may be somewhat slower (but still provably better than brute force search). The resulting algorithm only has to scan the input τ once and so is referred to as an ‘on-line’ algorithm.

The simplification is to take the ordering of $\mathcal{S}$ in which the first components come in the order $1, 2, \dots, k$. The order in which the second components come is then, by definition, the sequence $s_1, \dots, s_k$ in the original description of σ and, indeed, the entire algorithm can be presented in the more familiar setting of subsequences of images of σ and τ . Furthermore the first components of registrations need not be kept (indeed it is easily seen that $P_i = \{i\}$) since we shall be processing $\tau = t_1 \dots t_n$ in left to right order. This form of recognition is described in Algorithm 3.

Algorithm 3 On-line form of the recognition algorithm

```

for  $j := 1$  to  $n$  do
  for  $i := 0$  to  $k - 1$  do
    for each  $R(\theta) \in R_i$  do
      Let  $y, z$  be defined as in Lemma 1
      Let  $y_\theta, z_\theta$  be the elements of  $R(\theta)$  that correspond to  $y, z$ 
      if  $y_\theta < t_j < z_\theta$  then  $\{\phi = \theta t_j \in \Sigma_{i+1}\}$ 
        add  $R(\phi)$  to  $R_{i+1}$ 
      end if
    end for
  end for
end for

```

We define $d(\sigma) = \max |V_i|$. Arguing as in the previous section the execution time of this algorithm is $O(n^{1+d(\sigma)} \log n)$.

Lemma 5. $c(\sigma) - 1 \leq d(\sigma) \leq 2k/3$

Proof. Observe that V_i does not contain 3 consecutive values $j-1, j, j+1$ since, by definition, the condition $j \in V_i$ implies that one of $j-1$ and $j+1$ does not belong to $\pi_2(\mathcal{S}_i)$ and so does not belong to V_i . So, in each triple $3j-2, 3j-1, 3j$, at most two members can belong to V_i and the result follows.

Corollary 6. *The decision problem $\sigma \preceq \tau$ can be solved in time $O(n^{1+2k/3} \log n)$. More generally, the number of occurrences of σ as a pattern within τ can be computed in this time bound.*

Proof. The algorithm above stores registrations $R(\theta)$ where θ is now a subsequence of τ that is isomorphic to some σ_i . The registration is a subsequence of θ with enough information that we can determine whether θt_j is isomorphic to σ_{i+1} . As we saw in the previous lemma $|R(\theta)| \leq 2k/3$ and the results follow as in the previous section.

In many cases the upper bound given in the corollary can be greatly improved since $d(\sigma)$ is smaller than the upper bound in Lemma 5. In addition, we can often exploit special information about σ that is unavailable in general. As an example of such analyses we consider some classes of permutations σ for which very significant improvements can be made. These classes are ‘closed sets’ in the sense of [1] defined by their avoiding particular permutations. In general, let $A(\omega_1, \omega_2, \dots)$ denote the set of permutations which do not involve any of $\omega_1, \omega_2, \dots$.

We begin by considering the set $A(132, 312)$. It is easily seen that a permutation σ belongs to this set if the values of any initial segment of σ form a single interval. Equivalently, any initial segment of σ ends with its minimum or maximum value.

Proposition 7. *If $\sigma \in A(132, 312)$ then $d(\sigma) \leq 2$.*

Proof. The result is almost immediate from the preceding description of $A(132, 312)$. Since the initial segment of length i consists of an interval of values, V_i simply consists of the endpoints of that interval, or only one of the endpoints if 1 or k already occur among the first i positions, and hence has size at most 2. Thus $d(\sigma) = \max_i |V_i| \leq 2$.

This proposition establishes that there is an $O(n^3 \log n)$ algorithm for recognising whether $\sigma \preceq \tau$ when $\sigma \in A(132, 312)$. In fact, by a small modification of the registration procedure we can reduce the complexity of this algorithm to $O(n^2 \log n)$.

With notation as in the proposition above, consider the elements of V_i as pairs (a, b) representing the lower and upper endpoints of the corresponding

interval. In the naive version of the algorithm we might well register two such pairs (a, b) and (a', b') where

$$a' < a < b < b'.$$

In this case the pair (a', b') can never be useful in the recognition of σ , since any extensions which they allow will also be allowed by the (a, b) pair.

It follows that the registration information which we need to store for V_i can be thought of as a sequence of pairs $(a_1, b_1), (a_2, b_2), \dots, (a_j, b_j)$ where

$$\begin{aligned} a_1 < a_2 < \dots < a_j \quad \text{and} \\ b_1 < b_2 < \dots < b_j \end{aligned}$$

In particular there can be at most n such pairs, rather than the $O(n^2)$ which are budgeted for in the standard on-line algorithm. This modification reduces the time complexity as claimed. It transpires that a further reduction to $O(n \log n)$ is possible by the use of the data structures mentioned in the following section.

To within order isomorphism there are only three other infinite sets defined by two length 3 restrictions. By analysing the structure of the permutations σ in these classes we can prove fairly easily

Proposition 8. *If σ is a permutation that satisfies at least two length 3 restrictions then $d(\sigma) \leq 3$.*

4 Permutations of Length 4

We consider now the problem of finding efficient algorithms for recognising whether $\sigma \leq \tau$ in all cases where $|\sigma| = 4$. At first it seems that there are 24 individual problems of this type to be solved. However, the operations of: reversing a permutation; taking the complement of a permutation; and taking the inverse of a permutation, all respect the ordering $\preceq$, and can be carried out in $O(n \log n)$ time. So, if we can find an efficient algorithm for σ we also have one for its reverse, complement, etc. This reduces the number of cases that we need to consider in the present instance to 7, exemplified by:

$$\sigma = 1234, 2134, 2341, 2314, 1324, 2143, 2413.$$

In the first two cases $d(\sigma) = 1$ and so the on-line algorithms are of complexity $O(n^2 \log n)$. In both cases, and in general when $d(\sigma) = 1$, this is easily reduced to $O(n \log n)$. This is accomplished by storing the registration information R_i as a sorted list in such a way that we can search and insert in $O(\log n)$ time.

In the remaining cases $d(\sigma) = 2$, and so the on-line algorithms are of complexity $O(n^3 \log n)$. As in the case of $A(132, 312)$ though it is possible to “prune” the registration information when it consists of pairs, to a set of size $O(n)$, and thereby gain an improvement in the running time of the algorithm to $O(n^2 \log n)$. In fact, in each case the running time can be reduced to $O(n \log n)$. To accomplish this, requires the use of a tree-based data structure which permits answering queries of a form similar to:

What is the smallest $y > x$ which occurred between position q and the present position?

for arbitrary parameters x and q , in $O(\log n)$ time. The methods in each case are similar, and so for compactness we will present here a complete exposition only for the case 1324. In this case a permutation $x_1x_2x_3 \dots x_n$ will be processed sequentially, and for expository purposes it will be convenient to think of the position of a particular value in this permutation as its “time of arrival”, allowing for a clean distinction between positions and values.

We construct a binary tree, making it as as balanced as possible, whose vertices are labelled with subintervals of $\{1, 2, \dots, n\}$, (or rather with the endpoints of such intervals), and which also carry a key (of which more presently). The leaves are labelled left to right with the singleton intervals $\{1\}$, $\{2\}$, etc. It will be convenient, though not absolutely necessary, to be able to access the leaves directly via an array. The interval at a node is simply the union of its children’s intervals. Note that at each level the intervals of the vertices at that level form an ordered partition of $\{1, 2, \dots, n\}$ (that is, the left to right ordering of the intervals within a level is the same as that of the vertices.) Each node also has a pointer to its “rightward parent”. For left children this is the ordinary parent, but for right children it is the node immediately to the right of its real parent, at the same level of the tree as its parent node. All the keys are initially set to 0. We also reserve space for an array *min* indexed from 1 through n , and a single element b , initialised to $n + 1$.

Suppose that we have proceeded to a certain point p , the present time, without detecting a 1324. The assumptions we make about the data structures that we have to this point are the following:

- If a vertex v of the tree has key value $t \neq 0$, then t was the latest arrival, until now of an element whose value lies in the interval associated with v . A key value of 0 indicates that no such element has yet arrived.
- The value of b is the minimum value of a 3 occurring in a 132 pattern until now.
- The values $\text{min}(s)$ (for $1 \leq s < p$) are the minimum arrivals in times 1 through s for $s < p$.

If the present symbol, x , is larger than b we halt, indicating that a 1324 pattern exists. Otherwise we must update the data structures to maintain the properties specified above. Updating *min* is trivial.

We now proceed to update b (this step can be skipped if x was actually the minimum). To do this we need to find the smallest y such that there is a 132 pattern zyx , and replace the present b with $\min(b, y)$. To find this y we first find the earliest z such that zx is a 12 pattern. This can be accomplished in $O(\log n)$ time using binary search in the array *min*. Suppose that z arrived at time q . Now we must find “the smallest $y > x$ that arrived between time q and the present”. The tree structure allows us to resolve such problems in $O(\log n)$ time as follows.

Begin at the leaf $\{x + 1\}$. Examine the key value here. If it is greater than q then $y = x + 1$ forms a suitable 132 and we stop. If not, proceed to its right parent. Continue up the tree moving to a right parent each time until a key value greater than q is found (if no such key is found then no 132 exists involving x as the 2). Now descend back down the tree binary-search style, to find the left-most leaf containing such a key. This is the minimal y for which zyx is a 132 pattern. All this can be accomplished in $O(\log n)$ steps.

Finally, update the tree with the information from the present symbol. This is done simply by traversing the branch leading up from $\{x\}$ and replacing the key at each vertex visited by p .

5 Summary and Conclusions

We have shown that significant improvements over brute force search are available for the problem of detecting the involvement of a fixed permutation σ within another permutation τ . Fine tuning these procedures in specific cases generally provides further reductions in the complexity.

As the general problem of detecting the involvement of one permutation within another is NP-complete, there is presumably no hope of achieving a polynomial algorithm which does not require σ to be fixed. However, the question:

For a fixed permutation σ , what is the least constant k_σ such that there is a σ -recognition algorithm of complexity $O(n^{k_\sigma} \log n)$?

remains open. It is intriguing to speculate that there might be an absolute upper bound k for these constants k_σ .

References

1. M. D. Atkinson: Restricted permutations, *Discrete Math.* 195 (1999), 27–38. [355](#), [363](#)
2. P. Bose, J. F. Buss, A. Lubiw: Pattern matching for permutations, *Inform. Process. Lett.* 65 (1998), 277–283. [355](#), [356](#), [361](#)
3. M.-S. Chang, F.-H. Wang: Efficient algorithms for the maximum weight clique and maximum weight independent set problems on permutation graphs, *Inform. Process. Lett.* 43 (1992), 293–295. [356](#)
4. O. Guibert: Personal communication. [356](#)
5. D. E. Knuth: *Fundamental Algorithms, The Art of Computer Programming* Vol. 1 (First Edition), Addison-Wesley, Reading, Mass. (1967). [355](#)
6. V. R. Pratt: Computing permutations with double-ended queues, parallel stacks and parallel queues, *Proc. ACM Symp. Theory of Computing* 5 (1973), 268–277. [355](#)
7. R. Simion, F. W. Schmidt: Restricted permutations, *Europ. J. Combinatorics* 6 (1985), 383–406. [355](#)
8. Z. E. Stankova: Forbidden subsequences, *Discrete Math.* 132 (1994), 291–316. [355](#)
9. Z. E. Stankova: Classification of forbidden subsequences of length 4, *European J. Combin.* 17 (1996), no. 5, 501–517. [355](#)

10. R. E. Tarjan: Sorting using networks of queues and stacks, *Journal of the ACM* 19 (1972), 341–346. 355
11. J. West: Generating trees and forbidden sequences, *Discrete Math.* 157 (1996), 363–374. 355, 356

A Fast Algorithm for Enumerating Bipartite Perfect Matchings

Takeaki Uno

Foundations of Informatics Research Division, National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
uno@nii.ac.jp

Abstract. In this paper, we propose an algorithm for enumerating all the perfect matchings included in a given bipartite graph $G = (V, E)$. The algorithm is improved by the approach which we proposed at ISAAC98. Our algorithm takes $O(\log |V|)$ time per perfect matching while the current fastest algorithm takes $O(|V|)$ time per perfect matching.

Keyword: enumeration, enumerating algorithm, perfect matching.

1 Introduction

Enumeration is a fundamental problem for optimization, data bases, decision making, and many other scientific problems. Numerous problems are solved, or investigated by enumerating related objects. Therefore, enumeration algorithms need to be intensively analyzed in order to find ways to solve these problems.

At ISAAC'98, we proposed a new approach for speeding up enumeration algorithms. Currently, there had been only few studies on speeding up enumeration algorithms. Almost all their techniques are depend on the structures of their problems, hence their techniques can not be applied to other algorithms immediately. Those algorithms often use data structures, which is also make the improvement difficult to be generalized. Our approach, which we named “trimming and balancing,” is a general method for speeding up enumeration algorithms. It is not depend on structures of problems, and does not rely on data structures. Therefore, by using the approach, we can speedup several algorithms which we can not with the existing methods. In this paper, we speed up an algorithm for enumerating bipartite perfect matching by using the approach.

Let $G = (V = V_1 \cup V_2, E)$ be an undirected bipartite graph with vertex sets V_1 and V_2 and an edge set composed of edges in $V_1 \times V_2$. A matching M of the graph G is an edge set such that no two edges of M share their endpoints. If all vertices of G are incident to some edges of a matching M , then we say that M is a *perfect matching*. Let N be the number of perfect matchings in G . We consider the problem of enumerating all the perfect matchings in a given bipartite graph.

For this problem, some algorithms have been proposed. In 1993, K. Fukuda and T. Matsui proposed an enumeration algorithm [1]. The running time of

the algorithm is $O(|V|^{1/2}|E| + N(|E| + |V|))$ time. In 1997, we proposed an algorithm [3] running in $O(|V|^{1/2}|E| + N|V|)$ time. Our algorithm in this paper reduces the time complexity to $O(|V|^{1/2}|E| + N \log |V|)$ time.

In the next section, we explain the framework of “trimming and balancing.” In Section 3, we explain the basic algorithm arising from Fukuda and Matsui’s algorithm, and we describe our improvement in section 4.

2 Approach for Speeding Up Enumeration Algorithms

This section explains our approach, which we proposed at ISAAC 98. Here, we omit the details and proofs. Readers should refer [4,5]. The approach uses an amortized analysis. The analysis bounds time complexities of enumeration algorithms with two parameters. Since decrease of these two parameters result smaller time complexities, the goal of the approach is to improve algorithms to get small parameters. The way of improvement is to add two phases to each iteration of the algorithms, which decreases each parameter, respectively.

Firstly, we explain the amortized analysis. Consider enumeration algorithms based on recursive. For a given enumeration algorithm and its input, we define the *enumeration tree* by $\mathcal{T} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of all iterations occurring in the algorithm, and an edge of $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ connects two vertices iff one of them occurs in the other. In this paper, we define an iteration by computation in a recursive call excluding the computation in recursive calls occurring in the recursive call. For a vertex v of a tree, let $D(v)$ be the set of descendants of v , $Ch(v)$ be the set of children of v . For a vertex $x \in \mathcal{V}$, we denote the computation time in x by $t(x)$, and define $\hat{t}(\mathcal{T}) = \max_{x \in \mathcal{T}} \{t(x)/|D(x)|\}$.

The idea of the amortized analysis is to distribute the computation time of an iteration x to all the children of x such that each children y receives computation time proportional to $t(y)$ or $|D(y)|$. This is for the balance of amount of computation time which the descendants of children receive. This distribution almost amortizes computation time of iterations. By adding several modifications to this idea, we can avoid the bad cases, and can state that the sum of computation time in an enumeration tree $\mathcal{T}$ is $O(\hat{t}(\mathcal{T})x^*(\mathcal{T}))$ per iteration. Here $x^*(\mathcal{T})$ is a parameter of $\mathcal{T}$ which is bounded by the following ways.

Let $\mathcal{P}$ be the set of paths of $\mathcal{T}$ from the root to a leaf, and $\alpha > 1$ be a constant number. $x^*(\mathcal{T})$ is less than or equal to the maximum number of vertices in a path $P \in \mathcal{P}$ satisfying $T(x) > \frac{\alpha-1}{\alpha} \sum_{u \in Ch(x)} T(u)$.

This is a result of [4,5]. From this, we can get the following lemma.

Lemma 1. *If the enumeration tree satisfies the following conditions for a constant c , then $x^*(\mathcal{T}) = O(\log_{c/(c-1)} t(x_0))$.*

(1) $t(x) \geq t(y)$ for any child y of a vertex x

- (2) If a vertex w satisfies $t(w) < 4c^2$, then $|D(w)|$ is constant.
- (3) If a vertex w satisfies $t(w) \geq 4c^2$, then $Ch(w)$ can be split into two subset $Ch_1(w)$ and $Ch_2(w)$ such that $\sum_{u \in Ch_1(w)} t(u)$, $\sum_{u \in Ch_2(w)} t(u) \geq (1/c)t(w) - c$ satisfies.

Proof. We set $\alpha = 2c + 1$. On vertex w satisfying $t(w) > \frac{\alpha-1}{\alpha} \sum_{u \in Ch(w)} t(u)$, $\frac{2c+1}{2c}t(w) > \sum_{u \in Ch_1(w)} t(u) + \sum_{u \in Ch_2(w)} t(u)$ holds. Hence, from the assumption (3), we have

$$\begin{aligned} \sum_{u \in Ch_2(w)} t(u) &\leq \frac{2c+1}{2c}t(w) - \sum_{u \in Ch_1(w)} t(u) \\ &\leq \frac{2c+1}{2c}t(w) - \frac{2}{2c}t(w) + c \\ &\leq \frac{2c-1}{2c}t(w) + \frac{t(w)}{4c} \\ &= \frac{4c-1}{4c}t(w). \end{aligned}$$

Similarly, we have $\sum_{u \in Ch_1(w)} t(u) \leq \frac{4c-1}{4c}t(w)$. Hence, we get $t(u) \leq \frac{4c-1}{4c}t(w)$, for any child u of w . From the assumption (2), there are at most constant number of vertices satisfying $t(w) < 4c^2$ on any path $P \in \mathcal{P}$. Hence, P has at most $\log_{4c/(4c-1)} t(x_0) + O(1)$ vertices x satisfying $T(x) > \frac{\alpha-1}{\alpha} \sum_{u \in Ch(x)} T(u)$.

Therefore, $x^*(\mathcal{T}) = O(\log_{c/(c-1)} t(x_0))$. $\square$

From this, we can improve the algorithm by decreasing $\hat{t}(\mathcal{T})$ and bounding $x^*(\mathcal{T})$ with the three conditions of the lemma. For this purpose, our approach “trimming and balancing” does these by adding two phases. The first phase “trimming phase” reduces the input, i.e., removes unnecessary parts from the inputs, to decrease $t(x)$ so that the order of $\hat{t}(\mathcal{T})$ is reduced. The second phase “balancing phase” balance the size of subproblems so that each subproblem y has not so small size after the trimming phase, to satisfy the conditions of the lemma. We describe the framework of trimming and balancing approach.

Algorithm ENUMERATION_INIT (X)

Step 1: $X :=$ trimming phase (X)

Step 2: Call ENUMERATION (X)

Algorithm ENUMERATION (X)

Step 1: For $i := 1$ to (the number of subproblems)

Step 2: Generate the input X_i of subproblem i by balancing phase

Step 3: $X_i :=$ trimming phase to (X_i)

Step 4: Call ENUMERATION (X_i)

Step 5: End for

3 An Algorithm for Perfect Matchings

In this section, we explain the basic algorithm arising from Fukuda and Matsui's algorithm[1]. In the next section, we improve this algorithm by “trimming and balancing” approach. For a given bipartite graph $G = (V_1 \cup V_2, E)$, we denote the set of all the perfect matchings in G by $\mathcal{M}(G)$. For an edge subset E' , let $G \setminus E'$ be the graph obtained by deleting all the edges of E' from G . The algorithm utilizes the following properties to enumerate perfect matchings.

Property 1. Let E_1 and E_2 be edge sets such that $E_1 \cup E_2$ is the set of edges incident to a vertex v , and $E_1 \cap E_2 = \emptyset$. Then, $\mathcal{M}(G \setminus E_1) \cap \mathcal{M}(G \setminus E_2) = \emptyset$. and $\mathcal{M}(G \setminus E_1) \cup \mathcal{M}(G \setminus E_2) = \mathcal{M}(G)$.

Proof. A perfect matching M of G including an edge of E_1 is a perfect matching of $G \setminus E_2$ and vice versa. A perfect matching M of G including an edge of E_2 is a perfect matching of $G \setminus E_1$ and vice versa. M includes exactly one edge of $E_1 \cup E_2$, hence the statement holds. $\square$

By using this property, the enumeration problem can be partitioned into two subproblems of $G \setminus E_1$ and $G \setminus E_2$, if both $G \setminus E_1$ and $G \setminus E_2$ include a perfect matching, respectively. $G \setminus E_i$ has a perfect matching iff a perfect matchings M satisfy $M \cap E_i = \emptyset$. Hence, we find two distinct perfect matchings M and M' , and set E_1 and E_2 so that E_1 includes an edge $e \in M \setminus M'$ and E_2 includes an edge $e \in M' \setminus M$.

A perfect matching M can be found in $O(|V|^{1/2}|E|)$ time [2]. To find another perfect matching M' , we use *alternating cycles*. For a perfect matching M and a cycle C , if any two edges in $C \setminus M$ are not adjacent, then we call C an *alternating cycle*. In an alternating cycle, edges of M and edges not in M appear alternatively. By exchanging edges along an alternating cycle, we can obtain a perfect matching different from M . Alternating cycles satisfy the following condition [1].

Property 2. For a perfect matching M , there exists another perfect matching iff there exists an alternating cycle. $\square$

To find alternating cycles, we utilize a directed graph $DG(G, M)$ defined for a graph G and a matching M . The vertex set of $DG(G, M)$ is given by V . The arc set of $DG(G, M)$ is given by orienting edges of M from V_1 to V_2 , and edges of $E \setminus M$ in the opposite direction. For any directed cycle C in the graph $DG(G, M)$, arcs of M and the other arcs appear alternatively in the cycle of G corresponding to C . Hence, we can find an alternating cycle by finding a directed cycle of $DG(G, M)$. For conciseness, we treat an edge (u, v) (or (v, u)) of G and an arc (u, v) of $DG(G, M)$ as the same object, for example, arcs of $DG(G, M)$ which are included in M means arcs of $DG(G, M)$ corresponding to the edges of M .

By using these properties, we can construct the following enumeration algorithm. We note that we do not need to find a perfect matching in each iteration since we give M or M' to subproblems when we generate recursive calls.

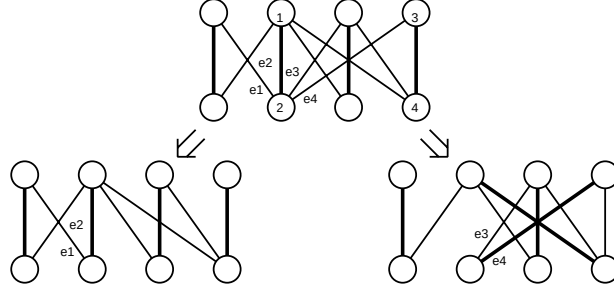


Fig. 1. An example of partitioning a problem: E_1 is composed of e_1 and e_2 , and E_2 is composed of e_3 and e_4 . M' is obtained from M with an alternating cycle $(1,2,3,4)$

ALGORITHM BASIC_ALGORITHM (G)

Step 1: If (G includes no perfect matching) **then** stop.

Step 2: $M :=$ (a perfect matching of G)

Step 3: Call BASIC_ALGORITHM_ITER (G, M)

ALGORITHM BASIC_ALGORITHM_ITER (G, M)

Step 1: Construct $DG(G, M)$.

Step 2: Find an alternating cycle C by finding a directed cycle of $DG(G, M)$.

Step 3: If (no directed cycle exists) **then output** M ; **stop**

Step 4: $M' :=$ the perfect matching obtained from M and C

Step 5: $e :=$ an edge in $M \setminus M'$; $v :=$ an endpoint of e

Step 6: $E_1 := \{e\}$; $E_2 := \{ \text{all the edges incident to } v \text{ except for } e \}$

Step 7: Call BASIC_ALGORITHM_ITER ($G \setminus E_2, M$)

Step 8: Call BASIC_ALGORITHM_ITER ($G \setminus E_1, M'$)

Let x be a vertex of an enumeration tree of the basic algorithm, and $G_x = (V_x, E_x)$ and M_x be the input graph and input matching of x . The time complexity of x is $O(|E_x| + |V_x|)$, which is the computation time in Steps 1 through 8 except for the computation done in generated recursive calls in Steps 7 and 8. Since each leaf of an enumeration tree corresponds to an output, and each internal vertex of the tree has two children, the number of iterations is less than twice the number of outputs, which is $2N$. Hence, the time complexity of this basic algorithm is $O(|E||V|^{1/2} + (|E| + |V|)N)$.

4 Improving the Basic Algorithm

In this section, we improve the basic algorithm by adding a trimming phase and a balancing phase. The trimming phase is composed of two parts, removing edges included in no perfect matching or all perfect matchings, and replacing consecutive degree 2 vertices by an edge.

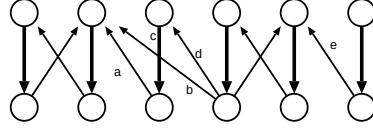


Fig. 2. An instance of $DG(G, M)$. Bold lines are edges of M . Arcs a, b, c, d, e and f are included in no directed cycle. a, b, d and e are included in no perfect matching, and c and f are included in all the perfect matchings

To explain the first part, we prove a lemma. Let $Trim'(DG(G, M))$ be the graph obtained by removing the arcs included in no directed cycle, and $Trim'(G)$ be the undirected version of $Trim'(DG(G, M))$. We denote the edges of M included in $Trim'(G)$ by $Trim'(M)$. Let $IS(G)$ be the graph obtained by removing all the isolated vertices of G .

Lemma 2. $\mathcal{M}(G) = \{M' \cup (M \setminus Trim'(M)) \mid M' \in \mathcal{M}(IS(Trim'(G)))\}$

Proof. An edge e is included in no directed cycle of $DG(G, M)$ if and only if e is included in all the perfect matchings, or no perfect matching. Hence, all the edges in $M \setminus Trim'(M)$ are included in any perfect matching of G . Since any edge of $Trim'(G)$ is incident to no edge of $M \setminus Trim'(M)$, $M' \cup (M \setminus Trim'(M))$ is included in $\mathcal{M}(G)$ for any $M' \in \mathcal{M}(IS(Trim'(G)))$. Moreover, for any $M \in \mathcal{M}(G)$, if a vertex v is incident to no edge of $Trim'(M)$, then no edge of $Trim'(G)$ is incident to v . Hence, $Trim'(M)$ is a perfect matching of $IS(Trim'(G))$. Therefore, the lemma holds. $\square$

Arcs included in no directed cycle can be detected by strongly connected component decomposition. Hence, we obtain $Trim'(G)$ in $O(|E| + |V|)$ time. Next we state the following lemma to explain the second part of the trimming algorithm.

Lemma 3. Suppose that two vertices u and v are incident to only edges (w_1, u) , (u, v) and (v, w_2) , and $w_1 \neq w_2$. Let G' be the graph obtained by removing (w_1, u) , (u, v) and (v, w_2) from G , and adding (w_1, w_2) to it. Then, $\mathcal{M}(G) = \{M \cup \{(u, v)\} \mid M \in \mathcal{M}(IS(G')), (w_1, w_2) \notin M\} \cup \{M \setminus \{(w_1, w_2)\} \cup \{(w_1, u), (v, w_2)\} \mid M \in \mathcal{M}(G'), (w_1, w_2) \in M\}$ holds.

Proof. For any $M \in \mathcal{M}(G)$, exactly one of $(w_1, u), (v, w_2) \in M$ and $(u, v) \in M$ hold. $(w_1, u), (v, w_2) \in M$ if and only if $M \setminus \{(w_1, u), (v, w_2)\} \cup \{(w_1, w_2)\} \in \mathcal{M}(IS(G'))$. $(u, v) \in M$ if and only if $M \setminus \{(w_1, u), (v, w_2)\} \cup \{(w_1, w_2)\} \in \mathcal{M}(IS(G'))$. Hence, the lemma holds. $\square$

Let $Trim(DG(G, M))$ be the graph obtained by applying this operation to $Trim'(DG(G, M))$ while G includes a pair of vertices with degree 2 adjacent to each other, and removing isolated vertices. Let $Trim(G)$ be the undirected version of $Trim(DG(G, M))$. $Trim(G)$ is obtained in $O(|E| + |V|)$ time. We note that $Trim'(DG(G, M)) = DG(Trim'(G), M')$ and $Trim(DG(G, M)) =$

$DG(Trim(G), M'')$ hold for some perfect matchings M' of $Trim'(G)$ and M'' of $Trim(G)$.

In the trimming phase operated before beginning of an iteration x , we construct $Trim(G_x)$ and set G_x to $Trim(G_x)$. After the trimming phase, we output all edges of $M_x \setminus Trim'(M_x)$, and the changes by the operation of Lemma 3. By this, when an iteration inputs an empty graph and output a perfect matching M , the all edges of M are already outputted, hence we can construct M by previous outputs. Thus, we output only a word “matching” when we have to output a perfect matching. since they are included in any perfect matching of the original G . At the end of the iteration x , we cancel the outputs generated in the above. By using this outputting method, we can reduce the computation time for the output as much as the other part of the iteration.

Here we describe the trimming algorithm, inputting G, M and outputting $Trim(G)$.

ALGORITHM TRIMMING_PERFECT_MATCHING (G, M)

Step 1: $G := G \setminus$ (edges corresponding to arcs included in no directed cycle of $DG(G, M)$)

Step 2: **If** (u and v are incident to only edges $(w_1, u), (u, v)$ and (v, w_2) , and $w_1 \neq w_2$)

then $E := E \setminus \{(w_1, u), (u, v), (v, w_2)\} \cup (w_1, w_2)$; **Go to Step 2**

Step 3 Output G

In a trimming and balancing algorithm, we operate the trimming phase for the generated subproblem before generating a recursive call, hence we assume that the input graph G in each iteration satisfies $G = Trim(G)$. This assumption gives a lemma. Let $cc(G)$ be the number of connected components of G , and $f(G)$ be $|E| - |V| + cc(G)$.

Lemma 4. $|\mathcal{M}(G)| \geq f(G) \geq |E|/5$.

Proof. To prove the lemma, we estimate the lower bound of the number of directed cycles in $DG(G, M)$. For a strongly connected component $D_i = (V_i, E_i)$ of $DG(G, M)$, we set a graph $C = (V_C, E_C)$ to a directed cycle of D_i . The number of directed cycles in C is $|E_C| - |V_C| + 1$. If $E_i \setminus E_C \neq \emptyset$, then the graph $(V_i, E_i \setminus E_C)$ contains a directed path $P = (V_P, E_P)$ whose endpoints are both included in C and whose internal vertices and edges are not in C since D_i is strongly connected. P satisfies $|E_P \setminus E_C| - |V_P \setminus V_C| = 1$. By adding P to C , at least one directed cycle including P is generated since C is strongly connected. This addition does not make C non-strongly connected. $|E_C| - |V_C| + 1$ increases only one by this addition. Hence, when $E_C = E_i$ holds, we have that the number of directed cycles in D_i is at least $|E_i| - |V_i| + 1$. Therefore, $DG(G, M)$ includes at least $\sum_{i=1}^{cc(Trim(G))} (|E_i| - |V_i| + 1) = |E| - |V| + cc(G) = f(G)$ directed cycles.

If D_i is a directed cycle with length 2, then $|E_i| - |V_i| + 1 = 1 > 0.2|E_i|$. If D_i is not a directed cycle with length 2, D_i does not include consecutive vertices with degree 2. Hence, $|E_i| \geq 1.25|V_i|$ holds, and we have $|E_i| - |V_i| + 1 = 0.2|E_i|$. Therefore, $f(G) \geq 0.2|E_t|$. $\square$

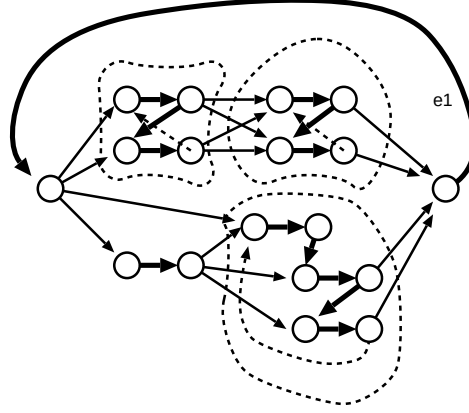


Fig. 3. An instance of DG' : dotted lines are arcs of $DG(G, M)$ not in DG' , and each dotted circle is DG'_i

From this lemma, we can see that G_x has at least $f(G_x)$ perfect matchings, thus $D(x) \geq f(G_x)$. Since the trimming phase and the balancing phase explained in below takes only $O(|E_x|)$ time, we have $\hat{t}(T) = O(1)$. Next we explain the balancing phase. In the balancing phase, we select edge sets E_1 and E_2 such that $f(\text{Trim}(G \setminus E_i)) \geq f(G)/4 - 2$.

If connected components $D_1, \dots, D_k$ of G are at least two, there exists D_i satisfying $f(D_i) \leq f(G)/2$. Since $f(G) = \sum_{i=1}^k f(D_i)$, any subsets E_1 and E_2 of edges incident to a vertex of D_i satisfies $f(\text{Trim}(G \setminus E_i)) \geq f(G)/4$.

In the case that G is connected, we get E_1 and E_2 by partitioning edges incident to a vertex $r \in V_2$. If $f(\text{Trim}(G \setminus E_i)) \geq f(G)/4$ does not hold, then we re-select E_1 and E_2 . Suppose that $f(\text{Trim}(G \setminus E_2)) < f(G)/4$. Let M be a perfect matching of G including an edge $e^* \in E_1$. In $DG(G, M)$, r is the head of e^* since $e^* \in M$. We denote the tail of e^* by r' . To re-select, we construct a directed graph DG' satisfying the following conditions.

Property 3. There exists a directed subgraph DG' of $DG(G, M)$ satisfying:

- (a) any directed cycle in DG' includes e^* ,
- (b) any arc e of DG' is included in a directed cycle, and
- (c) $f(DG') \geq 3f(G)/4$.

Proof. Let $D_i = (V_i, E_i)$ be each strongly connected component of $DG(G \setminus E_2, M)$, and E' be the set of the edges not included in any D_i . We denote the set of vertices in V_i which are heads of edges in E' by VH_i , and those which are tails of edges E' by VT_i . Since $DG(G, M)$ is strongly connected, $VH_i, VT_i \neq \emptyset$. Here we obtain DG'_i by the following operations for each i .

- (1) Choose a vertex $v \in VH_i$. Set $DG'_i = (V'_i, E'_i)$ to $(\{v\}, \emptyset)$

- (2) If there exists a vertex $u \in VT_i \setminus V'_i$, then find a directed path P from a vertex of V'_i to u such that all internal vertices of P are not included in V'_i , add P to DG'_i , and go to (2).
- (3) If there exists a vertex $u \in VH_i \setminus V'_i$, then find a directed path P from u to a vertex of V'_i such that all internal vertices of P are not included in V'_i , add P to DG'_i , and go to (3).

Here we set DG' to $(\bigcup V'_i, E' \cup \bigcup E'_i)$. Since any arc of E' is included in only directed cycles of $DG(G, M)$ including e^* , and (V_i, E_i) includes no directed cycle, we can see that any directed cycle of DG' includes e^* , thus DG' satisfies (a). Since any vertex v of DG' is the tail of an arc of DG' , and is also the head of an arc of DG' , we can see that DG' includes directed paths from v to r and r to v . Hence, DG' satisfies (b).

Since removals of isolated vertices does not change the value of f , we have $f(H) = f(IS(H))$ for any graph H . Since $f(G) = f(G')$ holds in Lemma 3, we have $f(Trim'(H)) = f(Trim(H))$ for any graph H . Thus, from $|E'_i| - |V_i| + cc((V_i, E'_i)) \geq 0$ and $cc((V, E' \cup \bigcup E'_i)) \geq \sum cc((V_i, E'_i)) - cc(Trim(G \setminus E_2)) + 1$, DG' satisfies (c) from the following inequation.

$$\begin{aligned}
f(DG') &= f((V, E' \cup \bigcup E'_i)) \\
&= |E'| + (\sum |E'_i|) - |V| + cc((V, E' \cup \bigcup E'_i)) \\
&\geq |E'| + (\sum |E'_i| - |V_i| + cc((V_i, E'_i))) - cc(Trim(G \setminus E_2)) + 1 \\
&\geq |E'| - cc(Trim'(G \setminus E_2)) + 1 \\
&= |E'| - (f(Trim'(G \setminus E_2)) - (|E| - |E'|) + |V|) + 1 \\
&= |E| - |V| + 1 - f(Trim'(G \setminus E_2)) \\
&= f(G) - f(Trim'(G \setminus E_2)) \\
&\geq 3f(G)/4. \square
\end{aligned}$$

Let $d'(v)$ be the out-going degree of v in DG' , which is the number of arcs of DG' whose tails are v . We note that $f(DG') = cc(DG') + \sum_{v \in V'} (d(v) - 1)$ where V' is the vertex set of DG' . This holds for any directed graph. Let Q be a directed path from r to r' including a maximum out-going degree vertex w of DG' . Note that $w \neq r'$ since $d'(r') = 1$. Let T be a directed spanning tree of DG' including Q whose root is r . For a vertex $v \in T$, we recall that $D(v)$ is the set of all the descendants of v . We note that v is a descendant of v . We also denote the set of all the arcs whose tails are v by $L(v)$, and the set of all the arcs whose tails are in $D(v)$ by $L(D(v))$. For an arc set $F \subseteq L(v)$, we define $D(F) = \{v\} \cup \bigcup_{(v, v') \in F} D(v')$, and $L(D(F)) = F \cup \bigcup_{(v, v') \in F} L(D(v'))$. $|L(D(r))| - D(r) \geq 3f(G)/4$ holds. Since w is not a leaf of T , any leaf v of T satisfies $d'(v) \leq \sum_{v \in V'} d(v)/2$, hence $|L(D(v))| - D(v) \geq 3f(G)/4$ holds. By using this, we re-construct E_1 and E_2 as follows.

- (1) Find a vertex v^* such that $|L(D(v^*))| - |D(v^*)| \geq 2f(G)/4$, and $|L(D(u))| - |D(u)| < 2f(G)/4$ for any child u of v^* .

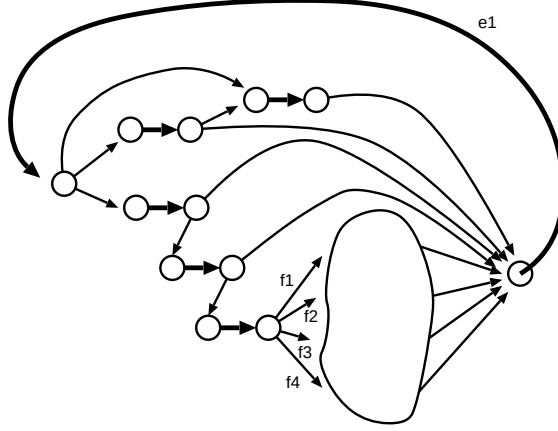


Fig. 4. An instance of re-selected $E_1 = \{f_1, f_2\}$ and $E_2 = \{f_3, f_4\}$. The circle is a subgraph with a large number of arcs

- (2) If an edge $e \in L(v^*)$ satisfies $L(D(\{e\})) \geq f(G)/4$, then we set E_2 to $\{e\}$. If not, we add an arc of $L(v)$ to E_2 iteratively until $|L(D(E_2))| - |D(E_2)| \geq f(G)/4$.

The obtained E_2 satisfies $|L(D(E_2))| - |D(E_2)| < 2f(G)/4$. Since $|L(D(L(v^*)))| - |D(L(v^*))| \geq |L(D(v^*))| - |D(v^*)|$, $E_2 \neq L(v^*)$. Let E_1 be the set of edges incident to v^* and not included in E_2 . Then, the following lemma holds.

Lemma 5. E_1 and E_2 satisfy $f(\text{Trim}(G \setminus E_1)) \geq f(G)/4 - 2$, $f(\text{Trim}(G \setminus E_2)) \geq f(G)/4 - 2$.

Proof. First, we show $f(\text{Trim}(G \setminus E_2)) \geq f(G)/4 - 2$. Let e be an arc whose tail v is in $V \setminus D(E_2)$, and C be a directed cycle of DG' including e . Suppose that C includes an arc of E_2 . Since DG' includes only directed cycles including e^* , at most one arc of E_2 is included in C . We obtain a directed cycle including no arc of E_2 as follows.

- (1) If a directed r - v path in C includes an arc of E_2 , we replace the path of C by the directed r - v path of T .
- (2) If a directed v - r path of C includes an arc of C_1 , we replace the directed v^* - r path of C by a directed v^* - r path including an arc of E_1 . We note that the directed path exists since $L(v^*)$ includes at least one arc of E_1 .

Therefore, e is included in $\text{Trim}'(DG(G \setminus C_1, M))$. From this, the out-going degree of v in $\text{Trim}'(DG(G \setminus E_2, M))$ is $d'(v)$. Similarly, the out-going degree

of v^* in $Trim'(DG(G \setminus E_2, M))$ is $|E_1| - 1$. Thus,

$$\begin{aligned}
 f(Trim(G \setminus E_2)) &= f(Trim'(DG(G \setminus E_2, M))) \\
 &\geq 1 + (|E_1| - 1) + \sum_{v \in V \setminus D(E_2)} (d'(v) - 1) \\
 &= f(DG') - 2 - (|E_2| + \sum_{v \in D(E_2) \setminus \{v^*\}} (d'(v) - 1)) \\
 &\geq 3f(G)/4 - 2 - (|L(D(E_2))| - |D(E_2)|) \\
 &\geq f(G)/4 - 2.
 \end{aligned}$$

We next show that $f(Trim'(DG(G \setminus E_1, M'))) \geq f(G)/4 - 2$. Suppose that C is an alternating cycle respect to M including an edge of E_2 and M' is the perfect matching obtained by C from M .

If an arc e of $DG(G, M)$ satisfies the following two conditions, then $G \setminus E_1$ contains both perfect matchings including e , and those not including e , hence e is included in $Trim'(DG(G \setminus E_1, M'))$.

- (1) There exists a directed cycle in $DG(G, M)$ including e and an arc of E_2 .
- (2) There exists a directed cycle in $DG(G, M)$ including an arc of E_2 and not including e .

Any arc e of $L(D(E_2))$ satisfies (1). If e is not included in C , then e satisfies (2) from the existence of C . Let B be the set of arcs of $C \cap L(D(v^*))$ not included in $Trim'(DG(G \setminus E_1, M'))$, and $d''(v)$ be the out-going degree of v in $Trim'(DG(G \setminus E_1, M'))$. For $v \in D(E_2) \setminus \{v^*\}$, $d''(v) = d'(v) - 1$ if v is the tail of an arc of B , and $d''(v) = d'(v)$ otherwise. Similarly, we can see $d''(v^*) \geq |E_2| - 1$.

Since any arc of B is included in no directed cycle of $Trim'(DG(G \setminus E_1, M'))$, each strongly connected component on which an arc of B has its tail is distinct. Hence, we have $cc(Trim'(DG(G \setminus E_1, M'))) \geq |B|$. From these, we obtain

$$\begin{aligned}
 &f(Trim(G \setminus E_1, M')) \\
 &= f(Trim'(DG(G \setminus E_1, M'))) \\
 &\geq cc(Trim'(DG(G \setminus E_1, M'))) + \sum_{v \in D(E_2)} (d''(v) - 1) \\
 &= cc(Trim'(DG(G \setminus E_1, M'))) + (|E_2| - 1) - 1 + \sum_{v \in D(E_2) \setminus \{v^*\}} (d'(v) - 1) - |B| \\
 &= cc(Trim'(DG(G \setminus E_1, M'))) + (|L(D(E_2))| - |B| - 1) - |D(E_2)| \\
 &\geq |L(D(E_2))| - |D(E_2)| - 1 \\
 &\geq f(G)/4 - 1. \square
 \end{aligned}$$

We describe the framework of our balancing phase as follows.

ALGORITHM BALANCING_PERFECT_MATCHING (G, M)

Step 1: $r :=$ (a vertex of strongly connected component with the minimum value of f)
Step 2: $E_1 :=$ (the set composed of an edge e^* incident to r)
Step 3: $E_2 :=$ (the set of edges incident to r except for e^*)
Step 4: **If** $(f(\text{Trim}(G \setminus E_i)) \leq f(G)/4)$ **then**
Step 5: Construct DG'
Step 6: $E_2 :=$ (a subset of $L(r)$ with $f(G)/4 \leq |L(D(E_2))| - |D(E_2)| < 2f(G)/4$)
Step 7: $E_1 :=$ (the set of edges incident to r and not included in E_2)
Step 8: **End if**
Step 9: **Output** E_1, E_2

Adding this balancing algorithm, we describe our trimming and balancing algorithm.

ALGORITHM ENUM_PERFECT_MATCHINGS_ITER (G, M)

Step 1: **If** G includes no edge, then **output** “matching” ; **return**
Step 2: $E_1, E_2 :=$ BALANCING_PERFECT_MATCHING (G, M)
Step 3: $C :=$ a directed cycle of $DG(G, M)$
Step 4: $M' :=$ the perfect matching obtained by C from M
Step 5: **For** $i := 1$ **to** 2 **do**
Step 6: $G :=$ TRIMMING_PERFECT_MATCHING $(G \setminus E_i, M)$
Output all edges of M not included in G
Step 8: **Call** ENUM_PERFECT_MATCHINGS_ITER $(G, \text{Trim}(M))$
Step 9: **Output** “delete” and all edges of M not included in G
Step 10: Recover the original G by doing the reverse operation of **Step 6**
Step 11: **End for**

For this algorithm, $t(x) = O(|E_x|)$ and $|D(x)| \geq |E_x| - |V_x| + cc(G_x)$ for any iteration x . Thus we have $\hat{t}(T) = O(1)$. Moreover, we obtain the following properties.

- (1) For any child y of x , $t(x) \geq t(y)$.
- (2) If $t(x)$ is constant, then $|D(x)|$ is constant since the size of G_x is constant.
- (3) For any child y of x , $t(y) \geq t(x)/4 - 1$ from the balancing phase.

Hence, from lemma 1, any enumeration tree $\mathcal{T}$ generated by this algorithm satisfies $x^*(\mathcal{T}) = O(\log |E|) = O(\log |V|)$. Therefore, we obtain the following theorem.

Theorem 1. *Perfect matchings in a bipartite graph $G = (V, E)$ can be enumerated in $O(|E||V|^{1/2})$ preprocessing time and $O(\log |V|)$ time per perfect matching.* $\square$

We note that the memory complexity of the algorithm is $O(|E| + |V|)$. The analysis of the memory complexity is same as [3].

Acknowledgment

We would like to thank Professor Akihisa Tamura of the University of Electro-Communications, and all the members of “myogadani-club” for their valuable advice.

References

1. K. Fukuda and T. Matsui, “Finding All the Perfect Matchings in Bipartite Graphs,” *Appl. Math. Lett.* **7**, No. 1, 15-18 (1994). 367, 370
2. J. E. Hopcroft and R. M. Karp, “An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs,” *SIAM J. Comp.* **2**, 225-231 (1973). 370
3. T. Uno, “Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs,” *Lecture Note in Computer Science* **1350**, Springer-Verlag, Algorithms and Computation, 92-101 (1997). 368, 378
4. T. Uno, “A New Approach for Speeding Up Enumeration Algorithms,” *Lecture Note in Computer Science* **1533**, Springer-Verlag, Algorithms and Computation, 287-296 (1998). 368
5. T. Uno, “A New Approach for Speeding Up Enumeration Algorithms and Its Application for Matroid Bases,” *Lecture Note in Computer Science* **1627**, Springer-Verlag, Computing and Combinatorics (Proceeding of COCOON99), 349-359, (1999)
368

On-Line Scheduling a Batch Processing System to Minimize Total Weighted Job Completion Time

Bo Chen^{*}, Xiaotie Deng, and Wenan Zang

¹ Warwick Business School, University of Warwick
Coventry, United Kingdom
B.Chen@warwick.ac.uk

² Department of Computer Science, City University of Hong Kong
Hong Kong, P. R. China
Deng@cs.cityu.edu.hk

³ Department of Mathematics, University of Hong Kong
Hong Kong, P. R. China
WZang@maths.hku.hk

Abstract. Scheduling a batch processing system has been extensively studied in the last decade. A batch processing system is modelled as a machine that can process up to b jobs simultaneously as a batch. The scheduling problem involves assigning all n jobs to batches and determining the batch sequence in such a way that certain objective function of job completion times C_j is minimized. In this paper, we address the scheduling problem under the *on-line* setting in the sense that we construct our schedule irrevocably as time proceeds and do not know of the existence of any job that may arrive later. Our objective is to minimize the total weighted completion time $\sum w_j C_j$. We provide a linear time on-line algorithm for the unrestrictive model (i.e., $b \geq n$) and show that the algorithm is $10/3$ -competitive. For the restrictive model (i.e., $b < n$), we first consider the (off-line) problem of finding a maximum independent vertex set in an interval graph with cost constraint (MISCP), which is NP-hard. We give a dual fully polynomial time approximation scheme for MISCP, which leads us to a $(4 + \epsilon)$ -competitive on-line algorithm for any $\epsilon > 0$ for the original on-line scheduling problem. These two on-line algorithms are the first deterministic algorithms of constant performance guarantees.

1 Introduction

Scheduling a batch processing system has been extensively studied in the last decade. A batch processing system is modelled as a machine that can process up to b jobs simultaneously as a batch. The processing time of a batch is the time required for processing the longest job in the batch. The scheduling problems

^{*} Corresponding author

involve assigning all n jobs to batches and determining the batch sequence in such a way that certain objective function of job completion times C_j is minimized.

There are two distinct models for the scheduling problems. In the *restrictive* model, the bound b for each batch size is effective, i.e., $b < n$. Problems of this model arise in the manufacture of integrated circuits [13]. The critical final stage in the production of circuits is the burn-in operation, in which chips are loaded onto boards which are then placed in an oven and exposed to high temperatures. Each chip has a prespecified minimum burn-in time and the burn-in oven has a limited capacity. In the *unrestrictive* model, there is effectively no limit on the sizes of batches, i.e., $b \geq n$. Scheduling problems of this model arise, for instance, in situations where compositions need to be hardened in kilns and the kiln is sufficiently large that it does not restrict batch sizes [1].

Polynomial solvability and simple approximations have been studied in [11,12,1,14] for the objective of minimizing the makespan $\max C_j$, in [13,15,1] for minimizing the number of tardy jobs, where a job j is said to be *tardy* if $C_j > d_j$, the given due date for job j , or minimizing the maximum lateness $\max(C_j - d_j)$. For minimizing total completion time $\sum C_j$, a branch-and-bound algorithm as well as dynamic programming algorithms for the special case of a fixed number of different job processing times are developed in [3,4,10,1] under the assumption that all job release times are the same. These procedures are only effective for small problem instances.

Let us concentrate on problems of the objective function $\sum_j C_j$. This objective, which is equivalent to minimizing the average time spent in the system by a job, increases throughput and reduces work-in-process inventories. As pointed out in [10], this is especially important in the scheduling of burn-in operations, which are often a bottleneck in the final stage of semiconductor production due to their long processing items relative to the other testing operations. Even if all release times are equal, the problem of restrictive model is claimed by Brucker et al. [1] to be undoubtedly the most vexing problem among all those with different objective functions. They give an $O(n^{b(b-1)})$ time dynamic programming algorithm for fixed $b > 1$. Note that if $b = 1$ then it is a classical scheduling problem and is solvable by a simple listing procedure in $O(n \log n)$ time. For variable b , although the problem complexity is still open, Hochbaum & Landy [10] nevertheless provide a 2-approximation algorithm, where a ρ -approximation algorithm ($\rho \geq 1$) is a polynomial algorithm that always delivers a schedule of objective value guaranteed to be no more ρ times optimal value. This approximation result is improved recently in [5] to a *polynomial time approximation scheme* (PTAS), i.e., a family of $(1 + \epsilon)$ -approximation algorithms for any $\epsilon > 0$.

If each job j is associate with a *weight* w_j and a more general objective—the total weighted completion time $\sum w_j C_j$ is considered, then a $O(n \log n)$ time algorithm is given in [1] for the unrestrictive model ($b \geq n$) still with the assumption that all jobs are released at the same time. For general release times, it is proved very recently in [7] that the problem of unrestrictive model is NP-hard, and a PTAS is given in [6] if all weights are equal.

In this paper, we consider the scheduling problems of both models with general release times and with general objective function $\sum w_j C_j$. Still further we address the problems under the *on-line* setting in the sense that we construct our schedule irrevocably as time proceeds and do not know of the existence of any job that may arrive later. Similar to the notion of ρ -approximation algorithms, an on-line (polynomial) algorithm is said to be ρ -competitive if it always delivers a schedule of objective value guaranteed to be no more ρ times (off-line) optimal value.

Because of lack of information, it is normally no longer possible to have an on-line algorithm that guarantees to deliver an optimal solution. In fact, even all the weights are equal, no on-line algorithm exists that is better than 2-competitive [16]. For the unrestrictive model, we provide a linear time on-line algorithm and show that the algorithm is 10/3-competitive. For the restrictive model, we first consider the (off-line) problem of finding a maximum independent vertex set in an interval graph with cost constraint (MISCP), which is NP-hard. We give a dual fully PTAS for MISCP (see Section 4 for definition), which leads us to a $(4+\epsilon)$ -competitive algorithm for any $\epsilon > 0$. These two on-line algorithms are the first (deterministic) algorithms of constant performance guarantees. We note that, using the *randomized* greedy framework described in [2], one can conceivably obtain a 2.89-competitive (in expectation) and $(2.89+\epsilon)$ -competitive (in expectation) on-line algorithms, respectively, for the unrestrictive and restrictive models.

The remainder of this paper is organized as follows. In Section 2 we outline a general framework of on-line scheduling for minimizing total weighted completion time of Hall et al. [8], which we will use later. We investigate both restrictive and unrestrictive models respectively in Sections 3 and 4.

2 Preliminaries

Let jobs $1, 2, \dots, n$ be released on-line at times $r_1 \leq r_2 \leq \dots \leq r_n$, where job j has a processing time $p_j > 0$ and a weight $w_j > 0$. Denote by C_j the completion time of job j , which is the completion time of the batch job j is assigned to. Our task is to construct an on-line schedule such that the total weighted completion time $\sum_{j=1}^n w_j C_j$ is minimized.

The two on-line algorithms that we are to present are motivated by a general on-line framework of Hall et al. [8], which is called Greedy-Interval and, for any $\rho \geq 1$, uses as a subroutine a dual ρ -approximation off-line algorithm for the following problem to obtain a 4ρ -competitive on-line algorithm:

THE MAXIMUM SCHEDULED WEIGHT PROBLEM (MSWP): Given a certain scheduling environment, which is a single batch processing machine here, and a deadline D , a set of jobs available at time 0, and a weight for each job, construct a feasible schedule that maximizes the total weight of jobs completed by time D .

A *dual* ρ -approximation algorithm for MSWP is a polynomial algorithm that always delivers a schedule of length at most ρD and whose total weight is at least the optimal weight for the deadline D .

Partition the time horizon of possible completion times at geometrically increasing points. Let $\tau_i = 2^i$, $i = 0, 1, \dots$, be points in the time horizon. Greedy-Interval constructs the schedule interactively. At iteration $i = 1, 2, \dots$, we wait until time τ_{i-1} , and then focus on the set of jobs that have been released by this time but not yet scheduled. These jobs are scheduled to run from time $\rho\tau_{i-1}$ to $\rho\tau_i$ by invoking the dual ρ -approximation (off-line) algorithm with deadline $D = \rho\tau_{i-1}$. According to [8], we have the following.

Lemma 1. *Any dual ρ -approximation off-line algorithm for MSWP is efficiently converted in the aforementioned way by Greedy-Interval into a 4ρ -competitive on-line algorithm for minimizing the total weighted completion time.*

3 An Algorithm for the Unrestrictive Model

Recall that in the unrestrictive model, the upper bound b on the sizes of batches is no smaller than the total number of jobs to be scheduled: $b \geq n$. It is evident in this situation that MSWP can be solved to optimality by simply putting into a single batch all those jobs that have a processing time no more than D . Thus a 4-competitive on-line algorithm follows directly from Lemma 1. In this section, we are to have a better on-line algorithm, which is $10/3$ -competitive.

3.1 Algorithm DelaySep

Let $T_i = [\tau_{i-1}, \tau_i)$ ($i = 1, 2, \dots$) be mutually disjoint time intervals. Denote $\alpha = 5/6$ and $\beta = 3/5$. For any $i \geq 1$, define

$$\begin{aligned} S_i &= \{j : r_j + p_j \in T_i\}, \\ A_i &= \{j \in S_i : p_j \leq \beta\tau_i\}, \\ B_i &= S_i \setminus A_i. \end{aligned}$$

Algorithm DelaySep works as follows. Initially set $B'_i := \emptyset$ for all $i \geq 1$. For $i = 1, 2, \dots$, do the following at time point τ_i until all jobs have been scheduled:

Let $w(A_i) = \theta_i w(S_i)$ for some $0 \leq \theta_i \leq 1$, where $w(S) = \sum_{j \in S} w_j$ for any set S of jobs. If $\theta_i \leq \alpha$, then put all jobs of S_i , together with jobs of set B'_i , in a single batch and schedule them in interval T_{i+1} . Otherwise, on the one hand, put all jobs of A_i , together with jobs of set B'_i , in a batch and schedule them in T_{i+1} . On the other hand, set $B'_{i+1} := B_i$.

Note that, at each time point τ_i ($i \geq 1$), algorithm DelaySep schedules jobs of set S_i into interval T_{i+1} or intervals $T_{i+1} \cup T_{i+2}$, depending on the total weight proportion of jobs of A_i . If this proportion is small, then all jobs of S_i are put in a single batch and scheduled in interval T_{i+1} . Otherwise, jobs of A_i and B_i are put in two separate batches and scheduled in T_{i+1} and T_{i+2} , respectively. Apparently, the algorithm runs on-line. Two basic observations underpin the design of the algorithm. First, the later a job can be completed, the less need for it to be very finely scheduled. Second, a schedule will not be good if a set of small jobs is put in the same batch with a set of large jobs that amount only to a small proportion of the combined total weight.

3.2 Analysis of the Algorithm

For any job j , recall that C_j denotes the completion times of job j in the heuristic schedule and $p_j > 0, w_j > 0$. Let C_j^* denote the completion time of job j in an optimal off-line schedule. Without loss of generality, we assume that $r_j + p_j \geq 1$ for all job j , which can be achieved by normalizing the time scale with respect to $t = \min\{\min_{r_j=0} p_j, \min_{r_j>0} r_j\}$. We will show that for each $i \geq 1$,

$$\sum_{j \in S_i} w_j C_j \leq \frac{10}{3} \sum_{j \in S_i} w_j C_j^*, \quad (1)$$

which yields the following theorem.

Theorem 1. *Algorithm DelaySep is 10/3-competitive.*

In the rest of this section, we assume without loss of generality that $S_i \neq \emptyset$. We start with establishing a lower bound for any optimal schedule.

Lemma 2. *For any $i \geq 1$, the following lower bound holds:*

$$\sum_{j \in S_i} w_j C_j^* \geq (\min\{1 - 2\beta\theta_i, 0\} + 2\beta) \tau_{i-1} w(S_i).$$

Proof. Fix any optimal schedule. Jobs of S_i are processed in one or more batches, possibly with other jobs. Consider the first, S'_i , of these batches. There are three possibilities: (a) $S'_i \cap S_i \subseteq A_i$. (b) $S'_i \cap S_i \subseteq B_i$. (c) Neither (a) nor (b). According to the definitions of S_i , A_i and B_i , we can easily see that the following quantities are lower bounds on $\sum_{j \in S_i} w_j C_j^*$ for the three corresponding cases: (a) $\tau_{i-1} w(A_i) + (\tau_{i-1} + \beta\tau_i) w(B_i)$. (b) $\beta\tau_i w(B_i) + \beta\tau_i w(A_i)$. (c) $\beta\tau_i w(S_i)$. Therefore, we are led directly to the claimed lower bound.

To compare $\sum_{j \in S_i} w_j C_j$ with $\sum_{j \in S_i} w_j C_j^*$, we consider two cases according to the description of the algorithm.

Case 1. $\theta_i \leq \alpha$. In this case, we have $C_j \leq \tau_{i+1}$ for any $j \in S_i$. Therefore, Lemma 2 leads to

$$\sum_{j \in S_i} w_j C_j^* \geq \beta\tau_i w(S_i) = \frac{\beta}{2} \tau_{i+1} w(S_i) \geq \frac{3}{10} \sum_{j \in S_i} w_j C_j.$$

Case 2. $\theta_i > \alpha$. Then $A_i \neq \emptyset$. Let $\beta_i = \max_{j \in A_i \cup B'_i} \{p_j / \tau_i\}$. Since $p_j \leq \tau_{i-1}$ for any $j \in B'_i$, we have $0 < \beta_i \leq \beta$. Let $\lambda_i = \sum_{j \in A_i} w_j C_j / \sum_{j \in S_i} w_j C_j$. Note that

$$C_j = (1 + \beta_i)\tau_i \text{ if } j \in A_i, \text{ and } C_j \leq \tau_{i+2} \text{ if } j \in B_i. \quad (2)$$

Hence

$$\begin{aligned} \lambda_i \sum_{j \in S_i} w_j C_j &= \sum_{j \in A_i} w_j C_j = (1 + \beta_i)\tau_i w(A_i) = (1 + \beta_i)\tau_i \theta_i w(S_i) > 0, \\ (1 - \lambda_i) \sum_{j \in S_i} w_j C_j &= \sum_{j \in B_i} w_j C_j \leq \tau_{i+2} w(B_i) = \tau_{i+2}(1 - \theta_i) w(S_i). \end{aligned}$$

Thus,

$$\frac{1 - \lambda_i}{\lambda_i} \leq \frac{4}{1 + \beta_i} \frac{1 - \theta_i}{\theta_i} < \frac{4}{1 + \beta_i} \frac{1 - \alpha}{\alpha} = \frac{4(2\beta - 1)}{1 + \beta_i},$$

or $\lambda_i \geq (1 + \beta_i)/(8\beta + \beta_i - 3)$, which, together with Lemma 2, (2) and the fact that $\frac{1}{2(1+\beta_i)} \geq \frac{1}{2(1+\beta)} > \frac{1+2\beta}{8}$, implies

$$\begin{aligned} \sum_{j \in S_i} w_j C_j^* &\geq \tau_{i-1} w(A_i) + (\tau_{i-1} + \beta \tau_i) w(B_i) \\ &\geq \left[\frac{1}{2(1 + \beta_i)} \lambda_i + \frac{1 + 2\beta}{8} (1 - \lambda_i) \right] \sum_{j \in S_i} w_j C_j \\ &\geq \left[\frac{1}{2(8\beta + \beta_i - 3)} + \frac{(1 + 2\beta)(8\beta - 4)}{8(8\beta + \beta_i - 3)} \right] \sum_{j \in S_i} w_j C_j \\ &\geq \frac{4\beta^2}{2(9\beta - 3)} \sum_{j \in S_i} w_j C_j = \frac{3}{10} \sum_{j \in S_i} w_j C_j. \end{aligned}$$

Therefore, inequality (1) and hence Theorem 1 follow.

Note that the bound in (1) is tight. To see this, consider the following instance of three jobs, all of which are released at time 0. The processing times and weights are $(0 < \epsilon < \frac{4}{5})$: (a) $p_1 = 1$, $p_2 = 2\beta$ and $p_3 = 2 - \epsilon$; (b) $w_1 = \alpha$, $w_2 = 1 - \alpha$ and $w_3 = \epsilon$. It is evident that $S_1 = \{1, 2, 3\}$, $A_1 = \{1, 2\}$ and $B_1 = \{3\}$, and $\theta_1 = \alpha/(1 + \epsilon) < \alpha$. Hence algorithm DelaySep puts all three jobs in one batch and processes them during time 2 to $4 - \epsilon$, which yields that $\sum_{j \in S_1} w_j C_j = (1 + \epsilon)(4 - \epsilon)$. However, a better schedule is that jobs 1 and 2 are processed together during time 0 and 2β and job 3 is processed immediately afterwards, which implies that $\sum_{j \in S_1} w_j C_j^* \leq 2\beta + (2\beta + 2 - \epsilon)\epsilon$. The performance ratio approaches to $\frac{10}{3}$ as ϵ goes to zero.

4 The Restrictive Model: Approximating MSWP

In this section we consider the other model in which $b < n$. We will first derive for MSWP a dual *fully polynomial time approximation scheme* (FPTAS), i.e., a dual $(1 + \epsilon)$ -approximation algorithm that runs in time also polynomial to $1/\epsilon$ for any $\epsilon > 0$. The dual FPTAS, together with the on-line framework of Section 2, will then result in a $(4 + \epsilon)$ -competitive on-line algorithm for our original scheduling problem. We first reduce our problem to one of interval graphs. For simplicity, we assume that all job processing times are different. Otherwise, we may apply the standard perturbation method as follows: Scale processing times and deadline D so that they are all integers. Then, for some $\delta < 1/2$, we change the length p_i to $p_i + \delta^i$, $i = 1, 2, \dots, n$, and the deadline D to $D + \delta/(1 - \delta)$.

4.1 Reduction to a Graph Problem

For any schedule π , in which batches $B_1, \dots, B_m$ are processed consecutively in that order, we denote it by $\pi = (B_1, \dots, B_m)$. Let $l(B_i)$ and $u(B_i)$ be the

minimum and maximum indices, respectively, of jobs in batch B_i . Our reduction starts with the following two basic properties of any optimal solution for MSWP.

Lemma 3. *If the processing times of jobs satisfy $p_1 < p_2 < \dots < p_n$. Then any optimal schedule $\pi = (B_1, \dots, B_m)$ satisfies the following properties:*

1. *For any pair of batches B_i and B_k in π , if $i < k$, then every job in B_i has a processing time less than that of any job in B_k .*
2. *Exclusion of any job j ($l(B_i) < j < u(B_i)$) in schedule π implies that its weight w_j is no more than that of any job in batch B_i .*

Proof. To the contrary of the first property, assume that some job in B_i has a larger processing time than another job in B_k ($i < k$). Then we have $p_j > p_{j'}$, where $j = u(B_i)$ and $j' = l(B_k)$. Hence $j > j'$. Consider a new schedule $\pi' = (B'_1, \dots, B'_m)$, where $B'_\ell = B_\ell$ for any $\ell \neq i, k$. If $j > u(B_k)$, then $B'_i = B_k$ and $B'_k = B_i$, i.e., π' is the same as π except that the positions of batches B_i and B_k are swapped. If $j' < j < u(B_k)$, then $B'_i = (B_i \cup \{j'\}) \setminus \{j\}$ and $B'_k = (B_k \cup \{j\}) \setminus \{j'\}$, i.e., π' is the same as π except that jobs j and j' are swapped with respect to the batches they are originally assigned to. We observe that jobs in each of batches $B'_i, \dots, B'_{k-1}$ (in the first case $j > u(B_k)$) or of batches $B'_i, \dots, B'_m$ (in the second case $j' < j < u(B_k)$) finish earlier than in π , while all the other jobs in π' finish at the same time as in π . Since each job has a positive weight, we obtain a better schedule than π , which contradicts its optimality. Therefore, property 1 holds.

To see that property 2 holds, suppose $w_j > w_{j'}$ for some job $j' \in B_i$. Then replacing job j' by job j in schedule π would result in a better schedule.

Property 1 is equivalent to $[l(B_i), u(B_i)] \cap [l(B_k), u(B_k)] = \emptyset$ for any pair of batches B_i and B_k . This allows us to establish a correspondence between a feasible schedule and an interval graph. For every subset B of jobs with $|B| \leq b$, create an interval $I_B = [l(B), u(B)]$ with its *weight* defined as the sum of weights of jobs in B and its *cost* defined as the maximum processing time of jobs in B . An independent vertex set of the interval graph corresponds to a feasible schedule that satisfies property 1 of Lemma 3, and vice versa. Therefore, MSWP becomes the problem of finding a maximum-weight independent vertex set with total cost no more than D . However, this interval graph has a vertex set of size $O(n^b)$, not a polynomial when b is part of the input. Property 2 of Lemma 3 allows us to overcome this difficulty by restricting our attention to a polynomial number of candidate intervals. For any fixed indices a and b , $1 \leq a \leq b \leq n$, a candidate batch B in an optimal schedule with $l(B) = a$ and $u(B) = b$ will contain up to b jobs of maximum weights among all jobs of indices between a and b . Since the total number of such candidate batches is $O(n^2)$, the interval graph we have constructed is polynomial in the input size of the original problem.

4.2 A Dynamic Programming Solution

As we have seen, MSWP can be polynomially reduced to the following problem of finding a maximum independent vertex set in an interval graph with cost

constraint (MISCP): Given a positive integer D and m nonempty closed intervals $I_1, I_2, \dots, I_m$ on the real line such that each interval I_k is associated with a weight $\omega_k > 0$ and a integral cost $c_k > 0$, find a set $\mathcal{I}$ of pairwise disjoint intervals of $I_1, I_2, \dots, I_m$ such that $\sum_{I_k \in \mathcal{I}} c_k \leq D$ and that $\sum_{I_k \in \mathcal{I}} \omega_k$ is maximized. Note that MISCP is NP-hard, since it becomes the well-known knapsack problem when $I_1, I_2, \dots, I_m$ are pairwise disjoint. By adding two dummy intervals of zero weight and cost, we can reduce MISCP to the problem of finding the longest path between two fixed nodes in a directed acyclic graph and hence an FPTAS exists for MISCP [9]. However, since we are interested in a *dual* FPTAS here, it is more efficient to provide a direct approach by first deriving a suitable pseudo-polynomial algorithm for solving MISCP to optimality.

Let interval $I_k = [a_k, b_k]$ for all $k = 1, \dots, m$. Reindex these intervals, if necessary, so that whenever $i < k$ we have $a_i < a_k$ or $a_i = a_k$ and $b_i < b_k$. For convenience, write $\{t_1, \dots, t_{m'}\} = \{a_1, \dots, a_m\} \cup \{b_1, \dots, b_m\}$, where $t_1 < \dots < t_{m'}$. Clearly $m' \leq 2m$. Denote by $\mathcal{F}_i$ any set of pairwise disjoint intervals of $\{I_1, I_2, \dots, I_i\}$. For each integral triple (i, j, d) with $1 \leq i \leq m$, $1 \leq j \leq m'$ and $0 \leq d \leq D$, define

$$W(i, j, d) = \max_{\mathcal{F}_i} \sum_{I_k \in \mathcal{F}_i} \omega_k, \quad (3)$$

where $\mathcal{F}_i$ is subject to the following conditions:

$$\sum_{I_k \in \mathcal{F}_i} c_k \leq d, \text{ and } \max_{I_k \in \mathcal{F}_i} b_k \leq t_j.$$

It is evident that an optimal solution to MISCP is such a set $\mathcal{F}_n$ that achieves the maximum in equation (3) for $W(n, m', D)$. Note that $W(i+1, j, d) \geq W(i, j, d)$. It is easy to see that for any j and d , $1 \leq j \leq m'$, $0 \leq d \leq D$, we have

$$W(1, j, d) = \begin{cases} 0 & \text{if } t_j < b_1 \text{ or } d < c_1 \\ \omega_1 & \text{otherwise.} \end{cases}$$

To derive the recurrence relation for $W(i+1, j, d)$, note that if $t_j < b_{i+1}$, then it follows instantly from definition that $W(i+1, j, d) = W(i, j, d)$. Suppose that $t_j \geq b_{i+1}$. Let $\mathcal{F}_{i+1}$ be a set of pairwise disjoint intervals in $\{I_1, I_2, \dots, I_{i+1}\}$ such that it achieves the maximum in (3) for $W(i+1, j, d)$. If $I_{i+1} \notin \mathcal{F}_{i+1}$ then $W(i+1, j, d) = W(i, j, d)$. Assume $I_{i+1} \in \mathcal{F}_{i+1}$. Then $d \geq c_{i+1}$ and each interval I_k in $\mathcal{F}_{i+1} \setminus \{I_{i+1}\}$ satisfies $b_k < a_{i+1}$ according to the indexing of the intervals and the definition of $\mathcal{F}_{i+1}$. Hence $W(i+1, j, d) = W(i, s, d - c_{i+1}) + \omega_{i+1}$, where s satisfies $t_{s+1} = a_{i+1}$. Summarizing, we obtain the following recurrence relation:

$$W(i+1, j, d) = \begin{cases} W(i, j, d) & \text{if } t_j < b_{i+1} \text{ or } d < c_{i+1} \\ W' & \text{otherwise,} \end{cases}$$

where $W' = \max\{W(i, j, d), W(i, s, d - c_{i+1}) + \omega_{i+1}\}$ and s satisfies $t_{s+1} = a_{i+1}$.

It is easy to see that the time complexity of the above dynamic programming algorithm is $O(mm'D) = O(m^2D)$.

4.3 An Approximation Scheme

With the pseudo-polynomial algorithm in the previous subsection, it is straightforward to construct a dual FPTAS for MISCP and hence for MSWP by applying the following rounding and scaling techniques.

Theorem 2. *For any constant $\epsilon > 0$, a dual $(1 + \epsilon)$ -approximate solution to MISCP can be found in $O(m^3/\epsilon)$ time.*

Proof. For any given ϵ , we round down $\{c_k\}$ and D to their nearest multiples of $\delta = \epsilon D/m$. More precisely, let $\bar{c}_k = \lfloor c_k/\delta \rfloor$ and $\bar{\omega}_k = \omega_k$ for all k and let $\bar{D} = \lfloor D/\delta \rfloor$. Now apply the dynamic programming procedure in Section 4.2 to MISCP with the new data, we obtain an optimal solution $\bar{\mathcal{I}}$ in time $O(m^2 \bar{D}) = O(m^3/\epsilon)$. We assert that $\bar{\mathcal{I}}$ is a dual $(1 + \epsilon)$ -approximation solution to the original MISCP. To see this, let $\mathcal{I}$ be an optimal solution to the original MISCP. Then $\mathcal{I}$ is a feasible solution to the rounded MISCP, since

$$\delta \sum_{I_k \in \mathcal{I}} \bar{c}_k \leq \sum_{I_k \in \mathcal{I}} c_k \leq D,$$

which implies that $\sum_{I_k \in \mathcal{I}} \bar{c}_k$ as an integer is no more than $\bar{D}$. Then the optimality of $\bar{\mathcal{I}}$ for the rounded MISCP implies that $\sum_{I_k \in \bar{\mathcal{I}}} \bar{\omega}_k \geq \sum_{I_k \in \mathcal{I}} \omega_k$. On the other hand, we have

$$\begin{aligned} \sum_{I_k \in \bar{\mathcal{I}}} c_k &\leq \sum_{I_k \in \bar{\mathcal{I}}} \delta(\bar{c}_k + 1) \leq \delta \sum_{I_k \in \bar{\mathcal{I}}} \bar{c}_k + m\delta \\ &\leq \delta \bar{D} + m\delta \leq D + \epsilon D = (1 + \epsilon)D. \end{aligned}$$

Therefore, $\bar{\mathcal{I}}$ is a dual $(1 + \epsilon)$ -approximation solution.

From Theorem 2 and the fact that MSWP can be polynomially reduced to MISCP, we conclude that there is a dual FPTAS for MSWP, which in turn implies with Lemma 1 that the dynamic programming procedure in Section 4.2 together with on-line framework Greedy-Interval yields a $(4 + \epsilon)$ -approximation algorithm for scheduling a batch processing system of restrictive model.

Acknowledgement

Research for the first two authors was supported in part by a research grant (Project #: 9040597) of the Research Grants Council of Hong Kong Special Administrative Region, China, and an SRG grant of City University of Hong Kong (Project #: 7001040), for the first author by ESRC Research Fellowship (UK), and for the third author by RGC grant 338/024/0009 and RGC grant 10202890.

References

1. P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. N. Potts, T. Tautenhahn, and S. L. Van de Velde, Scheduling a Batching Machine, *Journal of Scheduling* **1** (1998), 31-54. 381
2. S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein, Improved scheduling algorithms for minsum criteria. In: F. Meyer auf der Heide and B. Monien (eds.): *Automata, Languages and Programming*, Lecture Notes in Computer Science 1099, Springer: Berlin, 1996, 646-657. 382
3. V. Chandru, C.-Y. Lee and R. Uzsoy, Minimizing total completion time on batch processing machines, *International Journal of Production Research* **31** (1993), 2097-2121. 381
4. V. Chandru, C.-Y. Lee and R. Uzsoy, Minimizing total completion time on a batch processing machine with job families, *Operations Research Letters* **13** (1993), 61-65. 381
5. X. Deng, H. Feng, G. Li, G. Liu, A PTAS for minimizing total flow-time of bounded batch scheduling, unpublished manuscript. 381
6. X. Deng, H. Feng, P. Zhang, Y. Zhang, H. Zhu, Minimizing mean completion time in batch processing system, submitted for journal publication. 381
7. X. Deng and Y. Zhang, Minimizing mean response time in a batch processing system, COCOON99, *Lecture Notes in Computer Science* **1627** (1999), 231-240. 381
8. L. A. Hall, A. S. Schulz, D. B. Shmoys and J. Wein, Scheduling to minimize average completion time: off-line and on-line approximation algorithms, *Mathematics of Operations Research* **22** (1997), 513-544. 382, 383
9. R. Hassin, Approximation schemes for the restricted shortest path problem, *Mathematics of Operations Research* **17** (1992), 36-42. 387
10. D. S. Hochbaum and D. Landy, Scheduling semiconductor burn-in operations to minimize total flowtime, *Operations Research* **45** (1997), 874-885. 381
11. Y. Ikura and M. Gimple, Efficient scheduling algorithms for a single batch processing machine, *Operations Research Letters* **5** (1986), 61-65. 381
12. C.-Y. Lee and R. Uzsoy, Minimizing makespan on a single batch processing machine with dynamic job arrivals, *Preprint*, 1996. 381
13. C.-Y. Lee, R. Uzsoy and L. A. Martin-Vega, Efficient algorithms for scheduling semiconductor burn-in operations, *Operations Research* **40** (1992), 764-775. 381
14. Z. Liu and W. Yu, Scheduling one batch processor subject to job release dates, *Discrete Applied Mathematics* **105** (2000), 129-136. 381
15. R. Uzsoy, Scheduling batch processing machines with incompatible job families, *International Journal of Production Research* **33** (1995), 2685-2708. 381
16. A. P. A. Vestjens, *On-Line Machine Scheduling*, Ph.D. Dissertation, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 1997. 382

On the Complexity of Train Assignment Problems^{*}

Thomas Erlebach¹, Martin Gantenbein², Daniel Hürlimann³, Gabriele Neyer⁴,
Aris Pagourtzis⁵, Paolo Penna², Konrad Schlude², Kathleen Steinhöfel⁶, David
Scot Taylor², and Peter Widmayer²

¹ Computer Engineering and Networks Laboratory, ETH Zentrum, CH-8092 Zürich,
Switzerland

`erlebach@tik.ee.ethz.ch`

² Institute for Theoretical Computer Science, ETH Zentrum, CH-8092 Zürich,
Switzerland

`{lastname}@inf.ethz.ch`

³ Institute of Transportation, Traffic, Highway- and Railway-Engineering (IVT),
ETH Hönggerberg, CH-8093 Zürich, Switzerland

`huerlimann@ivt.baug.ethz.ch`

⁴ SMA and Partners Ltd., Transportation Engineers, Planners and Economists,
Gubelstrasse 28, CH-8050 Zürich, Switzerland

`GabrieleNeyer@gmx.ch`

⁵ Department of Computer Science, University of Liverpool, United Kingdom

`aris@csc.liv.ac.uk`

⁶ GMD - FIRST, Berlin, Germany

`Kathleen.Steinhofel@gmd.de`

Abstract. We consider a problem faced by train companies: How can trains be assigned to satisfy scheduled routes in a cost efficient way? Currently, many railway companies create solutions by hand, a time-consuming task which is too slow for interaction with the schedule creators. Further, it is difficult to measure how efficient the manual solutions are. We consider several variants of the problem. For some, we give efficient methods to solve them optimally, while for others, we prove hardness results and propose approximation algorithms.

1 Introduction

We consider the problem of assigning trains to the routes of a railway network so as to implement a given schedule and to minimize the associated cost, subject to various constraints. This problem is sometimes called *train assignment*, *train rostering*, *vehicle scheduling* or *rolling stock rostering*, and currently, it is commonly done by hand. For instance, to modify train schedules from one year to

^{*} Work partially supported by the Swiss Federal Office for Education and Science under the Human Potential Programme of the European Union under contract no. HPRN-CT-1999-00104 (AMORE).

the next within Switzerland, the Swiss Federal Railways SBB uses several man years of labor.

With today's powerful computers, the train assignment problem should lend itself nicely to automatic solutions. It has the additional benefit that it can take effect immediately: no customer acceptance of a new schedule is needed. Furthermore, a useful system need not be perfect: any tool that proposes an initial assignment and gives an interactive indication of how easy or difficult it is to make modifications will be useful. The final schedules and train assignments may still require human expertise.

We explore how different constraints change the problem from versions with efficient, optimal solutions, to versions which are APX-hard. Among the constraints we consider, we focus on scheduling the maintenance of trains and on allowing or disallowing movements of empty, non-scheduled trains (deadheading). For the APX-hard problem versions, we propose approximation algorithms.

1.1 The Basic Model

As input, we are given a set of *train routes*: each train route is specified by a *departure time/station* and an *arrival time/station*. The routes are *periodic*, and for the purpose of this paper, we will assume a daily period: *each route in the input runs every day*. Naturally, our results do not depend on the interpretation of the periods, and hence they also apply to other time frames, such as weekly schedules. The goal is to assign trains to perform the routes in a cost effective way, subject to constraints.

In Figure 1.a, we show a graphical representation of a two-routes two-stations

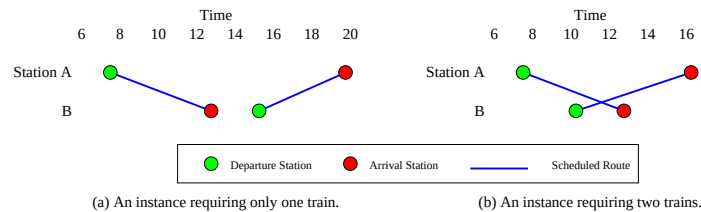


Fig. 1. Two simple train schedules

instance: the x -axis represents the time and the y -axis represents the stations; an edge between two points represents a route. For this example, one train is sufficient to cover both routes. A train t first begins at station A and travels from A to B to cover the first route. Once t arrives in B, it can wait and then cover the second route. At this point, the train is back at station A and is ready for covering the route from A to B of the next day. So, the train repeats the same cycle every day.

The reason one train can cover both the routes is that the arrival time of the first route precedes the departure time of the second one. Something different happens in the example of Figure 1.b: train t arrives at B too late to perform the second route. So, we need another train t' at station B at the beginning of the day. On the other hand, at the end of the day t is at station B and it can be used the next day for the second route. Similarly, t' is now at A and it can be used for the first route on the second day. Hence, both trains come back to their original position (i.e. at the same station at the same time) after *two days*.

In both examples, we can represent the train assignment as a cycle followed by the train(s): connect every arrival of one route with the departure of the other one (these edges represent waits within a station). If the arrival endpoint precedes the departure we have a wait within the same day; otherwise the edge represents an *overnight wait*. Then, the *length* of a cycle, measured in days, is defined as the sum of the waiting times between consecutive routes and the traveling times of all routes on the cycle (notice that every cycle takes at least one day). In general, it is possible to have cycles of several days and several routes (see the example in Figure 2).

There is a precise relationship between the cycle length and the number of trains: if a cycle takes k days, then k different trains are needed to serve the routes in that cycle within the same day. We can therefore define the following optimization problem:

BASIC ROLLING STOCK ROSTERING (RSR)

Instance: A set of stations $S = \{s_1, s_2, \dots, s_m\}$, and daily train routes, $R = \{r_1, r_2, \dots, r_n\}$. Each route r_i consists of a departure event (dsr_i, dtr_i) and an arrival event (asr_i, atr_i) , where dsr_i and asr_i represent departure and arrival stations of route r_i , and dtr_i and atr_i represent departure and arrival times.

Solution: A collection of ordered sets of routes. Each ordered set represents a cycle to be followed by at least one train: a route r_i precedes a route r_j if r_i and r_j are serviced consecutively by the same train; this is possible only if $asr_i = dsr_j$. Each route must occur in exactly¹ one of the ordered sets. We also call these sets cycles.

Cost: The number of trains needed, that is, the sum over all the cycles of the length (in number of days) of each cycle.

Note that an instance of RSR has a solution if and only if the number of arrival events equals the number of departure events at each station.

1.2 Model Variations and Assumptions

In addition to the basic model, we consider variants in which *empty movements* (deadheading) and/or *maintenance* are allowed or needed:

Empty movements allowed. For every pair of stations, we are given in the input the time for an empty, unscheduled train movement, from one station to the other one. The cycles may contain some of these empty movements.

¹ This imposes some restrictions on which kind of solutions we allow. In Section 2 we discuss this issue in detail.

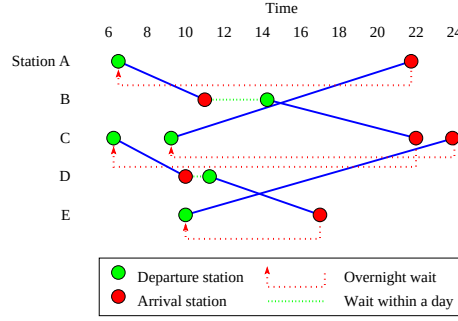


Fig. 2. A cycle of four days

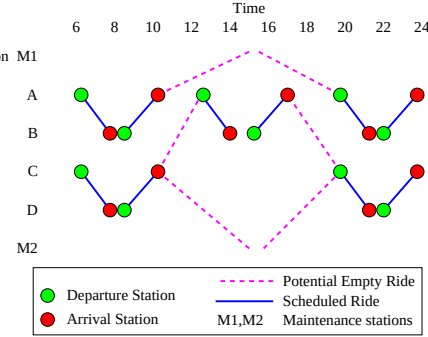


Fig. 3. An instance with empty movements and maintenance stations

Maintenance required. In the input, a (nonempty) subset of the stations is designated as *maintenance stations*. In order to be maintained, every train must eventually (periodically) pass through some maintenance station. So, every output cycle must contain a maintenance station.

Variants of RSR in which we allow empty movements, or require maintenance, or both, are denoted by RSR-E, RSR-M, and RSR-ME, respectively. For all those problems, we can further consider different possible costs to minimize. Further, we expect that an empty train movement is more expensive than simply waiting within a station, even if they take the same amount of time, due to added depreciation for track and train wear and repairs, fuel and labor costs, etc.

In defining the optimization problems we are implicitly making some assumptions. We discuss them in more detail in the sequel.

(Implicit) Assumptions. In this work, we only consider problems where all trains are identical, that is, any train can be used for any route. Two routes with the same departure and arrival stations may need different amounts of time, due to different paths taken. The latter is of no concern to us, since we do not take into account the intermediate stops between the departure and the arrival station; they need not even be specified in the input. We also address the case of routes which take more than one day: for these, there will sometimes be two (or more) trains on the same route at the same time, having started on different days.

We assume that maintenance is performed *instantaneously* as trains pass through the maintenance stations. This assumption is more realistic than it may seem: for instance, SBB keeps an inventory of about 10% extra trains to replace others in need of repair. By rotating these trains in and out of active duty, we can simplify maintenance scheduling, replacing an unmaintained train by a maintained one, once they are at the same station.

Trains do not need to pause between routes: we assume that if one route arrives at a station by the time another route leaves from that station, then one train can service both of those routes. Commonly, several minutes are needed

between routes to prepare the train for the second route. For the problems we consider, we can ‘pad’ all departure times by several minutes, and then ignore the need for preparation time, which will make the assumption true.

Finally, we are imposing a particular structure to the solutions, due to the fact that every route occurs in exactly one cycle. As we will see (Section 2) this, in almost all cases, does not affect optimality (w.r.t. more general solutions) and it allows for train assignments that are simpler to understand.

1.3 Related Previous Work

The simplest version of rolling stock rostering, where we only want to minimize the number of trains needed to run a given schedule, is known as the minimum fleet size problem [2]. Dantzig and Fulkerson [7] propose the first solution that models the problem as a minimum cost circulation problem. A number of survey articles [8,4] discuss this simple problem and more complex variations. Because the realistic problem variants have quite a few different objectives and a lot of constraints, the only resort is to engineer a heuristic solution. To this end, a wealth of heuristic approaches have been tried, from branch and bound, branch and cut, linear programming and relaxation, to simulated annealing, to name but a few [14,3,5,13,10,11]. Experiments show that in many of these cases, the obtained solutions for random data or even for real inputs come close to the optimum and sometimes even reach it. In an effort to come up with a guarantee for the quality of a solution, we are trying to understand the inherent approximation complexity of the problem; no such study has been reported in the literature thus far. In the process of our study, we also improve the runtime for the simplest problem version, RSR with no extra constraints [8,2].

1.4 Our Contribution

In Section 2, we show that our definition of the problem(s) imposes added structure on the solutions: the way two routes are combined within a cycle is the same every day. Although this is what has been done in practice so far, up to our knowledge, the optimality of such solutions (w.r.t. more general ones) has never been investigated. We prove that, for all but one (RSR-ME) of our problems, this optimality holds, and show why this is not the case for RSR-ME.

In Section 3, we present an $O(n \log n)$ -time algorithm for the basic rostering problem without maintenance, thus improving the running time of existing solutions for this version.

We consider maintenance in Section 4. First, we show that (even with our simplifications) both RSR-M and RSR-ME are APX-hard, that is, there exists a constant $r > 1$ for which even approximating the problem within a factor r is NP-hard. Then, in Section 4.2 we look at approximation algorithms and we show that RSR-M and RSR-ME have a polynomial-time 2- and 5-approximation algorithm, respectively. Finally, we show that the algorithms perform provably better if some additional hypotheses on the input hold.

Due to space limitations, most of the proofs are omitted and can be found in the full version of this paper [9].

2 Periodicity in the Solutions

Here we discuss the structure of our solution. We study solutions which look the same each day: if on day one, one train consecutively services routes r and r' , then whichever train services route r on any day will next service r' . We will call these *one day assignments*. While it may seem obvious that a periodic daily schedule can have an optimum train assignment (w.r.t. number of trains) which looks the same each day, this is not necessarily the case. We prove that one day assignments give best possible solutions for RSR, RSR-M, and RSR-E. For RSR-ME, however, we now give an example where any one day assignment uses more trains than a solution without this restriction.

Consider the RSR-ME example in Figure 3. In any one day assignment, at least two trains are needed, because two train routes are simultaneously scheduled. Clearly, just after the first 4 or last 4 routes, there will be a train at A and C. With only two trains, the only way to maintain the train at A without missing a scheduled route is to make an empty movement from A to M1 at the same time as an empty movement from C to A, and then move both trains back after the two mid-day routes. (We can assume that all empty movements not shown in Figure 3 are too lengthy to help.) Similarly, to service the train at C, it can make a mid-day unscheduled movement to M2, while the train at A services the two mid-day routes. By this argument, in a one day assignment, only one of the two trains can be maintained, and so 3 trains are needed. A two day assignment does not have this problem: we can alternate between maintaining the two trains every other day, as mentioned above, and satisfy the routes with just two trains.

The following result clarifies the relationship between one day assignments and more general forms of solutions (multiple day assignments). Its proof is based on several non-trivial results about the periodicity of general solutions and the cycle structure of one day assignments (see [9]).

Theorem 1. *For RSR, RSR-E, and RSR-M, considering only the cost of train ownership (and extra costs for empty movements in RSR-E), the best one day assignment is optimal for any solution.*

Theorem 1 tells us that our one day assignment output restriction will not increase our optimal solution costs for RSR, RSR-E, and RSR-M.

By employing aperiodic solutions that decrease the frequency of maintenance over time, the average daily cost of an RSR-ME solution can be made arbitrarily close to that of a solution without any maintenance. Therefore, it is meaningful to consider RSR-ME solutions restricted to one-day assignments, as we do.

3 Fast Basic Rostering

We return to the simplest problem version, RSR. This problem is sometimes called the minimum fleet size problem [2], and polynomial-time solutions are

known based on minimum cost bipartite matching, or flow problems. Here, each route is modeled by 2 vertices, for the arrival and departure events of the route.

An arrival vertex is connected to a departure vertex if they represent the same station, and the cost of the edge is set to the time between the arrival and the departure. A minimum perfect matching then minimizes the total waiting time of trains, because the time spent by trains performing scheduled routes is fixed. This minimizes the total number of trains used by the system.

First we notice that this basic problem can be solved more efficiently, without using the machinery of the minimum perfect bipartite matching algorithm. Our main task is to calculate, for each station, the number of trains at the start-of-day. We can then just make an arbitrary train assignment for one day which covers all scheduled routes, and this will be an optimal solution by Theorem 1.

The calculation starts by creating a list of all routes into or out of each station s . Then, we order all arrivals and departures within each station s . For each station s , we linearly (by time) search through all arrivals and departures from that station, and calculate the minimum number of trains such that the station begins each day with enough trains so that it will never have a negative number due to departures throughout the day. Finally, for any route out of a station, we pick any train that is currently in the station and assign it to that route. All steps, except for sorting, take linear time; altogether we get:

Theorem 2. *The RSR problem can be solved in $O(n \log n)$ time.*

The minimum cost perfect bipartite matching approach can be generalized to RSR-E by adding edges corresponding to all possible empty train movements to the bipartite graph, leading to a polynomial-time solution [2,7].

4 Rostering with Maintenance

We show that whether or not empty train movements are allowed, trying to minimize costs is hard once maintenance is needed. First, we prove that RSR-M and RSR-ME are APX-hard, thus implying that even approximating the problem within some constant factor $r > 1$ is NP-hard. Then, we present a 2-approximation algorithm for RSR-M and a 5-approximation algorithm for RSR-ME.

4.1 Hardness

We present an approximation preserving reduction from the minimum vertex cover problem on cubic graphs (i.e., graphs with maximum degree 3) to RSR-M. Since this restriction of minimum vertex cover is APX-hard [12,1], our reduction implies the same hardness result for RSR-M.

For an undirected graph $G = (V, E)$, a set $K \subseteq V$ is called a vertex cover if it contains at least one endpoint of every edge in E . Let thus $G = (V, E)$ be an undirected graph with maximum degree 3. We set $n = |V|$ and $m = |E|$. The reduction works as follows:

- We create a single maintenance station s_M and a station s_j for each vertex $v_j \in V$.
- For every edge $e \in E$, with $e = \{v_i, v_j\}$, we create an *edge cycle* composed of two routes: one going from station s_i to station s_j and the other one going back from s_j to s_i (we will specify their arrival and departure time in the sequel).
- We add a *maintenance cycle* consisting of routes from s_M to s_1 , from s_1 to s_2 , from s_2 to s_3 , ..., from s_{n-1} to s_n and from s_n to s_M .

Then, we want to assign departure and arrival time to the routes on each of these cycles so that the following properties are fulfilled:

- The optimal solution *without maintenance* simply consists of the union of the edge cycles and of the maintenance cycle. We call such a solution *trivial solution* and we denote its cost by C_{triv} .
- If we instead require maintenance, then there exists a solution of cost $C_{triv} + k$ if and only if G has a vertex cover of size k . Moreover, we force any feasible solution to consist of a single cycle by having only one route arriving at s_M and one route departing from s_M .

The main idea underlying the construction is the following. In order to transform the solution without maintenance into a solution with maintenance, we have to change the solution at some stations so that we can “merge” the edge cycles and the maintenance cycle into a single one. Moreover, by using a solution different from the trivial one at a station s_j , *all* the cycles passing through s_j can be combined into one cycle at an extra cost of one day (viewed differently, this corresponds to employing an extra train). Notice that the edge cycles passing through s_j correspond to those edges of G that have v_j as one endpoint. Intuitively, the station s_j “covers” all those edge cycles, that is, it allows to merge them with the maintenance cycle.

An example of the reduction. In order to illustrate our reduction from minimum vertex cover to the RSR-M problem, an example is given in Figure 4: the top part shows the graph G , while the lower part shows the edge and the maintenance cycles along with the departure/arrival times. In particular, all the routes take two days² and all the routes in the same cycle have the same arrival and departure time (for the sake of legibility, Figure 4 shows the departure and the arrival time – 12:00 and 12:00+2d, respectively – of only one route in the maintenance cycle). Hence, in every cycle, the arrival time of one route matches the departure time of the next route (on a different day due to the 2 days traveling time). Moreover, all the routes in the maintenance cycle have the same departure and arrival time.

Let us observe that the trivial solution consists of the maintenance cycle (yielding $2(n+1) = 10$ days) and of $m = 5$ edge cycles (i.e. $4m = 20$ days). So, we have $C_{triv} = 30$. However, the trivial solution is not feasible since none of

² We denote this by “+2d” in the arrival time.

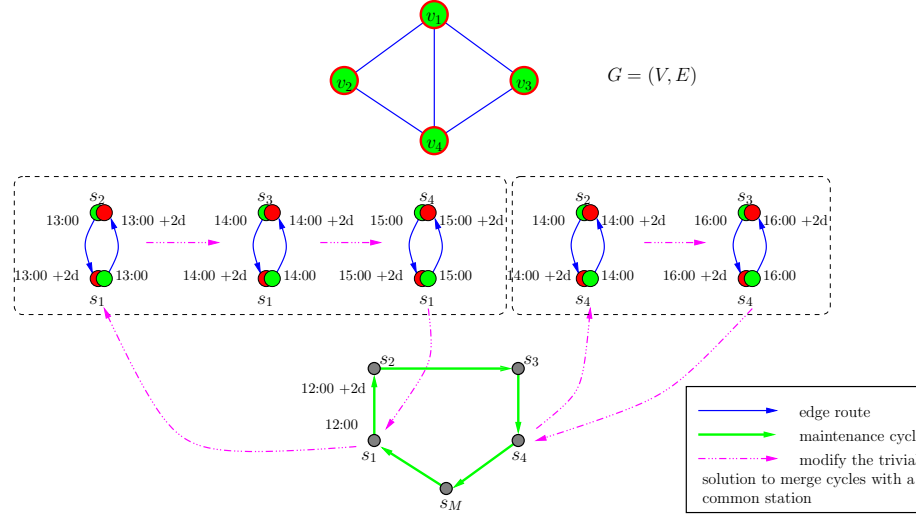


Fig. 4. An instance $G = (V, E)$ of minimum vertex cover (top) and the corresponding instance of RSR-M (bottom) along with a modified feasible solution corresponding to the vertex cover $K = \{v_1, v_4\}$

the edge cycles passes through s_M . To get a feasible solution, we must modify it in at least two stations, for example at stations s_1 and s_4 (corresponding to the vertex cover consisting of v_1 and v_4). We represent such changes in Figure 4 with dashed arrows: an arrow from a station in the maintenance cycle to an edge cycle means that we “leave” the maintenance cycle at this station and we “enter” in the corresponding edge cycle at the same station; an arrow between two edge cycles means that we follow them in the order given by the arrow. (Notice that there is always a station common to both cycles.) It is easy to verify that the cycle represented in Figure 4 has total length equal to $C_{triv} + 2 = 32$. (Whenever we leave the maintenance cycle and then we come back to the same station, we add one day.) $\square$

We now formally state the properties that the arrival and departure time of each route must satisfy in the reduction. In particular, the length of each route is 2 days and the following properties hold:

- P1.** Every route in the maintenance cycle has departure time equal to 12:00 (and thus arrival time 12:00 + 2d).
- P2.** Each edge cycle is formed by two routes having the same arrival and departure time. Thus, every edge cycle has length 4. Moreover, each departure time is greater than 12:00 and any two routes with a common station but in different cycles have different departure/arrival times.

It is worth to observe that property **P2** can be guaranteed using departure times of the form ‘ $t:00$ ’, where t is some integer in $[13, 24]$. Indeed, since G has

maximum degree 3, for every edge cycle, there are at most 4 other edge cycles with a common station. So, we are guaranteed that we can assign a departure time t to every edge cycle one by one (at each step there are only 4 values in [13, 24] that we cannot use). Hence, the construction can be performed in polynomial time.

We first observe that, if we ignore the maintenance constraint, then the optimal solution is given by the union of the edge cycles and the maintenance cycle. This is due to the fact that in every cycle the arrival time of one route matches the departure time of the next route (of course two days later). Since every edge cycle has length 4 days and the maintenance cycle has a duration of $2(n + 1)$ days, we have that the cost of this solution is

$$C_{triv} = 2(n + 1) + \sum_{e \in E} 4 = 2n + 4m + 2. \quad (1)$$

Lemma 1. *A feasible solution of cost at most $C_{triv} + k$ to the constructed instance of RSR-M can be converted into a vertex cover of size at most k in the original graph in polynomial time, and vice versa.*

By using the above lemma, it is possible to prove that our construction is a PTAS-reduction [6] from minimum vertex cover on cubic graphs to RSR-M. Moreover, the above lemma (and thus the reduction) also applies to RSR-ME. Since minimum vertex cover is known to be APX-complete in graphs with degree bounded by Δ , for any $\Delta \geq 3$ [12,1], the following result holds:

Theorem 3. *The RSR-M and the RSR-ME problems are APX-hard.*

We remark that, for RSR-M, the reduction above can be modified so that the routes have more realistic travel times, e.g. so that each route takes approximately one hour. However, we presented the reduction using two-day routes in order to easily generalize the result to the case of empty movements.

4.2 Approximation Algorithms

We present a simple 2-approximation algorithm for the RSR-M problem. First, we ignore the maintenance constraint and compute a minimum-cost partition of the given routes into cycles using the algorithm of Section 3. The solution we get may contain cycles that do not pass through a maintenance station. As long as there exists a cycle in our solution that does not go through a maintenance station, we merge this cycle with some other cycle. Each of these steps increases the cost of the current solution by at most one day: one overnight wait is sufficient to combine two cycles with a common station.

If at some time step there is a cycle that does not pass through a maintenance station, but no combination with another cycle is possible, then the given instance does not have a feasible solution (because the stations on this cycle do not appear on any route outside the cycle). Otherwise, every cycle goes through a maintenance station in the end, and we obtain a feasible solution.

Let k be the number of cycles in the initial solution (the minimum-cost solution ignoring the maintenance constraint). The cost C_{triv} of this initial solution is a lower bound on the cost OPT of an optimal feasible solution. Besides, the cost of the initial solution is at least k , since each cycle has cost at least 1. Each application of the transformation combines at least 2 cycles, so there can be at most $k - 1$ such transformations. Since each of them yields an extra cost of 1, the total cost of the final feasible solution is at most $C_{triv} + (k - 1) \leq 2 \cdot OPT$.

Theorem 4. *The RSR-M problem admits a polynomial-time 2-approximation algorithm.*

Now we present an approximation algorithm for the problem RSR-ME. We make the (reasonable) assumption that the costs for empty train movements are symmetric, i.e., the cost for an empty movement from s_1 to s_2 is the same as the cost for an empty movement from s_2 to s_1 .

First, we apply the algorithm of Theorem 4 and combine cycles containing a common station as long as possible (also combining two cycles not containing a maintenance station). Then, if a cycle does not pass through a maintenance station, it passes only through stations that do not occur on any other cycle. Therefore, we must use empty movements to combine such a cycle with another cycle.

We add empty movements by repeating the following step until the solution is feasible. Let σ be a cycle that does not pass through a maintenance station. For a station s on σ and a station s' not on σ , define $c(s, s')$ to be the sum of the cost of an empty movement from s to s' and an empty movement from s' to s . Select s and s' such that $c(s, s')$ is minimized. Add an empty movement from s to s' and one from s' to s and put extra trains at s and s' . Now we can assign the trains arriving at s and s' to outgoing routes from s and s' such that all cycles passing through s and s' are combined into one cycle.

Theorem 5. *The RSR-ME problem, restricted to empty movements with symmetric costs, admits a polynomial-time 5-approximation algorithm.*

By combining our approximation results with Theorem 3, we obtain the following result.

Corollary 1. *The RSR-M and the RSR-ME problem are APX-complete.*

The factor 2 in Theorem 4 comes from the fact that combining 2 cycles requires at most 1 extra train, which will only double the total solution cost if every train is in a one day cycle to begin with, and no two trains are ever in any station at once. In general, we expect our approximation algorithms to give better performance when applied to real data. The following theorem, providing a better analysis of the 2-approximation algorithm for RSR-M, gives a strong indication of this.

Theorem 6. *Consider an RSR instance, and let an optimal solution have a total of t trains. Also, let s be the number of stations and c be the minimum*

number of cycles possible for an optimal one day assignment to the RSR problem. For the same instance, but with maintenance stations specified, we can give a $\min\{t + c - 1, t + s - 1\}/t$ approximation for RSR-M.

In the SBB data we look at, we see that we need over 100 trains to cover all routes, but we only have about 40 terminal stations. Further, in seeing that many trains are often within a station at once, this is also an indication that the number of cycles, c can be much less than the number of trains. Thus, we can prove that on these instances, our approximation factor will be significantly less than the worst case bound. In fact, for the SBB data, we find that we can combine all train movements into one cycle, without increasing the solution cost, which takes more than 100 days to complete. (This does include some unscheduled movements which they perform.) This is a very good indication that in real problem instances, we can hope to find maintenance solutions within a small percentage of optimal.

Acknowledgments

We are grateful to Jean-Claude Strüby and Peter Hostettler from SBB for several useful discussions and for providing us with SBB data.

References

1. P. Alimonti and V. Kann. Hardness of approximating problems on cubic graphs. In *Proc. 3rd Italian Conference on Algorithms and Complexity*, LNCS 1203, pages 288–298, Berlin, 1997. Springer-Verlag. 396, 399
2. A. Bertossi, P. Carraresi, and G. Gallo. On some matching problems arising in vehicle scheduling models. *Networks*, 17:271–281, 1987. 394, 395, 396
3. P. Brucker, J.L. Hurink, and T. Rolfes. Routing of railway carriages: A case study. *Memorandum No. 1498, University of Twente, Fac. of Mathematical Sciences*, 1999. 394
4. M.R. Bussieck, T. Winter, and U.T. Zimmermann. Discrete optimization in public rail transport. *Mathematical Programming*, 79:415–444, 1997. 394
5. G. Carpaneto, M. Dell’Amico, M. Fischetti, and P. Toth. A branch and bound algorithm for the multiple depot vehicle scheduling problem. *Networks*, 19:531–548, 1989. 394
6. P. Crescenzi and L. Trevisan. On approximation scheme preserving reducibility and its applications. In *Proc. 14th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, LNCS 880, pages 330–341, Berlin, 1994. Springer-Verlag. 399
7. G. Dantzig and D. Fulkerson. Minimizing the number of tankers to meet a fixed schedule. *Nav. Res. Logistics Q.*, 1:217–222, 1954. 394, 396
8. J. Desrosiers, Y. Dumas, M.M. Solomon, and F. Soumis. *Time Constrained Routing and Scheduling*. Elsevier, 1995. 394
9. T. Erlebach, M. Gantenbein, D. Hürlimann, G. Neyer, A. Pagourtzis, P. Penna, K. Schlude, K. Steinhöfel, D.S. Taylor, and P. Widmayer. On the complexity of train assignment problems. Technical report, Swiss Federal Institute of Technology Zürich (ETH), 2001. Available at <http://www.inf.ethz.ch/>. 395

10. R. Freling, J.M.P. Paixão, and A.P.M. Wagelmans. Models and algorithms for vehicle scheduling. *Report 9562/A, Econometric Institute, Erasmus University Rotterdam*, 1999. [394](#)
11. A. Löbel. Optimal vehicle scheduling in public transit. *PhD thesis, TU Berlin*, 1998. [394](#)
12. C. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991. [396](#), [399](#)
13. C. Ribeiro and F. Soumis. A column generation approach to the multiple-depot vehicle scheduling problem. *Operations Research*, 42(1):41–52, 1994. [394](#)
14. A. Schrijver. Minimum circulation of railway stock. *CWI Quarterly*, 6(3):205–217, 1993. [394](#)

A Combinatorial Toolbox for Protein Sequence Design and Landscape Analysis in the Grand Canonical Model

James Aspnes^{1*}, Julia Hartling², Ming-Yang Kao^{3**}, Junhyong Kim^{2,4,5***},
and Gauri Shah^{1†}

¹ Department of Computer Science, Yale University
New Haven, CT 06520, USA
aspnes@cs.yale.edu
gauri.shah@yale.edu

² Department of Ecology and Evolutionary Biology, Yale University
julia.kreychman@yale.edu

³ Department of EECS, Tufts University
Medford, MA 02155, USA
kao@eecs.tufts.edu

⁴ Department of Molecular, Cellular, and Developmental Biology, Yale University
junhyong.kim@yale.edu

⁵ Department of Statistics, Yale University

Abstract. In modern biology, one of the most important research problems is to understand how protein sequences fold into their native 3D structures. To investigate this problem at a high level, one wishes to analyze the *protein landscapes*, i.e., the structures of the space of all protein sequences and their native 3D structures. Perhaps the most basic computational problem at this level is to take a target 3D structure as input and *design* a fittest protein sequence with respect to one or more fitness functions of the target 3D structure. We develop a toolbox of combinatorial techniques for protein landscape analysis in the Grand Canonical model of Sun, Brem, Chan, and Dill. The toolbox is based on linear programming, network flow, and a linear-size representation of all minimum cuts of a network. It not only substantially expands the network flow technique for protein sequence design in Kleinberg’s seminal work but also is applicable to a considerably broader collection of computational problems than those considered by Kleinberg. We have used this toolbox to obtain a number of efficient algorithms and hardness results. We have further used the algorithms to analyze 3D structures drawn from the Protein Data Bank and have discovered some novel relationships between such native 3D structures and the Grand Canonical model.

* NSF Grant CCR-9820888.

** NSF Grants CCR-9531028 and EIA-0112934.

*** Merck Genome Research Institute Grant and NSF Grant DEB-9806570.

† NSF Grant CCR-9820888.

1 Introduction

In modern biology, one of the most important research problems is to understand how protein sequences fold into their native 3D structures. This problem can be investigated at two complementary levels. At a low level, one wishes to determine how an individual protein sequence folds. A fundamental computational problem at this level is to take a protein sequence as input and find its native 3D structure. This problem is sometimes referred to as the protein *structure prediction* problem and has been shown to be NP-hard (e.g., [1]). At a high level, one wishes to analyze the *protein landscapes*, i.e., the structures of the space of all protein sequences and their native 3D structures. Perhaps the most basic computational problem at this level is to take a target 3D structure as input and ask for a fittest protein sequence with respect to one or more fitness functions of the target 3D structure. This problem has been called the protein *sequence design* problem and has been investigated in a number of studies (e.g., [2]).

The focus of this paper is on protein landscape analysis, for which several quantitative models have been proposed in the literature (e.g., [9]). As some recent studies on this topic have done (e.g., [6]), this paper employs the Grand Canonical (GC) model of Sun, Brem, Chan, and Dill [9], whose definition is given in Section 2. Generally speaking, the model is specified by (1) a 3D geometric representation of a target protein 3D structure with n amino acid residues, (2) a *binary folding code* in which the amino acids are classified as *hydrophobic* (H) or *polar* (P), and (3) a fitness function Φ defined in terms of the target 3D structure that favors protein sequences with a dense hydrophobic core and with few solvent-exposed hydrophobic residues.

In this paper, we develop a toolbox of combinatorial techniques for protein landscape analysis based on linear programming, network flow, and a linear-size representation of all minimum cuts of a network [7]. This toolbox not only substantially expands the network flow technique for protein sequence design in Kleinberg’s seminal paper [6] but also is applicable to a considerably broader collection of computational problems than those considered by Kleinberg. We have used this toolbox to obtain a number of efficient algorithms and hardness results. We have further used the algorithms to analyze 3D structures drawn from Protein Data Bank at <http://www.rcsb.org/pdb> and have discovered some novel relationships between such native 3D structures and the Grand Canonical model (Section 6). Specifically, we report new results on the following problems, where Δ is the number of terms in the fitness function or functions as further defined in Section 3. Many of the results depend on computing a maximum network flow in a graph of size $O(\Delta)$; in most cases, this network flow only needs to be computed once for each fitness function Φ .

- P1 Given a 3D structure, find all its fittest protein sequences. Note that there can be exponentially many fittest protein sequences. We show that these protein sequences together have a representation of size $O(\Delta)$ that can be computed in $O(\Delta)$ time after a certain maximum network flow is computed

- (Theorem 1), and that individual fittest protein sequences can be generated from this representation in $O(n)$ time per sequence (Theorem 5).
- P2 Given f 3D structures, find the set of all protein sequences that are the fittest simultaneously for all these 3D structures. This problem takes $O(\Delta)$ time after f maximum network flow computations (Theorem 4).
- P3 Given a protein sequence $\hat{x}$ and its native 3D structure, find the set of all fittest protein sequences that are also the most (or least) similar to $\hat{x}$ in terms of unweighted (or weighted) Hamming distances. This problem takes $O(\Delta)$ time after a certain maximum network flow is computed (Theorem 3).
- P4 Count the number of protein sequences in the solution to each of Problems P1, P2, and P3. These counting problems are computationally hard (Theorem 11).
- P5 Given a 3D structure and a bound ϵ , enumerate the protein sequences whose fitness function values are within an additive factor ϵ of that of the fittest protein sequences. This problem takes polynomial time to generate each desired protein sequence (Theorem 8).
- P6 Given a 3D structure, find the largest possible unweighted (or weighted) Hamming distance between any two fittest protein sequences. This problem takes $O(\Delta)$ time after a certain maximum network flow is computed (Theorem 6).
- P7 Given a protein sequence $\hat{x}$ and its native 3D structure, find the average unweighted (or weighted) Hamming distance between $\hat{x}$ and the fittest protein sequences for the 3D structure. This problem is computationally hard (Theorem 11).
- P8 Given a protein sequence $\hat{x}$, its native 3D structure, and two unweighted Hamming distances d_1 and d_2 , find a fittest protein sequence whose distance from $\hat{x}$ is also between d_1 and d_2 . This problem is computationally hard (Theorem 12(1)).
- P9 Given a protein sequence $\hat{x}$, its native 3D structure, and an unweighted Hamming distance d , find the fittest among the protein sequences which are at distance d from $\hat{x}$. This problem is computationally hard (Theorem 12(2)). We have a polynomial-time approximation algorithm for this problem (Theorem 9).
- P10 Given a protein sequence $\hat{x}$ and its native 3D structure, find all the ratios between the scaling factors α and β in Equation 1 in Section 2 for the GC model such that the smallest possible unweighted (or weighted) Hamming distance between $\hat{x}$ and any fittest protein sequence is minimized over all possible α and β . (This is a problem of tuning the GC model.) We have a polynomial-time algorithm for this problem (Theorem 10).
- P11 Given a 3D structure, determine whether the fittest protein sequences are *connected*, i.e., whether they can mutate into each other through allowable mutations, such as point mutations, while the intermediate protein sequences all remain the fittest (e.g., [8]). This problem takes $O(\Delta)$ time after a certain maximum network flow is computed (Theorem 7).
- P12 Given a 3D structure, in the case that the set of all fittest protein sequences is not connected, determine whether two given fittest protein sequences are

connected. This problem takes $O(\Delta)$ time after a certain maximum network flow is computed (Theorem 7).

P13 Given a 3D structure, find the smallest set of allowable mutations with respect to which the fittest protein sequences (or two given fittest protein sequences) are connected. This problem takes $O(\Delta)$ time after a certain maximum network flow is computed (Theorem 7).

Previously, Sun *et al.* [9] developed a heuristic algorithm to search the space of protein sequences for a fittest protein sequence without a guarantee of optimality or near-optimality. Hart [5] subsequently raised the computational tractability of constructing a single fittest protein sequence as an open question. Kleinberg [6] gave the first polynomial-time algorithm for this problem, which is based on network flow. In contrast, Problem P1 asks for all fittest protein sequences and yet can be solved with the same time complexity. Kleinberg also formulated more general versions of Problems P11 and P12 by extending the fitness function to a submodular function and gave polynomial-time algorithms. Our formulations of these two problems and Problem P13 are directly based on the fitness function of the GC model; furthermore, as is true with several other problems above, once a solution to Problem P1 is obtained, we can solve these three problems in $O(\Delta)$ time. Among the above thirteen problems, those not yet mentioned in this comparison were not considered by Kleinberg.

The remainder of this paper is organized as follows. Section 2 defines the GC model and states the basic computational assumptions. Section 3 describes our three basic tools based on linear programming, network flow, and an $O(\Delta)$ -size representation of minimum cuts. Section 4 extends these tools to optimize multiple objectives, analyze the structures of the space of all fittest protein sequences, and generate near-fittest protein sequences. Section 5 gives some hardness results related to counting fittest protein sequences and finding fittest protein sequences under additional restrictions. Finally, Section 6 discusses our analysis of empirical 3D structures from the Protein Data Bank.

Proofs of our results are omitted due to space limitations. They may be found in the full version of this paper, which is deposited in the Computing Research Repository as <http://www.arxiv.org/abs/cs.CE/0101015>.

2 The Grand Canonical Model

The Original Model Throughout this paper, all protein sequences are of n residues, unless explicitly stated otherwise. The GC model is specified by a fitness function Φ over all possible protein sequences x with respect to a given 3D structure of n residues [9]. In the model, to design a protein sequence x is to specify which residues are hydrophobic (H) and which ones are polar (P). Thus, we model x as a binary sequence $x_1, \dots, x_n$ or equivalently as a binary vector $(x_1, \dots, x_n)$, where the i -th residue in x is H (respectively, P) if and only if $x_i = 1$ (respectively, 0). Then, $\Phi(x)$ is defined as follows, where the smaller $\Phi(x)$ is, the fitter x is, as the definition is motivated by the requirements that H

residues in x (1) should have low solvent-accessible surface area and (2) should be close to one another in space to form a compact hydrophobic core.

$$\Phi(x) = \alpha \sum_{i,j \in H(x), i < j-2} g(d_{i,j}) + \beta \sum_{i \in H(x)} s_i \quad (1)$$

$$= \alpha \sum_{i < j-2} g(d_{i,j}) x_i x_j + \beta \sum_i s_i x_i, \text{ where} \quad (2)$$

- $H(x) = \{i \mid x_i = 1\}$,
- the scaling parameters $\alpha < 0$ and $\beta > 0$ have default values -2 and $\frac{1}{3}$ respectively and may require tuning for specific applications (see Section 4),
- $s_i \geq 0$ is the area of the solvent-accessible contact surface for the residue (in Å) [4],
- $d_{i,j} > 0$ is the distance between the residues i and j (in Å), and
- g is a sigmoidal function, defined by

$$g = \begin{cases} \frac{1}{1+\exp(d_{i,j}-6.5)} & \text{when } d_{i,j} \leq 6.5 \\ 0 & \text{when } d_{i,j} > 6.5. \end{cases}$$

Extending the Model with Computational Assumptions Let $\text{opt}(\Phi)$ be the set of all protein sequences x that minimize Φ . This paper is generally concerned with the structure of $\text{opt}(\Phi)$. Our computational problems assume that Φ is given as input; in other words, the computations of $\alpha, \beta, s_i, g(d_{i,j})$ are not included in the problems. Also, for the sake of computational generality and notational simplicity, we assume that α may be any nonpositive number, β any nonnegative number, s_i any arbitrary number, and $g(d_{i,j})$ any arbitrary non-negative number; and that the terms $g(d_{i,j})$ may range over $1 \leq i < j \leq n$, unless explicitly stated otherwise. Thus, in the full generality of these assumptions, Φ need not correspond to an actual protein 3D structure. Note that the relaxation that s_i is any number is technically useful for finding Φ -minimizing protein sequences x that satisfy additional constraints.

We write $a_{i,j} = -\alpha \cdot g(d_{i,j}) \geq 0$ and $b_i = \beta \cdot s_i$ and further assume that the coefficients $a_{i,j}$ and b_i are rational with some common denominator, that these coefficients are expressed with a polynomial number of bits, and that arithmetic operations on these coefficients take constant time.

With these assumptions, we define the following sets of specific assumptions about Φ to be used at different places of this paper.

- F1 Let $\Phi(x) = -\sum_{1 \leq i < j \leq n} a_{i,j} x_i x_j + \sum_{1 \leq i \leq n} b_i x_i$, where $a_{i,j} \geq 0$, b_i is arbitrary, and m of the coefficients $a_{i,j}$ are nonzero. Let $\Delta = n + m$.
- F2 For each $\beta \geq 0$, let $\Phi_\beta(x) = -\sum_{1 \leq i < j \leq n} a_{i,j} x_i x_j + \beta \sum_{1 \leq i \leq n} s_i x_i$, where $a_{i,j} \geq 0$, $s_i \geq 0$, and m of the coefficients $a_{i,j}$ are nonzero. Let $\Delta = n + m$.
- F3 For each ℓ from 1 to f , define the ℓ -th fitness function $\Phi^\ell(x) = -\sum_{1 \leq i < j \leq n} a_{i,j}^\ell x_i x_j + \sum_{1 \leq i \leq n} b_i^\ell x_i$, where $a_{i,j}^\ell \geq 0$ and b_i^ℓ is arbitrary. Let $\Delta = fn^2$.

Sometimes we measure the dissimilarity between a fittest protein sequence x and a target protein sequence $\hat{x}$ in terms of Hamming distance. This distance is essentially the count of the positions i where $x_i \neq \hat{x}_i$ and can be measured in two ways. The *unweighted* Hamming distance is $|x - \hat{x}|$, where $|y|$ denotes the *norm* of vector y , i.e., $\sum_{i=1}^n |y_i|$. The *weighted* Hamming distance is $\sum_{i=1}^n w_i \cdot |x_i - \hat{x}_i|$. Throughout this paper, the weights $w_1, \dots, w_n$ are all arbitrary unless explicitly stated otherwise.

3 Three Basic Tools

This section describes our basic tools for computing fittest and near-fittest protein sequences. For instance, Lemma 1 gives a representation of the problem of minimizing Φ as a linear program. Lemma 2 further gives a representation of this problem as a minimum-cut problem, which generalizes a similar representation of Kleinberg [6]. Theorem 1 gives a compact representation of the space $\text{opt}(\Phi)$ using a Picard-Queyranne graph [7].

Linear Programming From Equation 2, minimizing $\Phi(x)$ is an optimization problem in quadratic programming. Fortunately, because all the coefficients $a_{i,j}$ are nonnegative, it can be converted to a linear program, as shown in Lemma 1.

Lemma 1 (characterizing Φ via linear program). *Let Φ be as defined in Assumption F1. Consider the following linear program whose variables consist of the variables x_i , together with new variables $y_{i,j}$ for all i, j with $a_{i,j} \neq 0$:*

$$\begin{aligned} & \text{minimize } \Phi'(x, y) = -\sum a_{i,j} y_{i,j} + \sum b_i x_i \\ & \text{subject to} \\ & \left. \begin{array}{l} 0 \leq x_i \leq 1 \quad \forall i \\ 0 \leq y_{i,j} \leq 1 \\ y_{i,j} \leq x_i \\ y_{i,j} \leq x_j \end{array} \right\} \forall i, j : a_{i,j} \neq 0 \end{aligned} \quad (3)$$

There is a one-to-one correspondence that preserves x between the protein sequences that minimize $\Phi(x)$ and the basic optimal solutions to Linear Program (3).

Note that any x_i with a negative coefficient b_i is set to 1 in any optimal solution, as in this case all terms containing x_i have negative coefficients and are minimized when $x_i = 1$. So an alternative to allowing negative coefficients is to prune out any x_i with a negative coefficient. This process must be repeated recursively, since setting x_i to 1 reduces terms of the form $-a_{i,j} x_i x_j$ to $-a_{i,j} x_j$, and may yield more degree-1 terms with negative coefficients. To simplify our discussion, we let the linear program (or, in Section 3, the minimum-cut algorithm) handle this pruning.

Network Flow Recall that an s - t cut is a partition of the nodes of a digraph into two sets V_s and V_t , with $s \in V_s$ and $t \in V_t$. Also, a *minimum s - t cut* is an s - t cut with the smallest possible total capacity of all edges from nodes in V_s to nodes in V_t .

In Kleinberg's original construction [6], $\Phi(x)$ was minimized by solving an s - t minimum cut problem in an appropriate digraph G . Lemma 2 describes a more general construction that includes additional edges (s, v_i) to handle negative values for b_i .

Lemma 2 (characterizing Φ via network flow). *Let Φ be as defined in Assumption F1. Let G^Φ be a graph with a source node s , a sink node t , a node v_i for each i , and a node $u_{i,j}$ for each i, j with $a_{i,j} \neq 0$, for a total of $n + m + 2 = \Delta + 2$ nodes. Let the edge set of G^Φ consist of*

- $(s, u_{i,j})$ for each $u_{i,j}$, with capacity $a_{i,j}$,
- (v_i, t) for each v_i with $b_i > 0$, with capacity b_i ,
- (s, v_i) for each v_i with $b_i < 0$, with capacity $-b_i$, and
- $(u_{i,j}, v_i)$ and $(u_{i,j}, v_j)$, for each $u_{i,j}$, with infinite capacity,

for a total of $\Theta(\Delta)$ edges.

There is a one-to-one correspondence between the minimum s - t cuts in G^Φ and the protein sequences in $\text{opt}(\Phi)$, such that v_i is in the s -component of a cut if and only if $x_i = 1$ in the corresponding protein sequence.

Lemma 3. *Let Φ be as defined in Assumption F1. Given Φ as the input, we can find an $x \in \text{opt}(\Phi)$ in $O(\Delta^2 \log \Delta)$ time.*

A Compact Representation of Minimum Cuts A given Φ may have more than one fittest protein sequence. Theorem 1 shows that $\text{opt}(\Phi)$ can be summarized compactly using the Picard-Queyranne representation of the set of all minimum s - t cuts in a digraph G [7], which is computed by the following steps:

1. computing any maximum flow ϕ in G ;
2. computing strongly connected components in the residual graph G_ϕ whose edge set consists of all edges in G that are not saturated by ϕ , plus edges (v, u) for any edge (u, v) that has nonzero flow in ϕ ;
3. contracting G_ϕ by contracting into single supernodes the set of all nodes reachable from s , the set of all nodes that can reach t , and each strongly connected component in the remaining graph.

The resulting graph $G_{s,t}$ is a dag in which s and t are mapped to distinct supernodes by the contraction. Furthermore, there is a one-to-one correspondence between the minimum s - t cuts in G and the ideals in $G_{s,t}$, where an *ideal* is any node set I with the property that any predecessor of a node in I is also in I .

Lemma 4 (see [7]). *Given a digraph G with designated nodes s and t , there is a graph $G_{s,t}$ together with a mapping κ from $V(G)$ to $V(G_{s,t})$ with the following properties:*

1. $|V(G_{s,t})| \leq |V(G)|$.
2. The node $\kappa(s)$ has out-degree 0 while $\kappa(t)$ has in-degree 0.
3. Given G as the input, $G_{s,t}$ and κ can be computed using one maximum-flow computation and $O(|E(G)|)$ additional work.
4. A partition (V_s, V_t) of $V(G)$ is an s - t minimum cut in G if and only if $V_t = \kappa^{-1}(I)$ for some ideal I of $G_{s,t}$ that contains $\kappa(t)$ but not $\kappa(s)$.

Combining Lemmas 2 and 4 gives the desired compact representation of the space of all fittest protein sequences, as stated in the next theorem.

Theorem 1 (characterizing Φ via a dag). *Let Φ be as defined in Assumption F1. There exists a dag $G_{s,t}^\Phi$ with designated nodes s' and t' and a mapping ρ from $\{1, \dots, n\}$ to $V(G_{s,t}^\Phi)$ with the following properties:*

1. $G_{s,t}^\Phi$ has at most $n + 2$ nodes.
2. Given Φ as the input, $G_{s,t}^\Phi$ and ρ can be computed in $O(\Delta^2 \log \Delta)$ time.
3. There is a one-to-one correspondence between the protein sequences $x \in \text{opt}(\Phi)$ and the ideals of $\hat{G}_{s,t}^\Phi = G_{s,t}^\Phi - \{s', t'\}$, in which $x_i = 0$ if and only if $\rho(i) = t'$ or $\rho(i)$ is in the ideal corresponding to x .

Intuitively, what Theorem 1 says is the following. For any Φ , the residues in fittest protein sequences are grouped into clusters, where the cluster $\rho^{-1}(s)$ is always H , the cluster $\rho^{-1}(t)$ is always P , and for each of the remaining clusters, all residues in the cluster are either all H or all P . In addition, there is a dependence given by the edges of $\hat{G}_{s,t}^\Phi$, such that if a cluster corresponding to the source of an edge is all H then the cluster at the other end is also all H .

There is no additional restriction on the structure of the space of all fittest protein sequences beyond those that follow from correspondence with the ideals of some digraph. As shown in Theorem 2, any graph may appear as $\hat{G}_{s,t}^\Phi$, with any number of residues mapped to each supernode.

Theorem 2 (characterizing a dag via Φ). *Let $\hat{G}$ be an arbitrary digraph with n nodes, labeled 1 to n , and m edges. Let $\hat{G}_0$ be the component graph of $\hat{G}$ obtained by contracting each strongly connected component of $\hat{G}$ to a single supernode through a contraction map κ . Then, there exists some Φ as defined in Assumption F1 such that for the $G_{s,t}^\Phi$ and ρ defined in Theorem 1, an isomorphism exists between $\hat{G}_{s,t}^\Phi$ and $\hat{G}_0$ mapping each $\rho(i)$ to $\kappa(i)$.*

4 Further Tools for Protein Landscape Analysis

Optimizing Multiple Objectives We can extend the results of Section 3 beyond optimizing a single fitness function.

With more than one fittest protein sequence to choose from, we may wish to find a fittest protein sequence x that is the closet to some target protein sequence $\hat{x}$ in unweighted or weighted Hamming distance. Theorem 3 shows that this optimization problem is as easy as finding an arbitrary fittest protein sequence.

We may also wish to consider what protein sequences are simultaneously the fittest for more than one fitness function. Theorem 4 shows how to compute a representation of this set similar to that provided by Theorem 1.

Theorem 3 (optimizing Hamming distances and H -residue counts over $\text{opt}(\Phi)$). *Let Φ be as defined in Assumption F1.*

1. *Given a target protein sequence $\hat{x}$, some weights w_i , and Φ as the input, we can find in $O(\Delta^2 \log \Delta)$ time an $x \in \text{opt}(\Phi)$ with the minimum weighted Hamming distance $\sum_i w_i |x_i - \hat{x}_i|$ over $\text{opt}(\Phi)$.*
2. *Given Φ as the input, we can find in $O(\Delta^2 \log \Delta)$ time an $x \in \text{opt}(\Phi)$ with the largest (or smallest) possible number of H residues over $\text{opt}(\Phi)$.*

Theorem 4 (minimizing multiple fitness functions). *Let $\Phi^1, \dots, \Phi^f$ be as defined in Assumption F3. For each ℓ , let $G_{s,t}^{\Phi^\ell}$ and ρ^ℓ be the dag and map computed from Φ^ℓ in Theorem 1. Given all $G_{s,t}^{\Phi^\ell}$ and ρ^ℓ as the input, there is an $O(\Delta)$ -time algorithm that either (a) determines that there is no protein sequence x that simultaneously minimizes Φ^1 through Φ^f , or (b) constructs a dag $G_{s,t}^{\Phi^*}$ with designated nodes s' and t' and a mapping ρ^* from $\{1, \dots, n\}$ to $V(G_{s,t}^{\Phi^*})$, such that there is a one-to-one correspondence between the protein sequences x that simultaneously minimize all $\Phi^\ell(x)$ and the ideals of $\hat{G}_{s,t}^{\Phi^*} = G_{s,t}^{\Phi^*} - \{s', t'\}$, in which $x_i = 0$ if and only if $\rho^*(i) = t'$ or $\rho^*(i)$ is in the ideal corresponding to x .*

The Space of All Fittest Protein Sequences Here we discuss some applications of the representation of the space $\text{opt}(\Phi)$ given by Theorem 1. Theorem 5 gives an algorithm to enumerate this space. Theorem 6 gives an algorithm to compute the diameter of the space in nonnegatively weighted Hamming distance. Theorem 7 gives an algorithm to determine connectivity properties of the space with respect to various classes of mutations.

Theorem 5 (enumerating all protein sequences). *Let Φ be as defined in Assumption F1. Given the $G_{s,t}^{\Phi}$ and ρ defined in Theorem 1 as the input, the protein sequences in $\text{opt}(\Phi)$ can be enumerated in $O(n)$ time per protein sequence.*

Theorem 6 (computing the diameter). *Let Φ be as defined in Assumption F1. Given the $G_{s,t}^{\Phi}$ and ρ defined in Theorem 1 as the input, it takes $O(n)$ time to compute the diameter of $\text{opt}(\Phi)$ in weighted Hamming distance where the weights w_i are all nonnegative.*

We can use $\hat{G}_{s,t}^{\Phi}$ to determine whether $\text{opt}(\Phi)$ is connected for various models of mutations. For instance, we can determine whether the space is connected for one-point mutations, in which at most one residue changes with each mutation and all intermediate protein sequences must remain the fittest. More generally, we can determine the minimum k so that the space is connected where each mutation modifies at most k residues.

We adopt a general model proposed by Kleinberg [6]. In the model, there is a system Λ of subsets of $\{1, \dots, n\}$ that is *closed downward*, i.e., if $A \subseteq B \in \Lambda$,

then $A \in \Lambda$. Two protein sequences x and y are Λ -adjacent if they are in $\text{opt}(\Phi)$ and differ exactly at the positions indexed by elements of some member of Λ . A Λ -chain is a sequence of protein sequences in $\text{opt}(\Phi)$ where each adjacent pair is Λ -adjacent. Two protein sequences x and y are Λ -connected if there exists a Λ -chain between x and y . A set of protein sequences is Λ -connected if every pair of elements of the set are Λ -connected. We would like to tell for any given Λ and Φ whether particular protein sequences are Λ -connected and whether the entire $\text{opt}(\Phi)$ is Λ -connected.

Kleinberg [6] gives polynomial-time algorithms for these problems that take Λ as input (via oracle calls) and depend only on the fact that Φ is submodular. We describe a much simpler algorithm that uses $\hat{G}_{s,t}^\Phi$ from Theorem 1. This algorithm not only determines whether two protein sequences (alternatively, all protein sequences in $\text{opt}(\Phi)$) are connected for any given Λ , but also determines the unique minimum Λ for which the desired connectivity holds. Almost all of the work is done in the computation of $\hat{G}_{s,t}^\Phi$; once we have this representation, we can read off the connectivity of $\text{opt}(\Phi)$ directly.

Theorem 7 (connectivity via mutations). *Let Φ be as defined in Assumption F1. The following problems can both be solved in $O(n)$ time.*

1. *Given the $G_{s,t}^\Phi$ and ρ defined in Theorem 1 and two protein sequences x and x' in $\text{opt}(\Phi)$ as the input, compute the maximal elements of the smallest downward-closed set system Λ such that x and x' are Λ -connected.*
2. *Given the $G_{s,t}^\Phi$ and ρ defined in Theorem 1 as the input, compute the maximal elements of the smallest downward-closed set system Λ such that $\text{opt}(\Phi)$ is Λ -connected.*

Generating Near-Fittest Protein Sequences Finding good protein sequences other than the fittest is trickier, as Lemma 1 breaks down if we are not looking at the fittest protein sequences. Here we give two algorithms that avoid this problem. Theorem 8 describes an algorithm to generate all protein sequences x in order of increasing $\Phi(x)$. Theorem 9 describes an algorithm to generate the fittest protein sequences at different unweighted Hamming distances, which is useful for examining the trade-off between fitness and distance.

Theorem 8 (enumerating all protein sequences). *Let Φ be as defined in Assumption F1. With Φ as the input, we can enumerate all protein sequences x in order of increasing $\Phi(x)$ in time $O(n\Delta^2 \log \Delta)$ per protein sequence.*

Let $\hat{x}$ be a target protein sequence. For $d \in \{0, \dots, n\}$, let $F(d)$ be the smallest $\Phi(x)$ over all protein sequences x at unweighted Hamming distance d from $\hat{x}$. A basic task of landscape analysis is to plot the graph of F . As Theorem 12(2) in Section 5 shows, this task is computationally difficult in general. Therefore, one way to plot the graph of F would be to use Theorem 8 to enumerate all protein sequences x in order of increasing $\Phi(x)$ until for each d , at least one protein sequence at distance d from $\hat{x}$ has been enumerated. This solution may require processing exponentially many protein sequences before F is fully plotted. As an

alternative, Theorem 9 gives a tool for plotting F approximately in polynomial time.

Theorem 9 (approximately plotting the energy-distance landscape). *Let Φ be as defined in Assumption F1. For each ϵ , let $\Phi_\epsilon(x) = \Phi(x) + \epsilon \cdot |x - \hat{x}|$. Let $\bar{\Phi}(\epsilon)$ be the minimum $\Phi_\epsilon(x)$ over all x .*

1. $\bar{\Phi}$ is a continuous piecewise linear concave function defined on $\mathbf{R}$ with at most $n + 1$ segments and thus at most $n + 1$ corners.
2. Let $(\epsilon_1, \bar{\Phi}(\epsilon_1)), \dots, (\epsilon_k, \bar{\Phi}(\epsilon_k))$ be the corners of $\bar{\Phi}$, where $\epsilon_1 < \dots < \epsilon_k$. Let d_i be the slope of the segment immediately to the right of ϵ_i . Let d_0 be the slope of the segment immediately to the left of ϵ_1 . Then, $n = d_0 > d_1 > \dots > d_k = 0$.
3. Given Φ and $\hat{x}$ as the input, we can compute $(\epsilon_1, \bar{\Phi}(\epsilon_1)), \dots, (\epsilon_k, \bar{\Phi}(\epsilon_k))$ and $d_0, \dots, d_k$ in $O(n\Delta^2 \log \Delta)$ time.

Tuning the Parameters of the GC Model Here we show how to systematically tune the parameters α and β so that a fittest protein sequence for a given 3D structure matches the 3D structure's native protein sequence as closely as possible in terms of unweighted or weighted Hamming distance. For this purpose, we assume $s_i \geq 0$. Furthermore, since the fitness function does not have an absolute scale, we may fix α at -1 and vary β .

Theorem 10 (tuning α and β). *Let Φ be as defined in Assumption F2. Given a target protein sequence $\hat{x}$ and Φ as the input, we can find in $O(n\Delta^2 \log \Delta)$ time the set of all β where the closest unweighted (or weighted) Hamming distance between $\hat{x}$ and any protein sequence in $\text{opt}(\Phi_\beta)$ is the minimum over all β .*

5 Computational Hardness Results

Theorem 11 (hardness of counting and averaging). *Let Φ be as defined in Assumption F1. The following problems are all #P-complete:*

1. Given Φ as the input, compute the cardinality of $\text{opt}(\Phi)$.
2. Given $\Phi^1, \dots, \Phi^f$ as the input, where f is any fixed positive integer and $\Phi^1, \dots, \Phi^f$ are as defined in Assumption F3, compute the number of protein sequences x that simultaneously minimize $\Phi^\ell(x)$ for all $\ell = 1, \dots, f$.
3. Given Φ as the input, compute the average norm $|x|$, i.e., the average number of H residues in x , over all $x \in \text{opt}(\Phi)$.
4. Given Φ and a target protein sequence $\hat{x}$ as the input, compute the average unweighted Hamming distance $|x - \hat{x}|$ over all $x \in \text{opt}(\Phi)$.
5. Given Φ , a target protein sequence $\hat{x}$, and an integer d as the input, compute the number of protein sequences in $\text{opt}(\Phi)$ at unweighted Hamming distance d from $\hat{x}$.

Theorem 12 (hardness of plotting the energy-distance landscape). *Let Φ be as defined in Assumption F1.*

1. Given Φ and two integers d_1, d_2 as the input, it is NP-complete to determine whether there is an Φ -minimizing x with $d_1 \leq |x| \leq d_2$.
2. Let $\hat{x}$ be a target protein sequence. For $d \in \{0, \dots, n\}$, let $F(d)$ be the smallest $\Phi(x)$ over all protein sequences x at unweighted Hamming distance d from $\hat{x}$. Given Φ and d as the input, it is NP-hard to compute $F(d)$.

6 Applications to Empirical Protein 3D Structures

To demonstrate our algorithms, we chose 34 proteins with known 3D structures from the Protein Data Bank (PDB) at <http://www.rcsb.org/pdb>. These 3D structures included 8 from Kleinberg's study [6] but excluded the protein fragments and multimeric proteins used in that study. The chosen 3D structures were then represented by centroids for each side chain calculated from the coordinates of each atom in the side chain; in the case of 3D structures solved by NMR, hydrogen atoms were included into centroid calculations. For glycine, the centroid was taken to be the position of C_α . For each side chain, the area of solvent accessible surface was computed via the Web interface of the ASC program with default parameters [4]. In accordance to the GC model, each of the chosen native protein sequences was converted into a binary H/P sequence following Sun *et al.* [9], where A, C, F, I, L, M, V, W, Y are H , and the other amino acids are P .

The detailed results of this small-scale empirical study can be found in the full version of this paper, which is deposited in the Computing Research Repository as <http://www.arxiv.org/abs/cs.CE/0101015>.

As anticipated, our algorithms computed fittest protein sequences that are closer to native protein sequences than found by Kleinberg [6]. We further conjectured a significant relationship between a computed fittest protein sequence's similarity to a native protein sequence and the diversity of the native protein in nature. Such a relationship would be highly intriguing biologically. We examined this conjecture by assessing the diversity of native proteins using the database PFAM at <http://pfam.wustl.edu>, which is a database of protein families determined through Hidden Markov Models [3]. Our study confirmed this conjecture.

We are currently planning a large-scale analysis of further empirical protein 3D structures; the results will be reported in a subsequent paper.

References

1. J. Atkins and W. E. Hart. On the intractability of protein folding with a finite alphabet of amino acids. *Algorithmica*, 25(2-3):279–294, 1999. 404
2. J. Banavar, M. Cieplak, A. Maritan, G. Nadig, F. Seno, and S. Vishveshwara. Structure-based design of model proteins. *Proteins: Structure, Function, and Genetics*, 31:10–20, 1998. 404
3. A. Bateman, E. Birney, R. Durbin, S. R. Eddy, K. L. Howe, and E. L. L. Sonnhammer. PFAM—A database of protein domain family alignments and HMMs. *Nucleic Acids Research*, 28:263–266, 2000. 414

4. F. Eisenhaber, P. Lijnzaad, P. Argos, C. Sander, and M. Scharf. The double cube lattice method: Efficient approaches to numerical integration of surface area and volume and to dot surface contouring of molecular assemblies. *Journal of Computational Chemistry*, 16(N3):273–284, 1995. 407, 414
5. W. E. Hart. On the computational complexity of sequence design problems. In *RECOMB*, pages 128–136, 1997. 406
6. J. M. Kleinberg. Efficient algorithms for protein sequence design and the analysis of certain evolutionary fitness landscapes. In *RECOMB*, pages 226–237, 1999. 404, 406, 408, 409, 411, 412, 414
7. J.-C. Picard and M. Queyranne. On the structure of all minimum cuts in a network and applications. *Mathematical Programming Study*, (13):8–16, 1980. 404, 408, 409
8. C. Reidys, P. Stadler, and P. Schuster. Generic properties of combinatory maps: Neutral networks of RNA secondary structures. *Bulletin of Mathematical Biology*, 59:339–397, 1997. 405
9. S. J. Sun, R. Brem, H. S. Chan, and K. A. Dill. Designing amino acid sequences to fold with good hydrophobic cores. *Protein Engineering*, 8(12):1205–1213, Dec. 1995. 404, 406, 414

Complexity of Comparing Hidden Markov Models

Rune B. Lyngsø^{1*} and Christian N. S. Pedersen^{2**}

¹ Baskin Center for Computer Science and Engineering, University of California
Santa Cruz, CA 95064, USA

`rlyngsø@cse.ucsc.edu`

² BiRC – Bioinformatics Research Center ^{***}, Department of Computer Science,
University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark

`cstorm@daimi.au.dk`

`www.birc.dk`

Abstract. The basic theory of hidden Markov models was developed and applied to problems in speech recognition in the late 1960's, and has since then been applied to numerous problems, e.g. biological sequence analysis. In this paper we consider the problem of computing the most likely string generated by a given model, and its implications on the complexity of comparing hidden Markov models. We show that computing the most likely string, and approximating its probability within any constant factor, is **NP**-hard, and establish the **NP**-hardness of comparing two hidden Markov models under the L_∞ - and L_1 -norms. We discuss the applicability of the technique used to other measures of distance between probability distributions. In particular we show that it cannot be used to prove **NP**-hardness of determining the Kullback-Leibler distance between the probability distributions of two hidden Markov models, or of comparing them under the L_k -norm for any fixed even integer k .

Keywords Hidden Markov Models, Consensus String, Distance Measures, NP Hardness

1 Introduction

A hidden Markov model (HMM) is a description of a probability distribution over a set of strings. It is convenient to consider a HMM as a generative model in which a run generates a string with a certain probability. A run starts in a special start-state, and continues by following a first order Markov chain of states, called the path, until a special end-state is reached. A symbol from a finite alphabet is emitted according to some probability distribution each time a non-silent state is

* Supported by grants from Carlsbergfondet and the Prog. in Math. and Mol. Biology

** Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT)

*** funded by the University of Aarhus Research Foundation

entered. The theory of HMMs was developed and applied to problems in speech recognition in the late 1960's and early 1970's. Rabiner [14] gives a good overview of the theory of HMMs and its applications to problems in speech recognition. Hidden Markov models are also applied in other areas than speech recognition. One prominent example is computational biology where they have found many applications, e.g. modeling of DNA sequences [5], protein secondary structure prediction [2], gene finding [11], recognition of transmembrane proteins [16], and characterization of biological sequence families [12].

Applications of HMMs are often based on two fundamental questions. Given an HMM and a string we might want to determine the probability of the string under the model, i.e. the probability that the model has generated the string. This can be used for *classification* of the string as either belonging to the family of strings represented by the model or not. Or we might want to determine the most likely path of states through the model that generates the string. This can be used for *annotating* the string with states from the model. Dynamic programming algorithms solving these problems are described in e.g. [14].

In this paper we consider the problem of determining the most likely string generated by a given HMM, i.e. of determining the *consensus* string of the model, and its implications on the problem of comparing HMMs. We show that in polynomial time we cannot for any $\epsilon > 0$ approximate the probability of the most likely string under an HMM with n states within a factor of $n^{1/4-\epsilon}$ unless $\mathbf{P} = \mathbf{NP}$, and within a factor $n^{1/2-\epsilon}$ unless $\mathbf{ZPP} = \mathbf{NP}$. The hardness results hold even if we restrict the HMM to a model without silent states generating only strings over a binary alphabet. The problem of determining the consensus string of an HMM has not been addressed previously in the literature. However, it is useful for studying the hardness of comparing the probability distributions given by two HMMs. Comparing two HMMs is an interesting theoretical problem with practical applications as well, for example by comparing two profile HMMs, e.g. from the Pfam protein families database [3], we compare entire sequence families instead of just individual members. In [13] we present methods for comparing HMMs, and describe how to compute the Euclidean distance (the L_2 -distance) between two models in polynomial time.

Using the hardness of determining the consensus string, we show that comparing two HMMs under the L_∞ -norm is hard. Furthermore, we link the consensus string probability for models constructed in the consensus string hardness proof to the L_k -norm between a pair of models for any $k \in \mathbf{R}_+$. We utilize this link to prove the hardness of comparing two HMMs under the L_1 -norm but show that it cannot be used to establish the hardness of comparing two HMMs under the L_{2k} -norm for any $k \in \mathbf{N}$. The L_1 -distance is of special interest as it equals twice the *variation distance*, i.e. the maximum numerical difference between the probability of any set of events under the two distributions, see e.g. [6]. Comparing probability distributions by L_k -distances is a well-studied problem, see e.g. [8,9,4] for algorithms for comparing probability distributions over finite sets.

The rest of the paper is organized as follows. In Sect. 2 we discuss HMMs in more detail. In Sect. 3 we show that computing the most likely string, and

approximating its probability within any constant factor, is **NP**-hard. In Sect. 4 we consider the general problem of comparing HMMs, and show that comparison under the L_∞ - and L_1 -norms is **NP**-hard. In Sect. 5 we summarize the status of the tractability of comparing HMMs by various well-known distances.

2 Hidden Markov Models

Let M be an HMM that generates strings over some finite alphabet Σ with probability distribution P_M , i.e. $P_M(s)$ denotes the probability of $s \in \Sigma^*$ under model M . Like a classical Markov model, an HMM consists of a set of interconnected states. We use $a_{q,q'}^M$ to denote the probability of a transition from state q to state q' in model M . These probabilities are called *state transition probabilities*. The transition structure of an HMM is often shown as a directed graph with a node for each state, and an edge between two nodes if the corresponding state transition probability is non-zero. Unlike a classical Markov model, a state in an HMM can emit a symbol according to a local probability distribution over all possible symbols. We use $e_{q,\sigma}^M$ to denote the probability of emitting symbol $\sigma \in \Sigma$ in state q in model M . These probabilities are called *symbol emission probabilities*. A state without symbol emission probabilities is called a *silent* state.

It is convenient to consider an HMM as a generative model in which a *run* generates a string. A run of an HMM begins in a special start-state and continues from state to state according to the state transition probabilities until a special end-state is reached. Each time a non-silent state is entered, a symbol is emitted according to the symbol emission probabilities of the state. We refer to the Markovian sequence of states in a run as the *path* followed by the run. The string generated by a run is the concatenation of the symbols emitted along its path. The name “*hidden Markov model*” comes from the fact that the Markovian sequence of states followed by a run, the path, is hidden while only the emitted symbols, the generated string, is observable.

The probability $P_M(\pi)$ of following a path $\pi = (\pi_0, \pi_1, \dots, \pi_k)$ in model M is given by the state transition probabilities as

$$P_M(\pi) = \prod_{i=1}^k a_{\pi_{i-1}, \pi_i}^M. \quad (1)$$

The probability $P_M(\pi, s)$ of following a path $\pi = (\pi_0, \pi_1, \dots, \pi_k)$ in model M and emitting string s depends on the subsequence $(\pi_{i_1}, \pi_{i_2}, \dots, \pi_{i_l})$ of non-silent states on the path π . If the length of string $s = s_1 s_2 \dots s_l$ is different from the number of non-silent states along path π , the probability $P_M(\pi, s)$ is zero. Otherwise, the probability of following path π and emitting string s is

$$P_M(\pi, s) = P_M(\pi) \cdot P_M(s \mid \pi) = \prod_{i=1}^k a_{\pi_{i-1}, \pi_i}^M \cdot \prod_{j=1}^l e_{\pi_{i_j}, s_j}^M. \quad (2)$$

Since a run r of an HMM M is identified by a path π_r through the model and an emitted string s_r , we can define the probability of a run as $P_M(r) = P_M(\pi_r, s_r)$. Finally, the probability $P_M(s)$ of model M generating a string s is the probability of following any path and emitting string s , that is

$$P_M(s) = \sum_{\pi} P_M(\pi, s). \quad (3)$$

3 Finding the Most Likely String

In this section we will establish the hardness of finding the most likely string of an HMM, a question one might naturally ask about HMMs. We show that computing the probability of the most likely string generated by an HMM is **NP**-hard, but first we observe why there has to exist a most likely string: Let M be an HMM in which a run r emits a string s , i.e. $P_M(s) = \delta$ for some $\delta > 0$. Since $\sum_{s \in \Sigma^*} P_M(s) = 1$, there can be at most $1/\delta$ strings s' where $P_M(s') \geq \delta$. This implies that there cannot exist an infinite series of strings with increasing probabilities all greater than δ . Hence, there has to exist a most likely string.

We will show the hardness of computing the probability of the most likely string by a reduction from MAXCLIQUE, the problem of computing the size of the maximum clique in an undirected graph. The proposed reduction essentially preserves approximations. Hence, the approximation hardness results of [10,7] for MAXCLIQUE can be translated into approximation hardness results for computing the probability of the most likely string. Since the probability of a given string can be computed in polynomial time, cf. [14], the result also implies that finding the most likely string, and not only its probability, is **NP**-hard. We start with a lemma describing how to construct an HMM that by its probability distribution over finite strings captures the clique sizes of a graph.

Lemma 1. *For any graph $G = (V, E)$ we can in polynomial time construct an HMM M_G generating strings over the alphabet $\Sigma = V$ such that for all integers $k \geq 1$ it holds that $\exists s \in \Sigma^* : P_{M_G}(s) = k/\gamma_G$ if and only if G has a clique of size k , where $\gamma_G = \sum_{v \in V} 2^{\deg(v)}$ and $\deg(v)$ is the degree of v in G .*

Proof. For simplicity, we will assume that $V = \{1, 2, \dots, |V|\}$. The basic idea of the construction of M_G , illustrated in Fig. 1, is for each node v in G to construct a submodel with $2^{\deg(v)}$ possible paths of equal probability. Each of these paths generate one of the $2^{\deg(v)}$ ordered sequences of nodes where each node occurs at most once, v occurs exactly once, and all other occurring nodes are connected by an edge to v . We can construct these submodels simply by having a state emitting each of the possible symbols that can occur in the sequence with probability $1/2$, and then choose to either enter or skip these states in turn with equal probability $1/2$. By choosing the submodel of v with probability $2^{\deg(v)}/\gamma_G$ in the aggregate model consisting of all the submodels, all paths in the aggregate model will have probability $1/\gamma_G$. Thus, the probability of a string will be k/γ_G where k is the number of submodels that can generate it. Hence, the probability of a string “counts” the number of submodels that can generate it.

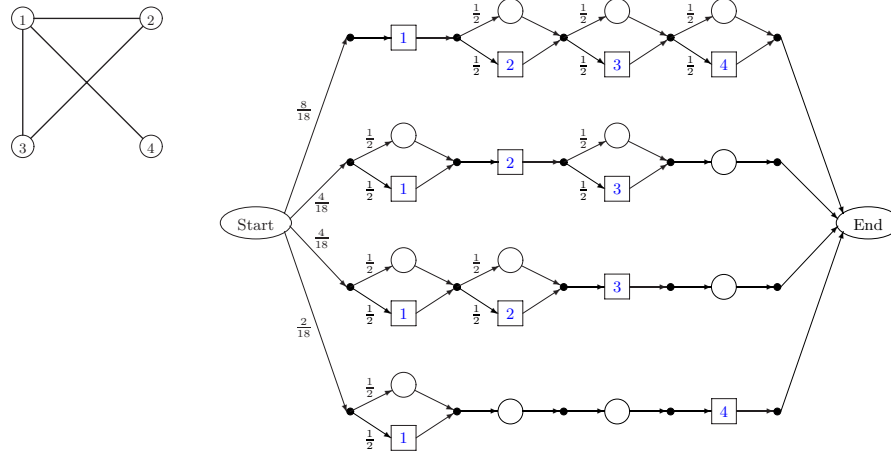


Fig. 1. A graph, $G = (\{1, 2, 3, 4\}, \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}\})$, and the HMM M_G constructed cf. Lemma 1. The square states are the non-silent states $n_{u,v}$ with the symbol they emit with probability 1 written inside. The large, hollow circular states are the silent states $s_{u,v}$ while the small, black circular states are the silent states $i_{u,v}$.

Formally, the model M_G consists of a start-state $start$, an end-state end , a set of silent states S (not essential for the construction but serves to make the structure of the model clearer), and a set of non-silent states N , where

$$S = \{s_{u,v} \mid u, v \in V, u \neq v\} \cup \{i_{u,v} \mid u \in V, v \in V \cup \{0\}\},$$

$$N = \{n_{u,v} \mid u, v \in V, \{u, v\} \in E \vee u = v\}.$$

The non-zero transition probabilities are

$$a_{start, i_{u,0}} = 2^{\deg(u)} / \gamma_G$$

$$a_{i_{u,v-1}, n_{u,v}} = \begin{cases} 1 & \text{if } u = v \\ 1/2 & \text{if } \{u, v\} \in E \\ \text{undefined} & \text{otherwise } (n_{u,v} \text{ is not a state in } M_G) \end{cases}$$

$$a_{i_{u,v-1}, s_{u,v}} = \begin{cases} \text{undefined} & \text{if } u = v \text{ } (s_{u,v} \text{ is not a state in } M_G) \\ 1/2 & \text{if } \{u, v\} \in E \\ 1 & \text{otherwise} \end{cases}$$

where $u, v \in V$ and $a_{n_{u,v}, i_{u,v}}, a_{s_{u,v}, i_{u,v}}, a_{i_{u,|V|}, end} = 1$ whenever they are well defined, i.e. whenever both states are in M_G . The non-zero emission probabilities are $e_{n_{u,v}, v} = 1$ for all non-silent states $n_{u,v} \in N$. Note that though each state can

emit only one particular symbol, numerous states can emit identical symbols. Thus the constructed model cannot be described as just a Markov model. The model can evidently be constructed in polynomial time.

We still need to prove the connection between clique sizes in G and string probabilities in M_G . We first observe that any run through M_G has probability $1/\gamma_G$, hence the probability of any string must be k/γ_G for some k . Now assume that there is a string s with $P_{M_G}(s) = k/\gamma_G$. Hence, s can be generated by the submodels of k nodes. Let $\{u_i\}_{1 \leq i \leq k}$ be the set of nodes whose submodels can generate s . We claim that $\{u_i\}_{1 \leq i \leq k}$ must be a clique in G . First, all u_i for $1 \leq i \leq k$ must occur in s , as all sequences generated by the submodel of node u_i contains u_i . Secondly, each u_i must be connected by an edge to all nodes, apart from itself, occurring in s . Hence, all pairs of nodes $u, v \in \{u_i\}_{1 \leq i \leq k}$ are connected by an edge.

Conversely, assume that $C = \{u_i\}_{1 \leq i \leq k}$, where $i < j \Leftrightarrow u_i < u_j$, is a clique in G . We claim that $P_{M_G}(u_1 u_2 \dots u_k) = k/\gamma_G$. First, $u_1 u_2 \dots u_k$ can be generated by the submodel of any node $u_i \in C$ as it contains u_i and as u_i is connected by an edge to any other node occurring in $u_1 u_2 \dots u_k$. Secondly, $u_1 u_2 \dots u_k$ cannot be generated by the submodel of any node $v \notin C$ as it does not contain v . Hence, $u_1 u_2 \dots u_k$ can be generated by precisely k submodels and thus has probability k/γ_G in M_G . $\square$

The model M_G constructed in Lemma 1 satisfies that the end-state can be reached from any other state with non-zero probability, i.e. there does not exist a state p where $a_{p,p}^{M_G} = 1$. The next two lemmata simplify the model. More precisely, we show that only non-silent states and a binary alphabet is necessary.

Lemma 2. *Lemma 1 still holds if we restrict the alphabet to be binary.*

Proof. In the proof of Lemma 1 we used an alphabet $\Sigma = V$. We can encode this alphabet in binary such that there to each $\sigma \in \Sigma$ corresponds a unique string in $\{0, 1\}^{\lceil \log |V| \rceil}$. Each non-silent $n_{u,v}$ state in M_G is now replaced with a sequence of $\lceil \log |V| \rceil$ non-silent states, where the i 'th state emits the i 'th bit in the binary encoding of v and has probability 1 for the transition to the $i + 1$ 'st state. $\square$

Lemma 3. *Let M be an HMM where the end-state can be reached from any other state with non-zero probability. We can construct an HMM M' with no silent states and $P_M = P_{M'}$.*

Proof. We prove the lemma by describing the simple procedure of removing one silent state, thus transforming M into a model M'' with one less silent state. This procedure can then be applied to all the silent states of M in turn. Let p be a silent state, i.e. it does not emit any symbols. Hence, the only effect p has is to allow going from one state q via p to another state r without emitting any symbols on the way. But this might as well be done with a direct transition. When eliminating p we thus have to update all other transition probabilities as

$$a_{q,r}^{M''} = a_{q,r}^M + a_{q,p}^M \cdot a_{p,r}^M / (1 - a_{p,p}^M)$$

if $a_{p,p}^M < 1$. Since the end-state by assumption can be reached from any state with non-zero probability, we can ignore the case $a_{p,p}^M = 1$. Hence, the described update yields a new model M'' with $P_{M''} = P_M$ but one less silent state. $\square$

Corollary 1. *Lemma 2 still holds with models not having any silent states.*

We are now ready to present the main result of this section, that the probability of the most likely string of an HMM is hard to approximate.

Proposition 1. *Let M be an HMM with n states generating strings over an alphabet Σ , where $|\Sigma| \geq 2$. For any $\epsilon > 0$ we cannot in polynomial time*

- *approximate the probability of the most likely string under M within a factor of $n^{1/4-\epsilon}$ unless $\mathbf{P} = \mathbf{NP}$*
- *approximate the probability of the most likely string under M within a factor of $n^{1/2-\epsilon}$ unless $\mathbf{ZPP} = \mathbf{NP}$*

Proof. In [10] it is proved that we in polynomial time cannot approximate the largest clique of a graph $G = (V, E)$ within a factor of $|V|^{1/2-\epsilon}$ unless $\mathbf{P} = \mathbf{NP}$ and cannot approximate it within $|V|^{1-\epsilon}$ unless $\mathbf{ZPP} = \mathbf{NP}$. By Lemma 3 and an inspection of the proofs of Lemmata 1 and 3 we can construct a model M_G with less than $|E| \lceil \log |V| \rceil$ states such that the most likely string in M_G has probability k/γ_G if and only if the largest clique of G is of size k . Assume we can approximate $\max\{P_{M_G}(s) \mid s \in \Sigma^*\}$ within a factor of n^c in polynomial time, i.e. that we can find $p \leq k/\gamma_G$ such that $p \cdot n^c \geq \max\{P_{M_G}(s) \mid s \in \Sigma^*\} = k/\gamma_G$. As $|E| \cdot \lceil \log |V| \rceil > n$ it follows that $p \cdot \gamma_G \cdot (|E| \cdot \lceil \log |V| \rceil)^c \geq k$. Furthermore, $|E| \cdot \lceil \log |V| \rceil = o(|V|^{2+\delta})$ for any $\delta > 0$. Hence, we can approximate the size of the largest clique in G within a factor $|V|^{2c+2\delta}$. The result now follows by choosing $c = \frac{1}{2}(1 - \epsilon - 2\delta)$ and $c = \frac{1}{2}(\frac{1}{2} - \epsilon - 2\delta)$, respectively. $\square$

4 Comparing Hidden Markov Models

What the most probable string of an HMM is, is a very natural question to ask. However, it does not seem to be of high practical importance. At least, it does not appear that any previous work has been concerned with this problem. Indeed, our main motivation for studying the problem of finding the most probable string was that we can use the hardness of this problem – and some specific details from the reduction proving the hardness – as basis for proving the results of this section, developing hardness results for comparing the probability distributions of two HMMs under L_k -norms.

Usually, HMMs are considered tools for analyzing data. But we may also view them as a compact representation of a probability distribution over finite sequences. E.g. in computational biology, an HMM for classifying sequences can be viewed as a representation of the family of sequences belonging to the class. Hence, the HMM itself can be considered data. Since comparing data is a common task in computational biology, it is interesting, both from a theoretical and

a practical viewpoint, to investigate how to compare two HMMs, i.e. how to compare the probability distributions described by the two models.

The L_k -norm between two models M and M' over the same alphabet Σ is $\|P_M - P_{M'}\|_k = \sum_{s \in \Sigma^*} \sqrt[k]{|P_M(s) - P_{M'}(s)|^k}$, and the L_∞ -norm is $\|P_M - P_{M'}\|_\infty = \max_{s \in \Sigma^*} |P_M(s) - P_{M'}(s)|$. That L_k -norms are well-defined for HMMs, even over infinite countable sets, follows from the comparison criteria for series. For the L_∞ -norm the well-definedness can be argued similar to the well-definedness of a most likely string. In [13] we describe how to compute the L_2 -distance between two models in polynomial time. In this section we will extend this result by proving that if $\mathbf{P} \neq \mathbf{NP}$ then neither the L_∞ -norm nor the L_1 -norm can be computed in polynomial time, but that the L_k -norm can be computed in polynomial time for k any fixed even integer. We conjecture these to be the only efficiently computable L_k -norms, i.e. the L_k -norm between the probability distributions of two HMMs can be computed in polynomial time if and only if k is fixed and an even integer. We start with the L_∞ -norm. This is closely linked to the probability of the most likely string by the following lemma.

Lemma 4. *Let M be an HMM that generates finite strings over an alphabet Σ . We can construct another HMM M' that generates finite strings over an alphabet $\Sigma \cup \{\$$ such that*

$$\max\{P_M(s) \mid s \in \Sigma^*\} = \|P_M - P_{M'}\|_\infty.$$

Proof. The model M' we construct will be an almost exact copy of M . The only difference is an extra state that emits a special symbol $\$ \notin \Sigma$ just prior to entering the end-state. Hence, we add a new state q to M' , as compared to M , with $e_{q,\sigma}^{M'} = \delta(\sigma, \$)$ and $a_{q,p}^{M'} = \delta(p, \text{end})$. The transition probabilities are updated to go to the new state q instead of to the end-state, i.e. $a_{p,q}^{M'} = a_{p,\text{end}}^M$ and $a_{p,\text{end}}^{M'} = 0$ for all $p \neq q, \text{end}$; otherwise emission as well as transition probabilities are the same for M and M' . With this construction the sets of sequences emitted by M and M' are disjoint and $P_M(s) = P_{M'}(s\$)$ for all s . Hence,

$$\begin{aligned} \|P_M - P_{M'}\|_\infty &= \max\{|P_M(s) - P_{M'}(s)| \mid s \in (\Sigma \cup \{\$\})^*\} \\ &= \max\{P_M(s) \mid s \in \Sigma^*\}. \end{aligned}$$

□

Corollary 2. *The hardness results of Proposition 1 for determining the most likely string of an HMM transfers directly to the problem of comparing the probability distributions of two HMMs M and M' under the L_∞ -norm.*

In general, the L_∞ -norm between two probability distributions, P and Q , defined on the same set, Ω , is defined as $\|P - Q\|_\infty = \max_{s \in \Omega} |P(s) - Q(s)|$. Hence, it measures the largest difference in probabilities we can obtain for any possible single observation. Another, seemingly similar, way to compare two probability distributions P and Q is by the *variation distance*, see e.g. [6], defined as $\|P - Q\| = \max_{\mathcal{A} \subseteq \Omega} |P(\mathcal{A}) - Q(\mathcal{A})|$. This measures the largest difference

in probabilities we can obtain for any subset of observations. It is well-known that the variation distance between two probability distributions is half the L_1 -distance between the two probability distributions, i.e. $\|P - Q\| = \frac{1}{2} \cdot \|P - Q\|_1$.

Proposition 2. *Comparing two HMMs under the L_1 -norm is **NP**-hard.*

Proof. The proof is again by a reduction from MAXCLIQUE. Or rather a reduction from the consensus string problem for the HMM M_G constructed from a graph $G = (V, E)$ in the previous section to establish the hardness of the consensus string problem. Recall that every string generated by M_G is a subsequence of $12 \dots |V|$ and have probability i/γ_G for some $i \in \{0, 1, \dots, |V|\}$, where $\gamma_G = \sum_{v \in V} 2^{\deg(v)}$. Let a_i denote the number of subsequences of $12 \dots |V|$ which the HMM M_G generates with probability i/γ_G . If we know the maximum k such that $a_k \neq 0$, we can conclude that the probability of the most likely string under M_G is k/γ_G , and by the result of the previous section, that the maximum clique in G has size k .

To clarify our proof technique, we initially ignore the fact that the probabilities of all sequences generated by a model has to sum to 1. For any $x \in \mathbf{R}$ we can construct a model $M_{|V|}^x$ that assigns a uniform probability of x to all subsequences of $12 \dots |V|$ and probability 0 to all other sequences. Comparing $M_{|V|}^{i/\gamma_G}$ to M_G under the L_1 -norm we get

$$\|P_{M_{|V|}^{i/\gamma_G}} - P_{M_G}\|_1 = \sum_s \left| P_{M_{|V|}^{i/\gamma_G}}(s) - P_{M_G}(s) \right| = \sum_{j=0}^{|V|} a_j \frac{|i-j|}{\gamma_G}. \quad (4)$$

Hence, comparing M_G with $M_{|V|}^{i/\gamma_G}$ under the L_1 -norm for all $i = 0, 1, \dots, |V|$ we obtain a system of linear equations

$$Ma = l \quad (5)$$

for determining the a_i , where M is the $(|V| + 1) \times (|V| + 1)$ matrix with entries $M_{ij} = |i - j|$, and l is the $(|V| + 1) \times 1$ vector with entries $l_i = \gamma_G \|P_{M_{|V|}^{i/\gamma_G}} - P_{M_G}\|_1$. The matrix M is invertible with inverse

$$(M^{-1})_{ij} = \begin{cases} (1 - |V|)/2|V| & \text{if } i = j = 1 \text{ or } i = j = |V| + 1. \\ -1 & \text{if } 1 < i = j \leq |V|. \\ 1/2 & \text{if } j = i \pm 1. \\ 1/2|V| & \text{if } i = 1, j = |V| + 1 \text{ or } i = |V| + 1, j = 1. \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

Thus, knowing l we can compute a in time polynomial in the size of G , which in turn allows us to determine the size of the largest clique in G .

Let us now extend the above technique when keeping in mind that the probabilities of all sequences generated by a model have to sum to 1. When the

probabilities of all sequences generated by a model are required to sum to 1, the only model $M_{|V|}$ with a uniform probability distribution over all subsequences of $1\ 2\ \dots\ |V|$ and all other strings having probability 0 is $M_{|V|} = M_{|V|}^{2^{-|V|}}$.

However, if we embed a model M as a submodel of an aggregate model M' that chooses model M with probability x and chooses another submodel that always generates a string with a single symbol $\$ \notin \Sigma$ with probability $1 - x$, we have scaled down the probabilities of all sequences originally generated by M by a factor x . Hence, for any $i = 0, 1, \dots, |V|$ we can make $M_{|V|}$ generate all subsequences of $0\ 1\ \dots\ |V|$ with probability equal to the probability of a string in M_G that can be generated by exactly i different paths in M_G . Either by rescaling $M_{|V|}$ by a factor $\frac{i/\gamma_G}{2^{-|V|}}$ if $i/\gamma_G \leq 2^{-|V|}$, or by rescaling M_G by a factor $\frac{2^{-|V|}}{i/\gamma_G}$ if $i/\gamma_G > 2^{-|V|}$. Denoting the rescaled models $M_{G,i}$ and $M_{|V|,i}$ we get

$$\begin{aligned} \|P_{M_{|V|,i}} - P_{M_{G,i}}\|_1 &= |P_{M_{|V|,i}}(\$) - P_{M_{G,i}}(\$)| + \sum_{s \neq \$} |P_{M_{|V|,i}}(s) - P_{M_{G,i}}(s)| \\ &= (1 - b_i) + c_i \sum_{j=0}^{|V|} a_j |i - j| \end{aligned} \quad (7)$$

where $b_i = \min \left\{ \frac{i/\gamma_G}{2^{-|V|}}, \frac{2^{-|V|}}{i/\gamma_G} \right\}$ and $c_i = \min \left\{ \frac{1}{\gamma_G}, \frac{2^{-|V|}}{i} \right\}$. By comparing $M_{G,i}$ and $M_{|V|,i}$ under the L_1 -norm for $i = 0, 1, \dots, |V|$ we thus obtain a system of linear equations identical to (5), except for the vector l . In the system of linear equations based on comparing rescaled models, one observes from (7) that the entries of l are $l_i = \frac{1}{c_i} (\|P_{M_{|V|,i}} - P_{M_{G,i}}\|_1 + b_i - 1)$. But the system being solvable does not depend on the value of l , only on the matrix M . Hence, if we can compare two HMMs under the L_1 -norm in polynomial time, we can determine a and thus the size of the largest clique in polynomial time. $\square$

In the proof of Proposition 2 we do not directly relate the L_1 -distance between two models to the most likely string of one (or both) of the models. Hence, the approximation hardness is not preserved by the reduction, and the hardness result obtained is only for the computation of the exact L_1 -distance. In the rest of the paper we examine the proof technique of Proposition 2 in more details.

Let P and Q be two probability distributions defined on a set Ω , and D be a distance between probability distributions defined as a sum of point-wise comparisons, i.e. $D(P, Q) = \sum_{s \in \Omega} d(P(s), Q(s))$. One can consider using the same technique as in the proof of Proposition 2 to prove it hard to compare two HMMs under the measure D . We can set up a system of linear equations similar to (5), as the distance between $M_{|V|,i}$ and $M_{G,i}$ under D is

$$D(P_{M_{|V|,i}}, P_{M_{G,i}}) = (1 - b_i) + \sum_{j=0}^{|V|} a_j \cdot d(c_i \cdot i, c_i \cdot j). \quad (8)$$

Whether this allows us to deduce anything about the hardness of comparing two HMMs under D depends on whether the resulting system of linear equation

system, with matrix M defined by $M_{ij} = d(c_i \cdot i, c_i \cdot j)$, can be solved with sufficient precision to determine the maximum clique size of G in time polynomial in the size of G .

As an example we can consider the L_k -norm for an even integer k . In general, for the L_r -norm we have

$$\begin{aligned} (\|P_{M_{|V|,i}}, P_{M_{G,i}}\|_r)^r &= (1 - b_i) + \sum_{j=0}^{|V|} a_j |c_i \cdot i - c_i \cdot j|^r \\ &= (1 - b_i) + c_i^r \sum_{j=0}^{|V|} a_j |i - j|^r. \end{aligned} \quad (9)$$

Hence, when considering the L_k -norm for an even integer k , the matrix M of the resulting system of linear equations has entries $M_{ij} = |i - j|^k = (i - j)^k$. Since any $m \times m$ matrix A defined by $A_{ij} = (x_i - y_j)^k$ is singular if $m > k + 1$ (a fact which can be proved e.g. using the fact that a polynomial of degree k is uniquely determined by its value in $k + 1$ points, hence we can choose a set of values in $m > k + 1$ points that does not agree with any polynomial of degree k), the system of linear equations defined by $M = (i - j)^k$, for an even integer k , cannot be solved for a unique solution if $|V| > k$, and we fail to prove the **NP**-hardness of comparing two HMMs under the L_k -norm for an even integer k . This failure is no surprise because the algorithmic technique for comparing two HMMs under the L_2 -norm described in [13] can easily be extended to compare two HMMs under the L_k -norm, k a fixed even integer, in time $O(n^{3k})$ where n is the number of states in the two models. The algorithmic technique of [13] cannot be applied to the computation of any other L_r -norm than those where r is an even integer.

Based on the **NP**-hardness of comparing two models under the L_1 -norm proved above, and that the L_r -norms where r is an even integer stands out as the only ones where the absolute value operation can be ignored, we conjecture that it is **NP**-hard to compare two models under any L_r -norm where r is not an even integer. This claim immediately follows, if we for a fixed r not an even integer can solve a linear equation system as in (5) with an $m \times m$ matrix M with entries $M_{ij} = |i - j|^r$ in time polynomial in m .

Similarly to the L_k -norm for k an even integer, we can rule out the Kullback-Leibler divergence, or relative entropy, as a case where the technique used in the proof of Proposition 2 is useful. Here the matrix M of the linear equation system can be defined by $M_{ij} = i \log \frac{i}{j} = i(\log i - \log j)$. Setting $x_i = \log i$ and $y_j = \log j$, by the above discussion the $m \times m$ matrix M' defined by $M'_{ij} = \log i - \log j$ is singular if $m > 2$. Hence, so is any matrix obtained by multiplying one or more rows of M' by scalars.

5 Conclusion

When choosing how to compare two probability distributions, one important consideration is how efficiently a given distance can be computed. In [1] a num-

ber of commonly used distance measures between probability distributions are listed: the χ^2 , variation, quadratic, Hellinger and Kullback-Leibler distances. In [13] we show how to compute the quadratic distance between the probability distributions of two HMMs in polynomial time. The Hellinger distance, i.e. the distance in Euclidean space between two probability distributions after they have been normalized to 1, can also be computed in polynomial time cf. [13], where we show how to compute the angle in Euclidean space between the two probability distributions when interpreted as infinite dimensional vectors.

In this paper we have proved that the variation distance is **NP**-hard to compute, and furthermore that the L_∞ -distance is hard even to approximate. Furthermore, we briefly mentioned that the distance based on the L_k -norm can be computed in polynomial time if k is an even integer. To our knowledge, the complexity of comparing the probability distributions of two HMMs under the χ^2 distance, the Kullback-Leibler distance and the distance based on the L_r -norm for any r not 1, ∞ , or an even integer remains an open problem, though a heuristic for approximating the Kullback-Leibler distance was proposed in [15].

Acknowledgements

The authors would like to thank Peter Bro Miltersen for bringing our attention to the problems discussed in this paper. We furthermore acknowledge the help of Ole Østerby and Kevin Karplus.

References

1. N. Abe and M. K. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992. 426
2. K. Asai, S. Hayamizu, and K. Handa. Prediction of protein secondary structure by the hidden markov model. *Comp. Appl. in the Biosciences*, 9:141–146, 1993. 417
3. A. Bateman, E. Birney, R. Durbin, S. R. Eddy, K. L. Howe, and E. L. L. Sonnhammer. The Pfam protein families database. *Nucleic Acid Research*, 28:263–266, 2000. 417
4. T. Batu, L. Fortnow, R. Rubinfeld, W. D. Smith, and P. White. Testing that distributions are close. In *Proc. 15th STOC*, pages 259–269, 2000. 417
5. G. A. Churchill. Stochastic models for heterogeneous DNA sequences. *Bull. Math. Biol.*, 51:79–94, 1989. 417
6. T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, New York, 1991. 417, 423
7. L. Engebretsen and J. Holmerin. Clique is hard to approximate within $n^{(1-o(1))}$. In *Proc. 27th ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 2–12, 2000. 419
8. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l^1 -difference algorithm for massive data streams. In *Proc. 40th FOCS*, pages 501–511, 1999. 417
9. J. Fong and M. Strauss. An approximate l^p -difference algorithm for massive data streams. In *Proc. 17th STACS*, 2000. 417

10. J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999. [419](#), [422](#)
11. A. Krogh. Two methods for improving performance of an HMM and their application for gene finding. In *Proc. 5th ISMB*, pages 179–186, 1997. [417](#)
12. A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden markov models in computational biology: Applications to protein modeling. *Jour. Mol. Biol.*, 235:1501–1531, 1994. [417](#)
13. R. B. Lyngsø, C. N. S. Pedersen, and H. Nielsen. Metrics and similarity measures for hidden Markov models. In *Proc. 7th ISMB*, pages 178–186, 1999. [417](#), [423](#), [426](#), [427](#)
14. L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proc. of the IEEE*, volume 77, pages 257–286, 1989. [417](#), [419](#)
15. Y. Singer and M. K. Warmuth. Training algorithms for hidden Markov models using entropy based distance functions. In *Proc. 9th NIPS*, pages 641–647, 1996. [427](#)
16. E. L. L. Sonnhammer, G. von Heijne, and A. Krogh. A hidden Markov model for predicting transmembrane helices in protein sequences. In *Proc. 6th ISMB*, 1998. [417](#)

DNA Self-Assembly For Constructing 3D Boxes

(Extended Abstract)

Ming-Yang Kao^{1,*} and Vijay Ramachandran^{2,**}

¹ Department of Computer Science, Northwestern University
Evanston, IL 60201, USA
`kao@cs.northwestern.edu`

² Department of Computer Science, Yale University
New Haven, CT 06520-8285, USA
`vijayr@cs.yale.edu`

Abstract. We propose a mathematical model of DNA self-assembly using 2D tiles to form 3D nanostructures. This is the first work to combine studies in self-assembly and nanotechnology in 3D, just as Rothemund and Winfree did in the 2D case. Our model is a more precise superset of their Tile Assembly Model that facilitates building scalable 3D molecules. Under our model, we present algorithms to build a hollow cube, which is intuitively one of the simplest 3D structures to construct. We also introduce five basic measures of complexity to analyze these algorithms. Our model and algorithmic techniques are applicable to more complex 2D and 3D nanostructures.

1 Introduction

DNA *nanotechnology* and DNA *self-assembly* are two related technologies with enormous potentials.

The goal of DNA nanotechnology is to construct small objects with high precision. Seeman's visionary work [8] in 1982 pioneered the molecular units used in self-assembly of such objects. More than a decade later, double-crossover (DX) molecules were proposed by Fu and Seeman [3] and triple-crossover (TX) molecules by LaBean *et al.* [5] as DNA self-assembly building blocks. Laboratory efforts have been successful in generating interesting three-dimensional (3D) molecular structures, including the small cube of Chen and Seeman [1]. However, these are immutable and limited in size, mainly because their fabrication is not based on a mathematical model that can be extended as necessary.

In parallel to DNA nanotechnology, studies on self-assembly of DNA tiles have focused on using local deterministic binding rules to perform computations. These rules are based on interactions between exposed DNA sequences

* Supported in part by NSF Grants CCR-9531028 and EIA-0112934. Part of this work was performed while this author was visiting the Department of Computer Science, Yale University, New Haven, CT 06520-8285, USA, `kao-ming-yang@cs.yale.edu`.

** Supported by a 2001 National Defense Science and Engineering Graduate Fellowship.

on individual tiles; tiles assemble into a particular 1D or 2D structure when in solution, encoding a computation. Winfree [10] formulated a model for 2D computations using DX molecules. Winfree *et al.* [11] used 1D tiles for 1D computations and 2D constructions with DX molecules. LaBean *et al.* [4] were the first to compute with TX molecules.

Combining these two technologies, several researchers have demonstrated the power of DNA self-assembly in nanostructure fabrication. Winfree *et al.* [13] investigated how to use self-assembly of DX molecules to build 2D lattice DNA crystals. Rothemund and Winfree [7] further proposed a mathematical model and a complexity measure for building such 2D structures.

A natural extension of the seminal 2D results of Winfree *et al.* [13] and Rothemund and Winfree [7] would be the creation of 3D nanostructures using tiling. To initiate such an extension, this paper (1) proposes a general mathematical model for constructing 3D structures from 2D tiles; (2) identifies a set of biological and algorithmic issues basic to the implementation of this model; and (3) provides basic computational concepts and techniques to address these issues. Under the model, the paper focuses on the problem of constructing a hollow cube, which is intuitively one of the simplest 3D structures to construct. We present algorithms for the problem and analyze them in terms of five basic measures of complexity.

There are three natural approaches to building a hollow cube. The first approach uses 1D tiles to form 2D DX-type tiles as in [11], and then uses these tiles to construct a cube. Our paper does not fully investigate this possibility because of the inconvenient shape of these molecules (see Sect. 2.1), but our algorithms can be modified to accommodate these DX-type tiles. The second approach builds a cube from genuine 2D tiles, which is the focus of this paper. The third approach is perhaps the most natural: build a cube from genuine 3D tiles. It is not yet clear how such 3D tiles could be created; conceivably, the cube of Chen and Seeman [1] may lead to tiles of this form. This paper does not fully investigate this possibility, either, because this approach is algorithmically straightforward and similar to the 2D case.

The basic idea of our algorithms is to use 2D tiles to form a shape on the plane that can fold into a box, as illustrated in Fig. 1(a)–(b). We can easily synthesize a set of tiles to create the initial 2D shape. To overcome a negligible probability of success due to biochemical factors, we must put many copies of these tiles into solution at once; but we must then worry about multiple copies of the shape interfering with each other, preventing folding, as in Figure 1(c).

To avoid this problem, we introduce randomization, so that different copies of the shape have unique sticky ends. The growth of tiles into a complete structure must still be deterministic (as it is based on Watson-Crick hybridization), but we randomize the computation input — the *seed tiles* from which the rest of the shape assembles. The edges then still relate to each other, but depend on the random input that is different for each shape in solution. If each input can form with only low probability, interference with another copy of the shape will be kept to a minimum.

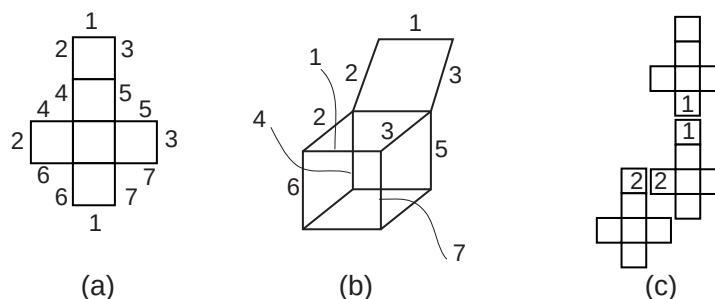


Fig. 1. (a) 2D planar shape that will fold into a box. Each section is formed from many smaller 2D DNA tiles. Edges with the same number have complementary sticky ends exposed so they can hybridize. (b) Folding of the shape in (a) into a box. Here, edges 4, 5, 6, and 7 have all hybridized. Hybridization of edges 2 and 3, whose two complements are now in close proximity, will cause edge 1 to hybridize and form the complete box. (c) Multiple copies of the 2D shape in solution. Copies of the shape can interfere and attach infinitely without control as long as edges have matching sticky ends

This raises another important issue — that of using self-assembly to communicate information from one part of the shape to another. Since the edges must relate to each other and the random input, designing local rules becomes nontrivial. In this paper, we explore and formalize patterns used in completing this task. In addition, we formalize biological steps that allow a specific subset of tiles to be added in an isolated period of time, thus allowing better control of growth. We couple this with the use of temperature to improve the probability of a successful construction.

The remainder of this paper is organized as follows. Sect. 2 describes the model of computation, including notation for DNA tiles and definitions of complexity measures. Sect. 3 describes the algorithms in detail, and Sect. 4 discusses future research possibilities. Some details have been omitted from this extended abstract; the full version is available electronically at

http://www.cs.yale.edu/~vijayr/papers/dna_3d.ps.

2 Model of Computation

In this section we formally introduce our model of self-assembly, the *Generalized Tile Assembly Model*, on both the mathematical and biological level. It is an extension of the model presented by Rothmund and Winfree in [7].

2.1 Molecular Units of Self-Assembly

We begin with the biological foundation for our model. We intend to build 3D structures using the folding technique shown in Fig. 1 and allow construction of all 2D structures possible with the Tile Assembly Model.

Our model relies on using the molecular building block of a *DNA tile*. Tiles can naturally hybridize to form stable shapes of varying sizes, and the individual tiles can easily be customized and replicated (via synthesis and PCR before the procedure) for a specific algorithm.

DNA tiles are small nucleotides with exposed action sites (also known as sticky ends of a DNA strand) consisting of a single-stranded sequence of base pairs. When this sequence matches a complementary sequence on an action site of another tile, the Watson-Crick hybridization property of DNA causes these two molecules to bind together, forming a larger structure. A tile can be synthesized in the laboratory to have specific sticky ends. Different combinations of sticky ends on a tile essentially yield uniquely-shaped puzzle pieces. The tiles will automatically hybridize when left in solution.

Most work in self-assembly uses DX and TX molecules for tiles, but the shape of these molecules causes a problem for 3D construction. Since the sticky ends are on diagonally opposite ends (see [3] and [5]), these tiles form structures with ragged edges when they hybridize, as in Fig. 2(a). Our algorithms can easily be modified to use these tiles by adjusting for proper alignment before folding into a box.

However, we propose a simpler alternative, which is using the branched molecules of Seeman [8] or a variant derived from the structure of tRNA. These molecules, sketched in Fig. 2(b) and (c), are truly 2D with sticky ends on four sides. The structure is stable while the sticky ends are free-floating in solution — so the molecules have flexibility to align properly during folding.

Such molecules offer a natural motivation for modeling them using Wang's theory of tiling [9], which allows us to abstract construction using these molecules to a symbolic level.

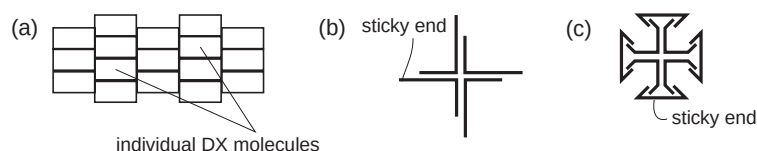


Fig. 2. (a) 2D structure formed from DX molecules. The left and right sides cannot hybridize because they are aligned improperly; the same is true for the top and bottom sides. (b) Branched-molecule DNA tile from [8]. (c) Synthetic DNA tile derived from the structure of tRNA

2.2 Symbolic Representation of Tiles

Definition 1. A DNA sequence of length n is an ordered sequence of base pairs $5' - b_1 b_2 \cdots b_n - 3'$ where the sequence has a 5-prime and 3-prime end, and $b_i \in \mathcal{B} = \{A, T, C, G\}$, the set of base pairs. We will assume that if the directions are not explicitly written, the sequence is written in the $5' \rightarrow 3'$ direction.

1. The Watson-Crick complement of sequence $s = 5' - b_1 b_2 \cdots b_n - 3'$, denoted $\bar{s}$, is the sequence¹ $3' - \bar{b}_1 \bar{b}_2 \cdots \bar{b}_n - 5'$, where $\bar{A} = T$, $\bar{C} = G$. Define $\overline{\bar{s}} = s$.
2. The concatenation of two sequences $s = s_1 \cdots s_n$ and $t = t_1 \cdots t_m$, denoted $s \cdot t$, or simply st , is the sequence $s_1 \cdots s_n t_1 \cdots t_m$.
3. The subsequence from i to j of sequence $s = 5' - b_1 b_2 \cdots b_n - 3'$, denoted $s[i : j]$, is the sequence $5' - b_i b_{i+1} \cdots b_{j-1} b_j - 3'$, where $1 \leq i < j \leq n$.

Given the above definitions, two DNA strands can hybridize if they have complementary sequences. Formally, $s = s_1 \cdots s_n$ and $t = t_1 \cdots t_m$ can hybridize if there exist integers $h_{s1}, h_{s2}, h_{t1}, h_{t2}$ such that $s[h_{s1} : h_{s2}] = \bar{t}[h_{t1} : h_{t2}]$. We assume there are no misbindings, that is, the above condition must be met exactly with no errors in base-pair binding.

Remark 1. Note that $\overline{st}$ (or $\overline{(s \cdot t)}$) $\neq \bar{s}\bar{t}$ (or $\bar{s} \cdot \bar{t}$); rather, $\overline{st} = \bar{t} \cdot \bar{s}$.

Definition 2. The threshold temperature for a DNA sequence is a temperature t in some fixed set $\mathcal{T}$ such that the sequence is unable to remain stably hybridized to its complement when the solution is at a temperature higher than $t' \in (t - \epsilon, t + \epsilon)$ for $\epsilon > 0$.² (Heating a solution generally denatures strands, so this definition has strong biological foundation. The consequences and methodology of using temperature in designing DNA sequences for tiles is discussed in [2].) If s has a lower threshold temperature than t , we say s binds weaker than t .

As with most work in DNA computing, our model uses DNA sequences to encode information³—in our case, an identifier specifying what kinds of matches are allowed between tiles on a given side. Since there are no misbindings, these identifiers map uniquely to DNA sequences present on the sides of tiles that can bind to each other. Formally, we have the following.

Definition 3. Let $\mathcal{S}$ be the set of symbols used to represent the patterns on the sides of our tiles. We assume $\mathcal{S}$ is closed under complementation, that is, if $s \in \mathcal{S}$ then there exists some $s' \in \mathcal{S}$ such that $s' = \bar{s}$ where $\bar{s}$ is the complement of s (the purpose of this will be clear below). Let $\mathcal{W} \subset \bigcup_i \mathcal{B}^i$ be the set of DNA

¹ The assumption that sequences written without directions are given $5' \rightarrow 3'$ means that the complement of $b_1 b_2 \cdots b_n$ is $\bar{b}_n \cdots \bar{b}_2 \bar{b}_1$, which is not standard convention but is technically correct.

² A fixed set of threshold temperatures simplifies the model and corresponds to the temperature parameter in [7]. To compensate we allow the actual threshold temperature to deviate slightly from the fixed point.

³ Condon, Corn, and Marathe [2] have done work on designing good DNA sequences for problems like this one.

sequences called DNA words such that the words do not interfere with each other or themselves (i.e., bind inappropriately). We then define the injective map $\text{enc} : \mathcal{S} \rightarrow \mathcal{W}$ that is the encoding of a symbol into a DNA word. This map obeys complementation: $\text{enc}(\bar{s}) = \text{enc}(s)$.

Definition 4. A DNA tile is a 4-tuple of symbols $T = (s_N, s_E, s_S, s_W)$ such that $s_i \in \mathcal{S}$ and $\text{enc}(s_i)$ is the exposed DNA sequence at the north, east, south, or west action site of the tile, for $i = N, E, S, W$. Given two tiles T_1 and T_2 , they will bind if two sides have complementary symbols. Properties of hybridization, including threshold temperature, carry over to the hybridization of tiles. We make a stronger no-misbinding assumption for tiles, requiring that the sticky ends on the tiles match exactly and fully.

At this stage, our model exactly matches that of Rothemund and Winfree in [7], except that our tiles can “rotate”; that is, $(s_N, s_E, s_S, s_W) = (s_E, s_S, s_W, s_N)$. This corresponds more closely to tile structure. The model, at this point, could require many symbols to express different tile types, and possibly an exponential number of DNA words. Ideally, we would like to arbitrarily extend the symbolic or informational content of each side of a tile. Therefore we make the following generalization.

Definition 5. Let Σ be a set of symbols closed under complementation, and let Ω be a set of corresponding DNA words. A k -level generalization of the model defines a map $g : \Sigma^k \rightarrow \mathcal{S}$ and a corresponding encoding $\text{genc} : \Sigma^k \rightarrow \mathcal{W}$, where $\text{genc}(\sigma) = \text{enc}(g(\sigma))$ for $\sigma \in \Sigma^k$ such that an abstract tile definition, which is a 4-tuple of k -tuples of symbols in Σ , is equivalent to a DNA tile.

We define complementation for a k -tuple in Σ^k as follows: let $\bar{\sigma} = (\bar{\sigma}_1, \dots, \bar{\sigma}_k)$ be $(\bar{\sigma}_1, \dots, \bar{\sigma}_k)$ so that $\text{genc}(\bar{\sigma}) = \text{enc}(g(\bar{\sigma})) = \text{enc}(g(\sigma)) = \text{enc}(g(\sigma))$. This makes the hybridization condition equivalent to having complementary symbols in k -tuples for sides that will bind.

The definition is purposefully broad in order to allow different algorithms to define the encoding based on the number of words and tiles needed. A 1-level generalization with $\Sigma = \mathcal{S}$ and $\Omega = \mathcal{W}$ where $g(s) = s$ and $\text{genc} = \text{enc}$ is the original Rothemund-Winfree Tile Assembly Model. In this paper, we use the following model.

Definition 6. The concatenation generalization is a k -level generalization where $\mathcal{W} \subset \Omega^k$ and g maps every combination of symbols in Σ^k to a unique symbol in $\mathcal{S}$. Partition $\mathcal{S}$ into $\mathcal{S}'$ and $\bar{\mathcal{S}}'$ such that each set contains the complement symbols of the other, and $\mathcal{S}' \cap \bar{\mathcal{S}}' = \emptyset$.⁴ For $\sigma \in \mathcal{S}'$, define $\text{genc}(g^{-1}(\sigma)) = \text{enc}(\sigma) = \omega_1\omega_2 \cdots \omega_k$, where $\text{enc}(\sigma_i) = \omega_i \in \Omega$ and $g^{-1}(\sigma) = (\sigma_1, \sigma_2, \dots, \sigma_k)$. Then for $\bar{\sigma} \in \bar{\mathcal{S}}'$, let $\text{enc}(\bar{\sigma}) = \bar{\omega}_k \cdot \bar{\omega}_{k-1} \cdots \bar{\omega}_1$, so that $\text{genc}(g^{-1}(\bar{\sigma})) = \text{genc}(g^{-1}(\sigma))$.

⁴ The map enc as defined will be one-to-one, and so is g , and so this can be done since the DNA sequence corresponding to any symbol has a unique complement, and therefore a unique complement symbol.

In other words, the concatenation generalization model is a straightforward extension of the tile model where each side of a tile corresponds to a k -tuple of symbols, where the DNA sequence at the corresponding action site is simply the concatenation of the encodings of the individual symbols.⁵ Using this simple model, we can reduce the number of DNA words needed to $|\Sigma|$ from $|\Sigma|^k$, and create simpler descriptions of our tiles.

2.3 Algorithmic Procedures

With the above models for tiles, we now discuss procedures for growing larger structures.

We follow Rothmund and Winfree [7] and Markov [6] and use the common self-assembly assumption that a structure begins with a seed tile and grows, at each timestep, by hybridization with another free-floating tile.⁶ The new tile hybridizes at a given position following one of two types of rules:

Deterministic Given the surrounding tiles at that position, only one tile type, with specific sticky ends on the non-binding sides, can fit.

Randomized Multiple tile types (with different sticky ends on the non-binding sides) could fit the position given the tiles present; a new action site is created with probability proportional to the concentration of its tile type in solution.

Therefore, to grow a structure, an algorithm repeats *steps* until the structure is complete: add tiles to solution; wait for them to adhere to the growing structure; optionally removes excess tiles from solution by “washing them away.” Cycling temperature during these steps to prevent or induce binding (based on threshold temperatures) can be done while waiting for hybridization, and is called *temperature-sensitive binding*.

2.4 Complexity

We consider five basic methods of analyzing algorithms using our model.

Time complexity Each algorithm is a sequence of self-assembly steps described above, thus the natural measure of *time complexity* in our model is the number of steps required (which describes laboratory time).

Space complexity The number of distinct physical tile types (not the actual number of molecules produced) is *space complexity*. Introduced by [7], this describes the amount of unique DNA synthesis necessary.

⁵ Potentially, the concatenation model could cause interference among tiles. If we maintain no misbindings, however, our model removes this from analysis. In addition, it is theoretically possible to design DNA words so interference does not occur, depending on the algorithm.

⁶ In reality, multiple tiles can hybridize at once and structures consisting of more than one tile can hybridize to each other, but we lose no generality with the Markov assumption.

Alphabet size The number of DNA words, or $|\Omega|$ or $|\mathcal{W}|$, has a rough laboratory limit [2], and so the size of the symbol set used ($|\Sigma|$ or $|\mathcal{S}|$), which corresponds directly to the number of words, has practical significance.

Generalization level The generalization level is the amount of information on a side of a tile. This is related to the length of the sticky ends (and thus has biological consequences) and the number of actual DNA words (via $|\mathcal{S}|$).

Probability of misformation *Misformed structures* contain tiles that are not bound properly on all sides. Assuming the Markov model, consider adding tile T to a partial structure S . If complete hybridization requires binding on two sides, but T manages to hybridize only on one side (while the other action site does not match), $S + T$ has a *misformation*. We quantify this probability with the following.

Definition 7. Let the success probability at step t be the probability that a free-floating tile in solution binds at all possible sides to a partial structure at a given spot. (Step t is the addition of a tile to that spot on structure S_t , resulting in S_{t+1} .) This is

$$\Pr(S_{t+1} \text{ is correct} | S_t \text{ is correct}) = \frac{N_{\text{correct}}}{N_{\text{all}}},$$

where N_{correct} is the number of tile types that can correctly bind, while N_{all} is the number of tile types in solution that could bind, possibly even incompletely. Call this q_t . Then the misformation probability at step t is $p_t = 1 - q_t$. If the algorithm has k additions, then the misformation probability for the algorithm is $1 - q_0 q_1 \cdots q_{k-1}$. Then an algorithm is *misformation-proof* if its misformation probability at every step is zero, yielding a zero total probability of misformation.

3 Hollow Cube Algorithms

In this section, we examine algorithms designed to use our model to build a 3D hollow cube using the folding technique shown in Fig. 1. Let the length of a side of the cube, n , be the input to the algorithms. We present the most interesting algorithm in detail and defer the discussion of others considered to the full version of the paper.

3.1 Overview

Figure 3(a) illustrates the planar shape our algorithms construct. We will reference the labels and shading of regions in the figure during our discussion.

As stated earlier in Sect. 1, we must make each shape unique so different partial structures in solution do not bind to and interfere with each other. Once we have a unique seed structure, we can then use self-assembly with basic rules to make the edges of the shape correspond so folding will occur.

There are three basic self-assembly *patterns* used to construct different parts of the shape.

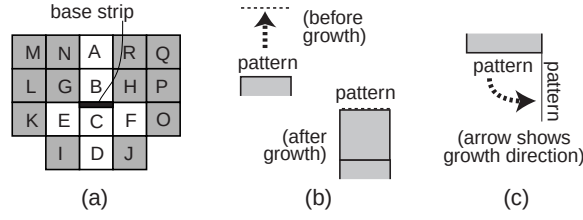


Fig. 3. (a) Regions of the 2D planar shape. (b) A straight-copy pattern. (c) A turn-copy pattern

Random assembly Implements a random rule (see Sect. 2.3). Formally, add all tiles in a set of distinct tiles R in equal concentrations so each could potentially hybridize completely at a given position. Thus the information at that position is completely random. The tiles differ by a component of their exposed k -tuples, assuming a k -level generalization.

Straight copy See Fig. 3(b). Tiles are added to copy the pattern along one end of a region through to a parallel end of an adjacent region being constructed. This rule is deterministic.

Turn copy See Fig. 3(c). Tiles are added to copy the pattern along one end of a region to a perpendicular end of an adjacent region being constructed. Counters will be required to position the tiles appropriately to complete this deterministic rule.

The algorithm begins by assembling a random pattern string that will be copied to the top and bottom of the box. Then random patterns are added for the remaining edges of the box, and these are copied to the corresponding edges accordingly. (Refer to Fig. 1 for corresponding edges.) Finally, the shape that will fold is cut out by raising the temperature, assuming that the bonds between tiles along the region-borderline have weak threshold temperatures. The regions that are shaded in Fig. 3(a) are cut away.

3.2 Notation

All of our algorithms will use a 3-level concatenation generalization model; thus a tile is a 4-tuple of triplets that we write $T_N \times T_S \times T_W \times T_E$, with each $T_i = (\sigma_1, \sigma_2, \sigma_3)$. We change the order of the tuple to more easily identify tiles that will bind, since most binding will be north-south or west-east. (This decision is arbitrary and purely notational.) We assume all tiles are oriented so that the directions are clear.

We define the set $\Pi \subset \Sigma$ to be the “random patterns” $\pi_1, \pi_2, \dots, \pi_p$ used as components of exposed triplets for tiles used in random assembly; their use will become clear when we discuss implementation of random assembly below.

We use counters to control the growth of our planar shape. This concept has been well explored in [7] and earlier papers. Each tile can be assigned a

position in the plane denoted by a horizontal and vertical coordinate. We create symbols for position counters and then allow tiles to hybridize if the positions match, creating a mechanism for algorithms to place tiles in absolute or relative positions. Let $H(i)$ and $V(j)$ be the symbols denoting horizontal position i and vertical position j , respectively.

3.3 Row-by-Row Algorithm

Summary After random assembly of the base strip, we use a row-by-row straight copy using a one-dimensional counter through regions **A-D** to copy the base strip's pattern. We then use straight copy to fill in the bodies of regions **E** and **F** and add the edges using random assembly, as these will correspond to other portions of the shape. We then use a turn copy through **G-J** to make those edges correspond. Finally, we do a sequence of straight and turn copies from **E** and **F** through **K-N** and **O-R** to complete the shape. We then raise the temperature to cut away the shaded regions.

Implementation of the self-assembly patterns are discussed briefly below. A complete discussion, including the specific tiles added for each step, can be found in the full version of the paper.

Implementation of Random Assembly The base strip shown in Fig. 3(a) contains the unique pattern that is copied to the edges of the shape. Randomness is achieved by adding tiles of type

$$(\pi_k, \kappa_2, V(0)) \times (\overline{\pi_k}, \overline{\kappa_2}, \overline{V(-1)}) \times (\kappa_1, H(i), V(0)) \times (\overline{\kappa_1}, \overline{H(i+1)}, \overline{V(0)}) \quad (1)$$

where the tiles in (1) vary over all i , $2 \leq i \leq n-2$, and all k , $1 \leq k \leq |II|$. Thus at any position i , a tile with any pattern π_k can adhere, so the final sequence of patterns on the assembled shape will be unique. The use of counters assures the strip will have length n .

Implementation of Straight Copy The straight-copy pattern for a region requires $2n+1$ steps. (Some regions can be done in parallel, so the number of steps is less than $2n+1$ times the number of straight-copy regions.) One counter, in the direction of growth, is used. Tiles are added one row at a time to prevent misformations; at each timestep, only tiles for the current row are added, which ensures (with the help of temperature) that the dominant factor in binding is the pattern sequence (of π_k 's). For example, tiles like the following are added, where the constant $X = (\varphi_1, \varphi_2, \varphi_3)$, for each step i , $1 \leq i \leq 2n-2$ and all patterns $\pi_k \in II$:

$$(\pi_k, \kappa_2, V(i)) \times (\overline{\pi_k}, \overline{\kappa_2}, \overline{V(i-1)}) \times X \times \overline{X} \quad (2)$$

$$(\pi_k, \kappa_2, V(-i)) \times (\overline{\pi_k}, \overline{\kappa_2}, \overline{V(-i-1)}) \times X \times \overline{X} \quad (3)$$

We assume that X binds weaker than κ_2 , so cycling the temperature ensures the tiles are attached on the κ_2 side.

Implementation of Turn Copy The turn-copy step, for example, copying the bottom edge of **E** to the left edge of **D** through **I** so the shape can fold, is done using vertical and horizontal counters, which essentially places a tile in a specific spot. Therefore we can add all the tiles at once to complete the region without possibility of misformation. For the above example region, we would add the following tiles. Let i and j vary such that $-n \leq i \leq -1$ and $-2n + 1 \leq j \leq -n$. For all i, j , add:

$$i = j + (n - 1), (\pi_k, H(i), V(j)) \times (\overline{\kappa_3}, \overline{H(i)}, \overline{V(j - 1)}) \times (\kappa_3, H(i), V(j)) \times (\overline{\pi_k}, \overline{H(i - 1)}, \overline{V(j)}) \quad (4)$$

$$i < j + (n - 1), (\pi_k, H(i), V(j)) \times (\overline{\pi_k}, \overline{H(i)}, \overline{V(j - 1)}) \times (\kappa_3, H(i), V(j)) \times (\overline{\kappa_3}, \overline{H(i - 1)}, \overline{V(j)}) \quad (5)$$

$$i > j + (n - 1), (\kappa_3, H(i), V(j)) \times (\overline{\kappa_3}, \overline{H(i)}, \overline{V(j - 1)}) \times (\pi_k, H(i), V(j)) \times (\overline{\pi_k}, \overline{H(i - 1)}, \overline{V(j)}) \quad (6)$$

As an extra precaution we can set κ_3 binding to be weaker than π_k , and cycle the temperature several times. In addition, we force the encoding of horizontal and vertical counters at the edges where the folding occurs to be the same, so that the sticky ends are in fact complementary.

Analysis of the Row-by-Row Algorithm Proofs for the following can be found in the full version of the paper; most are clear from evaluating, in detail, the steps described above.

Theorem 1. $|\Sigma| = 8n + |H| + O(1)$.

Theorem 2. *The algorithm has time complexity approximately $5n$.*

Theorem 3. *The space complexity of the row-by-row algorithm is approximately $6|H|n^2 + 10|H|n + 4|H| + 8n$ tiles.*

Theorem 4. *The number of distinct temperatures required is 3.*

Theorem 5. *The misformation probability of row-by-row is 0.*

Clearly, this algorithm does not take advantage of all the parallelism possible with self-assembly and has a rather large time complexity. The full version of the paper discusses other algorithms where fewer steps are required (in particular, straight copy is not done row-by-row), but also gives analysis showing that such algorithms have an increased misformation probability.

4 Conclusion

Our paper introduces a precise extension to the Tile Assembly Model [7] that allows greater information content per tile and scalability to three dimensions. The model better formalizes the abstraction of DNA tiles to symbols, introduces five complexity measures to analyze algorithms, and is the first to extend nanostructure fabrication to three dimensions.

In addition, our paper opens up wide-ranging avenues of research.

First of all, it may be possible to encode information on tiles more succinctly than our algorithms do to accomplish the copy patterns discussed. The existence of good 2-level or 1-level generalization algorithms is unknown.

Algorithms to form other 3D structures, having various applications in biology and computation, can be studied. More work also must be done to quantify the probabilities specified in the paper (possibly including a free-energy analysis of tile binding).

Finally, there remain some important biological issues. In particular, design of a strong tile suitable for our method of computation and design of a 3D building block are two important steps to increasing the feasibility of 3D self-assembly. The use of temperature may be further refined and exploited to improve some complexity results and the number of steps needed in the lab.

References

1. J. Chen and N. C. Seeman. The synthesis from DNA of a molecule with the connectivity of a cube. *Nature*, 350:631–633, 1991. 429, 430
2. A. Condon, R. M. Corn, and A. Marathe. On combinatorial DNA word design. In Winfree and Gifford [12]. 433, 436
3. T.-J. Fu and N. C. Seeman. DNA double-crossover molecules. *Biochemistry*, 32:3211–3220, 1993. 429, 432
4. T. H. LaBean, E. Winfree, and J. H. Reif. Experimental progress in computation by self-assembly of DNA tilings. In Winfree and Gifford [12]. 430
5. T. H. LaBean, H. Yan, J. Kopatsch, F. Liu, E. Winfree, H. Reif, and N. Seeman. The construction, analysis, ligation and self-assembly of DNA triple crossover complexes. *J. Am. Chem. Soc.*, 122:1848–1860, 2000. 429, 432
6. I. V. Markov. *Crystal Growth for Beginners: Fundamentals of Nucleation, Crystal Growth, and Epitaxy*. World Scientific, Singapore, 1995. 435
7. P. Rothmund and E. Winfree. The program-size complexity of self-assembled squares. In F. F. Yao, editor, *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, Portland, OR, 21–23 May 2000. ACM Special Interest Group on Algorithms and Computation Theory. 430, 431, 433, 434, 435, 437, 439
8. N. C. Seeman. Nucleic-acid junctions and lattices. *Journal of Theoretical Biology*, 2:237–247, 1982. 429, 432
9. H. Wang. Proving theorems by pattern recognition. *Bell System Technical Journal*, 40:1–42, 1961. 432
10. E. Winfree. On the computational power of DNA annealing and ligation. In E. B. Baum and R. J. Lipton, editors, *DNA Based Computers*, DIMACS: Series in Discrete Mathematics and Theoretical Computer Science, pages 199–210. American Mathematical Society, May 1995. 430
11. E. Winfree, T. Eng, and G. Rozenberg. String tile models for DNA computing by self-assembly. In A. Condon and G. Rozenberg, editors, *DNA Based Computers VI*, Leiden, The Netherlands, 13–17 June 2000. Leiden Center for Natural Computing. 430
12. E. Winfree and D. Gifford, editors. *Preliminary Proceedings, Fifth International Meeting on DNA Based Computers*, Cambridge, Massachusetts, 14–15 June 1999. DIMACS. 440

13. E. Winfree, F. Liu, L. A. Wenzler, and N. C. Seeman. Design and self-assembly of two-dimensional DNA crystals. *Nature*, 394:539–544, 1998. 430

Exact Solutions for CLOSEST STRING and Related Problems

Jens Gramm^{1*}, Rolf Niedermeier¹, and Peter Rossmanith²

¹ Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Sand 13, D-72076 Tübingen, Germany
{`gramm,niederm`}@informatik.uni-tuebingen.de

² Institut für Informatik, Technische Universität München,
Arcisstr. 21, D-80290 München, Germany
`rossmani@in.tum.de`

Abstract. CLOSEST STRING is one of the core problems in the field of consensus word analysis with particular importance for computational biology. Given k strings of same length and a positive integer d , find a “closest string” s such that none of the given strings has Hamming distance greater than d from s . CLOSEST STRING is NP-complete. We show how to solve CLOSEST STRING in linear time for constant d (the exponential growth is $O(d^d)$). We extend this result to the closely related problems d -MISMATCH and DISTINGUISHING STRING SELECTION. Moreover, we discuss fixed parameter tractability for parameter k and give an efficient linear time algorithm for CLOSEST STRING when $k = 3$. Finally, the practical usefulness of our findings is substantiated by some experimental results.

1 Introduction

Finding signals in DNA is a major problem in computational biology. A recently intensively studied facet of this problem is based on consensus word analysis [10, Section 8.6]. A central problem herein is the so-called CLOSEST STRING (or, equivalently, CONSENSUS STRING or CENTER STRING) problem: Given k strings $s_1, s_2, \dots, s_k$ over alphabet Σ of length L each, and a positive integer d , is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \dots, k$? Here, $d_H(s, s_i)$ denotes the Hamming distance between strings s and s_i . Related problems we also study here are the d -MISMATCH problem (which generalizes CLOSEST STRING in the way that we look for center strings of *aligned* substrings of a given set of strings) [11,12] and the so-called DISTINGUISHING STRING SELECTION problem [6] (for a brief overview on biological applications concerning signal finding and primer design refer to, e.g., [6]). All these problems are, in general, NP-hard [4,6]¹.

* Supported by the Deutsche Forschungsgemeinschaft (DFG), project OPAL (optimal solutions for hard problems in computational biology), NI 369/2-1.

¹ Frances and Litman [4] show the NP-completeness of CLOSEST STRING, considering it from the viewpoint of coding theory (so-called MINIMUM RADIUS problem).

Despite their hardness, these problems need to be solved in practice. Li *et al.* [9] gave a polynomial time approximation scheme (PTAS) for CLOSEST STRING. The constants and polynomials occurring in the running time, however, make this result of little practical value. Another very promising approach is to study the parameterized complexity [1,2] of these problems. Consider the two most natural parameters of CLOSEST STRING: the maximum Hamming distance d allowed and the number k of given input strings. Under the natural assumption that either d or k is (very) small (in particular, in biological applications it is appropriate to assume small d , e.g., $d < 10$ [3]), it is important to ask whether efficient polynomial or even better linear time algorithms are possible when d or k are constants. Put in slightly more general terms, this is the question for the *fixed parameter tractability* of these problems.

We present the following results. CLOSEST STRING can be solved in time $O(kL + kd \cdot d^d)$, yielding a linear time search tree algorithm for constant d . This answers the open question of Evans and Wareham [3] for the parameterized complexity of CLOSEST STRING with parameter d . Furthermore, we can generalize our result to d -MISMATCH, improving work and positively answering an open question of Stojanovic *et al.* [11], where a linear time algorithm for only $d = 1$ was given. Also, our result is extendible to DISTINGUISHING STRING SELECTION, for which we can derive a linear time algorithm in case of constant distance parameters and constant alphabet size. Our second, technically more involved main result is that CLOSEST STRING can be solved efficiently in linear time for $k = 3$. Using an integer linear program formulation, we can observe that the problem is fixed parameter tractable with respect to k —the exponential term in k is huge, however. Finally, we indicate the practical usefulness and potential of our algorithms by some experimental results based on implementations of our linear time algorithm for constant d . Due to the lack of space, we omit some proofs and details.

2 Preliminaries

For a string s of length L , we use $s[p]$, $1 \leq p \leq L$, to denote the character at position p in s . Then, $d_H(s_i, s_j)$ denotes the Hamming distance between strings s_i and s_j of same length L , i.e., $|\{p \mid s_i[p] \neq s_j[p]\}|$. Given a set of strings $S = \{s_1, s_2, \dots, s_k\}$, each string of length L , then a string s is an *optimal closest string* for S iff there is no string s' with $\max_{i=1, \dots, k} d_H(s', s_i) < \max_{i=1, \dots, k} d_H(s, s_i)$. By way of contrast, $\sum_{i=1, \dots, k} d_H(s', s_i) < \sum_{i=1, \dots, k} d_H(s, s_i)$. An optimal median string for $S = \{s_1, s_2, \dots, s_k\}$ can be computed by choosing in every column the letter occurring most often. We call this a *majority vote*; it, however, is not necessarily unique.

Given a set of k strings of length L , we can think of these strings as a $k \times L$ character matrix. The *columns* of a CLOSEST STRING instance are the columns of this matrix. For reordering the columns, we use a permutation on strings as follows. Given a string $s = c_1 c_2 \dots c_L$ with $c_1, \dots, c_L \in \Sigma$ for alphabet Σ and a permutation $\pi : \{1, \dots, L\} \rightarrow \{1, \dots, L\}$. Then, $\pi(s) = c_{\pi(1)} c_{\pi(2)} \dots c_{\pi(L)}$.

Lemma 1 *Given a set of strings $S = \{s_1, s_2, \dots, s_k\}$, each of length L , and a permutation $\pi : \{1, \dots, L\} \rightarrow \{1, \dots, L\}$. Then s is an optimal closest string for $\{s_1, s_2, \dots, s_k\}$ iff $\pi(s)$ is an optimal closest string for $\{\pi(s_1), \pi(s_2), \dots, \pi(s_k)\}$.*

Several columns can be identified due to *isomorphism*. The reason for this is the fact that the columns are independent from each other in the sense that the distance from the closest string is measured columnwise. For instance, consider the case of the two columns $(a, a, b)^t$ and $(b, b, a)^t$ when $k = 3$. Clearly, these two columns are isomorphic. Isomorphic columns form *column types*.

This can be generalized as follows. W.l.o.g., let a always denote the letter that occurs in a column most often, let b always denote the letter that has the second most often occurrences and so on. This property of being *normalized* can be easily achieved by a simple linear time preprocessing of the input instance. In addition, solving the normalized problem optimally, one again can compute the optimal solution of the original problem instance by simply reversing the above mapping done by the preprocessing. Hence:

Lemma 2 *To compute an optimal closest string, it is sufficient to solve a normalized and reordered instance. From this, the solution of the original instance can be derived in linear time.*

In the following, we call two input instances *isomorphic* if there is a one-to-one correspondence between the columns of both instances such that each thus determined pair of columns is isomorphic.

Lemma 3 *A CLOSEST STRING instance with arbitrary alphabet Σ , $|\Sigma| > k$, is isomorphic to a CLOSEST STRING instance with alphabet Σ' , $|\Sigma'| = k$.*

With the following observation by Evans and Wareham [3], we find that it is sufficient to solve instances containing less than kd columns. This yields a so-called “reduction to problem kernel” [1,2]. We call a column *dirty* iff it contains at least two different symbols from alphabet Σ . Clearly, “all the work” in solving CLOSEST STRING concentrates on the dirty columns of the input instance.

Lemma 4 *Given a CLOSEST STRING instance with k strings of length L and integer d . If the resulting $k \times L$ matrix has more than kd dirty columns, then there is no solution to this instance.*

3 A Linear Time Solution for Constant d and Applications

We show that CLOSEST STRING, although NP-complete in general, is solvable in linear time for constant d , discuss heuristic improvements, and apply this result to the related problems of d -MISMATCH and DISTINGUISHING STRING SELECTION.

Algorithm D, recursive procedure $CSd(s, \Delta d)$ Global variables: Set of strings $S = \{s_1, s_2, \dots, s_k\}$, integer d .Input: Candidate string s and integer Δd .Output: A string $\hat{s}$ with $\max_{i=1, \dots, k} d_H(\hat{s}, s_i) \leq d$ and $d_H(\hat{s}, s) \leq \Delta d$, if it exists, and “not found,” otherwise.

- (D0) If $(\Delta d < 0)$, then return “not found”;
- (D1) If $(d_H(s, s_i) > d + \Delta d)$ for some $i \in \{1, \dots, k\}$, then return “not found”;
- (D2) If $(d_H(s, s_i) \leq d)$ for all $i = 1, \dots, k$, then return s ;
- (D3) Choose $i \in \{1, \dots, k\}$ such that $d_H(s, s_i) > d$:
 - $P := \{p \mid s[p] \neq s_i[p]\}$;
 - Choose any $P' \subseteq P$ with $|P'| = d + 1$;
 - For all $p \in P'$ do
 - $s'_p := s$;
 - $s'_p[p] := s_i[p]$;
 - $s_{ret} := CSd(s'_p, \Delta d - 1)$;
 - If $s_{ret} \neq \text{“not found”}$, then return s_{ret} ;
- (D4) Return “not found”

Fig. 1. Algorithm D. Inputs are a CLOSEST STRING instance consisting of a set of strings $S = \{s_1, s_2, \dots, s_k\}$ of length L , and an integer d . First, we perform a preprocessing performing the reduction to problem kernel as shown in Lemma 4: We select the dirty columns. If there are more than kd many, then we reject the instance. If there are at most kd many, then we invoke the recursion with $CSd(s_1, d)$

3.1 Bounded Search Tree Algorithm

In Fig. 1, we outline a recursive algorithm solving CLOSEST STRING. It is based on the well-known bounded search tree paradigm often successfully applied in parameterized complexity [1, 2]. For the correctness of the algorithm, we need the following simple observation.

Lemma 5 *Given a set of strings $S = \{s_1, s_2, \dots, s_k\}$ and a positive integer d . If there are $i, j \in \{1, \dots, k\}$ with $d_H(s_i, s_j) > 2d$, then there is no string s with $\max_{i=1, \dots, k} d_H(s, s_i) \leq d$.*

Theorem 1 *Algorithm D solves CLOSEST STRING in time $O(kL + kd \cdot d^d)$.*

Proof. (Sketch) **Running time.** Prior to the recursion, we perform the reduction to problem kernel as described in Lemma 4. This preprocessing, reducing the size of the input instance to kd , can be done in time $O(kL)$. Now, we consider the recursive part of the algorithm. Parameter Δd is initialized to d . Every recursive call decreases Δd by one. The algorithm stops when $\Delta d < 0$. Therefore, the algorithm builds a search tree of height at most d . In one step of the recursion, the algorithm chooses, given the current candidate string s , a string s_i

such that $d_H(s, s_i) > d$. It creates a subcase for $d + 1$ of the positions in which s and s_i disagree (there are more than d but at most $2d$ such positions). This yields an upper bound of $(d + 1)^d$ on the search tree size. Every step of the recursion only needs linear time $O(kd)$. Before starting the recursion, we build a table containing the distances of the candidate s to all other given strings in time $O(kd)$. Using this table, instructions (D1) and (D2) can be done in time $O(k)$. In instruction (D3), we need time $O(k)$ to select the s_i for branching and time $O(kd)$ to find the positions in which s and s_i differ. For $d + 1$ of the differing positions we modify the candidate, update the table of distances, and call the procedure recursively. Since we changed only one position, we can update the table of distances in time $O(k)$.

Correctness. We have to show that Algorithm D will find a string s with $\max_{i=1, \dots, k} d_H(s, s_i) \leq d$, if one exists. Here, we explicitly show only the correctness of the first recursive step; the correctness of the algorithm then follows with an inductive application of the argument.

In the situation that s_1 satisfies $\max_{i=1, \dots, k} d_H(s_1, s_i) \leq d$, we immediately find a solution, namely s_1 . If s_1 is not a solution but there exists a closest string s for this instance with distance value d , then there is a string s_i , $i = 2, \dots, k$, such that $d_H(s_1, s_i) > d$. For branching, we consider the positions where s_1 and s_i differ, i.e., $P := \{p \mid s_1[p] \neq s_i[p]\}$. Algorithm D successively creates subcases for $d + 1$ positions p from P in order to create a new candidate by altering the respective position p from $s_1[p]$ to $s_i[p]$. Such a “move” is correct if we choose a position p from $P_1 := \{p \mid s_1[p] \neq s[p] = s_i[p]\}$. Now, we show that (at least) one of our $d + 1$ moves is a correct one. We observe that $P = P_1 \cup P_2$ for $P_2 := \{p \mid s[p] \neq s_i[p]\}$. Since $d_H(s, s_i) \leq d$, we know that $|P_2| \leq d$. Therefore, at least one of our $d + 1$ subcases will try a position from P_1 . Regarding instruction (D1), we can analogously to Lemma 5 observe that it is correct to omit those branches where the candidate string s satisfies $d_H(s, s_i) > d + \Delta d$ for some string s_i of the given strings $s_1, \dots, s_k$. $\square$

With Algorithm D, we can find a solution if one exists. We find *all* solutions if the given distance parameter d is optimal. We do not necessarily find all solutions to a given instance when d is not optimal. Using binary search, however, we can find the optimal distance value $\leq d$ at the cost of a constant time factor.

Finally, we note that we can further improve the exponential term d^d significantly by asymptotic considerations. We defer this to the long version of the paper.

3.2 Heuristic Improvements

Since the search tree size is the critical component in the algorithm’s running time, the goal is to keep it as small as possible. Keeping in mind the initial candidate string, there is no use in changing a position that has already been changed before. In addition, we store all characters that have been tried on the same or a higher level of recursion (and restored again after the corresponding branch of the search tree has been visited); there are at most k many for every

position. For branching, we consider only setting a character at a position if we didn't already try this character on the same or a higher level of recursion.

Concerning branching, a good strategy seems to be to select, of the strings s_i , $i = 1, \dots, k$ with $d_H(s, s_i) > \Delta d$, the string with maximal $d_H(s, s_i)$. Moreover, since we search the solution in the neighborhood of the initial s , a good choice is an s which is presumably close to the (unknown) solutions. A possible strategy is to select the string with a minimum median distance to all other strings.

3.3 Enhancements and Related Problems

As mentioned before, our basic algorithm does not find *all* solutions. We can, however, modify it in order to deliver all solutions in time generally better than a trivial brute force approach. We omit the details.

Solving d -MISMATCH. Let $s_{i,p,L}$ denote the length- L substring of a given string s_i starting at position p . Then, given strings $s_1, s_2, \dots, s_k$ of length n and integers L and d , the d -MISMATCH problem is the question of whether there is a string s of length L and a position p with $1 \leq p \leq n - L + 1$, such that $d_H(s, s_{i,p,L}) \leq d$ for all $i = 1, \dots, k$. We achieve a linear running time for constant d as follows. We use the problem kernel of size kd for CLOSEST STRING as given in Lemma 4. Considering only the first L columns of the $n \times k$ matrix, we can, in time $O(kL)$, build a FIFO queue of dirty columns. We update this queue when shifting the window of L consecutive columns under consideration from position p (containing columns p to $p + L - 1$) to position $p + 1$ in time $O(k)$: (1) If column p is dirty, we delete it from the front end of the queue. (2) If the “new” column $p + L$ is dirty, we append it to the back end of the queue. Thus, we can maintain the queue of dirty columns, at each position taking only time $O(k)$. After a one-position-shift in the $n \times k$ matrix, Algorithm D is invoked on the columns in the queue only if the queue contains less than kd columns:

Theorem 2 d -MISMATCH is solvable in time $O(kL + (n - L)kd \cdot d^d)$.

Solving DISTINGUISHING STRING SELECTION (DSS). In this problem, we are given “good” strings $s_1, \dots, s_{k_1}$, “bad” strings $s'_1, \dots, s'_{k_2}$, and positive integers d_1, d_2 . We ask for an s “close” to the good strings, i.e., $\max_{i=1, \dots, k_1} d_H(s, s_i) \leq d_1$, and “far away” from the bad ones, i.e., $\min_{j=1, \dots, k_2} d_H(s, s'_j) \geq L - d_2$.

Lemma 6 Given two sets of strings $S_1 = \{s_1, \dots, s_{k_1}\}$ and $S_2 = \{s'_1, \dots, s'_{k_2}\}$ and positive integers d_1 and d_2 . If there are $i \in \{1, \dots, k_1\}$ and $j \in \{1, \dots, k_2\}$ with $d_H(s_i, s'_j) < L - (d_1 + d_2)$, then there is no string s satisfying both $\max_{i=1, \dots, k_1} d_H(s, s_i) \leq d_1$ and $\min_{j=1, \dots, k_2} d_H(s, s'_j) \geq L - d_2$.

In what follows, we describe how to modify Algorithm D in order to solve DSS. Using Lemma 6, we can detect instances that cannot have a solution, i.e., instances where a good and a bad string have Hamming distance less than $L - (d_1 + d_2)$. For this reason, we can extend instruction (D1) in Algorithm D

by returning not only when $d_H(s, s_i) > d_1 + \Delta d_1$ for the candidate s and a good string s_i , but also when $d_H(s, s'_j) < L - (d_2 + \Delta d_1)$ for a bad string s'_j .

Of course, a solution in instruction (D2) is now found when the new goal is met, i.e., $\max_{i=1, \dots, k_1} d_H(s, s_i) \leq d_1$ and $\min_{j=1, \dots, k_2} d_H(s, s'_j) \geq L - d_2$.

Also instruction (D3) has to be extended. As long as the branching shown in (D3) applies, we still use it: If there is a good string s_i which our candidate s is too far away from, i.e., $d_H(s, s_i) > d_1$, we branch on $d_1 + 1$ many positions in which s and s_i differ.

When the candidate s satisfies $d_H(s, s_i) \leq d_1$ for all $i = 1, \dots, k_1$, but is too close to one of the bad strings s'_j , i.e., $d_H(s, s'_j) < L - d_2$, we introduce a new branching. We have to increase $d_H(s, s'_j)$ by changing in s a position p with $s[p] = s'_j[p]$. Since a solution s^* can have at most d_2 many positions p with $s^*[p] = s_j[p]$, it is sufficient to branch on $d_2 + 1$ positions with $s[p] = s'_j[p]$. We do, however, not know to which character $s[p]$ should be set. Trying all characters in this situation gives us an upper bound of $(d_2 + 1) \cdot |\Sigma|$ for the subcases to branch into.

Theorem 3 DSS is solvable in time $O((k_1 + k_2)L \cdot \max(d_1 + 1, (d_2 + 1)|\Sigma|)^{d_1})$.

4 Efficient Linear Time Solution for $k = 3$

For a constant number k of strings, CLOSEST STRING is solvable in linear time with the following argument. The number of column types for k strings depends only on k (namely, it is given by the Bell number $B(k) \leq k!$). Using the column types, CLOSEST STRING can be formulated as an integer linear program (ILP) having only $B(k) \cdot (k - 1)$ variables. Since ILPs with a constant number of variables can be solved in linear time [5,7,8], this is also true for CLOSEST STRING with constant k . The algorithms, however, lead to huge running times, even for moderate number of variables. For this reason, we present a direct (not using linear programming) and efficient linear time algorithm that solves CLOSEST STRING for $k = 3$. We start with transforming the instance into a normalized one and splitting it into “blocks.” We obtain a block by reordering (cf. Lemma 1) the columns of the $k \times L$ matrix and considering consecutive columns in the reordered instance as a block. By sorting, the columns are already ordered in the sequence in which we will process them:

- (0) “Identity Case.” All columns of type $(a, a, a)^t$.
- (1) “Diagonal Case.” All blocks of type $(baa, aba, aab)^t$.
- (2) “3/2 Letters Case.” All blocks of type $(aa, ba, cb)^t$, $(aa, bb, ca)^t$, or $(ab, ba, ca)^t$ (the order of these three types among each other does not matter).
- (3) “2/2 Letters Case.” All blocks of type $(aa, ab, ba)^t$, $(ab, aa, ba)^t$, or $(ab, ba, aa)^t$ (it will be shown in Lemma 9 that we can find only one of these three possibilities, since, otherwise, we would have been able to build an additional block of type (1)).
- (4) “Remaining 2 Letters Case.” All blocks of type $(a, a, b)^t$, $(a, b, a)^t$, or $(b, a, a)^t$ (as in case (3), we can find only one of these possibilities, since, otherwise, we would have been able to build an additional block in (3)).

Algorithm 3-StringsInput: Strings s_1, s_2, s_3 .Output: $CS3(s_1, s_2, s_3)$, which is an optimal closest string for s_1, s_2, s_3 .

- (K0) Given $\begin{bmatrix} a \cdot s'_1 \\ a \cdot s'_2 \\ a \cdot s'_3 \end{bmatrix}$, then return $a \cdot CS3(s'_1, s'_2, s'_3)$. “Identity Case”
- (K1) Given $\begin{bmatrix} baa \cdot s'_1 \\ aba \cdot s'_2 \\ aab \cdot s'_3 \end{bmatrix}$, then return $aaa \cdot CS3(s'_1, s'_2, s'_3)$. “Diagonal Case”
- (K2) Given $\begin{bmatrix} aa \cdot s'_1 \\ ba \cdot s'_2 \\ cb \cdot s'_3 \end{bmatrix}$ ($\begin{bmatrix} aa \cdot s'_1 \\ bb \cdot s'_2 \\ ca \cdot s'_3 \end{bmatrix}$ or $\begin{bmatrix} ab \cdot s'_1 \\ ba \cdot s'_2 \\ ca \cdot s'_3 \end{bmatrix}$, resp.), “3/2 Letters Case”
then return $ca \cdot CS3(s'_1, s'_2, s'_3)$ ($ba \cdot CS3(s'_1, s'_2, s'_3)$ or $aa \cdot CS3(s'_1, s'_2, s'_3)$, resp.).
- (K3) Given $\begin{bmatrix} aa \cdot s'_1 \\ ab \cdot s'_2 \\ ba \cdot s'_3 \end{bmatrix}$, $\begin{bmatrix} ab \cdot s'_1 \\ aa \cdot s'_2 \\ ba \cdot s'_3 \end{bmatrix}$ or $\begin{bmatrix} ab \cdot s'_1 \\ ba \cdot s'_2 \\ aa \cdot s'_3 \end{bmatrix}$, “2/2 Letters Case”
then return $aa \cdot CS3(s'_1, s'_2, s'_3)$.
- (K4) Given $\begin{bmatrix} a^l \\ a^l \\ b^l \end{bmatrix}$, $\begin{bmatrix} a^l \\ b^l \\ a^l \end{bmatrix}$, or $\begin{bmatrix} b^l \\ a^l \\ a^l \end{bmatrix}$ for some integer l , “Remaining 2 Letters Case”
then return $a^{\lceil l/2 \rceil} b^{\lfloor l/2 \rfloor}$.
- (K3') Given $\begin{bmatrix} aaa \cdot s'_1 \\ bbb \cdot s'_2 \\ ccc \cdot s'_3 \end{bmatrix}$, then return $abc \cdot CS3(s'_1, s'_2, s'_3)$. “3 × 3 Letters Case”
- (K4') Given $\begin{bmatrix} aa \\ bb \\ cc \end{bmatrix}$ (or $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$, resp.), “Remaining 3 Letters Case”
then return ab (or a , resp.).

Fig. 2. Algorithm 3-Strings solving CLOSEST STRING for $k = 3$ (3') “3×3 Letters Case.” All blocks of type $(aaa, bbb, ccc)^t$.(4') “Remaining 3 Letters Case.” All blocks of type $(a, b, c)^t$.

Thus, in a natural way, we obtain various *block types*. We can make sure that after this reordering process no columns are left. An instance in which the columns are ordered as explained, we call *ordered* instance. Transforming an arbitrary instance into an ordered instance can be done in linear time.

Algorithm 3-Strings shown in Fig. 2 considers the single blocks of a normalized and ordered instance, one after the other, and combines their solutions to a solution for the whole problem instance. Later in this section, we will show that the algorithm, in linear time, finds an optimal solution.

Lemma 7 *Let s be an optimal median string for s_1, s_2 , and s_3 and let $3 \cdot \max_{i=1,2,3} d_H(s, s_i) - \sum_{i=1,2,3} d_H(s, s_i) \leq 2$ (in particular, this is true when*

$d_H(s, s_1) = d_H(s, s_2) = d_H(s, s_3)$). Then s is an optimal closest string for s_1, s_2 , and s_3 .

Let $(Ki)^*$ denote that instruction (Ki) , $i \in \{0, 1, 2, 3, 4, 3', 4'\}$, is applied an arbitrary number of times (including zero).

Lemma 8 *Given a normalized and ordered CLOSEST STRING instance, then the only possible successions in the application of instructions in Algorithm 3-Strings are $(K0)^*(K1)^*(K2)^*(K3)^*(K4)^*$ and $(K0)^*(K1)^*(K2)^*(K3')^*(K4')^*$.*

Lemma 9 *Given a normalized and ordered CLOSEST STRING instance, we have as type (3) blocks only blocks $(aa, ab, ba)^t$, only blocks $(ab, aa, ba)^t$, or only blocks $(ab, ba, aa)^t$.*

Lemma 10 *Let s_1, s_2 , and s_3 be normalized, and let s be an additional string.*

(a) *Let for every column in $(s_1, s_2, s_3)^t$, the respective letter in s be a majority vote and let s_1, s_2 , and s_3 contain no column $(a, b, c)^t$ such that the respective letter in s is a . Further, let $d_H(s, s_1) \leq d_H(s, s_2) = d_H(s, s_3)$. Then s is an optimal closest string for s_1, s_2 , and s_3 .*

(b) *Let, for every column in $(s_1, s_2, s_3)^t$ that is not $(a, a, b)^t$, the respective letter in s be a majority vote, and let for every column $(a, b, c)^t$ the respective letter in s be c . Further, let $d_H(s, s_1) \leq d_H(s, s_2)$ and either $d_H(s, s_2) = d_H(s, s_3)$ or $d_H(s, s_2) = d_H(s, s_3) - 1$. Then, s is an optimal closest string for s_1, s_2 , and s_3 .*

Theorem 4 *CLOSEST STRING for $k = 3$ can be solved in linear time.*

Proof. (Sketch) **Running time.** Algorithm 3-Strings makes at most L recursive calls and each call takes only constant time, yielding linear running time.

Correctness. From Lemma 8, we know that the order of instructions is $(K0)^*(K1)^*(K2)^*(K3)^*(K4)^*$ or $(K0)^*(K1)^*(K2)^*(K3')^*(K4')^*$.

Now, the proof is given by considering the instructions separately. We assume a CLOSEST STRING instance $(s'_1 s''_1, s'_2 s''_2, s'_3 s''_3)$ with $|s'_1| = |s'_2| = |s'_3|$, such that s'_1, s'_2 , and s'_3 are those parts of the strings such that $(K0)$, $(K1)$, and $(K2)$ apply to them and produce s' . Then s''_1, s''_2 , and s''_3 are processed either by $(K3)$ and $(K4)$, or by $(K3')$ and $(K4')$, resulting in s'' . We first show that s' is an optimal closest string for s'_1, s'_2 , and s'_3 , and then show that $s = s' s''$ is an optimal closest string for the whole instance.

Instructions $(K0)$, $(K1)$, and $(K2)$: These instructions are applied to blocks of type (0), (1), and (2). We can easily check that they produce s' with $d_H(s', s'_1) = d_H(s', s'_2) = d_H(s', s'_3)$. Since we choose in every column the letter as majority vote, s' is an optimal median string for s'_1, s'_2 , and s'_3 . By Lemma 7 we conclude that s' is an optimal closest string for s'_1, s'_2 , and s'_3 .

Instructions $(K3)$ and $(K4)$: Following Lemma 9, we know that $(K3)$ is applied only to one of the three cases mentioned in the instruction. W.l.o.g., we assume that it is only applied on blocks $(aa, ab, ba)^t$. Let l be the total number of the applications of $(K3)$. Thus, $(K3)$ adds a^{2l} to the closest string constructed by

(K0) to (K2), resulting in string $\hat{s}$. If $\hat{s}_1$, $\hat{s}_2$, and $\hat{s}_3$ are the strings processed up to this point, we have $d_H(\hat{s}, \hat{s}_1) + l = d_H(\hat{s}, \hat{s}_2) = d_H(\hat{s}, \hat{s}_3)$ and all letters in (K3) are chosen as majority vote.

To satisfy the premises of Lemma 10(a), it remains to show that there is no column $(a, b, c)^t$ in s_1 , s_2 , and s_3 for which we set the respective position in s to a : For column $(a, b, c)^t$, we only set a in instruction (K2) if it occurs together with column $(b, a, a)^t$. In this case, however, we could form a block of type (1), since we have columns $(a, b, a)^t$ and $(a, a, b)^t$, necessary for (K3). This contradicts the assumption that the instance is ordered. It follows that no column $(a, b, c)^t$ is assigned a . Thus, we conclude with Lemma 10(a) that $\hat{s}$ is an optimal closest string for $\hat{s}_1$, $\hat{s}_2$, and $\hat{s}_3$. For instruction (K4) the correctness is shown, by use of Lemma 10(b), in a very similar way as for (K3). We omit the details here.

Instructions (K3') and (K4'). We are only left with columns of type $(a, b, c)^t$. Instruction (K3') applies as long as the number of these columns is larger than three. What remains are one or two columns of type $(a, b, c)^t$, which (K4') takes care of. Given input strings s_1 , s_2 , and s_3 , they are processed after the application of (K4'). We can check that now either $d_H(s, s_1) = d_H(s, s_2) - 1 = d_H(s, s_3) - 1$ or $d_H(s, s_1) = d_H(s, s_2) = d_H(s, s_3) - 1$. Since for all columns we chose the letter in s as majority vote, s is an optimal median string for s_1 , s_2 , and s_3 . By Lemma 7, we conclude that s is an optimal closest string. $\square$

5 Experimental Results

We implemented Algorithm D using the programming language C, including the heuristic improvements discussed in Subsections 3.2 and 3.3. We performed tests on a LINUX PC with 750 MHz processor and 192 MB main memory. First, we report about tests on random instances with $|\Sigma| = 4$ where we scan the whole search tree, i.e., we do not stop when the first solution is found. The displayed results are average results taken from a range of ten such random data sets.

Length/mismatch ratio. For instances containing only dirty columns, our experiments with randomly generated data show that not only the number d of mismatches allowed but, moreover, the ratio of string length L to d has a major impact on the difficulty of solving the problem. The results from Fig. 3(a) show that an increasing length L and a thereby increasing L/d ratio for a fixed value of d will significantly decrease the running time of the algorithm up to some point. When considering the values of d for which we can process CLOSEST STRING instances in practice, we have to take this ratio into account. E.g., for a “hard” ratio of 2, i.e., the string length is twice the number of mismatches, we solve instances with $d = 15$ ($L = 30$, $k = 50$) in about 200 sec, and for an “easier” ratio of 3 we can solve instances with $d = 20$ ($L = 60$, $k = 50$) in 100 sec.

Number of input strings. When considering the running times for an increasing number k of input strings (for fixed values of L , d), we encounter two competing factors. On the one hand, an increase in the number of strings means an increase

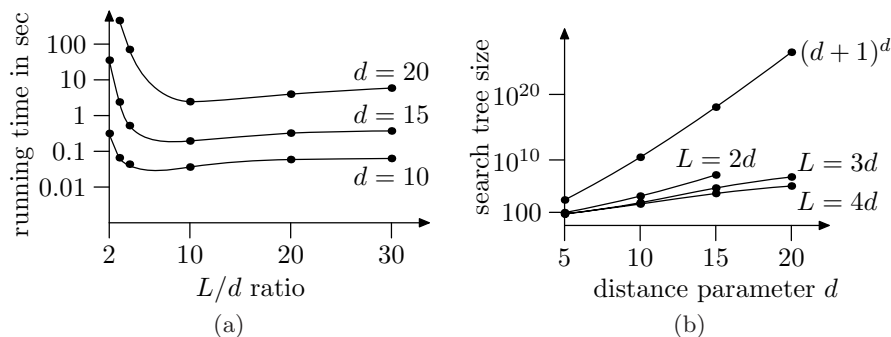


Fig. 3. (a) Comparing, on a logarithmic scale, the running time of CLOSEST STRING instances for differing length/mismatch ratio L/d ($k = 25$). (b) Comparing, also on a logarithmic scale, search tree sizes to the theoretical upper bound of $(d+1)^d$. Each line displays results for one fixed L/d ratio ($k = 25$)

in the linear time to be spent in every node of the search tree. On the other hand, a growing number of strings means a growing number of constraints on the solutions and, therefore, a decreasing size of the search tree. Our experience with random data sets shows a high running time for small numbers of strings, decreasing with growing number of strings up to some turning point. From then on running time increases again, since the linear factor spent in each search tree node becomes crucial. E.g., for $L = 24$, $d = 12$, we need 6.2 sec for $k = 10$ (search tree size 934892), 4.3 sec for $k = 100$ (search tree size 145390), and 8.5 sec for $k = 400$ (search tree size 91879).

Search tree size. In Fig. 3(b), we compare the size of the search tree for given instances with the theoretical upper bound of $(d+1)^d$. We note that the search trees are by far smaller than the bound predicts.

Primer design by solving a combination of d -MISMATCH and DISTINGUISHING STRING SELECTION. We applied our algorithm to compute candidates for primers, a task the biological expert otherwise does by hand. In our application, we are confronted with probes that may contain parasite DNA (mushrooms) as well as host DNA, and the goal is to design primers that exclusively bind to the parasite sequences. The given data in this example are an alignment of length 715 with five sequences of parasite DNA and four sequences of host DNA. We approach the problem by solving DSS with the parasite sequences as set of good strings and host sequences as set of bad strings. The desired length L of primers is between 15 to 20. Since the primers should have as few mismatches as possible, we consider here for d_1 only values ≤ 3 . E.g., with $L = 15$, $d_1 = 2$, the minimum value for which we find a primer candidate is $d_2 = 7$. For $L = 25$, we find a candidate with $d_1 = 2$ and $d_2 = 18$, or with $d_1 = 3$ and $d_2 = 15$. The advantage of the algorithm in this application is that it quickly (all runs

are done in less than a second) finds all positions where primer candidates meet the specified conditions (and also finds if certain values of L , d_1 and d_2 do not allow a solution), whereas the task is tedious for the human expert who might only find obvious candidates.

6 Conclusion

We described new and also practically promising exact algorithms for consensus word problems motivated by computational biology. In particular, all our algorithms for these, in general NP -complete, problems work in *linear* time for constant parameter values. This is of particular importance in signal finding and related applications where, e.g., small distance parameter d is normal (for instance, in primer design d -values around 5 are not unusual [3]). Our results improve and generalize previous work and answer some open questions [3,11]. It seems hard to extend our results to the more general CLOSEST SUBSTRING problem, which is more relevant in biological applications dealing with unaligned sequences.

Acknowledgment

We thank Michael Weiß from the Biology Department (group for Systematic Biology and Mycology, Universität Tübingen) for his suggestions and for providing us with biological data. We are grateful to Gerhard J. Woeginger (Twente) for pointing us to the ILP result of Lenstra.

References

1. J. Alber, J. Gramm, and R. Niedermeier. Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229(1-3):3–27, 2001. 442, 443, 444
2. R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999. 442, 443, 444
3. P. A. Evans and H. T. Wareham. Practical non-polynomial time algorithms for designing universal DNA oligonucleotides: a systematic approach. Manuscript, April 2001. 442, 443, 452
4. M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30:113–119, 1997. 441
5. R. Kannan. Minkowski’s convex body theorem and integer programming. *Mathematics of Operations Research*, 12:415–440, 1987. 447
6. J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proc. of 10th ACM-SIAM SODA*, pages 633–642, 1999, ACM Press. To appear in *Information and Computation*. 441
7. J. C. Lagarias. Point lattices. In R. L. Graham *et al.* (eds.) *Handbook of Combinatorics*, pages 919–966. MIT Press, 1995. 447
8. H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983. 447

9. M. Li, B. Ma, and L. Wang. Finding similar regions in many strings. In *Proc. of 31st ACM STOC*, pages 473–482, 1999. ACM Press. 442
10. P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, 2000. 441
11. N. Stojanovic, P. Berman, D. Gumucio, R. Hardison, and W. Miller. A linear-time algorithm for the 1-mismatch problem. In *Proc. of 5th WADS*, number 1272 in LNCS, pages 126–135, 1997, Springer. 441, 442, 452
12. N. Stojanovic, L. Florea, C. Riemer, D. Gumucio, J. Slightom, M. Goodman, W. Miller, and R. Hardison. Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, 27(19):3899–3910, 1999. 441

Topological Peeling and Implementation [★]

Danny Z. Chen¹, Shuang Luan¹, and Jinhui Xu^{2★★}

¹ Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, IN 46556, USA
{chen,sluan}@cse.nd.edu

² Department of Computer Science and Engineering, State University of New York
at Buffalo
201 Bell Hall Box 602000, Buffalo, NY, 14260, USA
jinhui@cse.buffalo.edu

Abstract. We present a new approach, called *topological peeling*, and its implementation for traversing a portion A_R of the arrangement formed by n lines within a convex region R on the plane. Topological peeling visits the cells of A_R in a fashion of propagating a “wave” of a special shape (called a *double-wriggle curve*) starting at a single source point. This special traversal fashion enables us to solve several problems (e.g., computing shortest paths) on planar arrangements to which previously best known arrangement traversal techniques such as topological sweep and topological walk may not be directly applicable. Our topological peeling algorithm takes $O(K + n \log(n + r))$ time and $O(n + r)$ space, where K is the number of cells in A_R and r is the number of boundary vertices of R . Comparing with topological walk, topological peeling uses a simpler and more efficient way to sweep different types of lines, and relies heavily on exploring small local structures, rather than a much larger global structure. Experiments show that, on average, topological peeling outperforms topological walk by 10 – 15% in execution time.

1 Introduction

Computing or traversing the arrangement of a set of lines on the plane is a fundamental problem in computational geometry [9]. Extensive research has been conducted on solving this problem, and several efficient techniques have been developed [3,4,6,9,10,11,13,21]. In [10], Edelsbrunner and Guibas presented a powerful *topological sweep* algorithm for traversing the whole arrangement of n planar lines within an optimal $O(n^2)$ time and $O(n)$ space. With this technique, a set of problems can be solved in a space (or even time) efficient manner [1,10,12,19]. To extend this technique to portion of an arrangement, Asano, Guibas, and

[★] This research was supported in part by NSF under Grants CCR-9623585 and CCR-9988468.

^{★★} Corresponding author. The research of this author was supported in part by a faculty start-up fund from the CSE department, SUNY Buffalo and an IBM faculty partnership award.

Tokuyama [3] invented an interesting input-sensitive *topological walk* algorithm. This algorithm sweeps the portion A_R of a planar arrangement A inside a convex region R in $O(K + n \log(n + r))$ time and $O(n + r)$ space, where K is the number of cells of A_R and r is the number of boundary vertices of R .

In a certain way, topological walk could be viewed as if a “wave” is propagated through the cells of arrangement A from one or more source points. The shape of such a “wave-front” curve of topological walk, however, may vary significantly from problem to problem. Despite many advantages offered by topological walk, it seems very difficult to exploit the behavior of this wave propagation for solving certain problems efficiently. For example, it is not clear to us how to use topological sweep and topological walk to compute single-source shortest paths on the arrangement (i.e., the paths lying on the lines of A) without substantial backtracking. Such a difficulty to topological walk is due to the following fact: Topological walk distinguishes two types of lines: *upper* (e.g., the lines whose left intersections with the boundary $B(R)$ of R lie on the upper boundary) and *lower* lines, and sweeps them differently (e.g., upper lines are treated as waiting lines, and lower lines as normal lines). Such an asymmetry causes topological walk to sweep in a “nice” wave propagation fashion, starting at the leftmost vertex of A_R , only for the arrangement of *lower lines*. For the arrangement of *upper lines*, the sweeping wave is actually propagated from different wave “sources”.

In this paper, we present a novel algorithm, called *topological peeling*, and its implementation for traversing a planar arrangement A of n lines. Our approach sweeps lines in a symmetric and more efficient way. A key observation we use is: The wave propagation of topological walk is well-behaved (e.g., of a *wriggle* shape) if it is applied to only one type of lines (i.e., either lower or upper lines, but not both). Our idea hence is to reduce the problem to two special cases: Topological walk on each of the two types of lines. However, to do that efficiently, we must overcome quite a few difficulties. The reason is that the interference (and the coordination) between these two types of sweepings can be very complicated. Furthermore, sweeping in this fashion violates some key properties used by topological walk, thus seemingly running a risk of raising the time bound. Interestingly, by exploiting a number of new geometric observations and techniques, as well as the special properties of our wave-front curves, we are able to partition the traversal task into a sequence of mutually interweaving sweepings on the two types of lines. Our algorithm retains the same time and space bounds as topological walk. In consequence, our wave-front curve is well under control (i.e., of a *double-wriggle* shape and starting at a single source point), and enables us to solve several problems which topological sweep and topological walk did not solve efficiently. Moreover, our sweeping exploits the local structures in a more efficient way. Unlike topological walk whose sweeping is always performed on a large-sized global structure, computation in our sweeping heavily relies on local structures with much smaller sizes. Our experiments show that, on average, topological peeling outperforms topological walk by 10 – 15% in execution time.

As a result, the double-wriggle shaped wave-front of our topological peeling allows us to form a special structure $AR(v)$, called *anchor region*, for each ver-

tex v of A_R , which is a convex subregion of R whose boundary contains both v and the wave source. Our peeling algorithm always maintains an important property that when a vertex v is visited, all cells of A_R inside $AR(v)$ have already been swept. Combining with several other interesting observations, this property leads to an efficient solution for the shortest path problem on arrangements. The previously best known result for computing a shortest path in a planar arrangement takes $O(n^2)$ time and space [5,14,15] (by reducing the problem to one on a planar graph [17]), and it has been an open problem to improve these bounds. Our algorithm takes $O(n^2)$ time and $O(n)$ space to report single-source shortest path lengths in A . An actual shortest path between two points of A can also be obtained in $O(n^2)$ time and $O(n)$ space, by using a technique by Chen *et al.* [8] for reporting actual shortest paths (without maintaining a single-source shortest path tree of size $O(n^2)$ on A).

Topological walk uses a *weak representation* for its cells (i.e., a cell is represented by only a subset of its boundary edges). Such a representation does not support cell-reporting directly (i.e., reporting the boundary edges). Much effort is needed to enforce such a functionality. In contrast, our topological peeling adopts a *strong representation* for the cells, which gives the benefit of being able to explicitly extract the entire boundary of each encountered cell all at once. Such boundary information is useful in solving a number of problems on arrangements. In [20], Nievergelt and Preparata showed that for many CAD problems, it is desired to list the boundary edges of each cell in a cyclic order (but their plane-sweeping algorithms for listing cell boundaries in this way introduce an additional $O(\log n)$ factor to the time bound). In [7], Chen *et al.* showed that the solutions to several geometric optimization problems [7,2,18] are to decompose such a problem into a set of instances of a certain special non-linear optimization problem, with each cell of an arrangement associating with one such problem instance. Since the boundary of each cell defines the domain of the corresponding problem instance, a weak representation along would not provide sufficient information. By combining topological peeling with the techniques in [2,7,18], such geometric optimization problems can be solved in a space-efficient manner.

Our topological peeling is also useful to solving many other problems. In [3,4,8,10], there are a number of problems on planar arrangements which are solvable by either topological sweep or topological walk, or both. Almost all of these are also solvable by topological peeling. Below is a list of problems solvable by topological peeling, with the best known complexity bounds as those in [3,4,8,10]: Computing a longest monotone path, a longest monotone concave path, a largest convex subset, a largest empty convex subset, a maximal stabbing line, the visibility graph for non-intersecting line segments, minimum-area triangles, and a maximum-weight cut line of a straight line graph. It is likely that topological peeling will be applicable to more problems.

Due to the space limit, we omit many details and proofs of lemmas from this extended abstract.

2 Cuts and Horizon Trees

Let H be a set of n planar lines, and $A(H)$ be their arrangement. Let R be a polygonal convex region, and A_R denote the portion of $A(H)$ inside R (see Figure 1(a)). The boundary $B(R)$ of R has r vertices. Starting at the left most vertex s of $B(R)$, the clockwise-ordered vertices along $B(R)$ form a chain, and is called the *stem* [3]. Without loss of generality (WLOG), we assume that in H no three lines share a common point, and each line is non-vertical and intersects $B(R)$ twice (the degeneracy case can be handled in the same way as that used by topological sweep). The two intersections of a line $l \in H$ with $B(R)$ are denoted by v_l^l and v_l^r , with v_l^l to the left of v_l^r (i.e., $x(v_l^l) < x(v_l^r)$).

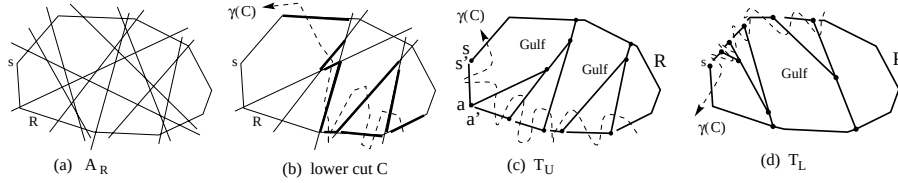


Fig. 1. Arrangement, cut, gulf, and upper and lower horizon trees

By cutting $B(R)$ at its leftmost and rightmost vertices, we obtain a lower (convex) chain and upper (concave) chain, denoted by $L(R)$ and $U(R)$, respectively. For a line $l \in H$, if v_l^l is on $L(R)$ (resp., $U(R)$), then l is called a *lower* (resp., *upper*) *line*. Further, for a lower (resp., upper) line l , if v_l^r also lies on $L(R)$ (resp., $U(R)$), then l is called a *weak lower* (resp., *upper*) *line*. H hence is partitioned into two subsets H_L and H_U containing the lower lines and upper lines, respectively. The arrangement of H_L (resp., H_U) inside R is denoted as A_R^L (resp., A_R^U). Obviously, A_R can be viewed as the overlap of A_R^L and A_R^U .

For two edges e_i and e_j of A_R^L (resp., A_R^U), we say e_i dominates e_j if a cell c of A_R^L is incident to both e_i and e_j , such that e_i is above (resp., below) c and e_j is below (resp., above) c .

Definition 1. A cut of $A_R^L \cup B(R)$ (resp., $A_R^U \cup B(R)$), called a lower (resp., upper) cut (see Figure 1(b)), is a list of open edges $C = (e_0, e_1, \dots, e_m)$ satisfying the following two conditions:

1. $e_0 \in U(R)$, $e_m \in L(R)$ (resp., $e_0 \in L(R)$, $e_m \in U(R)$) and $e_i \in A_R^L \cup L(R)$ (resp., $e_i \in A_R^U \cup U(R)$) for $1 \leq i \leq m-1$.
2. One of the following conditions holds for $i = 0, 1, \dots, m-1$: (a) e_i dominates e_{i+1} ; (b) e_{i+1} is an edge of A_R^L (resp., A_R^U) such that the left endpoint p_{i+1} of e_{i+1} is on $L(R)$ (resp., $U(R)$), and e_i is the leftmost edge of $L(R)$ (resp., $U(R)$) lying to the left of p_{i+1} such that there is no other lower (resp., upper) line l whose v_l^l lies on $L(R)$ (resp., $U(R)$) between e_i and p_{i+1} .

As shown in [3], there exist initial cuts in both A_R^L and A_R^U ; also, for each cut C of A_R^L (resp., A_R^U), there is a *pseudo-sweepline* $\gamma(C)$ which intersects all edges of C but no other edges of $A_R^L \cup B(R)$ (resp., $A_R^U \cup B(R)$).

For a lower cut C , we define an *upper horizon tree* $T_U(C)$ (see Figure 1 (c)), which is a simpler version of that of topological walk [3]. Similarly, for an upper cut C , we define a *lower horizon tree* $T_L(C)$ (see Figure 1 (d)). The initial upper (resp., lower) horizon tree partitions the region R into $O(|H_L|)$ (resp., $O(|H_U|)$) connected components which are called *lower* (resp., *upper*) *gulfs*.

A horizon tree $T_U(C)$ can be viewed as a graph embedded on the plane whose nodes are the branching points of the tree and the left endpoints of the cut edges. An edge (called *t*-edge) of this graph represents either a segment of a line in H or a subchain of $B(R)$. If we treat the graph as a tree rooted at the left endpoint of e_0 , then we can define *internal node*, *leaf branch node* (i.e., an internal node adjacent to at most one other internal node), and *twig* (i.e., a node at which two cut edges meet) in the same way as in [3]. For any vertex v of $T_U(C)$, we let $path(v)$ denote the convex path of $T_U(C)$ from v to the root of $T_U(C)$. Given a polygonal path P , an *l*-edge on P is a maximal segment contained in a straight line. Note that an *l*-edge in T_U may consist of multiple *t*-edges.

Given a lower cut $C = (e_0, e_1, \dots, e_m)$ and the corresponding upper horizon tree $T_U(C)$, a *bay bay* (e_i) of a cut edge $e_i, i = 1, 2, \dots, m$, is the subpath of $T_U(C)$ from the left endpoint of e_i to the lowest common ancestor of the left endpoints of e_i and e_{i-1} in $T_U(C)$.

3 Main Ideas

One of our goals is to achieve a well-behaved sweeping wave with a “nice” wave-front curve and a single wave source for traversing the arrangement A_R . For this purpose, we make the following two observations: (1) The asymmetric sweeping of the upper and lower lines causes the sweeping wave of topological walk to behave irregularly. (2) If there were only *one* kind of lines (e.g., the lower lines), Lemma 2 shows that topological walk would traverse A_R with a wave-front curve of a *wriggle* shape. Hence, it seems reasonable to try to reduce the problem to two special cases: Topological walk on each of the two types of lines. More specifically, we would like to use both the global upper and lower horizon trees, T_U and T_L , to sweep A_R^L and A_R^U , respectively. Of course, A_R is the overlapping of A_R^L and A_R^U . However, to sweep A_R efficiently in this manner is actually quite difficult. The reason is that the upper and lower horizon trees are independent of each other, and hence their sweeping loci are very much unrelated to one another. Fortunately, by exploiting a number of new geometric observations and techniques, we are able to partition the traversing of A_R into a sequence of interweaving sweepings, each based on T_U or T_L . The key is a judicious interweaving of the sweepings of the two horizon trees so that the overlapping of A_R^L and A_R^U is produced efficiently.

One difficulty is that the two horizon trees T_U and T_L can intersect with each other at $O(n^2)$ places, and our interweaving of these two kinds of sweepings

must avoid using super-linear space. Our idea for this interweaving is to use the sweeping of the upper horizon tree to guide the sweeping of the lower horizon tree. More specifically, when the sweeping of T_U produces a cell G of A_R^L (which we call the *current cell*), we sweep the part $A_R \cap G$ of A_R by making use of T_L .

To carry out the above approach successfully, three steps are performed for each current cell G .

In the first step, the entire boundary $B(G)$ of G is computed. Note that, to sweep $A_R \cap G$, $B(G)$ is needed for constructing the lower horizon tree $T_L(G)$ of all upper lines intersecting G . However, due to its weak representation of the cells, topological walk on A_R^L does not maintain sufficient information for extracting $B(G)$. Thus, additional data structures and proper modification on topological walk are needed.

The second step builds the local lower horizon tree $T_L(G)$. In this step, a complete new method is needed to efficiently construct $T_L(G)$. This is because that topological walk builds an upper horizon tree for the region R in $O(n \log(n+r))$ time, if a similar method were used straightforwardly to build $T_L(G)$ on each current cell G of A_R^L , it would result in an $O(n^3)$ -time solution. Based on several nontrivial geometric observations and an interesting lock-step search, we construct $T_L(G)$ in $O(|A_R \cap G|)$ time by cutting a dynamically maintained global tree T_L of all upper lines in H_U .

In the third step, the portion of arrangement A_R inside G is swept and the global tree T_L is dynamically maintained. The difficulty here is that if topological walk were used to sweep $A_R \cap G$ on $T_L(G)$, the total time (over all cells) could be $O(K \log n)$. To avoid this pitfall, we simulate topological walk on the local tree $T_L(G)$, but perform the actual sweeping on the global tree T_L . Such a mixed procedure demands an efficient way to coordinate the changes of the two trees (or equivalently an efficient algorithm to determine the intersection points between T_L and $B(G)$). By using a lockstep-like exponential search, we can produce all such points in linear time, and the total time (except the time for sweeping T_L) for sweeping $A_R \cap G$ is only $O(|A_R \cap G|)$.

In consequence, our sweeping proceeds in the way of propagating a special shaped wave from one source point through A_R . The curve bounding all visited cells of A_R (resp., A_R^L) is called the *wave-front* (resp., *lower wave-front*), and has a *double-wriggle* (resp., *wriggle*) shape. The shapes of the wave-fronts remain invariant topologically during the entire sweeping process. These structures are a key to our algorithm, and enable us to utilize other useful structures (e.g., the anchor regions of the vertices of A_R).

4 The Peeling Algorithm

Our interweaving sweeping performs a topological walk on T_U to compute the cells of A_R^L . A cell G of A_R^L becomes the current cell once all its boundary edges are completely traversed. For each encountered current cell G , a lower horizon tree $T_L(G)$ is constructed for the upper lines intersecting G . The portion of A_R^U on or inside the boundary $B(G)$ of G is then computed by sweeping $T_L(G)$.

Hence, our sweeping algorithm needs to handle the following key subproblems: (P1) how to extract $B(G)$, (P2) how to construct $T_L(G)$, and (P3) how to sweep the portion $A_R^U(G) = A_R \cap G$.

4.1 Extracting $B(G)$

To obtain the boundary $B(G)$ of the current cell G of A_R^L , we maintain a curve $LFront$ to bound the visited cells of A_R^L . Let lwf be the lower wave-front bounding all visited A_R^L cells. Suppose lwf is a clockwise oriented curve starting at s (i.e., the leftmost vertex of R). Then $LFront$ is the portion of lwf excluding the stem edges from both ends of lwf (i.e., the portion of lwf between the first and last endpoints of all non-stem edges on lwf). We have the following lemma on $LFront$ and $B(G)$.

Lemma 1. *Each current cell G is bounded by a concave subpath $B_{left}(G)$ of $LFront$ and a convex subpath $B_{right}(G)$ of the c -to-root path $path(c)$ in T_U , joining at a vertex of $B(G)$, where c is on $B(G)$ and is the currently encountered leaf vertex of T_U .*

To use Lemma 1 to extract $B(G)$ efficiently, we maintain pointers for $LFront$ and T_U appropriately, so that at each vertex in $T_U \cap LFront$, $LFront$ and T_U are reachable from each other in $O(1)$ time. Then for each current cell G , the time for extracting $B(G)$ and updating $LFront$ is $O(|B(G)|)$.

The next lemma shows some properties of the curve $LFront$, including its *wriggle* shape, which is important to our algorithm.

Lemma 2. *Let L_{LF} be the maximal concave subpath of $LFront$ starting from its leftmost segment, and $R_{LF} = LFront - L_{LF}$. Then, at any moment of the sweeping, R_{LF} is a convex path (R_{LF} can be $\emptyset$). Furthermore, the rightmost l -edge of L_{LF} is incident to the next current cell of A_R^L .*

4.2 Constructing $T_L(G)$ Using $B(G)$

To obtain the lower horizon tree $T_L(G)$, we need to first identify the set $H_U(G)$ of the upper lines intersecting G , and then construct $T_L(G)$. The way we do this is to use $B(G)$ to cut T_L (see Figure 2(a)), the lower horizon tree for H_U . This cutting of T_L results in a forest of lower horizon subtrees. We then connect all the intersections of T_L and $B(G)$ clockwise along $B(G)$, in a way similar to that used in topological walk to construct an upper horizon tree. This results in a tree $T'_L(G)$ rooted at the leftmost vertex of $B(G)$. It can be shown (by Lemma 3, 4, 5, 6 and 11) that $T'_L(G)$ indeed is the lower horizon tree for all upper lines in $H_U(G)$ (i.e., $T'_L(G) = T_L(G)$) for every current cell G .

Our purpose is to use $T_L(G)$ to sweep the cells of A_R inside G in a topological walk manner. In order to do that, we need to resolve three issues: (I1) In $T'_L(G)$, have we included all upper lines of H_U intersecting G (since an upper line could be stopped by another upper line in T_L before reaching $B(G)$)? (I2) For every

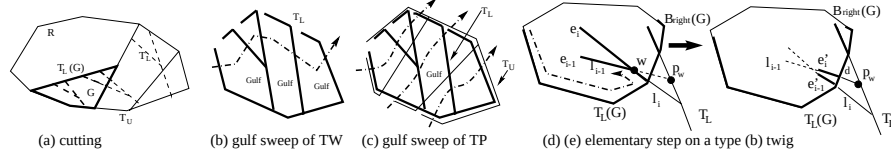


Fig. 2. Cutting, gulf sweep, and elementary step

upper line l of H_U intersecting G , is it true that the left intersection of l with $B(G)$ is on the upper boundary $B_U(G)$ of G (otherwise, we have to deal with waiting lines as in [3] and thus cannot guarantee the shape of the wave-front inside G)? (I3) How to efficiently compute the cutting of T_L and $B(G)$? The following lemma is useful to resolving these issues.

Lemma 3. *Let G be the current cell and l be any upper line intersecting G . Then the left intersection v of l with $B(G)$ lies on either $B_{left}(G)$ or $B_{right}(G) \cap U(R)$ ($B_{left}(G)$ and $B_{right}(G)$ are defined as in Lemma 1).*

Lemma 3 enables us to handle issue (I1). This is shown in the next lemma.

Lemma 4. *For any current cell G , let T_L be the lower horizon tree of H_U with the portion of A_R^U to the left of $LFront$ being completely swept. Then all upper lines intersecting G can be detected by computing the cutting of $B(G)$ and T_L .*

For issue (I2), let $P = B_{right}(G) - U(R)$, and c and v be the leftmost and rightmost vertices of P , respectively. The lemma below characterizes c and v .

Lemma 5. *Let c and v be defined as above. Then c and v are the leftmost and rightmost vertices of the current cell G , respectively.*

From the above lemmas, we immediately have the following observation.

Lemma 6. *No upper line intersects $B_{left}(G)$ twice.*

Lemmas 4 and 5 show that, if T_L is a properly maintained lower horizon tree at the moment of visiting G and if the subregion of R to the left of $LFront$ is already swept, then cutting T_L by $B(G)$ and connecting the resulted forest of T_L inside G along $B(G)$ yield an initial lower horizon tree for the upper lines of H_U intersecting G . Henceforth, we use $T_L(G)$ to denote the lower horizon tree on G thus generated. With $T_L(G)$, the A_R cells inside G can be swept in a topological walk manner. In order to do that, we need to satisfy two premises: (1) T_L must be properly maintained such that it remains the lower horizon tree of H_U during the entire period of sweeping A_R (Lemma 10 will show that our method for sweeping G maintains the structure of T_L dynamically). (2) we need to compute the cutting of T_L and $B(G)$ efficiently (i.e., resolving issue (I3)).

For computing the cutting of T_L and $B(G)$ we use an interesting “lock-step” procedure for searching intersections of $T_L \cap B(G)$ on the two path P and $Q = B_{left}(G) \cup (B_{right}(G) \cap U(R))$ (the details of this procedure are left to the full paper). The following lemma shows that the cutting can be efficiently computed.

Lemma 7. *All intersections of T_L and $P = B_{right}(G) - U(R)$ can be computed in $O(|T_L(G)|)$ time and $O(n + r)$ space.*

In the above procedure, we assume that the intersections of $Q \cap T_L$ are available when visiting G . To make the procedure applicable, before the sweeping of T_U , we need to compute the intersections of the initial lower horizon tree T_L and $B(R)$. During the sweeping, we maintain the intersections of $T_L \cap T_U$ on L_{Front} and on $B(R)$. When the sweeping on the current cell G is finished, L_{Front} is properly updated, and the intersections of $T_L \cap T_U$ on Q are discarded. Therefore, at any time, we only store the intersections of $T_L \cap T_U$ on L_{Front} and on $B(R)$, and possibly such intersections on P of the current cell G .

4.3 Sweeping the Part of Arrangement inside G

Now consider problem (P3). While using the lower horizon tree $T_L(G)$ to sweep the portion of A_R inside a current cell G in a topological walk manner, the global lower horizon tree T_L needs to be maintained dynamically. Straightforwardly using $T_L(G)$ to walk topologically on $A_R \cap G$ may give rise to two problems. First, topological walk updates the changing information only locally (i.e., on $T_L(G)$) when using $T_L(G)$ to sweep G , and hence may destroy the structure of T_L . This is because some upper lines stopped by other upper lines at vertices inside G need not be extended in T_L when elementary steps are performed on twigs at such vertices. Second, topological walk can take up to $O(K_i + I_i \log(I_i + L_i))$ time to sweep each current cell G_i of A_R^L , where K_i is the size of A_R inside G_i , I_i is the number of intersections of $T_L \cap B(G_i)$, and L_i is the number of vertices of $B(G_i)$. Summing over all A_R^L cells, the time bound could be $O(K_U + I \log(I + L))$, where K_U is the total number of vertices of A_R inside the cells of A_R^L , I is the intersections of A_R^L and A_R^U , and L is the number of all lower lines. This bound can be asymptotically larger than our claimed $O(K + n \log(n + r))$ bound when the intersections of A_R^L and A_R^U are the dominating part of all A_R vertices.

To avoid these pitfalls, we simulate topological walk on $T_L(G)$: Starting at the root of $T_L(G)$, perform a dynamic depth-first search on $T_L(G)$, with right branch preferred, to seek the rightmost twig w (see Figure 2(d)). Suppose the rightmost twig w is found. If w is a twig of T_L (i.e., w is not a twig involving some lower line), then we perform an elementary step on T_L instead of on $T_L(G)$ (i.e., ignore the existence of $B_{right}(G)$) (see Figure 2(e)), and update the information of the upper cut of T_L . This is done by a search that is similar to the mixed search in topological walk [3] on the corresponding bay of T_L . Note that we do not really cut $T_L(G)$ away from T_L . Instead, we use some pointers to represent $T_L(G)$, and the portion of $T_L(G)$ inside G remains attached to T_L .

A sweeping of this fashion on A_R^U is different from topological walk on T_L . In topological walk, it proceeds as a sequence of sweeps on the gulfs of T_L , with each gulf being swept just once, and this property is crucial to its time analysis. However, in our algorithm, since the sweeping of T_L is guided by the sweeping of T_U , a gulf may be peeled multiple times, and each time only a portion of its

arrangement is swept (see Figure 2(b)(c)). Fortunately, we are able to prove (in next subsection) that this does not yield a larger time bound for us.

The elementary steps performed on T_L can produce intersections on the boundary of G , and thus change the structure of $T_L(G)$ (e.g., d in Figure 2(e)). In order to efficiently capture all such intersections and update $T_L(G)$, we first set a flag f_u for each vertex u in $T_L(G)$ to indicate that u is inside G , and then simulate topological walk on $T_L(G)$. The simulation repeatedly finds twigs of $T_L(G)$. We distinguish three types of twigs involving $T_L(G)$, as follows: (a) Twigs on $B_{left}(G) \cap T_L$, (b) twigs in T_L , and (c) twigs created by an edge of T_L and an edge of $B_{right}(G)$.

If w is of type (a), then we just do a normal *elementary step* operation as in [3]. Each such operation takes $O(1)$ time since it only deletes the edge on $B_{left}(G)$ and no edge needs to be extended.

If w is of type (b), then the elementary step is performed on T_L (see Figure 2(d)(e)) instead of on $T_L(G)$. As discussed above, the extension of a segment to the right of w (e.g., e'_i in Figure 2(e)) may intersect $B(G)$, and we need to check the existence of such an intersection (e.g., d in Figure 2(e)) in order to maintain $T_L(G)$ correctly. Note that such an intersection, if exists, can only occur on $B_{right}(G)$. The reason for this is the following. Let the extended segment be on an upper line l_{i-1} . If l_{i-1} enters G from $B_{left}(G)$, then by Lemma 6, l_{i-1} does not intersect $B_{left}(G)$ again. If l_{i-1} enters G from $B_{right}(G) \cap U(R)$, then since $B_{right}(G) \cap U(R)$ is to the right of $B_{left}(G)$ (by Lemma 5), the right extension of l_{i-1} does not intersect $B_{left}(G)$. (Also note that l_{i-1} cannot enter G from $B_{right}(G) - U(R)$ by Lemma 3.) Based on the finger search tree [16], we have the following lemma.

Lemma 8. *For each current cell G , the total cost for checking and computing the intersections of the upper lines in $H_U(G)$ with $B_{right}(G)$ induced by elementary steps performed on the type (b) twigs of $T_L(G)$ is bounded by $O(|A_R \cap G|)$.*

Remark: Instead of using the finger search tree [16], we can also copy the information of $B_{right}(G)$ onto an array, and then use a lockstep-like exponential search to achieve Lemma 8.

For a twig w of type (c), the elementary step operation is slightly different from the normal one. Since such a twig is on $B_{right}(G)$ and the two involved edges e_i and e_j are on T_L and $B_{right}(G)$ respectively, it need not extend either one. Actually, the elementary step operation for twigs of this type only deletes e_i from T_L , and updates the upper cut of T_L (i.e., use the edge e'_i which is on the supporting line l_{e_i} of e_i and shares the common endpoint w with e_i to replace e_i in the upper cut of T_L). For e_j , the elementary step on w does not delete it from T_U (note that this is different from topological walk), since e_j will eventually be deleted by some elementary steps performed on T_U .

The above sweeping algorithm and the cutting procedure in Subsection 4.2 are all based on the assumption that T_L is a properly maintained lower horizon tree during the entire period of sweeping A_R . The following lemmas show that this assumption can indeed be achieved. In these lemmas, we say a vertex u is

swept if the elementary step on u is already performed and the two segments with u as the common right endpoint are deleted from T_L or T_U . We say a vertex u is *visited* if it is already traversed by the depth-first search performed on T_U or $T_L(G)$. We say two vertices u_1 and u_2 on the same line l of H are *swept out of order* if u_1 is to the right of u_2 and u_1 is swept before u_2 .

Lemma 9. *For any current cell G , suppose that T_L has been properly maintained and no out-of-order sweeping occurred before sweeping G . Then all vertices of A_R inside G are generated during the sweeping of G , without any out-of-order sweeping.*

Lemma 10. *Suppose that T_L has been properly maintained and no out-of-order sweeping occurred before sweeping G . Then when the sweeping on G is finished, T_L remains a global lower horizon tree.*

By applying Lemmas 9 and 10 to each current cell, we ensure the lower horizon tree structure of T_L is always maintained and each upper line is swept from left to right. Since each lower line is swept by topological walk, it is certainly swept from left to right. Thus we have the following lemma.

Lemma 11. *Topological peeling maintains the lower horizon tree structure of T_L in the entire period of sweeping A_R . Furthermore, each line of H is swept from left to right.*

After finishing the sweeping on each current cell G , we need to do some post-processing: First, discard the stored intersections on $B_{left}(G)$, and then update the lower wave-front curve $LFront$.

4.4 Analysis and Application

Based on a non-trivial analysis (left to full paper), we have the following theorem for topological peeling.

Theorem 1. *The arrangement A_R of n lines within a convex region R can be swept using topological peeling in $O(K + n \log(n + r))$ time and $O(n + r)$ space, where K is the size of the arrangement within R and r is the number of vertices of the boundary of R .*

As an application, we give the following theorem for computing shortest path length between a pair of vertices in an arrangement. The actual path can be obtained in the same amount time and space by applying a path-reporting technique in [8]. (The details of this theorem and the anchor region property of topological peeling are left to full paper).

Theorem 2. *The shortest path length (between a pair of vertices) on an arrangement of n planar lines can be computed in $O(n^2)$ time and $O(n)$ space.*

5 Implementation and Comparisons

We implemented both topological walk and topological peeling on Sun Ultra Sparc 30 workstations, using the C++ based library LEDA 4.1. The executable files for both algorithms have roughly the same size. We experimented the two algorithms on randomly generated arrangements of up to 2000 lines. The execution time for a given n is the average of 20 runs on arrangements of different configurations.

We made two comparisons, one for reporting all the vertices of A_R and the other for reporting the cells of A_R . Since topological walk, as presented in [3,4], does not directly report arrangement cells, we modified it so that it can also report cells. Figure 3 gives the execution times of the two algorithms. Our experimental results show that the execution times of the two algorithms very much observe the time bounds of the theoretical analysis. More interestingly, the experimental results suggest that on average, topological peeling runs 10 – 15% faster than topological walk. (A detailed analysis of the performance of topological peeling will be given in the full paper.)

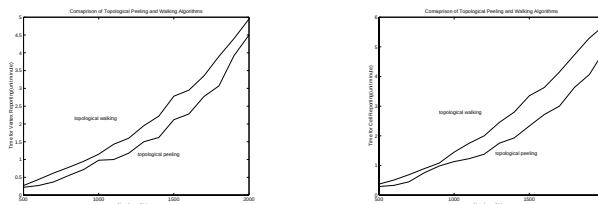


Fig. 3. Comparison of topological peeling and topological walk

References

1. E. G. Anagnostou, L. J. Guibas, and V. G. Polimenis, “Topological sweeping in three dimensions,” *Lecture Notes in Computer Science*, Vol. 450, *Proc. Int’l Symp. on Algorithms (SIGAL’90)*, Springer-Verlag, 1990, pp. 310-317. 454
2. E. M. Arkin, Y.-J. Chiang, M. Held, J. S. B. Mitchell, V. Sacristan, S. S. Skiena, and T.-C. Yang, “On minimum-area hulls,” *Algorithmica*, Vol. 21, 1998, pp. 119-136. 456
3. T. Asano, L. J. Guibas, and T. Tokuyama, “Walking in an arrangement topologically,” *International Journal of Computational Geometry and Applications*, Vol. 4, No. 2, 1994, pp. 123-151. 454, 455, 456, 457, 458, 461, 462, 463, 465
4. T. Asano and T. Tokuyama, “Topological walk revisited,” *Proc. 6th Canadian Conf. on Computational Geometry (CCCG)*, 1994, pp. 1-6. 454, 456, 465
5. P. Bose, W. Evans, D. Kirkpatrick, M. McAllister, and J. Snoeyink, “Approximating shortest paths in arrangements of lines,” *CCCG’96*, pp. 143-148. 456

6. B. Chazelle, L. J. Guibas, and D. T. Lee, "The power of geometric duality," *BIT*, Vol. 25, 1985, pp. 76-90. 454
7. D. Z. Chen, O. Daescu, X. S. Hu, X. Wu, and J. Xu, "Determining an optimal penetration among weighted regions in two and three dimensions," *Proc. 15th Annual ACM Symp. on Computational Geometry (SCG)*, 1999, pp. 322-331. 456
8. D. Z. Chen, O. Daescu, X. S. Hu, and J. Xu, "Finding an optimal path without growing the tree," *ESA '98*, pp. 356-367. 456, 464
9. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, New York, 1987. 454
10. H. Edelsbrunner and L. J. Guibas, "Topologically sweeping an arrangement," *Journal of Computer and System Sciences*, Vol. 38, 1989, pp. 165-194. 454, 456
11. H. Edelsbrunner, and J. O'Rourke, and R. Seidel, "Constructing arrangements of lines and hyperplanes with applications," *SIAM J. Computing*, Vol. 15, 1986, pp. 341-363. 454
12. H. Edelsbrunner and D. Souvaine, "Computing Median-of-Squares Regression Lines and Guided Topological Sweep," *Journal of the American Statistical Association*, Vol. 85, 1990, pp. 115-119. 454
13. H. Edelsbrunner and E. Welzl, "Constructing belts in two-dimensional arrangements with applications," *SIAM J. on Comp.*, Vol. 15, 1986, pp. 271-284. 454
14. D. Eppstein and D. Hart, "An efficient algorithm for shortest paths in vertical and horizontal segments," *WADS'97*, pp. 234-247. 456
15. D. Eppstein and D. Hart, "Shortest paths in an arrangement with k line orientations," *Proc. 10th ACM-SIAM Symp. on Discrete Algorithms (SODA)*, 1999, pp. 310-316. 456
16. K. Hoffman, K. Mehlhorn, R. Rosenstiehl, and R. Tarjan, "Sorting Jordan sequences in linear time using level-linked search trees," *Information and Control*, Vol. 68, 1986, pp. 170-184. 463
17. P. N. Klein, S. Rao, M. H. Rauch, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Proc. 26th Annual ACM Symp. Theory of Computing*, 1994, pp. 27-37. 456
18. J. Majhi, R. Janardan, M. Smid, and P. Gupta, "On some geometric optimization problems in layered manufacturing," *WADS'97*, pp. 136-149. 456
19. K. Miller, S. Ramaswami, P. Rousseeuw, T. Sellares, D. Souvaine, I. Streinu, and A. Struyf, "Fast implementation of depth contours using topological sweep," *Proc. 12th SODA*, 2001, pp. 690-699. 454
20. J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Comm. of the ACM*, Vol. 25, No. 10, 1982, pp. 739-747. 456
21. F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985. 454

Image Segmentation with Monotonicity and Smoothness Constraints^{*}

Danny Z. Chen¹, Jie Wang², and Xiaodong Wu¹ ^{**}

¹ Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, IN 46556, USA

chen@cse.nd.edu

xwu@cse.nd.edu

² Department of Computer Science, University of Massachusetts Lowell
One University Avenue, Lowell, MA 01854, USA

wang@cs.uml.edu

Abstract. Image segmentation with monotonicity and smoothness constraints has found applications in several areas such as biomedical image analysis and data mining. In this paper, we study the problem of segmenting monotone and smooth objects in 2-D and 3-D images. For the 2-D case of the problem, we present an $O(IJ \log J)$ time algorithm, improving the previously best known $O(IJ^2M)$ time algorithm by a factor of $O(\frac{JM}{\log J})$ time, where the size of the input 2-D image is $I \times J$ and M is the smoothness parameter with $1 \leq M \leq J$. Our algorithm is based on a combination of dynamic programming and divide-and-conquer strategy, and computes an optimal path in an implicitly represented graph. We also prove that a generalized version of the 3-D case of the problem is NP-hard.

1 Introduction

The ability to process and analyze image data is a key to solving problems in numerous applications such as medical diagnosis and treatment, mechanical and material study, computer vision, pattern recognition, database, data mining, etc. A central problem in processing and analyzing image data is to define accurate borders between the objects or regions of interest represented by the images. This task, called image segmentation, is in practice quite often performed by human manual tracing. While manual tracing is robust, it is tedious, time-consuming, and can have a significant inter-observer and intra-observer variability [13]. Hence, efficient and effective automated segmentation methods are highly desirable for many applications [2,9,11,13]. However, due to

^{*} The research of the first and the third authors was supported in part by the National Science Foundation under Grants CCR-9623585 and CCR-9988468 and by the 21st Century Research and Technology Fund from the State of Indiana. The work of the second author was supported in part by the National Science Foundation under Grant CCR-9820611.

^{**} Corresponding author.

the inherent visual complexity, efficient and accurate segmentation poses one of the major challenges in image understanding. Five main image segmentation approaches have been used [3]: threshold techniques, edge-based methods, region-based methods, hybrid techniques, and connectivity-preserving relaxation methods, each having its own share of advantages and disadvantages. Segmentation of data sets has also been formulated as optimization problems based on various criteria [8]. In some applications, image segmentation needs to make use of additional information because the target objects are expected to have certain topological or geometric structures or satisfy specific constraints.

In this paper, we study image segmentation with the monotonicity and smoothness constraints in two and three dimensions. Let $P(I, J)$ be a 2-D image of size $I \times J$ (i.e., $P(I, J) = \{(i, j) \mid i = 1, 2, \dots, I, j = 1, 2, \dots, J\}$), and w_{ij} be the brightness level of a pixel (i, j) of $P(I, J)$. Let $n = I \times J$ denote the total number of pixels of $P(I, J)$. The output object is denoted by S_0 , and $S_1 = P(I, J) - S_0$ is the background. A 3-D image can be viewed as an ordered sequence of 2-D images.

A 2-D (resp., 3-D) object Q is said to be *monotone* with respect to a line L (resp., plane P) if for every line L' that is orthogonal to L (resp., P), the intersection $Q \cap L'$ is a connected component (possibly an empty set). A 2-D (resp., 3-D) object is said to be *x-monotone* (resp., *xy-monotone*) if the line L (resp., plane P) is the x -axis (resp., xy -plane). Roughly speaking, the smoothness constraint means that two distinct pixels (i, j) and (k, l) of a 2-D image can be adjacent to each other on the boundary of a segmented object if the i -th and k -th rows are neighboring to each other (i.e., $|i - k| = 1$) and j is “close” enough to l (i.e., $|j - l| < M$, where M is an input parameter with $1 \leq M \leq J$). These constraints will be discussed more carefully in Sections 2 and 3.

Image segmentation of monotone or smooth objects appears in applications. Segmenting monotone and connected objects (which is seemingly quite restricted) has been used as an important step in image segmentation for more general settings [1]. Segmentation of monotone and connected objects has also been applied to extract optimized 2-D association rules from large databases for data mining and financial applications [5, 10, 14]. Certain medical image analysis (e.g., cardiac MRI and intravascular ultrasound imaging) is based on segmenting monotone and smooth objects in 2-D and 3-D [4, 11, 12, 13].

There are several known results on segmenting monotone and/or smooth objects in 2-D and 3-D images [1, 13]. Asano *et al.* [1] presented an $O(I^2 J^2)$ time algorithm for segmenting an x -monotone and connected object in a 2-D image based on optimizing the interclass variance criterion [7] and by using computational geometry techniques. For the problem of segmenting a monotone and smooth object in an image, an $O(IJ^2 M)$ time algorithm for the 2-D case was given in [12, 13], where M is the smoothness parameter, and an exponential time algorithm for the 3-D case was given in [11, 13]. The approaches in [11, 12, 13] are all based on graph searching techniques. For example, for the 2-D case, a graph of size $O(IJM)$ is built from the input image of size $I \times J$, and is repeatedly searched for J times to look for an optimal path [12, 13]. Heuristics for segmenting

a monotone and smooth object in a 3-D image have also been presented in [4,11], but without any theoretical guarantee on the optimality quality.

We consider the same problem as the one studied in [4,11,12,13], that is, image segmentation with the monotonicity and smoothness constraints (the precise definition of the problem will be given later). Our main results are summarized as follows:

- We present an $O(IJ \log J)$ time algorithm for the 2-D case of the problem, improving the previously best known solution [12,13] by a factor of $O(\frac{JM}{\log J})$ time. Note that our time bound is independent of the smoothness parameter M . Our algorithm is based on a combination of dynamic programming and divide-and-conquer strategy, and computes an optimal path in an implicitly represented graph (see Section 2).
- We prove that a generalized version of the 3-D case is NP-hard (see Section 3).

2 Detecting Border Contours in 2-D Images

In this section, we present our $O(IJ \log J)$ time algorithm for the 2-D case of the image segmentation problem with the monotonicity and smoothness constraints, improving the previously best known $O(IJ^2M)$ time solution in [12,13].

Let $G_M = (V, E)$ be a 2-D lattice graph, where $V = \{(i, j) \mid 0 \leq i < I, 0 \leq j < J\}$ and M is a given integer with $1 \leq M \leq J$. Each vertex (i, j) of G_M has a real valued weight w_{ij} . For each vertex $(i, j) \in V$, there is a directed edge going from (i, j) to every vertex $((i+1) \bmod I, j \pm q)$, where $0 \leq q < M, j - q \geq 0$, and $j + q < J$. Besides these edges, there is no other edge in the graph G_M . We call such a graph an *M-smoothness 2-D lattice graph*. For a $j \in \{0, 1, \dots, J-1\}$, let p_j be a path in G_M from the vertex $(0, j)$ to a vertex $\in \{(I-1, j \pm q) \mid q = 0, 1, \dots, M-1, j+q < J, j-q \geq 0\}$. Such a path is called a *c-path*. We define the weight of a path p in G_M , $w(p)$, as $\sum_{(i,j) \in p} w_{ij}$. For any $k = 0, 1, \dots, J-1$, let p_k^* be a minimum-weight c-path in G_M that starts at the vertex $(0, k)$. Our goal is to compute a c-path p^* , whose weight is the minimum among all c-paths in G_M , i.e., $w(p^*) = \min\{w(p_0^*), w(p_1^*), \dots, w(p_{J-1}^*)\}$.

The problem of computing an optimal c-path p^* in G_M is well motivated by the need of detecting the border contours of monotone and smooth objects in 2-D biomedical images. Monotonicity and smoothness characterize an abundance of objects in medical images, e.g., vessels, bones, ducts, spinal cords, and bowels. A 2-D image $P(I, J)$ can be viewed as representing a setting on a cylindrical surface, with the last row of $P(I, J)$ being treated as being adjacent to the first row (i.e., $P(I, J)$ is “bended” to form a cylinder). A “smooth” contour C_M in such a “cylindrical” image $P(I, J)$ can be defined as follows [12,13]:

1. C_M starts at a pixel $(0, j_0)$ in the first row of $P(I, J)$, for some $j_0 \in \{0, 1, \dots, J-1\}$.
2. C_M consists of a sequence of I pixels $(0, j_0), (1, j_1), \dots, (I-1, j_{I-1})$, one from each row of $P(I, J)$, such that for every $k = 0, 1, \dots, I-1$,

$|j_k - j_{(k+1)} \bmod I| < M$ (i.e., C_M satisfies the monotonicity and smoothness constraints).

Note that the contour C_M is really a closed path in the “cylindrical” $P(I, J)$ that is monotone and smooth. The boundaries of some medical objects in 2-D images can be modeled as such contours [4,11,12,13], and it is natural that one would like to find the “best” contour (i.e., based on certain optimality criteria) to bound a sought object.

We model an input 2-D image $P(I, J)$ as a directed acyclic graph $G_M = (V, E)$ with vertex weights, such that each pixel of $P(I, J)$ corresponds to a vertex in V , and the edges of E represent the connections among the pixels to form feasible object borders, which, in fact, enforce the monotonicity and smoothness constraints. The weight of a vertex in V is inversely related to the likelihood that it may be present at the desired border contour, which is usually determined by using simple low-level image features [13]. An optimal c-path p^* of the minimum total vertex weight in G_M is the desired border in certain applications [12,13], since such a path captures both the local and global information in determining an optimal contour in the image.

We now give an $O(IJ \log J)$ time algorithm for computing an optimal c-path p^* in G_M . For simplifying the discussion of c-paths, we modify G_M in the following way: Duplicate the first row of G_M , append it after the last row of G_M , let the vertices of the appended row all have a weight zero, and add directed edges from the vertices of the last row of G_M to the vertices of the appended row based on the M -smoothness constraint. We denote the appended row as row I and the modified graph as G_M^a . A 2-smoothness 2-D lattice graph G_M^a is shown in Figure 1(a), where the appended vertices are dashed circles. Note that any c-path p_j in G_M can be viewed as a c-path p_j^a in G_M^a that starts at the vertex $(0, j)$ and ends at the vertex (I, j) . In Figure 1(a), the path p consisting of solid thick edges is a c-path in G_M^a , while p' consisting of dashed edges is not. Henceforth, our focus will be on G_M^a and its c-paths, and we simply denote G_M^a by G_M and its c-paths by p_j .

Let p and p' be two c-paths in G_M starting at vertices $(0, j_0)$ and $(0, j'_0)$, respectively, with $0 \leq j_0 < j'_0 < J$. We say that each vertex (i, j_i) on p has a *corresponding vertex* (i, j'_i) on p' in the row i . In a similar way, we define the *corresponding subpath* s' on p' for each subpath s on p . A vertex (i, j_i) on p is said to be strictly to the left (resp., right) of p' if its corresponding vertex (i, j'_i) on p' has a larger (resp., smaller) column index, i.e., $j'_i > j_i$ (resp., $j'_i < j_i$). Two c-paths p and p' are said to *cross each other* if a vertex on p is strictly to the left of p' and another vertex on p is strictly to the right of p' . Given a subpath $s = \{(i, j_i), \dots, (i+m, j_{i+m})\}$ on p and its corresponding subpath $s' = \{(i, j'_i), \dots, (i+m, j'_{i+m})\}$ on p' , with $i > 0$ and $i+m < I$, s and s' are said to form a *crossed pair* if $j_{i-1} \leq j'_{i-1}, j_{i+k} > j'_{i+k}$ for $k = 0, 1, \dots, m$, and $j_{i+m+1} \leq j'_{i+m+1}$. If p and p' cross each other, then there is certainly at least one crossed pair between p and p' .

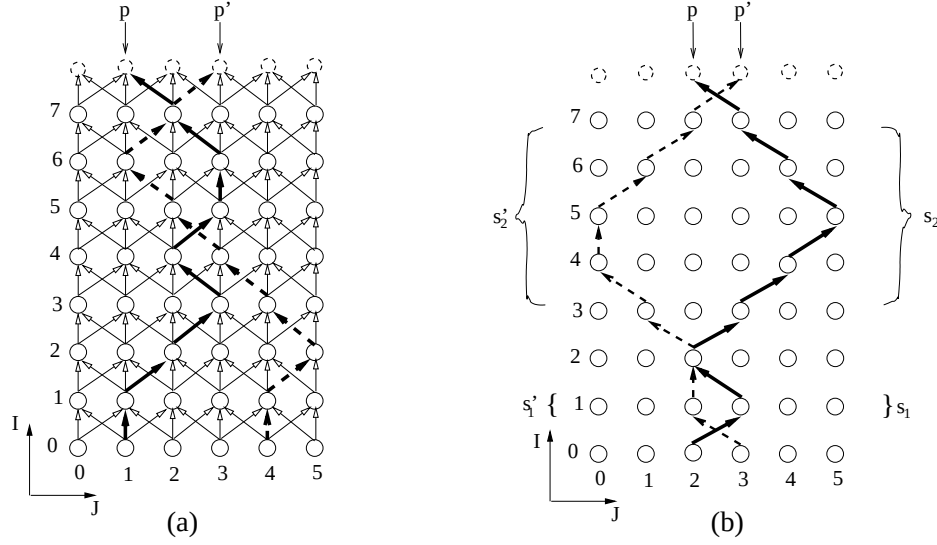


Fig. 1. (a) A 2-smoothness 2-D lattice graph, in which p is a c-path while p' is not. (b) Two c-paths crossing each other and their crossed pairs

Observation 1 Let two c-paths p and p' start at vertices $(0, j_0)$ and $(0, j'_0)$, respectively, with $j_0 < j'_0$. If p and p' cross each other, then they have a crossed pair.

Figure 1(b) illustrates two c-paths p and p' crossing each other. For simplicity, we only show the edges on the paths. Therein, the vertex $(0, 2)$ on p is strictly to the left of p' and the vertex $(4, 4)$ on p is strictly to the right of p' . There are two crossed pairs, (s_1, s_1') and (s_2, s_2') , between p and p' .

The next lemma is a key to our algorithm for computing the optimal path.

Lemma 1. There exist optimal c-paths $p_0^*, p_1^*, \dots$, and p_{J-1}^* that do not cross each other.

Lemma 1 provides a basis for a divide-and-conquer algorithm for computing the optimal c-paths p_j^* for every $j = 1, 2, \dots, J - 1$. Of course, the optimal path p^* can then be obtained from $p_0^*, p_1^*, \dots, p_{J-1}^*$. To compute all c-paths $p_0^*, p_1^*, \dots, p_{J-1}^*$ in G_M , we first compute the minimum-weight c-path $p_{\lceil (J-1)/2 \rceil}^* = \{(0, j_0^p), (1, j_1^p), \dots, (I, j_I^p)\}$, where $j_0^p = j_I^p = \lceil (J-1)/2 \rceil$. Using $p_{\lceil (J-1)/2 \rceil}^*$, we define two sets $J_i^L = \{0, 1, \dots, j_i^p\}$ and $J_i^R = \{j_i^p, j_i^p + 1, \dots, J-1\}$, for every $i = 0, 1, \dots, I$. Then along the c-path $p_{\lceil (J-1)/2 \rceil}^*$, we decompose the graph G_M into two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where $V_1 = \{(i, j) \mid i \in \{0, 1, \dots, I\}, j \in J_i^L\}$, $E_1 = \{e \in E \mid \text{both vertices of } e \text{ are in } V_1\}$, $V_2 = \{(i, j) \mid i \in \{0, 1, \dots, I\}, j \in J_i^R\}$, and $E_2 = \{e \in E \mid \text{both vertices of } e \text{ are in } V_2\}$. Figure 2 illustrates the decomposition of the graph G_M into two subgraphs G_1 and G_2 along the c-path p_3^* . Based on Lemma 1, there

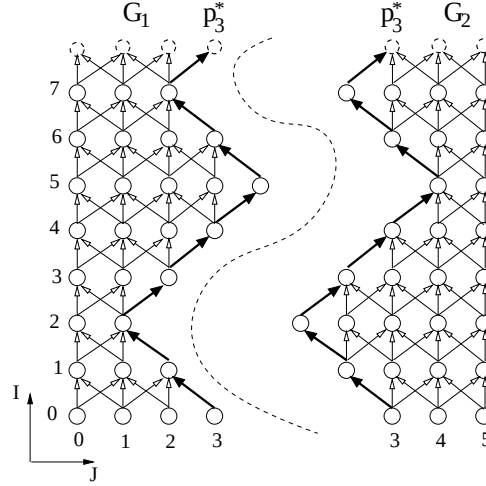


Fig. 2. Illustrating the divide-and-conquer algorithm for computing the optimal c_paths p_j^*

exist minimum-weight c_paths of G_M in G_1 , $p_{G_1}^*$, that do not cross the minimum-weight c_paths of G_M in G_2 , $p_{G_2}^*$. Therefore, we compute recursively $p_{G_1}^*$ and $p_{G_2}^*$ in G_1 and G_2 , respectively. Clearly, the recursion tree of our above divide-and-conquer algorithm has $O(\log J)$ levels; at each level, a subset of c_paths is computed (in certain subgraphs of G_M).

Next, we show how to efficiently compute one minimum-weight c_path in G_M , say, p_r^* that starts at the vertex $(0, r)$ for any $r \in \{0, 1, \dots, J-1\}$. Note that a straightforward dynamic programming algorithm can compute p_r^* in $O(IJM)$ time, because G_M is a directed acyclic graph with $O(IJ)$ vertices and $O(IJM)$ edges. But, we can do better by presenting an $O(IJ)$ time dynamic programming algorithm for computing p_r^* in G_M .

We begin with a less efficient algorithm for computing p_r^* in G_M . First, observe that the edges of G_M can be represented *implicitly*. That is, without explicitly storing its edges, we can determine for every vertex of G_M the set of its incoming and outgoing neighbors in $O(1)$ time. Our algorithm uses this implicit representation of G_M . We denote the minimum-weight path in G_M from the vertex $(0, r)$ to the vertex (i, j) by $p_{r(i,j)}$. Then, as shown in Figure 3, $p_{r(i,j)}$ depends on the set of optimal paths from $(0, r)$ to the interval of vertices $\{(i-1, j \pm q) \mid 0 \leq q < M, j-q \geq 0, j+q < J\}$. We denote the set $\{p_{r(i-1, j \pm q)} \mid 0 \leq q < M, j-q \geq 0, j+q < J\}$ of optimal paths by $S_{(i,j)}$. Hence, $w(p_{r(i,j)}) = \min\{w(p) \mid p \in S_{(i,j)}\} + w_{ij}$. One can certainly apply a dynamic programming technique to compute the minimum-weight paths $p_{r(i,k)}$ from $p_{r(i-1,k)}$, for all $k = 0, 1, \dots, J-1$. Suppose $p_{r(i-1,0)}, p_{r(i-1,1)}, \dots$, and $p_{r(i-1,J-1)}$ have all been computed and the weights of these paths are stored in an array W_{i-1} . We define the M -neighbors of an element $W_{i-1}[j]$ in W_{i-1} to be the set $\{W_{i-1}[j \pm q] \mid 0 \leq q < M, j-q \geq 0, j+$

$q < J$ }, denoted by $N_M^j(W_{i-1})$. To compute the minimum-weight path $p_{r(i,j)}$ at vertex (i, j) , we need to obtain the minimum value among $N_M^j(W_{i-1})$, denoted by $N_M^*(W_{i-1})[j]$. We call $N_M^*(W_{i-1})[j]$ the *minimum- M -neighbor* of $W_{i-1}[j]$. Also, note that $S_{(i,j+1)} = S_{(i,j)} \cup \{p_{r(i-1,j+M)}\} - \{p_{r(i-1,j-(M-1))}\}$ (see Figure 3). One way to compute $N_M^*(W_{i-1})[k]$ for every $k = 1, 2, \dots, J-1$ is to use a balanced search tree to maintain the minimum weight of the paths in $S_{(i,k)}$, and scan the vertices of row $i-1$ from left to right to compute $N_M^*(W_{i-1})[k]$. When moving from computing $N_M^*(W_{i-1})[k]$ to computing $N_M^*(W_{i-1})[k+1]$, $w(p_{r(i-1,k-(M-1))})$ is deleted from the balanced search tree and $w(p_{r(i-1,k+M)})$ is inserted into the search tree, which takes $O(\log M)$ time. Therefore, it takes $O(J \log M)$ time to obtain $N_M^*(W_{i-1})[k]$ for all $k = 1, 2, \dots, J-1$. Thus, we can compute the minimum-weight c-path p_r^* in $O(IJ \log M)$ time.

Interestingly, based on Lemma 2 below, we can further get rid of the $\log M$ factor for the time complexity.

For an array A of n real numbers, we define the *left minimum prefix* of A , A^L , as $A^L[i] = \min\{A[k] \mid k = 0, 1, \dots, i\}$, and the *right minimum prefix* of A , A^R , as $A^R[i] = \min\{A[k] \mid k = n-1, n-2, \dots, i\}$, for every $i = 0, 1, \dots, n-1$.

Lemma 2. *Given an array A of n real numbers and an integer M with $1 \leq M \leq n$, all the minimum- M -neighbors of A , $N_M^*(A)$, can be computed in $O(n)$ time.*

Proof. As illustrated in Figure 4, we partition A into $K = \lceil \frac{n}{2M-1} \rceil$ subarrays, $A_k = \{A[h] \mid (2M-1)*k \leq h < (2M-1)*(k+1)\}$, where $k = 0, 1, \dots, K-1$. For every A_k , we compute the left minimum prefix, A_k^L , and the right minimum prefix, A_k^R , separately, which takes altogether $O(n)$ time. As defined above, $N_M^*(A)[j] = \min B_j$, where $B_j = \{A[j \pm q] \mid 0 \leq q < M, j - q \geq 0, j + q < n\}$, for every $j = 0, 1, \dots, n-1$. We now show that $\min B_j$ can be obtained in $O(1)$ time for every j . Observe that B_j spans either one or two subarrays A_k of A . If B_j spans only one subarray, say A_k , where $k = \lfloor \frac{j}{2M-1} \rfloor$, then obviously $N_M^*(A)[j] = A_k^L[j + (M-1)]$. If B_j spans two subarrays, say A_k and A_{k+1} ,

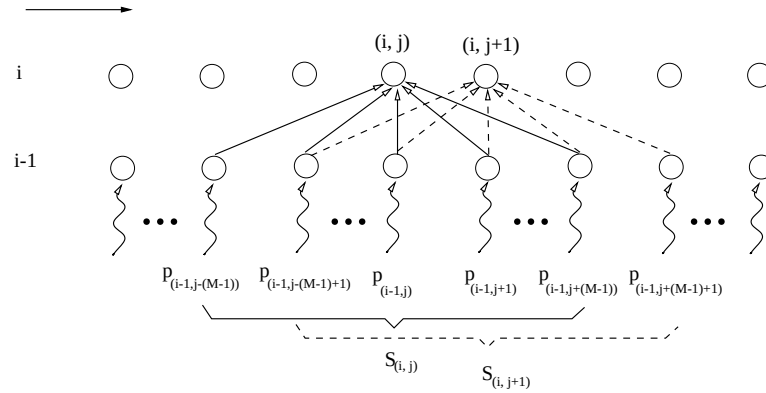


Fig. 3. Computing a minimum-weight c-path p_r^* from a vertex $(0, r)$

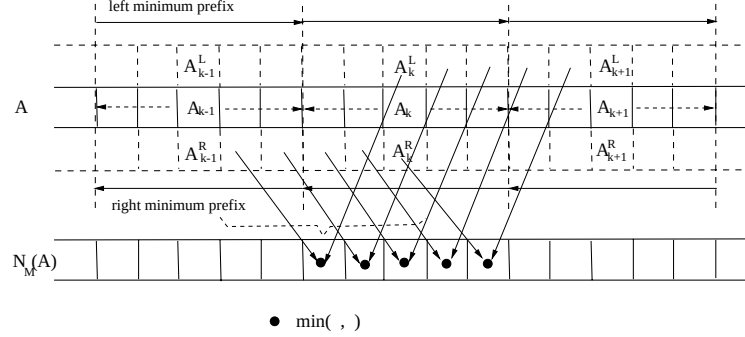


Fig. 4. Each B_j spans at most two subarrays A_k . Herein $M = 3$

where $k = \lfloor \frac{j-(M-1)}{2M-1} \rfloor$, then $N_M^*(A)[j] = \min\{A_k^R[j-(M-1)], A_{k+1}^L[j+(M-1)]\}$, since $A_k^R[j-(M-1)] = \min\{A[j-(M-1)], \dots, A[(k+1)*(2M-1)-1]\}$ and $A_{k+1}^L[j+(M-1)] = \min\{A[(k+1)*(2M-1)], \dots, A[j+(M-1)]\}$. Therefore, the total time for computing $N_M^*(A)$ is $O(n)$. $\square$

Our algorithm computes the minimum-weight paths in G_M from a vertex $(0, r)$ to all other vertices (i, j) (if they are reachable from $(0, r)$). The computation proceeds row by row. For a row $i-1$, assume W_{i-1} is the array storing the weights of optimal paths $p_{r(i-1, j)}$ for all $j = 0, 1, \dots, J-1$. Based on Lemma 2, we can compute all the minimum- M -neighbors of W_{i-1} , $N_M^*(W_{i-1})$, in $O(J)$ time. Thus, from the paths $p_{r(i-1, j)}$, we can obtain the minimum-weight paths $p_{r(i, j)}$ in $O(J)$ time, for every $j \in \{0, 1, \dots, J-1\}$. Therefore, the total time for computing the minimum-weight c_path p_r^* from the vertex $(0, r)$ is $O(IJ)$.

Theorem 1. *Given an implicitly represented M -smoothness 2-D lattice graph G_M , an minimum-weight c_path p^* in G_M can be computed in $O(IJ \log J)$ time.*

3 NP-Hardness of Detecting Border Surfaces in 3-D Images

In this section, we prove the hardness of a generalized version of the 3-D case of the image segmentation problem with the monotonicity and smoothness constraints.

We say that a graph $G = (V, E)$ is a 3-D lattice graph if $V = \{(i, j, k) \mid 0 \leq i < I, 0 \leq j < J, 0 \leq k < K\}$. In particular, we are interested in the following special kind of lattice graphs:

1. Each vertex (i, j, k) is associated with a real valued weight w_{ijk} .
2. There are two positive integers M and N such that every vertex (i, j, k) , $0 \leq i < I-1$, $0 \leq j < J$, $0 \leq k < K$, is connected by directed edges to vertices $(i, (j+1) \bmod J, k \pm p)$ and to vertices $(i+1, j, k \pm q)$, where $0 \leq p < M$, $0 \leq q < N$, $k-p \geq 0$, $k+p < K$, $k-q \geq 0$, and $k+q < K$.
3. Besides those edges defined in (2), there are no other edges in the graph.

Such a graph is called an (M, N) -smoothness 3-D lattice graph.

For each $i \in \{0, 1, \dots, I-1\}$, let $F_i^{(\mathbf{j}, \mathbf{k})} = \{(i, j, k) \mid (i, j, k) \in V, 0 \leq j < J, 0 \leq k < K\}$ and call it the i -th $(\mathbf{j}, \mathbf{k})$ -frame of G . For each $0 \leq j < J$, let $F_j^{(\mathbf{i}, \mathbf{k})} = \{(i, j, k) \mid (i, j, k) \in V, 0 \leq i < I, 0 \leq k < K\}$ and call it the j -th $(\mathbf{i}, \mathbf{k})$ -frame of G . Let p_i be an (M) -smooth path of G on $F_i^{(\mathbf{j}, \mathbf{k})}$ from a vertex in the set $\{(i, 0, k) \mid 0 \leq k < K\}$ to a vertex in the set $\{(i, J-1, k) \mid 0 \leq k < K\}$. We denote p_i by $\langle (i, 0, k_{i,0}), (i, 1, k_{i,1}), \dots, (i, J-1, k_{i,J-1}) \rangle$. Here, we require $k_{i,0} - M < k_{i,J-1} < k_{i,0} + M$. The weight of a path p_i is defined as $w(p_i) = \sum_{j=0}^{J-1} w_{ij k_{i,j}}$.

The **Optimal 3-D Border Detection (3OBD)** problem is to find a set of (M) -smooth paths p_i , one on each $(\mathbf{j}, \mathbf{k})$ -frame $F_i^{(\mathbf{j}, \mathbf{k})}$ of G ($0 \leq i < I$), such that the following two conditions hold.

1. Let V' be the subset of vertices of V that appear on any of the sought M -smooth paths p_i , and let $G' = (V', E')$ be the subgraph of G induced by V' . Then, for every $j = 0, 1, \dots, J-1$, there is an (N) -smooth path q_j in the induced subgraph G' connecting the following I vertices across all the $(\mathbf{j}, \mathbf{k})$ -frames: $(0, j, k_{0,j}), (1, j, k_{1,j}), \dots, (I-1, j, k_{I-1,j})$. The path q_j is on the $(\mathbf{i}, \mathbf{k})$ -frame $F_j^{(\mathbf{i}, \mathbf{k})}$ and is N -smooth. Note that for the two end vertices of q_j , we do not require $k_{0,j} - N < k_{I-1,j} < k_{0,j} + N$.
2. The summation $\sum_{i=0}^{I-1} \delta(p_i) w(p_i)$ is minimized, where $\delta(p_i)$ is a non-negative cost associating with the path p_i in each $(\mathbf{j}, \mathbf{k})$ -frame $F_i^{(\mathbf{j}, \mathbf{k})}$ of G .

Note that condition (1) above ensures that any two consecutive paths p_i and p_{i+1} that we seek are N -smooth with respect to each other, in the sense that, for any j , the two corresponding vertices $(i, j, k_{i,j})$ (on p_i) and $(i+1, j, k_{i+1,j})$ (on p_{i+1}) are such that $|k_{i,j} - k_{i+1,j}| < N$. Therefore, the set of desired paths p_i forms a monotone and (M, N) -smooth 3-D “surface” in the 3-D lattice graph G . The monotonicity of this 3-D surface is with respect to the IJ -plane and is ensured by the definition of the paths p_i . The M -smoothness of the surface is ensured by the definition of the paths p_i , and the N -smoothness of the surface is ensured by condition (1) above.

The 3OBD problem finds applications in 3-D biomedical image segmentation for monotone and smooth objects. With the advances of imaging techniques, 3-D volumetric image data are widely available from magnetic resonance, X-ray, ultrasound, and other tomographic scanners, which consist of stacked 2-D image slices. Segmenting volumetric biomedical images is to identify 3-D surfaces representing the boundaries of the sought objects in the 3-D space. In a common practice, 2-D image slices are more or less analyzed independently and then the 2-D results are stacked to form the final 3-D segmentation. It is intuitively obvious that a set of 2-D borders that are detected in individual slices may be far from the desired 3-D surface if the entire 3-D volume is considered, and concurrent analysis of the entire 3-D volume gives better results if a surface is globally determined [11]. In some applications, the sought object borders may further need to be sufficiently “smooth” (e.g., every two neighboring border vox-

els must be within certain specified distances, say M or N , with M characterizing the smoothness within the same slice, and N specifying the smoothness across neighboring slices). As in [4,11,13], we model this 3-D image segmentation as a monotone surface detection problem on an (M, N) -smoothness 3-D lattice graph $G = (V, E)$. Each vertex in V corresponds to a voxel of the input 3-D image. The edges of E enforce the smoothness constraints between corresponding pairs of voxels (i.e., there is an edge between two vertices if the corresponding two voxels meet the smoothness condition). The weight of a vertex is determined in a similar way as in Section 2. Thus, an optimal surface in G corresponds to the desired 3-D object border that we are looking for. Thedens, Skorton, and Fleagle [13] and Frank *et al.* [4] considered such a problem in which $\delta(\cdot) \equiv 1$ (cf. condition (2) above). However, the optimal 3-D border detection (3OBD) problem appears quite difficult to solve optimally.

We now show that 3OBD is NP-hard. The following is the decision version of 3OBD.

3-D Border Detection (3BD):

Instance: Given a quintuple (G, M, N, δ, C) , where $G = (V, E)$ is an (M, N) -smoothness 3-D lattice graph, $V = \{(i, j, k) \mid 0 \leq i < I, 0 \leq j < J, 0 \leq k < K\}$, $\delta(\cdot)$ is the cost function for every path p_i in each $(\mathbf{j}, \mathbf{k})$ -frame $F_i^{(\mathbf{j}, \mathbf{k})}$, and C is a positive integer.

Question: Are there paths p_i ($0 \leq i < I$) such that the following two conditions hold?

1. Same as Condition 1 in 3OBD.
2. $\sum_{i=0}^{I-1} \delta(p_i)w(p_i) < C$.

The following lemma is straightforward.

Lemma 3. *If 3OBD is solvable in polynomial time, then so is 3BD.*

Lemma 4. *3BD is NP-complete.*

Proof. We will reduce the NP-complete 3-dimensional matching [6] to 3BD.

3-DIMENSIONAL MATCHING (3DM):

Instance: Given a set $U \subseteq X \times Y \times Z$, where X, Y , and Z are disjoint sets each having q ($q > 1$) elements.

Question: Does U contain a matching, i.e., a subset $U' \subseteq U$ such that $|U'| = q$ and no two elements of U' agree in any coordinate?

Let $t \in X \times Y \times Z$. Denote by $t.x$ the first coordinate of t (called the x -coordinate), $t.y$ the second coordinate (called the y -coordinate), and $t.z$ the third coordinate (called the z -coordinate). Let $X = \{x_0, x_1, \dots, x_{q-1}\}$, $Y = \{y_0, y_1, \dots, y_{q-1}\}$, and $Z = \{z_0, z_1, \dots, z_{q-1}\}$. Partition U into q disjoint subsets $U_0, U_1, \dots, U_{q-1}$ such that each U_l ($0 \leq l < q$) consists of all elements in U whose first coordinate is x_l . We fix an enumeration for each U_l .

From the NP-completeness proof of 3DM in [6], we can further restrict U such that for every j_1 and j_2 between 0 and $q-1$, $|U_{j_1}| \leq |U_{j_2}| \pm 1$, and for all j between 0 and $q-1$, $|U_j| \leq \lceil |U|/q \rceil$ (i.e., with such a restriction on U , 3DM is still NP-complete).

From such a 3DM instance, we construct an instance of 3BD, as follows.

Let $I = 3$, $J = q$, and $K = |U|$. Let $M = \lceil |U|/q \rceil + 1$ and $N = 1$. The set of vertices for the desired lattice graph is defined as $V = \{(i, j, k) \mid 0 \leq i < I, 0 \leq j < J, 0 \leq k < K\}$, such that for each j ($0 \leq j < J$), $((0, j, k), (1, j, k), (2, j, k))$ corresponds to the $(k+1)$ -th element, say $u_{k+1} = (x_l, y_m, z_n)$, in a fixed ordering of $U_j, \dots, U_{q-1}, U_0, \dots, U_{j-1}$ where l, m , and n are the indices of $u_{k+1}.x$, $u_{k+1}.y$ and $u_{k+1}.z$ in the sets X , Y and Z , respectively. Then, $w_{0jk} = l + 1$, $w_{1jk} = m + 1$, and $w_{2jk} = n + 1$.

Next, we construct the edges in E as follows.

1. Each vertex (i, j, k) is connected to vertex $(i + 1, j, k)$, where $i \in \{0, 1\}$, $0 \leq j < J$, and $0 \leq k < K$.
2. For any vertex $(i, j, k) \in V$, connect (i, j, k) to vertices $(i, (j+1) \bmod J, k \pm p)$, where $0 \leq p < M$, $k - p \geq 0$, and $k + p < K$.
3. There are no other edges besides those defined in Steps 1–2.

We define $\sigma_i \triangleq \min\{|w(a) - w(b)| \mid a \text{ and } b \text{ are different vertices on } p_i\}$, $i = 0, 1, 2$. Let $\delta(p_i) = \frac{3q^2}{3q^2\sigma_i + 1}$ and $C = 3q^2$.

Then (G, M, N, δ, C) is an instance of 3BD. The desired reduction f maps U to (G, M, N, δ, C) in polynomial time in $|U|$.

Assume that U is a positive instance of 3DM. Then there is a subset $U' \subseteq U$ such that $|U'| = q$ and such that no two elements in U' agree in any coordinate. Let $u'_0, u'_1, \dots, u'_{q-1}$ be an enumeration of U' such that $u'_j.x = x_j$ for $0 \leq j < q$. For simplicity, we use $u'_j.x$, $u'_j.y$, and $u'_j.z$ to represent the vertices $(0, j, k_{0,j})$, $(1, j, k_{1,j})$, and $(2, j, k_{2,j})$ in G corresponding to the element $u'_j \in U'$, respectively. Let

$$\begin{aligned} p_0 &= \langle u'_0.x, \dots, u'_{q-1}.x \rangle, \\ p_1 &= \langle u'_0.y, \dots, u'_{q-1}.y \rangle, \\ p_2 &= \langle u'_0.z, \dots, u'_{q-1}.z \rangle. \end{aligned}$$

Recall that $|U_i| \leq \lceil |U|/q \rceil$ for each subset U_i . Hence, it follows from the way we set up the vertices of G that p_i ($0 \leq i < I$) is a path on $\{(i, j, k) \mid 0 \leq j < J \text{ and } 0 \leq k < K\}$. Since for any $j_1 \neq j_2$ between 0 and $q - 1$, $u'_{j_1}.x \neq u'_{j_2}.x$, $u'_{j_1}.y \neq u'_{j_2}.y$, and $u'_{j_1}.z \neq u'_{j_2}.z$, we have $w(p_i) = \sum_{s=1}^q s < q^2$ (note that $q > 1$), $i = 0, 1, 2$, and $\sigma_i \geq 1$. Thus, $\delta(p_i) < 1$, which implies that $\sum_{i=0}^2 \delta(p_i)w(p_i) < 3q^2$. Moreover, by the construction of G , there is a path q_j connecting $u'_j.x$, $u'_j.y$, and $u'_j.z$, for every $j = 0, 1, \dots, J - 1$. Hence, (G, M, N, δ, C) is a positive instance of 3BD.

Conversely, assume that (G, M, N, δ, C) is a positive instance of 3BD. Then there are three paths p_i ($i = 0, 1, 2$) with $\sum_{i=0}^2 \delta(p_i)w(p_i) < 3q^2$. For simplicity, let $k_{i,j}$ denote the vertex $(i, j, k_{i,j})$ in G . x'_j , y'_j , and z'_j denote the x -coordinate, y -coordinate, and z -coordinate of the element in U corresponding to the vertices $k_{0,j}$, $k_{1,j}$, and $k_{2,j}$, respectively, where $j = 0, 1, \dots, q - 1$. Let

$$\begin{aligned}
p_0 &= \langle k_{0,0}, \dots, k_{0,q-1} \rangle, \\
p_1 &= \langle k_{1,0}, \dots, k_{1,q-1} \rangle, \\
p_2 &= \langle k_{2,0}, \dots, k_{2,q-1} \rangle.
\end{aligned}$$

Then for any j_1 and j_2 between 0 and $q-1$, if $j_1 \neq j_2$, we have $x'_{j_1} \neq x'_{j_2}$, $y'_{j_1} \neq y'_{j_2}$, and $z'_{j_1} \neq z'_{j_2}$. To see why this must be the case, assume that one of these inequalities is not true, say $z'_{j_1} = z'_{j_2}$ for some $j_1 \neq j_2$, then $\sigma_2 = 0$. Thus, $\delta(p_2) = 3q^2$, which implies that $\delta(p_2)w(p_2) \geq 3q^3 > 3q^2$, a contradiction.

Since (G, M, N, δ, C) is a positive instance of 3BD, for each j between 0 and $q-1$, there is a path q_j connecting vertices $k_{0,j}$, $k_{1,j}$, and $k_{2,j}$ for $j = 0, 1, \dots, J-1$. Thus, $(x'_j, y'_j, z'_j) \in U$. It follows that $\{(x'_0, y'_0, z'_0), (x'_1, y'_1, z'_1), \dots, (x'_{q-1}, y'_{q-1}, z'_{q-1})\}$ is a subset of U and no two elements of this subset agree in any coordinate. Therefore, U is a positive instance of 3DM.

This completes the proof. $\square$

Theorem 2. *3OBD is NP-hard.*

References

1. T. Asano, D. Z. Chen, N. Katoh, and T. Tokuyama, Efficient algorithms for optimization-based image segmentation, accepted to *International Journal of Computational Geometry and Applications*. 468
2. G. P. Ashkar and J. Modestino, The contour extraction problem with biomedical applications, *Comput. Graphics Image Process.*, 7 (1978), pp. 331–355. 467
3. P. J. Besl and R. C. Jain, Segmentation through variable-order surface fitting, *IEEE Trans. Pattern and Machine Intell.*, 10 (1988), pp. 167–192. 468
4. R. J. Frank, D. D. McPherson, K. B. Chandran, and E. L. Dove, Optimal surface detection in intravascular ultrasound using multi-dimensional graph search, *Computers in Cardiology*, IEEE, Los Alamitos, CA, 1996, pp. 45–48. 468, 469, 470, 476
5. T. Fukuda, S. Morishita, Y. Morimoto, and T. Tokuyama, Data mining using two-dimensional optimized association rules – Scheme, algorithms, and visualization, *Proc. SIGMOD Int. Conf. on Management of Data*, 1996, pp. 13–23. 468
6. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, NY, 1979. 476
7. D. J. Hand, *Discrimination and Classification*, John Wiley & Sons, 1981. 468
8. J. Kleinberg, C. Papadimitriou, and P. Raghavan, Segmentation problems, *Proc. 30th Annual ACM Symp. on Theory of Computing*, 1998, pp. 473–482. 468
9. H. K. Liu, Two- and three-dimensional boundary detection, *Comput. Graphics Image Process.*, 6 (1977), pp. 123–134. 467
10. Y. Morimoto, T. Fukuda, H. Matsuzawa, T. Tokuyama, and K. Yoda, Multivariate rules in data mining: A key to handling correlations in financial data, *Proc. KDD Workshop in Finance*, AAAI, 1998, pp. 54–59. 468
11. M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis, and Machine Vision*, 2nd edition, Brooks/Cole Publishing Company, Pacific Grove, CA, 1999, pp. 199–205. 467, 468, 469, 470, 475, 476

12. D. R. Thedens, Automated cardiac border detection in image sequences with application to magnetic resonance imaging, M. S. thesis, Department of Electrical Engineering, University of Iowa, August 1993. 468, 469, 470
13. D. R. Thedens, D. J. Skorton, and S. R. Fleagle, Methods of graph searching for border detection in image sequences with applications to cardiac magnetic resonance imaging, *IEEE Trans. on Medical Imaging*, 14 (1) (1995), pp. 42–55. 467, 468, 469, 470, 476
14. T. Tokuyama, Application of algorithm theory to data mining, *Proc. of Australasian Computer Science Conf.*, 1998, pp. 1-15. Also in *Proc. CATS98*. 468

Optimization Algorithms for Sweeping a Polygonal Region with Mobile Guards

Jae-Ha Lee, Sang-Min Park, and Kyung-Yong Chwa

Department of Computer Science
KAIST, Korea
{jhlee,smpark,kychwa}@jupiter.kaist.ac.kr

Abstract. We study the problem of sweeping a simple polygon using a chain of mobile guards. The basic question is as follows: Given a simple polygon P in the plane, is it possible for two guards to simultaneously walk along the boundary of P from one point to another point in such a way that two guards are always mutually visible and any target moving continuously inside P should eventually lie on the line segment between two guards? It is known that an $O(n^2)$ -time algorithm can decide this question. Our contribution is to present efficient algorithms for the following optimization problems:

- Given an n -sided polygon, we present an $O(n^2 \log n)$ -time algorithm for computing a *shortest walk* in which the total length of the paths that two guards traverse is minimized.
- Given an n -sided polygon, we present an $O(n^2)$ -time algorithm for computing a *minimum diameter walk* in which the maximum distance between two guards is minimized.

Finally we allow more than two guards. Here the guards should form a simple chain within the polygon such that any consecutive two guards along the chain are mutually visible and the first and last guard have to move along the boundary but others do not.

- We present an $O(n^2)$ -time algorithm for computing the minimum number of guards to sweep an n -sided polygon and an $O(n^3)$ -time algorithm for computing such a schedule.

1 Introduction

The visibility-based pursuit-evasion problem is that of planning the motion of one or more searchers in a polygonal environment to eventually see an intruder that is unpredictable, has unknown initial position, and is capable of moving arbitrarily fast [14,7,2,5,10,12]. This problem can model many practical applications such as search for an intruder in a house, rescue of a victim in a dangerous house and other surveillance with autonomous mobile robots. The motion plan calculated could be used by robots or human searchers.

In this paper, we look at a more constrained but still realistic model of visibility-based pursuit-evasion. Imagine that there is a fugitive inside a polygonal region P and he is capable of moving arbitrarily fast. Two guards move along the boundary of P and there is a laser-beam detector between them. The

fugitive is *detected* by the two guards if he touches the laser beam between the two guards. (Note that the region visible from two guards amounts to a line segment joining them.) Can the two guards always detect the fugitive? Or can the fugitive keep sneaking out of detection by two guards? It depends on the geometry of the polygon. We can easily obtain an $O(n^2)$ -time algorithm for testing feasibility based on many previous work [9,10,8,3], which will be reviewed in Section 3.

The above problem was first studied in a different setting [7]: We are given a simple polygon P and a pair of distinct points s and g on the boundary of P . Two guards are required to walk from s to g in such a way that each guard lies in each of the two boundary chains from s to g and the line segment connecting the two guards is fully contained in P all the time. Icking and Klein [7] showed a characterization of the class of *walkable* polygons (P, s, g) and presented an $O(n \log n + k)$ -time algorithm for constructing a walk of length k , where n is the number of edges in P . Narashimhan [11], Heffernan [6], and Tseng *et al.* [15] improved this result in various directions and recently, Bhattacharya *et al.* [1] obtained a linear-time algorithm.

In these previous studies, however, *two guards cannot move past the prespecified points s and g* . In this respect, they regard the points s and g as *doors* at which the polygon boundary is disconnected [7,2]. Even in the case that s and g are not pre-specified (like [15,1]), such points past which two guards never go were pursued. In this paper we do not assume the existence of *door* points on the boundary of the polygon. It is possible for two guards to go past the starting point or the ending point. Figure 1 depicts an example of a walk under our model. It is easily seen that there is no points s and g such that (P, s, g) is walkable under the Icking and Klein's model.

In this paper we focus on optimization problems and their algorithmic solutions. Our approach is to extend the framework previously used for feasibility test. It is somewhat counter-intuitive because the framework for feasibility test, described in Section 3, first gets rid of some geometric information which seems to be crucial for optimization solutions.

The first problem that we consider is to find a *shortest walk* in which the total length of the paths that two guards traverse is minimized (section 4). We present an $O(n^2 \log n)$ -time algorithm for an n -sided polygon. Second, we

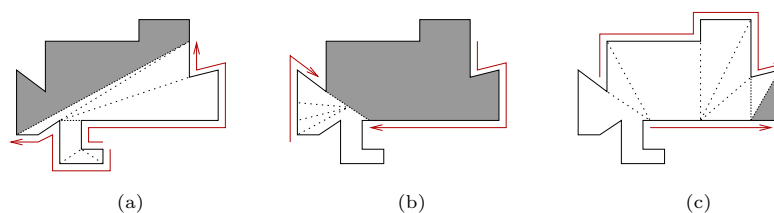


Fig. 1. Example of a walk. The bright region does not contain any undetected fugitive but the dark region does

study the problem of finding a *minimum diameter walk* in which the maximum distance between two guards is minimized (section 5). We present an $O(n^2)$ -time algorithm. To the knowledge of the authors, there have been no results on the above two problems. The techniques developed under Icking and Klein's model do not appear to be easily extended into our model.

Next we turn to another type of optimization problems, by allowing more than two guards. Here the guards should form a simple chain within the polygon in such a way that consecutive guards along the chain are mutually visible and the first and last guard have to move along the boundary but others do not. We are interested in a *marching* walk of the guards that sweeps a polygon. This problem was first studied by Efrat *et al.* [3]. They presented several algorithms: an $O(n^3)$ -time algorithm for computing the minimum number, say m^* , of guards required to sweep an n -sided polygon and an $O(n^4)$ -time algorithm for computing such a sweep schedule, an $O(n \log n)$ -time algorithm that approximates m^* with additive constant and an $O(n^2)$ -time algorithm for computing an integer value of at most $m^* + 2$.

We present an $O(n^2)$ -time algorithm for computing the minimum number of guards to sweep an n -sided polygon and an $O(n^3)$ -time algorithm for computing such a sweep schedule (section 6), which improve on the previous algorithms of time complexities $O(n^3)$ and $O(n^4)$ respectively. Actually our algorithm is a modification of the basic framework for feasibility test and the previous approximation algorithm in [3].

Due to lack of space, we defer some proofs to the full version of the paper.

2 Preliminaries

Let ∂P denote the boundary of a simple polygon P in the plane. We assume that ∂P is of unit length and the real line $\mathbb{R}$ is embedded along ∂P ; As a point moves along ∂P clockwise, its coordinate in $\mathbb{R}$ increases. In this paper, we adopt the convention that a real number r corresponds to a position in $\mathbb{R}$ as well as a point in ∂P . Thus, any continuous function on $\mathbb{R}$ corresponds to a continuous tour on ∂P .

Definition 1

(i) Let $l(t)$ and $r(t)$ denote the positions of two guards at time $t \in [0, 1]$. A *walk* on P is a pair (l, r) of continuous paths such that (Figure 1):

$$\begin{aligned} l &: [0, 1] \rightarrow \mathbb{R}, \partial P & r &: [0, 1] \rightarrow \mathbb{R}, \partial P \\ l(0) &= r(0) & l(1) - r(1) &= 1 \\ 0 &< l(t) - r(t) < 1 & \text{for all } t &\in (0, 1) \\ l(t) &\text{ is visible from } r(t) & \text{for all } t &\in [0, 1] \end{aligned}$$

(ii) P is said to be *walkable* if it admits a walk.

For simplicity of notation, let $\mathcal{U} = \{u_i \mid i \in \mathbb{Z}_{2n}\}$ denote the set of vertices and edges of P numbered in clockwise order, where u_{2i-1} denotes the i -th vertex

and u_{2i} denotes the edge between two vertices u_{2i-1} and u_{2i+1} . Throughout this paper, we assume that all edges are *open*, that is, edge u_{2i} does not contain vertices u_{2i-1} and u_{2i+1} . The indices are computed modulo $2n$. Let $N(u_i)$ denote the set of *neighbors* of u_i , that is, $N(u_i) = \{u_{i-1}, u_{i+1}\}$.

In this paper, we use the standard definitions of visibility. Two points p and q are *visible* from each other, if the segment $\overline{pq}$ is entirely contained in P . An edge e is *visible* from an edge e' (respectively, from a point p), if some point in e is visible from some point in e' (respectively, from p). Given a point or an edge x , the set of points in P that are visible from x specifies the *visibility polygon* of x , denoted by $VP(x)$.

Remark The output format of a walk is not important but we fix it for concreteness. A walk is represented as a sequence of pairs of line segments. Each pair denotes the paths along which l and r move straight for a certain time interval. The dynamics (speed or the exact location) on each segment should be specified by a constant number of parameters. The output size is the length of output walk.

3 Basic Framework and Walkability Test

In this section we review the algorithmic framework that has been implicitly used in many previous results [9,10,8,3]. We are given a simple polygon P . The following framework gives an $O(n^2)$ -time algorithm for testing the walkability of P and for constructing a walk, if one exists. This framework will be refined in the subsequent sections.

To test the walkability of a polygon P and to construct a walk of two guards, we transform P into an undirected graph G . The basic idea is simple: G is a state-transition diagram such that a node (u_i, u_j) corresponds to a set of configurations in which one guard l lies in $u_i \in \mathcal{U}$ and the other guard r lies in $u_j \in \mathcal{U}$. Arcs in G represent state-transitions. By collecting all possible states and transitions, we can make a transition diagram, called *roadmap*. To help later extensions, we formally define it.

1. **[Roadmap]** Transform P into an *st*-graph $G = (V, E)$.

G is made on the grid $2n \times 2n$ so that the node set is $\{(u_i, u_j) \in \mathcal{U} \times \mathcal{U} \mid u_i \text{ is visible from } u_j, \text{ possibly } i = j\} \cup \{s, t\}$. A node (u_i, u_j) lies in (i, j) position of the grid. Two nodes are connected by an arc if and only if they are vertically or horizontally adjacent in the grid. We also connect the nodes on the boundary of G to the corresponding nodes on the other side of G (i.e., we “glue” together the top side of G to the bottom side of G , and the left side of G to the right side of G). This grid graph should be modified further. We note that the transition from (u_i, u_i) to (u_i, u_{i+1}) is invalid because l and r should be continuous and satisfy $l(t) \geq r(t)$. Thus we remove arcs between (u_i, u_i) and its right/lower neighbors in the grid for all i . In addition, when sweeping starts, two guards lie on the same point, so we add an arc from s to node (u_i, u_i) for every i . When sweeping

finishes, two guards lie on an (closed) edge,¹ so we add an arc from node (u_i, u_{i+1}) to t for every i . See Figure 2 a. Formally, the edge set is $\{\langle s, (u_i, u_i) \rangle | u_i \in \mathcal{U}\} \cup \{\langle (u_i, u_{i+1}), t \rangle | u_i \in \mathcal{U}\} \cup \{\langle (u_i, u_j), (u_i, x) \rangle | x \in N(u_j)\} \cup \{\langle (u_i, u_j), (x, u_j) \rangle | x \in N(u_i)\}$ excluding $\{\langle (u_i, u_i), (u_i, u_{i+1}) \rangle\}$ and $\{\langle (u_i, u_i), (u_{i-1}, u_i) \rangle\}$.

In what follows, we refer to polygon vertices as vertices and polygon edges as edges and graph vertices as *nodes* and graph edges as *arcs*.

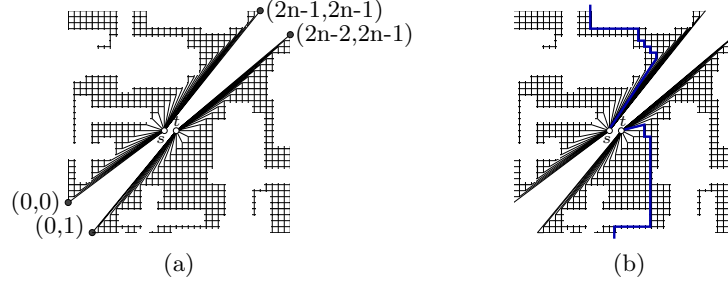


Fig. 2. (a) The roadmap of the polygon in Figure 1, where the leftmost lower corner vertex of P is u_1 . (b) The st -path corresponding to the walk in Figure 1

2. [**Reachability Query and Walk Construction**] If G contains an st -path, transform it to a walk and output it; Else output *Failure*.

One-to-one correspondence between an st -path in G and a walk in P is rather obvious from the definition of the roadmap. For example, the walk depicted in Figure 1 corresponds to the path in Figure 2b. We formally show it below. This proof is the most basic argument in this paper.

Lemma 1 P is walkable if and only if its roadmap G contains an st -path.

Proof. Suppose that P is walkable and let (l, r) be a walk in P . For a node $(u_i, u_j) \in V(G)$, if $l(t) \in u_i$ and $r(t) \in u_j$, we say that (l, r) lies in (u_i, u_j) at t . Consider the minimal subdivision $t_0(=0) < t_1 < \dots < t_M < 1$ of time interval $[0, 1]$ such that (l, r) lies in the same node during $[t_k, t_{k+1})$. Then, (l, r) can be represented as a sequence, σ , of M nodes in G . We attach s and t at both ends of σ ; Clearly σ is an st -path in G .

Now suppose $s, (x_0, y_0), \dots, (x_M, y_M), t$ be an st -path in G . We want to construct a walk. Let (p_k, q_k) , $0 \leq k \leq M$, be a pair of points such that $p_k \in x_k$ and $q_k \in y_k$ are mutually visible and assume $p_0 = q_0$. Note $x_0 = y_0$ and $y_M \in N(x_M)$. The following procedure generates a walk.

¹ Strictly speaking, when sweeping finishes, two guards must meet at one point. For notational convenience, we focus on the time instant at which two guards lie on one closed edge.

procedure Path-To-Walk

1. **for** $k = 0$ **to** M **do**
 $l(\frac{k}{M}) = p_k; r(\frac{k}{M}) = q_k;$
2. **for** $k = 0$ **to** $M - 1$ **do**
 Imagine a line $L(t)$ that rotates at some constant speed, for $t \in [\frac{k}{M}, \frac{k+1}{M}]$, in such a way that $L(\frac{k}{M})$ contains $\overline{p_k q_k}$ and $L(\frac{k+1}{M})$ contains $\overline{p_{k+1} q_{k+1}}$;
 $l(t) \leftarrow x_k \cap L(t)$ and $r(t) \leftarrow y_k \cap L(t)$, for $t \in (\frac{k}{M}, \frac{k+1}{M})$;

To show that the resulting (l, r) is a walk, it suffices to prove that $l(t)$ is visible from $r(t)$ for any $t \in [0, 1]$. Fix attention to a time interval $[t_k, t_{k+1}]$. If segments $\overline{p_k q_k}$ and $\overline{p_{k+1} q_{k+1}}$ intersect, we let c be the intersection point. In this case, no points of ∂P lie inside of the triangles $\triangle p_k p_{k+1} c$ and $\triangle q_k q_{k+1} c$ and the segment $l(t)r(t)$ is contained in the union of two triangles for any $t \in [t_k, t_{k+1}]$. Thus, $l(t)$ is visible from $r(t)$. If segments $\overline{p_k q_k}$ and $\overline{p_{k+1} q_{k+1}}$ do not intersect, the quadrangle $\square p_k p_{k+1} q_{k+1} q_k$ contains no points in ∂P . Since the segment $l(t)r(t)$ is contained in this quadrangle for any $t \in [t_k, t_{k+1}]$, $l(t)$ is visible from $r(t)$. Therefore, this lemma holds.

G can be constructed in $O(n^2)$ -time for an n -sided polygon P , basically by computing $VP(v)$ for each vertex v and $VP(e)$ for each edge e in linear time, from a triangulation of P [4]. Therefore, this theorem is immediate from the facts that reachability test can be done in time linear to the size of a graph and that the size of G is $O(n^2)$.

Theorem 2 *There is an $O(n^2)$ -time algorithm for finding a walk of an n -sided polygon.*

4 Finding a Shortest Walk

In this section we present an $O(n^2 \log n)$ -time algorithm for finding a shortest walk. Since we can test the walkability of a polygon P with the procedure in Section 3, we here assume that P is walkable. We will modify the framework given in the previous section. The main difficulty comes from the fact that during the construction of G , we removed most of geometric information but we need some geometric information in order to find a shortest walk.

Basic Idea. We make G_{sh} by modifying G . First, we divide each node (u_i, u_j) of G into four nodes depending on whether two guards l and r , respectively, have arrived at u_i and u_j from the clockwise neighbor or counterclockwise neighbor. We can view that each node in G_{sh} is associated with its *recent history*. Next, we connect two nodes by an arc if they have matching histories. Finally we assign a weight to each arc so that a shortest walk corresponds to a shortest st -path on G_{sh} .

1. **[Roadmap]** Transform P into an st -graph G_{sh} . We modify the basic roadmap G in Section 3. While G is an undirected graph, G_{sh} is a directed graph.

Each node (u_i, u_j) of G is replaced by four nodes, $(u_i, u_j) \times \{cw, ccw\} \times \{cw, ccw\}$ (see Figure 3a). The former $\{cw, ccw\}$ indicates whether l arrives at u_i from u_{i-1} (in case of cw) or u_{i+1} (in case of ccw). Analogously, the latter $\{cw, ccw\}$ indicates whether r arrives at u_j from u_{j-1} (in case of cw) or u_{j+1} (in case of ccw). Arcs should also be refined. There is an arc $\langle (u_i, u_j, f_1, f_2), (x, y, f'_1, f'_2) \rangle$ only if $\langle (u_i, u_j), (x, y) \rangle$ is an arc of G and their third and fourth fields *match* the transition. For example, when $x = u_i$ and $y = u_{j+1}$, the above arc exists only if $f_1 = f'_1$ and $f'_2 = cw$. Note that each node has at most four outgoing arcs (figure 3 b). To summarize,

- $V(G_{sh}) = \{s, t\} \cup [V(G)/\{s, t\}] \times \{cw, ccw\} \times \{cw, ccw\}$
- The set of (directed) arcs is $\{\langle s, (u_i, u_i, *, *) \rangle \mid u_i \in \mathcal{U}\}^2 \cup \{\langle (u_i, u_{i+1}, *, *), t \rangle \mid u_i \in \mathcal{U}\} \cup \{\langle (u_i, u_j, f, *), (u_i, u_{j+1}, f, cw) \rangle \mid i \neq j \text{ and } f \in \{cw, ccw\}\} \cup \{\langle (u_i, u_j, f, *), (u_i, u_{j-1}, f, ccw) \rangle \mid j \neq i+1 \text{ and } f \in \{cw, ccw\}\} \cup \{\langle (u_i, u_j, *, f), (u_{i+1}, u_j, cw, f) \rangle \mid j \neq i+1 \text{ and } f \in \{cw, ccw\}\} \cup \{\langle (u_i, u_j, *, f), (u_{i-1}, u_j, ccw, f) \rangle \mid i \neq j \text{ and } f \in \{cw, ccw\}\}.$

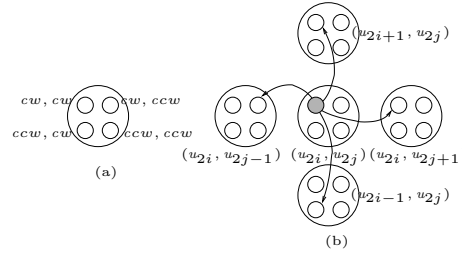


Fig. 3. (a) Each grid node in Figure 2 is quadrupled into four nodes. (b) Each node in G_{sh} has at most four outgoing arcs

2. [Weighting G_{sh}] Assign weights to arcs in G_{sh} .

Let us describe how to assign weights to arcs in G_{sh} . First we fix the *base positions* of each node in G_{sh} , with respect to which arc weights are calculated. For a node (u_i, u_j, ccw, ccw) (resp. (u_i, u_j, cw, cw) , (u_i, u_j, ccw, cw) , (u_i, u_j, cw, ccw)), the base positions are defined as (p, q) where p is the most clockwise (resp. counterclockwise, clockwise, counterclockwise) point on u_i that is visible from u_j , and q is the most clockwise (resp. counterclockwise, counterclockwise, clockwise) point on u_j that is visible from u_i . We use $base(v)$ to denote a pair of base positions of $v \in V(G_{sh})$. As an example, see Figure 4. Observe that $base(u_{2i}, u_{2j}, cw, cw) = (u_{2i-1}, u_{2j-1})$ and $base(u_{2i}, u_{2j}, ccw, ccw) = (u_{2i+1}, u_{2j})$.

There is one problem not mentioned above. The base positions of some node may be invisible from each other. For example, consider $(u_{2i}, u_{2j}, cw, ccw)$ in

² * indicates *don't care* condition, which means any of cw and ccw is valid.

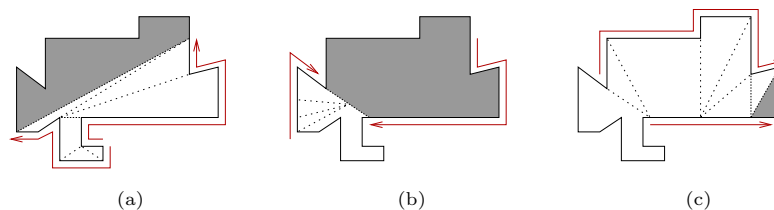


Fig. 4. The definition of the base positions of $(u_{2i}, u_{2j}, *, *)$. Two points in the base positions of each node are connected by a dashed segment, except for detour node $(u_{2i}, u_{2j}, cw, ccw)$

Figure 4. Its base positions, which are (u_{2i-1}, q) , consists of mutually invisible points. Such nodes are called *detour nodes*. It is easily seen that detour nodes are of the form $(u_{2i}, u_{2j}, ccw, cw)$ or $(u_{2i}, u_{2j}, cw, ccw)$. We can show that any shortest path never passes detour nodes (Lemma 3).

Lemma 3 *Any shortest walk never passes detour nodes (when we consider the mapping of a walk into an st -path in G_{sh} as in Lemma 1).*

Thus we can rule out the detour nodes in G_{sh} and so delete them. It is easily seen that base positions of all other nodes are *well-defined*, that is, are mutually visible.

For two points p and q , let $dist(p, q)$ denote the Euclidean distance between p and q . The weight $w(a)$ of an arc $a = \langle u, v \rangle$ is defined as follows: If $u = s$, let $w(a) = dist(base(v))$; if $v = t$, let $w(a) = dist(base(u))$; otherwise, let $w(a) = dist(p, p') + dist(q, q')$ where $base(u) = (p, q)$ and $base(v) = (p', q')$.

3. [Shortest Path] Find a shortest st -path in G_{sh} and convert it into a walk.

It remains to show that a shortest walk corresponds to a shortest st -path in G_{sh} . It is not difficult to show the next lemma using the procedure **Path-To-Walk**.

Lemma 4 *Any st -path in G_{sh} can be converted into a walk of the same length.*

Lemma 5 *Let (l, r) be a shortest walk. The length of (l, r) is no smaller than that of a shortest st -path in G_{sh} .*

Suppose (l, r) is a shortest walk of P . Without loss of generality, we assume that both $l(0)$ and $r(0)$ lie in u_i , l deviates u_i before r does, and that at the time instant that r deviates u_i first, l and r lie in u_j and u_k , respectively.

As in Section 3, we represent (l, r) as a sequence of nodes in G_{sh} : $s, (x_0, y_0, f_0, f'_0), \dots, (x_M, y_M, f_M, f'_M), t$ such that $0 \geq t_0 < \dots < t_M < 1$ and (l, r) lies in (x_k, y_k, f_k, f'_k) during $[t_k, t_{k+1})$. Our proof proceeds by induction on k . We should be cautious because some initial values of f'_k are not fixed until r deviates u_i first. Thus the base case of our induction is not $k = 1$ but $k = c$ for c such that f'_c is fixed for the first time.

Let L_t denote the total length of the paths that l and r traverse in the time interval $[0, t]$ and let $|SP(x_k, y_k, f_k, f'_k)|$ denote the length of a shortest path on G_{sh} from s to the node (x_k, y_k, f_k, f'_k) in G_{sh} . Let (p_k, q_k) denote the base positions of node (x_k, y_k, f_k, f'_k) . Then, we claim that

Claim. Assume that $l(t) = p \in x_k$ and $r(t) = q \in y_k$ and $(l(t), r(t))$ is mapped to (x_k, y_k, f_k, f'_k) in G_{sh} . Then L_t is at least $|SP(x_k, y_k, f_k, f'_k)| + \text{dist}(p, p_k) + \text{dist}(q, q_k)$.

It is not difficult to prove the base case of $k = c$. Afterwards we prove this claim by rather tedious but straightforward case analyses according to the *cw/ccw* fields of two consecutive nodes. Details are omitted in this abstract.

□

Therefore this lemma holds.

The construction of G_{sh} is nearly same as that of G in Theorem 2; We only have to calculate the base positions additionally. Next, it suffices to find a shortest path in G_{sh} by the well-known Dijkstra's algorithm whose time complexity is $O(n^2 \log n)$ for a graph with $O(n^2)$ nodes and $O(n^2)$ arcs.

Theorem 6 *There is an $O(n^2 \log n)$ -time algorithm for computing a shortest walk.*

5 Finding a Minimum Diameter Walk

The *diameter* of a walk (l, r) is defined to be $\max_{t \in [0, 1]} \text{dist}(l(t), r(t))$. In this section, we present an $O(n^2)$ -time algorithm for finding a minimum diameter walk. We assume that a polygon P is walkable.

1. **[Roadmap]** Transform P into an *st*-graph G_{dia} .

G_{dia} is same as G in Section 3, except that each node in G_{dia} has a weight. For a node $v = (u_i, u_j) \in V(G_{dia})$, $d(u_i, u_j)$ is defined to be $\inf_{p \in u_i, q \in u_j} \text{dist}(p, q)$ for mutually visible points p, q , where dist is the Euclidean distance between two points. For an *st*-path S , its *diameter* is $\max_{v \in S} d(v)$. The *diameter* of G_{dia} is the minimum over the diameters of all *st*-paths.

2. **[Min-Weight Path]** Find a minimum diameter *st*-path in G_{dia} and convert it into a walk.

Lemma 7 *The diameter of any walk is at least the diameter of G_{dia} . Any *st*-path in G_{dia} can be converted into a walk of the same diameter.*

The construction of G_{dia} is nearly same as that of G . Afterwards, it suffices to find a min-weight *st*-path, which can be computed in $O(n^2)$ -time using the well-known Dijkstra's algorithm.

Theorem 8 *There is an $O(n^2)$ -time algorithm for computing a minimum diameter walk.*

6 Finding a Walk for a Chain of Guards

As a generalization of two guards, we define a chain of guards.

Definition 2

(i) A *walk of m -chain guards* on P is an m -tuple $(l_1, \dots, l_m)$ of continuous functions such that:

$$\begin{aligned} l_1, l_m : [0, 1] &\rightarrow \mathbb{R}, \partial P & l_2, \dots, l_{m-1} : [0, 1] &\rightarrow P \\ l_1(0) = l_2(0) = \dots = l_m(0) & & l_1(1) - l_m(1) &= 1 \\ 0 < l_1(t) - l_m(t) < 1 & \text{ for all } t \in (0, 1) \\ l_i(t) \text{ and } l_{i+1}(t) & \text{ are mutually visible for all } 1 \leq i \leq m-1 \\ & \text{and } t \in [0, 1] \end{aligned}$$

(ii) P is said to be *m -walkable* if it admits an m -chain walk.

We present an $O(n^2)$ -time algorithm for finding the minimum number, m^* , such that P is m^* -walkable and an $O(n^3)$ -time algorithm for computing a walk of size $O(m^*n^2)$ for an n -sided polygon. Our algorithm is a modification of the basic framework given in Section 3 and an approximation algorithm described in [3].

Given two points $p, q \in P$, a *minimum-link path* between p and q is a piecewise-linear path between p and q that is contained in P and has the minimum number of line segments; the link distance $d_L(p, q)$ between p and q is the number of line segments in this path. (Note that $d_L(p, q) + 1$ guards can form a valid configuration with two end-guards being at p and q .) Given two points $p, q \in P$ that are mutually visible, let l be the line passing p and q . Then the *extension* of (p, q) is the connected component of $l \cap P$ that contains the segment pq .

The *window partition* $\mathcal{U}_p$ of a point $p \in P$ is a partition of P into maximal regions of constant link distance from p . An edge of $\mathcal{U}_p$ is either a portion of an edge of P or is a segment that separates two regions of $\mathcal{U}_p$; we call such a segment a *window* of $\mathcal{U}_p$. Suri [13] introduced the notion of window partition and showed that it can be constructed in time and space $O(n)$. The definitions of window partition extend naturally to the case when the source is a line segment, instead of a point. We can use the window partition $\mathcal{U}_p$ to compute a min-link path from p to any other point in P . In general, min-link paths are not unique. The *canonical min-link path* between $p \in P$ and $q \in P$ is a path that uses only extensions of windows in $\mathcal{U}_p$, with the last link chosen to pass through the last turning point of the shortest path between p and q within P .

Here $d_L(u_i, u_j)$ denotes $\min_{p \in u_i, q \in u_j} d_L(p, q)$ and $\pi_L(u_i, u_j)$ denotes a min-link path with link-length $d_L(u_i, u_j)$. Define a $2n \times 2n$ matrix $\mathcal{M}$, where $\mathcal{M}_{ij}$ is $d_L(u_i, u_j)$. The matrix $\mathcal{M}$ can be computed in $O(n^2)$, by computing the link distance from u_i to all other vertices and edges in $O(n)$ -time [13, 3].

1. **[Roadmap]** Transform P into an st -graph G_m .

The node set of G_m is $\{s, t\} \cup \mathcal{U} \times \mathcal{U}$. Each node (u_i, u_j) has weight $d_L(u_i, u_j)$. The arc set is defined by the same rule as in G ; it contains $\{\langle s, (u_i, u_i) \rangle \mid u_i \in \mathcal{U}\} \cup \{\langle (u_i, u_{i+1}), t \rangle \mid u_i \in \mathcal{U}\} \cup \{\langle (u_i, u_j), (u_i, x) \rangle \mid x \in N(u_j)\} \cup \{\langle (u_i, u_j), (x, u_j) \rangle \mid x \in N(u_i)\}$ excluding $\{\langle (u_i, u_i), (u_i, u_{i+1}) \rangle\}$ and $\{\langle (u_i, u_i), (u_{i-1}, u_i) \rangle\}$.

2. **[Min-Weight st -Path]** Find a min-weight st -path in G_m and transform it into a walk of chain guards.

It is easy to verify that a walk for P can be interpreted as an st -path σ in G_m as in Lemma 1, so that the maximum weight of nodes along σ plus one is the number of guards needed to sweep P .

On the other hand, an st -path σ in G_m such that the maximum weight along σ is w can be transformed into an $(w + 1)$ -chain walk (consisting of w segments). More exactly we map each node (x, y) in σ into a valid configuration on the canonical min-link path from a point on x to a point on y and morph two consecutive configurations along σ using Lemma 9.

Lemma 9 *Let π_1 and π_2 be two canonical min-link paths such that π_i connects p_i to p'_i ($i = 1, 2$) and p_1 and p_2 are mutually visible and p'_1 and p'_2 are mutually visible. Let m_i denote the link-length of π_i . Then, we can morph π_1 to π_2 using at most $m(= \max(m_1, m_2)) + 1$ guards. Moreover, we can compute in $O(n)$ -time a morphing strategy that issues $O(m)$ commands to guards.*

Let $\pi_i = (p_{0i}, p_{1i}, \dots, p_{m_i i})$ for $i = 1, 2$. The proof proceeds by induction on m . If $m = 1$, the desired morphing is exactly same as in **Path-To-Walk**. Depending on the number of intersections, r , between π_1 and π_2 , there are several cases. We can easily analyze when $r = 0, 1$ by case analysis. Here we focus on the case that $r \geq 2$. It is easily seen using the notion of window that the first segment of π_1 intersects that of π_2 . Suppose that the suffix chain of π_1 starting from p_{11} will enter $P \setminus VP(p_{02})$ (other cases are symmetric or very simple). See Figure 5. Let Q be the intersection. If p_{11} and p_{12} are mutually visible (Figure 5a), we can morph the first segment of π_1 into that of π_2 in parallel with the morph of the remaining chain of π_1 into that of π_2 ; The latter morph is possible by induction hypothesis. Thus assume otherwise, that is, p_{11} and p_{12} are invisible from each other (Figure 5b). It is easily seen that Q lies on the second (not third, fourth, $\dots$) segment of π_1 . Then we first morph segment $\overline{p_{01}p_{11}}$ into $\overline{p_{02}, Q}$ and then morph the suffix chain of π_1 starting from Q into the suffix chain of π_2 starting from p_{12} , which is possible by induction hypothesis. Details are omitted in this abstract.

We can show that the morphing is done with $O(m)$ commands of the guards, where each command is straight movements of the guards while the line through two guards rotating at some fixed speed. The algorithm for finding a morphing between two paths runs in $O(n)$ -time using window construction [13]. A min-weight path σ in G_m can be computed in $O(n^2)$ -time using Dijkstra's algorithm [3].

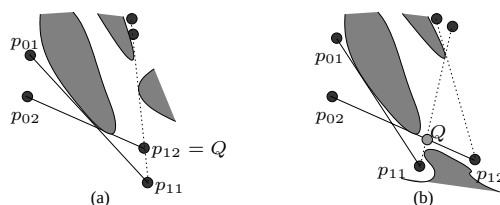


Fig. 5. Proof of Lemma 9

Theorem 10 *Given a simple polygon, one can compute m^* in $O(n^2)$ -time. Moreover, one can compute in $O(n^3)$ -time a sweeping strategy with $O(m^*n^2)$ commands issued to the guards.*

References

1. B. Bhattacharya, A. Mukhopadhyay, and G. Narasimhan. Two-guard walkability of simple polygons. In *Proc. of 7th WADS*, 2001. 481
2. D. Crass, I. Suzuki, and M. Yamashita. Searching for a mobile intruder in a corridor—the open edge variant of the polygon search problem. *Int. J. of Comp. Geom. and Appl.*, 5(4):397–412, 1995. 480, 481
3. A. Efrat, L. Guibas, S. Har-Peled, D. Lin, J. Mitchell, and T. Murali. Sweeping simple polygons with a chain of guards. In *Proc. of 16th Symp. on Discrete Algorithm*, pages 927–936, 2000. 481, 482, 483, 489, 490
4. L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygon. *Algorithmica*, 2:209–233, 1987. 485
5. L. J. Guibas, J. C. Latombe, S. M. Lavalle, D. Lin, and R. Motwani. A visibility-based pursuit-evasion problem. *Int. J. of Comp. Geom. and Appl.*, 9(4):471–493, 1999. 480
6. P. J. Heffernan. An optimal algorithm for the two-guard problem. *Int. J. of Comp. Geom. and Appl.*, 6(1):15–44, 1996. 481
7. C. Icking and R. Klein. The two guards problem. *Int. J. of Comp. Geom. and Appl.*, 2(3):257–285, 1992. 480, 481
8. S. M. LaValle, B. H. Simov, and G. Slutzki. An algorithm for searching a polygonal region with a flashlight. In *Proc. of 16th Symp. on Computational Geometry*, 2000. 481, 483
9. Jae-Ha Lee, Sang-Min Park, and Kyung-Yong Chwa. Searching a polygonal room with one door by a 1-searcher. *Int. J. of Comp. Geom. and Appl.*, 10(2), 2000. 481, 483
10. Jae-Ha Lee, Sung Yong Shin, and Kyung-Yong Chwa. Visibility-based pursuit-evasion in a polygonal room with a door. In *Proc. 15th ACM Sympos. on Computational Geometry*, pages 281–290, 1999. 480, 481, 483
11. G. Narashimhan. On hamiltonian triangulations in simple polygons. *Int. J. of Comp. Geom. and Appl.*, 9(3):261–275, 1999. 481
12. Sang-Min Park, Jae-Ha Lee, and Kyung-Yong Chwa. Visibility-based pursuit-evasion in a polygonal region by a single searcher. In *Proc. of ICALP*, pages 456–468, 2001. 480

13. S. Suri. On some link distance problems in a simple polygon. *IEEE Trans. Robot. Autom.*, 6:108–113, 1990. [489](#), [490](#)
14. I. Suzuki and M. Yamashita. Searching for a mobile intruder in a polygonal region. *SIAM J. Comp.*, 21(5):863–888, 1992. [480](#)
15. L. H. Tseng, P. Heffernan, and D. T. Lee. Two-guard walkability of simple polygons. *Int. J. of Comp. Geom. and Appl.*, 8:85–116, 1998. [481](#)

Approximation of a Geometric Set Covering Problem

Sofia Kovaleva and Frits C. R. Spieksma

Maastricht University, Department of Mathematics
P.O. Box 616, NL-6200 MD Maastricht, The Netherlands

Abstract. We consider a special set covering problem. This problem is a generalization of finding a minimum clique cover in an interval graph. When formulated as an integer program, the 0-1 constraint matrix of this integer program can be partitioned into an interval matrix and a special 0-1 matrix with a single 1 per column. We show that the value of this formulation is bounded by $\frac{2k}{k+1}$ times the value of the LP-relaxation, where k is the maximum row sum of the special matrix. For the "smallest" difficult case, i.e., $k = 2$, this bound is tight. Also we provide an $O(n)^{\frac{3}{2}}$ -approximation algorithm in case $k = 2$.

1 Introduction

Packing and covering problems admit a natural formulation as an integer program. When the constraint matrix of such an integer program is an interval matrix they constitute well-understood, basic problems in combinatorial optimization (see e.g. [6]). In this paper we investigate a covering problem with a constraint matrix that is the parallel composition of an interval matrix and special 0-1 matrix with a single 1 per column. More specifically, we consider the following integer programming formulation:

$$\begin{aligned} (P) \quad & \text{minimize } 1 \cdot y + 1 \cdot z \\ & \text{subject to} \\ & (y \mid z) \begin{pmatrix} A \\ D \end{pmatrix} \geq 1 \\ & y, z \in \{0, 1\} . \end{aligned}$$

In this formulation A is an 0-1 $r \times n$ submatrix with consecutive ones in the columns, D is a 0-1 $l \times n$ submatrix such that each column contains exactly one 1, y is a $1 \times r$ vector and z is a $1 \times l$ vector of variables. We assume that the constraint matrix has no zero rows. Further, let k be the maximum row sum of the matrix D .

An alternative, more geometrically oriented description of this problem is as follows. Given is a grid consisting of rows and columns. Each row contains at most k intervals, each of arbitrary integral length and placed at an arbitrary position. Consider now the following question. What is the minimum number

of rows and columns needed to stab all intervals? One easily verifies that these two descriptions are equivalent: given the constraint matrix, if $a_{ij} = 1$ then interval j is stabbed by column i , else it is not. Moreover those columns in the constraint matrix for which $d_{ij} = 1$ for some i correspond to intervals j that lie on a same row of the grid in the geometric description. Vice versa, given a grid with intervals it is straightforward to find the constraint matrix.

It is not hard to see that our problem is a special case of the well-known set covering problem: let the intervals correspond to elements in the ground set, and let a set of intervals that are on a same row or a set of intervals that share a column correspond to a set in the collection.

Example: Choose $r = 4, n = 3, l = 2$ and let $A = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$, and let $D = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$. This specifies an instance of (P) . Figure 1 contains the corresponding geometric description. Observe that the optimum value for this instance is 2.

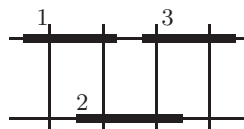


Fig. 1. An example

Finally, yet another way of viewing this problem is by posing it as a graph-theoretical problem. Indeed, given the grid with its intervals, construct a graph as follows. There is a node for each interval, and two nodes are connected if they share a column of the grid (blue edge) or if they are on the same row of the grid (red edge). Notice that an edge can be red as well as blue in case two intervals of a same row share a column (if one finds bicolored edges awkward, one could alternatively work with parallel edges of different colors). Thus the graph constructed is the edge-union of an interval graph (the blue edges) and a k -partite graph. The covering question is equivalent to finding a monochromatic minimum clique cover. Notice that in case $k = 1$ the covering question reduces to finding a minimum clique cover in an interval graph. Since interval graphs are perfect (see for instance [4]) this problem can be solved in polynomial time.

Applications. Many practical problems feature a constraint matrix of the form described in (P) . However, most applications involve a packing type of problem (see [2]). Applications of the covering problem seem less abundantly present in literature; however, the following type of situation leads naturally to instances

of (P). Each of a number of items (patients to receive treatment, products to undergo chemical processes, machines subject to inspection) has to undergo treatment on a regular basis. More precisely, for each item a set of intervals is given during which a treatment must take place. The treatment itself takes one time-unit and is provided by some kind of machine with unbounded capacity (that is, it can process any number of items), and consists of "turning the machine on" at some point in time, say t (this corresponds to selecting column t). Then the items corresponding to intervals that are stabbed by column t undergo the treatment. The objective is to minimize the number of times the machine is turned on plus the number of items not processed (an item is not processed when at least one of its intervals has not undergone the treatment (this corresponds to selecting the row corresponding to that item))(see [7]).

Related Results. In [8] it is proved that (P) is MAX SNP-hard for each fixed $k \geq 2$ (see [9] for an overview). This implies that, unless $\mathcal{P} = \mathcal{NP}$, optimum solution of (P) cannot be approximated arbitrarily closely in polynomial time (see [1]). Also, a 2-approximation algorithm is given for the covering problem with arbitrary right hand sides and arbitrary objective coefficients. Another special case of the set covering problem for which a constant approximation factor is achieved is described in [3]. They present a 2-approximation for special set cover instances which they call tree-representable. It is easy to verify that there exist instances of (P) that are not tree-representable.

Our Results. Given some instance I of (P), let $v_{LP}(I)$ be the value of the LP-relaxation of (P), and let $OPT(I)$ be the value of an optimum solution. We prove that $OPT(I) \leq \frac{2k}{k+1}v_{LP}(I)$ for all instances I (see Section 2). In particular, for the "smallest" difficult case of $k = 2$, this implies the existence of a $\frac{4}{3}$ -approximation algorithm. Further, we describe an $O(n)^{\frac{3}{2}}$ -approximation algorithm for the case $k = 2$ (see Section 3). Finally, we indicate direction in which the results can be generalized (see Section 4).

2 An Approximation Result

Theorem 1. $OPT(I) \leq \frac{2k}{k+1}v_{LP}(I)$ for all instances I .

Proof. The idea of the proof is as follows: we describe a way to round the value of each of the z -variables in an optimal LP-solution to either 0 or 1 while preserving feasibility of the finally constructed solution. Observe that the cost of a feasible LP-solution with integral z -variables is an upper bound for the cost of an optimal integral solution. Thus, if we can argue that the final solution with integral z -variables is feasible and its cost (denoted by v_C) is bounded by $\frac{2k}{k+1}$ times the cost of the LP-solution (denoted by v_{LP}), we are done.

We assume without loss of generality that the value of each variable in the LP-solution is a multiple of $\frac{1}{s}$ for some $s \in N$. We will refer to $\frac{1}{s}$ as a *unit*; this

enables us to say that (the value of) any particular variable consists of a number of units.

In our rounding procedure we make use of a so-called *doubling step*, which amounts to the following. Consider some positive z -variable with value z , and consider some positive y -variables with value $y_i, i = 1, 2, \dots$. Select one unit from this z -variable, and select r_i units from the y -variables with value $y_i, i = 1, 2, \dots$ such that $0 \leq \sum_i r_i \leq k$. Now, replace z by $z - \frac{1}{s}$, and replace y_i by $y_i + \frac{r_i}{s}$, $i = 1, 2, \dots$. In a sense we have doubled the values of the units of the y -variables involved; henceforth the particular units of the y -variables that are involved are now called *doubled* units. Notice that such a step replaces a weight of $\frac{\sum_i r_i + 1}{s}$ in the original LP-solution with $\frac{2 \sum_i r_i}{s}$.

Given an optimal LP-solution, we distinguish three types of z -variables:

- a z -variable is called *small* if $0 \leq z_i \leq \frac{1}{2}$; let $S = \{i \mid 0 \leq z_i \leq \frac{1}{2}\}$,
- a z -variable is called *medium* if $\frac{1}{2} < z_i < \frac{k+1}{2k}$; let $M = \{i \mid \frac{1}{2} < z_i < \frac{k+1}{2k}\}$, and
- a z -variable is called *large* if $\frac{k+1}{2k} \leq z_i$; let $L = \{i \mid \frac{k+1}{2k} \leq z_i\}$.

First, we deal with the large z -variables. We simply round each large z -variable up to 1. Observe that this costs no more than $\frac{2k}{k+1}$ times the weight of the original values. Indeed, we have:

$$|L| \leq \frac{2k}{k+1} \sum_{i \in L} z_i. \quad (1)$$

Second, we now sketch a procedure that deals with the small z -variables. Basically, this procedure selects an arbitrary small z -variable and by using the doubling step repeatedly, attempts to round the z -variable down to 0, while preserving feasibility of the constructed solution. Notice that a doubling step is only allowed to use units that are not doubled in previous steps. Of course, a doubling step performed to round down a specific z -variable has impact on other z -variables since a doubled unit may intersect intervals of other z -variables.

Lemma 1. *Consider any feasible LP-solution. If a z -variable is small, there exists a sequence of doubling steps that produces a feasible LP-solution such that each interval of this z -variable receives weight 1 from y -variables only.*

Proof. Consider the current weight induced by y -variables of the at most k intervals of a specific z -variable. For any such interval with weight received by y -variables less than 1, there must be an undoubled unit of a y -variable that intersects this interval (if all units were doubled, then, since $z_i \leq \frac{1}{2}$, this interval would have received at least 1 by the y -variables). Also, since we preserve feasibility, the current value of the z -variable is still positive. Since there can be at most k intervals that do not receive weight 1 by the y -variables, we can exhibit a doubling step that preserves feasibility. $\square$

Thus, when we select a small z -variable we can perform a sequence of doubling steps as described above to arrive at a situation where each interval of this z -variable receives weight 1 from y -variables only. Then, if the remaining value of the z -variable is still positive, we simply set it to 0. Thus, we repeatedly apply Lemma 2 to round down all small z -variables.

What about the costs induced by such a sequence of doubling steps? Let D denote the total number of doubled units after all small variables have been rounded down, and let L denote the number of remaining undoubled units. In the following lemma we bound the costs of the doubled units in the current solution.

Lemma 2.

$$2\frac{D}{s} \leq \frac{2k}{k+1}\left(\frac{D}{s} + \sum_{i \in S} z_i\right). \quad (2)$$

Proof. The analysis of a single doubling step is easy: we replace $\frac{\sum_i r_i + 1}{s}$ by $\frac{2\sum_i r_i}{s}$ while $0 \leq \sum_i r_i \leq k$. Obviously, we have that $\frac{2\sum_i r_i}{s} \leq \frac{2k}{k+1}\left(\frac{\sum_i r_i + 1}{s} + \frac{1}{s}\right)$. Since any doubling step uses only undoubled units, it follows that we can apply this analysis to a sequence of steps as well. $\square$

Third, let us now deal with the medium z -variables, and let us assume that there are p of them, i.e., $|M| = p$. We will round all these variables either to 0 or to 1, depending upon the value of L . To facilitate calculations we define:

$$Q = \frac{(k+1)p - 2k \sum_{i \in M} z_i}{k-1}. \quad (3)$$

Case 1: $\frac{L}{s} \geq Q$. In that case we claim that there are enough undoubled units in the current LP-solution to round up all medium z -variables. The following lemma bounds the cost of rounded up medium z -variables plus the weight of sQ undoubled units:

Lemma 3.

$$p + Q \leq \frac{2k}{k+1}\left(Q + \sum_{i \in M} z_i\right). \quad (4)$$

Proof. Observe that the left hand side reflects the value of the rounded up medium z -variables plus the weight induced by an amount of undoubled units. One easily verifies the correctness of this inequality: when substituting Q , both the left hand side and the right hand side simplify to $\frac{2k}{k-1}(p - \sum_{i \in M} z_i)$. $\square$

Case 2: $\frac{L}{s} < Q$. Now we round down all medium z -variables to 0, and double all hitherto undoubled units. Next, for each interval that does not receive weight 1 from y -variables, we simply select an y -variable that stabs this interval and increase its value to the required amount. We can bound the costs using the following Lemma:

Lemma 4.

$$\frac{2L}{s} + k(2 \sum_{i \in M} z_i - p) \leq \frac{2k}{k+1} \left(\sum_{i \in M} z_i + \frac{L}{s} \right). \quad (5)$$

Proof. Observe that the first term on the left hand side of this inequality corresponds to the value of the doubled remaining units. The second term is an upper bound on the total increase of the y -variables. This can be explained as follows: since all units of all y -variables are doubled, we know that each interval receives at least $2(1 - z_i)$. Thus, the total amount that is still required is $k \cdot \sum_{i \in M} (1 - (2 - 2z_i)) = k(2 \sum_{i \in M} z_i - p)$, where the first k comes from the fact that there are k intervals for each variable. Let us now prove the inequality. First, we claim that

$$\frac{2L}{s} \leq \frac{2k}{k+1} \left(\frac{L}{s} + \frac{(k+1)p - 2k \sum_{i \in M} z_i}{k(k-1)} \right). \quad (6)$$

Indeed, since $\frac{2L}{s} = \frac{2k}{k+1} \left(\frac{L}{s} + \frac{L}{ks} \right)$, and, since $\frac{L}{ks} < \frac{Q}{k} = \frac{(k+1)p - 2k \sum_{i \in M} z_i}{k(k-1)}$ (we are in Case 2), (6) is true. Further we claim that

$$k(2 \sum_{i \in M} z_i - p) \leq \frac{2k}{k+1} \left(\sum_{i \in M} z_i - \frac{(k+1)p - 2k \sum_{i \in M} z_i}{k(k-1)} \right). \quad (7)$$

The right hand side of (7) simplifies to

$$\frac{2k}{k-1} \sum_{i \in M} z_i - \frac{2p}{k-1}. \quad (8)$$

Then the question is whether the following inequality is true:

$$\begin{aligned} \frac{2k}{k-1} \sum_{i \in M} z_i - \frac{2p}{k-1} &\geq k(2 \sum_{i \in M} z_i - p) \Leftrightarrow \\ p(k - \frac{2}{k-1}) &\geq 2k \sum_{i \in M} z_i (1 - \frac{1}{k-1}) \Leftrightarrow \\ p(k^2 - k - 2) &\geq 2k \sum_{i \in M} z_i (k-2) \Leftrightarrow \\ p(k-2)(k+1) &\geq 2k(k-2) \sum_{i \in M} z_i. \end{aligned}$$

Since by definition of a medium z -variable we have $\sum_{i \in M} z_i \leq p \frac{k+1}{2k}$ the last inequality holds and thus (7) is true. Next, inequalities (6) and (7) imply the lemma. $\square$

Finally, let us argue that v_C is bounded by $\frac{2k}{k+1}$ times v_{LP} . In Case 1 we get using (1), Lemma's 3 and 4:

$$v_C = |L| + \frac{2D}{s} + p + \frac{L}{s} = |L| + \frac{2D}{s} + p + Q + \left(\frac{L}{s} - Q \right) \leq$$

$$\begin{aligned} & \frac{2k}{k+1} \left(\sum_{i \in L} z_i + \frac{D}{s} + \sum_{i \in S} z_i + Q + \sum_{i \in M} z_i \right) + \left(\frac{L}{s} - Q \right) \leq \\ & \frac{2k}{k+1} \left(\sum_{i \in L} z_i + \frac{D}{s} + \sum_{i \in S} z_i + Q + \sum_{i \in M} z_i + \frac{L}{s} - Q \right) \leq \frac{2k}{k+1} v_{LP} . \end{aligned}$$

Similarly, in Case 2, we get using (1), Lemma's 3 and 5:

$$\begin{aligned} v_C &= |L| + \frac{2D}{s} + \frac{2L}{s} + k \left(2 \sum_{i \in M} z_i - p \right) \leq \\ & \frac{2k}{k+1} \left(\sum_{i \in L} z_i + \frac{D}{s} + \sum_{i \in S} z_i + \sum_{i \in M} z_i + \frac{L}{s} \right) \leq \frac{2k}{k+1} v_{LP} . \end{aligned}$$

This proves Theorem 1. $\square$

An interesting question concerns the tightness of this result. At least for $k = 2$ the result is tight as can be verified by considering the example. The value of the LP-relaxation equals $\frac{3}{2}$, whereas the value of an optimal solution equals 2, implying that the bound of Theorem 1 is tight for $k = 2$.

3 An $O(n)$ Heuristic

Of course, to apply the approach described in the previous section, we need as input the values of the variables in an optimal LP-solution. Since computing these values can be time-consuming (albeit polynomial), it is of interest to investigate algorithms that do not need the LP-solution.

In this section we sketch such an algorithm for the case $k = 2$. In particular, we describe an $O(n)$ algorithm that finds a solution with a cost guaranteed to be no more than $\frac{3}{2}$ times the cost of an optimal solution.

Let us again consider formulation (P), while assuming that each row of submatrix D contains exactly two 1's. (This can be accomplished without loss of generality; since $k = 2$ each row of D has at most two 1's, and if a row of D contains a single 1, we simply copy the associated column: this is akin to duplicating an interval in the geometric description).

The following type of relaxation is proposed in a general framework in [5]. By introducing variables z_{i1} and z_{i2} , replace each z_i in the objective of (P) by $(z_{i1} + z_{i2})/2$, replace one (of the 2) occurrence of z_i in the constraints of (P) by z_{i1} and the other by z_{i2} , and finally replace the integrality constraints in (P) by appropriate upper bound constraints, we obtain a relaxation of (P) called (PREL). Observe that the constraint matrix of (PREL) is totally unimodular, and thus any LP-algorithm outputs integral values for all the y - and all the z_{ij} -variables, $j = 1, 2$. Even more, it is not difficult to verify that the constraint matrix is a so-called *greedy* matrix (recall that a matrix is greedy if by permuting columns it can be brought into the *standard greedy form*, i.e. not containing an induced submatrix of this form: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$, see [10]). Results in [11] imply that there is an $O(n)$ algorithm to compute an optimal solution of the problem (PREL).

Now, given an instance I of (P) with $k = 2$, let $OPT(I)$ be the value of the corresponding optimal solution and let $v_{PREL}(I)$ be the value of the optimal solution of model (PREL). Further, let the corresponding solution in (PREL) be described by $\hat{y}, \hat{z}_{ij}$ (found by the greedy algorithm of [11]). Set $\hat{z}_i = (\hat{z}_{i1} + \hat{z}_{i2})/2$ for all $i = 1, \dots, l$, and define $Z_1 = \{\hat{z}_i \mid \hat{z}_i = 1, i = 1, \dots, l\}$ and $Z_{\frac{1}{2}} = \{\hat{z}_i \mid \hat{z}_i = \frac{1}{2}, i = 1, \dots, l\}$. We have:

$$v_{PREL}(I) = \sum_{j=1}^r \hat{y}_j + |Z_1| + \frac{1}{2}|Z_{\frac{1}{2}}| \leq OPT(I) . \quad (9)$$

Observe that one can think of a $\hat{z}_{ij}$ variable as corresponding to some single interval. Consider the set of $\hat{z}_{ij}$ variables that are equal to 1. The claim is that this set of variables corresponds to a set of non-intersecting intervals. Indeed, if a pair would intersect, there is a y -variable which can be set to 1, so that both z_{ij} -variables can get value 0 and the value of the objective function remains unaltered. (Notice that this can be accomplished in $O(n)$ time). Thus, since no two intervals in $Z_{\frac{1}{2}}$ share a common column or row in the grid, it is obvious that any feasible, and in particular optimal solution of I satisfies

$$|Z_{\frac{1}{2}}| \leq OPT(I) . \quad (10)$$

Let us introduce heuristic H_{PREL} for (P) with $k = 2$: input I , solve (PREL), compute the $\hat{z}$ -variables as described above and round them up to the nearest integer. Then:

$$\begin{aligned} c(H_{PREL}(I)) &= \sum_{j=1}^r \hat{y}_j + \sum_{i:\hat{z}_i=1} \hat{z}_i + 2 \sum_{i:\hat{z}_i=1/2} \hat{z}_i = \\ &= \sum_{j=1}^r \hat{y}_j + |Z_1| + |Z_{\frac{1}{2}}| \leq \frac{3}{2}OPT(I) , \end{aligned}$$

using the 2 inequalities from above, which shows that H_{PREL} is a $\frac{3}{2}$ -approximation algorithm for (P) with $k = 2$.

The instance given by the figure 2 shows that the analysis of this heuristic is tight.

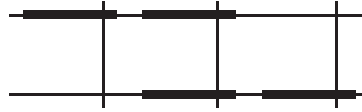


Fig. 2. An instance proving tightness of the analysis

4 Extensions

Here we shortly mention 2 extensions.

- The approach in Section 2 can be generalized to deal with arbitrary integral right hand sides.
- The algorithm described in Section 3 can be extended to deal with arbitrary values of k at the expense of the approximation factor. Indeed (while omitting details) we claim that a straightforward generalization of this approach achieves a ratio of $1 + \frac{k}{4}$ if k is even and $1 + \frac{k}{4} - \frac{1}{4k}$ if k is odd.

References

1. Arora, S., C. Lund, R. Motwani, M. Sudan, and M. Szegedy (1992), 'Proof verification and hardness of approximation problems', Proceedings of the 33rd IEEE Symposium on the Foundations of Computer Science, 14-23.
2. Bar-Noy, A., S. Gupta, J. Naor and B. Schieber (1999), "Approximating the Throughput of Multiple Machines in Real-Time Scheduling", Proceedings of the 31st ACM Symposium on Theory of Computing.
3. Garg, N., V. V. Vazirani and M. Yannakakis (1997), "Primal-Dual Approximation Algorithms for Integral Flow and Multicut in Trees", *Algorithmica* 18, 3-20.
4. Golumbic, M.C., *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, San Diego, California, 1980.
5. Hochbaum, D. S. (1998), "Instant Recognition of Half Integrality and 2-Approximations", in: Proceedings of the APPROX'98 Conference, editors: Klaus Jansen and José Rolim, Lecture Notes in Computer Science 1444, 99-110.
6. Schrijver, A., *Theory of Linear and Integer Programming*, John Wiley & Sons, 1998.
7. Aerts, J., Marinissen, E. J. (1998), "Scan chain design for test time reduction in core-based ICs", Paper 18.1 at International test conference, Washington, D. C., October 1998.
8. Kovaleva, S., Spieksma, F. C. R. (2001), "Primal-dual approximation algorithms for a packing-covering pair of problems", Report M 01-01, Department of Mathematics, Maastricht University, P. O.Box 616, NL-6200 MD, Maastricht, The Netherlands.
9. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., Protasi, M., *Complexity and Approximation*, Springer-Verlag, Berlin, 1999.
10. Nemhauser, G. L., Wolsey, L. A., *Integer and Combinatorial Optimization*, Wiley & Sons, 1999.
11. Hoffman, A. J., Kolen, A. W. J., Sakarovitch, M., (1985), "Totally-balanced and greedy matrices", *SIAM Journal of Algebraic and Discrete Methods* 6, 721-730.

Shortest Path Algorithms: Engineering Aspects

Andrew V. Goldberg

STAR Lab., InterTrust Technologies Corp.
4750 Patrick Henry Dr., Santa Clara, CA 95054, USA
goldberg@intertrust.com

Abstract. We review shortest path algorithms based on the multi-level bucket data structure [6] and discuss the interplay between theory and engineering choices that leads to efficient implementations. Our experimental results suggest that the caliber heuristic [17] and adaptive parameter selection give an efficient algorithm, both on typical and on hard inputs, for a wide range of arc lengths.

1 Introduction

The three most common ways to evaluate algorithm performance are worst-case analysis, average-case analysis, and experimental evaluation. The most effective way to evaluate an algorithm is to use all three approaches. Good algorithm engineering combines theoretically justified ideas, common-sense heuristics, and experimental feedback to develop an efficient and robust code.

In this paper we study the important problem of finding shortest paths from a source to all other vertices in a directed graph with nonnegative arc lengths (the *NSP problem*). This problem has been well-studied. The worst-case bounds for the problem have a long history; see e.g., [1,2,5,6,7,9,10,14,19,20,24,25,26,28,29,30]. The currently best bounds are near-linear. Let n and m denote the number of vertices and arcs in the input graph, respectively. If the input arc lengths are integral, let U denote the maximum arc length. Let C denote the ratio between the biggest and the smallest nonzero arc lengths. In the pointer model of computation, one can get an $O(m + n \log n)$ time bound [13]. In a RAM model with word operations, the fastest currently known algorithms achieve the following bounds: $O(m + n(\log U \log \log U)^{1/3})$ [26], $O(m + n(\sqrt{\log n}))$ [25], $O(m \log \log U)$ [19], and $O(m \log \log n)$ [29]. In the special case when the graph is undirected, Thorup's algorithm [28] runs in linear time.

The average-case results for the NSP problem are interesting because they apply to natural input distributions or to potentially practical algorithms. In particular, Noshita [23] shows that under relatively weak assumptions on the input distribution, the average-case bound on the binary heap implementation of Dijkstra's algorithm is better than the worst-case bound. Mayer [21] shows that the problem can be solved in linear average time if input arc lengths are independent and uniformly distributed. Goldberg [17] shows that a simple modification of the algorithm of [6] yields an algorithm with average running time

on the uniform arc length distribution (without the independence assumption). Below we refer to the algorithm of [17] as the *smart queue* algorithm.

Computational work on the NSP problem has received a lot of attention as well; see, e.g., [3,4,8,15,18,16,31]. This work leads to the general agreement that the problem can be solved well in practice. However, this work does not show how close existing NSP algorithm implementations are to a practical lower bound.

We focus our attention on NSP algorithms based on the *multi-level bucket (MLB)* data structure and its variants. This data structure was originally proposed by Denardo and Fox [6]. Enhancements to it have been proposed in [1,5]. For a long time, the folklore, originating from the original paper [6] and the fact that computer memories were small, was that the MLB data structure was not competitive in practice. The work of Cherkassky et al. [4] and the followup work of [5,18] suggest that two- or three-level MLB variants are competitive on many problems.

In this paper we review the MLB data structure and discuss an implementation of the MLB data structure that takes advantage of the newer theoretical results and improves upon previous implementations. We also discuss an implementation of the smart queue version of the algorithm. Our experimental results show that the use of smart queues makes the MLB algorithm with a large number of levels practical. In particular, an implementation with the number of levels optimized for the worst-case theoretical performance works well on both typical and bad-case inputs. For example, for 32-bit arc lengths, the code runs in time less than 2.5 times that of breadth-first search, even on the hardest problems we were able to construct. Our results lead to better understanding of NSP algorithm implementations and show how close their performance is to the lower bound provided by breadth-first search.

2 The Multi-level Bucket Data Structure

Assuming that the reader is familiar with the labeling method [11,12] (see also [27]) and Dijkstra's algorithm [9], we review the MLB algorithm and related results. For more details, see [5,6,17].

The MLB structure B implements the priority queue operations **insert**, **decrease-key**, and **extract-min**. The bucket structure has two parameters, the number of levels k and the base Δ . A k -level structure contains k regular levels and a special (top) level. Given an input graph with maximum arc length U , the two parameters are related by $k = \lceil \log_{\Delta} U \rceil$. Except for the top level, a level contains Δ buckets. Conceptually, the top level contains infinitely many buckets. However, one can show that at most two consecutive top-level buckets can be nonempty at any given time, and that one can maintain only these buckets. We denote bucket j at level i by $B(i, j)$; i ranges from 0 (bottom level) to k (top), and j ranges from 0 to $\Delta - 1$. A bucket contains a set of vertices maintained as a doubly linked list.

We maintain μ such that μ is a lower bound on the distance labels of labeled vertices. Initially $\mu = 0$. Every time an **extract-min** operation removes a vertex v from B , we set $\mu = d(v)$, the distance label of v . Consider the base- Δ representation of the distance labels and number digit positions starting from 0 for the least significant digit. One can show that μ and the $k+1$ least significant digits of the base- Δ representation of $d(u)$ uniquely determine $d(u)$.

Next we define the position of a vertex u in B with respect to μ . Let μ' and $d'(u)$ be μ and $d(u)$, respectively, truncated to the $k+1$ least significant digits. Let i be the index of the most significant digit in which μ' and $d'(u)$ differ, or 0 if they match. Let j be the digit of $d(u)$ in position i . Then we say that the *position of u in B* is (i, j) . When u is inserted into B , it is inserted into $B(i, j)$. For each vertex in B , we store its position.

Each bucket $B(i, j)$ corresponds to a range of values that depends on μ . Suppose the position of u in B is (i, j) . We say that u belongs to the range of $B(i', j')$ if $(i', j') = (i, j)$ or if $i' > i$. The *width* of a bucket at level i is equal to Δ^i : the bucket contains Δ^i distinct values.

Given μ , one can compute the position of a vertex u from $d(u)$ in constant time. This gives a simple constant-time implementation of **insert** and **decrease-key** operations. To implement the **extract-min** operation, we find the lowest nonempty level i and the first nonempty bucket $B(i, j)$ on this level. We do the latter by starting from a bucket whose range contains μ and scanning consecutive buckets until a nonempty one is found. If the level is the bottom level, we delete and return a vertex from $B(i, j)$. Otherwise we *expand* the bucket by finding and deleting a vertex with the smallest label in the bucket, setting μ to the distance label of this vertex, and moving remaining vertices from $B(i, j)$ to their new locations. An important fact is that the vertices always move to *lower* levels.

The analysis of the algorithm amortizes all work over vertex scans except for the work of scanning empty buckets and the work involved in moving vertices during the bucket expansion operation.

Theorem 1. *The algorithm runs in $O(m + n + \rho + \phi)$ time, where ρ is the number of empty bucket scans and ϕ is the number of times vertices move from a bucket to a lower level bucket.*

If we charge scanning empty buckets to the vertex the algorithm finds and scans immediately afterwards, we get an $O(n\Delta)$ bound on ρ . The fact that vertices move to lower levels of B implies an $O(kn)$ bound on ϕ . To improve efficiency, one has to balance ρ and ϕ . From the worst-case analysis point of view, setting $\Delta = \Theta(k)$ minimizes the running time. If $\Delta = k = \Theta\left(\frac{\log U}{\log \log U}\right)$, then the algorithm runs in $O(m + n \frac{\log U}{\log \log U})$ time.

Next we discuss two heuristics that improve the MLB algorithm performance. We define the *caliber* of a vertex v , $c(v)$, to be the minimum length of an arc entering v , or infinity if no arc enters v . We say that a distance label of a labeled vertex is *exact* if the label is equal to the distance from the source to the vertex. The following *caliber lemma* is implicit in [10,26] and explicitly stated in [17].

The lemma allows us to relax Dijkstra's minimum distance label selection rule while maintaining the invariant that each vertex is scanned at most once.

Lemma 1. *Suppose ℓ is nonnegative and let μ be a lower bound on the distance labels of labeled vertices. Let v be a vertex such that $\mu + c(v) \geq d(v)$. Then $d(v)$ is exact.*

Previous implementations of the MLB algorithm, as well as those described below, use the following *wide bucket* heuristic. Let L be the smallest nonzero arc length and pick w such that $0 < w \leq L$. Then the MLB algorithm remains correct if one multiplies the bucket width on every level by w . With this heuristic and $w = \Theta(L)$, the algorithm needs only $\lceil \log_{\Delta}(U/L) \rceil = O(\log_{\Delta} C)$ levels, and one can replace “ U ” by “ C ” in the time bounds.

The *caliber heuristic* [17] uses Lemma 1 to detect and scan vertices with exact distance labels. To use the heuristic, we modify the MLB algorithm to keep labeled vertices in one of two places: a set F and a priority queue B implemented using the MLB structure. We refer to the modified algorithm as the *smart queue* algorithm.

At a high level, the algorithm works as follows. Vertices in F have exact distance labels. If F is nonempty, we remove and scan a vertex from F . If F is empty, we remove and scan a vertex from B with the minimum distance label. Suppose a distance label of a vertex u decreases. Note that u cannot belong to F . If u belongs to B , then we apply the **decrease-key** operation to u . This operation either relocates u within B or discovers that u 's distance label is exact and moves u to F . If u was neither in B nor F , we apply the **insert** operation to u , and u is inserted either into B or into F , depending on whether Lemma 1 applies or not.

The caliber heuristic provably improves algorithm performance for certain input distributions. For example if Δ is a constant and arc lengths are distributed uniformly over $[1, M]$ for some M that is the same for all arcs, then the expected running time of the MLB algorithm with the heuristic is $O(n + m)$.

3 Algorithm Implementations

We implemented several variants of the MLB algorithm. The MB code implements the algorithm with the wide bucket heuristic, and the SQ code adds the caliber heuristic. Next we discuss these codes and engineering considerations involved in their development.

Our implementation of MB is very similar to that of [18], except in the details of the **insert** operation. The previous implementation maintained a range of distance values for each level, updating the ranges when the value of μ changed. To insert a vertex, one looks for the lowest level to which the vertex belongs, and then computes the offset of the bucket to which the vertex belongs. The MB implementation computes the vertex position with respect to μ as described above. This is slightly more efficient when the number of levels is large. The

efficiency gain is bigger for SQ because it does not necessarily examine all levels for a given value of μ . The new implementation is also simpler than the old one.

We always set Δ to a power of two. This allows us to use bit shifts instead of divisions. Our codes set w to the biggest power of two not exceeding L . We use an array to represent each level of buckets.

One can give MB either k or Δ as a parameter. Then MB sets the other based on the input arc lengths. We refer to the code with the number of levels k set to two by MB2L, and to the code with Δ set to two by MB2D. These are the two extreme cases that we study. (We do not study the single-level case because it often would have needed too much memory and time.)

Alternatively, one can let MB choose the values of both k and Δ based on the input. We refer to this adaptive variant as MB-A. The adaptive variant of the algorithm uses the relationship $\Delta = \Theta(k)$ suggested by the worst-case analysis. To choose the constant hidden by the Θ location, we observe the following. Examining empty buckets involves looking at a single pointer and has good locality properties as we access the buckets sequentially. Moving vertices to lower levels, on the other hand, requires changing several pointers, and has poor locality. This suggests that Δ should be substantially greater than k , and experiments confirm this.

In more detail, MB-A sets k and Δ as follows. First we find the smallest value of k such that k is a power of two and $(16k)^k \geq U/w$. Then we set Δ to $16k$. At this point, however, both Δ and k may be larger than they need to be. While $(\Delta/2)^k \geq U/w$ we reduce Δ . Finally while $(\Delta)^{k-1} \geq U/w$ we reduce k . This typically leads to $16k \leq \Delta \leq 128k$ and works well in our tests.

We obtain our SQ code by adding the caliber heuristic to MB. The modification of MB is relatively straightforward. We use a stack to implement the set F needed by the caliber heuristic. The adaptive variant of the code, SQ-A, uses the same procedure to set k and Δ as MB-A does.

4 Experimental Methodology and Setup

Following Moret and Shapiro [22], we use a baseline code – breadth-first search (BFS) in our case – and measure running times of our shortest path codes on an input relative to the BFS running time on this input. Our BSF code computes distances and a shortest path tree for the unit length function. The breadth-first search problem is a simple special case of NSP and, modulo BFS implementation efficiency, the BFS running time is a lower bound on the NSP codes. Baseline running times give a good indication of how close to optimal the running times are and removes dependencies on some low-level implementation and architecture details.

However, some of the dependencies, in particular cache dependencies, remain. Our codes put arc and vertex records in consecutive locations. Input IDs of the vertices determine their ordering in memory. In general, breadth-first search examines vertices in a different order than an NSP algorithm. This may – and in some cases does – lead to very different caching behavior of the two codes

for certain vertex orderings. To deal with this dependency on input IDs, our generators permute the IDs at random. Thus all our problem generators are randomized.

For every input problem type and any set of parameter values, we run the corresponding generator five times and report the averages. We report the baseline BFS time in seconds and all other times in units of the BFS time. In addition, we count operations that determine ρ and ϕ in Theorem 1. For each of these operations, we give the number of the operations divided by the number of vertices, so that the amortized operation cost is immediate. The two kinds of operations we count are examinations of empty buckets and the number of vertices processed during bucket expansion operations.

We use 64-bit integers for internal representation of arc lengths and distances. If the graph contains paths longer than 2^{64} , our codes may get overflows. Note that for 32-bit input arc lengths, no overflow can happen unless the number of vertices exceeds 2^{32} , which is too many to fit into the memory of modern computers.

Our experiments have been conducted on a 933 MHz Pentium III machine with 512M of memory, 256K cache, and running RedHat Linux 7.1. All our shortest path codes and the baseline code are written in C++, in the same style, and compiled with the gcc compiler using the `-O6` optimization option. Our BFS code uses the same data structures as the MLB code.

5 Problem Families

We report data on seven problem families produced by three problem generators. Since we are interested in the efficiency of shortest path data structures, we restrict our study to sparse graphs, for which the data structure manipulation time is most apparent.

Our first generator, SPRAND, builds a Hamiltonian cycle and then adds arcs at random. The generator may produce parallel arcs but not self-loops. Arc lengths are chosen independently and uniformly from $[\ell, u]$. Vertex 0 is the source. If the number of arcs is large enough, SPRAND graphs are expanders and the average number of vertices in the priority queue during a shortest path computation is large.

We use SPRAND to generate two problem families, RAND-I and RAND-C. For both families, $\ell = 1$ and $m = 4n$. For RAND-I, $u = n$, and n increases by a factor of two from one set of parameter values to the next one. We chose the initial value of n large enough so that the running time is nonnegligable and the final value as large as possible subject to the constraint that all our codes run without paging. For RAND-C, $n = 2^{20}$, u starts at 1 and then takes on integer multiples of four from 4 to 32. Up to $u = 20$, the minimal arc length L in all test inputs is one. For $u = 24$, L is greater than one for some inputs. For $u = 28$ and 32, L is always greater than one. Note that the expected value of C does not change for $u \geq 28$, and therefore the results for $u > 32$ would have been very similar to those for $u = 28$ and 32.

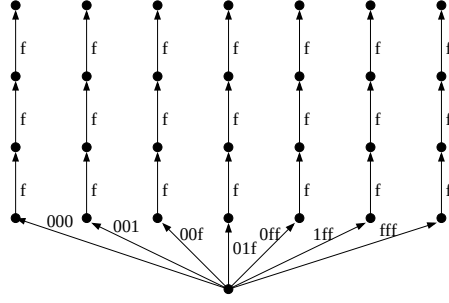


Fig. 1. An example of a hard problem instance; $k = 3$ and $\Delta = 16$. Arc lengths are given in hexadecimal. We omit the extra vertex with arcs designed to manipulate vertex calibers

Our second generator SPGRID produces grid-like graphs. An x, y grid graph contains $x \cdot y$ vertices, (i, j) , for $0 \leq i < x$ and $0 \leq j < y$. A vertex (i, j) is connected to the adjacent vertices in the same layer, $(i, j + 1 \bmod y)$ and $(i, j - 1 \bmod y)$. In addition, for $i < x - 1$, each vertex (i, j) is connected to the vertex $(i + 1, j)$. Arc lengths are chosen independently and uniformly from $[\ell, u]$. Vertex 0 is the source. We use SPGRID to generate two problem families, LONG-I and LONG-C. Both families contain *long* grid graphs with $y = 8$ and x large. For these graphs, the average number of vertices in the priority queue is small.

The LONG-I and LONG-C problem families are similar to the RAND-I and RAND-C families. For LONG-I, $u = n$ and n increases by a factor of two from the value that yields a reasonable running time to the maximum value that does not cause paging. The LONG-C problem family uses the same values of u as the RAND-C problem family, and for the same reasons.

Our last problem generator is SPHARD. This generator produces problems aimed to be hard for MLB algorithms for certain values of k and Δ . Graphs produced by this generator consist of $2k + 1$ vertex-disjoint paths, with the source connecting to the beginning of each path. (See Figure 1 for an example.) These paths have the same number of arcs, which can be adjusted to get a graph of the desired size. Path arcs have a length of Δ . The lengths of the source arcs are as follows. One arc has zero length. Out of the remaining arcs, k arcs have the following base- Δ representation. For $1 \leq i \leq k$, the first i digits are $\Delta - 1$ and the remaining digits are 0. The last k arcs, for $1 \leq j \leq k$, have the first $j - 1$ digits $\Delta - 1$, the j -th digit 1, and the remaining digits 0. The graph also contains an extra vertex with no incoming arcs connected to every other vertex of the graph. The length of the arc connecting the vertex to the source is zero to make sure that the minimum arc length is zero. Lengths of the other arcs are all the same. These lengths can be zero (to force every vertex caliber to zero) or large (so that the calibers are determined by the other arcs).

Note that if the SPHARD generator with parameters k and Δ produces an input, our adaptive codes may select different parameters. For $D = \log \Delta$, a problem produced by SPRAND has $(k \cdot D)$ -bit lengths. These lengths determine parameters selected by the adaptive codes.

The three SPHARD problem families we study are HARD1, HARD0, and HARDEST-SQ. The first two problem families differ only in the length of the arcs which determine vertex calibers: the length is large for the first family and zero for the second. All problems in this family have approximately 2^{20} vertices, and the number of arcs is approximately the same in all problems. To create a problem in this family, we chose k and D such that $k \cdot D = 36$ and generate a problem which is hard for MB with k levels and $\Delta = 2^D$. (We exclude $k = 1$ as always.) Each HARDEST-SQ problem also has approximately 2^{20} vertices. Problems in this family differ by the k and Δ values. These values are selected so that both the generator and the adaptive codes use the same k and Δ parameters.

6 Experimental Results

Previous work [4,18] suggests that MB2L performs well except on some problems with large arc lengths. Our experimental results confirm this, and suggest that the caliber heuristic, combined with adaptive selection of parameters, leads to a more robust code. Empty bucket scans cause bad performance of MB2L. For example, the data in Table 2 shows that MB2L performs similarly to SQ-A when C is small, but the latter code is faster when C is large. HARD1 and HARD0 problems (Table 3) show that in the worst case the difference is huge.

Next we compare performance of our MB and SQ codes. Data for HARD1 and HARD0 problems, given in Table 3, shows that the caliber heuristic can give significant savings in the number of operations or no savings at all. In the former case, SQ is much faster than MB. In the latter case, SQ is a little slower. Looking at the uniform arc length data, Tables 1 – 2, we see that the caliber heuristic provides significant improvement when the number of levels is large, as theoretical analysis predicts. This heuristic makes bucket structures with many levels more practical. In particular, it makes SQ practical.

Table 1. RAND-I (left) and LONG-I (right) family data

$n (= C)$	BFS	MB2L	SQ2L	MB2D	SQ2D	MB-A	SQ-A
2^{17} time	0.15	1.55	1.56	3.86	2.26	1.88	1.63
emp./n	sec.	3.01	1.09	0.74	0.01	2.05	0.06
exp./n		1.09	1.04	7.14	1.56	2.00	1.05
2^{18} time	0.30	1.73	1.64	4.15	2.26	1.93	1.73
emp./n	sec.	3.03	0.79	0.74	0.01	2.04	0.04
exp./n		1.45	1.19	7.65	1.56	2.36	1.19
2^{19} time	0.62	1.68	1.63	4.41	2.31	1.89	1.71
emp./n	sec.	3.34	1.12	0.74	0.00	2.36	0.05
exp./n		1.06	1.01	8.15	1.58	1.96	1.01
2^{20} time	1.30	1.83	1.79	4.64	2.35	1.94	1.79
emp./n	sec.	3.34	0.79	0.74	0.00	2.37	0.03
exp./n		1.41	1.20	8.65	1.58	2.09	1.06
2^{21} time	2.90	1.83	1.77	4.73	2.29	2.00	1.78
emp./n	sec.	3.67	1.13	0.74	0.00	2.36	0.03
exp./n		1.16	1.03	9.13	1.56	2.33	1.16

$n (= C)$	BFS	MB2L	SQ2L	MB2D	SQ2D	MB-A	SQ-A
2^{17} time	0.08	1.71	1.71	2.71	2.14	1.86	1.71
emp./n	sec.	8.59	4.77	1.00	0.29	4.88	1.00
exp./n		0.46	0.39	1.51	0.83	0.61	0.41
2^{18} time	0.17	1.66	1.61	2.80	2.13	1.81	1.68
emp./n	sec.	12.45	5.10	1.00	0.29	4.17	1.31
exp./n		0.27	0.22	1.51	0.83	0.63	0.46
2^{19} time	0.35	1.65	1.65	2.74	2.17	1.73	1.61
emp./n	sec.	13.65	9.43	1.00	0.29	8.89	1.40
exp./n		0.42	0.38	1.51	0.83	0.52	0.34
2^{20} time	0.75	1.59	1.60	2.72	2.10	1.64	1.60
emp./n	sec.	17.89	10.09	1.00	0.29	6.98	1.47
exp./n		0.23	0.21	1.51	0.83	0.45	0.31
2^{21} time	1.61	1.60	1.63	2.65	2.06	1.62	1.59
emp./n	sec.	23.59	18.84	1.00	0.29	5.88	2.44
exp./n		0.40	0.37	1.51	0.83	0.52	0.42

Table 2. RAND-C (left) and LONG-C (right) family data

bits	BFS	MB2L	SQ2L	MB2D	SQ2D	MLB-A	SQ-A
1 time	2.97	1.39	1.35	1.39	1.36	1.38	1.35
emp. sec.	0.00	0.00	0.00	0.00	0.00	0.00	0.00
exp.	0.48	0.48	0.48	0.48	0.48	0.48	0.48
4 time	2.99	1.74	1.59	1.99	1.73	1.47	1.42
emp. sec.	0.00	0.00	0.00	0.00	0.00	0.00	0.00
exp.	1.04	0.88	1.64	1.15	0.42	0.39	0.39
4 time	3.03	1.81	1.69	2.80	1.96	1.79	1.70
emp. sec.	0.00	0.00	0.00	0.00	0.00	0.00	0.00
exp.	1.26	1.05	3.51	1.49	1.26	1.05	
12 time	3.00	1.86	1.73	3.41	2.07	1.85	1.74
emp. sec.	0.01	0.01	0.01	0.00	0.01	0.01	0.01
exp.	1.35	1.13	5.49	1.56	1.35	1.13	
16 time	3.00	1.84	1.74	3.91	2.16	1.95	1.71
emp. sec.	0.14	0.07	0.08	0.00	0.12	0.04	
exp.	1.37	1.16	7.44	1.56	2.04	1.08	
20 time	2.99	1.82	1.75	4.52	2.27	1.89	1.71
emp. sec.	1.84	0.56	0.57	0.00	1.35	0.02	
exp.	1.37	1.16	8.97	1.56	2.11	1.04	
24 time	2.99	2.08	1.80	4.95	2.30	2.11	1.76
emp. sec.	18.43	1.32	0.93	0.00	6.12	0.03	
exp.	1.33	1.13	9.23	1.56	2.99	1.12	
28 time	2.92	2.59	1.85	5.22	2.37	2.13	1.75
emp. sec.	55.33	1.35	0.95	0.00	11.72	0.04	
exp.	1.33	1.13	9.23	1.56	2.79	1.06	
32 time	2.98	2.49	1.85	5.11	2.36	2.13	1.74
emp. sec.	55.33	1.36	0.95	0.00	11.72	0.04	
exp.	1.33	1.13	9.23	1.56	2.79	1.06	

bits	BFS	MLB2L	SQ2L	MLB2D	SQ2D	MLB-A	SQ-A
1 time	1.62	1.35	1.35	1.34	1.34	1.34	1.34
emp. sec.	0.13	0.06	0.13	0.06	0.13	0.06	
exp.	0.44	0.44	0.44	0.44	0.44	0.44	
4 time	1.63	1.54	1.54	1.72	1.60	1.42	1.49
emp. sec.	0.61	0.34	0.55	0.23	0.69	0.66	
exp.	0.88	0.71	1.29	0.80	0.39	0.38	
8 time	1.63	1.60	1.56	2.06	1.76	1.60	1.56
emp. sec.	2.64	0.77	0.96	0.29	2.64	0.77	
exp.	0.77	0.54	1.51	0.83	0.77	0.54	
12 time	1.63	1.56	1.55	2.28	1.87	1.56	1.55
emp. sec.	5.82	2.44	1.00	0.29	5.82	2.44	
exp.	0.52	0.42	1.51	0.83	0.52	0.42	
16 time	1.63	1.56	1.57	2.52	1.99	1.61	1.54
emp. sec.	13.68	9.45	1.00	0.29	8.87	1.40	
exp.	0.42	0.38	1.51	0.83	0.52	0.34	
20 time	1.63	1.67	1.69	2.75	2.10	1.57	1.54
emp. sec.	43.27	37.60	1.00	0.29	9.30	2.64	
exp.	0.39	0.37	1.51	0.83	0.34	0.25	
24 time	1.62	2.22	2.15	2.99	2.23	1.64	1.61
emp. sec.	146.96	136.51	1.00	0.30	6.10	2.30	
exp.	0.35	0.33	1.51	0.84	0.51	0.40	
28 time	1.62	3.08	3.02	3.05	2.33	1.65	1.64
emp. sec.	212.05	205.08	1.00	0.35	10.96	2.72	
exp.	0.31	0.30	1.51	0.91	0.43	0.38	
32 time	1.63	3.18	3.08	3.06	2.38	1.66	1.67
emp. sec.	212.00	206.77	1.00	0.36	10.96	2.91	
exp.	0.31	0.31	1.51	0.92	0.43	0.40	

If the number of empty bucket examinations per vertex is moderate (e.g., ten), they are well-amortized by other operations on vertices and do not have a noticeable effect on the running time. When the number of these operations reaches a hundred per vertex, they do have an effect. See e.g., Table 2. The same table shows that processing vertices during bucket expansion is more expensive. Processing one vertex influences the running time more than scanning a hundred empty buckets. These observations justify our choice of k and Δ in our adaptive algorithms.

HARD0 problems (Table 3) illustrate that adaptive selection of k and Δ is important from the worst-case point of view. (For 36-bit lengths used in HARD1 and HARD0 problems, our adaptive codes set $k = 6$.) As far as typical performance goes, our adaptive codes are never significantly slower, and sometimes significantly faster, than the corresponding non-adaptive codes. See, e.g., Table 2.

7 Concluding Remarks

Our data suggests that the caliber heuristic makes the MLB algorithm with adaptive parameter selection practical. Even on the hardest problem we used, when input arc lengths fit into 32-bit words, the running time of SQ-A is always

Table 3. hard1 (left) and hard0 (center) and hard-sq (right) data

k	BFS	MB	SQ	k	BFS	MB	SQ	bits log Δ k	BFS SQ-A
2 time	0.63	1189.20	1.38	2 time	0.62	1199.67	1201.13	4 4 1 time	0.62 1.37
emp. sec.	52428.94	0.40		emp. sec.	52428.94	52428.94		emp. sec.	0.33
exp.	0.60	0.30		exp.	0.60	0.60		exp.	0.04
3 time	0.63	10.66	1.39	3 time	0.62	9.48	9.39	6 3 2 time	0.63 1.50
emp. sec.	1170.14	0.15		emp. sec.	1170.14	1170.14		emp. sec.	1.60
exp.	1.14	0.57		exp.	1.14	1.14		exp.	0.80
4 time	0.62	2.68	1.44	4 time	0.63	2.67	2.82	8 4 2 time	0.62 1.51
emp. sec.	170.44	0.11		emp. sec.	170.44	170.44		emp. sec.	3.20
exp.	1.56	0.67		exp.	1.56	1.56		exp.	0.80
6 time	0.63	2.25	1.53	6 time	0.62	2.25	2.45	15 5 3 time	0.62 1.73
emp. sec.	24.31	0.08		emp. sec.	24.31	24.31		emp. sec.	9.00
exp.	2.46	0.77		exp.	2.46	2.46		exp.	1.14
9 time	0.62	2.87	1.60	9 time	0.62	2.87	3.08	18 6 3 time	0.62 1.78
emp. sec.	6.37	0.05		emp. sec.	6.37	6.37		emp. sec.	18.14
exp.	3.89	0.85		exp.	3.89	3.89		exp.	1.14
12 time	0.63	3.70	1.66	12 time	0.62	3.72	3.93	24 6 4 time	0.62 1.98
emp. sec.	3.12	0.04		emp. sec.	3.12	3.12		emp. sec.	21.11
exp.	5.36	0.88		exp.	5.36	5.36		exp.	1.56
18 time	0.63	5.86	1.82	18 time	0.63	5.86	6.05	30 6 5 time	0.62 2.18
emp. sec.	1.41	0.03		emp. sec.	1.41	1.41		emp. sec.	23.00
exp.	8.32	0.93		exp.	8.32	8.32		exp.	2.00
36 time	0.62	16.32	2.32	36 time	0.63	16.24	16.43	35 7 5 time	0.62 2.32
emp. sec.	0.49	0.01		emp. sec.	0.49	0.49		emp. sec.	46.27
exp.	17.75	0.97		exp.	17.75	17.75		exp.	2.00
								42 7 6 time	0.62 2.55
								emp. sec.	48.92
								exp.	2.46
								49 7 7 time	0.62 2.80
								emp. sec.	50.87
								exp.	2.93

within a factor of 2.5 of the running time of breadth-first search. We conjecture that for no input will this code run longer than the baseline code by more than a factor of three, given that the problem's size is much bigger than the cache size but smaller than the memory size. On many real-life problems, this code will run within a factor of two of the baseline code.

We also experimented with a hot queue [5] version of our code. Although we do not give details due to the lack of space, this code performs better than SQ-A on bad-case problems and almost as well on typical problems.

References

1. R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster Algorithms for the Shortest Path Problem. *J. Assoc. Comput. Mach.*, 37(2):213–223, April 1990. 502, 503
2. P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and Implementation of an Efficient Priority Queue. *Math. Systems Theory*, 10:99–127, 1977. 502
3. B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Math. Prog.*, 85:277–311, 1999. 503
4. B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest Paths Algorithms: Theory and Experimental Evaluation. *Math. Prog.*, 73:129–174, 1996. 503, 509
5. B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, Heaps, Lists, and Monotone Priority Queues. *SIAM J. Comput.*, 28:1326–1346, 1999. 502, 503, 511
6. E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979. 502, 503
7. R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969. 502
8. R. B. Dial, F. Glover, D. Karney, and D. Klingman. A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees. *Networks*, 9:215–248, 1979. 503
9. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959. 502, 503
10. E. A. Dinic. Economical algorithms for finding shortest paths in a network. In Yu.S. Popkov and B. L. Shmulyan, editors, *Transportation Modeling Systems*, pages 36–44. Institute for System Studies, Moscow, 1978. In Russian. 502, 504
11. L. Ford. Network Flow Theory. Technical Report P-932, The Rand Corporation, 1956. 503
12. L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962. 503
13. M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987. 502
14. M. L. Fredman and D. E. Willard. Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comp. and Syst. Sci.*, 48:533–551, 1994. 502
15. G. Gallo and S. Pallottino. Shortest Paths Algorithms. *Annals of Oper. Res.*, 13:3–79, 1988. 503
16. F. Glover, R. Glover, and D. Klingman. Computational Study of an Improved Shortest Path Algorithm. *Networks*, 14:25–37, 1984. 503
17. A. V. Goldberg. A Simple Shortest Path Algorithm with Linear Average Time. Technical Report STAR-TR-01-03, STAR Lab., InterTrust Tech., Inc., Santa Clara, CA, USA, 2001. 502, 503, 504, 505
18. A. V. Goldberg and C. Silverstein. Implementations of Dijkstra’s Algorithm Based on Multi-Level Buckets. In P. M. Pardalos, D. W. Hearn, and W. W. Hages, editors, *Lecture Notes in Economics and Mathematical System 450 (Refereed Proceedings)*, pages 292–327. Springer Verlag, 1997. 503, 505, 509
19. T. Hagerup. Improved Shortest Paths in the Word RAM. In *27th Int. Colloq. on Automata, Languages and Programming, Geneva, Switzerland*, pages 61–72, 2000. 502

20. D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *J. Assoc. Comput. Mach.*, 24:1–13, 1977. 502
21. U. Meyer. Single-Source Shortest Paths on Arbitrary Directed Graphs in Linear Average Time. In *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 797–806, 2001. A full version available as TR MPI-I-2001-1-002, Max-Planck-Institut für Informatik, Saarbrücken, Germany. 502
22. B. M. E. Moret and H. D. Shapiro. An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree. In *Proc. 2nd Workshop on Algorithms and Data Structures*, 1991. 506
23. K. Noshita. A Theorem on the Expected Complexity of Dijkstra’s Shortest Path Algorithm. *J. Algorithms*, 6:400–408, 1985. 502
24. R. Raman. Fast Algorithms for Shortest Paths and Sorting. Technical Report TR 96-06, King’s College, London, 1996. 502
25. R. Raman. Priority Queues: Small, Monotone and Trans-Dichotomous. In *Proc. 4th Annual European Symposium Algorithms*, pages 121–137. Springer-Verlag, Lect. Notes in CS 1136, 1996. 502
26. R. Raman. Recent Results on Single-Source Shortest Paths Problem. *SIGACT News*, 28:81–87, 1997. 502, 504
27. R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983. 503
28. M. Thorup. Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time. *J. Assoc. Comput. Mach.*, 46:362–394, 1999. 502
29. M. Thorup. On RAM Priority Queues. *SIAM Journal on Computing*, 30:86–109, 2000. 502
30. J. W. J. Williams. Algorithm 232 (Heapsort). *Comm. ACM*, 7:347–348, 1964. 502
31. F. B. Zhan and C. E. Noon. Shortest Path Algorithms: An Evaluation using Real Road Networks. *Transp. Sci.*, 32:65–73, 1998. 503

Efficient Algorithms for Weighted Colorings of Series-Parallel Graphs

Xiao Zhou and Takao Nishizeki

Graduate School of Information Sciences, Tohoku University
Aoba-yama 05, Sendai, 980-8579, Japan.
{zhou,nishi}@ecei.tohoku.ac.jp

Abstract. Let G be a weighted graph such that each vertex v has a positive integer weight $\omega(v)$. A weighted coloring of G is to assign a set of $\omega(v)$ colors to each vertex v so that any two adjacent vertices receive disjoint sets of colors. This paper gives an efficient algorithm to find the minimum number of colors required for a weighted coloring of a given series-parallel graph G in time $O(n\omega_{\max})$, where n is the number of vertices and $\omega_{\max}$ is the maximum vertex-weight of G .

1 Introduction

A *vertex-coloring of a graph* is to color all the vertices so that any two adjacent vertices are colored with different colors. Let $G = (V, E)$ be a weighted graph such that each vertex $v \in V$ has a positive integer weight $\omega(v)$, and let C be a set of colors. A *weighted coloring* $\Gamma : V \rightarrow 2^C$ of G is to assign a subset $\Gamma(v)$ of C to each vertex $v \in V$ so that $|\Gamma(v)| = \omega(v)$ and $\Gamma(u) \cap \Gamma(w) = \emptyset$ for any adjacent vertices $u, w \in V$. Thus the ordinary vertex-coloring is merely a weighted coloring for the case $\omega(v) = 1$ for every vertex v . The *weighted chromatic number* $\chi_\omega(G)$ of G is the minimum number of colors required for a weighted coloring of G , that is,

$$\chi_\omega(G) = \min\{|C| : C \text{ is a set of colors, and} \\ \text{there is a weighted coloring } \Gamma : V \rightarrow 2^C\}.$$

The *weighted coloring problem* is to compute the weighted chromatic number $\chi_\omega(G)$ of a given graph G . Figure 1(a) depicts a weighted graph G with $\chi_\omega(G) = 6$, and Figure 1(b) illustrates a weighted coloring of G with six colors $c_1, c_2, \dots, c_6$.

The weighted coloring problem has a natural application in scheduling theory [6]. Consider a set V of jobs such that each job $v \in V$ needs a total of $\omega(v)$ units of time to be finished and there are several pairs of jobs which cannot be executed simultaneously. This problem can be modeled by a graph G in which a vertex corresponds to a job and an edge corresponds to a pair of jobs which cannot be executed simultaneously. A weighted coloring of G corresponds to a preemptive schedule; if vertex v receives $\omega(v)$ colors, say $c_{i_1}, c_{i_2}, \dots, c_{i_{\omega(v)}}$, then

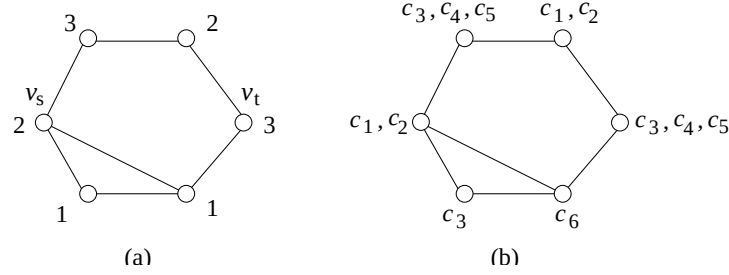


Fig. 1. (a) Series-parallel graph G and (b) a weighted coloring Γ of G with $r(\Gamma) = 0$

job v is executed in the i_1 th, i_2 th, $\dots$, and $i_{\omega(v)}$ th time slots, total in $\omega(v)$ time slots. The goal is to find a preemptive schedule of the minimum completion time of all jobs. Clearly, the minimum completion time is equal to the weighted chromatic number $\chi_\omega(G)$ of G .

Since the vertex-coloring problem is NP-hard, the weighted coloring problem is of course NP-hard and hence it is very unlikely that the weighted coloring problem can be efficiently solved for general graphs. However, there may exist a polynomial-time algorithm to solve the weighted coloring problem for a restricted class of graphs. Indeed, the problem can be solved for trees in time $O(n)$ [2,5], for triangulated graphs in time $O(n^2)$ [2,5], and for perfect graphs in time $O(mn)$ [5], where m is the number of edges and n is the number of vertices of a given graph G .

In this paper we consider another class of graphs, called “series-parallel graphs.” A series-parallel graph can be constructed from single-edge graphs by repeatedly applying series and parallel connections; the formal definition will be given in the succeeding section. (See Figs. 2 and 3.) A series-parallel graph often appears as a constraint graph of scheduling. It is thus expected to obtain an efficient algorithm to solve the weighted coloring problem for series-parallel graphs. Takamizawa *et al.* gave a general method to design dynamic programming algorithms for solving in linear time many combinatorial problems on series-parallel graphs, including the vertex-coloring problem [7]. However, an algorithm directly derived from the general method takes time $O(n\{\chi_\omega(G)\}^{2\omega_{\max}}) = O(n(3\omega_{\max})^{2\omega_{\max}})$ to solve the weighted coloring problem for series-parallel graphs G , where $\omega_{\max}$ is the maximum vertex weight, that is, $\omega_{\max} = \max_{v \in V} \omega(v)$. Note that $\chi_\omega(G) \leq 3\omega_{\max}$ since any series-parallel graph G has a vertex coloring with at most three colors. The algorithm thus does not run in time polynomial in n and $\omega_{\max}$.

In this paper we give an algorithm to solve the weighted coloring algorithm for series-parallel graphs $G = (V, E)$ in time $O(n\omega_{\max})$. Thus the algorithm is much faster than the straightforward algorithm, and takes linear time if $O(\omega_{\max})$ is bounded. It should be noted that a representation of a weighted coloring of G requires space $\Omega(\sum_{v \in V} \omega(v))$.

The paper is organized as follows. Section 2 includes basic definitions and notations. Section 3 gives a simple algorithm to compute $\chi_\omega(G)$ of a given series-parallel graph G in time $O(n\omega_{\max}^3)$. Section 4 improves the time-complexity to $O(n\omega_{\max})$.

2 Terminology and Definitions

In this section we present some definitions and easy observations. We denote by $G = (V, E)$ a graph with vertex set V and edge set E . A *(two-terminal) series-parallel graph* is defined recursively as follows [7,9]:

- 1 A graph G of a single edge is a series-parallel graph. The ends v_s and v_t of the edge are called the *terminals* of G and denoted by $v_s(G)$ and $v_t(G)$.
- 2 Let G_1 be a series-parallel graph with terminals $v_s(G_1)$ and $v_t(G_1)$, and let G_2 be a series-parallel graph with terminals $v_s(G_2)$ and $v_t(G_2)$.
 - (a) A graph G obtained from G_1 and G_2 by identifying vertex $v_t(G_1)$ with vertex $v_s(G_2)$ is a series-parallel graph whose terminals are $v_s(G) = v_s(G_1)$ and $v_t(G) = v_t(G_2)$. Such a connection is called a *series connection*, and G is denoted by $G = G_1 \bullet G_2$. (See Fig. 2(a).)
 - (b) A graph G obtained from G_1 and G_2 by identifying $v_s(G_1)$ with $v_s(G_2)$ and $v_t(G_1)$ with $v_t(G_2)$ is a series-parallel graph whose terminals are $v_s(G) = v_s(G_1) = v_s(G_2)$ and $v_t(G) = v_t(G_1) = v_t(G_2)$. Such a connection is called a *parallel connection*, and G is denoted by $G = G_1 \parallel G_2$. (See Fig. 2(b).)

The terminals $v_s(G)$ and $v_t(G)$ of G are often denoted simply by v_s and v_t . A series-parallel graph G can be represented by a “binary decomposition tree” [7]. Figure 3 illustrates a series-parallel graph G and its binary decomposition tree T_b . Labels s and p attached to internal nodes in T_b indicate series and parallel connections, respectively. Nodes labeled s and p are called *s-* and *p-nodes*, respectively. Every leaf of T_b represents a subgraph of G induced by an edge. An edge joining vertices u and v is denoted by (u, v) . A node u of tree T_b corresponds to a subgraph G_u of G induced by all edges represented by the leaves that are

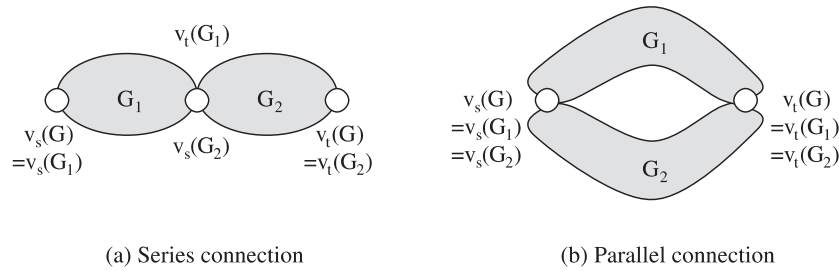


Fig. 2. Series and parallel connections

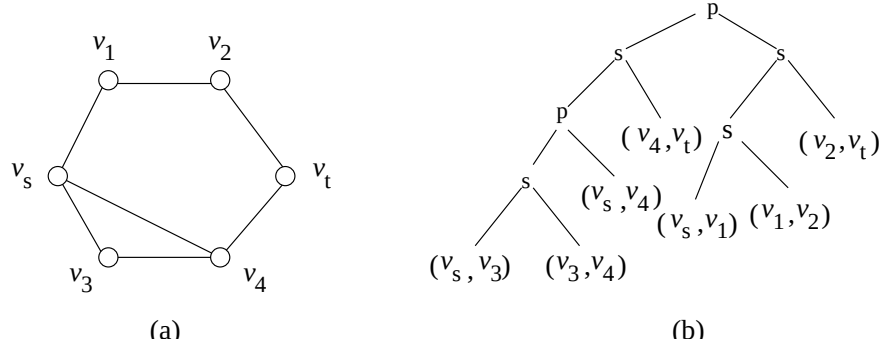


Fig. 3. (a) A series-parallel graph G and (b) its binary decomposition tree T_b

descendants of u in T_b . Thus $G = G_r$ for the root r of T_b . One can find a binary decomposition tree of a given series-parallel graph in linear time [7]. Since we deal with the weighted (vertex-)coloring, we may assume without loss of generality that G is a simple graph, that is, G has no multiple edges.

Let C be a set of colors, and let $G = (V, E)$ be a weighted graph such that each vertex $v \in V$ has a positive integer weight $\omega(v)$. A *weighted coloring* $\Gamma : V \rightarrow 2^C$ of G is to assign a subset $\Gamma(v)$ of C to each vertex $v \in V$ so that $|\Gamma(v)| = \omega(v)$ and $\Gamma(u) \cap \Gamma(w) = \emptyset$ for any adjacent vertices $u, w \in V$. The *weighted chromatic number* $\chi_\omega(G)$ of a graph G is the minimum number of colors required by a weighted coloring of G . The number of colors used by a weighted coloring Γ is denoted by $\#\Gamma$.

A graph $G = (V, E)$ is defined to be a *k-tree* if it is a complete graph of k vertices or it has a vertex $v \in V$ of degree k whose neighbors induce a clique of size k and the graph $G - \{v\}$ obtained from G by deleting the vertex v and all edges incident to v is again a *k-tree*. A graph is defined to be a *partial k-tree* if it is a subgraph of a *k-tree* [1,3,4,8]. A series-parallel simple graph is a partial 2-tree.

One can observe that a weighted coloring of a graph $G = (V, E)$ is an ordinary vertex-coloring of a new graph G_ω , defined as follows. Replace each vertex $v \in V$ with a complete graph $K_{\omega(v)}$ having $\omega(v)$ vertices, join each vertex of $K_{\omega(v)}$ to all vertices in $K_{\omega(w)}$ for each edge $(v, w) \in E$, and let G_ω be the resulting graph. (See Fig. 4.) Then a weighted coloring of G induces an ordinary vertex-coloring of G_ω , and *vice versa*. Thus the weighted coloring problem for G can be reduced to the vertex-coloring problem for G_ω . Since G_ω has $\sum_{v \in V} \omega(v)$ ($\leq n\omega_{\max}$) vertices and $O((n+m)\omega_{\max}^2)$ edges, the reduction takes time $O((n+m)\omega_{\max}^2)$, where $n = |V|$ and $m = |E|$. Unfortunately, G_ω is no more a series-parallel graph even if G is a series-parallel graph. Thus one cannot solve the weighted coloring problem for a series-parallel graph G by applying a linear-time vertex-coloring algorithm for series-parallel graphs in [7] to G_ω . However, G_ω is a partial $3\omega_{\max}$ -tree if G is a series-parallel graph and hence G is a partial 2-tree. The vertex-coloring problem can be solved for partial k -trees in time $O(n(k+1)^{2(k+1)})$ [1,4]. Thus one can

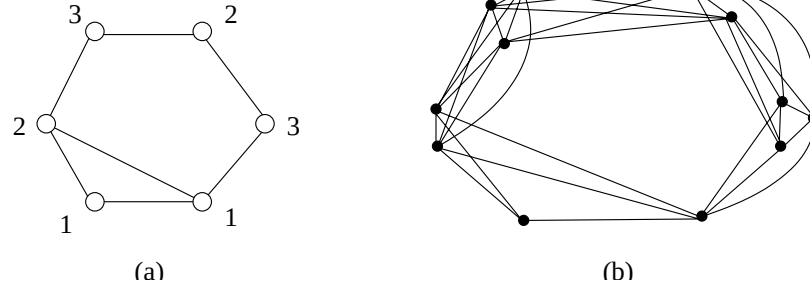


Fig. 4. Transformation of (a) G to (b) G_ω

solve the weighted coloring problem for a series-parallel graph G by applying the vertex-coloring algorithm to G_ω , but it takes time $O((\sum_{v \in V} \omega(v))(3\omega_{\max} + 1)^{2(3\omega_{\max} + 1)}) = O(n(3\omega_{\max} + 1)^{6\omega_{\max} + 3})$, which is not polynomial in $\omega_{\max}$. Our $O(n\omega_{\max})$ algorithm is much faster than this algorithm.

3 Simple Algorithm

In this section we present a simple algorithm to solve the weighted coloring problem for series-parallel graphs G in time $O(n\omega_{\max}^3)$. The algorithm runs in time polynomial in n and $\omega_{\max}$. In the remainder of the paper a weighted coloring is simply called a *coloring*. We say that a coloring Γ of G is *optimal* if $\#\Gamma = \chi_\omega(G)$.

It is easy to observe that $\chi_\omega(G) = \max\{\chi_\omega(G_1), \chi_\omega(G_2)\}$ when $G = G_1 \bullet G_2$. One can thus immediately compute $\chi_\omega(G)$ from $\chi_\omega(G_1)$ and $\chi_\omega(G_2)$ when $G = G_1 \bullet G_2$. However, this is not the case when $G = G_1 \parallel G_2$. For example, the graph depicted in Fig. 1(a) is obtained from the graphs G_1 and G_2 depicted in Fig. 5 by a parallel connection, that is, $G = G_1 \parallel G_2$, but $\chi_\omega(G_1) = 5$, $\chi_\omega(G_2) = 4$, $\chi_\omega(G) = 6$ and hence $\max\{\chi_\omega(G_1), \chi_\omega(G_2)\} < \chi_\omega(G)$. Thus $\chi_\omega(G)$ cannot be computed directly from $\chi_\omega(G_1)$ and $\chi_\omega(G_2)$ when $G = G_1 \parallel G_2$.

Our main idea is to introduce new invariants $\chi_\omega(G, i)$ for nonnegative integers i , and to compute the set of invariants $\chi_\omega(G, i)$ from the counterparts of G_1 and G_2 . The number of colors assigned to both v_s and v_t by a coloring Γ is denoted by $r(\Gamma)$, that is, $r(\Gamma) = |\Gamma(v_s) \cap \Gamma(v_t)|$. Then $\chi_\omega(G, i)$ is defined to be the minimum number of colors used by a coloring of G that assigns exactly i common colors to both v_s and v_t , that is,

$$\chi_\omega(G, i) = \begin{cases} \min\{\#\Gamma \mid \Gamma \text{ is a coloring of } G \text{ with } r(\Gamma) = i\} & \text{if } G \text{ has such a coloring;} \\ \infty & \text{otherwise.} \end{cases}$$

Clearly $\chi_\omega(G, 0) \neq \infty$, because a trivial coloring Γ , assigning all vertices pairwise disjoint color sets, satisfies $r(\Gamma) = 0$. Let $\delta_\omega(G) = \min\{\omega(v_s), \omega(v_t)\}$, then any

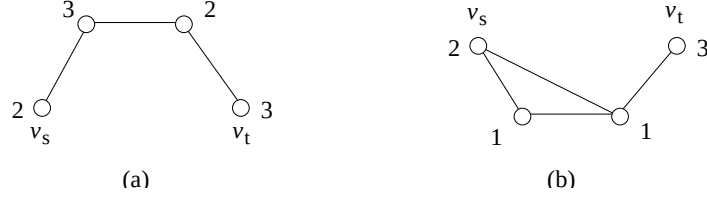


Fig. 5. Series-parallel graphs (a) G_1 and (b) G_2

coloring Γ of G satisfies $0 \leq r(\Gamma) \leq \delta_\omega(G)$. Therefore

$$\chi_\omega(G) = \min\{\chi_\omega(G, i) \mid 0 \leq i \leq \delta_\omega(G)\}.$$

For the graph G depicted in Fig. 1(a), $\chi_\omega(G, 0) = 6$, $\chi_\omega(G, 1) = 6$, $\chi_\omega(G, 2) = 7$, $\chi_\omega(G, i) = \infty$ for any integer $i > \delta_\omega(G) = 2$, and hence $\chi_\omega(G) = 6$. We say that a coloring Γ of G is an i -coloring if $r(\Gamma) = i$, and that Γ is an i -optimal coloring of G if $r(\Gamma) = i$ and $\#\Gamma = \chi_\omega(G, i)$.

We will show in Lemmas 1 and 2 that the set of all values $\chi_\omega(G, i)$, $0 \leq i \leq \delta_\omega(G)$, can be computed from the counterparts of G_1 and G_2 when $G = G_1 \parallel G_2$ or $G = G_1 \bullet G_2$.

Consider first the case where $G = G_1 \parallel G_2$. We show that the value $\chi_\omega(G, i)$ can be directly computed from the two values $\chi_\omega(G_1, i)$ and $\chi_\omega(G_2, i)$ in this case although $\chi_\omega(G)$ cannot be computed from $\chi_\omega(G_1)$ and $\chi_\omega(G_2)$. Suppose that G has an i -optimal coloring Γ for an integer i , $0 \leq i \leq \delta_\omega(G)$. Let $\Gamma_1 = \Gamma|_{G_1}$ be the restriction of Γ to G_1 , that is, $\Gamma_1(v) = \Gamma(v)$ for each vertex v in G_1 . Let $\Gamma_2 = \Gamma|_{G_2}$ be the restriction of Γ to G_2 , that is, $\Gamma_2(v) = \Gamma(v)$ for each vertex v in G_2 . Then Γ_1 and Γ_2 are i -colorings of G_1 and G_2 , respectively, and hence

$$\begin{aligned} \chi_\omega(G, i) &= \#\Gamma \\ &\geq \max\{\#\Gamma_1, \#\Gamma_2\} \\ &\geq \max\{\chi_\omega(G_1, i), \chi_\omega(G_2, i)\}. \end{aligned}$$

Suppose conversely that G_1 and G_2 have i -optimal colorings Γ_1 and Γ_2 , respectively. Then one can easily construct an i -coloring Γ of G with $\#\Gamma = \max\{\#\Gamma_1, \#\Gamma_2\}$ from Γ_1 and Γ_2 by combining Γ_1 and Γ_2 and renaming some colors in G_1 or G_2 , and hence

$$\begin{aligned} \chi_\omega(G, i) &\leq \#\Gamma \\ &= \max\{\#\Gamma_1, \#\Gamma_2\} \\ &= \max\{\chi_\omega(G_1, i), \chi_\omega(G_2, i)\}. \end{aligned}$$

We thus have the following lemma.

Lemma 1. *Let $G = G_1 \parallel G_2$, then for every i , $0 \leq i \leq \delta_\omega(G)$,*

$$\chi_\omega(G, i) = \max\{\chi_\omega(G_1, i), \chi_\omega(G_2, i)\}.$$

Thus it is rather easy to compute $\chi_\omega(G, i)$ from $\chi_\omega(G_1, i)$ and $\chi_\omega(G_2, i)$ when $G = G_1 \parallel G_2$. However, it is difficult to do so when $G = G_1 \bullet G_2$. For example, the graph G_1 in Fig. 5(a) is obtained from the graphs G'_1 and G''_1 in Fig. 6 by a series connection, that is, $G_1 = G'_1 \bullet G''_1$, but $\chi_\omega(G_1, 0) = 5$, $\chi_\omega(G'_1, 0) = 7$, $\chi_\omega(G''_1, 0) = 5$, and hence $\chi_\omega(G_1, 0) < \max\{\chi_\omega(G'_1, 0), \chi_\omega(G''_1, 0)\}$.

Although one cannot compute the value $\chi_\omega(G, i)$, $0 \leq i \leq \delta_\omega(G)$, directly from the two values $\chi_\omega(G_1, i)$ and $\chi_\omega(G_2, i)$ for the same integer i when $G = G_1 \bullet G_2$, we claim that the value $\chi_\omega(G, i)$ can be computed from the following two sets of values:

$$\{\chi_\omega(G_1, i_1) \mid 0 \leq i_1 \leq \delta_\omega(G_1)\}$$

and

$$\{\chi_\omega(G_2, i_2) \mid 0 \leq i_2 \leq \delta_\omega(G_2)\}.$$

Let $0 \leq i \leq \delta_\omega(G)$, $0 \leq i_1 \leq \delta_\omega(G_1)$ and $0 \leq i_2 \leq \delta_\omega(G_2)$, and let Γ be an i -optimal coloring of $G = G_1 \bullet G_2$ such that $\Gamma_1 = \Gamma|_{G_1}$ is an i_1 -coloring of G_1 and $\Gamma_2 = \Gamma|_{G_2}$ is an i_2 -coloring of G_2 . Then, although we omit the proof in this extended abstract, we have

$$l(i_1, i_2) \leq i \leq h(i_1, i_2) \quad (1)$$

and

$$\alpha(i, i_1, i_2) \leq \#\Gamma \quad (2)$$

where

$$\begin{aligned} l(i_1, i_2) &= \max\{0, i_1 + i_2 - \omega(v)\}, \\ v &= v_t(G_1) = v_s(G_2), \text{ (see Fig. 2(a))} \\ h(i_1, i_2) &= \min\{\delta_\omega(G), \omega(v_s) - i_1 + i_2, \omega(v_t) - i_2 + i_1\}, \text{ and} \\ \alpha(i, i_1, i_2) &= \max\{\chi_\omega(G_1, i_1), \chi_\omega(G_2, i_2), \omega(v_s) + \omega(v_t) - i, \\ &\quad \omega(v_s) + \omega(v_t) - i + \omega(v) - i_1 - i_2\}. \end{aligned}$$

Thus $l(i_1, i_2)$ is a lower bound on i , $h(i_1, i_2)$ is an upper bound on i , and $\alpha(i, i_1, i_2)$ is a lower bound on $\#\Gamma$. Conversely, we can prove that G indeed has an i -coloring Γ for which Eq. (2) holds in equality, that is,

$$\#\Gamma = \alpha(i, i_1, i_2)$$

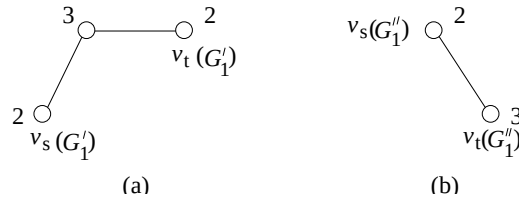


Fig. 6. Series-parallel graphs (a) G'_1 and (b) G''_1

whenever i , i_1 and i_2 satisfy Eq. (1) and G_1 and G_2 have i_1 -optimal and i_2 -optimal colorings, respectively. We thus have the following lemma.

Lemma 2. *Let $G = G_1 \bullet G_2$, then for every i , $0 \leq i \leq \delta_\omega(G)$,*

$$\chi_\omega(G, i) = \min\{\alpha(i, i_1, i_2) \mid l(i_1, i_2) \leq i \leq h(i_1, i_2), \\ 0 \leq i_1 \leq \delta_\omega(G_1), 0 \leq i_2 \leq \delta_\omega(G_2)\}.$$

By Lemmas 1 and 2 we obtain the following straightforward algorithm to compute $\chi_\omega(G)$ for a series-parallel graph G . Let T_b be a binary decomposition tree of G , and let u be a node in T_b , and let G_u be the subgraph of G corresponding to the subtree of T_b rooted at u . (See Fig. 3.) Then the following procedure $\text{Color}(G_u)$ computes the values $\chi_\omega(G_u, i)$ for all integers i , $0 \leq i \leq \delta_\omega(G_u)$. Since $G = G_r$ for the root r of T_b , we call procedure $\text{Color}(G_r)$ to compute the values $\chi_\omega(G, i)$ for i , $0 \leq i \leq \delta_\omega(G)$. From them one can easily compute the weighted chromatic number $\chi_\omega(G) = \min\{\chi_\omega(G, i) \mid 0 \leq i \leq \delta_\omega(G)\}$.

```

1 Procedure  $\text{Color}(G_u)$ ;
   {  $G_u$  is the subgraph of  $G$  corresponding to node  $u$  of  $T_b$  }
2 begin
3   if  $u$  is a leaf node then begin
4     let  $v_s$  and  $v_t$  be the ends of the edge in  $G_u$ ;
5      $\delta_\omega(G_u) := \min\{\omega(v_s), \omega(v_t)\}$ ;
6      $\chi_\omega(G_u, 0) := \omega(v_s) + \omega(v_t)$ 
7   end
8   else if  $u$  is a p-node of  $T_b$  then begin
9     let  $v_1$  and  $v_2$  be the children of node  $u$  in  $T_b$ ;
                                     {  $G_u = G_{v_1} \parallel G_{v_2}$  }
10     $\text{Color}(G_{v_1})$ ;  $\text{Color}(G_{v_2})$ ;
11    compute  $\delta_\omega(G_u)$  and all  $\chi_\omega(G_u, i)$  by using Lemma 1
12  end
13  else begin
14    let  $v_1$  and  $v_2$  be the children of node  $u$  in  $T_b$ ;
                                     {  $G_u = G_{v_1} \bullet G_{v_2}$  }
15     $\text{Color}(G_{v_1})$ ;  $\text{Color}(G_{v_2})$ ;
16    compute  $\delta_\omega(G_u)$  and all  $\chi_\omega(G_u, i)$  by using Lemma 2
17  end
18 end;
```

The variables i , i_1 and i_2 range from 0 to at most $\omega_{\max}$, where $\omega_{\max} = \max\{\omega(v) \mid v \in V\}$. Therefore Lines 11 and 16 in the algorithm above can be done in time $O(\omega_{\max}^3)$. Since $G = (V, E)$ is a series-parallel simple graph, $|E| \leq 2n - 3$ [7]. Therefore T_b contains $2n - 3$ leaves and hence $4n - 7$ nodes in total. Consequently the recursive calls occur at most $4n - 7$ times during the execution of $\text{Color}(G_r)$. Thus the total running time of the algorithm is $O(n\omega_{\max}^3)$.

4 Efficient Algorithm

In Section 3 we have shown that $\chi_\omega(G, i)$ for all i and hence $\chi_\omega(G)$ can be computed in time $O(n\omega_{\max}^3)$. In this section we improve the time complexity to $O(n\omega_{\max})$, that is, we give an algorithm to compute the weighted chromatic number $\chi_\omega(G)$ of a series-parallel graph G in time $O(n\omega_{\max})$.

The first idea behind the efficient algorithm is to observe that $\chi_\omega(G, i)$ is a convex and “unit-staircase” function with respect to i , as illustrated in Fig. 7. That is, the following two lemmas hold.

Lemma 3. *Let G be a series-parallel graph. If $(v_s, v_t) \notin E$ and $0 \leq i < \delta_\omega(G)$, then $|\chi_\omega(G, i) - \chi_\omega(G, i+1)| \leq 1$.*

Lemma 4. *Let G be a series-parallel graph, and let i, l and h be any integers such that $l \leq i \leq h$. Then $\chi_\omega(G, i) \leq \max\{\chi_\omega(G, l), \chi_\omega(G, h)\}$.*

By Lemma 4 $\chi_\omega(G, i)$ is a convex function. One can therefore consider a kind of inverse functions of $\chi_\omega(G, i)$, which are denoted by $i_{\min}(G, j)$ and $i_{\max}(G, j)$ and defined for an integer j as follows:

$$i_{\min}(G, j) = \begin{cases} \min\{i \mid \chi_\omega(G, i) \leq j\} & \text{if } j \geq \chi_\omega(G); \\ +\infty & \text{otherwise,} \end{cases}$$

and

$$i_{\max}(G, j) = \begin{cases} \max\{i \mid \chi_\omega(G, i) \leq j\} & \text{if } j \geq \chi_\omega(G); \\ -\infty & \text{otherwise.} \end{cases}$$

Then $\chi_\omega(G, i) \leq j$ if and only if $i_{\min}(G, j) \leq i \leq i_{\max}(G, j)$. See Fig. 7.

The second idea behind the efficient algorithm is to recursively compute $\chi_\omega(G)$ by using $i_{\min}(G, j)$ and $i_{\max}(G, j)$ in place of $\chi_\omega(G, i)$. By Lemma 3, if $\chi_\omega(G, i) \neq \infty$, then $\chi_\omega(G, i) \leq \chi_\omega(G) + \delta_\omega(G)$. It therefore suffices to compute $i_{\min}(G, j)$ and $i_{\max}(G, j)$ only for all j such that $\chi_\omega(G) \leq j \leq \chi_\omega(G) + \delta_\omega(G)$. Define a *min-max triple set* $\mathcal{T}(G)$ as follows:

$$\mathcal{T}(G) = \{(j, i_{\min}(G, j), i_{\max}(G, j)) \mid \chi_\omega(G) \leq j \leq \chi_\omega(G) + \delta_\omega(G)\}.$$

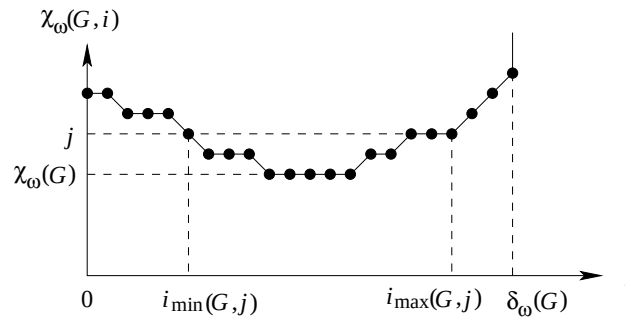


Fig. 7. Illustration for functions $\chi_\omega(G, i)$, $i_{\min}(G, j)$ and $i_{\max}(G, j)$

Then one can compute $\chi_\omega(G)$ from $\mathcal{T}(G)$ in $O(\delta_\omega(G))$ time since

$$\chi_\omega(G) = \min\{j \mid (j, i_{\min}(G, j), i_{\max}(G, j)) \in \mathcal{T}(G)\}.$$

The third idea is to notice that $\mathcal{T}(G)$ can be computed directly from $\mathcal{T}(G_1)$ and $\mathcal{T}(G_2)$ when $G = G_1 \bullet G_2$ or $G_1 \parallel G_2$. When one computes $\mathcal{T}(G)$ from $\mathcal{T}(G_1)$ and $\mathcal{T}(G_2)$, one needs to know $\chi_\omega(G)$ since j ranges from $\chi_\omega(G)$ to $\chi_\omega(G) + \delta_\omega(G)$. One can indeed compute $\mathcal{T}(G)$ and $\chi_\omega(G)$ from $\mathcal{T}(G_1)$ and $\mathcal{T}(G_2)$ as shown in the following Lemmas 5 and 6 for the cases $G = G_1 \parallel G_2$ and $G = G_1 \bullet G_2$, respectively.

Lemma 5. *Let $G = G_1 \parallel G_2$ and $j \geq \chi_\omega(G)$, then the following (a), (b) and (c) hold:*

(a)

$$\chi_\omega(G) = \min\{x \geq b_2 \mid i_{\max}(G_1, x) \geq i_{\min}(G_2, x), \\ i_{\max}(G_2, x) \geq i_{\min}(G_1, x)\},$$

where $b_2 = \max\{\chi_\omega(G_1), \chi_\omega(G_2)\}$ is a trivial lower bound on $\chi_\omega(G)$;

(b) $i_{\min}(G, j) = \max\{i_{\min}(G_1, j), i_{\min}(G_2, j)\}$; and

(c) $i_{\max}(G, j) = \min\{i_{\max}(G_1, j), i_{\max}(G_2, j)\}$.

Lemma 6. *Let $G = G_1 \bullet G_2$, $v = v_t(G_1) = v_s(G_2)$, and $j \geq \chi_\omega(G)$, then the following (a), (b) and (c) hold:*

(a) $\chi_\omega(G) = \max\{\chi_\omega(G_1), \chi_\omega(G_2)\}$;

(b)

$$i_{\min}(G, j) = \max\{0, \omega(v_s) + \omega(v_t) - j, \\ \omega(v_s) + \omega(v_t) - j + \omega(v) - i_{\max}(G_1, j) - i_{\max}(G_2, j)\};$$

and

(c)

$$i_{\max}(G, j) = \min\{\delta_\omega(G), \\ \omega(v_s) - i_{\min}(G_1, j) + i_{\max}(G_2, j), \\ \omega(v_t) - i_{\min}(G_2, j) + i_{\max}(G_1, j)\}.$$

From these lemmas we have the following lemma.

Lemma 7. *When $G = G_1 \bullet G_2$ or $G = G_1 \parallel G_2$, one can compute $\mathcal{T}(G)$ from $\mathcal{T}(G_1)$ and $\mathcal{T}(G_2)$ in time $O(\omega_{\max})$.*

From the lemmas above we can obtain the following theorem.

Theorem 1. *The weighted chromatic number $\chi_\omega(G)$ of a series-parallel graph G can be computed in $O(n\omega_{\max})$ time.*

It is easy to modify our algorithm so that it not only computes $\chi_\omega(G)$ but also actually finds an optimal coloring of G using $\chi_\omega(G)$ colors.

References

1. S. Arnborg and J. Lagergren. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991. 517
2. E. Balas and J. Xue. Minimum weighted colouring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs. *SIAM. J. Comput.*, 20:209–221, 1991. 515
3. H. L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k -trees. *Journal of Algorithms*, 11(4):631–643, 1990. 517
4. R. B. Borie, R. G. Parker, and C. A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7:555–581, 1992. 517
5. C. T. Hoàng. Efficient algorithms for minimum weighted colouring of some classes of perfect graphs. *Discrete Applied Mathematics*, 55:133–143, 1994. 515
6. D. Karger, C. Stein, and J. Wein. Scheduling algorithms, in “Algorithms and Theory of Computation Handbook” (Editor M. J. Atallah), CRC Press, 1998. 514
7. K. Takamizawa, T. Nishizeki, and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of ACM*, 29(3):623–641, 1982. 515, 516, 517, 521
8. X. Zhou, S. Nakano, and T. Nishizeki. Edge-coloring partial k -trees. *Journal of Algorithms*, 21:598–617, 1996. 517
9. X. Zhou, H. Suzuki, and T. Nishizeki. A linear algorithm for edge-coloring series-parallel multigraphs, *Journal of Algorithms*, 20:174–201, 1996. 516

Go with the Winners Algorithms for Cliques in Random Graphs

Marcus Peinado

Microsoft
Redmond, WA 98052, USA
marcuspe@microsoft.com

Abstract. This paper analyzes the performance of the *Go with the Winners* algorithm (GWTW) of Aldous and Vazirani [1] on random instances of the clique problem. In particular, we consider the uniform distribution on the set of all graphs with $n \in \mathbb{N}$ vertices. We prove a lower bound of $n^{\Omega(\log n)}$ and a matching upper bound on the time needed by GWTW to find a clique of size $(1 + \epsilon) \log n$ (for any constant $\epsilon > 0$). We extend the lower bound result to other distributions, under which graphs are guaranteed to have large cliques.

1 Introduction

A *clique* of size k in a graph $G = (V, E)$ is a complete subgraph on k vertices, i.e., a set $C \subseteq V$ of k vertices, such that every pair of vertices in C is connected by an edge. The clique problem is that of deciding if a graph G contains a clique of size k , given G and k . In the construction version of the problem, the task is to find a clique of size k .

The clique problem is one of Karp's original NP-complete problems [15]. More recently, there has been a sequence of results culminating in [9,24], which show that it is hard to find even an approximate solution. The best known positive result is the algorithm by Boppana and Halldórsson [6].

In addition to being one of the classical problems in theoretical computer science, the clique problem can be used as an abstract model for problems of practical interest – either directly or via its close relationship to the graph coloring problem. Examples include pattern matching, printed circuit board testing, scheduling and assignment of resources, such as radio frequencies or CPU registers (cf. [13] for references and details).

In light of the negative worst-case results, we focus our attention on the average case. We consider the Erdős-Rényi random graph model $\mathcal{G}(n, p)$ ($0 \leq p \leq 1$) over graph instances containing n vertices. A graph G may be drawn from this distribution by inserting each of the $\binom{n}{2}$ possible edges into G independently with probability p . The analysis in this paper considers the case $p = 1/2$, i.e., the uniform distribution. However, it is straightforward to generalize the results for a broad range of values of p . For graphs generated from the uniform distribution, with high probability, the largest clique has size $2 \log_2 n - O(\log \log n)$ [5,18].

Cliques of size up to $(1 - \epsilon) \log_2 n$ (for any $\epsilon > 0$) can be found in linear time by a simple greedy algorithm [11].

In contrast, it is a long-standing open problem to determine if even slightly larger cliques $((1 + \epsilon) \log n)$ can be found in polynomial time. Karp [16] first issued the challenge of finding such an algorithm 25 years ago. Proving that no such algorithm can be found would appear to require new techniques in complexity theory. Lacking general results, it is natural to analyze the performance of concrete algorithms. No positive results are known. A small number of authors analyze concrete algorithms for the clique problem and prove negative results about their ability to find large cliques in different types of random graphs [11, 12, 20]. Most notably, Jerrum [12] demonstrates the existence of an initial state from which the Metropolis algorithm, a fixed-temperature variant of simulated annealing, cannot find a clique of size $(1 + \epsilon) \log_2 n$ for any constant $\epsilon > 0$ in expected polynomial time.

In this paper, we analyze GWTW algorithms [1]. GWTW is a class of randomized optimization algorithms. It was first introduced and analyzed in [1]. GWTW algorithms can be viewed as an abstraction of certain genetic algorithms, which use mutation and reproduction rules. Dimitriou and Impagliazzo [7, 8] apply a modified version of GWTW to the graph bisection problem on random graphs. Peinado and Lengauer analyze parallel [22] and random generation [23] versions of GWTW.

Our results are threefold. Let $0 < \epsilon < 1$. We prove that the probability that the GWTW algorithm finds a clique of size $\lceil (1 + \epsilon) \log n \rceil$ does not exceed $T/n^{\Omega(\log n)}$, on almost every graph $G \in \mathcal{G}(n, 1/2)$, where T is the running time of the algorithm. Thus, unless the running time is set to be superpolynomial ($n^{\Omega(\log n)}$), GWTW is unlikely to find a large clique. Our second result is a matching upper bound. We show that GWTW will find a clique of size $\lceil (1 + \epsilon) \log n \rceil$ in time $n^{O(\log n)}$. Our third result extends the lower bound to the uniform distribution over the set of graphs, which contain a clique of size $\lceil n^\beta \rceil$, where $\beta < 1/2$. Distributions of this type have been studied by several authors in connection with different algorithms [2, 12].

A standard way to boost the success probability of a Monte Carlo algorithm is to run it many times with independent random bits. In many applications, the Monte Carlo algorithm is a randomized approximation algorithm for some hard combinatorial optimization problem and a ‘success’ is the event that the algorithm finds a solution of a given quality.

The GWTW algorithm introduces interactions among different runs which are amenable to rigorous analysis. The idea is to monitor the progress of the different runs at intermediate steps. Runs without hope of succeeding are aborted and replaced by copies of runs which appear to be making good progress. Aldous and Vazirani prove that, depending on a certain parameter κ (determined by the concrete application), GWTW can achieve an exponential speedup over independent runs. The main task in bounding the running time of any concrete instantiation of the algorithm is to compute κ .

In light of the fact that, under the uniform distribution, the size of the largest clique in a graph is only about $2 \log n$ with high probability, several authors consider the uniform distribution $\mathcal{G}(n, 1/2, q)$ over all n -vertex graphs, which contain a clique of size $q = \lceil n^\beta \rceil$ ($0 < \beta < 1$) [2,12]. For $\beta > 1/2$, the large clique is revealed by the degree sequence. More sophisticated methods have led to polynomial time algorithms even for $\beta = 1/2$ [2,10]. However, no polynomial-time solution is known for $\beta < 1/2$, and Jerrum's extended challenge [12] remains open: Is there a randomized, polynomial-time algorithm capable of finding a clique of size $1.01 \log_2 n$ with probability $1/2$ over random graphs containing a clique of size $n^{0.49}$?

Krivelevich and Vu [17] describe an algorithm with worst-case approximation ratio $O(\sqrt{np}/\log n)$, whose expected running time over $\mathcal{G}(n, p)$ is polynomial. A survey of experimental studies of the clique problem can be found in [13]. Possible cryptographic implications of the hardness of the clique problem on random graphs have been studied in [14].

Intuitively, our results rely on two properties of random graphs and of GWTW: (a) The fraction of 'relevant' cliques in random graphs is $n^{-\Theta(\log n)}$. The term 'relevant' refers either to the large cliques of size $(1 + \epsilon) \log n$, which we would like to find, or to certain smaller cliques ('gateways' in [12]) the algorithm has to traverse, in order to find a clique of size $(1 + \epsilon) \log n$. (b) GWTW does not have strong bias in favor of or against these relevant cliques. For $k \leq \log n$, all cliques of size k have about the same probability of being found by GWTW. When combined, properties (a) and (b) yield the upper bound and the lower bound.

The rest of the paper is organized as follows. Section 2 gives a detailed description of the general GWTW framework and of how we apply it to the clique problem. Section 3 analyzes the relative probabilities of different cliques of the same size being found by GWTW. The results of Section 3 establish property (b) and are the basis for the bounds in the following sections. Section 4 contains the proof of the first main result: the lower bound on the running time needed to achieve constant success probability. Section 5 proves the matching upper bound. Section 6 extends the lower bound to $\mathcal{G}(n, 1/2, q)$ (for $q < n^\beta$, $\beta < 1/2$). Section 7 contains conclusions and suggestions for how the results might be extended.

Due to the page limit, significant parts of the the proofs had to be omitted from this version of the paper. A complete version of the paper is available [21].

1.1 Notation

As stated above, given a positive integer n and $p \in (0, 1)$, let $\mathcal{G}(n, p)$ denote the following probability distribution on the set Ω of labeled undirected n -vertex graphs

$$\Pr(G) = p^{|E|}(1-p)^{\binom{n}{2}-|E|}$$

for any graph $G = (V, E) \in \Omega$. Intuitively, this distribution results if the edges exist independently and with probability p . For $p = 1/2$, this is the uniform

distribution. We will use $G \in \mathcal{G}(n, p)$ to denote that the graph G is generated according to distribution $\mathcal{G}(n, p)$. We will use the phrase *for almost every (a.e.) graph* $G \in \mathcal{G}(n, p)$ to express that a graph property has probability $1 - o(1)$ under $\mathcal{G}(n, p)$.

Given $G = (V, E)$ and $W \subseteq V$, let $N(W)$ denote the number of neighbors of W . That is, $N(W)$ is the number of vertices in $V \setminus W$, which are adjacent to all vertices in W . Given G and $k \in \mathbb{N}$, let $\mathcal{C}_k = \mathcal{C}_k(G)$ denote the set of k cliques in G . Let $\mathcal{D}_k = \mathcal{D}_k(G)$ be the set of ordered sequences of k vertices from V , which (if the order is ignored) form k cliques. Given a probability space and an event A , let $\mathbb{1}_A$ denote the indicator random variable of A . Typically, in this paper, the (finite) probability space will be $(\Omega, \mathcal{G}(n, 1/2))$ and A will be a graph property. In this case, $\mathbb{1}_A(G) = 1$ if the random variable $G \in \mathcal{G}(n, 1/2)$ has property A . Otherwise, $\mathbb{1}_A = 0$.

In addition to $(\Omega, \mathcal{G}(n, 1/2))$, we consider the probability space given by the coin flips of the GWTW algorithm. At any given point, we will be working in only one of the two spaces. We separate the two spaces in our analysis by proving that certain graph properties hold with high probability under $\mathcal{G}(n, p)$, and basing the subsequent analysis of GWTW on these properties.

Throughout the paper, we will denote individual vertices by lower-case letters (typically u, v, w). Vertex sets will be denoted by capital letters (e.g. C, V, W), and collections of sets will be denoted by script font (e.g. $\mathcal{C}_k, \mathcal{D}_k$). Given a positive number x , we will use $\log x$ to denote the base 2 logarithm of x and $\ln x$ for the natural logarithm (base e). The asymptotic notation used throughout this paper (e.g. $o(1)$, $\Omega(\log n)$) refers to the variable n (number of vertices in the input graph). In order to simplify the notation, we have largely omitted the symbols for rounding to the next integer. For example, we write $\log n$ rather than $\lceil \log n \rceil$.

2 The Go with the Winners Algorithm

Let an instance of maximization problem be given by a set S of states and by a function $f : S \rightarrow \mathbb{N}$, which assigns a value to each state. The task is to find states $s \in S$, for which $f(s)$ is large. A local search algorithm for a given maximization problem typically defines a set of transitions T (directed edges) between states of S and rules for making state transitions on the resulting directed graph (S, T) .

Aldous and Vazirani [1] consider the case, in which the edges define a tree or a directed acyclic graph (DAG) on the states (vertices), such that all states s at any given depth have the same value $f(s)$, and $f(s)$ increases in the depth. The current state of the algorithm can be interpreted as a *particle* starting at the root of the tree (initial state) and moving down the tree towards a leaf vertex. From any given non-leaf vertex v , the algorithm can move to any of its children w according to some probability distribution $p(w|v)$. It is the goal of the algorithm to find a deep vertex in the tree. This concept leads to the randomized greedy algorithm and is summarized in [1, Alg. 0]: **Greedy**: “Start at the root, repeatedly choose a child at random until reaching a leaf, then stop.”

The success probability of this algorithm can be increased through many independent runs. If a success is defined as finding a vertex at depth d , $\Theta(1/a(d))$ independent runs are needed to make the success probability constant, where $a(d)$ is the probability that a single run reaches depth d . Before we can describe the ‘go with the winners’ alternative to independent runs, we have to define some notation.

Let V_d be the set of vertices at depth d . Given a vertex v , let $p(v)$ be the probability that Algorithm 0 visits vertex v and let $a(d) = \sum_{v \in V_d} p(v)$ be the probability that it reaches at least depth d . We also need the corresponding conditional probabilities $p(w|v)$, the probability that the particle visits vertex w given that it is at vertex v , and $a(d|v) = \sum_{w \in V_d} p(w|v)$, the probability that the particle reaches depth d given that it is at vertex v . Now consider the following alternative to independent runs [1, Alg. 2]:

GWTW: Repeat the following procedure, starting at stage 0 with B particles at the root: At stage i there are a random number of particles all at depth i . If all the particles are at leaves then stop. Otherwise, for each particle at a non-leaf, add at that particle’s position a random number of particles, this random number having distribution $R(\theta_i^{-1} - 1)$. Particles at leaf vertices are removed. Finally, move each particle from its current position to a child chosen at random.

The θ_i are real constants $0 < \theta_i \leq 1$ (to be specified) and $R(c)$ is an integral valued random variable with expectation c : $\mathbf{Pr}(R(c) = \lfloor c \rfloor) = \lceil c \rceil - c$ and $\mathbf{Pr}(R(c) = \lceil c \rceil) = c - \lfloor c \rfloor$. The goal is to keep the expected number of particles constant over the stages. This is the case if $\theta_i = a(i+1)/a(i)$. The $a(i)$ are generally not directly available. However, [1] describes how Algorithm 2 itself can be used to sample the θ_i . In this case, the θ_i become random variables. For simplicity, we do not make this fact explicit in our notation and treat the θ_i as constants. This does not impact our results.

It is shown in [1] that the probability that the algorithm does not reach the desired depth d is

$$p_{AV} = O\left(B^{-1} \kappa d^4 \left(1 + \frac{1}{\beta d}\right)\right) \quad \text{where} \quad (1)$$

$$\kappa = \max_{0 \leq i < j \leq d} \kappa_{i,j} \quad , \quad \kappa_{i,j} = \frac{a(i)}{a^2(j)} \sum_{v \in V_i} p(v) a^2(j|v) \quad \text{and} \quad (2)$$

$$\beta = \min_{0 \leq i < d} a(i+1)/a(i) \quad . \quad (3)$$

Intuitively, κ measures the ‘imbalance’ of the tree: Given i, j ($j > i$), consider the probability of reaching depth j from depth i as a random variable which takes value $a(j|v)$ with probability $p(v)$ for any $v \in V_i$. $\kappa_{i,j}$ is the second moment of this random variable divided by the square of its expectation – thus providing a measure for its deviation from its expectation. κ is the main parameter used to characterize the tree. Under the assumption that $\beta = \Omega(1/d)$, (1) means that choosing $B = \Theta(\kappa d^4)$ is sufficient to give the algorithm a constant probability of success.

Let $s_j = \prod_{i=1}^{j-1} \theta_i$. Let X_v be the number of particles that reach vertex v and let $S_i = \sum_{v \in V_i} X_v$ be the number of particles at depth i at the start of stage i . It is shown in [1] that

$$\mathbf{E}X_v = p(v)B/s_k . \quad (4)$$

That is, the expected number of GWTW particles in a tree vertex v is proportional to the probability $p(v)$ that the greedy algorithm visits v .

2.1 GWTW for the Clique Problem

We apply ‘go with the winners’ interactions to the well known randomized greedy algorithm: *Start with the empty clique. At stage i randomly select one vertex from those which are connected to all $i - 1$ vertices gathered so far. If no such vertex exists then stop.*

This leads to the definition of a GWTW tree (for each graph G), whose vertices are ordered cliques. More precisely, the root of the tree is the empty set. The set of tree vertices at depth $k > 0$ is $\mathcal{D}_k(G)$. Let the edges of the tree be defined as follows: There exists an edge between an (ordered) clique $C \in \mathcal{D}_k(G)$ and an (ordered) clique $D \in \mathcal{D}_{k+1}(G)$ if and only if D extends C by one vertex. The edge is labeled with $p(D|C) = 1/N(C)$.

Equivalently, the process can be modeled by a directed acyclic graph (DAG). The single source of the DAG is the empty set. The set of DAG vertices at distance k from the source is the set $\mathcal{C}_k(G)$ of k cliques of G . There exists an edge between cliques $C \in \mathcal{C}_k(G)$ and $D \in \mathcal{C}_{k+1}(G)$ if and only if C is a subset of D . Again, $p(D|C) = 1/N(C)$. The DAG model can be obtained from the tree model by merging all $k!$ tree vertices, which are permutations of the same k clique. Thus, for any k and $C \in \mathcal{C}_k$, $p(C) = \sum_i p(C_i)$, where the sum goes over all permutations $C_i \in \mathcal{D}_k$ of the vertices in C .

3 Path Probabilities

In this section, we derive property (b) from the introduction. For $k \leq \log n$, all cliques $C \in \mathcal{C}_k$ have approximately the same probability of being found by GWTW. More precisely, we will show that $\max\{p(C) : C \in \mathcal{C}_k\} \leq \min\{p(C) : C \in \mathcal{C}_k\} \cdot \alpha_k$, where α_k is a small factor defined below. The analysis is easier in the tree model than in the DAG model of GWTW. Consider $D \in \mathcal{D}_k$ and note that $p(D)$ is simply the probability of staying on the path to D at every depth between the root and D . More precisely, given an arbitrary graph G , $k \in \mathbb{N}$ and $D = (d_1, \dots, d_k) \in \mathcal{D}_k(G)$, let $C_i = C_i(D) = \bigcup_{j=1}^i \{d_j\}$ for $1 \leq i \leq k$. The C_i identify the nodes of the GWTW tree, which lie on the path from the root to D . By the definition of the GWTW tree and by the independence of the coin flips of GWTW,

$$p(D) = \prod_{i=0}^{k-1} p(C_{i+1}|C_i) = \frac{1}{\prod_{i=0}^{k-1} N(C_i)} . \quad (5)$$

Thus, the main task is to derive tight bounds on the neighborhood size $N(C_i)$ of cliques of size up to $\log n$. Our main tool will be Chernoff bounds. Lemma 1 bounds the neighborhood sizes for all depths up to $\log n - 4 \log \log n$. Proposition 1 derives a somewhat weaker bound for the remaining $4 \log \log n$ depths up to $\log n$. Finally, Lemma 2 combines the bounds on $N(C_i)$ into a bound on $p(D)$.

Note that for $G = (V, E) \in \mathcal{G}(n, 1/2)$, any k and $C \in \mathcal{C}_k(G)$,

$$N(C) = \sum_{a \in V \setminus C} \mathbb{1}_{\forall b \in C: \{a, b\} \in E} .$$

That is, $N(C)$ is the number of vertices $a \in V \setminus C$, which are adjacent to all vertices in C . By linearity of expectation,

$$\mathbf{E}N(C) = \sum_{a \in V \setminus C} \mathbf{Pr}(\forall b \in C: \{a, b\} \in E) = (n - k)2^{-k} . \quad (6)$$

Lemma 1. *The following statement holds for a.e. $G = (V, E) \in \mathcal{G}(n, 1/2)$. For all $k \leq \log n - 4 \log \log n$ and all $C \in \mathcal{C}_k(G)$:*

$$(1 - \epsilon)(n - k)2^{-k} \leq N(C) \leq (1 + \epsilon)(n - k)2^{-k} ,$$

where $\epsilon = 2/\log n$.

Proof. $N(C)$ is the sum of independent random $(0, 1)$ -variables. Thus, we can apply Chernoff bounds [19, Thm. 4.1, 4.2, 4.3]. Let $0 < \epsilon < 1/10$, and use l as an abbreviation for $\log n - 4 \log \log n$.

$$\begin{aligned} & \mathbf{Pr}(\exists k \leq l \exists C \in \mathcal{C}_k : N(C) < (1 - \epsilon) \mathbf{E}N(C) \text{ or } N(C) > (1 + \epsilon) \mathbf{E}N(C)) \\ & \leq \sum_{k \leq l} \sum_{C \in \mathcal{C}_k} (\mathbf{Pr}(N(C) < (1 - \epsilon) \mathbf{E}N(C)) + \mathbf{Pr}(N(C) > (1 + \epsilon) \mathbf{E}N(C))) \\ & \leq 2 \sum_{k \leq l} \sum_{C \in \mathcal{C}_k} e^{-\epsilon^2(n-k)2^{-k}/3} \leq 2 \sum_{k \leq l} \left(\frac{ne}{k}\right)^k e^{-\epsilon^2(n-k)2^{-k}/3} \end{aligned}$$

Let $p_k = (ne/k)^k e^{-\epsilon^2(n-k)2^{-k}/3}$. It is easily verified that $p_i \leq p_k$ for $i \leq k \leq l$ and that $p_l = o(1/\log n)$. Thus, $p_k = o(1/\log n)$ for all $k \leq l$ and $\sum_{k \leq l} p_k = o(1)$. $\square$

The next proposition bounds the neighborhood sizes for the depths not covered by the previous lemma.

Proposition 1. *For a.e. $G \in \mathcal{G}(n, 1/2)$: For all k , such that $\log n - 4 \log \log n < k \leq \log n$, all $D \in \mathcal{D}_k(G)$, and all $i < k$,*

$$(n - i)2^{-i}/(2 \log^2 n) \leq N(C_i(D)) \leq (n - i)2^{-i} 2 \log^2 n . \quad (7)$$

The proof (omitted) is based on Chernoff bounds. Let

$$\alpha_k = \begin{cases} e^4 & \text{for } k \leq \log n - 4 \log \log n \\ e^4 (\log n)^{16 \log \log n + 9} & \text{for } \log n - 4 \log \log n < k \leq \log n \end{cases} \quad (8)$$

The next lemma combines the bounds on the neighborhood sizes of Lemma 1 and Prop. 1 into a bound on $p(D)$.

Lemma 2. *The following statement holds for a.e. $G \in \mathcal{G}(n, 1/2)$. For all $k \leq \log n$*

$$\begin{aligned} \max\{p(D) : D \in \mathcal{D}_k\} &\leq \min\{p(D) : D \in \mathcal{D}_k\} \cdot \alpha_k \quad \text{and} \\ \max\{p(C) : C \in \mathcal{C}_k\} &\leq \min\{p(C) : C \in \mathcal{C}_k\} \cdot \alpha_k \end{aligned}$$

Proof. The statement follows from (5) by multiplying the bounds from Lemma 1 and from Proposition 1. $\square$

In summary, for all $0 \leq k \leq \log n$, and all cliques C at the same depth k , $p(C)$ is bounded within a small constant factor for $k < \log n - 4 \log \log n$ and within a small sublinear factor for $\log n - 4 \log \log n < k \leq \log n$.

4 Lower Bound

Given any graph $G = (V, E)$, let $\mathcal{C}_{k,m}$ be the set of k -cliques, which are subcliques of m -cliques in G (for $1 \leq k \leq m \leq n$). Note that, for any $C \in \mathcal{C}_k$, $a(m|C) = 0$ unless $C \in \mathcal{C}_{k,m}$. While the particles of GWTW will find an abundance of k -cliques $C \in \mathcal{C}_k$ at stage k , we will show that, with high probability, not a single particle will find a clique from $\mathcal{C}_{k,m}$ for $k \approx \log n$ and $m = \lceil (1 + \epsilon) \log n \rceil$. Since not a single one of the k -cliques found at stage k can be extended into an m -clique, GWTW will not find an m -clique. The next lemma shows that $\mathcal{C}_k$ is a tiny subset of $\mathcal{C}_{k,m}$ (cf. property (a) in the introduction). The subsequent theorem, combines this fact with the path properties established in Sect. 3.

Lemma 3. *For a.e. $G \in \mathcal{G}(n, 1/2)$ and $1 \leq k \leq m \leq (2 - \epsilon) \log n$ ($\epsilon > 0$):*

$$|\mathcal{C}_{k,m}| \leq |\mathcal{C}_k| (1 + o(1)) \binom{n-k}{m-k} 2^{-(\binom{m}{2} + \binom{k}{2})}.$$

Proof. Consider a graph $G = (V, E) \in \mathcal{G}(n, p)$. We estimate the ratio of $|\mathcal{C}_{k,m}|$ and $|\mathcal{C}_k|$ by computing the ratio of the expectations and using sharp concentration results. It is well known [3] that $\mathbf{E}|\mathcal{C}_k| = \binom{n}{k} 2^{-\binom{k}{2}}$. Thus,

$$\binom{m}{k} \mathbf{E}|\mathcal{C}_m| = \binom{m}{k} \binom{n}{m} 2^{-\binom{m}{2}} = \binom{n-k}{m-k} 2^{-(\binom{m}{2} + \binom{k}{2})} \mathbf{E}|\mathcal{C}_k|. \quad (9)$$

It remains to bound the deviation of each of the two random variables from its mean. It can be shown by means of the second moment method that, with

high probability, $|\mathcal{C}_k|$ lies within an arbitrarily small factor of its expectation. More precisely, Chebyshev's inequality in connection with well-known bounds on $\text{var}|\mathcal{C}_k|$ and $\mathbf{E}|\mathcal{C}_k|$, see [4, p. 254, Eq. 4], implies that, for a.e. $G \in \mathcal{G}(n, 1/2)$, $|\mathcal{C}_k| = (1 \pm o(1)) \mathbf{E}|\mathcal{C}_k|$. Combining this with (9), we obtain

$$\binom{m}{k} |\mathcal{C}_m| \leq (1 + o(1)) \binom{n-k}{m-k} 2^{-\binom{m}{2} + \binom{k}{2}} |\mathcal{C}_k|.$$

It remains to observe that $|\mathcal{C}_{k,m}| \leq \binom{m}{k} |\mathcal{C}_m|$.

Theorem 1. *For a.e. $G \in \mathcal{G}(n, 1/2)$: The probability that GWTW when run with $B \geq 1$ particles finds a clique of size $\lceil (1 + \epsilon) \log n \rceil$ ($\epsilon > 0$) is at most $T/n^{\Omega(\log n)}$, where T is the number of particle moves performed by the algorithm.*

Proof. For $\epsilon > 0$, we apply Lemma 3 to $m = \lceil (1 + \epsilon) \log n \rceil$ and $k = \lceil \log n - 4 \log \log n \rceil$. Elementary calculations show that

$$|\mathcal{C}_{k,m}| \leq |\mathcal{C}_k| n^{-(1-o(1))\epsilon^2 \log n / 2} \quad (10)$$

holds for a.e. $G \in \mathcal{G}(n, 1/2)$. For the rest of this proof, we will assume that the input graph G to GWTW has property (10) as well as the properties stated in Lemmas 1 and 2. Note that a.e. $G \in \mathcal{G}(n, 1/2)$ has these properties.

For a given input graph G , let $S_{k,m}$ denote the number of GWTW particles, which reach nodes of $\mathcal{C}_{k,m}$ at depth k . Note that $S_{k,m}$ is a random variable in the probability space given by the random coin flips of GWTW. We are considering $S_{k,m}$, because GWTW will fail to find a clique of size $m = \lceil (1 + \epsilon) \log n \rceil$ if $S_{k,m} = 0$, even if S_k is large. We will show that $\Pr(S_{k,m} \geq 1)$ is small. For this purpose, we bound $\mathbf{E}S_{k,m}$ in terms of $\mathbf{E}S_k$ and show that the two random variables are unlikely to deviate significantly from their respective expectations.

$$\begin{aligned} \mathbf{E}S_{k,m} &= \mathbf{E} \sum_{C \in \mathcal{C}_{k,m}} X_C = \sum_{C \in \mathcal{C}_{k,m}} \mathbf{E}X_C = \sum_{C \in \mathcal{C}_{k,m}} p(C)B/s_k \quad \text{by (4)} \\ &\leq B/s_k |\mathcal{C}_{k,m}| \max\{p(C) : C \in \mathcal{C}_k\} \\ &\leq B/s_k |\mathcal{C}_k| \min\{p(C) : C \in \mathcal{C}_k\} n^{-\frac{(1-o(1))\epsilon^2}{2} \log n} \quad \text{by (10) and Lemma 2} \\ &\leq n^{-\frac{(1-o(1))\epsilon^2}{2} \log n} \sum_{C \in \mathcal{C}_k} p(C)B/s_k = n^{-\frac{(1-o(1))\epsilon^2}{2} \log n} \mathbf{E}S_k \end{aligned}$$

Hence $\Pr(S_{k,m} \geq 1) \leq \mathbf{E}S_{k,m} \leq n^{-\frac{(1-o(1))\epsilon^2}{2} \log n} \mathbf{E}S_k$.

It remains to show that the work GWTW has to perform is not significantly smaller than $\mathbf{E}S_k$. We measure the work of GWTW in ‘particle moves’, i.e. in the number of times GWTW has to move a particle from a given tree node to one of its children. By Lemma 1, for a.e. $G \in \mathcal{G}(n, 1/2)$, no leafs exist in the first $\log n - 4 \log \log n$ depth levels of the GWTW tree. Thus, every particle, which is generated during the first $\log n - 4 \log \log n$ stages of the algorithm, reaches

depth $\log n - 4 \log \log n$. A very crude bound on the (random) number of added particles will suffice. Let $M = B \prod_{i=1}^k \lfloor \theta_i^{-1} \rfloor$. Then

$$M = B \prod_{i=1}^k \lfloor \theta_i^{-1} \rfloor \leq S_k \leq B \prod_{i=1}^k (\lfloor \theta_i^{-1} \rfloor + 1) \leq nM.$$

Thus, $\mathbf{E}S_k \leq Mn$ and $\Pr(S_{k,m} \geq 1) \leq n^{-\frac{(1-o(1))\epsilon^2}{2} \log n} M$. It remains to notice that the algorithm has to perform at least M particle moves. $\square$

5 Upper Bound

We will use the following crude bound for κ :

$$\kappa_{i,j} = \frac{a(i)}{a^2(j)} \sum_{D \in \mathcal{D}_i} p(D) a^2(j|D) \leq \frac{a(i)}{a^2(j)} \sum_{D \in \mathcal{D}_i} p(D) a(j|D) = \frac{a(i)}{a(j)} \quad (11)$$

Recall the definition of α_k from (8).

Proposition 2. *For a.e. $G \in \mathcal{G}(n, 1/2)$: If $j \leq \log n$ then $\kappa_{ij} \leq 2\alpha_j$.*

Theorem 2. *For a.e. $G \in \mathcal{G}(n, 1/2)$, if GWTW is run on input G , it will find a clique of size $(2 - \delta) \log n$ ($\delta > 0$) in time $n^{O(\log n)}$ with probability $1 - o(1)$.*

6 Generalizations

As in the case of the Metropolis process, the lower bound for $\mathcal{G}(n, 1/2)$ holds for a broader range of distributions. Jerrum [12] shows that the Metropolis algorithm is unlikely to find a large clique even if the graphs are generated from the following distributions, under which graphs are guaranteed to have large cliques.

For $1 \leq q \leq n$, let $\mathcal{G}(n, 1/2, q)$ denote the probability space given by the uniform distribution over the set of all n -vertex graphs, which contain a clique of size q . Let $\mathcal{G}'(n, 1/2, q)$ denote the probability space over n -vertex graphs, defined by the following graph generation procedure: (1) Select a random subset Q of size q from the set of n vertices, and make Q a clique by inserting all edges between pairs of vertices from Q . (2) Each of the remaining edges exists with probability $1/2$ independently of all other edges. Given $G \in \mathcal{G}'(n, 1/2, q)$ and $1 \leq i \leq n$, let $\mathcal{C}_i^0 = \{C \in \mathcal{C}_i : |C \cap Q| = 0\}$ be the set of cliques from $\mathcal{C}_i$, which do not intersect Q . Similarly, given k, m , let $\mathcal{C}_{k,m}^0 \subseteq \mathcal{C}_{k,m}$ denote the set of cliques from $\mathcal{C}_{k,m}$, which do not intersect Q .

The proofs of Lem. 4 and Thm. 3 use elements of the analysis of Sect. 4 and of [12].

Lemma 4. *Let $0 < \epsilon < 1$, $0 < \beta < 1/2$, $k = \lceil \log n - 4 \log \log n \rceil$ and $m = \lceil (1 + \epsilon) \log n \rceil$. For a.e. $G \in \mathcal{G}'(n, 1/2, \lceil n^\beta \rceil)$, $|\mathcal{C}_{k,m}| \leq n^{-\Omega(\log n)} |\mathcal{C}_k^0|$.*

Theorem 3. *For any $\epsilon > 0$, $0 < \beta < 1/2$ and a.e. $G \in \mathcal{G}(n, 1/2, \lceil n^\beta \rceil)$: The probability that GWTW when run with $B \geq 1$ particles finds a clique of size $\lceil (1 + \epsilon) \log n \rceil$ ($\epsilon > 0$) is at most $T/n^{\Omega(\log n)}$, where T is the number of particle moves performed by the algorithm.*

7 Conclusions

The paper shows that GWTW requires $n^{\Theta(\log n)}$ steps to find cliques of size $(1 + \epsilon) \log n$ ($\epsilon > 0$) in random graphs, by providing upper and lower bounds. Thus, GWTW can be included in the list of clique-finding algorithms, such as the Metropolis process [12] or the greedy algorithm [11], which are known not to solve the challenge of [12]. An interesting observation in the analysis leading to the lower bound (Sect. 4) is the lack of effectiveness of the GWTW interactions between particles. On the GWTW trees, which arise from random graphs, GWTW behaves largely like the greedy algorithm.

The results can be generalized in several directions. One way to generalize the lower bound is to consider different edge densities, i.e. distributions of the form $\mathcal{G}(n, p)$, where $0 < p < 1$ is not necessarily $1/2$. This generalization is analogous to [12, Sect. 5]. Let $0 < p < 1$ be constant. The size of the maximum clique in a.e. $G \in \mathcal{G}(n, p)$ is $2 \log_{1/p} n + O(\log \log n)$ [4, p. 255]. No algorithm is known to find cliques of size $(1 + \epsilon) \log_{1/p} n$ in polynomial time. The lower bound of Sect. 4 can be generalized to cover $\mathcal{G}(n, p)$ for any constant p . Details will appear in the full version of the paper.

In addition to broadening the class of distributions, it would be interesting to generalize the class of algorithms, to which the lower bound applies. It appears that GWTW, the Metropolis process [12] and the greedy algorithm [11] are defeated by similar properties of random graphs: The cliques of size k (where $k = \log n$ or slightly larger), which "lead" to cliques of size $(1 + \epsilon) \log n$ form only a tiny fraction ($1/n^{\Omega(\log n)}$) of the set of all k -cliques. Furthermore, none of the popular local search algorithms appears to have significant bias toward this small set of 'gateways'. It might be possible to prove a lower bound covering a broad class of local search algorithms.

Finally, one might analyze the GWTW algorithm for different types of trees. Several authors [1, 8] combine GWTW with simulated annealing, such that each step down the GWTW tree corresponds to a random walk. In the case of cliques, an analysis of a process of this type appears to require new techniques. In particular, it appears that the lower bound of [12] would have to be generalized to cover arbitrary initial states of the Metropolis algorithm.

References

1. D. Aldous and U. Vazirani. "Go with the winners" algorithms. In *Proceedings 35th IEEE Symposium on the Foundations of Computer Science*, pages 492–501. IEEE Computer Society Press, 1994. 525, 526, 528, 529, 530, 535
2. N. Alon, M. Krivelevich, and B. Sudakov. Finding a large hidden clique in a random graph. *Random Structures and Algorithms*, 13:457–466, 1998. 526, 527
3. N. Alon and J. Spencer. *The Probabilistic Method*. Wiley, 1992. 532
4. B. Bollobás. *Random Graphs*. Academic Press, 1985. 533, 535
5. B. Bollobás and P. Erdős. Cliques in random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 80:419–427, 1976. 525
6. R. Boppana and M. M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT*, 32:180–196, 1992. 525

7. T. Dimitriou and R. Impagliazzo. Towards an analysis of local optimization algorithms. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 304–313. ACM Press, 1996. 526
8. T. Dimitriou and R. Impagliazzo. Go with the winners for graph bisection. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 510–520, San Francisco, California, January 1998. SIAM. 526, 535
9. L. Engebretsen and J. Holmerin. Clique is hard to approximate within $n^{1-o(1)}$. In *Proceedings of the 27th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 2–12. Springer-Verlag, 2000. 525
10. U. Feige and R. Krauthgamer. Finding and certifying a large hidden clique in a semirandom graph. *Random Structures and Algorithms*, 16:195–208, 2000. 527
11. G. R. Grimmett and C. J. H. McDiarmid. On coloring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77:313–324, 1975. 526, 535
12. M. Jerrum. Large cliques elude the Metropolis process. *Random Structures and Algorithms*, 3(4):347–360, 1992. 526, 527, 534, 535
13. D. S. Johnson and M. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. American Mathematical Society, 1996. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 525, 527
14. A. Juels and M. Peinado. Hiding cliques for cryptographic security. *Designs, Codes and Cryptography*, 20(3):269–280, July 2000. 527
15. R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 525
16. R. M. Karp. Probabilistic analysis of some combinatorial search problems. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, 1976. 526
17. M. Krivelevich and V. Vu. Approximating the independence number and chromatic number in expected polynomial time. In *Proceedings of the 27th International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 13–24. Springer-Verlag, 2000. 527
18. D. Matula. The largest clique size in a random graph. Technical Report CS 7608, Southern Methodist University, 1976. 525
19. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 531
20. M. Peinado. Improved lower bounds for the randomized Boppana-Halldórsson algorithm for MAXCLIQUE. In *Proceedings of the First Annual Computing and Combinatorics Conference*, pages 549–558. Springer-Verlag, 1995. 526
21. M. Peinado. Go with the winners algorithms for cliques in random graphs. <http://cs-people.bu.edu/mpe/publications.html>, 2001. 527
22. M. Peinado and T. Lengauer. ‘Go with the winners’ generators with applications to molecular modeling. In J. Rolim, editor, *Proceedings of the 1st International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 135–149. Springer-Verlag, 1997. Lecture Notes in Computer Science 1269. 526
23. M. Peinado and T. Lengauer. Parallel ‘Go with the winners’ algorithms on the LogP model. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 656–664, Los Alamitos, California, 1997. IEEE Computer Society Press. 526

24. J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 627–636. IEEE Computer Society Press, 1996. 525

Complexity of Partial Covers of Graphs

Jiří Fiala ^{*} and Jan Kratochvíl ^{**}

Charles University, Department of Applied Mathematics and
Institute for Theoretical Computer Science ^{***}
Malostranské nám 25, Prague, Czech Republic
{fiala,honza}@kam.mff.cuni.cz

Abstract. A graph G *partially covers* a graph H if it allows a locally injective homomorphism from G to H , i.e. an edge-preserving vertex mapping which is injective on the closed neighborhood of each vertex of G . The notion of partial covers is closely related to the generalized frequency assignment problem. We study the computational complexity of the question whether an input graph G partially covers a fixed graph H . Since this problem is at least as difficult as deciding the existence of a full covering projection (a locally bijective homomorphism), we concentrate on classes of problems (described by parameter graphs H) for which the full cover problem is polynomially solvable. In particular, we treat graphs H which contain at most two vertices of degree greater than two, and for such graphs we exhibit both *NP*-complete and polynomially solvable instances. The techniques are based on newly introduced notions of generalized matchings and edge precoloring extension of bipartite graphs.

1 Introduction

Given finite simple graphs G and H , a mapping $f : V(G) \rightarrow V(H)$ is a *graph homomorphism* if for any edge (u, v) of G , the pair $(f(u), f(v))$ is an edge of H . A homomorphism f is called a *partial covering projection* (or in other words, a *locally injective homomorphism*) if the mapping is injective on the closed neighborhood of every vertex of G . The notion is derived from the *full covering projection* (or a *locally bijective homomorphism*) which in addition maintains vertex degrees, i.e., in which case f acts as a bijection between the closed neighborhoods $N_G[u]$ and $N_H[f(u)]$, for every vertex u of G . See an illustrative example in Fig. 1.

The notion of a covering projection appeared in the early sixties in Conway's construction of infinitely many 5-arc-transitive cubic graphs [4]. Its first application in computer science were results of graph recognition by parallel networks

^{*} Supported in part by Czech Research grant GAČR 201/99/0242 and GAUK 158/99.

^{**} Supported in part by Czech Research grants GAUK 158/99 and KONTAKT ME338.

^{***} Supported by the Ministry of Education of the Czech Republic as project LN00A056.

of processors in [2,3,7,22]. An application to emulation of distributed computing is shown in [5], and in the same paper the author posed the question of the computational complexity of the existence of a full covering projection between two input graphs, and proved that it is at least as hard as testing graph isomorphism.

The decision problem H -COVER was defined in [1]. For a fixed graph H , it asks if an input graph G allows a covering projection onto H . Note that each graph H (up to isomorphism) represents a different H -COVER problem. The computational complexity of the H -COVER problems was investigated in [1,10,17,18,19], however, today there is still no good conjecture about the boundary between easy (i.e., polynomially solvable) and hard (NP -complete) H -COVER problems. The H -PARTIALCOVER problem was similarly defined in [10]. It asks if an input graph G allows a partial covering projection to a fixed graph H .

Partial covers are closely related to the notion of $\lambda_{(2,1)}$ -labeling of a graph G [10]. This is a labeling of vertices by integers such that the labels of adjacent vertices differ by at least 2 and vertices at distance 2 receive different labels. This is a simple model of the channel assignment problem and it has been intensively studied [6,13,14,24,26,27]. The relationship between covers and $\lambda_{(2,1)}$ -labelings was used in [11] as a tool for a complete classification of the computational complexity of existence of a $\lambda_{(2,1)}$ -labeling. This problem was recently examined also for planar graphs in [12].

Both the H -COVER and H -PARTIALCOVER problems are special cases of the so called $H(\sigma, \rho)$ -COLORING problems defined by Kristiansen and Telle [21]. For number sets σ and ρ , a vertex mapping $f : V(G) \rightarrow V(H)$ is an $H(\sigma, \rho)$ -coloring of G if for every vertex $u \in V(G)$, the neighbors of u are mapped into the closed neighborhood $N_H[f(u)]$, the number of neighbors of u mapped onto $f(u)$ is in σ and for every neighbor y of $f(u)$, the number of neighbors of u mapped onto y is in ρ . Though maybe somewhat artificial at first sight, this definition naturally arises from the concept of (σ, ρ) -DOMINATION in graphs (cf.[25]) and encaptures (besides other natural partition problems) all the above mentioned locally constrained homomorphism type problems: H -HOMOMORPHISM is $H(\{0\}, \{0, 1, \dots\})$ -COLORING and also the locally surjective homomorphism problem called H -COLORDOMINATION [21] is $H(\{0\}, \{1, 2, \dots\})$ -COLORING. For this last mentioned variant, the authors show that H -COLORDOMINATION is at least as difficult as the H -COVER problem, for every graph H , and they show several NP -completeness results for relatively simple parameter graphs H for which the H -COVER problem is polynomially solvable. They conjecture that H -COLORDOMINATION is NP -complete for every connected graph H with at least three vertices.

In this paper we deal with the complexity of the H -PARTIALCOVER problem. Similarly as in the case of locally surjective homomorphisms, the problem is at least as difficult as H -COVER (this is proved in [10]), and therefore we direct our attention to the simplest classes of polynomially solvable H -COVER problems, namely those defined by graphs H which contain at most two vertices of degree greater than two (for full characterization of polynomial and NP -complete

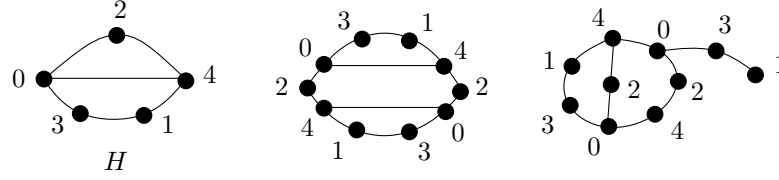


Fig. 1. Examples of full and partial covers

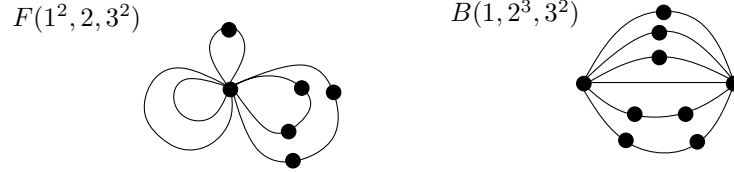


Fig. 2. Examples of F and B graphs

instances of H -COVER for such graphs see [17]). We show that already among these simple graphs one encounters nontrivial polynomial instances as well as interesting NP -complete cases of the H -PARTIALCOVER problem. The latter relate to a certain new edge-coloring result, while the polynomial cases are derived from a generalized matching algorithm whose formulation is also new.

The paper is organized as follows. In the next section we overview the results, and in Section 3 we introduce the techniques. Section 4 then shows application of generalized matchings to polynomial instances of H -PARTIALCOVER and Section 5 concludes with NP -completeness proofs. We omit several proofs due to space restrictions.

2 Overview of Results

We start with an example of covering and partial covering projections. In Fig. 1 left we depict the target graph H . This graph is later on referred to as $B(1, 2, 3)$ expressing the fact that the two vertices of degree greater than two are connected by paths of lengths 1, 2 and 3. In the middle we depict a full cover of $B(1, 2, 3)$ and in the right we see an example of a graph which partially covers $B(1, 2, 3)$.

The covering projections are visualized by writing the name of the image of each particular vertex near the source vertex. Note that while in the case of full covers, the preimages of vertices of degree 3 in the target graph have degree in the source graph, this is not necessarily the case for partial covers. This example also illustrates the connection between partial covers and $\lambda_{(2,1)}$ -labelings — the names of vertices of H were chosen so that the partial covering projection is exactly a $\lambda_{(2,1)}$ -labeling by labels $\{0, 1, 2, 3, 4\}$.

We will consider target graphs with one or two vertices of degree greater than two. In particular, for a k -tuple of positive integers $(a_1, \dots, a_k)$, we denote by $F(a_1, \dots, a_k)$ the graph with one vertex of degree $2k$ that is the unique intersection of k cycles of lengths $a_1, \dots, a_k$, and by $B(a_1, \dots, a_k)$ the graph with two vertices of degree k connected by k paths of lengths $a_1, \dots, a_k$. We use the abbreviated notation $(a_1^{i_1}, a_2^{i_2}, \dots, a_m^{i_m})$ for a k -tuple which contains the number a_j precisely i_j -times, for $j = 1, 2, \dots, m$ and $k = i_1 + i_2 + \dots + i_m$.

It was proved in [19] that the problems $F(a_1, \dots, a_k)$ -COVER and $B(a_1, \dots, a_k)$ -COVER are solvable in polynomial time for any k -tuple $(a_1, \dots, a_k)$. In contrast we prove here that the complexity of partial coverings of these graphs differs already when there appear only two distinct values among $(a_1, \dots, a_k)$:

Theorem 1. *The $F(a^i, b^j)$ -PARTIALCOVER problem is polynomially solvable for any positive integers a, b, i, j .*

Theorem 2. *The $B(a^i, b^j)$ -PARTIALCOVER problem is polynomially solvable if a and b are divisible by the same power of 2, or if $i + j \leq 2$, and NP-complete otherwise.*

We conjecture that $B(a_1^{i_1}, a_2^{i_2}, \dots, a_m^{i_m})$ -PARTIALCOVER is NP-complete whenever $m \geq 3$. We support the conjecture by the following fairly general results for the case of triples of three distinct integers:

Theorem 3. *The problems $B(1, 2, a)$ -PARTIALCOVER and $B(1, 3, b)$ -PARTIALCOVER are NP-complete for all positive integers $a > 2, b > 3$.*

More partial results about partial covers can be found in [9], but to keep this paper of reasonable length, we omit them here. We conclude this section by the following simple but useful lemma:

Lemma 1. *For a positive integer t , denote by H^t the graph that arises from a graph H by subdividing each edge by $t - 1$ extra new vertices. Then the H -PARTIALCOVER and H^t -PARTIALCOVER problems are polynomially equivalent.*

Hence in the complexity analysis of $F(a_1, \dots, a_k)$ -PARTIALCOVER and $B(a_1, \dots, a_k)$ -PARTIALCOVER we may assume that the parameter k -tuple $(a_1, \dots, a_k)$ is irreducible, i.e., the greatest common divisor of $a_1, \dots, a_k$ is 1.

3 Techniques

3.1 Flag Factors

Let G be a multigraph. A *flag* (sometimes also called a halfedge) is a pair $[u, e]$, where u is a vertex of G and e is an edge incident with u . We denote by $F(G) = \{[u, e] : u \in V, e \in E(G)\}$ the multiset of flags in G . Note that a loop in G gives rise to a double flag.

Suppose we are given sets of nonnegative integers I_u , for every vertex u , and for every edge $e = (u, v)$, a direction is chosen (say $[u, v]$) and a set find a set $S \subseteq F(G)$ of flags such that the number of flags of S emanating from a vertex u is in I_u , for every vertex u , and for every directed edge $e = [u, v]$, $|\{[u, e]\} \cap S|, |\{[v, e]\} \cap S| \in J_e$. In other words, the sets I_u represent permitted degrees of vertices in the ‘subgraph’ determined by S , and the sets J_e contain permitted characteristic vectors of S reduced to the flags that arise from the edge e . We refer to the problem of deciding the existence of S as the FLAG FACTOR problem. We observe (and use in the sequel) that the FLAG FACTOR problem is polynomially solvable if the permitted-degree sets are intervals, while the sets J_e may be arbitrary:

Lemma 2. *The FLAG FACTOR problem is solvable in polynomial time if all the sets $I_u, u \in V(G)$, are intervals (of integers).*

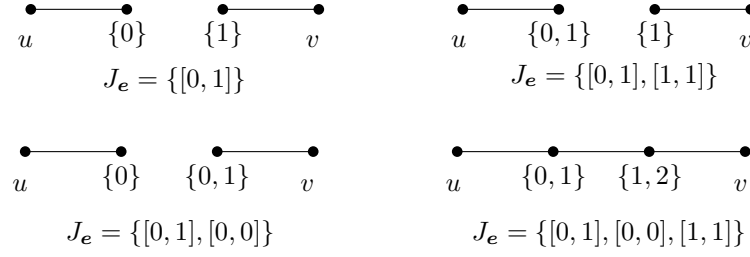
Proof. If $J_e = \{[0, 0], [1, 1]\}$ for each edge $e \in E(G)$, a flag factor contains either both or none of the halfedges $[u, e], [v, e]$, for every edge $e = (u, v)$. In this case a flag factor corresponds to a spanning subgraph of $G' \subset G$ such that $\deg_{G'}(u) \in I_u$ for every vertex u . If all I_u ’s are intervals, this question can be reduced to maximum matching and hence it is solvable in polynomial time, (cf. Exercise 10.2.2 in [23]).

We will show that by performing local reductions at edges of G , we can reduce a general instance to the above described situation. Since in the graph $\tilde{G}$ constructed in this way all sets $\tilde{J}_e$ will be the same (equal to $\{[0, 0], [1, 1]\}$), we only need to define the sets $\tilde{I}_u$:

1. Each vertex u of G will remain a vertex of $\tilde{G}$ and also $\tilde{I}_u = I_u$ for all $u \in V(G)$.
2. For edges $e \in E(G)$ such that $J_e = \{[0, 0], [1, 1]\}$, nothing is changed.
3. If the set J_e contains none or both the asymmetric pairs $[0, 1]$ and $[1, 0]$, then insert into e an extra new vertex x_e , and set $\tilde{I}_{x_e} = \{a + b, [a, b] \in J_e\}$.
4. In the last case, w.l.o.g. assume that $e = (u, v)$ allows exactly one asymmetric pair, say $[0, 1] \in J_e$. There are four possible cases for the set J_e , and the corresponding replacement rules of $e = [u, v]$ together with the definition of the intervals $\tilde{I}$ for the new vertices are depicted in Fig. 3.

We claim that a flag factor S exists in G if and only if the new graph $\tilde{G}$ contains a spanning subgraph G' , s.t. $\deg_{G'}(u) \in \tilde{I}_u$ for all vertices $u \in V(\tilde{G})$.

Suppose first that G' exists. Consider, e.g., a vertex x_e which was created by application of rule 3, i.e., x_e is subdividing an edge $e = (u, v)$. If both edges $(u, x_e), (v, x_e)$ belong to G' , we have $2 \in \tilde{I}_{x_e}$, which is only possible if $[1, 1] \in J_e$, and thus putting both flags $[u, e], [v, e]$ into S keeps the degrees of u and v and is compatible with J_e . If none of $(u, x_e), (v, x_e)$ belongs to G' , we have $0 \in \tilde{I}_{x_e}$, i.e., $[0, 0] \in J_e$, and thus leaving both flags $[u, e], [v, e]$ out of S keeps the degrees of u and v and is compatible with J_e . If just one of $(u, x_e), (v, x_e)$ belongs to G' ,

**Fig. 3.** The four cases of rule 4

say $(u, x_e) \in E(G')$, we have $1 \in \tilde{I}_{x_e}$, which means $\{[1, 0], [0, 1]\} \subseteq J_e$, and thus putting $[u, e]$ into S keeps the degrees of u and v and is compatible with J_e .

The case analysis of the application of the other rules is similar, and the opposite implication (the existence of a flag factor implies existence of G') is then straightforward.

3.2 Special Coloring Problems

Some of our NP -completeness proofs for B -PARTIALCOVER problem will be based on the following type of black/white coloring of graphs:

$BW(k, j)$

Input: A $(k + j)$ -regular graph G

Question: Does there exist a coloring of $V(G)$ with black and white colors s.t. each vertex is adjacent to exactly k vertices of its own color?

When k or j is zero or both are one, the problem is trivially solvable, but all other cases are NP -complete: The $BW(2, 1)$ problem was proven to be NP -complete in [16]. For the NP -completeness of the case of an even $k \geq 2$ and an arbitrary $j \geq 1$, see [17]. The remaining cases of an odd k can be treated similarly.

For a particular instance of the B -PARTIALCOVER problem we use the following result:

Proposition 1. [8,9] *The question, whether there exists a proper edge 3-coloring of a cubic bipartite graph extending a given precoloring is NP -complete.*

Note that this extends the NP -completeness result of Holyer [15], who proved that deciding edge-3-colorability of cubic graphs is NP -complete. Of course, we cannot expect that this problem would remain NP -complete for bipartite graphs (since every bipartite cubic graph is edge-3-colorable by the famous theorem of Petersen) and so the NP -completeness of the precolored version is all we may have hoped for. On the other hand, it is known that vertex-3-coloring of precolored perfect graphs is NP -complete [20], and thus Proposition 1 is a direct strengthening of this result for line graphs of bipartite graphs, one of the simplest subclasses of perfect graphs.

4 Proofs - the Polynomial Cases

For the convenience of the reader we restate the theorem:

Theorem 1 *The $F(a^i, b^j)$ -PARTIALCOVER problem is polynomially solvable for any positive integers a, b, i, j .*

Proof. We may assume that $i + j \geq 2$, since a partial covering of a cycle is obviously polynomial. In the proof, the order of parameters a, b does not matter, and we assume without loss of generality that $i \geq j$. Now, let G be the graph whose partial covering to $F = F(a^i, b^j)$ is questioned.

Assume that G is connected, otherwise we perform the computation separately for each component of G . If G is a cycle, then it covers F if and only if its length is a nonnegative linear combination of a and b (when $i, j \geq 1$) or a multiple of a (when $j = 0$). This question can be easily tested in linear time.

Now, assume G is not a cycle, and denote by v the central vertex of F . By the local injectivity, every vertex of G of degree at least three must be mapped onto v by any partial covering projection. It remains to find the images of vertices of degree at most two. Consider a maximal subpath in the graph G with both endpoints of degree at least three. We decide whether none, one or both terminal edges of the path can be mapped into a cycle of length a in F . This decision can be done in constant time, since the outcome depends only on the length l of the path, and for $l > ab$ all three cases are possible. Denote the set of all possible cases by $J(l)$, more formally, put $[0, 0] \in J(l)$ if the equation $l = pa + qb$ allows a nonnegative integer solution with $q \geq 2$, put $[1, 0], [0, 1] \in J(l)$ when $p, q \geq 1$, and finally, $[1, 1] \in J(l)$ if $p \geq 2$.

In G , replace each maximal subpath (whose internal vertices have all degree two) of length l by a single edge e , and set $J_e = \{[0, 0], [0, 1], [1, 0], [1, 1]\}$, when e ends in a vertex of degree one, and set $J_e = J(l)$ otherwise. Call the new multigraph G' .

Assign $I_u = [\max(\deg(u) - 2j, 0), \min(\deg(u), 2i)]$ to every vertex u of G' and ask whether a flag factor S for G' exists, with respect to the sets I_u and J_e . Due to Lemma 2 the question can be answered in polynomial time. If the result is negative, then G cannot partially cover $F(a^i, b^j)$, since the existence of a flag factor S is an obvious necessary condition.

We argue that this necessary condition is also sufficient for the existence of a partial covering projection. Suppose now that a flag factor S exists. We construct a partial covering projection as follows. Vertices of degree greater than two in G will map onto vertex v , and along each path (corresponding to an edge of G') we use a partial covering projection compatible with the flag factor S . E.g., if $S \cap \{[u, e], [w, e]\} = \{[u, e]\}$ for an edge $e = (u, w)$, the beginning of this $u - w$ path is mapped onto a cycle of length a and its end segment (near w) is mapped onto a cycle of length b . It needs to be shown that we can really distribute the a -cycles (and b -cycles) properly, i.e., we can say onto *what* a -cycle (b -cycle) a segment of a path is mapped.

To see this, direct the cycles of H cyclically and number the a -cycles $1, \dots, i$ and number the b -cycles $1, \dots, j$. There is a natural correspondence between the

flags in $F(G')$ and the edges of $G'^{:2}$. Let G'_a be the bipartite subgraph of $G'^{:2}$ restricted to the flags of S .

Further, let G''_a be the graph obtained from G'_a by replacing a path $u, [u, e], [w, e], w$ by the edge $e = (u, w)$ if both flags $[u, e], [w, e]$ belong to S . The definition of interval I_u guarantees that maximum degree in G''_a is $\leq 2i$. Then G''_a has an orientation with maximum indegree as well as maximum outdegree $\leq i$ (to see this, add edges to embed G''_a into a $2i$ -regular graph and direct its edges along an Euler circuit).

Now color the edges of G''_a with i colors so that each vertex has at most one outgoing and at most one ingoing arc of each color in the chosen orientation (this is possible by splitting each vertex into two - one being the endvertex of all incoming edges and the other one being the starting vertex of all outgoing edges - obtaining a bipartite graph of maximum degree i , which is edge- i -colorable by Petersen Theorem). This means that for each color, the subgraph of G''_a determined by edges of this color is a disjoint union of directed cycles and/or paths.

Now in G , we map the ‘outer’ vertices of paths corresponding to edges colored in G''_a by the color h onto the h -th a -cycle of H . E.g., if this a -cycle has vertices $v, x_1, \dots, x_{a-1}$ and an edge $e = (u, w)$ of G''_a is directed from u to w and corresponds to a path in G of length $l = pa + bq$ where $p \geq 2$ or $q = 0$, we map the vertices along this path (from u to w) onto $v, x_1, \dots, x_{a-1}$, then q -times onto arbitrary b -cycle (in the direction of the cycle), then $(p-1)$ -times onto $x_1, \dots, x_{a-1}$ and finally w is mapped onto v . Similarly, we handle the edges of G''_a that correspond halfedges of G' . Note that if an edge of G' gives rise to only one flag in S , the covering projection along its preimage path in G starts with a mapping onto a a -cycle and ends with a mapping onto a b -cycle.

In this way we guarantee that each vertex of degree greater than two in G has at most one neighbor mapped onto x_1 and at most one neighbor mapped onto x_{a-1} . Since this holds true for all $h = 1, 2, \dots, i$, and since a similar procedure works for the b -cycles, we see that the mapping constructed is indeed a partial covering projection.

Theorem 2 (polynomial part) *The $B(a^i, b^j)$ -PARTIALCOVER problem is polynomially solvable if a and b are divisible by the same power of 2, or if $i + j \leq 2$.*

The proof is similar and it is omitted here. The key point of the nontrivial case, i.e. if both a and b are odd, is the fact that then the target graph B is bipartite, and if an input graph G partially covers B , it must be bipartite as well. Moreover, vertices of degree greater than two from the same class of bipartition must map onto the same of the two vertices of degree greater than two in B .

5 Proofs – NP-Completeness Reductions

5.1 Two Parameters

The following proposition concludes the proof of Theorem 2.

Proposition 2. *The $B(a^i, b^j)$ -PARTIALCOVER problem is NP-complete whenever $|a - b|$ is odd, $i, j \geq 1$ and $i + j \geq 3$.*

Proof. Assume a is odd, b is even, and both parameters are relatively prime. We discuss the case $i, j \geq 2$ first. Let G be the $(i + j)$ -regular graph whose black and white coloring is questioned. We replace each edge of G by a path of length $l = ab$. It can be easily seen that the new graph G' partially covers $B = B(a^i, b^j)$ if and only if a proper $BW(i, j)$ -coloring of G exists.

Similarly, we use $l = ab + (a - 1)a$ to reduce the $BW(i, 1)$ problem to the $B(a^i, b)$ -PARTIALCOVER problem when $j = 1$ and $i > 1$. In the case of $i = 1$ and $j > 1$ we use $l = ab + (b - 1)b$ and reduce $BW(j, 1)$ to $B(a, b^j)$ -PARTIALCOVER.

5.2 Three Parameters

In this subsection we consider the $B(a, b, c)$ -PARTIALCOVER problem, for $a \neq b \neq c \neq a$. We may also assume a, b, c do not have a nontrivial common divisor. Our argument is based on the following approach:

Definition 1. *Let $J = \{j_1, \dots, j_k\}$ be a set of distinct positive integers. We say that a number m has a path covering pattern with respect to J of type (a, b) and length l if there exist integers $x_i, 1 \leq i \leq l$ satisfying*

$$\begin{aligned} m &= x_1 + \dots + x_l \\ x_i &\in J, \quad 1 \leq i \leq l; \quad x_1 = a, \quad x_l = b, \\ x_{p-1} &\neq x_p \neq x_{p+1} \text{ whenever } x_{p-1} \text{ or } x_{p+1} \text{ are defined.} \end{aligned}$$

Note that whenever m has a solution of type (a, b) , then it can be transformed into a solution of type (b, a) of the same length. Hence, the type of a solution will be always expressed by an unordered pair.

Lemma 3. *The $B(a, b, c)$ -PARTIALCOVER problem is NP-complete if there exists an m , which has a path covering pattern of type (c, c) of an odd length, and a pattern of type (a, b) of an even length, and no other covering patterns exist with respect to $J = \{a, b, c\}$.*

Proof. We show a reduction from the $BW(2, 1)$ -coloring problem. Let G be a cubic graph, whose black and white coloring is questioned. We replace each edge of G by a path of length m , and show that the new graph $G' = G^{\cdot m}$ allows a partial covering to $B = B(a, b, c)$ if and only if G has a proper $BW(2, 1)$ -coloring.

Denote by v, w the two vertices of degree three in the graph B , and assume that a partial covering projection $f : G' \rightarrow B$ exists. Then every vertex of degree three in G' is mapped either on v or w . Color each vertex $u \in V(G)$ black, if $f(u) = v$, and color it white otherwise. The mapping f is locally injective on neighborhood of any u in G' , hence, one of the incident edges (u, x) is mapped into a c -path. The maximal subpath of length m that starts with the exposed edge can be covered only by the pattern of type (c, c) . The odd length of the path covering pattern implies that the opposite end x of the maximal subpath

will be mapped onto the other vertex of degree three in B , causing that x gets a different color from the color of u .

By the same argument we can show that the even length of the path covering pattern of type (a, b) implies that every vertex of G has two neighbors colored by the same color.

In the opposite direction, assume a $BW(2, 1)$ -coloring of the graph G . A partial covering projection can be found by the technique already described in the proof of NP -completeness of the $B(a, b, b)$ -PARTIALCOVER problem.

Corollary 1. *The $B(a, b, c)$ -PARTIALCOVER problem is NP -complete whenever $a + b$ divides c .*

Proof. We apply Lemma 3 for m equal to c . The only covering patterns are $m = c$ of type (c, c) and $m = a + b + a + b + \dots + a + b$ of type (a, b) .

The above approach yields the complete characterization of the computational complexity of the $B(1, 2, c)$ -PARTIALCOVER problem for $c > 2$, and half of the cases for $B(1, 3, c)$ -PARTIALCOVER, $c > 3$.

Corollary 2. *The $B(1, 2, c)$ -PARTIALCOVER problem is NP -complete for all $c \geq 3$.*

Proof. If $c = 3k$, then the result follows directly from Corollary 1. When $c = 3k + 1$, then setting $m = c + 1$, we get the following covering patterns $m = 2 + 1 + 2 + 1 + 2 \dots + 2$ and $m = c + 1$. Similarly, for $c = 3k + 2$ we set $m = c + 2$ to get patterns $m = 1 + 2 + \dots + 1 = 1 + c + 1$ (odd length) and $m = c + 2$.

Corollary 3. *The $B(1, 3, c)$ -PARTIALCOVER problem is NP -complete for all even $c \geq 3$.*

Proof. If $c = 4k$, then the result follows directly from Corollary 1. When $c = 4k + 2$, then setting $m = c + 1$, we get the following covering patterns $m = 3 + 1 + 3 + 1 + 3 \dots + 3$ and $m = c + 1$.

Now we show the case when all three parameters a, b, c are odd:

Lemma 4. *The $B(a, b, c)$ -PARTIALCOVER problem is NP -complete if there exists an m which has path covering patterns of types (a, a) , (b, b) and (c, c) , all of odd lengths, and no other covering patterns exist with respect to $J = \{a, b, c\}$.*

Proof. Assume $a < b < c$. We show a reduction from the 3-edge-precoloring extension problem for bipartite graphs. Let G be a cubic bipartite graph with some edges properly precolored by colors black, white and blue. We replace each unprecolored edge of G by a path of length m , each black edge by a path of length a , each white edge by a path of length b and each blue edge (u, v) by a copy of the gadget depicted in Fig. 4. We claim that the new graph G' allows a partial covering to $B(a, b, c)$ if and only if G allows a proper edge-coloring extending the precoloring.

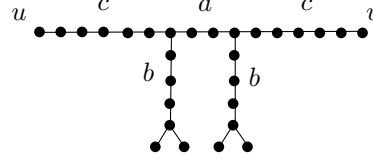


Fig. 4. Gadget for blue edges (the numbers a, b, c refer to the lengths of paths between vertices of degree greater than 2)

Denote by v, w the two vertices of degree three in the graph $B(a, b, c)$, and assume that a partial covering projection $f : G' \rightarrow B(a, b, c)$ exists. Then every vertex of degree three in G' is mapped either on v or w . Because of the parity assumption, vertices of one bipartition class of G are mapped onto v and those from the other class onto w . (This is the key point why we needed G to be bipartite.) The assumption of the existence of covering patterns guarantees a one-to-one correspondence between color patterns used on these paths and colors which we assign to the originally unprecolored edges of G ((a, a) –black, (b, b) –white and (c, c) –blue). To complete the proof, we need to check that the gadgets used to replace precolored edges force a coloring in accord with this correspondence. Indeed, a path of length a only allows pattern (a, a) , a path of length b only the pattern (b, b) and the gadget replacing blue edges only the pattern illustrated in the figure. The opposite implication is straightforward.

Now we are ready to complete the proof of Theorem 3

Corollary 4. *The $B(1, 3, c)$ -PARTIALCOVER problem is NP-complete for every odd $c > 3$.*

Proof. We use Lemma 4. Since c is odd, we have either $c = 4k + 1$ or $c = 4k + 3$. In the former case we set $m = 2c + 1$, allowing only the path covering patterns $2c + 1 = c + 1 + c = 1 + c + 1 + 3 + 1 + \dots + 1 = 3 + c + 3 + 1 + 3 + \dots + 3 = 3 + 1 + 3 + \dots + 3$, while in the latter case we set $m = 2c + 3$, allowing the only path covering patterns $2c + 3 = 1 + c + 1 + c + 1 = c + 3 + c = 1 + c + 1 + 3 + 1 + \dots + 1 = 3 + c + 3 + 1 + 3 + \dots + 3 = 1 + 3 + 1 + \dots + 1$. It is a matter of routine calculation to check that no other patterns are possible.

Acknowledgment

We thank Jan Arne Telle for his support and encouragement in the project.

References

1. ABELLO, J., FELLOWS, M. R., AND STILLWELL, J. C. On the complexity and combinatorics of covering finite complexes. *Australian Journal of Combinatorics* 4 (1991), 103–112.

2. ANGLUIN, D. Local and global properties in networks of processors. *Proceedings of the 12th ACM Symposium on Theory of Computing* (1980), 82–93.
3. ANGLUIN, D., AND GARDINER, A. Finite common coverings of pairs of regular graphs. *Journal of Combinatorial Theory B* 30 (1981), 184–187.
4. BIGGS, N. *Algebraic Graph Theory*. Cambridge University Press, 1974.
5. BODLAENDER, H. L. The classification of coverings of processor networks. *Journal of Parallel Distributed Computing* 6 (1989), 166–182.
6. CHANG, G. J., AND KUO, D. The $L(2,1)$ -labeling problem on graphs. *SIAM Journal of Discrete Mathematics* 9, 2 (May 1996), 309–316.
7. COURCELLE, B., AND MÉTIVIER, Y. Coverings and minors: Applications to local computations in graphs. *European Journal of Combinatorics* 15 (1994), 127–138.
8. FIALA, J. The NP completeness of the edge precoloring extension problem on bipartite graphs. submitted.
9. FIALA, J. Locally injective homomorphisms. PhD thesis, Charles University, Prague, 2000.
10. FIALA, J., AND KRATOCHVÍL, J. Partial covers of graphs. submitted.
11. FIALA, J., KRATOCHVÍL, J., AND KLOKS, T. Fixed-parameter tractability of λ -colorings. In *Graph-Theoretical Concepts in Computer Science, 25th WG '99, Ascona* (1999), no. 1665 in LNCS, Springer Verlag, pp. 350–363.
12. FOTAKIS, D. A., NIKOLETSEAS, S. E., PAPADOPOULOU, V. G., AND SPIRAKIS, P. G. NP -completeness results and efficient approximations for radiocoloring in planar graphs. In *MFCS* (2000), no. 1893 in LNCS, Springer Verlag, pp. 363–372.
13. GEORGES, J. P., MAURO, D. W., AND WHITTLESEY, M. A. Relating path coverings to vertex labellings with a condition at distance two. *Discrete Mathematics* 135 (1994), 103–111.
14. GRIGGS, J. R., AND YEH, R. K. Labelling graphs with a condition at distance 2. *SIAM Journal of Discrete Mathematics* 5, 4 (Nov 1992), 586–595.
15. HOLYER, I. The NP -completeness of edge coloring. *SIAM Journal of Computing* 10, 4 (1981), 718–720.
16. KRATOCHVÍL, J. *Perfect codes in general graphs*. Academia Praha, 1991.
17. KRATOCHVÍL, J., PROSKUROWSKI, A., AND TELLE, J. A. Covering directed multigraphs I. colored directed multigraphs. In *Graph-Theoretical Concepts in Comp. Sci., 23rd WG '97, Berlin* (1997), no. 1335 in LNCS, Springer Verlag, pp. 242–257.
18. KRATOCHVÍL, J., PROSKUROWSKI, A., AND TELLE, J. A. Covering regular graphs. *Journal of Combinatorial Theory B* 71, 1 (Sept 1997), 1–16.
19. KRATOCHVÍL, J., PROSKUROWSKI, A., AND TELLE, J. A. Complexity of graph covering problems. *Nordic Journal of Computing* 5 (1998), 173–195.
20. KRATOCHVÍL, J., AND SEBÖ, A. Coloring precolored perfect graphs. *Journal of Graph Theory* 25, 3 (1997), 207–215.
21. KRISTIANSEN, P., AND TELLE, J. A. Generalized H -coloring of graphs. In *International Symposium on Algorithms And Computation, 7th ISAAC '00, Taipei* (2000), no. 1969 in LNCS, pp. 456–466. Springer Verlag,
22. LEIGHTON, F. T. Finite common coverings of graphs. *Journal of Combinatorial Theory B* 33 (1982), 231–238.
23. LOVÁSZ, L., AND PLUMMER, M. D. *Matching Theory*. Akadémiai Kiadó, Budapest, 1986.
24. SAKAI, D. Labeling chordal graphs: distance two condition. *SIAM Journal of Discrete Mathematics* 7, 1 (February 1994), 133–140.
25. TELLE, J. A. Complexity of domination-type problems in graphs *Nordic Journal of Computing* 1 (1994), 157–171.

26. VAN DEN HEUVEL, J., LEESE, R. A., AND SHEPHERD, M. A. Graph labeling and radio channel assignment. *Journal of Graph Theory* 29, 4 (1998), 263–283.
27. YEH, K.-C. *Labeling graphs with a condition at distance two*. PhD thesis, University of South Carolina, 1990.

On Game-Theoretic Models of Networks

Hans L. Bodlaender¹, Michael J. Dinneen², and Bakhadyr Khoussainov²

¹ Institute of Information and Computing Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands
`hansb@cs.uu.nl`

² Department of Computer Science, The University of Auckland
Auckland, New Zealand
`{mjd,bmk}@cs.auckland.ac.nz`

Abstract. In this paper, we study the complexity of deciding which player has a winning strategy in certain types of McNaughton games. These graph games can be used as models for computational problems and processes of infinite duration. We consider the cases (1) where the first player wins when vertices in a specified set are visited infinitely often and vertices in another specified set are visited finitely often, (2) where the first player wins when exactly those vertices in one of a number of specified disjoint sets are visited infinitely often, and (3) a generalization of these first two cases. We give polynomial time algorithms to determine which player has a winning strategy in each of the games considered.

Keywords: graph and network algorithms, complexity, infinite graph games, McNaughton games.

1 Introduction and Basic Definitions

Motivated by the work of Gurevich and Harrington [3], McNaughton [5] introduced a type of infinite game played on finite graphs. These games can be used as models for certain computational problems and can provide game-theoretic foundations for studying infinite-duration processes such as operating systems, networks, communication systems and concurrent computations. For example, Nerode *et al.* [7,6] introduce the idea of investigating and identifying distributed concurrent programs as strategies in Gurevich-Harrington and McNaughton type of games. We also mention a related paper [4] that uses a modal logic version of these games as a model for problems in control theory.

Assume we have an infinite-duration system. A run of the system can be thought of as an infinite sequence $s_0, s_1, s_2, s_3, \dots$ of states. The state s_0 is the initial state. The state s_{i+1} is obtained by the execution of a certain command at s_i . The success of the run depends on whether or not the run satisfies certain specifications given by (or inherited from) software or hardware of the system. One can look at this run as a play between two players, *Survivor* and *Adversary*. The goal of one of the players, say *Survivor*, is to satisfy the specifications, while the goal of the opponent (in this case *Adversary*) is not to allow the specifications

to be satisfied. During the play there is no termination point. Instead there are some special events that may happen continually. If some combination of these events happens infinitely often then one player wins, otherwise the other player wins. We now formalize these games, as was first done in [5].

Definition 1. A game $\mathcal{G}$ is a seven tuple $(V, S, A, E, v_0, W, \Omega)$, where:

1. V is the set of nodes called **positions**.
2. S and A are subsets of V such that $S \cap A = \emptyset$ and $S \cup A = V$. The nodes of S are **positions** of Survivor, and the nodes of A are **positions** of Adversary.
3. $E \subseteq S \times A \cup A \times S$ is a set of directed edges between S and A such that
 - (a) for each $s \in S$ there exists at least one $a \in A$ with $(s, a) \in E$, and
 - (b) for each $a \in A$ there exists at least one $s \in S$ with $(a, s) \in E$.
4. v_0 is the initial position of the game.
5. W is a subset of V called the set of **special positions**.
6. Finally, Ω is a set of some subsets of W . These are called **winning sets** or **winning conditions** for Survivor.

For the game $\mathcal{G}$ the **graph of the game** is the graph $(A \cup S, E)$. All plays of $\mathcal{G}$ occur in the graph of the game. To visualize a play we describe it informally as follows. There is a placemaker, that is initially placed on node v_0 . At any given time the placemaker is placed on a node. If the node is in S , then it is *Survivor's* turn to move the placemaker. Otherwise it is *Adversary's* turn. The placemaker is always moved along the edges of the game graph determined by E . There is always a possibility to move the placemaker as stipulated by conditions 3a) and 3b) of the definition.

Let s_0 be a position, say of *Survivor*. Assume that *Survivor* begins its move by putting the placemaker on a_0 (so $(s_0, a_0) \in E$). *Adversary* responds by putting the placemaker on a s_1 (so $(a_0, s_1) \in E$). This procedure repeats and the players' actions produce an infinite sequence:

$$\mathbf{p} = s_0, a_0, s_1, a_1, \dots$$

called a **play** that begins from position s_0 . In the play $\mathbf{p}$ consider the set of all nodes t that have the following properties:

1. t belongs to W , and
2. t occurs in the play $\mathbf{p}$ infinitely often.

We denote this set by $In(\mathbf{p})$ and call it the **infinity set of $\mathbf{p}$** . *Survivor* wins the play if $In(\mathbf{p}) \in \Omega$. Otherwise *Adversary* wins the play. Thus, every play is won by one of the players.

The **histories** of the play $\mathbf{p} = q_0, q_1, q_2, \dots$ are the finite prefixes of $\mathbf{p}$. The set $H(S)$ consists of all histories whose last positions are positions where *Survivor* makes move. The set $H(A)$ is defined similarly. A **strategy** for *Survivor* is a function f that maps $H(S)$ into A such that for all $u = q_0 \dots q_n \in H(S)$, $(q_n, f(u)) \in E$. A **strategy** for *Adversary* is defined similarly.

Let f be a strategy for a player. Let q be a position in the game. Consider all the plays that begin from q which are played when the player follows the strategy f . We call these **plays consistent with f from q** .

Definition 2. *The strategy f of a player is a **winning strategy** if all plays consistent with f from v_0 are won by the player. In this case we say that the player **wins the game**.*

McNaughton [5] proved that for every McNaughton game, it is decidable who has a winning strategy. However, his algorithm is by no means an efficient one. Thus, it is natural to ask for which type of McNaughton games it can be decided in polynomial time which player has a winning strategy. Some polynomial time solvable instances were given by Dinneen and Khoussainov in [2] and Nerode et al. in [6]. In [2] games with $W = V$ and $\Omega = \{V\}$, called **update network games** are studied and it is shown that there is an $O(|V||E|)$ time algorithm to determine if *Survivor* wins these games. In this paper, we extend this result. First, we consider for networks with a partition of the set of nodes into three sets $V = I \cup F \cup D$, games of the form $(V, S, A, E, v_0, I \cup F, \{I\})$. I.e., *Survivor* wins if every node in I is visited infinitely often, and every node in F is visited finitely often. Thus, each play in such games is indifferent whether or not the nodes in $D = V \setminus W$ are visited finitely or infinitely often. Therefore we call the nodes D **don't care nodes**. We provide a $O(|V||E|)$ time algorithm to decide which player has a winning strategy in such games. Secondly, we consider the games where $W = V$ and Ω is a collection of pairwise disjoint winning sets. We show that there exists a polynomial time algorithm to decide who wins such games. Finally, we combine these results, and allow W to be a proper subset of V , with Ω a collection of pairwise disjoint non-empty winning sets.

2 Preliminary Results

Given a McNaughton game $\mathcal{G}$ and a subset of the nodes $X \subseteq V$, a node v is in the set $\text{REACH}(S, X)$ if *Survivor* can force every play starting at v into a node in X after a finite number of steps. Note that $\text{REACH}(S, \emptyset)$ is assumed to be $\emptyset$ which is consistent with the definition.

Lemma 1. *The set $\text{REACH}(S, X)$ can be computed in $O(|V| + |E|)$ time.*

Proof. We build a set R , that will eventually be $\text{REACH}(S, V)$. Initially, we take $R = X$. If a node x , owned by *Survivor*, has an edge to a node in R , then x is added to R . If a node x , owned by player *Adversary*, has only edges to nodes in R , then x is added to R . One can note that from every node in R *Survivor* can always force a play to go to a node in X . Moreover, when no nodes can be added to R anymore, then $R = \text{REACH}(S, X)$. *Adversary* has a strategy such that only nodes in $V \setminus R$ are visited. Indeed, *Adversary* has a strategy to always stay inside of $V \setminus R$ when game begins in a node from $V \setminus R$. The procedure of constructing $\text{REACH}(S, X)$ can be implemented in $O(|V| + |E|)$ time, by giving each node not in X a counter, that is initially 1 for nodes owned by *Survivor* and its outdegree for nodes owned by *Adversary*. Whenever we add a node v to R , we subtract 1 from the counters of each node with an edge to v ; when a counter becomes 0 then the node is also added to R . $\square$

Let $v \notin \text{REACH}(S, X)$ be an *Adversary's* node. We iteratively define the set $\text{AVOID}(v, A, X)$ as follows. Initially, we take $\text{AVOID}(v, A, X) = \{v\}$. If a node x is owned by *Adversary* and $x \in \text{AVOID}(v, A, X)$ then we add a neighbor y into $\text{AVOID}(v, A, X)$ if $(x, y) \in E$ and $y \notin \text{REACH}(S, X)$. If a node x is owned by *Survivor* and $x \in \text{AVOID}(v, A, X)$ then we add all y into $\text{AVOID}(v, A, X)$ for which $(x, y) \in E$.

From Lemma 1 we obtain the following lemma.

Lemma 2. *Given X and $v \notin \text{REACH}(S, X)$ the set $\text{AVOID}(v, A, X)$ has the following properties:*

1. *The set $\text{AVOID}(v, A, X)$ can be constructed in $O(|V| + |E|)$ time.*
2. *$\text{AVOID}(v, A, X) \cap \text{REACH}(S, X) = \emptyset$.*
3. *Adversary has a strategy, such that when the game visits a node in $\text{AVOID}(v, A, X)$ then all nodes visited afterwards also belong to $\text{AVOID}(v, A, X)$.*
4. *For all s in $\text{AVOID}(v, A, X) \cap S$ and all $a \in A$ if $(s, a) \in E$ then a is in $\text{AVOID}(v, A, X)$.*

If the game starts at v , then a strategy for *Adversary* not to play to a node in X is to always play to a node in $\text{AVOID}(v, A, X)$. Note that the sets $\text{REACH}(A, X)$ and $\text{AVOID}(v, S, X)$ can be defined in a similar matter. The two lemmas above hold true for these sets too.

3 Relaxed Update Networks

In [2] the games where $W = V$ and $\Omega = \{V\}$ are studied. These games are called **update network games**. An update network game is an **update network** if *Survivor* wins the game. We generalize these games in the following definition.

Definition 3. *A game $\mathcal{G}$ is relaxed update network game if Ω consists of a fixed subset I of W . We say that a relaxed update network game from a position q is a **relaxed update network** if *Survivor* has a winning strategy from q .*

Thus, in a relaxed update network the set of nodes is partitioned into three sets $V = I \cup F \cup D$, where I is a given subset of W , $F = W \setminus I$, and $D = V \setminus W$. *Survivor* wins a play if every node in I is visited infinitely often, and every node in F is visited finitely often. Thus, each play in such games is indifferent whether or not the nodes in D are visited finitely or infinitely often. Therefore we can call the nodes in D **don't care nodes**.

3.1 The Case $I = \emptyset$

Let $\mathcal{G}$ be a relaxed update game. Here we consider the case that $I = \emptyset$, i.e., we have nodes that must be visited only finitely often (F) and don't care nodes. Of course, the problem is trivial when $F = \emptyset$ and $I = \emptyset$. So we assume that $F \neq \emptyset$.

Let $V_0 = V \setminus \text{REACH}(A, F)$. If V_0 is empty, then *Adversary* has a winning strategy: from every node, *Adversary* has a forced play into a node in F . Thus, *Adversary* can force some of the nodes in F to be visited infinitely often.

If *Survivor* begins the game from a node v in V_0 , then *Survivor* has a winning strategy: he plays always inside the set $\text{AVOID}(v, S, F)$ which is possible by Lemma 2.

If neither V_0 is empty nor the game starts at a node in V_0 , then we start with an iterative process. In order to describe the process we make the following notes.

Consider $\text{REACH}(S, V_0)$. Note that for each node in $\text{REACH}(S, V_0)$, *Survivor* has a winning strategy when the game starts at that node. *Survivor* can force all plays from the node into V_0 . When a node in V_0 is reached *Survivor* has a strategy such that no node in F is visited anymore. Thus, *Adversary* should not play into a node in $\text{REACH}(S, V_0)$. In particular, *Adversary* should not play to nodes in $F \cap \text{REACH}(S, V_0)$.

Let us consider $F_1 = F \setminus \text{REACH}(S, V_0)$ and $V_1 = V \setminus \text{REACH}(A, F_1)$. Note that $V_0 \subseteq V_1$. *Survivor* has a winning strategy when the game starts at a node of V_1 . *Survivor* can always play inside V_1 again by Lemma 2, and hence no nodes in F_1 are visited. So the only nodes in F *Adversary* can possibly direct the plays to are those in $\text{REACH}(S, V_0)$. But from these nodes *Survivor* can force all plays into V_0 . Hence nodes in F are visited in total a finite number of times.

Thus, when the game starts at a node in V_1 we are done: *Survivor* has a winning strategy. When $V_1 = V_0$ and the game starts at a node in $V \setminus V_1$, then we are also done as *Adversary* has a winning strategy; *Adversary* always forces all the plays into $F_1 = F$ staying in $V \setminus V_1$.

The step above can be repeated which leads us to an iterative procedure. Thus, let $F_0 = F$ and $V_0 = V \setminus \text{REACH}(A, F_0)$. For each $i \geq 1$, let

$$F_i = F_{i-1} \setminus \text{REACH}(S, V_{i-1}) \quad \text{and} \quad V_i = V \setminus \text{REACH}(A, F_i).$$

With arguments similar as above, we can show that *Survivor* has a winning strategy in all nodes in V_i .

As each $V_i \subseteq V_{i+1}$, the process stops when we have an i with $V_i = V_{i+1}$. In that case, there is a winning strategy for *Survivor* if and only if the game starts at a node in V_i . Suppose the game starts at a node in $V \setminus V_i$ and $V_i = V_{i+1}$. Then, *Adversary* can force a play to a vertex in F_{i+1} ; and either it is owned by *Survivor* and has all outgoing edges to a vertex in $V \setminus V_i$ or is owned by *Adversary* and has one outgoing edge to $V \setminus V_i$, (as follows from Lemma 2), hence *Adversary* can force the game to stay in $V \setminus V_i$.

This gives a polynomial time algorithm for the problem with $I = \emptyset$. The algorithm takes $O(|V||E|)$ time, as there are $O(|V|)$ iterations, each taking $O(|V| + |E|)$ time.

3.2 Reducing to the Case $F = \emptyset$

Now assume $I \neq \emptyset$. In this section, we show that an instance with $F \neq \emptyset$ can be transformed to an equivalent instance with $F = \emptyset$, assuming $I \neq \emptyset$.

We may assume that the initial position belongs to $\text{REACH}(S, I)$; if not, then clearly *Adversary* has a winning strategy from Lemma 2. (*Adversary* forces that no node in I is ever reached.) Now, *Survivor* can start the game by forcing to go to any node in I , and, as all nodes in I have to be visited infinitely often, it is not important for the analysis to which node in I the game goes first.

There are two cases:

$\text{REACH}(A, F) \cap I \neq \emptyset$. This means that there is a node $i \in I$, such that *Adversary* has a strategy that forces all the plays of the game (consistent with the strategy) to visit a node in F after a finite number of steps. If this is the case, then *Adversary* has a winning strategy for the game. Here either i is not visited infinitely often, or he can force after every visit to i a play to a node in F , in which case at least one node in F is visited infinitely often.

$\text{REACH}(A, F) \cap I = \emptyset$. This means that for all nodes $i \in I$, *Adversary* can not force any play of the game visit a node in F . Therefore if *Survivor* has a winning strategy then he has one that prevents movement to a node in F after the first node in I has been reached.

Once a node in I has been reached, *Survivor* wants to avoid the plays reaching nodes in F . (Any play to a node in F now could possibly be repeated by *Adversary*.) So if *Survivor* can avoid reaching a node in F infinitely many times, he can avoid visiting it once.

So, what we can do is compute $\text{REACH}(A, F)$, and remove all nodes in $\text{REACH}(A, F)$ from the graph, and obtain an equivalent instance, but now with $F = \emptyset$.

3.3 Case with Infinite-Visit Nodes

In this section, we consider the game with $F = \emptyset$ and $I \neq \emptyset$. Suppose the game starts at node v_0 .

Lemma 3. *There is a winning strategy for Survivor if and only if the node $v_0 \in \text{REACH}(S, I)$ and $I \subseteq \text{REACH}(S, \{v\})$ for all $v \in I$.*

Proof. Suppose $w \in I$, $w \notin \text{REACH}(S, \{v\})$. Then *Adversary* has a winning strategy. If w is never visited in the game, then *Survivor* loses. If w is visited, then after w has been visited, *Adversary* has a strategy that avoids v , so *Adversary* again wins.

If $v_0 \notin \text{REACH}(S, I)$, then *Adversary* can prevent any node in I to be visited as follows from Lemma 2.

Now suppose for all $v \in I$, $I \subseteq \text{REACH}(S, \{v\})$, and $v_0 \in \text{REACH}(S, I)$. The latter condition makes that *Survivor* can start by forcing all plays from v_0 into I . The former condition means that for every pair of nodes $v, w \in I$, *Survivor* has a strategy that forces, after w has been visited, that in a finite number of moves v will be visited. This enables *Survivor* to force that every vertex in I to be visited infinitely often. $\square$

The condition of Lemma 3 can be checked in $O(|V||E|)$ time. Thus we have proved the following theorem.

Theorem 1. *There is a $O(|V||E|)$ time algorithm to decide whether a given game is a relaxed update network.*

3.4 P-Completeness

The previous section shows that we can decide in polynomial time if a relaxed update game is a relaxed update network. Let's call this the RELAXEDNETWORK problem. Here we show that this problem is P-complete and hence any decision algorithm for it is inherently sequential.

Proposition 1. *The RELAXEDNETWORK problem is P-complete.*

Proof. All that remains to see is that RELAXEDNETWORK is log-space hard for the complexity class P. To do this, we reduce the AGAP (And/Or Graph Accessibility Problem), which is known to be P-complete [1], to RELAXEDNETWORK. An instance of AGAP is an and/or graph $D = (V, A)$ with two vertices s and t . The problem is to decide whether t is reachable from s . We say that t is reachable from s in an and/or graph $D = (V, A)$ if a pebble can be placed on the specified vertex t by using the following rules:

1. We can place a pebble on s .
2. For an AND vertex v , a pebble can be placed if all in-neighbors of v are pebbled.
3. For an OR vertex v , a pebble can be placed if at least one in-neighbor is pebbled.

We can transform this instance into an instance of RELAXEDNETWORK as follows. We map an instance (D, s, t) of AGAP into a game instance $(V, S, A, E, v_0, I \cup F, \{I\})$. First let D' be a bipartite version of D where we subdivide any arcs with two end-points of the same type (the new vertex is the opposite type). Then declare $V = V(D') \cup \{t'\}$, S equal to the OR vertices, A equal to the AND vertices, $E = E(D') \cup \{(t, t'), (t', t)\} \setminus \{(t, v) \mid v \in V(D')\}$, $v_0 = s$, $I = \{t, t'\}$, $F = V \setminus I$. There is a pebbled path from s to t in D if and only if Survivor wins the game defined. This transformation is clearly doable in log space. $\square$

3.5 A Dual Case

This case is obtained when we interchange the players of games. Let us consider the case when $I = \emptyset$ in a relaxed update game and interchange the roles of the players. Thus, now *Survivor's* winning conditions are nonempty subsets of F . Then Subsection 3.1 can be explained as follows. Consider the following sequence:

$$F_0 = \text{REACH}(S, F), \quad F_{i+1} = \{x \mid x \in \text{REACH}(S, F_i \setminus \{x\}) \text{ and } x \in F_i\}.$$

The iteration guarantees that F_i consists of all nodes from which *Survivor* can visit the set F at least $i + 1$ times. Note that $F_{i+1} \subseteq F_i$ for all i . Let i be such that $F_i = F_{i+1}$. We can show that *Survivor* wins the game from v if and only if $v \in \text{REACH}(S, F_i)$. The proof is basically given in Subsection 3.1.

4 Partition Games and Partition Networks

In this section we study games where winning conditions are pairwise disjoint collections of nonempty sets with $W = V$. Formally, a **partition network game** is a game $\mathcal{G}$ of the form $(V, S, A, E, v_0, V, \{W_1, \dots, W_n\})$, where $W_1, \dots, W_n$ is a collection of pairwise disjoint nonempty winning sets. We say that a partition network game is a **partition network** if *Survivor* is the winner of the game. An important concept of closed winning conditions (sets) is defined as follows:

Definition 4. A winning condition W_i in a game $\mathcal{G}$ is **S-closed** if the following two conditions are satisfied:

1. For any *Survivor*'s position $s \in W_i$ there exists an a such that $(s, a) \in E$ and $a \in W_i$.
2. For any *Adversary*'s position $a \in W_i$ and all s such that $(a, s) \in E$ we have $s \in W_i$.

Informally, if W_i is a closed winning set then *Survivor* can always stay inside of W_i no matter what the opponent does. The next lemma gives a necessary condition for *Survivor* to win a partition network game.

Lemma 4. If *Survivor* wins the partition network game $\mathcal{G}$ then one of the winning conditions must be S-closed.

Proof. Suppose that each W_i is not S-closed. Then for each W_i one of the following cases hold:

1. There exists a *Survivor*'s node $s_i \in W_i$ so that all the outgoing edges from s lead to nodes outside of W_i .
2. There exists an *Adversary*'s node $a_i \in W_i$ such that $(a_i, s_i) \in E$ and $s_i \notin W_i$.

We construct the following strategy g for *Adversary*. For all *Adversary*'s positions a if $a = a_i$ then $g(a) = s_i$; in all other cases $g(a)$ is the first node s for which $(a, s) \in E$. We claim that g is a winning strategy for *Adversary* thus contradicting the assumption. Indeed let $\mathbf{p} = p_0, p_1, \dots$ be a play consistent with g . Consider the infinity set $In(\mathbf{p})$. Assume that $In(\mathbf{p}) = W_i$. Then from some stage m in the play all nodes from W_i and only those will appear infinitely often. Therefore W_i does not satisfy the first case listed above. Hence for W_i there exists an *Adversary*'s node $a_i \in W_i$ such that $(a_i, s_i) \in E$ and $s_i \notin W_i$. From the definition of g , as a_i must appear in $\mathbf{p}$ after point m , we see that $\mathbf{p}$ must contain a position from outside of W_i after stage m . This contradicts the choice of m . Therefore $In(\mathbf{p}) \neq W_i$ for all winning sets W_i . $\square$

For our next lemma we need the following concept. We say that a winning condition W is an **update component** if W is S-closed and *Survivor* wins the update game played in W .

Lemma 5. If *Survivor* wins the partition network game $\mathcal{G} (V, S, A, E, v_0, V, \{W_1, \dots, W_n\})$, then one of the winning conditions is an update component.

Proof. By the lemma above, one of the winning conditions W_i must be S -closed. Without loss of generality we may assume that $W_1, \dots, W_k$ are all the S -closed winning conditions among $W_1, \dots, W_n$, where $k \leq n$.

In order to obtain a contradiction, assume that none of $W_1, \dots, W_k$ is an update component. Hence for every t with $1 \leq t \leq k$ and every $x \in W_t$, *Adversary* has a winning strategy $g_{t,x}$ to win the update game (W_t, x) from x . Note that for each W_i , $i > k$, one of the following cases hold:

1. There exists a *Survivor*'s node $s_i \in W_i$ so that all the outgoing edges from s lead to nodes outside of W_i .
2. There exists an *Adversary*'s node $a_i \in W_i$ such that $(a_i, s_i) \in E$ and $s_i \notin W_i$.

Now we define the following strategy g for *Adversary*. Let a be an *Adversary*'s position. Consider any finite history $h = p_0, \dots, p_m$ of a play that begins from v so that $a = p_m$. If $a = a_i$ for some $i > k$ then $g(h) = s_i$. Now assume $a \in W_t$ with $1 \leq t \leq k$. Let p_r be a node in the history so that all $p_r, \dots, p_m \in W_t$ and $p_{r-1} \notin W_t$. Then $g(h) = g_{t,p_r}(p_r, \dots, p_m)$. In all other cases, $g(h)$ is the first s with $(a, s) \in E$.

We claim that g is a winning strategy for *Adversary*. Indeed, let $\mathbf{p} = p_0, p_1, p_2, \dots$ be a play consistent with g . Consider the infinity set $In(\mathbf{p})$. Assume that $In(\mathbf{p}) = W_i$. Then $i \leq k$ which can be proved by using the reasoning similar to the proof of the previous lemma. Assume that $i \leq k$. Let m be the first point in the play $\mathbf{p}$ so that all nodes from W_i and only those will appear infinitely often. Then g will always follow the strategy g_{i,p_m} . Hence $In(\mathbf{p})$ can not be equal to W_i . Again we have a contradiction. $\square$

From these two lemmas we have the following result.

Corollary 1. *In a partition network game, if either (1) each winning conditions is not S -closed or (2) each S -closed winning condition does not form an update component then *Adversary* wins the partition game.*

Now assume that one of the winning conditions of the partition network game is an update component. Without loss of generality we can assume that it is W_1 . Consider the set $REACH(S, W_1)$. If $v_0 \in REACH(S, W_1)$ then *Survivor* clearly wins the game. Otherwise, we define the following game $\mathcal{G}'$:

1. Set $V' = \text{AVOID}(v_0, A, W_1)$.
2. For each W_i if $W_i \cap REACH(S, W_1) \neq \emptyset$ then W_i is not a winning set of the new game. Otherwise, W_i is a winning set of the new game.
3. The set E' of edges is obtained by restricting E to V' .
4. The initial position of the game is v_0 .

Lemma 6. *Assume W_1 is an update network component and $v_0 \notin REACH(S, W_1)$. *Survivor* wins the original game if and only if *Survivor* wins the new game $\mathcal{G}'$.*

Proof. Indeed, assume that *Adversary* wins the new game $\mathcal{G}'$. Let g' be winning strategy. Then since g' is inside the $\text{AVOID}(v_0, A, W_1)$ strategy, we see that *Adversary* wins the whole game. Assume that *Survivor* wins the new game. Let f' be winning strategy. Define a strategy f as follows. If a play is inside the game $\mathcal{G}'$ then always follow f' . Otherwise, force the place into W_1 and win the update game W_1 . It is not hard to see that *Survivor* wins the game. $\square$

We call the game $\mathcal{G}'$ obtained from $\mathcal{G}$ the reduced game at v_0 . Now consider the following procedure that for any $x \in V$ proceeds by stages as follows.

Stage 0. Set $\mathcal{G}_0 = \mathcal{G}$.

Stage $i+1$. Consider $\mathcal{G}_i$. If all of the winning conditions of $\mathcal{G}_i$ are not S -closed or all S -closed winning conditions of $\mathcal{G}_i$ are not update components then declare *Adversary* the winner. Otherwise take the first winning condition W which is an update network component. If $x \in \text{REACH}(S, W)$ then *Survivor* is the winner. If not, reduce $\mathcal{G}_i$ to $\mathcal{G}_{i+1}$ at node x .

Note that at some stage k the process stops at which the winner at x is found. The algorithm to decide the game runs in $O(|E||V|^2)$ time yielding:

Theorem 2. *There is a $O(|V|^2|E|)$ time algorithm to decide whether a given game is a partition network.*

5 Relaxed Partition Networks

In this section, we combine the results of Sections 3 and 4. We consider partition games where possibly $W \neq V$. We now have **relaxed partition network games** of the form

$$\mathcal{G} = (V, S, A, E, v_0, W, \{W_1, \dots, W_n\}),$$

where $W \subseteq V$, and $W_1, \dots, W_n$ is a collection of pairwise disjoint nonempty winning sets, each a subset of W . Again, the set of don't care nodes is denoted by $D = V \setminus W$.

For sets $X, Y \subseteq V$, $X \cap Y = \emptyset$, define the set $RA(S, X, Y)$ of nodes from which *Survivor* can force a play that reaches, in a finite number of steps, a node in X by avoiding Y . Thus, $v \in RA(S, X, Y)$ if *Survivor* has a winning strategy in the game that starts at a v where *Survivor* wins as soon as a node in X is visited; *Adversary* wins as soon as a node in Y is visited or when infinitely many moves occur without a visit to a node in $X \cup Y$.

Lemma 7. *Given X, Y , $X \cap Y = \emptyset$, $RA(X, Y)$ can be computed in $O(|V||E|)$ time.*

Proof. The set $RA(S, X, Y)$ can be computed as follows. Initially, set $R = X$. If a node $s \in S \setminus Y$ has an edge $(s, a) \in E$ and $a \in R$, then add s to R . If a node $a \in A \setminus Y$ has for all s with $(a, s) \in E$, $s \in R$, then add s to R . Repeat this process until we cannot add nodes to R using these rules. One easily sees with induction that $R \subseteq RA(S, X, Y)$. We also have, after no further nodes can

be added to R , that $R = RA(S, X, Y)$; any Adversary node in $V \setminus R \setminus Y$ has an edge to a node in $V \setminus R$, and any Survivor node in $V \setminus R \setminus Y$ has only edges to nodes in $V \setminus R$. Thus, when *Adversary* follows a strategy to always play to nodes in $V \setminus R$, he wins either by having the game moved to a node in Y , or by an infinite play. Finally, use the same data structure as in Lemma 1. $\square$

Definition 5. A winning condition W_i in a game G is S -closed with respect to W , if the following two conditions are satisfied:

1. For any Survivor's position $s \in W_i$, there exists an a such that $(s, a) \in E$ and $a \in RA(S, W_i, W \setminus W_i)$.
2. For any Adversary's position $s \in W_i$ and all a with $(s, a) \in E$, we have $a \in RA(S, W_i, W \setminus W_i)$.

Note that the definition of S -closedness of the previous section is the same as S -closedness with respect to V . Informally, when W_i is an S -closed winning set with respect to W , then *Survivor* can force a play that visits only nodes in W_i and don't care nodes in D . Similar to the Lemma 4, we can show:

Lemma 8. If *Survivor* wins the relaxed partition network game $\mathcal{G}$ then one of the winning conditions must be S -closed with respect to W .

For a set of nodes $X \subseteq V$ with for all $s \in X \cap S$, there is an $a \in X \cap A$ with $(s, a) \in E$ and for all $a \in X \cap A$, there is an $s \in X \cap S$ with $(a, s) \in E$, we can define the subgame, induced by X with initial position $v' \in X$:

$$(X, S \cap X, A \cap X, E \cap (X \times X), v', W \cap X, \Omega \cap \mathcal{P}(X)),$$

where $\Omega \cap \mathcal{P}(X)$ is the collection of sets in Ω that are a subset of X . In other words, the game is similar to the original game, but now only nodes in X are visited.

Lemma 9. If *Survivor* wins the relaxed partition network game $\mathcal{G}$ then for one of the winning conditions W_i , we have that W_i is S -closed with respect to W , the subgame, induced by $RA(S, W_i, W \setminus W_i)$, with initial position an arbitrary $v \in W_i$ has a winning strategy for *Survivor*, and the start node v_0 of $\mathcal{G}$ belongs to $REACH(S, W_i)$.

The proof of this lemma is similar to (but somewhat more detailed as) the proof of Lemma 5. The conditions of these lemmas can again be checked in $O(|V||E|)$ time, as the game, induced by $RA(S, W_i, W \setminus W_i)$ is a relaxed update game.

Suppose the conditions of the preceding lemma is fulfilled for winning condition W_1 . If $v_0 \in REACH(S, W_1)$, *Survivor* wins the game. Otherwise, game $\mathcal{G}'$ can be defined as in the previous section, and we again have that *Survivor* wins the game, if and only if *Survivor* wins game $\mathcal{G}'$. The time to decide which player has a winning strategy is again bounded by $O(|E||V|^2)$. Thus, we finally have the following result.

Theorem 3. There is a $O(|V|^2|E|)$ time algorithm to decide whether a given game is a relaxed partition network.

6 Conclusions

In this paper, we gave some types of McNaughton games where one can decide in polynomial time which player has a winning strategy. The interest in these games is that they can be used as a model for infinite processes.

Several directions for further research remain open. At one hand, one can try to design faster algorithms for the problems solved in this paper. In addition, it would be interesting to see which kind of conditions on the winning sets produce efficient algorithms to solve the games, and what conditions turn this problem computationally intractable. Another problem is to pinpoint the precise complexity (in terms of complexity class) of the question to decide if a given player has a winning strategy for a given McNaughton game.

References

1. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114-133, January 1981. 556
2. M. J. Dinneen and B. Khoussainov. Update networks and their routing strategies. In *Proceedings of the 26th International Workshop on Graph-Theoretic Concepts in Computer Science, WG2000*, volume 1928 of *Lecture Notes on Computer Science*, pages 127-136. Springer-Verlag, June 2000. 552, 553
3. Y. Gurevich and L. Harrington. Trees, Automata, and Games, STOCS, 1982, pages 60-65. 550
4. J. F. Knight and B. Luense. Control Theory, Modal Logic, and Games, In *Hybrid Systems IV*. Panos J. Antsaklis, Wolf Kohn, Anil Nerode, Shankar Sastry (Eds.), volume 1273 of *Lecture Notes in Computer Science*, pages 160-173. Springer, 1997. 550
5. R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65:149-184, 1993. 550, 551, 552
6. A. Nerode, J. Remmel, and A. Yakhnis. McNaughton games and extracting strategies for concurrent programs. *Annals of Pure and Applied Logic*, 78:203-242, 1996. 550, 552
7. A. Nerode, A. Yakhnis, V. Yakhnis. Distributed concurrent programs as strategies in games. Logical methods (Ithaca, NY, 1992), pages 624-653, *Progr. Comput. Sci. Appl. Logic*, 12, Birkhauser Boston, Boston, MA, 1993. 550

The Complexity of Some Basic Problems for Dynamic Process Graphs [★]

Andreas Jakoby¹ and Maciej Liśkiewicz^{1,2}

¹ Institut für Theoretische Informatik, Med. Universität zu Lübeck
{jakoby, liskiewi}@informatik.mu-luebeck.de

² Instytut Informatyki, Uniwersytet Wrocławski

Abstract. A fundamental problem in programming multiprocessors is scheduling elementary tasks on the available hardware efficiently. Traditionally, one represents tasks and precedence constraints by a *data-flow graph*. This representation requires that the set of tasks is known beforehand. Such an approach is not appropriate in situations where the set of tasks is not known exactly in advance, for example, when different options how to continue a program are possible. In this paper *dynamic process graph (DPG)* will be used to represent the set of all possible executions of a given program. An important feature of this model is that graphs are encoded in a very succinct way. The encoded executions are directed acyclic graphs with a "regular" structure that is typical for parallel programs.

With respect to such a graph representation we investigate the computational complexity of some basic graph-theoretic problems like e.g. *what is the minimum depth of a graph represented by a DPG?* or *what is the size of a subgraph induced by a given node v?* In this paper the complexities of these problems are determined precisely. As a consequence approximations of the computational complexity of some variants of scheduling problems are obtained.

1 Introduction

In programming multiprocessors to describe the elementary steps of computation and the logical dependencies among them one uses traditionally *data-flow graphs* that allow to extract parallelism automatically. Nodes of a data-flow graph represent tasks to be executed and edges indicate the precedence constraints. Such graphs are also called *precedence graphs*. Since any nontrivial application involves a huge number of task executions one would like to keep the description of the graph as compact as possible and in such a way that parallelism can be still extracted easily from the representation. For this purpose, we have introduced in [9] a new graph model called *dynamic process graph*, DPG for short.

This graph model allows a natural representation for parallel and distributed programs. In particular due to different modes for the input and output behaviour for each task they model basic primitives for specifying parallel pro-

[★] Supported by DFG research grant RE 672/3.

grams, like *fork* and *join*. PAR *output mode* of a task models a typical fork-constructor which allows to split a process into subprocesses running in parallel. Another fork-constructor allows a program to carry on in two or more alternative ways. The selection of an alternative can be *nondeterministic* or piggybacked on a *guard*. For example if each guard is a read instruction associated with a specific channel, we continue with the alternative guarded by the channel which supplies some data first. Hence, the decision how to continue the program often depends on the scheduler itself. We model such an alternative fork operation by an ALT *output mode*. Speaking more formally, we require that if the *output mode* of a task v is PAR then *all* of its direct successors have to be initiated and in case of ALT mode *one* of its direct successors has to be initiated. In a similar way we also specify the *input mode* of tasks. For the ALT input mode *one* of its direct predecessors (resp. *all* of them in case PAR) has to be completed before v starts.

One of the motivating questions for our study of DPGs is *how efficiently a compactly specified program Π can be executed on a multiprocessor system*. In [9], [10] and [11] we have investigated the complexity of scheduling strategies, where the compiler *completely* precomputes when and where each task will be executed. Changing from compile-time strategies to scheduling tasks at run-time one gets the potential of reducing the total execution time of a program only because the resources are better used, but in general there will be more effort necessary at run-time. Therefore, many existing parallel systems use both attempts to schedule parallel programs efficiently solving, however most details of the schedule already at compile time. E.g. some systems assign execution instances to the processors at compile-time and a local run-time scheduler invokes execution instances assigned to the processors (see [8] and [3]).

In [9,10] we have proven that it is intractable for the compiler to construct a *complete* schedule of minimum length for a given program Π . Speaking more precisely we have shown that for some types of programs the appropriate decision problems become $\mathcal{NEXPTIME}$ -complete. In this paper we continue our study and investigate some basic questions concerning efficient execution of a given parallel or distributed program using DPGs to model the programs. We give the precise computational complexities of these problems for various variants of DPGs. As consequence we obtain that computing some simple details of the schedule already at compile time is intractable, too.

Assuming a common graph representation (e.g. adjacency matrix), one obtains that the following simple problems (i) *what is the depth of a given input graph?* or (ii) *what is the size of a subgraph induced by a given node v ?* can be solved in $\mathcal{P}$. The corresponding decision problems are $\mathcal{NL}$ -complete. In this paper we prove that for DPG representation the first problem is $\mathcal{NP}$ -complete. Hence the complexity jumps from $\mathcal{NL}$ -complete for a common graph representation to $\mathcal{NP}$ -complete when representing the input as a DPG. A similar jump has been observed for some other classical graph problems in [5,14,12] where the authors have shown that simple graph properties become $\mathcal{NP}$ -complete when the graph is represented in a particular succinct way using generating circuits

or a hierarchical decomposition. On the other hand under these representations graph properties that are ordinarily $\mathcal{NP}$ -complete, like HAMILTON CYCLE, 3-COLORABILITY, CLIQUE etc., become $\mathcal{NEXPTIME}$ -complete. Later Feigenbaum *et al.* [4] have proven a similar property for graphs represented as OBDDs. In this paper we show that the problem (ii) above is $\mathcal{NEXPTIME}$ -complete for DPG representation what implies an astonishing jump from $\mathcal{NL}$ - to $\mathcal{NEXPTIME}$ -completeness.

The remaining part of this paper is organised as follows. In Section 2 we give a formal definition of DPGs and some important properties of this representation. Next, in Section 3 the basic graph problems for DPGs are defined and relationships between these problems and some canonical scheduling problems are given. Sections 4 and 5 deal with the complexities of decision, resp. counting problems. For definitions of standard notions in complexity we refer e.g. to [13].

2 Preliminaries

Given a DAG $G = (V, E)$ with node set V and edges E , for $v \in V$ let $\mathbf{pred}(v)$ denote the set of direct predecessors of v , and $\mathbf{succ}(v)$ its direct successors. Let $\mathbf{pred}^*(v)$ be the set of all ancestors of v (including v).

Definition 1. A **dynamic process graph**, DPG for short, $\mathcal{G} = (V, E, I, O)$ consists of a DAG (directed acyclic graph) with nodes V and edges E and two node labellings $I, O : V \rightarrow \{\text{ALT}, \text{PAR}\}$. $V = \{v_1, \dots, v_n\}$ represents a set of **processes** and E dependencies among them. I and O describe **input**, (resp. **output**) **modes** of the processes v_i . A finite DAG $H = (W, F)$ is a **run** of $\mathcal{G}$ iff the following conditions are fulfilled:

1. The set W is partitioned into subsets $W(v_1) \dot{\cup} W(v_2) \dot{\cup} \dots \dot{\cup} W(v_n)$. The nodes in $W(v_i)$ are **execution instances** of the process v_i .
2. Each source node of $\mathcal{G}$, which represents a starting operation of the program modelled by $\mathcal{G}$, has exactly one execution instance in H .
3. For every $v \in V$ with $\mathbf{pred}(v) = \{u_1, \dots, u_p\}$ and $\mathbf{succ}(v) = \{w_1, \dots, w_r\}$ and every execution instance $x \in W(v)$ it holds:
 - if $I(v) = \text{ALT}$ then x has a unique predecessor $y \in \bigcup_{i \in \{1, \dots, p\}} W(u_i)$;
 - if $I(v) = \text{PAR}$ then $\mathbf{pred}(x) = \{y_1, \dots, y_p\}$ with $y_i \in W(u_i)$ for each i ;
 - if $O(v) = \text{ALT}$ then x has a unique successor $z \in \bigcup_{i \in \{1, \dots, r\}} W(w_i)$;
 - if $O(v) = \text{PAR}$ then $\mathbf{succ}(x) = \{z_1, \dots, z_r\}$ with $z_j \in W(w_j)$ for each j .

We call a DPG $\mathcal{G}$ **executable** if and only if there exist runs for it. Given $\mathcal{G}$ with run $H = (W, F)$, for each edge (u, v) in $\mathcal{G}$ we define $F(u, v) := \{(y, z) \in F \mid y \in W(u) \text{ and } z \in W(v)\}$.

Throughout the paper we will illustrate the nodes of a DPG by boxes. Moreover the ALT input mode of a node will be illustrated by a white upper-part of a box and the PAR input mode by a black upper-part. Analogously, white and black lower-parts of boxes will correspond to ALT, resp. PAR output modes. An example of a DPG and its runs is given in Fig. 1.

DPGs can be used to specify parallel programs in a compact way. Then a run corresponds to an actual execution of the program. Note that if one considers

runs for a given DPG $\mathcal{G} = (V, E, I, O)$ then the enumeration of the execution instances inside each set $W(v)$ is inessential for the analyses of most properties of the run. Therefore we say that two runs $H_1 = (W_1, F_1)$ and $H_2 = (W_2, F_2)$ are equal if for $V = \{v_1, \dots, v_n\}$ it holds: (i) $|W_1(v_i)| = |W_2(v_i)|$ for any $v_i \in V$ and (ii) there exist permutations $\pi_1, \dots, \pi_n$ such that for any $W_1(v_i) = \{u_1, \dots, u_p\}$, $W_1(v_j) = \{w_1, \dots, w_r\}$, and $W_2(v_i) = \{u'_1, \dots, u'_p\}$, $W_2(v_j) = \{w'_1, \dots, w'_r\}$, it holds $(u_k, w_\ell) \in F_1$ iff $(u'_{\pi_i(k)}, w'_{\pi_j(\ell)}) \in F_2$.

Note that (according to this definition) all runs in Fig. 1 are different though they are isomorphic in a common graph-theoretic sense.

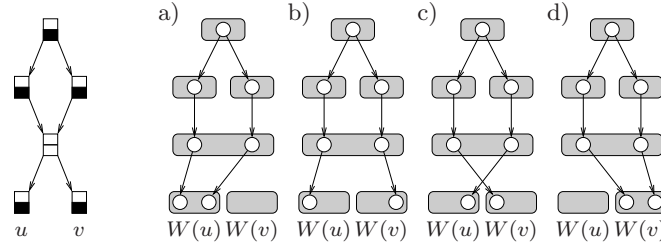


Fig. 1. A DPG (left) with four different runs. The gray boxes are used for clarity only and they indicate the execution instances of particular precesses

Observe that a run can be smaller than its DPG (for an example see [9]) but more typically a run will be larger than the DPG itself since the **PAR**-constructor allows task duplications like in Fig. 1. A similar behaviour can be achieved by a *replicative* fork operator (see for example the replicative **PAR** statement in the parallel programming language OCCAM). The following lemma gives an upper bound on the blow-up, resp. the possible compaction ratio of DPGs.

Lemma A [9]. *Let $\mathcal{G} = (V, E, I, O)$ be a DPG and $H = (W, F)$ be a corresponding run. It holds $|W| \leq 2^{|V|-1}$, and this general upper bound is best possible.*

Thus, certain DPGs have processes with exponential many execution instances.

Definition 2. *Assume that $H = (W, F)$ is a run of a DPG $\mathcal{G}$ with processes $V = \{v_1, \dots, v_n\}$. The sequence $\chi = (|W(v_1)|, |W(v_2)|, \dots, |W(v_n)|)$ will be called the **characteristic vector** of H and the matrix $\psi = (|F(v_i, v_j)|)_{1 \leq i, j \leq n}$ the **characteristic matrix** of H . Furthermore, let $\chi(\mathcal{G})$ be the set of all characteristic vectors of $\mathcal{G}$ and $\psi(\mathcal{G})$ be the set of all its characteristic matrices.*

Note that if χ is a characteristic vector of a DPG with n processes then according to Lemma A the values $\chi(i)$ are bounded by 2^{n-1} (in a full version of [9] we show that $\chi(i) \in [0..2^{n-2}]$ and that the upper bound can occur). Below we give necessary and sufficient conditions that χ is a characteristic vector and ψ a characteristic matrix of a given DPG.

Lemma B [9]. *Let $\mathcal{G}$ be a DPG with processes $V = \{v_1, \dots, v_n\}$. Then $\chi \in \mathbb{N}^n$ is a characteristic vector and an $n \times n$ matrix ψ is a corresponding characteristic*

matrix of a run of $\mathcal{G}$ iff for any source v_s , $\chi(s) = 1$, and for any non-sink v_i , any non-source v_j and, any edge (v_i, v_k) and (v_k, v_j) the following conditions hold:

$$\chi(i) = \begin{cases} \sum_{\ell} \psi(i, \ell) & \text{if } O(v_i) = \text{ALT} \\ \psi(i, k) & \text{if } O(v_i) = \text{PAR} \end{cases} \quad \chi(j) = \begin{cases} \sum_{\ell} \psi(\ell, j) & \text{if } I(v_j) = \text{ALT} \\ \psi(k, j) & \text{if } I(v_j) = \text{PAR}. \end{cases}$$

Hence a characteristic matrix fully determines an appropriate characteristic vector of a run. But, given a characteristic vector there exist runs having different characteristic matrices. Let $\text{run}(\mathcal{G}) := \{ H \mid H \text{ is a run of } \mathcal{G} \}$. Then define $\#\text{run}(\mathcal{G})$ as the number of different runs of $\mathcal{G}$. In [9] we have shown

Lemma C [9]. *Any dynamic process graph $\mathcal{G}$ has at most double exponential many different runs and this bound can actually occur.*

The formulae below gives an exact number of runs for any DPG.

Lemma 1. *Let for $\psi \in \psi(\mathcal{G})$, χ_ψ denote an appropriate characteristic vector for ψ . Then it holds*

$$\#\text{run}(\mathcal{G}) = \sum_{\psi \in \psi(\mathcal{G})} \left(\prod_{I(v_i)=\text{PAR}} (\chi_\psi(i)!)^{|\text{pred}(v_i)|-1} \right) \prod_{\substack{\text{non-sink } v_i \\ \text{with } O(v_i)=\text{ALT}}} \frac{\chi_\psi(i)!}{\prod_{1 \leq j \leq n} \psi(i, j)!}.$$

Definition 3. *Let $\mathcal{G} = (V, E, I, O)$ be a DPG and $H = (W, F)$ be a run of $\mathcal{G}$. A subrun $\mathcal{R}(v)$ of H is a subgraph that is induced by one of its sink nodes v , i.e. it consists of a sink v and all nodes in $\text{pred}^*(v)$ together with all their connections in F .*

The maximal size of a subrun of H gives a better upper time bound for executing H than just the size of H because all subruns can be executed independently in parallel. Hence H can be executed at least in linear time with respect to the maximal size of a subrun of H .

Lemma D [10]. *There exists a family of DPGs $\mathcal{G}_k = (V_k, E_k, I_k, O_k)$ with $|V_k| = 2k+1$ such that every run H_k of $\mathcal{G}_k$ has a subrun of size $3 \cdot 2^{((|V_k|-1)/2)-1}$.*

This means that for some cases a subrun can be huge. However, in [10] we have shown that the situation changes drastically if one considers DPGs with output mode either $O \equiv \text{ALT}$, or $O \equiv \text{PAR}$. Then the subruns have size at most quadratic with respect to the size of the DPG. In this paper we will investigate the complexity to compute the exact size of such subruns.

3 Basic Problems and Motivating Questions

In the previous papers [9,10,11] we have investigated what is the complexity of static scheduling for programs given as DPGs. Static scheduling is a problem, where the compiler has to completely precompute when and where each execution instance will be executed. Assuming that all dependencies among particular execution instances are data-independent and hence all of them can be

determined during compile-time we have given a precise characterisation of the computational complexity of the problem to find an optimal schedule. For our study we have assumed a massive parallelism, i.e. unbounded number of processors with communication delay between executions of successive tasks executed on different processors.

Since it is extremely difficult to construct an optimal schedule during compile-time we consider in this paper some natural and simple graph functions for DPGs which can be applied to approximate the length of an optimal schedule. We will discuss such approximations for the m -processor scheduling as well as the scheduling with communication delay. Let $\mathcal{G}$ be a DPG. Then we define

$$\begin{aligned} \text{run-size}(\mathcal{G}) &:= \min_{H \in \text{run}(\mathcal{G})} |W_H|, & \text{run-depth}(\mathcal{G}) &:= \min_{H \in \text{run}(\mathcal{G})} \text{depth}(H), \\ \text{subrun-size}(\mathcal{G}) &:= \min_{H \in \text{run}(\mathcal{G})} \max_{v \in W_H} |\text{pred}^*(v)|, \end{aligned}$$

where W_H denotes nodes of H and $\min_{x \in A} x := 0$ for the empty set A . Note that if $\mathcal{G}$ describes a parallel program then $\text{run-depth}(\mathcal{G})$ corresponds just to the parallel execution time of the fastest run of this program in the system where one assumes unit execution time of each execution instance, no communications delay between processors, and no bound on the number of available processors. Moreover $\text{run-size}(\mathcal{G})$ corresponds to the smallest work-time of a run for the program (i.e. the total number of operation) and as we will see later $\text{subrun-size}(\mathcal{G})$ approximates the parallel execution time of the fastest run on a multiprocessor system with communications delay between processors. In [11] we have given some nontrivial upper bounds for $\text{subrun-size}(\mathcal{G})$.

Based on these elementary functions we define the following three decision problems that we will call the **BASIC-RUN problems**:

Definition 4. Let $\mathcal{G}$ be a given DPG and B be a bound. Then define:

RUN-SIZE problem: Does $\text{run-size}(\mathcal{G}) \leq B$ hold?

SUBRUN-SIZE problem: Does $\text{subrun-size}(\mathcal{G}) \leq B$ hold?

RUN-DEPTH problem: Does $\text{run-depth}(\mathcal{G}) \leq B$ hold?

The main results of this paper give the precise computational complexities for these problems.

Before presenting these results in detail let us discuss some examples how one can use the basic functions defined above to estimate how efficiently a program can be executed. Let us consider first the following canonical scheduling problem: Given a set of execution instances W , each having length 1, a number $m \in \mathbb{N}$ of identical processors, a directed acyclic graph $H = (W, F)$ describing a partial order on W , and a deadline $T^* \in \mathbb{N}$, decide whether there is an m -processor schedule S for W that obeys the precedence constraints H , i.e. such that (u, v) implies $S(u) \geq S(v) + 1$, with $T(S) \leq T^*$? Here $T(S)$ denotes the duration of S , i.e. the point of time when S has executed all execution instances. It is well known that this problem is $\mathcal{NP}$ -complete (see e.g. [6]). An m -processor schedule S for a DPG $\mathcal{G}$ is an m -processor schedule of a run $H = (W, F)$ of $\mathcal{G}$. Let $T_{\text{opt}}(\mathcal{G}, m) := \min\{T(S) : S \text{ is an } m\text{-processor schedule for } \mathcal{G}\}$.

The **m -processor DPG SCHEDULE Problem** is defined as follows: Given a DPG $\mathcal{G}$ and two numbers m and T^* , does $T_{opt}(\mathcal{G}, m) \leq T^*$ hold?

It is easy to see that the function run-size gives a nontrivial approximation for this scheduling problems. Namely, for given DPG $\mathcal{G}$ and any $m > 0$ it holds: $\text{run-size}(\mathcal{G})/m \leq T_{opt}(\mathcal{G}, m) \leq \text{run-size}(\mathcal{G})$. If the number of processors m is large enough then the common m -processor scheduling problem becomes tractable: e.g. if $m \geq |H|$ then the optimal time of a schedule for $H = (W, F)$ is equal to the depth of H . For the corresponding problem for DPGs we obtain similarly that the optimal scheduling time $T_{opt}(\mathcal{G}, m)$ is equal to $\text{run-depth}(\mathcal{G})$. However, as we will see later the problem to compute $\text{run-depth}(\mathcal{G})$ remains intractable.

Let us consider now the scheduling problem for systems with communication delays between processors. Speaking more formally we consider systems which take delays into account occurring when one processor sends a piece of data to another one. This scheduling problem was introduced by Papadimitriou and Yannakakis in [15].

For a given set of execution instances W , each having length 1, and a directed acyclic graph $H = (W, F)$ describing a partial order on W , the communication delay will be specified by a function $\delta : F \rightarrow \mathbb{N}$, which defines the time necessary to send data from one processor to another one. For simplification we assume that this delay is independent of the particular pair of processors. Scheduling with communication delays requires the following condition to be fulfilled: *if a task v is executed on processor p at time t then for each direct predecessor u of v holds that u has been finished either on p by time $t - 1$, or on some other processor p' by time $t - 1 - \delta(u, v)$.* For the appropriate scheduling problems no bound will be assumed on the number of processors available in the system.

A schedule S for a DPG $\mathcal{G} = (V, E)$ with delay δ is a schedule of a run $H = (W, F)$ with delay δ_H such that for any $e \in F(u, v)$, with $u, v \in V$ it holds $\delta_H(e) = \delta(u, v)$. Now define the function $T_{opt}(\mathcal{G}, \delta) := \min\{T(S) : S \text{ is a schedule for } \mathcal{G} \text{ with delay } \delta\}$. The **DPG SCHEDULE with communication delay** is defined as follows: Given a DPG $\mathcal{G}$ with communication delay δ and a deadline T^* , does $T_{opt}(\mathcal{G}, \delta) \leq T^*$ hold?

Using the run functions we get: $\text{run-depth}(\mathcal{G}) \leq T_{opt}(\mathcal{G}, \delta) \leq \text{subrun-size}(\mathcal{G})$ that holds for any DPG $\mathcal{G}$ and communication delay δ . Moreover for sufficiently large δ we obtain $T_{opt}(\mathcal{G}, \delta) = \text{subrun-size}(\mathcal{G})$.

Note that for a given DPG $\mathcal{G}$ it plays a crucial role whether $\#\text{run}(\mathcal{G}) > 0$ holds? In case $\#\text{run}(\mathcal{G}) = 0$, $\mathcal{G}$ has simply no run and hence it is not executable. In [9] we have investigated the complexity of this problem that we called the EXECUTION problem. In Section 5 we investigate the exact complexity for computing the value $\#\text{run}(\mathcal{G})$.

4 The Complexity of Basic-Run Problems

An interesting feature of ALT-input mode is that the execution instances of processes with ALT-input, work in parallel without any synchronization between

them. If one restricts e.g. input mode of all nodes of a DPG to ALT only, then each run of such DPG is just a forest of trees each rooted with a node corresponding to an appropriate source of the DPG. Each execution instance of such runs initiates new execution instances that work disjointly. Hence such DPGs describe programs executing multithreads running without any synchronization. Equivalently they correspond to the computation graphs of alternating Turing machines (ATMs for short). Recall that by a computation graph of a Turing machine M on input X we mean a directed acyclic graph with nodes describing configurations of M on X and edges corresponding to direct computation steps. Then the question whether an input string X is accepted by an ATM M working in logarithmic space is equivalent to the problem whether for a given DPG $\mathcal{G}$, $\#run(\mathcal{G}) > 0$ (we skip formal definitions and a proof for this equivalence in this extended abstract).

On the other hand the synchronization between execution instances is possible due to PAR input mode. Below we discuss the complexity of the BASIC-RUN problems for different variants of DPGs starting with the simplest model where no synchronization mechanisms are allowed, i.e. with DPGs restricting to ALT inputs. To present our results in a compact way let $[I | O]$, where $I, O \subseteq \{\text{ALT}, \text{PAR}\}$, denote all DPGs with the input modes and output modes, restricted only to I , resp. O .

4.1 Complexity Issues for DPGs without Synchronization

As we observed previously runs of $[\text{ALT} | \text{ALT}, \text{PAR}]$ DPGs correspond in a natural way to computations of alternating Turing machines. Restricting further the output mode only to ALT-mode we obtain $[\text{ALT} | \text{ALT}]$ DPGs that describe exactly computation graphs of nondeterministic Turing machines working in logarithmic space. Restricting outputs to PAR only, we obtain $[\text{ALT} | \text{PAR}]$ DPGs that are equivalent with the computation graphs of co-nondeterministic Turing machines (i.e. II -TMs).

Theorem 1. *For $[\text{ALT} | \text{ALT}]$ DPGs all BASIC-RUN Problems are $\mathcal{NL}$ -complete and for $[\text{ALT} | \text{ALT}, \text{PAR}]$ DPGs they become $\mathcal{P}$ -complete. Moreover for $[\text{ALT} | \text{PAR}]$ DPGs the SUBRUN-SIZE and RUN-DEPTH problems remain $\mathcal{NL}$ -complete and the RUN-SIZE problem is $C=\mathcal{L}$ -complete.*

Here $C=\mathcal{L}$ denotes a class of decision problems for which there exist nondeterministic Turing machines such that (i) the machines work in logarithmic space and (ii) for any input string the number of accepting computation paths is equal to the number of rejecting computations. For a formal definition and for motivation to study this class see [1].

4.2 DPGs with Strictly Synchronized Runs

It is easy to see that for a common representation of graphs the problem to compute the depth of given directed acyclic graph G and the problem to compute

the size of a subgraph that is induced by one of its nodes are tractable. If one considers a $[\text{PAR} \mid \text{PAR}]$ DPG $\mathcal{G}$ then obviously its *unique* run H is just a graph isomorphic to $\mathcal{G}$. Therefore all the BASIC-RUN problems become tractable. The situation changes drastically if one allows ALT output mode. Then we obtain that even for $[\text{PAR} \mid \text{ALT}]$ DPGs the BASIC-RUN problems become $\mathcal{NP}$ -hard. More specifically we have

Theorem 2. *Restricting DPGs to $[\text{PAR} \mid \text{PAR}]$ the SUBRUN-SIZE and RUN-DEPTH problems remain $\mathcal{NL}$ -complete and the RUN-SIZE problem is in $\mathcal{NC}^1$. For $[\text{PAR} \mid \text{ALT}]$ DPGs all BASIC-RUN Problems are $\mathcal{NP}$ -complete. Moreover they remain $\mathcal{NP}$ -complete for $[\text{PAR} \mid \text{ALT}, \text{PAR}]$ DPGs.*

4.3 The General Case

In this section we focus on DPGs allowing both ALT and PAR input modes. First we consider DPGs restricted to ALT output mode. Since each run of a $[\text{ALT}, \text{PAR} \mid \text{ALT}]$ DPG is of polynomial size, one can still solve the BASIC-RUN Problems in $\mathcal{NP}$. Hence from Theorem 2 we conclude

Theorem 3. *For $[\text{ALT}, \text{PAR} \mid \text{ALT}]$ DPGs the BASIC-RUN problems remain $\mathcal{NP}$ -complete.*

Below we present one of the most important results of this paper. It shows that for our representation of graphs the computational complexity of some decision problems increases dramatically, namely, it jumps from $\mathcal{NL}$ -complete for a common representation to $\mathcal{NEXP}$ -complete when we represent the input as a DPGs.

Theorem 4. *Restricting DPGs to $[\text{ALT}, \text{PAR} \mid \text{PAR}]$ the SUBRUN-SIZE problem becomes $\mathcal{NEXP}$ -complete. Moreover, the RUN-SIZE problem is $\mathcal{C=L}$ -complete and RUN-DEPTH problem remains $\mathcal{NL}$ -complete.*

To show the $\mathcal{NEXP}$ -hardness for SUBRUN-SIZE is the most difficult and complicated part of the proof for this theorem. The main idea which enabled us to prove this result is that DPGs can be used to model computations of Boolean circuits. We used this idea previously in [9] to show the $\mathcal{NEXP}$ -completeness for DPG-SCHEDULING with communication delay. In this paper we extend and modify the methods of [9].

Theorem 5. *For unrestricted DPGs the SUBRUN-SIZE problem is $\mathcal{NEXP}$ -complete and the RUN-SIZE and RUN-DEPTH problems are $\mathcal{NP}$ -complete.*

The $\mathcal{NEXP}$ -hardness for the SUBRUN-SIZE problem follows directly from Theorem 4. To solve the problem in $\mathcal{NEXP}$ one can just guess nondeterministically a run H for a given DPG $\mathcal{G}$ and then compute deterministically the maximal size of a subgraph induced by a node of H . Since the size of any run for $\mathcal{G}$ is at most exponential with respect to the size of $\mathcal{G}$ (Lemma A) this can be done in exponential time. Similarly, the $\mathcal{NP}$ -hardness of RUN-SIZE and RUN-DEPTH

problems follows from the hardness for [PAR | ALT] DPGs (Theorem 2). Hence it remains to show $\mathcal{NP}$ algorithms for RUN-SIZE and RUN-DEPTH. Note that this task is not obvious. First, the sizes of the runs can be huge (i.e. even exponential with respect to the size of the given DPG), hence an algorithm which just guesses a run and then tests an appropriate property is useless. Moreover, to compute the depth of a run it does not suffice e.g. to estimate the length of all the paths in the given DPG because some of the paths have no counterparts in a run at all (see Fig. 1). Similarly, considering a given run H , one can observe that some subruns of H are non-isomorphic to any subgraph of the given DPG (see Fig. 2). Below we show how to solve these difficulties.

To decide in $\mathcal{NP}$ whether $\text{run-size}(\mathcal{G}) \leq B$ one guess nondeterministically a characteristic vector χ and a characteristic matrix ψ for $\mathcal{G}$. Using the conditions of Lemma B one can verify whether these guesses are correct. Then it suffices to test whether $\sum_{v_i \in V} \chi(i) \leq B$. Note that this can be done in polynomial time because the size of a binary representation of χ and ψ is polynomial with respect to $|V|$. The RUN-DEPTH problem can be solved in $\mathcal{NP}$ as follows:

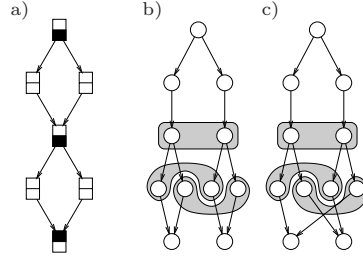


Fig. 2. A DPG (a) with two runs: (b) and (c). Any subrun of the run (c) is not isomorphic to a subgraph of the DPG

```

procedure run-depth( $\mathcal{G} = (V, E, I, O), B, \chi, \psi$ )
1 for each  $v_i$  choose nondet.  $A_0(i), \dots, A_B(i)$  such that  $\chi(i) = \sum_{d \leq B} A_d(i)$ 
2 for each  $(v_i, v_j)$  and  $d, k$  with  $0 \leq d < k \leq B$ 
3   choose nondet.  $A_{d,k}(i, j)$  such that  $\psi(i, j) = \sum_{d < k \leq B} A_{d,k}(i, j)$ 
4 for all non-sinks  $v_i$  of  $\mathcal{G}$  and for each depth  $d \leq B$  do
5   if  $O(v_i) = \text{ALT} \wedge A_d(i) \neq \sum_{(v_i, v_j) \in E} \sum_{k > d} A_{d,k}(i, j)$  then reject
6   if  $O(v_i) = \text{PAR} \wedge \exists (v_i, v_j) \in E : A_d(i) \neq \sum_{k > d} A_{d,k}(i, j)$  then reject
7 for all non-sources  $v_i$  of  $\mathcal{G}$  and for each depth  $d \leq B$  do
8   if  $I(v_i) = \text{ALT} \wedge A_d(i) \neq \sum_{(v_j, v_i) \in E} \sum_{k < d} A_{k,d}(i, j)$  then reject
9   if  $I(v_i) = \text{PAR} \wedge \exists (v_i, v_j) \in E : A_d(i) \neq \sum_{k < d} A_{k,d}(j, i)$  then reject
10 accept

```

5 Counting Runs

In this section we investigate a complexity of the following problem: *for a given DPG $\mathcal{G}$ compute $\# \text{run}(\mathcal{G})$* . Let us denote this problem by $\# \text{RUN}$. Similarly as for the decision problems we show, starting with the most complex case, that the complexity of $\# \text{RUN}$ varies with the variants of DPGs.

Theorem 6. *For a given DPG $\mathcal{G}$ $\# \text{RUN}$ can be solved in polynomial space.*

To prove this theorem one computes $\#run(\mathcal{G})$ using the Chinese Remainder Representation for integers. Then we achieve the claim combining the prime number theorem and recent result of Chiu, Davida, and Litow [2].

Because of the space limitation we will only shortly adumbrate the results of more restricted cases, and give some hints to the main ideas for proving the computational complexity.

Restricting DPGs to $[ALT, PAR \mid ALT]$ or to $[PAR \mid ALT, PAR]$ the values of characteristic vectors are linear with respect to the number of nodes of a given DPG and not exponential as in the general case. Hence the size of a representation of any characteristic vector χ is bounded polynomially with respect to the size of $\mathcal{G}$. Additionally the question whether a given χ is a correct characteristic vector of $\mathcal{G}$ can be answered deterministically in polynomial time (using e.g. formulas of Lemma 1). Therefore we have that $\#RUN$ is in $\#P$. On the other hand the $\#P$ -hardness of this problem for $[PAR \mid ALT]$ DPGs can be proved by a reduction from the two-dimensional matching problem – a canonical complete problem for $\#P$.

For the $[ALT \mid ALT, PAR]$ DPGs the $\#RUN$ problem becomes tractable. A straightforward analysis shows that the computation of $\#run(\mathcal{G})$ can be done in $\mathcal{FP}$. Restricting DPGs to $[ALT \mid ALT]$ we decrease further the complexity of $\#RUN$ problem. One can show that for such DPGs the problem is equivalent to evaluating of the number of accepting computation paths for a nondeterministic logarithmic space bounded TM. Hence, it is $\#L$ -complete. The easiest cases are the restrictions to $[ALT \mid PAR]$ or $[PAR \mid PAR]$ DPGs because for any such graph $\mathcal{G}$ we have $\#run(\mathcal{G}) = 1$.

From the formulae of Lemma 1 for $\#run(\mathcal{G})$ one obtains a very natural approximation for this value, namely for any DPG $\mathcal{G}$ we have that $\#run(\mathcal{G}) \geq |\psi(\mathcal{G})|$. Recall that $\psi(\mathcal{G})$ denotes the set of all characteristic matrices of $\mathcal{G}$. It is easy to see that for some variants of DPGs these two numbers are equal. For some other DPGs one can approximate both products in the formulae of Lemma 1 by using the maximum value in the characteristic matrix and hence using additionally the value of $|\psi(\mathcal{G})|$ one obtains a good approximation for $\#run(\mathcal{G})$. Therefore it is reasonable to investigate the complexity of evaluating $|\psi(\mathcal{G})|$. Our first observation which seems to simplify this question says that: *the complexity of computing $|\psi(\mathcal{G})|$ is equal to the complexity for computing $|\chi(\mathcal{G})|$* . Hence we will concentrate on the complexity issues for the number of vectors. Let $\#\chi(\mathcal{G}) := |\chi(\mathcal{G})|$ and $\#\chi$ denote the appropriate counting problem.

In Table 1 we summarise our results for the problem $\#\chi$. It seems that the most interesting cases are the restrictions to $[ALT \mid ALT]$ and $[ALT \mid ALT, PAR]$ DPGs. The $\#\chi$ problem is $\#P$ -complete for DPGs restricted even to $[ALT \mid ALT]$. One can show this using a polynomial-time one-Turing reduction from counting PERFECT MATCHINGS of bipartite graphs ([16]).

Theorem 7. *For a given DPG the counting problems $\#RUN$ and $\#\chi$ have the complexities given in the Table 1 below.*

Table 1. The computational complexity of the $\#RUN$ and $\#\chi$ problems for DPGs with respect to input and output modes that may occur in DPGs. Remark: for $[ALT, PAR \mid PAR]$ DPGs we can consider the $\#\chi$ counting problem just as a decision problem because for such DPGs the value of $\#\chi$ is either 0 or 1

input mode	output mode	#RUN	# χ	
ALT or PAR	PAR	trivial		
ALT	ALT	# $\mathcal{L}$ -complete	# $\mathcal{P}$ -complete	
ALT	ALT,PAR	$\mathcal{FP}$ -complete		
PAR	ALT or ALT,PAR	# $\mathcal{P}$ -complete		
ALT, PAR	ALT			
ALT, PAR	PAR	$\mathcal{PSPACE}$	$C=\mathcal{L}$ -complete	
unrestricted			# $\mathcal{P}$ -complete	

6 Conclusions

In this paper characterization of the complexity of some basic graph-theoretic problems are given assuming DPG representation. We have proven that they are intractable for most variants of DPGs. Hence, one can expect that also the scheduling problems for DPGs, namely SCHEDULE with communication delay and m -processor SCHEDULE are intractable, too. In fact, using the inequalities approximating the optimal scheduling times from Section 3 one can deduce from the results of Section 4 that the scheduling problems become $\mathcal{NEXPTIME}$ -complete. For scheduling with communication delay the completeness results have been already presented in [9,10].

Assuming succinct representations of graphs like generating Boolean circuits ([5,14]), hierarchical decompositions ([12]), or OBDDs ([4]) all $\mathcal{NP}$ -complete graph problems become $\mathcal{NEXPTIME}$ -complete. An interesting question is if a similar property holds for DPG representation. We answer this question negatively. Namely, let us consider the following RUN-COLOUR problem: *For a given DPG $\mathcal{G}$ and bound B , is it possible to colour a run H of $\mathcal{G}$ using at most B colours in such a way that for any edge (u, v) of H u and v have different colours?* Obviously the problem is $\mathcal{NP}$ -complete for $[PAR \mid PAR]$ DPGs. On the other hand modifying the procedure run-depth from Section 4.3 one can obtain an $\mathcal{NP}$ algorithm for this problem and hence it remains $\mathcal{NP}$ -complete even for $[ALT, PAR \mid ALT, PAR]$ DPGs.

References

1. E. Allender and M. Ogihara, *Relationships among PL, $\#L$, and the determinant*, RAIRO - Theoretical Informatics and Applications, Vol. 30, 1996, 1-21. 569
2. A. Chiu, G. Davida, and B. Litow *NC1 Division*, Unpublished manuscript, 1999. 572
3. H. El-Rewini and H. H. Ali, *Static Scheduling of Conditional Branches in Parallel Programs*, J. Par. Distrib. Comput. 24, 1995, 41-54. 563
4. J. Feigenbaum, S. Kannan, M. Vardi, and M. Viswanathan, *Complexity of Problems on Graphs Represented as OBDDs*, LNCS Proc. 15. STACS'98, 1998, 216-226. 564, 573

5. H. Galperin and A. Wigderson, *Succinct Representations of Graphs*, Information and Control, 56, 1983, 183-198. 563, 573
6. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide To the Theory of $\mathcal{NP}$ -Completeness*, Freeman, 1979. 567
7. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation – P-Completeness Theory*, Oxford University Press, 1994.
8. S. Ha and E. Lee, *Compile-time Scheduling and Assignment of Data-flow Program Graphs with Data-dependent Iteration*, IEEE Trans. Computers 40, 1991, 1225-1238. 563
9. A. Jakoby, M. Liśkiewicz, and R. Reischuk, *Scheduling Dynamic Graphs*, LNCS Proc. 16. STACS'99, 1999, 383-392. 562, 563, 565, 566, 568, 570, 573
10. A. Jakoby, M. Liśkiewicz, and R. Reischuk, *The Expressive Power and Complexity of Dynamic Process Graphs*, LNCS Proc. 26. WG'00, 2000, 230-243. 563, 566, 573
11. A. Jakoby, M. Liśkiewicz, and R. Reischuk, *Approximating Schedules for Dynamic Graphs Efficiently*, ECCC report TR01-038, 2001. 563, 566, 567
12. T. Lengauer, K. Wagner, *The Correlation between the Complexities of the Non-hierarchical and Hierarchical Versions of Graph Problems*, J. CSS 44, 1992, 63-93. 563, 573
13. C. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994. 564
14. C. Papadimitriou and M. Yannakakis, *A Note on Succinct Representations of Graphs*, Information and Control, 71, 1986, 181-185. 563, 573
15. C. Papadimitriou and M. Yannakakis, *Towards an Architecture-Independent Analysis of Parallel Algorithms*, SIAM J. Comput. 19, 1990, 322-328. 568
16. L. G. Valiant, *The Complexity of Enumeration and Reliability Problems*, SIAM J. Comput., Vol. 8, No. 3, 1979. 572

Delay Optimizations in Quorum Consensus

Xuemin Lin

School of Computer Science and Engineering, The University of New South Wales
Sydney, NSW 2052, Australia.
lxue@cse.unsw.edu.au

Abstract. The management of replicated data in distributed database systems is a classic problem with great practical importance. Quorum consensus is one of the popular methods, combining with eager replication, for managing replicated data. In this paper, we will investigate the problems of delay-optimal quorum consensus. Firstly, we will show that the problem of minimizing the total-delay (or mean-delay) is NP-hard. However, we can show that the problem restricted to some specific network topology, such as trees, rings, and meshes, can be solved in polynomial time. We also developed an approximate algorithm for a general case. The algorithm gives an approximate ratio less than 2. Secondly, we will present an efficient algorithm, based on the dynamic programming technique, to solve the problem of minimizing the maximal-delay.

Keywords: Quorum Consensus, Replicated Data Management, and Optimizations.

1 Introduction

The replicated data management in distributed databases is a classic problem with great practical importance. Distributed data warehouses and data marts contain a huge amount of replicated data distributed among a number of sites. Therefore, in recent developments [3,4,6,10] of the area there is always a trade-off among system efficiency, data availability, data freshness, and data consistency.

Two replicated data management methods are available in the literature: *eager* and *lazy*. Eager replication management gives the data *consistency* and the highest data *freshness*. However, it suffers from the system efficiency due to an application of 2-phase commit protocol [18]. On the other hand, lazy replication management provides high system efficiency but does not necessarily provide the data freshness. Moreover, pure lazy replication management does not generally guarantee the data consistency. Recent research results [3] reveal that it is the best if these two methods can be combined.

To achieve a high system efficiency, “quorum consensus” is often adopted in eager replication. In this paper, we will investigate the quorum consensus method. A quorum consensus method is based on the design of a “coterie” (to be formally defined in section 2), such that each data processing (read or write) operation is executed on a subset (*quorum* - an element in the coterie) of the data sites over the network.

Recent developments in quorum consensus are mainly focused on 1) minimizing the total communication costs for processing a given set of transactions, and 2) minimizing the number of remote sites to be communicated while assembling a quorum. A number of quorum consensus protocols [5,9,11,12,13,14,16,17,20,21] have been developed for these purposes.

Note that in quorum consensus, since messages are sent (possibly by the *multicast* mechanism [19]) to the multiple nodes in a quorum in order to ensure consistency of the operations the delays by passing messages through a long distance communication channel in a wide area network can create a bottleneck [18,8] in the response time. In this paper we will investigate the problems of minimizing the “average” (or total) delay and minimizing the “maximal” delay.

These two problems were first investigated in [8]. It provided several algorithms for the problem of minimizing the maximal delay with respect to special classes of network topologies, such as rings, trees, and clustered graphs, while the complexity in general was left open. For the problem of minimizing the average (or total) delay, the paper [8] provides an approximate algorithm with approximation ratio 1.25 for rings with uniform links and uniform nodes. However, the algorithm in [8] does not generally guarantee a constant approximation ratio.

The first contribution of this paper is that we show that the problem of minimizing the average (or total) delays is NP-hard. Besides this, we provide an approximate algorithm with approximation ratio 2 in general. Note that the average-delay minimization problem, which we study in the paper, is more general than the problem in [8] where each node takes only a unit weight. While the NP-hardness we show in this paper is restricted to the case where each node takes a unit weight, the approximation ratio of our algorithm covers a general case where each node can take an arbitrary weight. Further, we can show that our approximate algorithm guarantees the exact solution for rings with uniform links and uniform nodes in contrast with the approximation result in [8]. Moreover, we can show that our approximate algorithm can guarantee exact solutions for the popular network topologies, such as meshes [18] and trees. The second contribution of the paper is that we present an efficient polynomial time algorithm to solve the problem of minimizing the maximum delay of quorum consensus. Again, this improves the results in [8] which can handle only some special graph topology.

Note that in [1,19], the *file* allocation has been studied in an *on-line* environment with the assumption that each read is processed by reading one copy and each write has to be propagated to each copy - read one and write all policy. The duplicated data management, discussed in this paper, assumes that a file allocation is given, and investigates read/write policies instead of the policy of read one and write all. Therefore, the results and techniques in [1,19] are not applicable.

The rest of the paper is organized as follows. In the second section, we provide the background knowledge, and precisely define the problems. In the third section, we present our results for the problem of minimizing the average (or total) delays. The fourth section presents our algorithm for solving the problem

of minimizing the maximum delays. This is followed by conclusions and remarks. Due to the space limitation, we will not detail every proof; the interested readers may refer to the full paper [15] for details.

2 Preliminaries

A network is represented by a weighted graph $G = (V, E)$ where for each edge (physical link) $(u, v) \in E$, $d_{u,v}$ denotes the communication delay to send a unit message along the link from node u to v . A set S of subsets of V is *coterie* for G if and only if the following conditions hold [9,20]:

1. *Intersection*: $\forall Q_1, Q_2 \in S, Q_1 \cap Q_2 \neq \emptyset$.
2. *Non-redundancy*: $\forall Q_1, Q_2 \in S, Q_1 \not\subseteq Q_2$.
3. *Connectivity*: $\forall Q \in S$, the subgraph G_Q induced [7] by Q from G is connected.

Each element Q in a coterie is called a *quorum*. The intersection property guarantees that any pair of quorums in a coterie have at least one common node. The connectivity of the nodes (vertices) in each quorum of a coterie is an important and practical restriction [11,20]. If a vertex u is connected to vertex v only through z in a network, then we consider $\{u, z, v\}$ as a quorum rather than $\{u, v\}$; this is because a message between u and v must go through z .

Below is the notation to be used in the paper. We use, $V(G)$ to denote the vertex set of G and $l_{u,v}|_G$ to denote the length of the *shortest path* between u and v in G .

Suppose that S is a coterie for a network G ; and that a vertex u uses $Q \in S$ as a quorum to execute a data operation issued from u . If the network adopts a multicast mechanism, then the *delay* of sending a message from u to every vertex (site) in Q is the length of the longest shortest path from u to a vertex of Q in G_Q ; and it can be defined as: $\text{delay}(u, Q) = \max_{v \in Q} \{l_{u,v}|_{G_Q}\}$.

Given a coterie S , for each node u we define the smallest value of its delay with respect to all quorums as the *delay* of u in S ; that is, $\text{delay}(u, S) = \min_{Q \in S} \{\text{delay}(u, Q)\}$. Further, for a coterie S we define the *max-delay* as:

$$\text{max-delay}(S) = \max_{u \in V(G)} \{\text{delay}(u, S)\}. \quad (1)$$

The problem of minimizing the maximum delay (MMD) is to find a coterie for a given network G such that (1) is minimized.

Note that to apply a quorum consensus method, each data object must have a coterie designed. Different data objects may have different coterie designed. The problems discussed in this paper are about an optimal design of coterie for a given data object. Without loss of generality, we may assume a full replication of each data object, that is, a data object is replicated to every node (vertex) over the network. (For a data object partially replicated over the network, the results in this paper are also applicable; and we will discuss this in the last section.)

To define the problem of minimizing the average delay for a given data object, we assume that each vertex u is also associated with a weight w_u to represent the number of the messages issued at this site for reading/writing the data object, which have to be passed to an entire quorum. The average delay of a coterie S is defined as:

$$\text{avg-delay}(S) = \frac{\sum_{u \in V(G)} w_u \text{delay}(u, S)}{\sum_{u \in V(G)} w_u} \quad (2)$$

The problem of minimizing the average delay for a given data object over a network with weights specified on each vertex is to find a coterie such that (2) is minimized. As $\sum_{u \in V(G)} w_u$ is a constant, the problem of minimizing the average delay is equivalent to the problem of minimizing the total delays:

$$\text{tot-delay}(S) = \sum_{u \in V(G)} w_u \text{delay}(u, S) \quad (3)$$

In this paper we will present our results for the problem of minimizing the total delays (MTD).

In a coterie S , it is clear that for each vertex u there is a quorum $Q_u \in S$ such that $\text{delay}(u, Q_u) = \text{delay}(u, S)$, and $\{Q_u : u \in V\}$ also form a coterie. This implies that both MMD and MTD problems can be simplified as:

Minimizing Max-Delays (MMD)

INSTANCE: A weighted graph $G = (V, E)$, and $\forall (u, v) \in E$ $d_{u,v}$ is an integer.

QUESTION: Find a set $\Pi = \{G_u : u \in V\}$ of connected subgraphs of G such that

- $S = \{V(G_u) : G_u \in \Pi\}$ is a coterie of G ;
- $\forall u \in V$, $\text{delay}(u, V(G_u)) = \text{delay}(u, S)$;
- $\max_{u \in V} \{\text{delay}(u, V(G_u))\}$ is minimized among all such possible sets Π of connected subgraphs.

Note that in such a set Π as specified above, G_u is not necessarily different with G_v when $u \neq v$. For instance, in Figure 1 the 4 subgraphs (depicted by polylines), right-upper corner, right-bottom corner, left-upper corner, and left-bottom corner form a coterie for the graph in the figure. In the example, if each edge takes the weight 1, each circle node uses the right-upper corner subgraph as a quorum, each hexagon node uses the right-bottom corner subgraph as a quorum, each square node uses the left-upper corner subgraph as a quorum, and each triangle node uses the left-bottom corner subgraph as a quorum, then the coterie is the optimal solution for MMD.

Minimizing Total-Delays (MTD)

INSTANCE: A weighted graph $G = (V, E)$, $\forall (u, v) \in E$ $d_{u,v}$ is an integer, and $\forall u \in V$ w_u is an integer.

QUESTION: Find a set $\Pi = \{G_u : u \in V\}$ of connected subgraphs such that

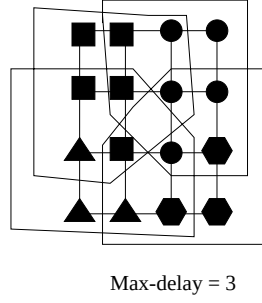


Fig. 1. Max-delays

- $S = \{V(G_u) : G_u \in \Pi\}$ is a coterie of G ;
- $\forall u \in V, \text{delay}(u, V(G_u)) = \text{delay}(u, S)$;
- $\text{tot-delay}(S)$ is minimized among all such possible sets Π of connected subgraphs.

Note that the problems of MMD and MTD were studied in [8]. However, our definition of MTD is more general than that in [8] which studied only MTD restricted to the case where each vertex takes a unit weight.

In the next two sections, we will present our results for MTD and MMD problems.

3 Minimizing Total-Delays

The section is organized below. It starts with an investigation of the complexity of the problem. Then we will present an approximate algorithm to solve the problem. It can be shown that the proposed algorithm has approximate ratio 2; and the bound is quite tight. Moreover, we can show that the algorithm can guarantee exact solutions for quite a large class of graphs.

3.1 Complexity of Minimizing Total-Delays

In this subsection, we prove the NP-hardness of MTD problem; this is done by proving the NP-completeness of the corresponding decision problem.

MTD decision (MTDD) problem

INSTANCE: A weighted graph $G = (V, E)$, $\forall (u, v) \in E$ $d_{u,v}$ is an integer, $\forall u \in V$ w_u is an integer, and an integer N .

QUESTION: Are there a set $\Pi = \{G_u : u \in V\}$ of connected subgraphs such that 1) $S = \{V(G_u) : G_u \in \Pi\}$ is a coterie of G ; 2) $\forall u \in V, \text{delay}(u, V(G_u)) = \text{delay}(u, S)$; and 3) $\text{tot-delay}(S) \leq N$.

Theorem 1. *MTDD is NP-complete.*

To prove Theorem 1, below we transform the vertex cover problem to a special case of MTDD.

Vertex Cover

INSTANCE: A graph $G = (V, E)$, positive integer $K \leq |V|$.

QUESTION: Is there a vertex cover of size K or less for G ?

For each instance $I_{VC} = \{G, K\}$ in the vertex cover, we construct the corresponding instance I_{VC}^{MTDD} of MTDD, say $I_{VC}^{MTDD} = \{G', N'\}$, as follows.

- $V(G) \subset V(G')$, and the vertices in $V(G') - V(G)$ are added along with the new edges created below.
- In G' , replace each edge (u, v) in G by 4 edges (u, u') , (u', v) , (u, v') , (v', v) such that $d_{u,u'} = 1.5$, $d_{u',v} = 1$, $d_{u,v'} = 1$, and $d_{v',v} = 1.5$.
- For each pair $\{u, v\}$ of vertices in G such that $(u, v) \notin E(G)$, add two edges (u, u') and (u', v) to G' such that $d_{u,u'} = 1$ and $d_{u',v} = 1$.
- For each pair $\{u, v\}$ of vertices in $V(G') - V(G)$, G' has one edge (u, v) connected and $d_{u,v} = 1$. Note that the induced graph on $V(G') - V(G)$ (with respect to G') is thus fully connected.
- In G' , for each vertex $u \in V(G)$ ($V(G) \subset V(G')$) let $w_u = 2n^4$ where n is the number of vertices in G ; and for each vertex $u \in V(G') - V(G)$ let $w_u = 1$.
- Let $N' = 2n^5 + Kn^4 + n^2$.

An example of such a transformation is depicted in Figure 2(a) and (b), where Figure 2 (a) gives an instance in the vertex cover, the square vertices in Figure 2(b) represent the newly added vertices, the dotted edges carry the weight 1, and the square vertices are connected to form a complete subgraph.

Proof Sketch of Theorem 1: Clearly, the above transformation is polynomial with respect to n . It can also be shown in the full paper [15] that for each instance I_{VC} in the vertex cover problem, it has a solution if and only if the corresponding instance I_{VC}^{MTDD} has a solution. $\square$

Further, we can have a stronger version of Theorem 1.

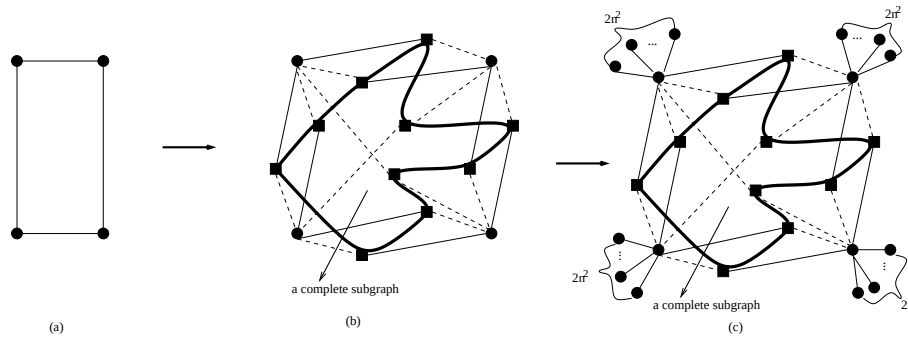


Fig. 2. A transformation from vertex cover to MTDD

Corollary 1. *MTDD is NP-complete even restricted to the case when each vertex has a unit weight.*

Proof: To prove the corollary, we do the following modification on the instance $I_{VC}^{MTDD} = (G', N')$ constructed as above from I_{VC} .

- *Modification of G' .* For each vertex u in G' with weight $2n^4$, add $2n^4$ new vertices which are connected to u as a star, where those new added edges have weight 1. Figure 2(c) illustrates such a modification from the graph as depicted in Figure 2(b) regarding the instance (Figure 2(a)) of the vertex cover.
- *Modification of N' .* We also modify N' from $2n^5 + Kn^4 + n^2$ to $4n^5 + Kn^4 + n^2$.

Similar arguments to those in the proof of Theorem 1 [15] can lead to a proof of this corollary. $\square$

3.2 An Approximate Algorithm

In this subsection, we present a heuristic for MTD. The algorithm consists of three steps as follows. Suppose that a weighted network G is given.

Algorithm MTD

Step 1 Compute all pair of shortest paths $\{L_{u,v} : \forall u, v \in V(G)\}$.

Step 2 Choose a vertex u^* such that $\sum_{v \in G} w_v \times l_{u^*,v}$ is minimized (note that $l_{u^*,v}$ denotes the length of the shortest path $L_{u^*,v}$).

Step 3 $\forall v \in V(G)$, let $G_v = \{u^*\}$.

Note that the algorithm MTD gives a coterie consisting of only one vertex $\{u^*\}$. The algorithm can be easily implemented in time $O(n^3)$. This is because the step 1 can be implemented in $O(n^3)$ time by the algorithm Floyd-Warshall [7], while step 2 and step 3 can be trivially implemented in $O(n^2)$ time.

Next we show the approximation behaviour of the algorithm.

Theorem 2. *The approximation ratio of the algorithm MTD is not greater than 2.*

Sketch of the Proof: For each pair of vertices v_1 and v_2 , let l_1 and l_2 respectively denote the delays of v_1 and v_2 for a given coterie S . Then, one can immediately verify $l_{u_1, u_2} \leq l_1 + l_2 \leq \max\{l_1, l_2\}$, where l_{u_1, u_2} denotes the shortest path between u_1 and u_2 . This leads to a proof of the theorem; refer to the full paper [15] for the details. $\square$

We can show that the ratio, obtained for our algorithm, is fairly tight by the following example. As depicted in Figure 3, a class of graphs $G = (V, E)$ have the following structure:

- $V = V_1 \cup V_2$ where V_1 has n vertices (circle vertices in V_1) and V_2 (square vertices in V_2) has $\frac{n(n-1)}{2}$ vertices;

- each pair of vertices in V_1 correspond to a vertex in V_2 , and are connected by two edges via the vertex in V_2 ;
- the induced graph on V_2 is complete;
- each edge has a unit weight;
- the weight of a vertex in V_1 is $N = n^k$ (k is sufficiently large) and the weight of a vertex in V_2 is 1.

It can be immediately shown that the Algorithm MTD will randomly choose a square vertex as the coterie; and it produces the total delay is $T_{MTD} = 2(n - 1)N + \frac{n(n-1)}{2} - 1$. On the other hand, for this class of graphs the minimal total delay will be the coterie where the delay of a quorum for each vertex is 1; and thus $T_{OPT} = Nn + \frac{n(n-1)}{2}$. Therefore,

$$\lim_{n \rightarrow \infty} \frac{T_{MTD}}{T_{OPT}} = 2 \quad (4)$$

This means that the approximation bound in Theorem 2 is asymptotically tight. Figure 3 illustrates such a graph when $n = 4$. Moreover, the asymptotically lower bound illustrated in (4) also holds even restricted to the case when each vertex has a unit weight; this can be shown by adding N vertices “star-like” attached to each vertex with weight N in the way as depicted in Figure 2(c) and the new edges all take a very small weight ϵ .

Furthermore, we can obtain a tighter expression of the approximation ratio restricted to the case when all vertices have a unit weight. The proof Theorem 3 adopts a similar idea to that of Theorem 2; and we also leave it to the full paper [15].

Theorem 3. *Algorithm MTD has approximation ratio $2 - \frac{4}{n}$ if each vertex is equally weighted and $n \geq 4$. Here, n is the number of vertices in the network.*

Next we show that the algorithm MTD can guarantee the optimal solutions for quite a large class of graphs, including rings, meshes, and trees.

Theorem 4. *Suppose that G is a weighted graph with n vertices, where*

1. *Each vertex in G is equally weighted.*
2. *The vertex set V can be divided into m ($n/2$ or $(n-1)/2$) disjoint pairs of vertices:*

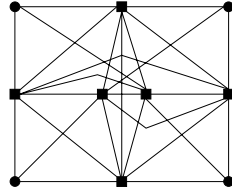


Fig. 3. To shown the tightness of the ratio

- if $n = |V|$ is even, $V = \{u_i, v_i : 1 \leq i \leq \frac{n}{2}\}$, and
- if $n = |V|$ is odd, $V = \{u_i, v_i : 1 \leq i \leq \frac{n-1}{2}\} \cup \{z_0\}$,

such that a shortest path between each pair of v_i and u_i shares the common vertex; and the common vertex is z_0 if n is odd. Then the algorithm MTD will produce the optimal solution for G .

Proof: Let m denote $\frac{n}{2}$ when n is even, and denote $\frac{n-1}{2}$ when n is odd. Let l_i denote the length of the shortest path between v_i and u_i for $1 \leq i \leq m$.

Suppose that $S = \{V_v : v \in V\}$ is a coterie that leads to the optimal solution for MTD, where for each $v \in V$, $\text{delay}(v, V_v) = \text{delay}(v, S)$. We have that for $1 \leq i \leq m$ $\text{delay}(u_i, V_{u_i}) + \text{delay}(v_i, V_{v_i}) \geq l_i$; and thus $T_{OPT} \geq \sum_{i=1}^m l_i$.

On the other hand, if we choose this common vertex v (it is z_0 if n is odd) of G as a coterie, the total delay is $\sum_{i=1}^m (l_{v_i, v} + l_{u_i, v})$. Therefore, $T_{MTD} \leq \sum_{i=1}^m (l_{v_i, v} + l_{u_i, v})$.

Note that $l_{v_i, v} + l_{u_i, v} = l_i$. This implies that $T_{MTD} \leq \sum_{i=1}^m l_i$. Consequently, $T_{MTD} \leq T_{OPT}$. Thus, $T_{MTD} = T_{OPT}$. $\square$

Theorem 4 implies that for those popular network topologies [18], such as *rings* (see Figure 4(a) for an example), *meshes* (Figure 4(b)), and *trees* (Figure 4(c)), the algorithm MTD can produce the optimal solution for MTD when each vertex takes the same weight and each edge also takes the same weight. Note that for rings where each edge and vertex take a unit weight, the algorithm in [8] can guarantee only the approximation ratio 1.25.

4 Minimizing Maximal-Delays

In this section, we investigate the problem of minimizing maximum-delays. Particularly, we present an efficient algorithm based on a mixture of dynamic programming and greedy paradigms. The algorithm conceptually consists of three steps as follows. Suppose that $G = (V, E)$ is the network.

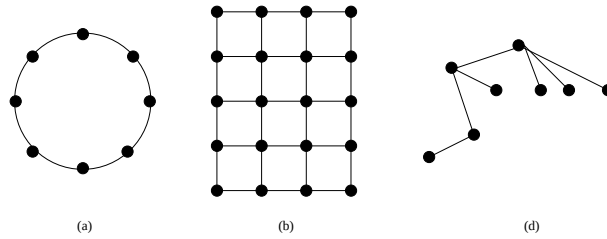


Fig. 4. Rings, meshes, and trees

Algorithm MMD

- Step 1** In G , compute all pair shortest paths $\{L_{u,v} : u, v \in V\}$ and their lengths $\{l_{u,v} : u, v \in V\}$.
- Step 2** Sort $\{l_{u,v} : u, v \in V\}$ in an increasing order and store them in D .
- Step 3** Scan D according to the increasing ordering of $l_{u,v}$, iteratively construct subgraphs by an expansion till each pair of subgraphs have an intersection. Then check non-redundancy and remove all redundant quorums if exist.

As mentioned in last section, applying the algorithm Floyd-Warshall in [7], the step 1 can be implemented in $O(n^3)$ time where n is the number of vertices. Clearly, Step 2 can be implemented in $O(n^2 \log n)$ time as there are n^2 pairs of vertices. In fact, step 3 can also be implemented in $O(n^3)$ time; and it is described below.

In our implementation of Step 3, for each vertex v_k in G we use a linear data structure (for instance, a linked list) A_k to store the vertices w such that $l_{v_k,w}$ is less than the current element $l_{u,v}$ of D . Clearly, each induced [7] subgraph G_k by the vertices in A_k is connected, and $\text{delay}(v_k, A_k) \leq l_{u,v}$.

Moreover, suppose that A_k is constructed up to $l_{u,v}$ in D . Then A_k also gives us the information that for any pair $\{v_i, v_j\}$ of vertices and any two quorums Q_i and Q_j such that $\text{delay}(v_i, Q_i) \leq d_{u,v}$ and $\text{delay}(v_j, Q_j) \leq l_{u,v}$, Q_i intersects with Q_j at v_k if and only if v_i and v_j are both in A_k . Further, for each pair $\{v_i, v_j\}$ we use $B_{i,j}$ ($i < j$) to indicate whether or not the current A_i and A_j already intersect. As mentioned earlier, Step 3 consists of two sub-steps:

Step 3a: construct a vertex set A_i for each vertex v_i , and then

Step 3b: check non-redundancy: if $A_i \subset A_j$ then replace A_j by A_i .

Below is the detailed implementation of Step 3a.

Step 3a

Set each A_k ($1 \leq k \leq n$) to empty; and set each $B_{i,j}$ ($1 \leq i < j \leq n$) to zero; $m \leftarrow 0$;

for each l_{v_i, v_j} **do**: (Scan D according to the increasing ordering of its elements)
 $\{ A_i \leftarrow A_i \cup \{v_j\}; A_j \leftarrow A_j \cup \{v_i\};$
 for each vertex v_x in $A_i - \{v_j\}$ **do**
 if $B_{\min\{x,j\}, \max\{x,j\}} = 0$ **then**
 $\{ B_{\min\{x,j\}, \max\{x,j\}} \leftarrow 1; m \leftarrow m + 1;$
 if $m = \frac{n(n-1)}{2}$ **then** terminate the algorithm; }
 for each vertex v_x in $A_j - \{v_i\}$ **do**
 if $B_{\min\{x,i\}, \max\{x,i\}} = 0$ **then**
 $\{ B_{\min\{x,i\}, \max\{x,i\}} \leftarrow 1; m \leftarrow m + 1;$
 if $m = \frac{n(n-1)}{2}$ **then** terminate the algorithm; } $\square$

It can be immediately shown that the algorithm runs in time $O(n^3)$. To implement Step 3b efficiently, we first sort all A_i in an increasing order according to the cardinalities of each A_i . Then we test the set inclusion property for A_i and A_j according to the ordering where $|A_i| < |A_j|$. In fact, the test of the set

inclusion property for a pair A_i and A_j can be done by $O(\min\{|A_i|, |A_j|\})$ if all vertices A_i and A_j are stored in a global ordering applied to each A_k for $1 \leq k \leq n$. Therefore, Step 3b can also be implemented in $O(n^3)$. These imply that the algorithm MMD can be implemented in $O(n^3)$ time.

We can also immediately show that the algorithm MMD guarantees providing an exact solution for MMD problem; this is because we use an iteration procedure and the algorithm immediately terminates once each pair of quorums have an intersection.

5 Conclusion and Remarks

In this paper, we investigated the complexity issues of minimizing the max-delay (MMD) and minimizing the total-delay (MTD) for designing coteries (quorum consensus methods). Firstly, we showed that MTD is NP-hard even restricted to the case where each vertex takes a unit weight. Then we proposed a cubic approximate algorithm for solving MTD. The approximate algorithm has an approximation ratio 2 in general, and has an approximation ratio $2 - \frac{n}{4}$ for the case where each vertex takes a unit weight. We also showed that our approximation ratios are asymptotically tight. Moreover, we showed that the approximate algorithm can guarantee exact solutions for a quite large class of graphs such as “uniform” rings, uniform trees, and uniform meshes. Secondly, in the paper we presented a cubic algorithm to solve the problem MMD.

Note that for presentation simplicity, the results presented in the paper follow the assumption, made in [8], that each data object is fully replicated over the network. In case if a data object is partially replicated in the network, the design of coteries must be restricted to the vertices that hold the replicas. In this case, our results still hold if we add this restriction to our algorithms when generate quorums. However, we have to drop the connectivity property from a coterie; this is because that those sites holding replicas may not necessarily be connected.

As a possible future study, we are interested in MTD for non-uniform rings and meshes, that is, links and vertices do not necessarily take unit weight. Further, we are interested in MTD for design of “wr”-coteries [8,11,20].

Acknowledgement

Part of the work was done when the author took study leave in SEEM Department at the Chinese University of Hong Kong (CUHK). The author would also like to thank Prof. S. Zhang from CUHK for useful discussions.

References

1. B. Awerbuch, Y. Bartal, and A. Fiat, Distributed paging for general networks, *Journal of Algorithms*, 28:67-104, 1998. 576
2. P. Bernstein, V. Hadzilocs and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.

3. Y. Breitbart, R. Komondoor, R. Rastogi, and S. Seshadri, Update Propagation Protocols for Replicated Databases, *Proceedings of the 1999 ACM SIGMOD*, 97 - 108, 1999. 575
4. Y. Breitbart and H. F. Korth, Replication and Consistency: Being lazy helps sometimes, *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson, Arizona, 1997. 575
5. H. Chen and J. Tang, An Efficient Method for Mutual Exclusion in Truly Distributed Systems, *Int. Conf. on Distributed Computing Systems*, 97-104, 1994. 576
6. P. Chundi, D. J. Rosenkratz, and S. S. Ravi, Deferred Updates and Data Placement in Distributed Databases, *Proceedings of the Twelveth International Conference on Data Engineering*, New Orleans, Louisiana, 1996. 575
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT press, 1990. 577, 581, 584
8. A. W. Fu, Delay-Optimal Quorum Consensus for Distributed Systems, *IEEE Transactions on Parallel and Distributed Systems*, 8(1), 59-69, 1997. 576, 579, 583, 585
9. H. Garcia-Molina and D. Barbara, How to Assign Votes in a Distributed Systems, *J. ACM*, 32(4), 841-860, 1985. 576, 577
10. J. Gray, P. Helland, P. O'Neil, and Dennis Shasha, The Dangers of Replication and a Solution, *Proceedings of the ACM SIGMOD Conference*, Montreal, Quebec, Canada, 1996. 575
11. T. Ibaraki, H. Nagamochi, and T. Kameda, Optimal Coteries for Rings and Related Networks, *Distributed Computing*, 8, 191-201, 1995. 576, 577, 585
12. A. Kumar, Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data, *IEEE Transactions on Computers*, 40(9), 996-1004, 1991. 576
13. A. Kumar and A. Segev, Cost and Availability Tradeoffs in Replicated Data Concurrency Control, *ACM Transactions on Database Systems*, 18(1), 102-131, 1993. 576
14. X. Lin, A Fully Distributed Quorum Consensus Method with High Fault-Tolerance and Low Communication Overhead, *Theoretical Computer Science*, 185(2), 259-275, 1997. 576
15. X. Lin, Delay Optimizations in Quorum Consensus, *full paper*, 2001. (<http://www.cse.unsw.edu.au/~lxue>.) 577, 580, 581, 582
16. X. Lin and M. E. Orlowska, An Efficient Optimal Algorithm for Minimizing the Overall Communication Cost in Replicated Data Management, *J. Parallel and Distributed Comput.* 35, 97-103, 1996. 576
17. M. Maekawa, A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Transactions on Computer Systems*, 3(2), 145-159, 1985. 576
18. M. T. Ozsü and P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, 1991. 575, 576, 583
19. B. Maggs, F. Mayer auf der Heide, B. Vocking, and M. Westermann, Exploiting Locality for Data Management in Systems of Limited Bandwidth, *Proc. of the 38th IEEE Symp. on Foundations of Computer Science (FOCS)*, 284-293, 1997. 576
20. C. H. Papadimitriou and M. Sideri, Optimal Coteries, *Proceedings of ACM 10th Symposium of Distributed Computing*, 75-80, 1991. 576, 577, 585
21. D. Saha, S. Rangarajan, and S. K. Tripathi, An Analysis of the Average Message Overhead in Replica Control Protocols, *IEEE Transactions on Parallel and Distributed Systems*, 7(10), 1026 - 1034, 1995. 576

Randomized Shared Queues Applied to Distributed Optimization Algorithms

Hyunyoung Lee and Jennifer L. Welch

Department of Computer Science, Texas A&M University
College Station, TX 77843-3112, U.S.A.
{hlee,welch}@cs.tamu.edu

Abstract. This paper presents a specification of a randomized shared queue that can lose some elements or return them out of order, and shows that the specification can be implemented with the probabilistic quorum algorithm of [5,6]. Distributed algorithms that incorporate the producer-consumer style of interprocess communication are candidate applications for using random shared queues in lieu of the message queues. The modified algorithms will inherit positive attributes concerning load and availability from the underlying queue implementation. The behavior of a generic combinatorial optimization algorithm, when it is implemented using random queues, is analyzed.

1 Introduction

Quorum systems have been receiving significant attention because they provide consistency and availability of replicated data and reduce the communication bottleneck of some distributed algorithms (cf. [6] for references). The probabilistic quorum model [6] relaxes the intersection property of strict quorum systems, such that pairs of quorums only need to intersect with high probability. In earlier work [4], we showed that probabilistic quorums implement *random registers*, memory cells from which out-of-date values are sometimes read. Such an implementation inherits the positive load and availability properties of probabilistic quorums. Random registers were shown to be strong enough to implement an interesting class of iterative algorithms that converge with high probability.

In this paper, we extend the results of [4], which considers only read-write registers, to one of the fundamental abstract data structures: the queue. We propose a specification of a randomized shared queue data structure (*random queue*) that can exhibit certain errors — namely the loss of enqueued values — with some small probability. The random queue preserves the order in which individual processes enqueue, but makes no attempt to provide ordering across enqueueers. We show that this kind of random queue can be implemented with the probabilistic quorum algorithm of [5,6].

Queues are a fundamental concept in many areas of computer science. A common application in distributed computing are message queues in communication networks. Many distributed algorithms use high-level communication

operations, such as scattering or all-to-all broadcasts (cf. Chapter 1 of [2] for an overview). These algorithms can typically tolerate inaccuracies in the order in which the queue returns its elements, as the order of the elements in the message queue is typically impacted by the unpredictability of the communications network. Furthermore, we consider randomized algorithms, in which the queue elements contain data that can be incorrect or otherwise inappropriate with some probability. Algorithms of this type can typically tolerate the random disappearance of elements in the queue (with some small probability). We believe that this constitutes a large class of algorithms, which can take advantage of random queues and their benefits of optimal load and high availability. As an example of applications from this class, we analyze the behavior of a class of optimization algorithms [1], when used with random queues.

Randomization is used in [10,11] to implement a *task queue*, an unordered collection of tasks with priorities which are used for load balancing in irregular applications; in these papers, the randomization affects only the priorities, while the number of enqueued tasks is preserved. In [3], randomized distributed queues are shown to have improved performance but no random behavior of the queue operations is specified.

2 Definitions

In this section, we define the system model. (The presentation of this material is similar to that in [4].)

The data type of a shared object is defined by a set of operations and set of allowable sequences of those operations. An **operation** consists of an **invocation** and a matching **response**. The invocation indicates the specific object and contains any inputs, while the response also indicates the relevant object and contains any outputs. Throughout this paper, we assume that each process has at most one operation pending at a time.

A **process** is a (possibly infinite) state machine which has access to a random number generator. The process has a distinguished state called the **initial state**.

We assume a system consisting of a collection of n **client** processes and r **server** processes. A client process runs on a processor that also runs an application process which is part of a distributed application that is written assuming shared data objects. The client process communicates with the shared memory application process above it and with the message passing system below it. A server process stores replicated data and interacts with client processes through the message passing system. We will restrict attention to algorithms (such as ours) in which only client processes use randomization; trivial extensions to the model would allow servers also to be randomized.

There is some set of **triggers** that can take place in the system. Triggers consist of operation invocations and message receptions. The occurrence of a trigger at a process causes the process to take a **step**. During the step, the process applies its transition function to its current state, the particular trigger, and a random number to generate a new state and some **outputs**. The outputs

can include (at most) one operation response and a set of messages to be sent. A step is completely described by the current state, the trigger, the random number, the new state, and the set of outputs.

There are three potential sources of nondeterminism in the system from the viewpoint of the shared object implementation: the sequences of random numbers available to the client processes (due to the random number generators), the sequences in which operation invocations are made on the client processes (due to the application program that is using the shared object layer), and variability in the message delays. We abstract the last two sources of nondeterminism into a construct called an “adversary.” Formally, an **adversary** is a partial function from the set of all sequences of steps to the set of triggers. That is, given a sequence of steps that have occurred so far, the adversary determines what trigger will happen next. Note that the adversary *cannot* influence what random number is received in the next step, only the trigger. Let RAND be the set of all n -tuples of the form $\langle R^1, \dots, R^n \rangle$ where each R^i is an infinite sequence of integers in $\{0, \dots, D\}$. D indicates the range of the random numbers. R^i describes the sequence of random numbers available to client process i in an execution — R_j^i is the random number available at step j . Call each element in RAND a **random tuple**.

Given an adversary A and a random tuple $\mathcal{R} = \langle R^1, \dots, R^n \rangle$, define an **execution** $\text{exec}(A, \mathcal{R})$ to be the sequence of steps $\sigma_1 \sigma_2 \dots$ such that:

- the current state in the first step of each process (client and server) i is i ’s initial state;
- the current state in the j -th step of process i is the same as the new state in the $(j - 1)$ -st step of i , for all processes i and all $j > 1$;
- the trigger in σ_j equals $A(\sigma_1 \dots \sigma_{j-1})$, for all $j \geq 1$ (the trigger is chosen by the adversary);
- the random number in σ_j equals R_j^i , where i is the process in σ_j ’s trigger (the random number comes from $\mathcal{R}$, not the adversary).

We put the following restrictions on the adversary:

- (Application related) The sequence of operation invocations at each process is consistent with the application layer above. That is, the operation invocations reflect the shared memory accesses of the application.
- (Message passing related) Every message received was previously sent and every message sent is eventually delivered exactly once. That is, the message passing system is asynchronous and reliable, with the exact delays under the control of the adversary.

An execution e is **complete** if either it is infinite or $A(e)$ is undefined. In a finite complete execution, the application is through making calls on the shared objects and no messages are in transit.

3 A Random Queue

In this section, we specify a randomized shared queue and propose an implementation for it. We then analyze the behavior of the implementation.

3.1 Specification of Random Queue

A **queue** Q shared by several processes supports two operations, $\text{Enq}(Q, v)$ and $\text{Deq}(Q, v)$. $\text{Enq}_i(Q, v)$ is the invocation by process i to enqueue the value v , $\text{Ack}_i(Q)$ is the response to i 's enqueue invocation, $\text{Deq}_i(Q, v)$ is the invocation by i of a dequeue operation, and $\text{Ret}_i(Q, v)$ is the response to i 's dequeue invocation which returns the value v . A possible return value is also $\perp$, indicating an empty queue. The set of values from which v is drawn is unconstrained. We will focus on *multi-enqueuer*, *single-dequeuer* queues; thus, the enqueue can be invoked by all the processes while the dequeue can be invoked only by one process. We assume for notational simplicity that, in every execution, every enqueued value is uniquely identified.

Given a real number p that is between 0 and 1, a system is said to implement a **p -random queue** if the following conditions hold for every adversary A . In every complete execution (of the adversary),

- (Liveness) every operation invocation has a following matching response;
- (Integrity) every operation response has a preceding matching invocation;
- (No Duplicates) for each value x , $\text{Deq}(Q, x)$ occurs at most once;
- (Per Process Ordering) for all i , if $\text{Enq}_i(Q, x_1)$ ends before $\text{Enq}_i(Q, x_2)$ begins, then x_2 is not dequeued before x_1 is dequeued.

(Probabilistic No Loss) For every enqueued value x , $\Pr[x \text{ is dequeued}] \geq p$.

That is, each enqueued element is either never dequeued (with probability at most $1 - p$) or is dequeued once (with probability at least p). For a given adversary, the probability space is all extensions (of that adversary) of any finite execution of the adversary that ends with the invocation to enqueue x .

3.2 Implementation of Random Queue

We now describe an implementation of a p -random queue. The next subsection computes the value of p , assuming that the application program using the shared queue satisfies certain properties.

The random queue algorithm (Algorithm 1) is based on the probabilistic quorum algorithm of Malkhi et al. [6]. There are r replicated memory servers. First, we describe the algorithm for the special case of a single enqueuer. The case of multiple enqueueers is explained later.

The enqueue operation (Enq) mirrors the probabilistic quorum write operation: The local timestamp is incremented by one and attached to the element that is to be enqueued. The resulting pair is sent to the replicas in the chosen quorum, a randomly chosen group of k servers.

The key notion in the dequeue operation (SingleDeq) is a timestamp limit (T). At any given time, all timestamps that are smaller than the current value T are considered to be outdated. T is included in the dequeue messages to the replica servers and allows them to discard all outdated values. Beyond this, SingleDeq mirrors the probabilistic quorum read operation: The client selects a

Algorithm for client process – for single enqueueer and single dequeuer:

Initially local variable $t = 0$ // enqueue timestamp

$T = 1$ // expected dequeue timestamp

when $\text{Enq}(Q, v)$ occurs:

$t := t + 1$

send $\langle \text{enq}, v, t \rangle$ to a randomly chosen quorum of size k and wait for acks

$\text{Ack}(Q)$ // response to application

when $\text{SingleDeq}(Q)$ occurs:

send $\langle \text{deq}, T \rangle$ to a randomly chosen quorum of size k and wait for replies

choose value v with smallest timestamp t_d

($\perp$ is considered to have largest timestamp)

if v is not $\perp$ then $T := t_d + 1$

$\text{Ret}(Q, v)$ // response to application

Algorithm for server process i , $1 \leq i \leq r$:

Initially local variable Q_{copy} , a queue, is empty

when $\langle \text{enq}, v, T \rangle$ is received from client j :

enqueue (v, T) to Q_{copy}

send $\langle \text{ack} \rangle$ to client j

when $\langle \text{deq}, T \rangle$ is received from client j :

remove (dequeue) every element of Q_{copy} whose timestamp smaller than T

if Q_{copy} is empty let $w = \perp$

otherwise let w be the result of dequeue on Q_{copy}

send $\langle w \rangle$ to client j

Algorithm for a dequeuer extension for $n > 1$ enqueueers:

Initially local variable $i = 0$, shared queue $Q = (Q_1, \dots, Q_n)$

// an array of n single enqueueer queues

when $\text{Deq}(Q)$ occurs

$i := (i \bmod n) + 1$

$\text{SingleDeq}(Q_i, v)$ // v is value returned by SingleDeq

$\text{Ret}(Q, v)$ // response to application

Algorithm 1: Implementation of p -random queue Q

random quorum, sends dequeue messages to all replica servers in the quorum and selects the response with the smallest timestamp t_d . It updates the timestamp limit to $T := t_d + 1$ and returns the element that corresponds to t_d .

Each replica server implements a conventional queue with access operations enqueue and dequeue. In addition, the dequeue operation receives the current timestamp limit as input and discards all outdated values. The purpose of this is to ensure that there are exactly k replica servers that will return the element v_T with timestamp T in response to a dequeue request. Thus, the probability of

finding this element (in the current dequeue operation) is exactly the probability that two quorums intersect. This property is of critical importance in the analysis in the following section. It does not hold if outdated values are allowed to remain in the replica queues, as those values could be returned instead of v_T by some of the replica servers containing v_T .

For the case of $n > 1$ enqueueers, we extend the single-enqueueer, single-dequeueer queue by having n single-enqueueer queues $(Q_1, \dots, Q_n)$, one per enqueueer. The i -th enqueueer ($1 \leq i \leq n$) enqueues to Q_i . The single dequeueer dequeues from all n queues by making calls to the function $\text{Deq}()$, which selects one of the queues and tries to dequeue from it. $\text{Deq}()$ checks the next queue in sequence. The round-robin sequence used in Algorithm 1 can be replaced by any other queue selection criterion that queries all queues with approximately the same frequency. The selection criterion will impact the order in which elements from the different queues are returned. However, it does not impact the probability of any given element being dequeued (eventually), as the queues do not affect each other, and the attempt to dequeue from an empty queue does not change its state.

3.3 Analysis of Random Queue Implementation

For this analysis, we assume that the application program invoking the operations on the shared random queue satisfies a certain property. Every complete execution of every adversary consists of a sequence of segments. Each **segment** is a sequence of enqueues followed by a sequence of dequeues, which has at least as many dequeues as enqueues. Fix a segment. Let m_e , resp., m_d , be the total number of enqueue, resp., dequeue, operations in this segment. Let $m = m_e + m_d$. Let Y_i be the indicator random variable for the event that the i -th element is returned by a dequeue operation ($1 \leq i \leq m_e$). In the following lemma, the probability space is given by the enqueue and dequeue quorums which are selected by the queue access operations. More precisely, let $\mathcal{P}_k(r)$ denote the collection of all subsets of size k of the set $\{1, \dots, r\}$. Since there are m enqueue and dequeue operations, we let $\Omega = \mathcal{P}_k(r)^m$ be the universe. The probability space for the following lemma is given by Ω and the uniform distribution on Ω .

Lemma 1. *The random variables Y_i ($1 \leq i \leq m_e$) are mutually independent and identically distributed with $\Pr(Y_i = 1) = p = \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right)$.*

Proof. Since the queues $Q_1, \dots, Q_n$ do not interfere with each other, they can be considered in isolation. That is, it is sufficient to prove the lemma for any given single enqueueer queue Q_i . Consider any single enqueueer queue Q_z and let m_z denote the number of enqueued elements. In order to prove mutual independence, we have to show

$$\Pr\left(\bigwedge_{i=1}^{m_z} Y_i = a_i\right) = \prod_{i=1}^{m_z} \Pr(Y_i = a_i) \quad (1)$$

for all possible assignments of $\{0, 1\}$ -values to the constants a_i , for which the probability on the left-hand side is greater than zero. Thus, the following conditional probabilities are well-defined. For $h = 1$: trivially, $\Pr(\bigwedge_{i=1}^1 Y_i = a_i) = \prod_{i=1}^1 \Pr(Y_i = a_i)$. For all $1 < h \leq m_z$:

$$\Pr\left(\bigwedge_{i=1}^h Y_i = a_i\right) = \Pr(Y_h = a_h \mid \bigwedge_{i=1}^{h-1} Y_i = a_i) \cdot \Pr\left(\bigwedge_{i=1}^{h-1} Y_i = a_i\right). \quad (2)$$

Let $j = \max\{i < h : a_i = 1\}$ ¹. Clearly, the event $Y_h = 1$ does not depend on any event $Y_i = a_i$ for $i < j$. Thus

$$\Pr(Y_h = 1 \mid \bigwedge_{i=1}^{h-1} Y_i = a_i) = \Pr(Y_h = 1 \mid Y_j = 1 \wedge \bigwedge_{i=j+1}^{h-1} Y_i = 0).$$

The condition corresponds to the following case: The last dequeue operation has returned the j -th element. The dequeue operation immediately following the dequeue operation that dequeued j -th element misses elements $j + 1$ to $h - 1$. That is, the dequeue quorum R of the dequeue operation does not intersect the enqueue quorum S_i of any element $i \in \{j + 1, \dots, h - 1\}$. Thus

$$\begin{aligned} \Pr(Y_h = 1 \mid Y_j = 1 \wedge \bigwedge_{i=j+1}^{h-1} Y_i = 0) &= \Pr(R \cap S_h \neq \emptyset \mid \bigwedge_{i=j+1}^{h-1} R \cap S_i = \emptyset) \\ &= \Pr(R \cap S_h \neq \emptyset) = \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right) = p \end{aligned}$$

The second equality is because quorums are chosen independently. In summary, for all $1 < h \leq m_z$ and assignments of $\{0, 1\}$ to a_i ,

$$\Pr(Y_h = 1 \mid \bigwedge_{i=1}^{h-1} Y_i = a_i) = p.$$

By the formula of total probabilities, $\Pr(Y_h = 1) = p$. Thus, returning to (2):

$$\Pr\left(\bigwedge_{i=1}^h Y_i = a_i\right) = \Pr(Y_h = a_h) \Pr\left(\bigwedge_{i=1}^{h-1} Y_i = a_i\right).$$

Mutual independence (1) follows from this by induction.

Theorem 1. *Algorithm 1 implements a random queue.*

Proof. The Integrity and Liveness conditions are satisfied since the adversary cannot create or destroy messages. The No Duplicates and Per Process Ordering conditions are satisfied by the definition of the algorithm. The Probabilistic No Loss condition follows from Lemma 1, which states that each enqueued value is dequeued with probability $p = \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right)$.

¹ To handle the case when $a_i = 0$ for all $i < h$, define $Y_0 = a_0 = 1$.

4 Application of Random Queue: Go with the Winners

In this section we show how to incorporate random queues to implement a class of randomized optimization algorithms called Go with the Winners (GWTW), proposed by Aldous and Vazirani [1]. We analyze how the weaker consistency provided by random queues affects the success probability of GWTW. Our goal is to show that the success probability is not significantly reduced.

4.1 The Framework of GWTW

GWTW is a generic randomized optimization algorithm. A combinatorial optimization problem is given by a state space S (typically exponentially large) and an *objective function* f , which assigns a ‘quality’ value to each state. The task is to find a state $s \in S$, which maximizes (or minimizes) $f(s)$. It is often sufficient to find approximate solutions. For example, in the case of the clique problem, S can be the set of all cliques in a given graph and $f(s)$ can be the size of clique s .

In order to apply GWTW to an optimization problem, the state space has to be organized in the form of a tree or a DAG, such that the following conditions are met: (a) The single root is known. (b) Given a node s , it is easy to determine if s is a leaf node. (c) Given a node s , it is easy to find all child nodes of s . The parent-child relationship is entirely problem-dependent, given that $f(\text{child})$ is better than $f(\text{parent})$. For example, when applied to the clique problem on a graph G , there will be one node for each clique. The empty clique is the root. The child nodes of a clique s of size k are all the cliques of size $k + 1$ that contain s . Thus, the nodes at depth i are exactly the i -cliques. The resulting structure is a DAG. We can define a tree by considering ordered sequences of vertices.

Greedy algorithms, when formulated in the tree model, typically start at the root node and walk down the tree until they reach a leaf. The GWTW algorithm follows the same strategy, but tries to avoid leaf nodes with poor values of f , by doing several runs of the algorithm simultaneously, in order to bound the running time and boost the success probability (success means a node is found with a sufficiently good value of f). We call each of these runs a *particle* – which carries with it its current location in the tree and moves down the tree until it reaches a leaf node. The algorithm works in synchronous stages. During the k -th stage, the particles move from depth k to depth $k + 1$. Each particle in a non-leaf node is moved to a randomly chosen child node. Particles in leaf nodes are removed. To compensate for the removed particles, an appropriate number of copies of each of the remaining particles is added.

The main theme to achieve a certain constant probability of success is to try to keep the total number of particles at each stage close to the constant B .

The framework of the GWTW algorithms is as follows: *At stage 0, start with B particles at the root. Repeat the following procedure until all the particles are at leaves: At stage i , remove the particles at leaf nodes, and for each particle at a non-leaf node v , add at v a random number of particles, this random number having some specified distribution. Then, move each particle from its current position to a child chosen at random.*

Shared variables are random queues Q_i , $1 \leq i \leq n$, each dequeued by process i and initially empty
 Code for process i , $1 \leq i \leq n$:
 Local variable: integer s , initially 0.
 Initially $\frac{B}{n}$ particles are at the root.
 while true do
 $s++$
 for each particle at a non-leaf node v // clone the particles
 add at v a random number of particles, with some specified distribution
 endfor
 remove the particles at leaf nodes
 for each particle j // move j to some process x 's queue
 pick a random number $x \in \{1, \dots, n\}$
 Enq(Q_x, j)
 endfor
 while not all particles are dequeued // read from own queue
 Deq(Q_i, j)
 endwhile
 move each particle from its current position to a child chosen at random
endwhile

Algorithm 2: Distributed version of GWTW framework

We consider a distributed version of the GWTW framework (Algorithm 2), which is a modification from the parallel algorithm of [8]. Consider an execution of Algorithm 2 on n processes. At the beginning of the algorithm (stage 0), B particles are evenly distributed among the n processes. Since, at the end of each stage, some particles may be removed and some particles may be added, the processes need to communicate with each other to perform load balancing of the particles (global exchange). We use shared-memory communication among the processes. In particular, we use shared queues to distribute the particles among processes. Between enqueues and dequeues in Algorithm 2, we need some mechanism to recognize the total number of enqueued particles in a queue. It can be implemented by sending one-to-one messages among the processes or by having the maximum possible number of dequeues per stage. (Finding more efficient, yet probabilistically safe, ways to end a stage is work in progress.)

When using random queues, the errors will affect GWTW, since some particles disappear with some probability. However, we show that this does not affect the performance of the algorithms significantly. In particular, we estimate how the disappearance of particles caused by the random queue affects the success probability of GWTW.

4.2 Analysis of GWTW with Random Queues

We now show that Algorithm 2 when implemented with random queues will work as well as the original algorithms in [1].

We use the notation of [1] for the original GWTW algorithm (in which no particles are lost by random queues): Let X_v be a random variable denoting the number of particles at a given vertex v . Let S_i be the number of particles at the start of stage i . At stage 0, we start with B particles. Then $S_0 = B$ and $S_i = \sum_{v \in V_\ell} X_v$, for $i > 0$, where V_ℓ is the set of all vertices at depth ℓ . Let $p(v)$ be the chance the particle visits vertex v . Then $a(j) = \sum_{v \in V_j} p(v)$ is the chance the particle reaches depth j at least. $p(w|v)$ is defined to be the chance the particle visits vertex w conditioning on it visits vertex v . The values $s_i, 1 \leq i < \ell$ are constants which govern the particle reproduction rate of GWTWs. The parameter κ is defined to express the “imbalance” of the tree as follows: For $i < j$, $\kappa_{ij} = \frac{a(i)}{a^2(j)} \sum_{v \in V_i} p(v) a^2(j|v)$, and $\kappa = \max_{0 \leq i < j \leq d'} \kappa_{ij}$.

Aldous and Vazirani [1] prove

Lemma 2.

$$\mathbf{E}S_i = B \frac{a(i)}{s_i}, \quad 0 \leq i \leq d, \quad \text{and} \quad \mathbf{var}S_i \leq \kappa B \frac{a^2(i)}{s_i^2} \sum_{j=0}^i \frac{s_j}{a(j)}, \quad 0 \leq i \leq d.$$

We will use this lemma to prove similar bounds for the distributed version of the algorithm, in which errors in the queues can affect particles. For this purpose, we formulate the effect of the random queues in the GWTW framework.

More precisely, given any original GWTW tree T , we define a modified tree T' , which accounts for the effect of the random queues. Given a GWTW tree T , let T' be defined as follows: For every vertex in T , there is a vertex in T' . For every edge in T , there is a corresponding edge in T' . In addition to the basic tree structure of T , each non-leaf node v of T has an additional child w in T' . This child w is a leaf node. The purpose of the additional leaf nodes is to account for the probability with which particles can disappear in the random queues in Algorithm 2.

Given any node w in T' (which is not the root) and its parent v , let $p'(w|v)$ denote the probability of moving to w conditional on being in v . For the additional leaf nodes w in T' , we set $p'(w|v) = 1 - p$, where $1 - p$ is the probability that a given particle is lost in the queue. For all other pairs (w, v) , let $p'(w|v) = p \cdot p(w|v)$. Then $a'(i)$, $a'(i|v)$, S'_i , s'_i , X'_v , and κ' can be defined similarly for T' .

Given a vertex v of T , let $\bar{p}(v)$ denote the probability that Algorithm 2, when run with a single particle and without reproduction, reaches vertex v . The term “without reproduction” means that the distribution mentioned in the first “for” loop of the algorithm is such that the number of added particles is always zero. The main property of the construction of T' is:

Fact 1 For any vertex v of the original tree T , $p'(v) = \bar{p}(v)$. Furthermore,

$$\Pr(\text{Algorithm 2 reaches depth } \ell) = p \cdot \Pr(\text{GWTW on } T' \text{ reaches depth } \ell)$$

for any $\ell \geq 0$.

Proof. We prove the first statement by induction on the depth of v . At depth $d = 0$ (base case), v is the root and $p'(v) = \bar{p}(v) = 1$. For the inductive step, let $v \in V_{\ell+1}$ for $\ell \in \mathbb{N}$. Let $u \in V_\ell$ be the immediate ancestor of v . Now,

$$p'(v) = p'(v|u)p'(u) = p \cdot p(v|u)p'(u) = p \cdot p(v|u)\bar{p}(u) = \bar{p}(v|u)\bar{p}(u) = \bar{p}(v).$$

For the second statement, it is sufficient to note that

$$\Pr(\text{Algorithm 2 reaches depth } \ell) = \sum_{v \in V_\ell} \bar{p}(v) = \sum_{v \in V_\ell} p'(v) = p \cdot \sum_{v \in V'_\ell} p'(v).$$

We can now analyze the success probability of Algorithm 2 (a combination of GWTW and random queues) by means of analyzing the success probability of baseline GWTW on a slightly modified tree. This allows us to use the results of [1] in our analysis. In particular,

Lemma 3.

$$\mathbf{E}S'_i = B' \frac{p^{i-1}a(i)}{s'_i} \text{ and } \mathbf{var}S'_i \leq \frac{1}{p} \kappa B' \frac{p^{i-1}a^2(i)}{s_i'^2} \sum_{j=0}^i \frac{s'_j}{p^{j-1}a(j)}, 0 \leq i \leq d$$

Proof. We apply Lemma 2 to the GWTW process on T' and show that $\kappa' = \kappa/p$ and $a'(i) = p^{i-1}a(i)$ for all i . Note that for any $i \leq \ell$ and $v \in V_i$, $p'(v) = p(v)p^i$. Thus, for any $1 < i \leq \ell$

$$\begin{aligned} a'(i) &= \sum_{w \in V'_i} p'(w) = \sum_{w \in V'_i} \sum_{v \in V_{i-1}} p'(w|v)p'(v) \\ &= \sum_{v \in V_{i-1}} p'(v) \sum_{w \in V_i} p(w|v) = p^{i-1} \sum_{v \in V_{i-1}} p(v) \sum_{w \in V_i} p(w|v) = p^{i-1}a(i) \end{aligned}$$

For any $0 \leq i < j \leq \ell$,

$$\begin{aligned} \kappa'_{ij} &= \frac{a'(i)}{a'^2(j)} \sum_{v \in V'_i} p'(v) a'^2(j|v) \\ &= \frac{p^{i-1}a(i)}{p^{2j-2}a^2(j)} \sum_{v \in V_i} p(v) p^i a^2(j|v) p^{2(j-i-1)} \\ &= p^{-1} \frac{a(i)}{a^2(j)} \sum_{v \in V_i} p(v) a^2(j|v) = \kappa_{ij}/p \end{aligned}$$

In order to allow a direct comparison between the bounds of Lemmas 2 and 3, it is necessary to relate the constants $(s_i)_{1 \leq i < \ell}$ and $(s'_i)_{1 \leq i < \ell}$. These constants govern the particle reproduction rate of GWTW and can either be set externally or determined by a sampling procedure described in [1]. If we set $s'_i = p^{i-1}s_i$ then the expectations of Lemmas 2 and 3 are equal and the variance bounds are within a factor of p of each other. The variance bound is used in [1] in connection with Chebyshev's inequality to provide a lower bound on the success probability of GWTW. It follows that the negative effect of random queues on the GWTW variance bounds can be compensated for by increasing the number B of particles at the root by a factor of $1/p$.

5 Future Work

Another possible class of applications for a random queue is randomized Byzantine agreement algorithms in which the set of faulty processes can change from round to round (e.g. Rabin's algorithm [9,7]). Random errors in the queue can be attributed to faulty processes. Issues to be resolved include how to adapt the message passing algorithms to the situation when too few messages are received; also whether probabilistic quorum algorithms in [6] that tolerate Byzantine failures can be exploited here.

Actually, the applications we identified do not even require the per-process ordering — a shared multiset would work just as well. An open question is whether there is a randomized implementation of a multiset, with no ordering guarantees, that is more efficient in some measure than the algorithm presented in this paper. A complementary question is to identify distributed applications that would need ordering properties on a shared queue. Clearly one can imagine a variety of weakened queue definitions and a variety of implementations. Specifying and analyzing these are challenges for future work.

Acknowledgments

We thank Marcus Peinado for helpful conversations on GWTW.

References

1. Aldous, D., Vazirani, U.: “Go With the Winners” Algorithms. *Proc. 35th IEEE Symp. on Foundations of Computer Science*, pp. 492–501, 1994. 588, 594, 596, 597
2. Bertsekas, D., Tsitsiklis, J.: *Parallel and Distributed Computation*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1989. 588
3. Chakrabarti, S., Ranade, A., Yelick, K.: Randomized Load Balancing for Tree-structured Computation. *Proc. IEEE Scalable High Performance Comp. Conf.*, pp. 666–673, 1994. 588
4. Lee, H., Welch, J. L.: Applications of Probabilistic Quorums to Iterative Algorithms. *Proc. 21st IEEE Int. Conf. on Distributed Computing Systems*, pp. 21–28, 2001. 587, 588
5. Malkhi, D., Reiter, M.: Byzantine Quorum Systems. *Proc. 29th ACM Symp. on Theory of Computing*, pp. 569–578, 1997. 587
6. Malkhi, D., Reiter, M., Wright, R.: Probabilistic Quorum Systems. *Proc. 16th ACM Symp. on Principles of Dist. Comp.*, pp. 267–273, 1997. 587, 590, 598
7. Motwani, R., Raghavan, P.: *Randomized Algorithms*, Cambridge Univ. Press, 1995. 598
8. Peinado, M., Lengauer, T.: Parallel ‘Go with the Winners’ Algorithms in the LogP Model. *Proc. IPPS 97*, 1997. 595
9. Rabin, M. O.: Randomized Byzantine Generals. *Proc. 24th Symp. on Foundations of Computer Science*, pp. 403–409, 1983. 598
10. Yelick, K. et al.: Parallel Data Structures for Symbolic Computation. Workshop on Parallel Symbolic Languages and Systems, Oct. 1995. 588

11. Yelick, K. et al.: Data Structures for Irregular Applications. DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems, June 1993. 588

Multiprocess Time Queue [★]

Andrej Brodnik^{1,2} and Johan Karlsson¹

¹ Department of Computer Science and Electrical Engineering
Luleå University of Technology, Luleå, Sweden
{[andrej.brodnik](mailto:andrej.brodnik@sm.luth.se),[johan.karlsson](mailto:johan.karlsson@sm.luth.se)}@sm.luth.se

² Department of Theoretical Computer Science
Institute of Mathematics, Physics, and Mechanics, Ljubljana, Slovenia

Abstract. We show how to implement a bounded time queue for two different processes. The time queue is a variant of a priority queue with elements from a discrete universe. The bounded time queue has elements from a discrete bounded universe. One process has time constraints and may only spend constant worst case time on each operation while the other process may spend more time. The time constrained process only has to be able to perform some of the time queue operations while the other process has to be able to perform all operations. We show how to do a deamortization of the `deleteMin` cost and to provide mutual exclusion for the parts of the data structure that both processes maintain.

1 Introduction

In this paper we look at a special variant of the *Priority Queue* problem which we call the *Time Queue* problem. A time queue is a queue that stores elements together with a time stamp. Newly inserted elements must have a time stamp that lies in the future. The time queue can be used in various ways. One task might be as a time-out manager, where an element has to be processed before some given time otherwise it should be considered to have timed-out and be handled specially. The time queue can also be used for the *simulation event set* problem [3] and other *scheduling* problems.

The time queue supports, given a set $\mathcal{N}$ of N elements, the ordinary operations of a priority queue, `insert`, `min` and `deleteMin`. By convention the highest priority has the lowest numerical value, hence `min`. We refer to the element with the minimum numerical priority value as the *min element* and use t_0 to denote its priority.

Usually a priority queue supports the `decrease-key` operation, which decreases the priority of an element in the queue. The `increase-key` operation is also supported by the time queue and we combine these operations into a general `update` operation, which updates the priority of an element.

Further, a general `delete` operation is also supported in the time queue. Therefore, we let `insert` return a *finger* to the inserted element, which can be

[★] An extended version has been published as tech. report IMFM-(2001)-PS-771.
<http://www.ijp.si/ftp/pub/preprints/ps/2001/pp771.ps>

used by other operation such as **delete** and **update**. It is convenient that the **min** operation also returns a finger. Since we use fingers we need operations to get the priority and element from the finger, **value** and **data** respectively. In this paper we use the terms element and finger to an element interchangeably.

Finally, the time queue supports deletion of all elements with a priority less than a specific value, using the operation **delLessThan**. The **delLessThan** can be augmented with an additional function, $\mathcal{F}$, that is called for each deleted element. Note that this forces the **delLessThan** to take $\Omega(d \cdot F)$ time where d is the number of deleted elements and F is the running time of the function $\mathcal{F}$. Without loss of generality we assume that $F = \Theta(1)$.

In the time queue the priorities are times. We assume that time is a discrete value and hence the time queue is restricted to only support priorities that are discrete (e.g. integers). We require that for the time t_e of the newly inserted or of the updated element e must hold $t_e > t_0$, which means that the min time t_0 is non-decreasing, the time queue is *monotonic* [12]. Moreover, we require that the time for any element in the time queue is less than $t_0 + C$, where C denotes the maximum duration of any element (cf. *maximum event duration* [4]). To sum up: time is drawn from a bounded discrete universe.

The above description gives the following formal definition:

Definition 1. *The Time Queue problem is the problem of maintaining a set, $\mathcal{N}$, of elements to support the following operations:*

insert(e, t): f *Iff $t_0 < t \leq t_0 + C$ then let $\mathcal{N} := \mathcal{N} \cup \{e\}$ and return a finger f to the newly inserted element.*

delete(f) *let $\mathcal{N} := \mathcal{N} \setminus \{f\}$.*

min() : f *Find the min element and return a finger, f , to it.*

deleteMin() *Delete the min element.*

update(f, t) *Iff $t_0 < t \leq t_0 + C$ then change the time of f to t .*

delLessThan($t, \mathcal{F}$) *Delete all elements with time less than t and call the function ' $\mathcal{F}$ ' for each of the deleted elements.*

where t_0 is the priority of the min element and C is the maximum duration of any element.

This research was initiated by a manufacturer of a firewall. In their firewall IP packets are processed in two different paths called *fast* and *slow* path. The fast path must not be delayed when using the time-out manager and this process needs only some of the operations of the time queue. The slow process has to be able to perform all the operations, but it is not that time sensitive.

Hence, in our model, we have two different processes manipulating the data structure. The first process (*fast*) has to be able to perform the **min**, **value**, **data** and a restricted **update** operations. The second process (*slow*) has to be able to perform all the operations on the time queue. The *fast* process is time critical and must not be delayed, i.e., the operations it uses must run in $O(1)$ worst case time. The fast process **update** only needs to update elements in the near future, i.e., only elements with current and new time less than $t_0 + \epsilon$ (ϵ is to be defined later).

Our main goal is to implement the operations of the *fast* process to run in $O(1)$ worst case time, hence amortized or expected time is not good enough. To do this, we let only the operations `deleteMin` and the `delLessThan` change the min element. this makes the operations `delete` and `update` more restricted, and, consequently, less complicated than `deleteMin` and the `delLessThan`. We refer to the time queue problem with these restrictions as the *restricted Time Queue problem*.

Furthermore, `delLessThan` is called by the slow, and this at least every c time units for some small value c .

To allow the two processes to share data we need mutual exclusion of the operations. For this we use locks and the interface to the locks has to provide both blocking and non-blocking locking functions. We also assume that the processes can pass messages asynchronously.

To compare different priority queues both theoretically and practically the *hold model* [10] has been used. In this model a priority queue of size N is created and a *hold* operation is performed a number of times. The *hold* operations is a sequence of `min`; `deleteMin`; and `insert` operations, hence N is not changed. The priority of the newly inserted element is $t_0 + d$ for some value d .

In the following section we look at how other solutions can be used to solve the restricted time queue problem, in particular the *Calendar Queue* by Brown [3]. In Sect. 3 we present our solution, a modification of the calendar queue, to support the operations of the fast process while Sect. 4 concludes the paper.

2 Previous Work

A number of solutions for the priority queue problem can be used to solve the time queue problem for one process with only small modifications if any. The standard *heap* described by Williams [18] can be modified to use fingers by adding a dictionary that stores the position in the heap for each element. The heap solution (*heap* in Table 1) even works if the maximum duration is unbounded and it only needs $O(N)$ space. The model used is the pointer machine model [11].

Van Emde Boas et al. proposed a data structure they call a *stratified tree* which supports the time queue operations in $O(\lg \lg C)$ time (*vEB* in Table 1) [13,15]. However, the stratified tree needs $O(C + N)$ space. The model is the pointer machine model.

Willard shows how perfect hashing (see [8,6]) can be used to improve the space bound to $O(N)$ for the stratified tree [17] (*vEB-W* in Table 1). The model is the RAM model [14] of the stronger cell probe model [19] due to the hashing.

More recently Andersson and Thorup improved their *exponential search* trees to achieve worst case performance of $O(\sqrt{\lg N / \lg \lg N})$ [1] (*EST* in Table 1). The model used here is the RAM model.

Brodnik et al. showed how a *split tagged tree* can be used to achieve worst case constant time for all the time queue operations (*SST* in Table 1) [2]. They use $O(C + N)$ space in the Yggdrasil implementation [2] of the RAMBO model [9].

Table 1. Time bounds for different solutions to the Time Queue problem

Operation	Heap	vEB	vEB-W	EST	STT	CQ
insert	$O(\lg N)$	$O(\lg \lg C)$	$\exp O(\lg \lg C)$	$O(\sqrt{\lg N / \lg \lg N})$	$O(1)$	am $O(1)$
delete	$O(\lg N)$	$O(\lg \lg C)$	$\exp O(\lg \lg C)$	$O(\sqrt{\lg N / \lg \lg N})$	$O(1)$	am $O(1)$
min	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
deleteMin	$O(\lg N)$	$O(\lg \lg C)$	$\exp O(\lg \lg C)$	$O(\sqrt{\lg N / \lg \lg N})$	$O(1)$	am $\exp O(1)$
hold	$O(\lg N)$	$O(\lg \lg C)$	$\exp O(\lg \lg C)$	$O(\sqrt{\lg N / \lg \lg N})$	$O(1)$	$\exp O(1)$
update	$O(\lg N)$	$O(\lg \lg C)$	$\exp O(\lg \lg C)$	$O(\sqrt{\lg N / \lg \lg N})$	$O(1)$	$O(1)$
Space	$O(N)$	$O(C + N)$	$O(N)$	$O(N)$	$O(C + N)$	$O(N)$

So far we have seen the bounds in Table 1, with the Calendar queue (CQ) presented below.

2.1 The Calendar Queue

The *Calendar Queue* data structure described by Brown [3] and analyzed by Erickson et al. [7] is a priority queue specially designed for the event set problem. Erickson et al. give a short and good description of the calendar queue that we restate here.

“A *calendar queue* has M buckets numbered 0 to $M - 1$, a current bucket with index i_0 , a bucket width δ , and a current time t_0 . We have the relationship that $i_0 = (t_0 \div \delta) \bmod M$. For each element e in the calendar queue, $t_e \geq t_0$, and element e is located in bucket i if and only if $i \leq (t_e \div \delta) \bmod M < (i + 1)$.”

The calendar queue is implemented as an array of lists, which we denote **buckets**. Depending on *bucket discipline* the lists in the buckets are either sorted or unsorted. In unsorted buckets **insert** takes constant time and **min** takes time proportional to the number of elements in the bucket. On the other hand, in sorted buckets **min** (**deleteMin**) takes constant time and **insert** time proportional to the log of the number of elements in the bucket. In Brown’s and Erickson’s descriptions all buckets use the same bucket discipline.

Brown [3] suggests to use a doubling technique to adjust the number of buckets M to be $\Theta(N)$ where N is the number of elements in the queue. Hence, when inserting an element and N becomes greater than M , we allocate $2M$ new buckets, copy all the elements to the new buckets and deallocate the old buckets. When deleting an element and N becomes less than $M/4$, we allocate $M/2$ buckets, copy all the elements and deallocate the old buckets. We see that, if a doubling of the number of buckets occurs when there are N_0 elements, at least N_0 new elements has to be inserted into the queue before the next doubling will occur. Hence the copying cost of the $2N_0$ elements at the second double can be charged to the insertion of the N_0 elements. Similarly for deletes and the copying cost when halving the number of buckets. The bucket width δ should be adjusted to match the average distance between elements in the queue in order to get an expected constant number of elements in each bucket. Hence, insert

and delete can be done in expected $O(1)$ amortized time. Brown gave empirical evidence that the calendar queue achieves expected constant time for the **hold** operation. In other words, if we choose δ and M properly, the number of elements in each bucket will be $O(1)$.

Erickson et al. (see “Optimizing Static Calendar Queues” [7] for details, *Static* here means that the number of elements in the queue is unchanged, not that all events have to be known in advance) analyzed the calendar queue with unsorted buckets. They describe how to choose δ and M under the assumption that only the **hold** operation is used (the case for which Brown gave empirical evidence). The value d in the **hold** operations is here defined by a random variable with probability density e . In essence, choose $\delta = \sqrt{2} \frac{\mu}{N}$ where μ is a function of e . Using this bucket width and infinitely many buckets the expected time is constant for the **hold** operation. Given a maximum duration C , choosing $M \geq C \div \delta + 1$ will guarantee no loss of performance over choosing infinitely many buckets. If a small degradation of the performance is acceptable one can choose $M = rN$, where r depends on the allowed degradation.

A variation of the time queue problem has been studied by Varghese and Lauck [16]. They look at the problem of providing a timer facility for an operating system. In the timer facility problem the **delLessThan** operation is called once for each time t (i.e., $c = 1$). Also even if $t < t_0$. The solution suggested by Varghese and Lauck, called *Hashed and Hierarchical Timing Wheel*, is very similar to the Calendar Queue.

3 Our Solution

We will now modify the calendar queue to achieve $O(1)$ worst case time for the **min**, **update**, **value** and **data** operations, and see under what conditions we can expect **deleteMin** and **delLessThan** to run in $O(1)$ time per deleted element. As Erickson et al. we will use the *unsorted* bucket discipline to achieve $O(1)$ worst case time for insertion into a bucket. We use lists of doubly linked nodes in each bucket and let a finger be a reference to the node that stores the element. Given a finger to the element, this achieves $O(1)$ worst case time for deletion in a bucket.

As pointed out by Thorup [12] we can always, in any monotonic priority queue, make the **min** operation run in $O(1)$ worst case time by remembering the element (and its priority) that was deleted by the last **deleteMin** and consider it part of the priority queue. We implement this by letting **deleteMin** find the element that will be min when the current min is deleted and store a finger to this element.

Since an **update** is a **delete** followed by an **insert**, if we can support **delete** and **insert** in $O(1)$ worst case time we also have **update** in $O(1)$ worst case time. The reason for the amortization in the calendar queue is the copying of elements when M is changed. If we never need to copy any elements during an **insert** (**delete**) the time for these operations is worst case. Hence, if M and δ are fixed

the copying is never needed and we achieve $O(1)$ worst case time for the `min`, `update`, `value` and `data` operations.

Under what conditions can we expect `deleteMin` to run in $O(1)$ time, and can we improve these conditions in some way? The approach with fixed M and δ is what Brown started with in his description of calendar queue. He noted that this will lead to inefficient space use if $N \ll M$. Moreover, if $N \ll M$, `deleteMin` may have to search many empty buckets to find the next element, `deleteMin` takes $O(M)$ time in the worst case. On the other hand, if $N \gg M$, the current bucket may contain many elements, `deleteMin` takes $O(N)$ time in the worst case. However, on the average $O(M/N + N/M)$ time is needed. From the discussion above we conclude that we can expect $O(1)$ time for `deleteMin` if $N = \Theta(M)$ and the elements are evenly distributed among the buckets.

To improve these conditions we will focus on the case where $N = \Omega(M)$, and the main problem of `deleteMin` is to find the next element in the current bucket. Since the elements in the buckets are unsorted it takes time proportional to the number of elements in the bucket to find the new min.

One way of reducing the time could be to keep the current bucket sorted, then deletion of the min element in the bucket would take $O(1)$ time. Each element would then be involved in one sorting and the amortized cost per element would be $s(k)$ where $s(k)k$ is the cost of sorting k elements (cf. equivalence between sorting and priority queues [12]). This makes `update` of elements within the bucket i_0 too expensive for the *fast* process.

Instead, we use δ buckets of width 1, implemented as an array of doubly linked lists denoted `head`. We store the elements of the current bucket i_0 in `head[j]` where $j = t_e \bmod \delta$. Hence, each list in the head only stores elements with the same priority. Now `update`, `insert`, `delete` and `min` are still $O(1)$ in the worst case even though the constant is a bit higher.

In the analysis of `deleteMin` we denote the number of elements in a bucket i by B_i and the number of elements in the `head` by H . Finding the new min in the head is similar to finding the next non empty bucket in the calendar queue, which is done in $O(M/N)$ time on the average. Hence in a head with more than one element it takes $O(\delta/H + 1)$ time on the average. If $H = \Omega(\delta)$ this is $O(1)$. When the last element of the head is deleted, and all the δ buckets are empty, we need to move all the B_{i_0+1} elements of bucket $i_0 + 1$ into the head and increase i_0 by one. The cost of the copying is $O(B_{i_0+1})$, which indicates that the worst case cost of `deleteMin` is $O(N)$. However in an amortized analysis the cost of copying an element can be charged to the operation that deletes the element from the head, which makes the amortized time $O(1)$ for `deleteMin`.

Finally, we do a deamortization of the `deleteMin` operation to achieve $O(1)$ expected time instead of amortized time. The deamortization is done by using a second head denoted `head2` and move $\lceil B_{i_0+1}/H \rceil$ elements from bucket $i_0 + 1$ into `head2` in each `deleteMin` operation. When the last element is deleted from the head the rest of the elements are moved from bucket $i_0 + 1$ into `head2` and the two heads are swapped. If an element should be inserted (updated) into bucket $i_0 + 1$ it will instead be inserted into `head2`. Hence B_{i_0+1} will never increase and

Table 2. Time bounds and space for our solution to the Time Queue problem

Operation	Our modified CQ
insert	$O(1)$
delete	$O(1)$
min	$O(1)$
deleteMin	$\exp O(1)$ if $H = \Omega(\delta), H = \Omega(B_{i_0+1})$
hold	$\exp O(1)$ if $H = \Omega(\delta), H = \Omega(B_{i_0+1})$
update	$O(1)$
delLessThan	$\exp O(1)$ if $\delta = \Omega(B_{i_0+1})$
Space	$O(\sqrt{C} + N)$

therefore the number of elements in bucket $i_0 + 1$ will be $O(1)$ when the last element is deleted from the head. If $H = \Omega(B_{i_0+1})$ the cost is $O(1)$ for copying the elements.

Now if $N = \Omega(M)$, $H = \Omega(\delta)$ and $H = \Omega(B_{i_0+1})$ we have $O(1)$ expected time for **deleteMin**. If not, the time for **deleteMin** and **hold** is $\exp O(M/N + B_{i_0+1}/H + \delta/H)$ and $O(M + N + \delta)$ worst case. If we choose $\delta = \Theta(M)$ the above conditions reduce to $H = \Omega(\delta)$ (since $N \geq H$) and $H = \Omega(B_{i_0+1})$ where the second condition depends on the distribution of the elements among the buckets.

Now let us see what conditions are needed if a sequence of **delLessThan** calls are used instead of **deleteMin**. First we note that $O(\delta/c)$ calls are made between two changes of heads. Consequently, if $\lceil B_{i_0+1}/(\delta/c) \rceil$ elements are moved each time **delLessThan** is called, all the elements in bucket $i_0 + 1$ will be moved to the second head before the next head swap. If $\delta = \Omega(B_{i_0+1})$ then only a constant number of elements are moved each time. If the min element is deleted, **delLessThan** needs to find the next element to be min which is done in $O(1)$ time on the average if $H = \Omega(\delta)$. However, assume that p empty buckets have to be scanned to find the next element to be min, when the min element is deleted, then approximately p/c **delLessThan** calls are made before the min is deleted again. Hence on the average c buckets are scanned by **delLessThan** even if $H = o(\delta)$. Note that this is true even if we need to scan the **buckets** array to find the next non empty bucket. The condition we are left with is $\delta = \Omega(B_{i_0+1})$ which is fulfilled if there is only a constant number of elements with equal priority.

Since we know the maximum duration C of all the elements, we can choose δ and M to cover this range, hence $C = M\delta$. We choose $M, \delta = O(\sqrt{C})$ to have $\delta = \Theta(M)$. This gives a space bound of $O(\sqrt{C} + N)$ since the number of buckets and the number of buckets in the heads are $O(\sqrt{C})$. We note that when using $\delta = M = \sqrt{C}$ both $\text{mod}\sqrt{C}$ and $\text{div}\sqrt{C}$ can be computed fast if $C = 2^h$ and even if $C \neq 2^h$ the approximation $\delta = 2^{\frac{\lg C}{2}}$ is good enough. The above analysis leads to the time and space bounds in Table 2.

3.1 Representation and Algorithms

The data structure consists of buckets, two heads of buckets, a finger to the min element and an index into the current bucket (see Algorithm 1). Both the buckets and the two heads of buckets are arrays of lists of doubly linked nodes. We let a finger be a reference to a node.

```
typedef struct tq {
    LIST      buckets[M];
    LIST      head[ $\delta$ ];      int  $H$ ;
    LIST      head2[ $\delta$ ];      int  $H2$ ;
    NODE *    min;
    int  $i_0$ ;
} TQ;
```

Algorithm 1: Representation of the Time Queue

We first look at the algorithms for only one process (the slow one) and later see what modifications are needed for the fast process.

- During **insert** we calculate the bucket index for the new element and check if the element should be in either of the heads. If it is we calculate the head index j and insert the new element at the end of the list, otherwise we insert it at the end of the list of the proper bucket.
- In the **delete** we have the finger the element and we can easily delete it from the appropriate list. If it is the last element in the list we mark the bucket as empty.
- The **update** is, as said, a deletion followed by an insertion.
- The **min** returns the stored min finger.
- The **deleteMin** first deletes the min element. Then it searches for the next element that should be min and updates the min reference. Finally, the routine moves some of the elements from the next bucket into **head2**. If the deleted element was the last element in the head, it first search for the next non empty bucket, moves the remaining elements from that bucket to **head2**, swaps the two heads, and increases i_0 . Then it continues with the search for the next element to be min.
- As long as the time t_0 of the min element is less than the specified time, **delLessThan** gets min, calls the function $\mathcal{F}$ on it, deletes it and finds the new min. Finally, it moves some elements from the next bucket into **head2**. If **delLessThan** deletes the last element in the head it search for the next non empty bucket, moves the remaining elements from that bucket to **head2**, swaps the two heads, and increases i_0 .
- The operations to get the data and time from a finger, **data** and **value** respectively, only returns the data and time from the linked list node.

3.2 Support for Concurrent Processes

We assume that there are just two processes: one fast process and one slow process. The fast process only has to update elements in the near future ϵ while the remaining updates are sent to the slow process. Without loss of generality we assume that $\epsilon \leq \delta$ and hence only elements in bucket i_0 and bucket $i_0 + 1$ may be updated by the fast process. The elements in bucket i_0 are stored in **head** while the elements in bucket $i_0 + 1$ may be in both **buckets** $[i_0 + 1]$ and **head2**. We will use three locks to ensure mutual exclusive access to these entities. The assumption that $\epsilon \leq \delta$ is not really a restriction since if this is not true we only need to add more locks for the buckets that need protection.

Whenever **head**, **H**, i_0 or **min** is read or written we acquire **headLock**. Similarly for **head2Lock** and **bucketLock**. Since only the slow process modifies i_0 and **min** it does not need to acquire the **headLock** in order to read these variables. The fast process always has to acquire the corresponding lock. To avoid deadlocks, we choose to break the circular chain condition by imposing a linear order of the locks [5]. If a process needs more than one lock it has to acquire them in the following order: **headLock**, **head2Lock** and **bucketLock**. The representation of the time queue includes these locks (Algorithm 2). The **update** for the fast

```
typedef struct tq {
    LIST      buckets[M];
    LIST      head[ $\delta$ ];          int H;
    LIST      head2[ $\delta$ ];        int H2;
    NODE *    min;
    int i_0;
    LOCK      headLock;
    LOCK      head2Lock;
    LOCK      bucketLock;
} TQ;
```

Algorithm 2: Representation of the Time Queue with locks

process is only allowed to move elements to/from the heads and corresponding buckets. If the element should be moved to/from another bucket it passes a request to the slow process.

If there are more than one process of each kind special care is needed for the slow processes. We need also to acquire the lock when reading variables in the slow process and have locks for all the different parts of the time queue data structure. More than one fast process can be handled without any special care.

4 Conclusion

We have proposed a solution for two different processes to simultaneously maintain a time queue. One of the processes performs only a subset of the operations in $O(1)$ worst case time, while the other process shall perform all operations. All operations except `deleteMin` and `delLessThan` are performed in $O(1)$ worst case time. `deleteMin` is performed in $O(1)$ expected time and `delLessThan` is performed in $O(1)$ expected time per deleted element.

The main difference from the Hashed and Hierarchical Timing Wheels by Varghese and Lauck [16] is the deamortization of the `deleteMin` and the concurrent solution.

Furthermore, we have shown how to allow one fast and one slow process to maintain our data structure by using locks to provide mutual exclusion.

References

1. Arne Andersson and Mikkell Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, pages 335–342, Portland, Oregon, US, May 21–23 2000. ACM Press. 601
2. Andrej Brodnik, Svante Carlsson, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 523–528, Washington, DC, US, 7–9 January 2001. 601
3. Randy Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988. 599, 601, 602
4. Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 83–92, New Orleans, Louisiana, US, 5–7 January 1997. 600
5. E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971. 607
6. Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–791, 1994. 601
7. K. Bruce Erickson, Richard E. Ladner, and Anthony LaMarca. Optimizing static calendar queues. In *35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 732–742. IEEE, 20–22 November 1994. Also published as tech. report TR-94-09-02 <ftp://ftp.cs.washington.edu/tr/1994/09/UW-CSE-94-09-02.PS.Z>. 602, 603
8. Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984. 601
9. Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, US, May 14–17 1989. ACM Press. 601

10. Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986. 601
11. Arnold Schönhage. Storage modifications machines. *SIAM Journal on Computing*, 9(3):490–508, August 1980. 601
12. Mikkel Thorup. On RAM priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 59–67, Atlanta, Georgia, US, 28–30 January 1996. 600, 603, 604
13. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Symposium on Foundations of Computer Science*, pages 75–84, 1975. 601
14. P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 3–66. Elsevier/MIT Press, Amsterdam, 1990. 601
15. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977. 601
16. George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Efficient data structure for implementing a timer facility. *IEEE/ACM Transaction on Networking*, 5(6):824–834, December 1997. 603, 608
17. Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 24 August 1983. 601
18. J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7:347–348, 1964. 601
19. Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):614–628, July 1981. 601

Labeling Points with Weights

Sheung-Hung Poon¹, Chan-Su Shin², Tycho Strijk³, and Alexander Wolff⁴

¹ Dept. of Comp. Science, HKUST, Hong Kong
hung@cs.ust.hk

² School of Electronics and Information Engineering
Hankuk University of Foreign Studies, Korea
cssin@hufs.ac.kr

³ Dept. of Comp. Science, Utrecht University, The Netherlands
tycho@cs.ruu.nl

⁴ Institut für Mathematik und Informatik, Universität Greifswald, Germany
awolff@uni-greifswald.de

Abstract. Annotating maps, graphs, and diagrams with pieces of text is an important step in information visualization that is usually referred to as label placement. We define nine label-placement models for labeling points with axis-parallel rectangles given a weight for each point. There are two groups; fixed-position models and slider models. We aim to maximize the weight sum of those points that receive a label.

We first compare our models by giving bounds for the ratios between the weights of maximum-weight labelings in different models. Then we present algorithms for labeling n points with unit-height rectangles. We show how an $O(n \log n)$ -time factor-2 approximation algorithm and a PTAS for fixed-position models can be extended to handle the weighted case. Our main contribution is the first algorithm for weighted sliding labels. Its approximation factor is $(2 + \varepsilon)$, it runs in $O(n^2/\varepsilon)$ time and uses $O(n/\varepsilon)$ space. We also investigate some special cases.

1 Introduction

Label placement is one of the key tasks in the process of information visualization. In diagrams, maps, technical or graph drawings, features like points, lines, and polygons must be labeled to convey information. The interest in algorithms that automate this task has increased with the advance in type-setting technology and the amount of information to be visualized. Due to the NP-hardness of the general label-placement problem [6], cartographers, graph drawers, and computational geometers have suggested numerous approaches. Several heuristic methods have been analyzed experimentally [4]. An extensive bibliography about label placement can be found at [13]. The ACM Computational Geometry Impact Task Force report [3] denotes label placement as an important research area.

This paper deals with one of the most common label-placement problems, namely labeling points with axis-parallel rectangles. With two exceptions this is the first paper that gives approximation algorithms for labeling *weighted* points.

The aim is to maximize the sum of the weights of those points whose labels can be placed without intersection. Solving this problem is extremely important in praxis: on a map of Germany, e.g., attributing Berlin a higher priority (weight) than Wannsee ensures that in case of limited space the capital rather than one of its districts receives a label. The only two other approximation algorithms for weighted label placement are the following. First, Iturriaga [8] showed how a factor- $O(\log n)$ approximation algorithm of Agarwal et al. [1] for maximum-independent set on rectangle-intersection graphs can be extended to handle weighted rectangles as well (n is the number of rectangles here). Second, Erlebach et al. [5] recently improved these results for squares; they give a polynomial-time approximation scheme (PTAS) for the weighted case.

Van Kreveld et al. defined six point-labeling models [12]. They forged the term of *slider models* where a label can slide along one or several edges under the constraint that it touches the point it labels, see Figure 1. This is opposed to *fixed-position models* that allow only a constant number (like 4 or 8) of *label candidates* per point. Van Kreveld et al. compared three fixed-position (namely 1P, 2PH, and 4P in Figure 1) and three slider models (1SH, 2SH, and 4S) with respect to how many more points can be labeled in one model than in another using unit square labels [12]. Since we are considering labels with equal height but variable length, we need to classify more models. Figure 1 shows all nine fixed-position models and slider models that we will consider in this paper. In that figure, each rectangle stands for a feasible label position. An arrow between two rectangle indicates that additionally all label position are feasible that arise when moving one rectangle on a straight line onto the other. We refer the reader to [12] for a more formal definition.

For each of their six labeling models, van Kreveld et al. gave approximation algorithms for unit-height labels in the unweighted case. They also did an experimental comparison that showed that algorithms for sliding labels perform especially well on dense point sets such as scatterplots. Other applications with dense point sets include drill holes or electrophoresis gels.

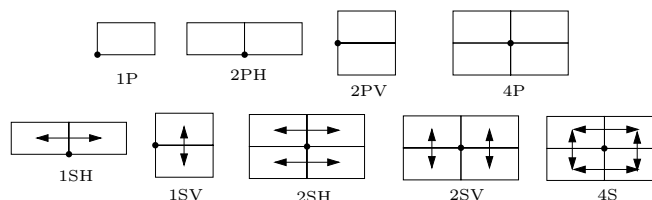


Fig. 1. Each model has an abbreviation of the form xMD where $M \in \{P, S\}$ stands for fixed-position model (P) or slider model (S), $x \in \{1, 2, 4\}$ refers to the number of fixed positions or sliding directions, and $D \in \{\emptyset, H, V\}$ indicates the horizontal or vertical direction in which fixed-position labels are arranged or labels can slide

We extend the results of van Kreveld et al. by taking weights into account. More specifically we present the following results. In all but the last section we assume unit-height labels. First, in Section 2, we compare our nine labeling models by giving bounds for the ratios between the weights of maximum-weight labelings in different models. In Section 3, we show how to extend an $O(n \log n)$ -time factor-2 approximation algorithm and a PTAS for fixed-position models [1] to the weighted case. The main contribution of this paper besides the comparison of labeling models is the first approximation algorithm for weighted sliding labels. Its approximation factor is $(2 + \varepsilon)$, it runs in $O(n^2/\varepsilon)$ time and uses $O(n/\varepsilon)$ space. The algorithms for both the fixed-position and the slider models use line stabbing, a technique that has already been used successfully for label placement [1,12]. The idea is to partition the two-dimensional problem into easier one-dimensional subproblems by stabbing the unit-height label candidates of the input points by horizontal lines of at least unit distance such that each label candidate is stabbed. If the resulting subproblems can be solved optimally (near-optimally), then the union of the subsolutions corresponding to either all odd or all even stabbing lines gives a factor-2 $(2 + \varepsilon)$ approximation for the original problem.

Section 4 and 5 deal with two restrictions of the one-dimensional problem for sliding labels (i.e. intervals) that can be solved optimally. In Section 4, we consider the case that the number of different weights is bounded and receive a factor-2 approximation for all slider models. In Section 5, we restrict all intervals to unit length and combine the resulting exact one-dimensional algorithm with a dynamic-programming algorithm of Agarwal et al. [1] to construct a PTAS for labeling points with sliding unit-square labels. In Section 6, we finally drop the restriction on label heights and give algorithms with approximation factors of $3\lceil \log_2 \beta \rceil$ and $(3 + \varepsilon)\lceil \log_2 \beta \rceil$ for fixed-position and slider models, respectively, where β is the ratio of maximum and minimum label height. Throughout this paper, we assume that labels are topologically open, i.e. they may touch. Due to space constraints we had to sketch some of the proofs. For the details we refer the reader to the full paper [10].

2 Comparing Labeling Models

Let M_1 and M_2 be any two different labeling models from Figure 1. Given a finite set P of points in the plane, where each point $p \in P$ is associated with a weight $w(p)$, let $W_M(P)$ denote the maximum sum of weights of points whose labels can be placed without intersections given labeling model M . Then the (M_1, M_2) -ratio is defined as $\Psi(M_1, M_2) = \lim_{n \rightarrow \infty} \max_{|P|=n} \frac{W_{M_1}(P)}{W_{M_2}(P)}$.

Our bounds for ratios between different labeling models are summarized in Figures 2 and 3. The numbers that are attached to the arcs between two models M_1 and M_2 give the (M_1, M_2) -ratio; intervals specify lower and upper bounds. The proofs for the constant ratios are similar to those in [12], those for linear ratios are simple. For both we refer the reader to the full paper [10]. Instead we

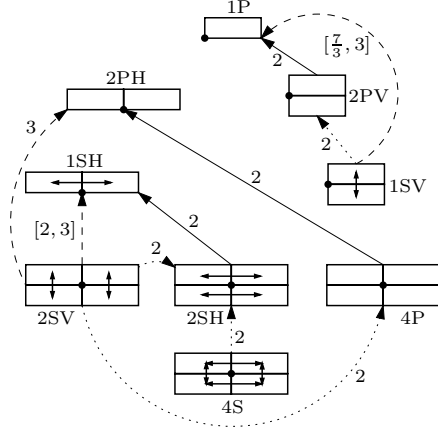


Fig. 2. Constant ratios between different labeling models

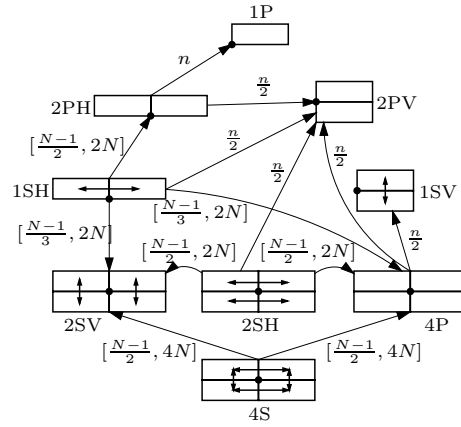


Fig. 3. Ratios that cannot be bounded by constants N is shorthand for $\log n$

investigate the more interesting logarithmic ratios. These do not appear in [12] since there only square labels were taken into account.

To bound $\Psi(1SH, 2PH)$, we consider two one-dimensional labeling problems that correspond to 1SH and 2PH. Given n points on the x -axis, each with a label length and a weight, find a feasible label placement that maximizes the sum of weights of the labeled points. We can interpret these labels as intervals on a line. In analogy to 1SH and 2PH we define two labeling models; a slider model 1d-1SH where a label can be attached to its point anywhere between its endpoints and a fixed-position model 1d-2PH in which a label must be attached to its point at one of its two endpoints. We have the following result.

Lemma 1 $\frac{1}{2} \log n \leq \Psi(1d-1SH, 1d-2PH) \leq \log n$.

Proof. Let P be a set of n points on the x -axis. For each point $p \in P$ we are given its position on the x -axis $x(p)$, its weight $w(p)$, and the length $\ell(p)$ of its label. If l is the label of p , then l must be placed within a “window” $[r, d]$ on the x -axis where $r = x(p) - \ell(p)$ and $d = x(p) + \ell(p)$. (The choice of the variable names is due to the similarity of our problem to scheduling problems which we will exploit again in Section 3. In scheduling, each job has a release time r and a deadline d .)

For the upper bound we assume that $n = 2^k$ for some integer $k > 0$. The main observation that we will use repeatedly below is the following. Consider a pair of adjacent labels $l = [a, b]$ and $l' = [a', b']$ of points p and p' in a fixed optimal 1d-1SH-labeling. Let l be to the left of l' and assume wlog. that l' is not shorter than l . Then the right endpoint d of the window of l must lie in the interval $[a, b']$. As a result, we can move (at least) l within $[a, b']$ to a valid

1d-2PH-position. The label l will possibly intersect l' but no other 1d-1SH-labels because the translation is done only within $[a, b']$.

The overall translation is performed in k phases as follows. Let P_0 be the subset of points of P that are labeled in the optimal 1d-1SH-labeling. Number the 1d-1SH-labels from left to right starting at 0, and pair labels with the numbers $2i$ and $(2i + 1)$. In phase 1, translate for each pair (at least) one label to a valid 1d-2PH-position as above. Denote by $P_1 \subseteq P_0$ the set of points whose labels have just been translated. Then $W_{1d-1SH}(P_1) = W_{1d-2PH}(P_1) \leq W_{1d-2PH}(P_0)$ and $|P_1| \geq |P_0|/2$. Recursively repeat the same process at phase j with $P_0 \setminus \bigcup_{i=1}^{j-1} P_i$ and set P_j to the subset whose labels are translated. After phase j we have that

$$W_{1d-1SH}(P_j) = W_{1d-2PH}(P_j) \leq W_{1d-2PH}(P_0)$$

and $|P_j| \geq |P_{j-1}|/2$. Due to the second inequality the process terminates after $k = \log n$ phases. Summing up the first inequality yields

$$\sum_{j=1}^{\log n} W_{1d-1SH}(P_j) \leq \log n \cdot W_{1d-2PH}(P_0).$$

Since the subsets P_j partition P_0 and P_0 is the subset of P that is labeled in the optimal 1d-1SH-labeling, the left term is equal to $W_{1d-1SH}(P_0) = W_{1d-1SH}(P)$. The right term is at most $\log n \cdot W_{1d-2PH}(P)$ since $P_0 \subseteq P$. Thus $\Psi(1d-1SH, 1d-2PH) \leq \log n$.

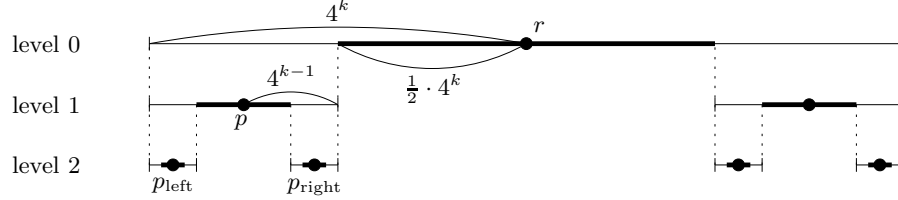


Fig. 4. The lower bound construction for $\Psi(1d-1SH, 1d-2PH)$. The points are all meant to lie on the x -axis. Their windows are delimited by vertical strokes. The labels of an optimal 1d-1SH-labeling are indicated by the bold line segments

For the lower bound consider a set P of n points, where we assume n to be $2^k - 1$ for convenience. The construction is similar to the recursive construction of a complete binary tree of k levels, see Figure 4. At level 0 we place the root r at $x(r) = 2^{2k-1}$. At level i ($0 \leq i \leq k-1$) we place 2^i points, where each point p has weight $w(p) = 2^{k-i}$ and a label of length $\ell(p) = 4^{k-i}$. If $i < k-1$ then p has two children p_{left} and p_{right} that lie on level $(i+1)$ at $x(p_{\text{left}}) = x(p) - \frac{3}{4}\ell(p)$ and $x(p_{\text{right}}) = x(p) + \frac{3}{4}\ell(p)$. The window of p is $[x(p) - \ell(p), x(p) + \ell(p)]$. An optimal 1d-1SH-labeling labels all points in P by centering the label of each point p within

its window, i.e. at $[x(p) - \ell(p)/2, x(p) + \ell(p)/2]$, see the bold line segments in Figure 4. Due to our construction no two labels intersect. Since the weights of the points at each level sum up to 2^k , the total sum is $W_{1d-1SH}(P) = k2^k \geq n \log n$.

However, any 1d-2PH-labeling can assign a label $l = [a, b]$ to a point p only such that either a or b coincides with $x(p)$. In either way, the points in one of the subtrees of p cannot be labeled because they lie entirely in l . We claim that the weight of an optimal 1d-2PH-labeling is at most $2(2^k - 1) = 2n$, which proves the lower bound. The proof is by induction on k , the number of levels of the tree. If $k = 1$, P consists only of one point whose weight is 2, so the claim clearly holds. Assume that for every tree with $i < k$ levels, the sum of weights of the points labeled is at most $2(2^i - 1)$. Now consider the tree T with k levels. This tree consists of a point at level 0 with weight 2^k and of two subtrees L and R with $k - 1$ levels each. The weight $W(T)$ of an optimal 1d-2PH-labeling of T is at most $\max\{\max\{W(L), W(R)\} + 2^k, W(L) + W(R)\}$ because the 1d-2PH-labeling has the choice to assign a label to the point at level 0 or not. By our assumption $\max\{W(L), W(R)\} + 2^k \leq 2(2^{k-1} - 1) + 2^k = 2(2^k - 1)$ and $W(L) + W(R) \leq 2(2^k - 2)$. Thus $W(T) \leq 2(2^k - 1)$, which completes the proof. Our proof also shows that exactly one point per level is labeled in the optimal 1d-2PH-labeling. $\square$

Lemma 2 $\frac{1}{2} \log n \leq \Psi(1SH, 2PH) \leq 2 \log n$

Proof. The lower bound is a direct consequence of Lemma 1. The upper bound is obtained by reducing 2PH to two sets of one-dimensional problems with the help of line stabbing as in [12], and by then applying Lemma 1. $\square$

In fact, Lemma 2 even holds for fixed-position models with any finite number of label positions. We will now extend the arguments of Lemma 1 and 2 to prove other $\Theta(\log n)$ -bounds. For the sake of brevity we will write $\{4P, 2SV\}$ when we mean that a statement holds for both 4P and 2SV.

Lemma 3 $\frac{1}{2}(\log n) - \frac{1}{2} \leq \Psi(2SH, \{4P, 2SV\}) \leq 2 \log n$.

Proof. For the lower bounds we construct an instance P that consists of two point sets with the tree-like structure used in Lemma 1. We place a set T of $n/2 = 2^t - 1$ points on the x -axis and a copy T' slightly above. This means that any 4P- or 2SV-labeling for T and T' cannot do better than 1d-2PH-labeling for T and T' . Thus $W_{\{4P, 2SV\}}(P) = 2 \cdot W_{1d-2PH}(T) = 2 \cdot 2(2^t - 1) = 2n$. However, the optimal 2SH-labeling can label all points in P , so $W_{2SH}(P) = 2 \cdot t2^t$. Hence $\Psi(2SH, \{4P, 2SV\}) = t/2 \cdot 2^t / (2^t - 1) \geq t/2 \geq \frac{1}{2}(\log n) - \frac{1}{2}$.

The upper bounds are achieved by the same argument as in Lemma 2. $\square$

Lemma 4 $\frac{1}{3}(\log n) - \frac{1}{3} \leq \Psi(1SH, \{4P, 2SV\}) \leq 2 \log n$.

Proof. The upper bound can be obtained as in the proof of Lemma 2. For the lower bound we split our point set P in two equal halves T and T' of $n/2 = 2^t - 1$ points as in the proof of Lemma 3. Again, both have the tree-like structure used

in Lemma 1. Other than in Lemma 3, however, we place T' at a distance of 1 above T . Thus all points can be 1SH-labeled and $W_{1SH}(P) = t^{2^{t+1}}$.

Now let us consider an optimal 4P-labeling. We split the available space into three regions: The space above T' , the space below T and the space between T and T' . The weight of the labels of an optimal labeling in each of these three areas is at most the weight of a labeling that is optimal with respect to that area. Lemma 1 says that an optimal labeling for the space above T' and the space below T each has at most weight $2(2^t - 1) \leq 2^{t+1}$. For the space in between we argue as follows. Let L be a label in that area. We claim that the weight of any labeling within L has at most the weight of L . This can be shown by induction over the level of L . By our claim the weight of two labels at level 0 is an upper bound for the weight of an optimal labeling that uses exclusively the space between T and T' . Thus $W_{4P}(P) \leq 2 \cdot 2^{t+1} + 2 \cdot 2^t = 3 \cdot 2^{t+1}$ and hence $\Psi(1SH, 4P) \geq t/3 \geq \frac{1}{3}(\log n) - \frac{1}{3}$. The case 2SV is analogous. $\square$

Lemma 5 $\frac{1}{2}(\log n) - \frac{1}{2} \leq \Psi(4S, \{4P, 2SV\}) \leq 4 \log n$

Proof. The lower bounds come from the same argument as that in Lemma 3 since each 2SH-labeling is also a 4S-labeling. The upper bounds are obtained by first two-way sliding a 4S-labeling into a 2SH-labeling with a factor-2 loss and by then translating the 2SH-labeling as above into 4P- and 2SV-labelings with another loss of a factor of 2 $\log n$. $\square$

3 Approximation Algorithms for Unit-Height Labels

In this section we present approximation algorithms for unit-height labels under all labeling models shown in Figure 1. Our algorithms employ line stabbing, a technique that has been used before to tackle labeling problems with unit-height labels [1, 12].

We first consider the problem *1d-1P* of finding a maximum weight independent set (MWIS) of n (topologically open) intervals on the x -axis. The problem is exactly the one-dimensional version of 1P, and it can be solved in $O(n \log n)$ time by a simple dynamic programming algorithm [7]. The one-dimensional version 1d-2PH corresponding to 2PH is as follows. Given a set of n points and for each an interval length, find a MWIS from the $2n$ intervals that either start or end at one of the input points. We generally view intervals as topologically open but now make them intersect artificially if they belong to the same point. This can be achieved by a symbolic comparison rule, which allows to use the above algorithm, although Hsiao et al. assume disjoint interval endpoints [7].

We can generalize 1d-2PH to the problem 1d- k PH in which each input point p has at most k candidate intervals that all contain p . Applying the 1d-1P algorithm to the resulting collection of kn intervals gives rise to:

Lemma 6 *The problem 1d- k PH where each input point has at most k candidate intervals can be solved in $O(kn \log n)$ time.*

Combining line stabbing with the above lemma and with dynamic programming as in [1] yields the following result.

Theorem 1 *The weighted fixed-position labeling problems 1P, 2PH, 2PV, and 4P can be 2-approximated in $O(n \log n)$ time with linear space. These problems can be $(1 + \frac{1}{k})$ -approximated in $O(n^{2k-1})$ time and space for any $k \geq 2$.*

The above lemma also yields an $O(kn \log n)$ -time factor-2 approximation algorithm for the two-dimensional analog kP of 1d- kP .

In what follows, we consider approximation algorithms for problems with sliding labels. Again we first tackle the corresponding one-dimensional problem. Given a set of n points, each with a weight and an interval length, the problem 1d-1SH consists of maximizing the weight sum of those points that can be labeled by intervals of the prescribed length such that each interval contains its point and no two intervals intersect. Due to its close relationship to 1d-1SH we now state a scheduling problem, namely *single-machine throughput maximization*: Given a set $\mathcal{J}$ of n jobs $J_1, \dots, J_n$, each with a weight w_i , a *processing time* (or job length) ℓ_i , a *release time* r_i and a *deadline* d_i , find a schedule that maximizes the *throughput* on a single machine, i.e. find a maximum-weight subset $\mathcal{J}'$ of the jobs and for each job $J_i \in \mathcal{J}'$ an open interval I_i of length ℓ_i that is contained in the *execution window* $[r_i, d_i]$ of J_i such that no two intervals intersect.

Lemma 7 *The single-machine throughput-maximization problem for a set $\mathcal{J}$ of n jobs can be approximated within a factor of $(1 + \epsilon)$ in $O(n^2/\epsilon)$ time using $O(n/\epsilon)$ storage if the stretch factor $\alpha = \max_i \{(d_i - r_i)/\ell_i\}$ of $\mathcal{J}$ is at most 2.*

Proof. In [2], Berman and DasGupta present a two-phase algorithm, ϵ -2PA, for single-machine throughput maximization with bounded stretch factor α . Their algorithm has an approximation factor of $2/(1 + \frac{1}{2\lceil\alpha\rceil+1-2-\lfloor\alpha\rfloor}) + \epsilon$ for any $\epsilon > 0$ and runs in $O(n^2/\epsilon)$ time. In the case $\alpha = 2$ this yields a factor- $(8/5 + \epsilon)$ approximation. However, using a symbolic comparison rule as in Lemma 6, we get the same approximation factor as for $1 < \alpha < 2$, i.e. $(1 + \epsilon)$.

Their algorithm uses $O(n^2/\epsilon)$ storage. We will now show how this can be reduced to $O(n/\epsilon)$ for $\alpha \leq 2$ assuming the above-mentioned symbolic comparison which ensures that all intervals of the same job intersect. In phase I, the *evaluation phase*, ϵ -2PA discretizes the problem depending on ϵ and on the job weights w_i . Intervals are generated in order of non-decreasing x -value of their right endpoint and are put on a stack S . In phase II, the *selection phase*, the intervals are successively taken off the stack and either put into the solution if they do not intersect any other interval there, or discarded otherwise.

The main idea behind the discretization is a *value* $v_I = w_I - \sum_{I' \cap I \neq \emptyset, I' \in S} v_{I'}$ that is attributed to each interval I when it is pushed on S . The weight w_I of I is the weight of the job to which I belongs. Phase I of ϵ -2PA consists of nothing but repeatedly determining the interval I^* whose right endpoint is leftmost among all intervals I with $v_I \geq \epsilon w_I$ and then pushing I^* on S . The threshold for v_I ensures that at most $1/\epsilon$ intervals of one job are pushed on S since they all intersect each other in our case. Thus $|S| \leq n/\epsilon$ at the end of phase I.

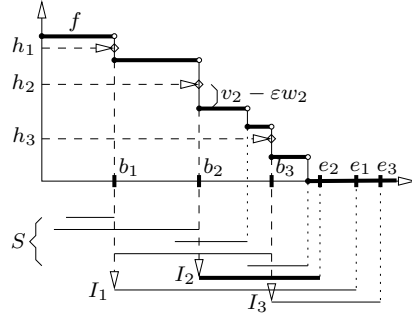


Fig. 5. Stack S and staircase function f

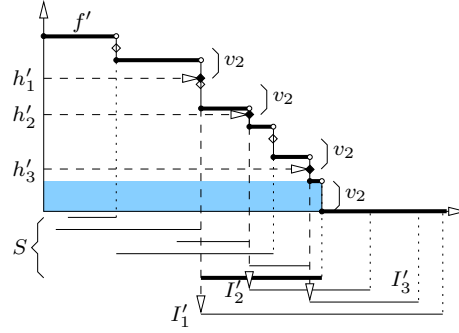


Fig. 6. After pushing I^* on the stack

In order to determine I^* we maintain a monotonically decreasing staircase function f that maps x to the sum of the values of all intervals on S that intersect (x, ∞) . For each job $h_i = (1-\varepsilon)w_i$ is the difference between weight and threshold. If $h_i > \max f$ let $b_i = r_i$. Otherwise maintain a marker at height h_i “on” f and its projection b_i on the x -axis as in Figure 5. Let $e_i = h_i + \ell_i$. Note that for each job, (b_i, e_i) is the leftmost interval whose value is above the threshold. Now I^* is the interval (b_j, e_j) of job J_j with e_j minimum among all e_i with $e_i \leq d_i$. In Figure 5 $I^* = I_2$. By construction $v_j = w_j - f(b_j) \geq w_j - h_j = \varepsilon w_j$. After pushing I^* on S , a new stair (shaded in Figure 6) of height v_j and length e_j (measured from origin) is attached to f from below and all markers are moved downwards by v_j , see Figure 6. Phase I terminates when $e_i > d_i$ for all jobs J_i .

Since f consists of $|S| \leq n/\varepsilon$ stairs at the end of phase I, f can be maintained in $O(n/\varepsilon)$ time and space simply as a list of numbers $e^1, v^1, e^2, v^2, \dots, e^{|S|}, v^{|S|}$, where $I^k = (b^k, e^k)$ is the k -th interval counted from the bottom of S and v^k its value. There are n markers and each must climb down at most $|S|$ stairs, which takes $O(n^2/\varepsilon)$ time in total. The minimum over the e_i can be updated in $O(n)$ time whenever a new interval is pushed on the stack, i.e. $|S|$ times. Thus phase I takes $O(n^2/\varepsilon)$ total time and uses $O(n/\varepsilon)$ space. Since the right endpoints of the intervals on S are non-decreasing, phase II takes only constant time per interval, i.e. $O(n/\varepsilon)$ time in total. For the proof of the approximation factor, see [2]. $\square$

Throughput maximization with a stretch factor of 2 is equivalent to 1d-1SH: for each input point p_i of the labeling problem, we define a job J_i by setting its weight to that of p_i , its length to the interval length $\ell(p_i)$ of p_i , and its execution window to $[x(p_i) - \ell(p_i), x(p_i) + \ell(p_i)]$. Then the length of the execution window of each job is exactly twice the job length, so we can solve 1d-1SH near-optimally.

Theorem 2 *The weighted sliding problems 1SH, 1SV, 2SH, 2SV, and 4S can be $(2 + \varepsilon)$ -approximated in $O(n^2/\varepsilon)$ time using $O(n/\varepsilon)$ space.*

The difficulty in proving Theorem 2 comes only into play with vertical sliding. Even then, however, line stabbing and ε -2PA can be applied without modification. It is only the analysis that becomes more involved, see [10].

4 An Exact Scheduling Algorithm for a Bounded Number of Different Weights

The following two sections deal with two restrictions of the problem 1d-1SH that can be solved optimally. In this section we consider the case that the number of different weights is bounded. We state our result in the language of scheduling.

Theorem 3 *Let $\mathcal{J}$ be a set of n jobs $J_1, \dots, J_n$. If the stretch factor α of $\mathcal{J}$ is less than 2, the number of different weights is k , and $V_k = O(n^k)$ is the number of possible throughputs, then there is an algorithm that computes a schedule with maximum throughput in $O(nV_k \log V_k)$ time using $O(V_k)$ storage.*

Note that $V_k = nk$ if the weights are the first k integers. If additionally k is considered a constant, throughput maximization can be solved in $O(n^2 \log n)$ time. The same holds for 1d-1SH with the slight restriction that the intervals cannot use the full window for sliding, but only its interior. Thus we receive factor-2 approximation algorithms for all (in the above sense restricted) slider models as in the proof of Theorem 2. We do not know how to relax the restriction $\alpha < 2$ to $\alpha \leq 2$.

Proof. We use dynamic programming with a table T of size $V_k + 1$. There is an entry $T[v]$ for each possible throughput v that stores the finish time of the *leftmost schedule* with throughput v . $T[0]$ is a dummy entry. The leftmost schedule with throughput v is the schedule that has the earliest finish time among all schedules with throughput v .

First we build a binary tree over all possible throughputs. The leaves are linked to the entries of the dynamic programming table. We fill the table in order of increasing throughput. Initially all entries have value $-\infty$. We compute $T[v]$ as follows.

A job J_i is given by its weight w_i , release time r_i , deadline d_i and length l_i . For each job we check whether $w_i \leq v$ and if so, whether J_i can be scheduled to the right of $T[v - w_i]$. If yes, we schedule J_i as early as possible. If at least one job among $J_1, \dots, J_n$ is scheduled, we set $T[v]$ to the earliest finish time among these at most n schedules, otherwise to $+\infty$.

The maximum throughput $v_{\max}$ of $\mathcal{J}$ is the largest v for which $T[v] < \infty$. The corresponding schedule s can be computed by using an additional entry $L[v]$ that stored the index of the last job that has been scheduled when computing $T[v]$. Let $i = L[v_{\max}]$. Then s consists of job J_i scheduled at $(T[v] - l_i, T[v])$ and the jobs that can be computed recursively by investigating $T[v_{\max} - w_i]$.

The running time is $O(nV_k \log V_k)$ since for each of the V_k entries and for each of the n jobs we have to do at most one look-up in the binary tree over T , and each look-up takes $O(\log V_k)$ time.

The proof of correctness is by induction over the throughput, see [10]. $\square$

5 An Approximation Scheme for Unit-Square Labels

This section deals with a special case of the problem 1d-1SH where all intervals have unit length. This corresponds to labeling points with unit squares. We address this special case since the more general problem of designing a PTAS for unit-height rectangles seems to be difficult in the weighted case, and is solved in the unweighted case [12].

The idea of our algorithm for 1d-1SH for unit-length intervals is to discretize the continuous space of label positions of each point to a small number of label candidates such that each optimal solution of the continuous problem corresponds to a solution of the discrete problem that has the same weight. Then Lemma 6 solves the problem.

The algorithm is as follows. Sort the n different input points from left to right and denote them by $p_1, p_2, \dots, p_n$ in this order. Clearly p_1 can do with only one label candidate, namely its leftmost, $[x_1 - \ell_1, x_1]$. For p_i ($i > 1$) we also take its leftmost candidate but additionally all the endpoints of the candidates of p_{i-1} that fall into the label window $[x_i - \ell_i, x_i + \ell_i]$ of p_i . Note that other than in the general case at most *one* of the two endpoints can do that for each candidate of p_{i-1} . Intuitively speaking, we do not have to worry about the candidates of points p_j with $j < i - 1$ since their endpoints either do not fall into the window of p_i or, if they do, they also fall into that of p_{i-1} and thus will be taken into account. Hence p_i has at most i candidates. Lemma 6 yields

Lemma 8 *For unit-length intervals the problem 1d-1SH can be solved in $O(n^2 \log n)$ time using $O(n^2)$ space.*

The proof is elementary, see [10]. Combining the above discretization for 1d-1SH with line stabbing and the dynamic-programming algorithm of Agarwal et al. [1] gives us a PTAS for labeling points with sliding unit-square labels.

Corollary 1 *Given a set P of n points and an integer $k \geq 1$ there is an algorithm that finds a 1S-labeling for P whose weight is at least $\frac{k}{k+1}$ times the maximum weight. The algorithm takes $O(n^{4k-2})$ time and uses $O(n^{4k-2})$ space.*

6 An Approximation Algorithm for Labeling Instances with Bounded Height Ratio

In this section, we label points with weighted sliding labels whose heights may vary, but only within a constant factor. For each input point p in P we are given its label length $\ell(p)$ and height $h(p)$. Let β be the ratio of maximum and minimum label height, i.e. $\beta = \max_{p \in P} h(p) / \min_{p \in P} h(p)$. Usually a map or diagram uses only a small number of different fonts whose sizes do not vary too much, thus β is relatively small in practice and it is worthwhile to design an algorithm whose approximation factor depends on β .

For the case of fixed-position models and arbitrary label heights, algorithms for (weighted) MIS in rectangle intersection graphs can be used. Agarwal et

al. achieve an approximation factor of $O(\log n)$ in the unweighted case [1], Iturriaga explains how the ideas of Agarwal et al. can be extended to handle weighted rectangles as well [8]. Recently Erlebach et al. have improved this result for weighted squares by giving a PTAS [5].

Strijk and van Kreveld [11] presented a practical factor- $(1+\beta)$ approximation algorithm for labeling *unweighted* points with sliding labels. Their algorithm takes $O(rn \log n)$ time, r the number of different label heights. We present a new approximation algorithm for the weighted case. Its runtime is independent of r and its approximation factor is better than that of [11] for $\beta > 11$.

The idea is to partition P into $\lceil \log_2 \beta \rceil$ groups such that label heights within a group differ at most by a factor of 2. By the pigeon-hole principle there is a group whose maximum-weight labeling W is at least $1/\lceil \log_2 \beta \rceil$ times the maximum weight of a labeling for P . We combine line stabbing with the 1d-algorithms of Section 3 to compute a labeling of weight at least $W/3$ or $W/(3+\varepsilon)$ for each P_j . For the details refer to [10].

Theorem 4 *Let P be a set of n points, each with a label, and let β be the ratio of maximum to minimum height among these labels. Then the maximum-weight labeling for P can be $3\lceil \log_2 \beta \rceil$ -approximated in $O(kn \log n)$ time given a fixed-position model with at most k positions per point and $(3+\varepsilon)\lceil \log_2 \beta \rceil$ -approximated in $O(n^2/\varepsilon)$ time for slider models.*

Acknowledgments

We thank Otfried Cheong and Kyung-Yong Chwa to give us a chance to discuss this problem at HKUST and KAIST, respectively. We thank Jae-Hoon Kim for contributing an idea to the proof of Theorem 4.

References

1. P. K. Agarwal, M. van Kreveld, and S. Suri. Label placement by maximum independent set in rectangles. *Comp. Geom.: Theory and Appl.*, 11:209–218, 1998. 611, 612, 616, 617, 620, 621
2. P. Berman and B. DasGupta. Multi-phase algorithms for throughput maximization for real-time scheduling. *J. Comb. Optimization*, 4(3):307–323, Sept. 2000. 617, 618
3. B. Chazelle and 36 co-authors. The computational geometry impact task force report. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223, pages 407–463. AMS, 1999. 610
4. J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transact. on Graphics*, 14(3):203–232, 1995. 610
5. T. Erlebach, K. Jansen, and E. Seidel. Polynomial-time approximation schemes for geometric graphs. In *Proc. SODA'01*, pages 671–679, Washington, 2001. 611, 621

6. M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 281–288, 1991. 610
7. J. Y. Hsiao, C. Y. Tang, and R. S. Chang. An efficient algorithm for finding a maximum weight 2-independent set on interval graphs. *IPL*, 43(5):229–235, 1992. 616
8. C. Iturriaga. *Map Labeling Problems*. PhD thesis, University of Waterloo, 1999. 611, 621
9. J. L. Morrison. Computer technology and cartographic change. In D. Taylor, ed., *The Computer in Contemporary Cartography*. J. Hopkins Univ. Press, NY, 1980.
10. S.-H. Poon, C.-S. Shin, T. Strijk, and A. Wolff. Labeling points with weights. TR 7/2001, Universität Greifswald, May 2001. Available at <http://www.math-inf.uni-greifswald.de/map-labeling/papers/pssw-lpw-01t.ps.gz>. 612, 618, 619, 620, 621
11. T. Strijk and M. van Kreveld. Practical extensions of point labeling in the slider model. In *Proc. 7th ACM Symp. on Advances in GIS*, pages 47–52, 1999. 621
12. M. van Kreveld, T. Strijk, and A. Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999. 611, 612, 613, 615, 616, 620
13. A. Wolff and T. Strijk. The Map-Labeling Bibliography. <http://www.math-inf.uni-greifswald.de/map-labeling/bibliography/>, 1996. 610

Small Convex Quadrangulations of Point Sets

David Bremner¹ *, Ferran Hurtado² **,
Suneeta Ramaswami³ ***, and Vera Sacristán² †

¹ Faculty of Computer Science, University of New Brunswick
bremner@unb.ca

² Dep. Matemàtica Aplicada II, Universitat Politècnica de Catalunya
{hurtado,vera}@ma2.upc.es

³ Dep. Computer Science, Rutgers University, Camden
rsuneeta@camden.rutgers.edu

Abstract. In this paper, we give upper and lower bounds on the number of Steiner points required to construct a strictly convex quadrilateral mesh for a planar point set. In particular, we show that $3\lfloor \frac{n}{2} \rfloor$ internal Steiner points are always sufficient for a convex quadrangulation of n points in the plane. Furthermore, for any given $n \geq 4$, there are point sets for which $\lceil \frac{n-3}{2} \rceil - 1$ Steiner points are necessary for a convex quadrangulation.

1 Introduction

Discrete approximations of a surface or volume are necessary in numerous applications. Some examples are models of human organs in medical imaging, terrain models in GIS, or models of parts in a CAD/CAM system. These applications typically assume that the geometric domain under consideration is divided into small, simple pieces called *finite elements*. The collection of finite elements is referred to as a *mesh*. For several applications, *quadrilateral/hexahedral* mesh elements are preferred over triangles/tetrahedra owing to their numerous benefits, both geometric and numerical; for example, quadrilateral meshes give lower approximation errors in finite element methods for elasticity analysis [1,3] or metal forming processes [12]. However, much less is known about quadrilateralizations and hexahedralizations and in general, high-quality quadrilateral/hexahedral meshes are harder to generate than good triangular/tetrahedral ones.

Whereas triangulations (tetrahedralizations) of polygons, two-dimensional (2D) and three-dimensional (3D) point sets, and convex polyhedra always exist (not so for non-convex ones [21]), quadrilateralizations do not. Hence it

* Partially supported by an NSERC Individual Research Grant

** Partially supported by CUR Gen. Cat. 1999SGR00356 and Proyecto DGES-MEC PB98-0933.

*** Partially supported by a Rutgers University Research Council Grant URF-G-00-01.

† Partially supported by CUR Gen. Cat. 1999SGR00356 and Proyecto DGES-MEC PB98-0933.

becomes necessary to add extra points, called *Steiner points*, to the geometric domain. This raises the issue of bounding the number of Steiner points, and hence the mesh complexity, while also providing guarantees on the quality of element shape. A theoretical treatment of this topic has only recently begun [4,7,9,16,17,18,19]. Some work on quadrilateralizations (also known as *quadrangulations*) of restricted classes of polygons has been done in the computational geometry community [8,13,14,20]. However, there are numerous unresolved questions. For example, even the fundamental question of deciding if a 2D set of points admits a convex quadrangulation without the addition of Steiner points, is unsolved. A survey of results on quadrangulations of planar sets appears in [22].

Any planar point set can be quadrangulated with at most one Steiner point, which is required only if the number of points on the convex hull is odd [7]. For planar simple n -gons, $\lfloor n/4 \rfloor$ internal Steiner points suffice to quadrangulate the polygon [19]. In both cases, the quadrilaterals of the resulting mesh will be, in general, non-convex. However, for many applications, an important requirement is that the quadrangulation be *strictly convex*, i.e., every quadrilateral of the mesh must have interior angles strictly less than 180° . A natural problem then is to construct strictly convex quadrilateral meshes for planar geometric domains, such as polygons or point sets, with a bounded number of Steiner points. Some results on convex quadrangulations of planar simple polygons are known. For example, it was shown in [10] that any simple n -gon can be decomposed into at most $5(n-2)/3$ strictly convex quadrilaterals and that $n-2$ are sometimes necessary. Furthermore, circle-packing techniques [4,5,15] have been used to generate, for a simple polygon, quadrilateral meshes in which no quadrilateral has angle greater than 120° . For planar point sets, experimental results on the use of some heuristics to construct quadrangulations with many convex quadrangles appear in [6]. In [11], it is shown that a minimum weight convex quadrangulation (i.e. where the sum of the edge lengths is minimized) can be found in polynomial time for point sets constrained to lie on a fixed number of convex layers.

In this paper, we study the problem of constructing a strictly convex quadrilateral mesh for a planar point set using a bounded number of Steiner points. If the number of extreme points of the set is even, it is always possible to convex-quadrangulate the set using Steiner points which are all internal to the convex hull. If the number of points on the convex hull is odd, the same is true, assuming that in the quadrangulation we are allowed to have exactly one triangle. We provide upper and lower bounds on the number of Steiner points required for a convex quadrangulation of a planar point set. In particular, in Section 2, we prove that $3\lfloor \frac{n}{2} \rfloor$ internal Steiner points are always sufficient to convex-quadrangulate any set of n points. In Section 3, we prove that for any $n \geq 4$, $\lceil \frac{n-3}{2} \rceil - 1$ Steiner points may sometimes be necessary to convex-quadrangulate a set of n points.

2 Upper Bound

Given a set S of n points in the plane, a *convex-quadrangulation* of S is a decomposition of $\text{conv}(S)$ into strictly convex quadrangles and at most one triangle, such that no cell contains a point of S in its interior. The vertices of the quadrangulation that do not belong to S are called *Steiner points*. In what follows, we treat angles of 180° as reflex.

Theorem 1. *Any set of n points can be convex-quadrangulated using at most $3\lfloor \frac{n}{2} \rfloor$ Steiner points.*

Proof. (Sketch) Any set S of n points has a path triangulation (a triangulation whose dual graph has a Hamiltonian path), which can be constructed in $O(n \log n)$ time [2,7]. Denote by t the number of triangles in any triangulation of n points with h extreme points ($t = 2n - 2 - h$). By pairing up the triangles along the path, we obtain a path quadrangulation of S with possibly one unpaired triangle. We will prove in Section 2.1 that it is always possible to convex-quadrangulate a pair of consecutive quadrangles by using at most 3 internal Steiner points. Consideration of the various possibilities for unpaired triangles and quadrangles yields the bound above. Note that the number of quadrilaterals in the quadrangulation is at most $5\lfloor \frac{n}{2} \rfloor - \frac{h}{2}$. $\square$

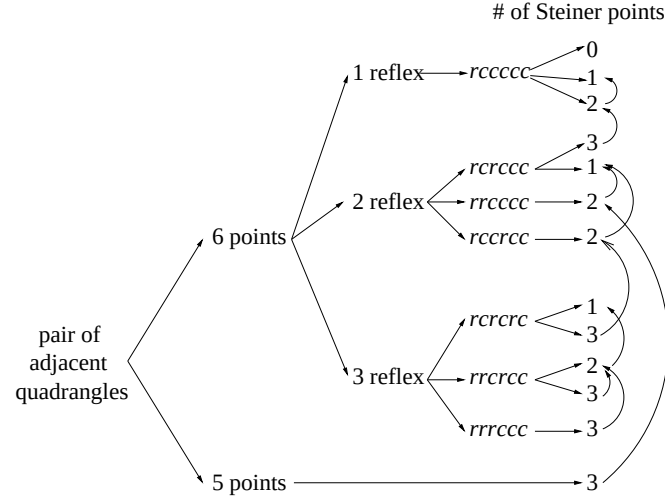
2.1 Pairing up Quadrangles

Given two points p and q , we will denote by $L(p, q)$ (resp. $R(p, q)$) the left (resp. right) open half-plane defined by the oriented line from p to q . Throughout this paper, vertices of polygons will be enumerated counterclockwise. Given a vertex v of a polygon P , we denote its successor (resp. predecessor) by v^+ (resp. v^-), and we write $\text{wedge}(v)$ to mean $L(v^-, v) \cap R(v^+, v) \cap \text{int}(P)$. If v is reflex, $\text{wedge}(v)$ will denote the locus of points (inside P) that can be connected to v forming strictly convex angles at v . If v is convex, $\text{wedge}(v)$ is the interior of the visibility region of v in P . Given three points p, q , and r , $\triangle(pqr)$ is the open triangle defined by the three points, i.e. $\triangle(pqr) = \text{int conv}(p, q, r)$. We use $\text{kernel}(P)$ to denote the kernel of the polygon. Note that $\text{int kernel}(P) = \bigcap_{v \in P} L(v, v^+)$. The following observations will prove useful below.

$$\text{int kernel}(P) = \bigcap_i \text{wedge}(v_{2i}) \quad (1)$$

$$= \bigcap_{v \text{ convex}} \text{wedge}(v) \quad (2)$$

Consider a pair of consecutive quadrangles in the path quadrangulation. They may share one edge or two edges. In the first case, their union is a hexagon, while in the second case it is a quadrangle containing a fifth point in its interior. In the rest of this section we will examine in detail how to convex-quadrangulate the union of two quadrangles. Table 1 provides a summary of all the cases and their

Table 1. Scheme of the proof

interdependencies. Most of the cases are given a mnemonic label describing the cyclic order of reflex and convex vertices around the polygon boundary and the total number of Steiner points necessary. The last column reports the number of Steiner points used in each case. The arrows on the right indicate the reductions, after adding one Steiner point, from one case to another. As is suggested by Table 1, the majority of our effort in the remainder of this section will be devoted to proving the following theorem.

Theorem 2. *Any hexagon can be convex-quadrangulated by placing at most 3 Steiner points in its interior.*

Independent Triples We call a set of vertices of a polygon *independent* if no two of them are endpoints of the same edge. In the following lemmas, let $\{a, c, e\}$ denote an independent triple for a hexagon $P = abcdef$.

Lemma 1. *If $\triangle(ace) \subset P$ then $\triangle(ace) \cap \text{wedge}(a) = \triangle(ac'e')$, where $c'e' \subseteq ce$ and $c' \neq e'$.*

Lemma 2. *If $\triangle(ace) \subset P$ then $\text{wedge}(a) \cap \text{wedge}(c) \cap \triangle(ace) \neq \emptyset$.*

Lemma 3. *If $\triangle(ace) \subset P$ then $\triangle(ace) \cap \text{wedge}(a^-) \cap \text{wedge}(a^+) \neq \emptyset$.*

Lemma 4. *If P is starshaped and $\triangle(ace) \subset P$, then one Steiner point suffices to convex-quadrangulate P .*

Lemma 5. *If c does not see e , and a is the only reflex vertex other than possibly c or e , then $\text{wedge}(a) \cap \text{wedge}(c) \neq \emptyset$, and $\text{wedge}(a) \subset L(a, c)$.*

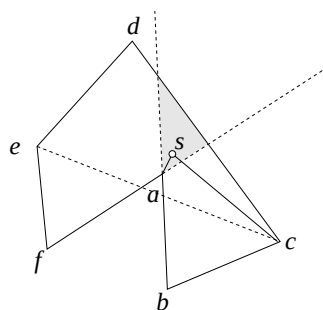


Fig. 1. One Steiner point reduces case *rrccccc-2* to case *rrccccc-1*

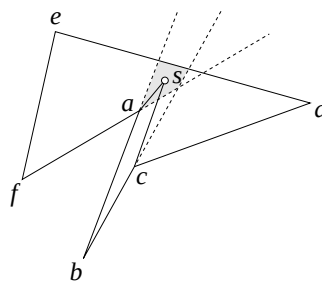


Fig. 2. One Steiner point reduces the problem to the one reflex vertex case

Proof of Theorem 2. A hexagon may have zero, one, two, or three reflex vertices; we consider each of these cases in turn. Because of space limitations, we present only a summary of the argument for several of the cases.

Hexagon with no reflex vertices. In this case, the hexagon can be trivially decomposed into two convex quadrangles without using any Steiner points.

Hexagon with one reflex vertex. Suppose w.l.o.g. that vertex a is reflex.

1. If $d \in \text{wedge}(a)$ then no Steiner points are needed. Connecting d with a will produce a convex quadrangulation of the hexagon.
2. If $d \notin \text{wedge}(a)$, then d must lie on one side of $\text{wedge}(a)$ and at least one of e or c , w.l.o.g. e , must lie on the same side.
 - 2.1. (*rrccccc-1*) If $ce \subset P$, by Lemma 4 one Steiner point is sufficient.
 - 2.2. (*rrccccc-2*) If c and e do not see each other, two Steiner points are enough. Placing a Steiner point s in $\text{wedge}(a)$ and connecting it to a and c decomposes the hexagon into a quadrangle $abcs$ and a hexagon $ascdef$ (see Figure 1). The hexagon $ascdef$ is as in the previous case *rrccccc-1*.

Hexagon with two reflex vertices. There are several different cases, depending on the relative positions of the two reflex vertices in the polygon boundary.

1. (*rrcrrcc*) Suppose that the two reflex vertices (w.l.o.g. a and c) are separated by a convex vertex of the polygon.
 - 1.1. (*rrcrrcc-1*) If both a and c can see e , then one Steiner point is enough. Note that since e is convex $\triangle(ace) \subset \text{wedge}(e)$. By Lemma 2 $\text{wedge}(a) \cap \text{wedge}(c) \cap \triangle(ace) \neq \emptyset$. It follows from (1), that the hexagon is starshaped. We can then apply Lemma 4.
 - 1.2. (*rrcrrcc-3*) Otherwise, one of the reflex vertices, w.l.o.g. c , obstructs the visibility from the other reflex vertex to e . In this case 3 Steiner points suffice. By Lemma 5 $\text{wedge}(a) \cap \text{wedge}(c) \cap L(a, c) \neq \emptyset$. Place a Steiner point s in this region and connect it to a and c (see Figure 2). The remaining hexagon has only one reflex vertex s , hence can be convex-quadrangulated with at most 2 additional Steiner points.

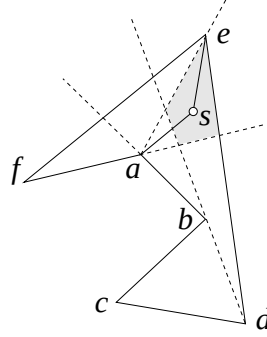


Fig. 3. One Steiner point reduces the problem to the *rccccc-1* case

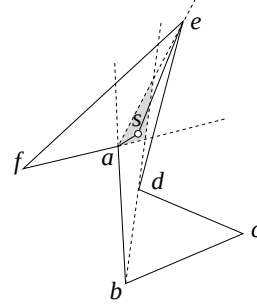


Fig. 4. One Steiner point reduces *rccccc-2* to *rccccc-1* case

2. (*rrccccc-2*) If the two reflex vertices are consecutive, then two Steiner points are always sufficient. Let a and b be the two reflex vertices. Placing a Steiner point $s \in \text{wedge}(a) \cap R(a, e) \cap L(b, d)$ and connecting s to a and e (see Figure 3) reduces this case to the *rccccc-1* case.
3. (*rccrccc-2*) We are left with the case in which there are two convex vertices between the two reflex vertices, both clockwise and counterclockwise. In this case, two Steiner points suffice. Let a and d be the reflex vertices. We claim that either the two diagonals ae and bd are internal to the polygon or ac and df are. Let us assume that ae and bd are internal diagonals (see Figure 4). Then one Steiner point s can be placed in $\text{wedge}(a) \cap R(a, e) \cap L(b, d)$. Connect s to a and e . The quadrangle $asef$ is convex. The remaining polygon is the *rccccc-1* type: s and d are its reflex vertices, and they both see b , since $s \in L(b, d)$.

Hexagon with three reflex vertices. Again, there are different situations, depending on the relative positions of the reflex vertices along the polygon boundary.

1. (*rccrccc*) We start with the case in which the reflex and the convex vertices alternate.
 - 1.1. (*rccrccc-1*) If $\triangle(ace)$ is inside the polygon and the polygon is star shaped, then one Steiner point suffices. Apply Lemma 4.
 - 1.2. (*rccrccc-3*) The region $\rho = \text{wedge}(a) \cap \text{wedge}(e) \cap R(a, e)$ must be non-empty. There are two different possibilities. If $\triangle(ace)$ is inside the polygon, then ρ is non-empty as a consequence of Lemma 2. If on the other hand one of the edges of $\triangle(ace)$, w.l.o.g. ac is obstructed, then ρ is non-empty by Lemma 5. Place a Steiner point s inside ρ . Connect s to a and e . The quadrangle $efas$ is convex. The hexagon $sabcde$ is of type *rccccc-2* since $s \in \text{wedge}(d) \cap \text{wedge}(f)$.
2. (*rrccccc*) We now study the case in which there are exactly two consecutive reflex vertices. These polygons are always star-shaped, since if a , b , and d are the reflex vertices, $\text{wedge}(f) \cap \text{wedge}(c) \cap \text{wedge}(e) \neq \emptyset$. We have two cases depending on whether e sees at least one of a and b .

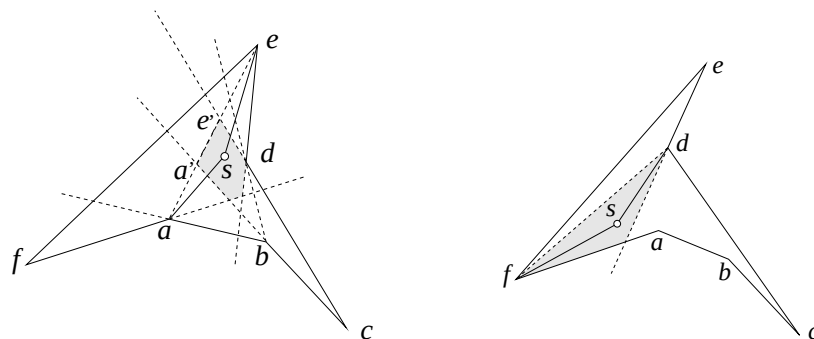


Fig. 5. One Steiner point reduces the problem to the *rrcrcc-1* case

Fig. 6. One Steiner point reduces the problem to the *rrcrcc-2* case

- 2.1. (*rrcrcc-2*) If e sees at least a , two Steiner points suffice. In particular the region $\text{kernel}(P) \cap R(a, e) \cap L(b, d)$ (see Figure 5) cannot be empty. Place a Steiner point s in the region, and connect it to a and e . The quadrangle $asef$ is convex: a is convex because $s \in \text{wedge}(a)$, and s is convex because $s \in R(a, e)$. The hexagon $abcdes$ is of the *rrcrcc-1* type because s, b, d are mutually visible (since $s \in L(b, d)$).
- 2.2. (*rrcrcc-3*) If e sees neither a nor b , then three Steiner points suffice. Placing a Steiner point s in the region $\text{wedge}(e) \cap R(f, d)$ and connecting s to f and d reduces this case to the *rrcrcc-2* case (see Figure 6).
3. (*rrrccc*) We are left with the case in which the three reflex vertices are consecutive. This case can be solved with three Steiner points. Suppose that the three reflex vertices are a, b and c . Place a Steiner point s in the region $\text{wedge}(a) \cap R(a, e) \cap L(b, e)$. Connecting s with a and e gives rise to the convex quadrangle $asef$. The remaining hexagon is of the *rrcrcc-2* type, since e sees b and c , because $s \in L(b, e)$.

This completes the (sketch of the) proof of Theorem 2. It remains to consider the case when the union of two quadrangles is not a hexagon.

Quadrangle with One Interior Point. As stated earlier, when two quadrangles share two edges, their union is a quadrangle which contains one of the vertices of the original quadrangles in its interior. We will show that three Steiner points suffice to convex-quadrangulate this polygon, thus establishing the following theorem:

Theorem 3. *Any union of two quadrangles can be convex-quadrangulated with at most three Steiner points.*

Proof. We consider here only the case where the union is not a hexagon. Let us call the four vertices of the union quadrangle r, a, b and c , where r is the only (possibly) reflex vertex. Let i be the interior point. Since only r may be

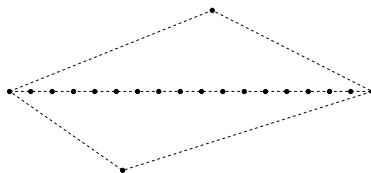


Fig. 7. The point set S has $m + 1$ points along the line, plus the top and the bottom points. Its convex hull is a quadrangle

reflex, i must see either a or c . Suppose that i sees a . Since $i \in \text{wedge}(a)$, $\text{wedge}(r) \cap L(r, i) \cap L(a, i) \neq \emptyset$. Place one Steiner point s in the region. Then the quadrangle $rais$ is convex: r is convex because $s \in \text{wedge}(r)$, i is convex because $s \in L(a, i)$, and s is convex because $s \in L(r, i)$. On the other hand, the hexagon $siabcr$ is a *rrcccc* hexagon, which can be convex-quadrangulated with two Steiner points. $\square$

Each of the cases described in this section runs in constant time, thus:

Theorem 4. *A convex-quadrangulation of n points using at most $3\lfloor \frac{n}{2} \rfloor$ Steiner points can be computed in $O(n \log n)$ time.*

3 Lower Bound

In this section we describe a particular configuration of $m + 3 \geq 4$ points which requires at least $\lceil \frac{m}{2} \rceil - 1$ Steiner points to be convex-quadrangulated. We also show a convex-quadrangulation of the set that uses close to that few Steiner points.

Description of the configuration of points: The configuration of $m + 3$ points consists of $m + 1$ points placed along a line ℓ , with one point above the line and another point below the line, such that the convex hull of the set has 4 vertices, namely the extreme points on the line and the top and bottom points (see Figure 7). We refer to the vertices on ℓ as *line vertices*. We will refer to the entire configuration as S .

Consider any strictly convex quadrangulation $\mathcal{C}$ of the set. Since all the quadrangles in $\mathcal{C}$ are strictly convex, each point on ℓ must belong to at least one edge of the quadrangulation lying strictly above the line, and at least one edge lying strictly below the line. Quadrangulation edges incident on an input point and lying above (below) ℓ will be called *upward* (*downward*) edges.

Consider two consecutive points a_1 and a_2 on ℓ with a_1 to the left of a_2 . Let u_1 be the clockwise last upward edge incident on a_1 , and let u_2 be the counterclockwise last upward edge incident on a_2 . Symmetrically, let d_1 be the counterclockwise last downward edge incident on a_1 and let d_2 be the clockwise last downward edge incident on a_2 (see Figure 8). If (a_1, a_2) is an edge of $\mathcal{C}$, then it must form one quadrangle of $\mathcal{C}$ together with u_1 and u_2 , and another one

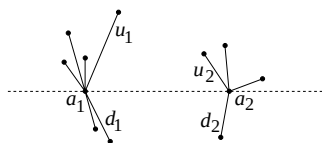


Fig. 8. Relevant upward and downward edges

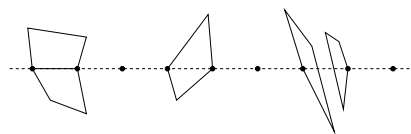


Fig. 9. Squares, diamonds and half-diamonds

with d_1 and d_2 . We call these two faces *squares*. If (a_1, a_2) is not an edge of $\mathcal{C}$, u_1 and d_1 must belong to the same quadrangle, and so must also u_2 and d_2 . If these two quadrangles are the same, we call it a *diamond*. If they are different, we call them a *pair of half-diamonds*. These three cases are illustrated in Figure 9.

Theorem 5. S needs at least $\lceil \frac{m}{2} \rceil - 1$ Steiner points to be convex-quadrangulated.

Proof. (Sketch) Consider the graph $G = (V, E)$ formed by taking the union of all the squares, diamonds and half-diamonds, together with the convex hull edges. This graph, which is a subgraph of $\mathcal{C}$, is planar and its faces consist of the squares, the diamonds, the half-diamonds, and possibly some other faces that we will call “extra faces”. Its edges are all square, diamond, half-diamond, or convex hull edges. Let q be the number of squares, d the number of diamonds and h the number of half-diamonds. We have $m = \frac{q}{2} + d + \frac{h}{2}$. Let v, e, f denote the number of vertices, edges and faces of G . Let s be the number of vertices that did not belong to the original set, i.e., the number of Steiner points in $\mathcal{C}$. Let x be the number of extra faces. We have $v \leq m + 3 + s$ (because not every Steiner point need be a vertex of G), and $f = q + d + h + x$. Since G is planar, we can apply Euler’s formula as follows:

$$s \geq e - \frac{3}{2}q - 2d - \frac{3}{2}h - x - 1$$

Now, if we can prove that

$$e \geq \frac{7}{4}q + \frac{5}{2}d + \frac{7}{4}h + x, \quad (3)$$

we will obtain that

$$s \geq \frac{q}{4} + \frac{d}{2} + \frac{h}{4} - 1 = \frac{m}{2} - 1 \geq \left\lceil \frac{m}{2} \right\rceil - 1.$$

The general scheme to establish (3) will be to partition the edges of (quadrangles in) G into three sets, and then charge each edge to the faces bounded by the edge. The classification of edges and the charging scheme are as follows:

- Line edges: edges with both endpoints on the line ℓ . Each such edge is shared by a pair of squares. Each square gets charged $1/2$.

- Steiner edges: edges with neither endpoint on the line ℓ . Each such edge charges $1/2$ to each of the faces that it bounds.
- Vertical edges: edges with exactly one endpoint on the line ℓ .
 - If a vertical edge is shared by two diamonds, each diamond gets charged $1/2$.
 - If it is shared by a diamond and an extra face, the diamond gets charged $3/4$ and the extra face gets charged $1/4$.
 - If it belongs to a square or a half-diamond, the square or half-diamond gets charged $3/8$, and the other face gets charged $5/8$.

By summing charges over faces, noting in particular that if a diamond shares a vertical edge with another diamond or half diamond, its other edge from the same line vertex shares an edge with an extra face, (3) follows. $\square$

Theorem 6. S can be convex-quadrangulated with $s \leq \lceil \frac{m+3}{2} \rceil$ Steiner points.

Proof. (Sketch) It is possible to convex-quadrangulate the given point set configuration with s Steiner points, where

$$s = \begin{cases} \frac{m}{2} + 1, & \text{if } m \equiv 0 \pmod{4} \\ \frac{m+1}{2} + 1, & \text{if } m \equiv 1 \pmod{4} \\ \frac{m}{2} + 2, & \text{if } m \equiv 2 \pmod{4} \\ \frac{m+1}{2}, & \text{if } m \equiv 3 \pmod{4} \end{cases} \leq \left\lceil \frac{m+3}{2} \right\rceil$$

A solution is presented in Figure 10. This solution can be described as follows. Let v_i , $i \in \{1, \dots, m+1\}$ be the points on the line ℓ , and t and b the top and bottom points. Place one Steiner point s below ℓ , and in $L(b, v_2) \cap R(b, v_m)$. We call the line segment $v_i v_{i+1}$ the i th virtual edge e_i . Suppose $m = 4k + r$, $0 \leq r \leq 3$. Starting from both ends of ℓ , $2k$ Steiner points p_i are placed alternately above and below every other virtual edge on ℓ . After placing $2k$ Steiner points, we are left with r “untreated” virtual edges $e'_1, e'_2, \dots, e'_r$ in the center. If $r \leq 2$, we place Steiner points as follows: one point above (resp. below) each e'_i if k is odd (resp. even). If $r = 3$ then we place point below (resp. above) e'_2 if k is odd (resp. even).

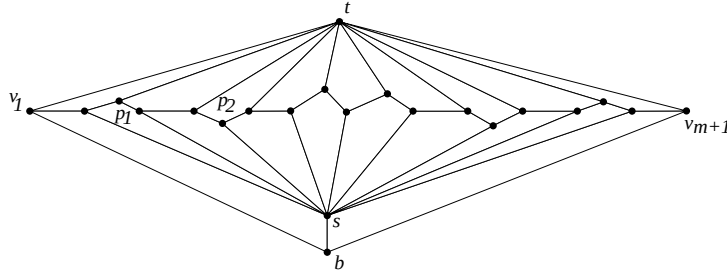


Fig. 10. A convex-quadrangulation using $\lceil \frac{m}{2} \rceil$ Steiner points

In all cases we insure the the Steiner point is within the two wedges defined by the virtual edge, s and t . The strict convexity of the quadrangles created by this procedure is ensured by placing each Steiner point in the intersection of these two wedges. $\square$

Theorem 5 uses a highly degenerate configuration, where most of the points lie on a straight line. It turns out that the same lower bound result cannot be obtained from this point configuration if it is perturbed. We now describe a perturbable (i.e. non-degenerate) point set configuration that requires at least $\frac{n}{4}$ Steiner points for a strictly convex quadrangulation.

Description of the perturbable configuration of points: Let $n = 2k$. Place k points in convex position. Place the remaining k points such that if (a_i, a_{i+1}) is an edge of the convex hull, the new point b_i must be located so that $a_{i+2} \in L(a_i, b_i)$ and $a_{i-1} \in R(a_{i+1}, b_i)$, as illustrated in Figure 11. Call this point set P .

Theorem 7. P requires at least $\frac{n}{4}$ Steiner points to be convex-quadrangulated.

Proof. By definition, each convex hull edge (a_i, a_{i+1}) must belong to one quadrangle Q_i . For Q_i to be convex and not contain any interior point, its remaining two vertices must belong to the region $G(i) = R(a_i, b_i) \cup L(a_{i+1}, b_i)$; one of these vertices may be b_i (See Figure 11). Hence, for every convex hull edge there is at least one Steiner point in region $G(i)$. Since only consecutive regions intersect, at least one Steiner point is needed for every pair of convex hull edges. $\square$

Theorem 8. P can be convex-quadrangulated with $\frac{n}{4} + 1$ Steiner points.

Proof omitted; refer to Figure 12.

4 Concluding Remarks

We have given upper and lower bounds on the number of Steiner points required to construct a convex quadrangulation for a planar set of points. Both bounds are constructive, and the upper bound yields a straightforward $O(n \log n)$ time algorithm. The obvious open problem is that of reducing the gap between the lower

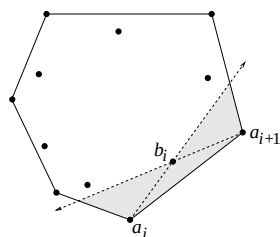


Fig. 11. The point set P

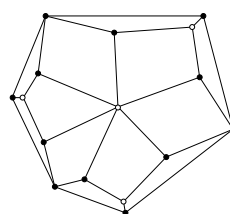


Fig. 12. A convex-quadrangulation of P that uses $\frac{n}{4} + 1$ Steiner points

and upper bounds. One way to reduce the upper bound may be by constructing a convex quadrangulation of the point set directly, rather than by converting a triangulation (by combining triangles and then quadrangles) as we do now.

References

1. D. J. Allman. A quadrilateral finite element including vertex rotations for plane elasticity analysis. *Int. J. Num. Methods Eng.*, 26:717–730, 1988. 623
2. E. Arkin, M. Held, J. Mitchell, and S. Skiena. Hamiltonian triangulations for fast rendering. In *ESA '94*, LNCS 855, pages 36–47, Utrecht, September 1994. 625
3. S.E. Benzley, E. Perry, K. Merkley, B. Clark, and K. Sjaardema. A comparison of all-hexahedral and all-tetrahedral finite element meshes for elastic and elastoplastic analysis. In *4th Int. Meshing Roundtable*, pages 179–191, 1995. 623
4. M. Bern and D. Eppstein. Quadrilateral meshing by circle packing. In *6th Int. Meshing Roundtable*, pages 7–19, 1997. 624
5. M. Bern, S. A. Mitchell, and J. Ruppert. Linear-size nonobtuse triangulation of polygons. *Disc. Comput. Geom.*, 14:411–428, 1995. 624
6. P. Bose, S. Ramaswami, A. Turki, and G. Toussaint. Experimental comparison of quadrangulation algorithms for sets of points. In *12th Euro. Workshop Comp. Geom.*, Münster, Germany, 1996. 624
7. P. Bose and G. Toussaint. Characterizing and efficiently computing quadrangulations of planar point sets. *Comp. Aided Geom. Design*, 14:763–785, 1997. 624, 625
8. H. Edelsbrunner, J. O'Rourke, and E. Welzl. Stationing guards in rectilinear art galleries. *Comp. Vis. Graphics Image Proc.*, 27:167–176, 1984. 624
9. D. Eppstein. Linear complexity hexahedral mesh generation. In *Proc. 12th ACM Symp. Comp. Geom.*, pages 58–67, 1996. 624
10. H. Everett, W. Lenhart, M. Overmars, T. Shermer, and J. Urrutia. Strictly convex quadrilateralizations of polygons. In *Proc. 4th Canad. Conf. Comput. Geom.*, pages 77–82, St. Johns, Newfoundland, 1992. 624
11. T. Fevens, H. Meijer, and D. Rappaport. Minimum weight convex quadrilateralization of a constrained point set. In *2nd CGC Workshop Comp. Geom.*, Durham, NC, USA, 1997. 624
12. B. P. Johnston, J. M. Sullivan, and A. Kwasnik. Automatic conversion of triangular finite meshes to quadrilateral elements. *Int. J. Num. Methods Eng.*, 31(1):67–84, 1991. 623
13. J. Kahn, M. Klawe, and D. Kleitman. Traditional galleries require fewer watchmen. *SIAM J. Alg. Disc. Methods*, 4(2):194–206, June 1983. 624
14. A. Lubiw. Decomposing polygonal regions into convex quadrilaterals. In *Proc. 1st ACM Symp. Comp. Geom.*, pages 97–106, 1985. 624
15. M. Bern and D. Eppstein. Polynomial-size nonobtuse triangulation of polygons. *Int. J. Comp. Geom. App.*, 2:241–255, 1992. 624
16. S. Mitchell. A characterization of the quadrilateral meshes of a surface which admit a compatible hexahedral mesh of the enclosed volume. In *5th MSI Workshop Comp. Geom.*, 1995. 624
17. S. Mitchell. Hexahedral mesh generation via the dual. In *Proc. 11th ACM Symp. Comp. Geom.*, pages C4–C5, 1995. 624
18. M. Müller-Hannemann and K. Weihe. Quadrangular refinements of polygons with an application to finite-element meshes. In *Proc. 13th ACM Symp. Comp. Geom.*, 1997. 624

19. S. Ramaswami, P. Ramos, and G. Toussaint. Converting triangulations to quadrangulations. *Comput. Geom.*, 9:257–276, 1998. 624
20. J. R. Sack. An $O(n \log n)$ algorithm for decomposing simple rectilinear polygons into convex quadrilaterals. In *Proc. 20th Allerton Conf.*, pages 64–75, October 1982. 624
21. E. Schönhardt. Über die zerlegung von dreieckspolyedern in tetraeder. *Math. Annalen*, 98:309–312, 1928. 623
22. G. Toussaint. Quadrangulations of planar sets. In *Workshop Alg. Data Struct.*, LNCS, pages 218–227. Springer-Verlag, August 1995. 624

How to Color a Checkerboard with a Given Distribution – Matrix Rounding Achieving Low 2×2 -Discrepancy

Tetsuo Asano¹ and Takeshi Tokuyama²

¹ School of Information Science, Japan Advanced Institute of Science and Technology
t-asano@jaist.ac.jp

² Graduate School of Information Sciences, Tohoku University
tokuyama@dais.is.tohoku.ac.jp

Abstract. Motivated by a digital halftoning application to convert a continuous-tone image into a binary image, we discuss how to round a $[0, 1]$ -valued matrix into a $\{0, 1\}$ binary matrix achieving low discrepancy with respect to the family of all 2×2 square submatrices (or regions). A trivial upper bound of the discrepancy is 2 and the known lower bound is 1. In this paper we shall show how to achieve a new upper bound $5/3$ using a new proof technique based on modified graph matching.

1 Introduction

Rounding real numbers into discrete values frequently occur in practice. In this paper we are interested in rounding a two-dimensional matrix of real entries in the interval $[0, 1]$ into a binary (i.e., $\{0, 1\}$ -valued) matrix. To measure the discrepancy between an input real matrix and the resulting binary matrix, we introduce a family $\mathcal{F}$ of regions (submatrices) over the matrix and define the discrepancy by the maximum difference between the sums of entries in all regions in the family. It is known [4] that we can bound the discrepancy by 1 when the family consists of all rows and all columns.

Little is known for a family consisting of small-sized two-dimensional regions. The authors proved that the problem to find an optimal binary matrix minimizing the discrepancy with an input real matrix is NP-hard even for a family of all 2×2 regions [2,3]. On the other hand, if we have two different partitions of a matrix into 2×2 square regions, we can find an optimal rounding into a binary matrix and also we can show that the discrepancy is always strictly less than 1 for the family of 2×2 regions in these two partitions.

For the family $\mathcal{F}_2$ of all 2×2 regions, based on an odd cycle argument, we can show that there is a $[0, 1]$ -valued matrix such that the discrepancy of an optimal rounding is exactly 1 for the family of all 2×2 regions. On the other hand, it is quite easy to give a rounding with discrepancy 2 by rounding each entry independently to its nearer integer. Since the error generated from each entry is at most $1/2$, the error amounts to 2 for a 2×2 region. However, it is nontrivial to improve these upper bound and lower bound. Previously, the authors claimed

a $7/4$ upper bound in a conference paper [2]; Unfortunately, the proof has not been published yet formally, since it is based on complicated case analysis.

In this paper, we give an improved upper bound $5/3$ for the family $\mathcal{F}_2$ of all 2×2 regions. We give a systematic argument as well as the improved bound. A key idea is to apply a recursive rounding procedure after discretizing input real values into several distinct values. Also, we apply a variation of matching in a graph to give the construction.

If each entry of the original matrix has a value 0.5 , it is obvious that the parity-rounding (rounding an entry into 1 if and only if the sum of its row index and its column index is even) gives a perfect (zero-error) rounding. If we consider the rounded matrix as a square $n \times n$ array on a playing board, and color a cell black (resp. white) if it corresponds to a 1 -valued (resp. 0 -valued) entry, the parity rounding gives the checkerboard pattern. Thus, what we are aiming at is a combinatorial problem to design a checkerboard pattern approximating a given general $[0, 1]$ -valued distribution instead of the special uniform distribution. We remind that the discrepancy theory (with respect to a wider class of region families) on a uniform distribution is a major topic in combinatorics and Monte-Carlo simulation [5]. Fig. 1 gives an example of rounding (its rounding error for $\mathcal{F}_2$ is 0.5) and its corresponding checkerboard pattern.

Besides its combinatorial charm, this work is motivated by an application to digital halftoning, which is an important technique to generate a binary image that looks similar to an input continuous-tone image. This kind of technique is indispensable to print an image on an output device that produces black dots only, such as facsimiles and laser printers. Up to now, a large number of methods and algorithms for digital halftoning have been proposed (see, e.g., [8,7,9]). A common criterion for the quality of output binary image is FWMSE (Frequency Weighted Mean Square Error). Simply speaking, it is to minimize the sum of all squared errors for a family of all $k \times k$ regions where error in a region is given by difference of the weighted sums in the input and output images. This criterion corresponds to L_2 distance. Our criterion based on the discrepancy is the L_∞ distance version of the problem for $k = 2$. We omit proofs of several lemmas in this version because of space limitation.

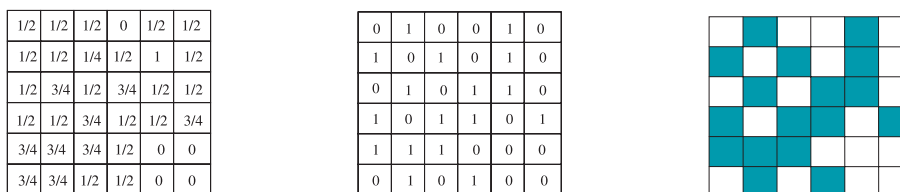


Fig. 1. A rounding and its corresponding (generalized) checkerboard

2 Matrix Rounding Problem with Related Works

Given a real number α , its *rounding* is either $\lfloor \alpha \rfloor$ or $\lceil \alpha \rceil$. Given an $n \times n$ matrix $A = (a_{ij})_{1 \leq i, j \leq n}$ of real numbers, its rounding is an integral matrix $B = (b_{ij})_{1 \leq i, j \leq n}$ such that each entry b_{ij} is a rounding of a_{ij} . There are 2^{n^2} possible roundings of a given A , and we would like to find an optimal rounding with respect to a given criterion. This is called the *matrix rounding problem*. In this paper we are interested in the case in which each entry of A is in the closed interval $[0, 1]$ and each entry is rounded to either 0 or 1. It is a special case of discrepancy problems [6].

In order to give a criterion to determine the quality of roundings, we define a distance in the space $\mathcal{A}$ of all $[0, 1]$ -valued matrices. We introduce a family $\mathcal{F}$ of regions over the $n \times n$ integer grid. Let R be a region in $\mathcal{F}$. For an element $A \in \mathcal{A}$, let $A(R)$ be the sum of entries of A located in the region R . The l_∞ distance between two elements A and A' in $\mathcal{A}$ with respect to $\mathcal{F}$ is defined by

$$\text{Dist}_\infty^{\mathcal{F}}(A, A') = \max_{R \in \mathcal{F}} |A(R) - A'(R)|.$$

Although analogously defined l_1 and l_2 distances are also popular, we are concerned with the l_∞ distance in this paper.

Once we define a distance in $\mathcal{A}$, the optimal rounding B of a given $[0, 1]$ -valued matrix A is a binary matrix in $\mathcal{A}$ that is closest to A in the sense of the above-defined distance. Such a binary matrix B is called the *optimal rounding* of A , and the distance between A and B is referred to as the *optimal rounding error*.

The supremum of the optimal rounding error $\sup_{A \in \mathcal{A}} \min_{B \in \{0,1\}^d} \text{Dist}_\infty^{\mathcal{F}}(A, B)$ is called the *inhomogeneous discrepancy* of $\mathcal{A}$ with respect to the family $\mathcal{F}$ [6]. We consider the following problem:

Discrepancy Problem: For a given region family $\mathcal{F}$, give combinatorial upper and lower bounds of the inhomogeneous discrepancy with respect to $\mathcal{F}$.

The difficulty of the above problem depends on geometric property of the family $\mathcal{F}$ of regions. We could consider the one-dimensional version of the problem, which is referred to as the sequence rounding problem. The inhomogeneous discrepancy with respect to $\text{Dist}_\infty^{\mathcal{F}}$ is less than 1 for any family $\mathcal{F}$ of intervals. On the other hand, it can be infinitesimally near to 1 even if we consider the family of all intervals of length 2. Therefore, the discrepancy problem is easily settled. Moreover, the authors showed in [2] that the optimal rounding of a sequence can be computed in $O(\sqrt{n}|\mathcal{F}|\log^2 n)$ time with respect to any given family $\mathcal{F}$ of intervals. A basic idea was a procedure to detect a negative cycle in a network.

For the matrix rounding problem, the inhomogeneous discrepancy depends on the choice of the family $\mathcal{F}$ of regions: If $\mathcal{F}$ is the set of all orthogonal regions, an $O(\log^3 n)$ upper bound and $\Omega(\log n)$ lower bound are known [6]. On the other hand, Baranyai [4] showed that the inhomogeneous discrepancy is less

than 1 if $\mathcal{F}$ consists of $2n + 1$ regions corresponding to all rows, columns and the whole matrix. Baranyai's result is applied to problems in operations research ([1] pp.171–172).

Motivated from an application in *digital halftoning*, we would like to consider the family $\mathcal{F}_k$ consisting of all $k \times k$ square regions for a small k . An $O(\log^3 k)$ upper bound and an $\Omega(\log k)$ lower bound of the inhomogeneous discrepancy can be obtained straightforwardly from the above mentioned known results.

However, it is combinatorially attractive to give better bounds for a small fixed constant k , and the problem seems to be highly nontrivial even for $k = 2$. Thus, we focus on the family $\mathcal{F}_2$ in this paper, and give a nontrivial $5/3$ upper bound for the inhomogeneous discrepancy.

3 Low Discrepancy Theorem for $\mathcal{F}_2$

Let $A = (a_{i,j})$ be an $n \times n$ matrix whose entries are real numbers in the interval $[0, 1]$. We denote the sum of entries $a_{i,j}, a_{i+1,j}, a_{i,j+1}, a_{i+1,j+1}$ by $A^{(2)}(i, j)$ for $1 \leq i, j \leq n-1$. Given a $\{0, 1\}$ -valued matrix B , the 2×2 discrepancy between A and B is $\max_{1 \leq i, j \leq n-1} |A^{(2)}(i, j) - B^{(2)}(i, j)|$. We prove the following theorem:

Theorem 1. *For an arbitrary $[0, 1]$ -valued matrix A there exists a $\{0, 1\}$ -valued matrix B such that 2×2 discrepancy between A and B is at most $5/3$.*

The following is a key lemma:

Lemma 1. *If each entry of A is among $0, 1/4, 1/2, 3/4$, and 1 , then there exists a $\{0, 1\}$ -valued matrix B such that $|A(R) - B(R)| \leq 5/4$ holds for every 2×2 region R .*

We first derive the theorem assuming the lemma is true. Let α is an upper bound of the discrepancy. Given A , we construct the matrix $C = (c_{i,j})$ where $c_{i,j} = a_{i,j} - \lfloor 4a_{i,j} \rfloor / 4$. Thus, $4C$ is a $[0, 1]$ -valued matrix. We have a rounding D of $4C$ with discrepancy less than α . Consider the matrix $H = A - C + (D/4)$. It is easy to observe that each entry of H is among $0, 1/4, 1/2, 3/4, 1$. Hence, we have a rounding B of H such that 2×2 discrepancy between B and H is at most $5/4$. Thus, the discrepancy between B and A is less than $(5 + \alpha)/4$. We continue this argument to have a recursion $\alpha \leq (5 + \alpha)/4$, and hence $\alpha \leq 5/3$.

3.1 Basic Observations

Let $A = (a_{i,j})$ be a matrix in which each entry has a discrete value among $0, 1/4, 1/2, 3/4$ and 1 . An entry is called *large* (*small*, respectively) if its value is $3/4$ or 1 ($1/4$ or 0 , respectively). The entries with the values $1/2$ are called *medium* entries. A entry is indicated by a symbol L (S , respectively) if its value is $3/4$ ($1/4$, respectively). A medium entry $a_{i,j}$ is indicated by either m or M according to its parity, that is, m if $i + j$ is even, and M otherwise. Thus, two m entries are arranged diagonally or offdiagonally but never be aligned horizontally or vertically. We often indicate an integral entry (0 or 1) of A by I .

A rounding B of A is called a *tame rounding* if it satisfies the following conditions:

- (1) Every large entry is rounded to 1. (2) Every small entry is rounded to 0.

We basically consider tame roundings. Thus, our control is just rounding medium entries. Indeed, this is a little cheating, since we will flip some S or L entries in the final stage of the construction. However, until then, we only consider tame roundings. Given a rounding B , a 2×2 region (rigid submatrix) R is called a *violating region* if $|B(R) - A(R)| > 5/4$. Otherwise, it is called a *safe region*. The following lemmas are elementary:

Lemma 2. *Let R be a 2×2 region in a given matrix. Then, R is a safe region for any tame rounding if (i) R has at most one medium entry, or (ii) R has both a large entry and a small entry.*

Lemma 3. *If a 2×2 region R has at least one medium entry and at least one S (L , resp.) entry, then R is safe as far as the medium entry is rounded to 1 (0, resp.).*

Lemma 4. *If a 2×2 region R has two medium entries, then R is safe as far as the two medium entries are rounded to different binary values, one to 0 and the other to 1.*

Lemma 5. *If three entries characterized as SmS or SMS (LmL or LML , resp.) are aligned horizontally or vertically in order, any 2×2 region containing two of them is safe as far as the middle medium entry is rounded to 1 (0, resp.).*

We call a medium element a *sandwiched element* if it is between two S or two L elements on a row or a column as in the above lemma.

3.2 Proof of Lemma 1

We are now ready to prove our main lemma 1 which guarantees that we can always round a matrix A consisting of $0, 1/4, 1/2, 3/4$ and 1 into a binary matrix so that the rounding error is between $-5/4$ and $5/4$ for any 2×2 region in the matrix.

We first round all the sandwiched elements so that those between two S elements is turned into 1, and those between two L elements is turned into 0. It may happen that an element is sandwiched by both two S elements and two L elements (vertically and horizontally), where we flip the element into 0. We finalize the rounded values of sandwiched elements as above. For simplicity, the finalized sandwiched elements are denoted by F . From Lemma 5, no region containing an F element can become a violating region.

We next apply *parity rounding*, which rounds m to 0 and M to 1 for the rest of medium elements. Table 1 summarizes the error caused by pairs of elements in the parity rounding. We omit pairs containing F elements, since every region containing an F element is safe.

Neither the pattern mm nor MM can occur vertically or horizontally in a 2×2 region. Thus, from the above table, violating regions are characterized by

Table 1. Error caused by pairs in the parity rounding

error = ± 1	$\pm 3/4$	$\pm 1/2$	$\pm 1/4$	0
mm, MM	Sm, LM	$0m, 1m, SS, 1M, 0M, LL$	$0S, 1S, SM, Lm, 0L, 1L$	$00, mM, SL, 11$

$\{S, m, S, m\}$ and $\{L, M, L, M\}$ where two medium entries are arranged diagonally or offdiagonally. If there is no such region, then the parity rounding gives us a rounding with discrepancy bounded by $5/4$.

The parity rounding is an intermediately stage. In the subsequent process, if we flip m to 1, we denote it by m^* , and if we flip M to 0, we denote it by M^* . We consider a violating region R consisting of S and m entries. We can symmetrically treat a region consisting of L and M entries.

R is either $\begin{pmatrix} S & m \\ m & S \end{pmatrix}$ or its rotated pattern $\begin{pmatrix} m & S \\ S & m \end{pmatrix}$ because of the parity condition. We flip at least one of the two m entries to make it safe. Such a flipped entry is denoted by m^* .

We consider the first case where two S entries are in the main diagonal position. It is easy to adapt the following argument to the second case above. Suppose we flip the m -entry in the first row. This flipping may cause side-effect. From Lemma 3, any 2×2 region containing the S entry and flipped medium entry m^* is always safe. Thus, we have to worry about only the region R_1 which intersects $R = R_0$ only at the flipped entry m^* . If the region R_1 is safe for the new rounding, then we stop any further flipping. We call R_1 the *sink region* of the flipping sequence and the region R the *source region*.

Note that there are many violating regions in the parity rounding, and this safe region might become violating again due to side-effect if we try to resolve other violating regions. We ignore such interaction for the time being. If a region is always safe once the flipping is done, it is called a *guarded region*.

Observation 2 *A region containing m^*M^* , Sm^* or LM^* is a guarded region, whereas a region containing a row or column of Im^* or IM^* is safe but not guarded, where I is an integral entry.*

We note that the source region is guarded because of the above observation. If R_1 is not safe, we have to continue flipping one of medium entries in R_1 . Here note that if R_1 has no other medium entry then Lemma 2 guarantees that R_1 is safe for any tame rounding. The region R_1 contains the flipped entry m^* at its lower left corner.

We can observe that R_1 is violating only if it is one of the following patterns: $\begin{pmatrix} M & \hat{L} \\ m^* & M \end{pmatrix}$, $\begin{pmatrix} L & L \\ m^* & M \end{pmatrix}$ and $\begin{pmatrix} M & L \\ m^* & L \end{pmatrix}$, where $\hat{L}$ is 0, 1, or L . In the first case we have two M entries which can be flipped into M^* (flipping one of them suffices to make R_1 safe but flipping both of them still makes R_1 safe). Thus, we have two possible flipping sequences branched from a flipping from R_1 . In the remaining

$$\begin{array}{cccc}
M & \hat{L} & M^* & \hat{L} \\
S & m^* & M^* & S \\
m & S & m & S
\end{array}
\quad
\begin{array}{cccc}
L & L & M^* & L \\
S & m^* & M^* & S \\
m & S & m & S
\end{array}$$

Fig. 2. Possible situations after flipping an entry of R_1 to remove side-effect

$$\begin{array}{ccccccc}
M & \hat{L} & L & L & L & & \\
S & m^* & M^* & m^* & M^* & m^* & \dots \\
m & S & S & S & S & S &
\end{array}
\quad
\begin{array}{ccccccc}
M & \hat{L} & L & L & & & \\
S & m^* & M^* & m^* & M^* & b & \\
m & S & S & S & a & S &
\end{array}$$

Fig. 3. Forced consecutive flipping operations (left), and its tail region (right)

two cases we have only one M entry to flip it into M^* . Possible situations are shown in Fig. 2.

The flipping(s) in R_1 may cause another side-effect, that is, a safe region may become violating by the flip. Such a region R_2 is characterized again by that intersecting only at the flipped M^* entry in R_1 .

Without loss of generality, we consider the case where R_1 has an M^* entry at its lower-right corner, and R_2 intersects with R_1 at the entry. Due to a similar argument, the region R_2 must be $\begin{pmatrix} M^* & m \\ m & \hat{S} \end{pmatrix}$ or $\begin{pmatrix} M^* & S \\ m & S \end{pmatrix}$ or $\begin{pmatrix} M^* & m \\ S & S \end{pmatrix}$, where $\hat{S}$ is 0, 1 or S . We can observe that the second case cannot happen; indeed, the m entry below M^* is sandwiched by two S entries, since if we write down both R and R_2 , we have $\begin{pmatrix} S & m^* & M^* & S \\ m & S & m & S \end{pmatrix}$. Thus, it should be $\begin{pmatrix} S & m^* & M^* & S \\ m & S & F & S \end{pmatrix}$ in truth, and R_2 is safe.

In the first case, if $\hat{S} = S$, we can similarly see that the m entry left to the S entry must be an F entry. Thus, we assume that the $\hat{S}$ is an integral element. We can stop this flipping operation by just flipping the m entry below the M^* entry. In this case, the sequence bends, and we call the sequence has a bending end. (Otherwise, we call it has a straight end). Then, we have a pair $(m^*, \hat{S})$ with the S entry in R , and thus after the flip any 2×2 region containing the last flipped entry is safe, since m^* element is sandwiched by $\hat{S}$ and S , and the rounding error of $(\hat{S}, m^*)$ is 0.5. Thus, no region containing the m^* element is violated in the current rounding, since neither MM nor mm appears as a row in a 2×2 region of the parity rounding. We remark that the region (we also call it the sink region of the flipping sequence) containing $(\hat{S}, m^*)$ and other two entries below them might become a violated region due to side-effect caused by resolving other violated region.

For the third case, the flipping operations may continue only when the new region contains exactly two medium entries guarded by S and L from both sides. Fig. 3 depicts a typical situation where consecutive flipping operations are forced.

Lemma 6. *Whenever we are forced to flip medium entries consecutively, they are aligned horizontally or vertically without any bend except at the last flip.*

Proof We have already seen that forced consecutive flipping sequence can proceed straight horizontally or vertically. So, it suffices to show that it never bends (except the last flip). Without loss of generality we consider the situation shown in the right pattern in Fig. 3. Let R_i be the 2×2 region intersecting only at the last flipped entry. Then, by the similar argument, the diagonal entry of R_i must be S . To change the flipping direction, the entry a just below M^* must be a medium entry m . What happens when we flip the m entry into m^* ? The region we have to worry about is the one containing S and m^* in its upper row. Since the lower row of the region cannot be MM , Mm^* , or m^*m^* because of the parity condition and our assumption that there is no other flipping sequence. This means the rounding error for the lower row never exceeds $-3/4$ and thus the region is safe. Therefore, we can stop the flipping sequence here at the position a in the right pattern of Fig. 3. $\square$

So far we have considered each violating region independently. Next we shall consider interaction among flipping sequences from different violating regions. Let us examine the safe but unguarded regions caused by a flipping sequence.

Lemma 7. *If a safe region is unguarded, it can become unsafe because of side-effect by other flipping sequences if it contains an I entry and three medium entries.*

Proof From Observation 2, we have a flipped medium entry and an I in a column. Thus, the only possibility that it becomes unsafe is that it has three medium entries and they are flipped into a same binary value. $\square$

Usually, only the sink region is the (possible) unguarded region containing a flipped element. Unfortunately, there are some exceptional cases, where the sequence stops as one of the patterns in Fig. 4. In each of the cases, the entries I and M^* in the bold letters cause a problem. Indeed, the pair I and M^* has error 0.5, and it is adjacent to m , M . Thus, it becomes violated if the M entry is flipped as a side effect of another flipping sequence; thus, it is a unguarded region that is not the sink region. If the original sink region itself is guarded, we regard the unguarded region as the sink region of the sequence; otherwise, we call the region a *subsink*, and a flipping sequence containing a subsink is called a *two-headed flipping sequence*. Patterns in Fig. 4 and their transformed analogues (i.e., figures obtained by rotating or reflecting, and/or exchanging every M with m and S with L) exhaust patterns of two-headed flipping sequences. We sometimes call *single-headed flipping sequences* for other sequences. For simplifying the subsequent argument, if the two-headed flipping sequence is ended with a bending end, and can flip the other medium entry (if exists) in the straight direction to have a single-headed flipping sequence, we take that choice instead of the two-headed flipping sequence.

Lemma 8. *There are either at least two single-headed flipping sequences or at least two two-headed flipping sequences from a given source region.*

$$\begin{array}{ccccc}
M & \mathbf{I} & M & & M & m & M & & M & \mathbf{I} & M \\
S & m^* & \mathbf{M}^* & m & & M & \mathbf{I} & \mathbf{M}^* & I & S & m^* & \mathbf{M}^* & m \\
m & S & & M & S & m^* & M^* & m^* & & m & S & m^* & I \\
& & & & m & S & S & S & & & & M & m
\end{array}$$

Fig. 4. Patterns causing two-head flipping sequences

Proof As seen in Fig. 4, a two-headed flipping sequence branches from

$$\begin{array}{c}
M \quad I \\
S \quad m^* \quad M^* \\
m \quad S
\end{array}$$

and hence there are other paths which flip M in the top row or flip m in the bottom row. Thus, we have at least three paths, and hence the lemma holds. $\square$

Lemma 9. *If both of subsink and sink regions are violated in a two-headed flipping sequence because of side-effect caused by other sequences, we can give flipping of some entries to resolve both of violated regions without influencing other regions.*

Proof We remark that we may destroy the tame condition here, and we emphasize that these operations are done in the final stage of the construction. Due to space limitation, the proof is given in the full version of the paper. $\square$

Definition 1 (Negative interaction of sequences). *We define that a pair of flipping sequences (originated from different source region) has negative interaction if they share an unguarded sink (or subsink) region but they have different flipped entries located in the diagonal or off-diagonal position to make the region violated.*

Lemma 10. *If we have a set of flipping sequences without negative interaction, the “union” of them creates no violating regions, where we mean “union” for the configuration obtained by flipping every medium entry that is flipped in at least one of the sequences.*

Proof A flipping sequence proceeds straight along a sequence of medium entries which are guarded by S and L from both sides, and all interior entries are flipped. We need not care a region containing a pair of entries with error at most 0.25, since our rounding is tame. Also, in the source region, we can flip both of the medium entries keeping it safe. Hence, we only worry about the region which is a sink region shared by more than one flipping sequences. Of course, if its only one entry is flipped, we have no problem. It is fine if two medium entries in a row or a column are simultaneously flipped, since these two have total error 0. Thus, there are diagonal or orthogonal flipped pair, and hence we have a negative interaction by definition. $\square$

Definition 2 (covering by flipping sequences). A set $\mathcal{C}$ of flipping sequences is called a good covering of the matrix if its sequences are classified into **active sequences** and **normal sequences** satisfying the following conditions: (1): An active sequence must be a two-head flipping sequence, (2) every violated region in the parity rounding becomes a source region of at least one sequence in $\mathcal{C}$, (3) each active sequence has negative interaction with normal sequences at both its sink and subsink, (4) each normal sequence can have negative interaction with only active sequences.

Lemma 11. If there exists a good covering, there exists a rounding whose discrepancy is at most 1.25.

Proof Consider the union of the flipping sequences. If there is no active sequence, there is no negative interaction, and we have no problem because of Lemma 10. An active sequence corresponds to a two-headed flipping sequence that has negative interactions at both of their sinks and subsinks. However, Lemma 9 assures that we can resolve sinks and subsinks in active sequences without influencing other regions. If a flipping sequence share its sink and/or subsink region(s) only with active sequences, we do not need to worry about the region, since it has been resolved within the active sequence. Thus, the lemma holds. $\square$

Hence, it suffices to find a good covering. We select (in an arbitrary manner) exactly either a pair of single-headed flipping sequences or a pair of two-headed flipping sequences for each source node. This is called *trimming operation*, and always possible because of Lemma 8. We will find a good covering as a subset of this trimmed set by translating the problem into a graph theoretic problem.

We construct a graph $G = (U \cup V, E \cup J)$ from our set of flipping sequences (after applying the trimming operation) as follows: The nodes in U are called *source nodes* while the nodes in V are called *sink nodes*, although G is not a bipartite graph in general. The edges in E is called *regular edges*, while those in J is called *joint edges*. For each source region R of a single-headed flipping sequence, we construct a source node $v(R)$. For each 2×2 region R , we define two sink nodes $v(R, +)$ and $v(R, -)$ in V . If there is a single-headed flipping sequence with a source region R and a sink region R' containing the final flipped entry in its diagonal (resp. off-diagonal) position, we define a regular edge e in E between $u(R)$ and $v(R', +)$ (resp. $v(R', -)$). Next, consider a source region R of two-headed flipping sequences (by our trimming operation, we have exactly two such sequences). For a two-headed flipping sequence with a source region R with a sink region R_1 and a subsink region R_2 , we define a joint edge between two sink nodes $v(R_1, \epsilon_1)$ and $v(R_2, \epsilon_2)$, where ϵ_i are determined from the position of the flipped entry in the same manner as the case of regular edges.

Let $\mathcal{R}$ be the set of all source regions of two-headed flipping sequences. In the construction of G we do not define source nodes for a region R in $\mathcal{R}$; however, corresponding joint edges are labeled by the source region R . In other words, for each region R in $\mathcal{R}$, we have a subset $\pi(R)$ of E consisting of two edges labeled by R .

We indeed consider the graph G together with $\mathcal{R}$ and the labeling function π (from $\mathcal{R}$ to the set of doubletons in E). We denote $[G, \mathcal{R}, \pi]$ for the triple.

Lemma 12. *In the graph G , the node degree of a source node is two, and the node degree of a sink node is at most two.*

Proof The first statement follows from definition. The proof of the second statement is omitted in this version. $\square$

Consider a three coloring (into red, blue, and white) of P , where the blue color is only used for some joint edges. Intuitively, blue edges correspond to active flipping sequences, and red edges correspond to normal flipping sequences in a good covering.

Definition 3. *A coloring of G is a good coloring if (1) each source node is adjacent to at least one red or blue edge, (2) at most one red edge is incident to a sink node, and each blue (joint) edge is adjacent to two red edges.*

Definition 4 (covering coloring). *A covering coloring of $[G, \mathcal{R}, \pi]$ is a good coloring of G satisfying that at least one edge of $\pi(R)$ is colored either red or blue for each $R \in \mathcal{R}$.*

To get intuition, if there is no joint edge, G is a bipartite graph, and containing a matching of size $|U|$ because of Hall's SDR theorem and Lemma 12. Thus, we have a good coloring (automatically a covering coloring if there is no joint edge) by coloring matching edges into red. We want to extend this fact to the general case, since we have the following lemma:

Lemma 13. *If G has a covering coloring, we have a good covering of the matrix by flipping sequences.*

Proof We consider the flipping sequences associated with the red edges and blue edges in the coloring. Since each source node is covered by such an edge, every source region that has one-headed flipping sequences is covered. Since at least one edge in $\pi(R)$ for a source node R having two-headed flipping sequence is colored into red or blue, such an R is also covered. The good coloring condition assures that the set of flipping sequences is a good covering. $\square$

Lemma 14. *The triple $[G, \mathcal{R}, \pi]$ has a covering coloring.*

Proof Let P be a connected component of G . Since the maximum node degree of G is two, P is either a cycle or a path. If it is a path, its end vertices must be in V , since node degree of a vertex in U must be two. The *critical edges* of P is (1) none if it has at most one joint edge, (2) the leftmost joint edge and the rightmost joint edge if P is a path with two or more joint edges, (3) all joint edges if P is a cycle with two or more joint edges.

We claim that if we fix any one of critical edges for each P , there exists a good coloring that colors all the joint edges except the fixed edges red or blue.

This claim can be constructively proved by using a greedy method. We omit details since it is routine.

Now, we consider a new bipartite graph $H = (\mathcal{R}, \mathcal{P}, F)$ where $\mathcal{P}$ corresponds to the connected components of G that are paths containing at least two joint edges. We have an arc from $R \in \mathcal{R}$ to $P \in \mathcal{P}$ if (at least) one of edges in $\pi(R)$ is in P as its critical edge. It is easy to see that the graph H has a matching of size $|\mathcal{P}|$. From the claim we have shown above, we can color all joint edges in G into red or blue except those corresponding to the arcs in the matching of H . Since at most one edge of $\pi(R)$ is selected in the matching for each $R \in \mathcal{R}$, at least one of them is red or blue. Thus, the coloring is a covering coloring. $\square$

Thus, from Lemma 13 and Lemma 11, we can conclude that there exists a rounding whose maximum error is bounded by 1.25 if each entry of the input matrix is an integral multiple of 0.25.

4 Concluding Remarks

In this paper we have discussed how to achieve low discrepancy with respect to 2×2 square regions when we round a $[0, 1]$ -valued matrix into a binary one. Our new upper bound is $5/3 \simeq 1.67$. There still exists a large gap between the lower bound ($= 1$) and the upper bound. Thus, a simple but interesting open question is to tighten the gap; indeed the authors are curious whether we can construct an example forcing the optimal rounding error to be 1.25 if the input matrix consists of entries of integral multiples of 0.25 (it is easy to make an example in which the rounding error is forced to be 1). Another direction is to extend the region size from 2×2 to k -by- k regions for $k \geq 3$. Even for the case $k = 3$, we have neither a nontrivial upper bound nor a lower bound.

References

1. R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows, Theory Algorithms and Applications*, Princeton Hall, 1993. 639
2. T. Asano, T. Matsui, and T. Tokuyama: "On the Complexities of the Optimal Rounding Problems of Sequences and Matrices," Proc. 7th Scandinavian Workshop on Algorithm Theory, pp.476-489, Bergen, Norway, July 2000. 636, 637, 638
3. T. Asano, T. Matsui, and T. Tokuyama: "Optimal Roundings of Sequences and Matrices," *Nordic Journal of Computing*, **7** (2000), 241-256. 636
4. Z. Baranyai, "On the factorization of the complete uniform hypergraphs", in *Infinite and Finite Sets*, eds. A. Hanaj, R. Rado and V. T. Sós, *Colloq. Math. Soc. János Bolyai* **10** (1974), pp.91-108. 636, 638
5. J. Beck and W. Chen, *Irregularities of Distribution*, Cambridge University Press, Cambridge, 1987. 637
6. J. Beck and V. T. Sós, *Discrepancy Theory*, in *Handbook of Combinatorics* Volume II, (ed. R. Graham, M. Grötschel, and L. Lovász) 1995, Elsevier. 638
7. R. W. Floyd and L. Steinberg: "An adaptive algorithm for spatial gray scale," *SID 75 Digest, Society for Information Display*, (1975) pp.36-37. 637

8. D. E. Knuth: “Digital halftones by dot diffusion,” ACM Trans. Graphics, 6-4 (1987), pp.245–273. 637
9. R. Ulichney: *Digital halftoning*, MIT Press, 1987. 637

Labeling Subway Lines^{*}

Mari Ángeles Garrido¹, Claudia Iturriaga², Alberto Márquez¹, José Ramón Portillo¹, Pedro Reyes¹, and Alexander Wolff³

¹ Dept. de Matemática Aplicada I, Universidad de Sevilla, Spain
{vizuete,almar,josera,preyes}@us.es

² Faculty of Computer Science, University of New Brunswick, Canada
citurria@unb.ca

³ Institut für Mathematik und Informatik, Universität Greifswald, Germany
awolff@uni-greifswald.de

Abstract. Graphical features on map, charts, diagrams and graph drawings usually must be annotated with text labels in order to convey their meaning. In this paper we focus on a problem that arises when labeling schematized maps, e.g. for subway networks. We present algorithms for labeling points on a line with axis-parallel rectangular labels of equal height. Our aim is to maximize label size under the constraint that all points must be labeled.

Even a seemingly strong simplification of the general point-labeling problem, namely to decide whether a set of points on a horizontal line can be labeled with sliding rectangular labels, turns out to be weakly NP-complete. This is the first labeling problem that is known to belong to this class. We give a pseudo-polynomial time algorithm for it.

In case of a sloping line points can be labeled with maximum-size square labels in $O(n \log n)$ time if four label positions per point are allowed and in $O(n^3 \log n)$ time if labels can slide. We also investigate rectangular labels.

1 Introduction

Label placement is one of the key tasks in the process of information visualization. In diagrams, maps, technical or graph drawings, features like points, lines, and polygons must be labeled to convey information. The interest in algorithms that automate this task has increased with the advance in type-setting technology and the amount of information to be visualized. Due to the computational complexity of the label-placement problem, cartographers, graph drawers, and computational geometers have suggested numerous approaches, such as expert systems [1], zero-one integer programming [16], approximation algorithms [7,12,13,14], simulated annealing [5] and force-driven algorithms [9] to name only a few. An extensive bibliography about label placement can be found at [15]. The ACM Computational Geometry Impact Task Force report [4] denotes label placement as an important research area. Manually labeling a map is a tedious task that is estimated to take 50% of total map production time.

^{*} Partially supported by PAI project FQM—0164

When producing schematized maps [3], e.g. for road or subway networks, an interesting new label-placement problem has to be solved: that of labeling points on a line, e.g. stations on a specific subway line. We assume that all labels are parallel to each other and contain text of the same font size, so we can model labels by axis-parallel rectangles of equal height. We investigate two different labeling models, 4P and 4S, that were introduced in [7] and [14], respectively. In the *fixed-position model* 4P a label must be placed such that one of its four corners coincides with the point site to be labeled. The *slider model* 4S is less restrictive in that a label can be placed such that *any* point of its boundary coincides with the site. See Figure 1 for a variety of point-labeling models that have been studied previously [14,12]. In that figure, each rectangle stands for a feasible label position. An arrow between two rectangle indicates that additionally all label position are feasible that arise when moving one rectangle on a straight line onto the other.

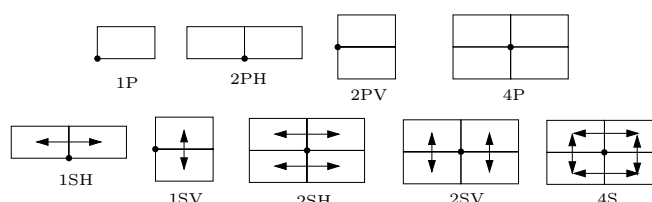


Fig. 1. Each model has an abbreviation of the form xMD where $M \in \{P, S\}$ stands for fixed-position model (P) or slider model (S), $x \in \{1, 2, 4\}$ refers to the number of fixed positions or sliding directions, and $D \in \{\emptyset, H, V\}$ indicates the horizontal or vertical direction in which fixed-position labels are arranged or labels can slide

While most point-labeling problems are computationally hard [7], one would expect to be in a better situation if the input points are not scattered all over the plane but lie on a line. We show, however, that this is not necessarily true: labeling points on a horizontal line with sliding rectangles remains NP-hard. We do give a pseudo-polynomial time algorithm for that problem and show that several simplifications—square labels or no sliding—all have efficient algorithms. Other point-labeling problems that are not NP-hard include labeling points with maximum-size rectangles in one of two positions [7] or with maximum-size rectangles of aspect ratio 1:2 in one of four special positions [13]. There is also a polynomial-time algorithm that decides whether points on the boundary of a rectangle can be labeled with so-called elastic labels, i.e. rectangular labels of fixed area but flexible length and height [10]. Last but not least the problems 1d- k PH and 1d-1SH of labeling points on a horizontal line with labels in a constant number of positions and with sliding labels, respectively, have been studied under the restriction that all labels must be placed on top of the line [11,12].

Our paper is structured as follows. In Section 2 we investigate the problem of labeling points on a horizontal line with axis-parallel rectangular labels that

touch the line. In Section 3 we consider labeling points on sloping lines with squares and sketch how some of these can be extended to rectangular labels. Throughout the paper we consider labels topologically open, i.e. they may touch other labels or input points. An M -labeling maps each input point to a label position that is allowed in labeling model M such that no two labels intersect. For a variety of labeling models, refer to Figure 1. In our paper the names of the models in Figure 1 are prefixed with “1d-” or “Slope-”, in order to refer to the corresponding problems where all input points lie on a horizontal or sloping line, respectively. An *optimal* labeling will refer to a labeling where all labels are scaled by the same factor and this factor is maximum (prefix “Max-”).

2 Points on a Horizontal Line

So far only Poon et al. have explicitly given algorithms for labeling points on a horizontal line [12]. They assume points with weights and investigate algorithms for maximizing the weighted sum of points that can be labeled *above* the line. Their aim actually is to label points with unit-height labels in the plane, but they reduce the difficult two-dimensional rectangle-placement problem into simpler one-dimensional interval-placement problems by means of line stabbing. Solving the 1d-problems (near-) optimally then gives approximation algorithms for the 2d-problem. See Figure 1 for the labeling models Poon et al. consider. The discrete case 1d- k PH, where each point has only a constant number of feasible label positions, is a special case of maximum-weight independent set (MWIS) on interval graphs. They use a MWIS algorithm to solve 1d- k PH in $O(kn \log n)$ time. In the weighted case the problem 1d-1SH where labels can slide horizontally above the given line (see Figure 1) is equivalent to a job scheduling problem, namely single-machine throughput maximization. It is not known whether a polynomial-time algorithm for 1d-1SH exists in the weighted case. Poon et al. modify a fully polynomial-time approximation scheme (FPTAS) for single-machine throughput maximization to approximate 1d-1SH: for each $\varepsilon > 0$ they obtain a factor- $(1 + \varepsilon)$ approximation algorithm that runs in $O(n^2/\varepsilon)$ time and uses $O(n/\varepsilon)$ space. They also give an exact pseudo-polynomial time algorithm based on dynamic programming for 1d-1SH with a bounded number of different weights and an exact $O(n^2 \log n)$ -time algorithm for the special case of square labels (i.e. intervals of fixed length). A similar approach can be used to approximate 1d-2SH (and, equivalently 1d-4S) in the weighted case: there is a factor- $(1.8 + \varepsilon)$ approximation algorithm for 2-machine throughput maximization that runs in $O(n^2/\varepsilon)$ time [2].

Kim et al. have investigated algorithms for labeling axis-parallel line segments with sliding maximum-width rectangles [11]. They also consider the 1d-case first and show that 1d-1SH can be decided in linear time for unit squares (unit-length intervals) in the unweighted case if points are given in left-to-right order. In the same paper they also give a linear-time algorithm for the problem Max-1d-1SH, where the label length is maximized under the restriction that all labels have the same length.

In this section we will investigate a problem that is more difficult than 1d-1SH: we allow to place labels both above and below the horizontal line, say the x -axis, on which the input points are given. Let us start by introducing some notions that we will use throughout the paper, both for horizontal and sloping lines. We will direct the line from left (bottom) to right (top) and process the points in this linear order.

Definition 1. Given a set $P = \{p_1, \dots, p_n\}$ of n points on a line ℓ in lexicographical order, we refer to ℓ as the input line and direct it according to the order on P . Given an axis-parallel label L_i for each $p_i \in P$ and a labeling model M , a k -tuple $R = (r_1, \dots, r_k)$ is a k -realization of P if each entry r_i encodes a position of L_i that is valid in M and no two labels intersect.

For the 4P-model an entry of a k -realization is simply an integer $r_i \in \{1, 2, 3, 4\}$ that specifies in which of the four quadrants (in canonical order) L_i lies relative to a coordinate system with origin p_i . For the 4S-model we take $r_i \in [1, 5[$ with the obvious meaning that e.g. 2.5 is half way in between position 2 and 3. In order to express minimality among realizations we need at least a partial order on the set of possible k -realizations and thus on the label positions r_i . Intuitively, a minimum k -realization should be a k -realization that leaves the maximum amount of freedom for the placement of label L_{k+1} . This leads to the concept of the *shadow* of a k -realization—space that cannot be used for placing L_{k+1} . Our definition depends on the fact that our labels are always axis-parallel rectangles.

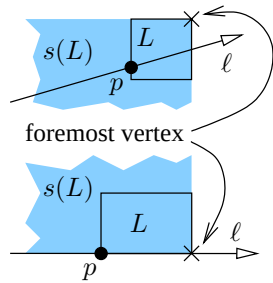
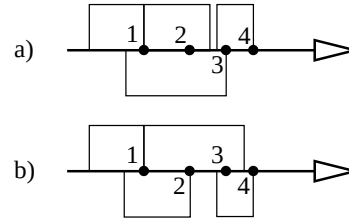
Definition 2. The foremost vertex of a label L is the point on the boundary of L that is furthest in the direction of the input line ℓ . In case of a tie a point on ℓ wins. The shadow $s(L)$ of a label L is the quadrant of the plane that contains L and is defined by the foremost vertex of L and the two adjacent edges of L . The shadow of a k -realization R is $s(R) = \cup_{i=1}^k s(L_i)$. Two k -realizations are equivalent if they have the same shadows.

For shadows of labels, see Figure 2. A shadow of a k -realization given a sloping line is depicted in Figure 5. Let us now focus on horizontal lines.

Definition 3. If ℓ is a horizontal line, the dual of a k -realization R is the k -realization R^* with $r_i^* = 5 - r_i$ for $i = 1, \dots, k$. We write $R \leq R'$ if $s(R) \subseteq s(R')$ or $s(R^*) \subseteq s(R')$. $R = (r_1, \dots, r_k)$ is a minimum k -realization if $(r_1, \dots, r_i) \leq R'$ for all i -realizations R' and for each $i = 1, \dots, k$.

If ℓ is horizontal, the shadow of a realization R can be denoted by (t, b) where t (b) is the x -coordinate of the right edge of the rightmost label in R above (below) ℓ . The dual R^* of R is obtained by mirroring R at ℓ . For a minimum and a non-minimum 4-realization, see Figure 3.

Lemma 1. If there is a k -realization R of P then there is also a minimum k -realization R' of P .

**Fig. 2.** Shadows of labels**Fig. 3.** A minimum (a) and a non-minimum realization (b)

Proof. Let $R_{|i}$ be the i -realization (for $i = 1, \dots, k-1$) obtained from R by removing its $k-i$ last entries (i.e. labels). Our proof is by induction over k . The claim is certainly valid for $k = 1$: in this case $R' = (2)$ or $R' = (3)$, i.e. place the label of p_1 leftmost. For $k > 1$ if $R_{|k}$ is not a minimum k -realization then by our induction hypothesis we have a minimum $(k-1)$ -realization R'' . Clearly $R'' \leq R_{|k-1}$, thus adding the label L_k of R to R'' gives a k -realization R' . To make sure that R' is in fact minimum, we push L_k as far left as possible, checking positions both below and above ℓ . If this new R' was not a minimum k -realization we would have a contradiction to the minimality of R'' . $\square$

Thus it is enough to keep track of minimum k -realizations to solve the decision problem. Among these only non-equivalent k -realizations are of interest. Their number can be bounded as follows.

Lemma 2. *Given 1d-4P there are at most two non-equivalent minimum k -realizations for $k = 1, \dots, n$.*

This is proved by induction over k and by going through all different possibilities according to the position of p_k . The duality of two realizations is important here.

Theorem 1. *If points are given in left-to-right order, 1d-4P with rectangles can be decided in linear time and space.*

Proof. We label points in the given order starting with the two minimum 1-realizations that we get from labeling p_1 leftmost, i.e. in positions 2 and 3. Then in each step we only have to combine each minimum $(k-1)$ -realization R with the two leftmost placements of label k that R allows (if any) and compare the resulting at most four realizations with each other. According to Lemma 2 at most two of these are minimum and have to be kept. If at some point label k cannot be combined with any of the minimum $(k-1)$ -realizations, then Lemma 1 guarantees that no k -realization exists. In this case the algorithm outputs “no”, otherwise it returns a minimum n -realization. $\square$

The maximization version Max-1d-4P of this problem is the following: given a set P of n points $p_1 = (x_1, 0), \dots, p_n = (x_n, 0)$ each with a label of length l_i and unit height, find the largest stretch factor $\lambda_{\max}$ such that there is a 1d-4P-labeling of P with labels of length $\lambda_{\max}l_1, \dots, \lambda_{\max}l_n$ and determine the corresponding labeling.

Theorem 2. Max-1d-4P can be solved in $O(n^2 \log n)$ time using $O(n^2)$ space or in $O(n^3)$ time using linear space.

Proof. First we sort the input points lexicographically. Let $\Delta x_{i,j} = |x_j - x_i|$. Since at least two labels must touch each other in an optimal labeling, $\lambda_{\max}$ must be in the list $L = \{\Delta x_{i,j}/l_i, \Delta x_{i,j}/l_j, \Delta x_{i,j}/(l_i + l_j) : 1 \leq i < j \leq n\}$ of all potentially optimal stretch factors. We compute L , sort L , and do a binary search on L calling our decision algorithm in each step. This takes $O(n^2 \log n)$ time and uses quadratic space for L . Instead we could also compute the elements of L on the fly and test them without sorting. $\square$

The problem becomes much harder when we allow labels to slide horizontally. We will show this by reducing a special variant of Partition to 1d-4S. Actually sliding vertically does not help when the input line is horizontal, so 1d-4S is equivalent to 1d-2SH, where only horizontal sliding is allowed. For the same reason the problem 1d-2SV, where only vertical sliding is allowed, is equivalent to 1d-4P, see Figure 1.

Theorem 3. 1d-4S is NP-complete.

Proof. The problem is in $\mathcal{NP}$ since we have the following non-deterministic polynomial-time decision algorithm. First guess an n -tuple c with entries $c_i \in \{a, b\}$ that encode whether label L_i lies above or below the input line in the solution. Then go through the points from left to right and place L_i at position 1 if $c_i = a$ and at position 4 if $c_i = b$. Push L_i left until it either hits a previously placed label or is in its leftmost position. If L_i cannot be placed without intersection, there cannot be a solution that conforms to c . However, if there is a 1d-4S-labeling of the given points, it is found with non-zero probability.

To show the NP-hardness we reduce the following NP-hard variant of Partition [8] to 1d-4S. Given positive integers $a_1, \dots, a_{2m}$ is there a subset I of $J = \{1, \dots, 2m\}$ such that I contains exactly one of $\{2i-1, 2i\}$ for $i = 1, \dots, m$ and $\sum_{i \in I} a_i = \sum_{i \in J \setminus I} a_i$? We will reduce an instance A of this problem to an instance (P, L) of 1d-4S such that A can be partitioned if and only if there is a 1d-4S-labeling of P with the corresponding labels from L .

First let C be a very large and c a very small number, e.g. $C = 1000 \sum_{i \in J} a_i$ and $c = \min_{i \in J} a_i / 1000$. Our point set P consists of 4 stoppers and $2m$ usual points $a, b, p_1, p_2, \dots, p_{2m}, y, z$ from left to right with distances $\overline{ab} = \overline{yz} = c$, $\overline{bp_1} = \overline{p_{2m}y} = C/2$, $\overline{p_{2i-1}p_{2i}} = (a_{2i-1} + a_{2i})/2$, and $\overline{p_{2i}p_{2i+1}} = C$, thus $\overline{by} = mC + \sum_{i \in J} a_i/2$, see Figure 4. The corresponding labels have length $l_a = l_b = l_y = l_z = \overline{by}$ and $l_i = C + a_i$, thus $\sum_{i \in J} l_i = 2mC + \sum_{i \in J} a_i = 2\overline{by}$. Due to the long labels of the stoppers, the other points must be labeled between these stoppers.

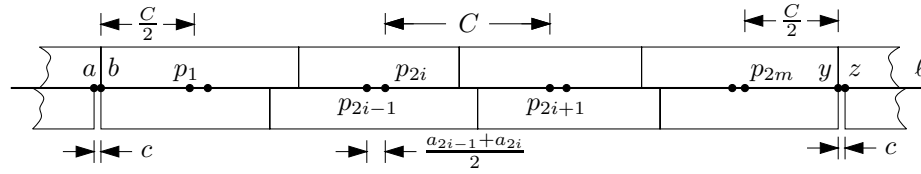


Fig. 4. Instance of 1d-4S to which Partition is reduced

The total space available above and below the input line is $\overline{by} + \overline{az} = 2\overline{by} + 2c$ and thus just slightly more than the total length of the labels.

If there is a labeling for this instance then it must be tight (neglecting the $2c$ extra space) and the number of labels above and below the input line ℓ must be equal. Due to its length a label is attached to its point roughly in its center. Thus the labels of p_{2i-1} and p_{2i} lie on opposite sides of ℓ . Therefore the indices of the points whose labels lie above ℓ give the desired partition I of J .

On the other hand if there is a partition I of J then we can label P as follows. For each p_i with $i \in I$ we place its label with the lower left corner at $x_b + \sum_{j \in I, j < i} l_j$, where x_b is the x -coordinate of b . The labels of the other m points are placed analogously below ℓ . $\square$

We needed extremely long labels and point distances to construct the reduction from Partition to 1d-4S above. In practice such labels are not common, which makes it worthwhile to design a pseudo-polynomial time algorithm whose running time depends not only on n , the size of the input, but also on $l_{\max}$, the length of the longest label. Pseudo-polynomial time algorithms have been suggested for point labeling before. There is a scheduling algorithm that can be used for weight maximization given 1d-1SH and runs in $O(dn \log \log d)$ time, where $d = x_n + l_n$ [2]. Another example is an approximation algorithm that labels points with circles of radius at least $R^*/3.6$ in $O(n \log n + n \log R^*)$, where R^* is the maximum label radius [6].

Theorem 4. *If the input consists exclusively of integers and points are sorted from left to right, 1d-4S can be solved in $O(nl_{\max}^2)$ time and space, where $l_{\max}$ is the length of the longest label.*

Proof. We will use dynamic programming with a table T of size $(n+1) \times (2l_{\max} + 1) \times (2l_{\max} + 1)$. Let a k -realization be *leftmost* if all its labels are pushed as far left as possible. Note that a leftmost realization is not necessarily minimum. Now an entry $T[k, t, b]$ is a boolean that answers the question “Is (t, b) the shadow of a leftmost k -realization?” where $t, b \in \{-l_{\max}, \dots, l_{\max}\}$ are measured relative to x_{k+1} assuming $x_{n+1} = x_n + l_n$. Initially all table entries are false except $T[0, -l_{\max}, -l_{\max}]$. The entries of level k are computed from those in level $k-1$ as follows. Let $f_k(x) = \max\{-l_{\max}, x - \Delta x_{k,k+1}\}$ be the function that maps a point x measured relative to x_k to a point $f_k(x)$ measured relative to x_{k+1} with a lower bound of $-l_{\max}$. For each level- $(k-1)$ entry $T[k-1, t, b]$ that

is true we switch at most two entries in level k to true: if $t \leq 0$ we generate a new leftmost k -realization by placing L_k leftmost above ℓ . If additionally $t \geq -l_k$ then the new label touches the last label above ℓ and we set $T[k, f_k(t+l_k), f_k(b)]$ to true, otherwise $T[k, f_k(0), f_k(b)]$. The case $b \leq 0$ is symmetric. The algorithm returns true if and only if there is an entry of value true at level n .

It is not difficult to modify the algorithm within the time and space bound of $O(nl_{\max}^2)$ such that it actually computes a labeling if one exists. The proof of correctness is by induction over k . $\square$

3 Points on a Sloping Line with Square Labels

In this section we will investigate labeling problems where the input line has positive slope and labels are equal-size squares. We will give decision and label-size maximization algorithms for both the discrete and the continuous versions Slope-4P and Slope-4S, respectively. We allow labels to intersect the input line, otherwise only two label positions per point would be valid, and the decision version could simply be reduced to 2-SAT and solved in linear time [7]. We start with the more difficult problem Slope-4S and later show how it can be simplified to Slope-4P. Since we do not have the notion of duality for sloping lines as in Definition 2, we redefine minimality as follows.

Definition 4. Let R, R' two k -realizations. We write $R \leq R'$ if $s(R) \subseteq s(R')$. $R = (r_1, \dots, r_k)$ is a minimum k -realization if $(r_1, \dots, r_i) \leq R'$ for all i -realizations R' and for each $i = 1, \dots, k$. A $(k+1)$ -realization is the child of a k -realization if their first k entries agree.

Lemma 3. A k -realization has at most two children that are minimum.

Proof. Given a k -realization R and a point p_{k+1} with a square label L , R has a child iff $p_{k+1} \notin s(R)$. In this case place L in position 1. There are two paths in which L can be slid towards its optimal position 3, either left-down on a path via position 2 or down-left on a path via position 4. Here optimality refers to the resulting $(k+1)$ -realization, and sliding means that the label is moved continuously from one position to the other while touching p_{k+1} . All shadows of L on one path are comparable to each other, while no shadow of L on one path is comparable with a shadow of L on the other path except for the two endpoints. Thus sliding L as far as possible without intersecting $s(R)$ on each of the two paths gives minimum shadows for L , see Figure 5. Appending these positions of L to R yields at most two children that can be minimum $(k+1)$ -realizations. $\square$

We will continue to use the terms left-down and down-left from the previous proof. We will say that a k -realization $R = (r_1, \dots, r_k)$ is of type LD if $r_k \leq 3$ and of type DL if $r_k \geq 3$. Extending this notation, we will say that R is of type X-X if the last two entries of R are both at most 3 or both at least 3, X-Y otherwise. The following two lemmas are easy to prove.

Lemma 4. *The shadow of a realization is determined by its last two labels.*

Lemma 5. *Two k -realizations of type DL-DL are comparable to each other, and their minimum can be determined in constant time.*

Due to symmetry the same holds for k -realizations of type LD-LD.

Theorem 5. *Given unit-square labels, Slope-4S can be solved in quadratic time and space.*

Proof. We sort the input points lexicographically and process them in this order. In each step we maintain a superset $\mathcal{R}$ of the set of minimum k -realizations. The idea to bound the size of $\mathcal{R}$ is the following. In step $k+1$ each of the k -realizations in $\mathcal{R}$ yields at most two $(k+1)$ -realizations according to Lemma 3, one of type LD and one of type DL. Of these $(k+1)$ -realizations at most $|\mathcal{R}|$ will be of type X-Y. All $(k+1)$ -realizations of type DL-DL are comparable to each other according to Lemma 5. Finding the minimum among them takes linear time, analogously for those of type LD-LD. Thus we keep at most $|\mathcal{R}|$ $(k+1)$ -realizations of type X-Y and 2 of type X-X. According to Lemma 3, the minimum $(k+1)$ -realizations must be among them. Since the number of realizations increases by two in each step, the total running time and space consumption are quadratic. $\square$

There are point sets with a linear number of minimum n -realizations of type X-Y, for an example see Figure 6, so one cannot hope to do better using the concept of minimum realizations.

The following lemma is shown in a similar way as Lemma 1 in [11]. The idea is that in an optimal solution there are paths of labels that touch each other, and among these paths there is a path whose first and last label touch input points—either with their top and bottom edge or with their left and right edge. Otherwise all labels could be slid and enlarged by a small factor.

Lemma 6. *Given Max-Slope-4S the maximum label size lies in the set $L = \{\Delta x_{i,j}/m, \Delta y_{i,j}/m : 1 \leq i < j \leq n, 1 \leq m \leq j-i\}$.*

This immediately yields an algorithm for Max-Slope-4S using binary search on the list L of potential label sizes and the decision algorithm of Theorem 5.

Corollary 1. *Given square labels, Max-Slope-4S can be solved in $O(n^3 \log n)$ time using $O(n^2)$ space.*

As expected the discrete version Slope-4P can be solved faster.

Theorem 6. *Given unit-square labels and points sorted lexicographically, Slope-4P can be solved in linear time and space.*

Proof. Lemma 4 holds in the discrete case Slope-4P as well. Here, however this immediately implies that there are at most $4^2 = 16$ (a case analysis reduces this to 2) different shadows in each step. When placing label L_{k+1} we get at most 32 $(k+1)$ -realizations by Lemma 3, but of these at most 16 can have different shadows and must be kept. Filtering these out can be done in constant time, which yields a linear-time algorithm. $\square$

Corollary 2. *Given square labels, Max-Slope-4P can be solved in $O(n \log n)$ time using linear space.*

Proof. It is not hard to see that in an optimal solution of Max-Slope-4P only labels of neighboring points or points that have a common neighbor can touch, see Figure 7. Thus there is a list $L = \{\Delta x_{i,j}/m, \Delta y_{i,j}/m : 1 \leq i \leq n, j \in \{i+1, i+2\}, m \in \{1, 2\}\}$ of linear length that contains the maximum label size. The algorithm is as follows: first sort P , then compute L , sort L , and finally do a binary search on L . $\square$

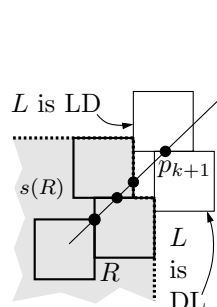


Fig. 5. The two paths of label L given 4S

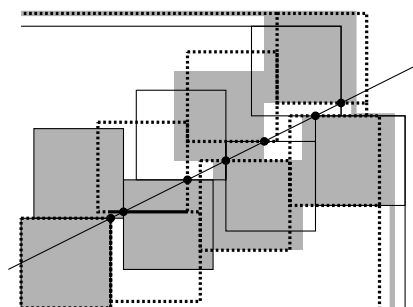


Fig. 6. Three incomparable shadows of type LD-DL

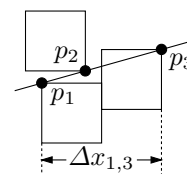


Fig. 7. Touching 4P-labels

The problem Slope-4P can be solved similarly for unit-height rectangles in linear time. The maximization version, however, takes longer, namely $O(n^2 \log n)$ time, since there is a quadratic number of possibly optimal stretch factors.

4 Conclusions

In this paper we have studied problems that arise when labeling schematic maps such as subway networks where the points to be labeled lie on a line. Even among these seemingly strong simplifications of the general point-labeling problem there are cases that remain weakly NP-hard.

There are plenty of open problems left in 1d-labeling. First of all we would like to see the time complexity of our algorithm for Max-Slope-4S reduced. Then it would be interesting to see how much the reduction to one dimension helps to solve the label-number or label-weight maximization problem. Given a set of points in the plane, each with its own label length but fixed height, there is a PTAS for finding the largest subset of points that can be labeled [14], while for weight maximization only a factor- $(\frac{1}{2} - \epsilon)$ approximation algorithm is known [12].

References

1. J. Ahn and H. Freeman. AUTONAP—an expert system for automatic map name placement. In *Proceedings International Symposium on Spatial Data Handling*, pages 544–569, 1984. 649
2. P. Berman and B. DasGupta. Multi-phase algorithms for throughput maximization for real-time scheduling. *Journal of Combinatorial Optimization*, 4(3):307–323, Sept. 2000. 651, 655
3. S. Cabello, M. de Berg, S. van Dijk, M. van Kreveld, and T. Strijk. Schematization of road networks. In *Proceedings of the 17th Annual ACM Symposium on Computational Geometry (SoCG'01)*, 2001. To appear. 650
4. B. Chazelle and 36 co-authors. The computational geometry impact task force report. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223, pages 407–463. American Mathematical Society, Providence, 1999. 649
5. J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995. 649
6. S. Doddi, M. V. Marathe, and B. M. Moret. Point set labeling with specified positions. In *Proc. 16th Annu. ACM Sympos. Comput. Geom. (SoCG'00)*, pages 182–190, Hongkong, 12–14 June 2000. 655
7. M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annu. ACM Sympos. Comput. Geom. (SoCG'91)*, pages 281–288, 1991. 649, 650, 656
8. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, 1979. 654
9. S. A. Hirsch. An algorithm for automatic name placement around point data. *The American Cartographer*, 9(1):5–17, 1982. 649
10. C. Iturriaga and A. Lubiw. Elastic labels around the perimeter of a map. In *Proceedings of the 8th International Workshop on Algorithms and Data Structures (WADS'99)*, volume 1663 of *Lecture Notes in Computer Science*, pages 306–317, Vancouver, B. C., Canada, 12–14 Aug. 1999. Springer-Verlag. 650
11. S. K. Kim, C.-S. Shin, and T.-C. Yang. Labeling a rectilinear map with sliding labels. *International Journal of Computational Geometry and Applications*, 11(2):167–179, Apr. 2001. 650, 651, 657
12. S.-H. Poon, C.-S. Shin, T. Strijk, and A. Wolff. Labeling points with weights. In *Proc. 12th Annual International Symposium on Algorithms and Computation (ISAAC'01)*, *Lecture Notes in Computer Science*, Christchurch, 19–21 Dec. 2001. Springer-Verlag. To appear. 649, 650, 651, 658
13. Z. Qin, A. Wolff, Y. Xu, and B. Zhu. New algorithms for two-label point labeling. In *Proc. 8th Annu. Europ. Symp. on Algorithms (ESA'00)*, volume 1879 of *Lecture Notes in Computer Science*, pages 368–379, Saarbrücken, 5–8 Sept. 2000. Springer-Verlag. 649, 650
14. M. van Kreveld, T. Strijk, and A. Wolff. Point labeling with sliding labels. *Computational Geometry: Theory and Applications*, 13:21–47, 1999. 649, 650, 658
15. A. Wolff and T. Strijk. The Map-Labeling Bibliography. <http://www.math-inf.uni-greifswald.de/map-labeling/bibliography/>, 1996. 649
16. S. Zoraster. The solution of large 0-1 integer programming problems encountered in automated cartography. *Operations Research*, 38(5):752–759, 1990. 649

Complexity Study on Two Clustering Problems (Extended Abstract)^{*}

Louxin Zhang and Song Zhu

¹ Department of Mathematics & KRDL, National University of Singapore
Singapore 117543

matzlx@nus.edu.sg

² Bioinformatics Lab, Kent Ridge Digital Labs
Singapore 119613

zhusong@krdl.org.sg

Abstract. The complexity issues of two clustering problems are studied. We prove that the Smooth Clustering and Biclustering problems are NP-hard; we also propose an 0.5 approximation algorithm and 0.8 inapproximability for a simplified clustering problem.

1 Introduction

By allowing biologists to observe thousands of genes being on or off under certain conditions, cDNA microarray technology is becoming a powerful and versatile tool for studying many important attributes of genes. It has been used for gene functional assignments; it has been used for study of gene regulation networks of a living cell; it has also been used for cancer classification and diagnosis. Because a large number of genes are often involved in rich cDNA microarray experiments, clustering is necessary for gene expression analysis as it would partition genes into different classes each containing genes expressing in similar patterns under certain conditions (such as tissues, environments) (see [1,5,8,7,2,4]).

Recently, the authors proposed a novel clustering approach for overcoming data errors such as data missing [9] and expression inconsistency across different experiments[3] in the stage of clustering. The approach is based on the so-called smooth score [10]. Gene expression data generated from cDNA microarray experiments are usually given as matrices, where each entry is a gene expression value under a condition. We assume the rows of such a matrix correspond to the genes and the columns to the conditions. The smooth score is not defined as a pairwise dissimilarity measure like Euclidean distance; instead, it measures the deviation of the expression level of a gene from the average expression level of all the involved genes under a condition. For a cluster I of genes obtained by considering conditions in J , its smooth score is formally defined in Equation (1). We formulate the Smooth Clustering problem as, given a set of conditions,

^{*} Supported in part by a NSTB grant.

finding a largest cluster of genes with its smooth score below a threshold under the given conditions. We would also like to find a largest smooth ‘bicluster’ with its smooth score below a threshold, grouping genes and conditions simultaneously as proposed in [4], which is called the Smooth BiClustering problem. The authors proposed efficient greedy algorithms for the Smooth Clustering and BiClustering problems in [10]. These algorithms were shown to perform well in finding co-regulation patterns in a test with a yeast data set.

In this paper, we study the computational complexity of these two clustering problems. The Smooth Clustering and Biclustering problems are introduced in Section 2 and are proved to be NP-hard in Section 3. In Section 3, a variant of the Smooth Clustering problem is also proved to be NP-hard. In Section 4, we focus on a special case of the Smooth Clustering Problem where the input gene expression matrix has only one column. A polynomial-time $\frac{1}{2}$ -approximation algorithm is presented for the special case. However, unless $\text{NP} = \text{P}$, there is no polynomial-time algorithm of approximation ratio better than $\frac{4}{5}$ even for the special case. Finally, we conclude the paper with an open problem in Section 5.

2 Clustering Problems

The gene expression data from cDNA microarray experiments is usually presented as a matrix. Each entry represents the relative abundance of the mRNA of a gene under a specific condition. Here, we assume that each row corresponds to a gene and a column to a condition. The logarithm transformation is often applied to gene expression values for converting multiplicative changes of the relative abundance into additive increments.

Let $A = (a_{ij})$ be a gene expression matrix with gene set X and condition set Y . Any subsets $I \subseteq X$ and $J \subseteq Y$ specify a submatrix $A(I, J)$. We associate it with the following smooth score

$$s(I, J) = \max_{j \in J} \max_{i \in I} |a_{ij} - \frac{1}{|I|} \sum_{k \in I} a_{kj}|, \quad (1)$$

where $\frac{1}{|I|} \sum_{k \in I} a_{kj}$ denotes the average expression level of genes in I under condition j . The smooth score $s(I, J)$ is actually a refinement of L_∞ -distance $d_\infty(\cdot, \cdot)$, a popular metric in functional analysis. Recall that, for any two n -dimensional vectors $\mathbf{x} = (x_i)$ and $\mathbf{y} = (y_i)$, $d_\infty(\mathbf{x}, \mathbf{y}) = \max_i |x_i - y_i|$. If a gene expression level is considered as a function with condition as variable, clustering aims to classifying genes into groups each containing genes with expression functions in similar shapes. Thus, we propose the smooth score for gene expression analysis. If $A(I, J)$ has the smooth score $s(I, J)$, then, for any rows v and v' in $A(I, J)$, $d_\infty(v, v') \leq 2s(I, J)$.

Given a small number $\epsilon > 0$, $A(I, J)$ is an ϵ -smooth cluster if $s(I, J) \leq \epsilon$. We formulate clustering as the following problem.

Smooth Clustering Problem [10]

Instance: A gene expression matrix $A = (a_{ij})$ with gene set X and condition set Y , a subset $J \subseteq Y$, and a number $\epsilon > 0$;

Question: Find a largest subset $I \subseteq X$ such that $A(I, J)$ is an ϵ -smooth cluster.

To facilitate the study of genes with multiple functions that may or may not be co-active under all conditions, we use an approach proposed by Cheng and Church by allowing clustering both genes and conditions [4]. Thus, we investigate the following problem:

Smooth Biclustering Problem [10]

Instance: A gene expression matrix A with gene set X and condition set Y , and a number $\epsilon > 0$;

Question: Find an ϵ -smooth submatrix $A(I, J)$, $I \subseteq X$ and $J \subseteq Y$, that maximizes $\min\{|I|, |J|\}$.

3 NP-Hardness Results

In this section, we prove the Smooth Clustering and Biclustering problems to be NP-hard. For basic notations and knowledge on NP-hardness and approximation algorithms, the reader is referred to [6].

Theorem 1. *The Smooth Clustering problem is NP-hard.*

Proof. We prove the theorem using a reduction from the INDEPENDENT SET problem, which is a basic NP-complete problem [6]. Recall that the INDEPENDENT SET problem is to, given a graph $G = (V, E)$ and an integer $k > 0$, find whether G contains a subset of $V' \subseteq V$ such that no two vertices in V' are jointed by an edge $e \in E$ and such that $|V'| \geq k$.

Given an instance $(G = (V, E), k)$ of the INDEPENDENT SET problem, we construct an instance of the Smooth Clustering problem as follows. For simplicity, assume that $V = \{1, 2, \dots, n\}$ and $E = \{e_1, e_2, \dots, e_m\}$. We define a matrix $A_{n \times m} = (a_{ij})$ as

$$a_{ij} = \begin{cases} 0 & \text{if } i \text{ is not an endpoint of } e_j, \\ -1 & \text{if } e_j = (i, i'), i < i', \\ 1 & \text{if } e_j = (i', i), i' < i. \end{cases}$$

Note that each column of A corresponds to an edge and has exactly two non-zero entries -1 and 1. Let $J = \{1, 2, \dots, m\}$ and $\epsilon = 1 - \frac{1}{n}$.

Let $V' = \{i_1, i_2, \dots, i_k\}$. For any column j of matrix A , if its corresponding edge e_j has two endpoints in V' , then, $\sum_{l \in V'} a_{lj} = 0$, and $\max_{i \in V'} |a_{ij} - \frac{1}{k} \sum_{l \in V'} a_{lj}| = 1$; Similarly, if e_j has only one endpoint in V' , $\max_{i \in V'} |a_{ij} - \frac{1}{k} \sum_{l \in V'} a_{lj}| = 1 - \frac{1}{k}$; and if e_j has no endpoints in V' ,

$\max_{i \in V'} |a_{ij} - \frac{1}{k} \sum_{l \in V'} a_{lj}| = 0$. This concludes that V' is an independent set of G if and only if V' is a feasible solution to the instance $(A_{n \times m}, J, \epsilon)$ of the Smooth Clustering problem. Hence, if we can find an optimal solution to $(A_{n \times m}, J, \epsilon)$, we can easily decide whether the graph G has an independent set of size k or not. Since the INDEPENDENT SET problem is NP-complete, the Smooth Clustering problem is NP-hard.

Now, we consider the following variant of the Smooth Clustering problem, which is of interest itself.

Square Smooth Clustering Problem

Instance: A gene expression matrix $A = (a_{ij})$ with gene set X and condition set Y , a subset $J \subseteq Y$, and a number $\epsilon > 0$;

Question: Find a largest subset $I \subseteq X$ such that, for every $j \in J$,

$$\sum_{i \in I} (a_{ij} - \frac{1}{|I|} \sum_{l \in I} a_{lj})^2 \leq \epsilon.$$

Theorem 2. *The Square Smooth Clustering Problem is NP-hard.*

Proof. We again prove the theorem by a reduction from the INDEPENDENT SET problem. Given an instance $(G = (V, E), k)$ of the INDEPENDENT SET problem, we will construct an instance (A, ϵ) of the Square Smooth Clustering problem as follows. We let $A = (a_{ij})$ be the adjacent matrix of the graph G , where each column of A corresponds to an edge and each row to a vertex. Then, by definition, a_{ij} is 1 if the i th vertex is an endpoint of the j th column and it is 0 otherwise. Since each edge has two endpoints, each column contains exactly two entries of value 1 (called 1-entries). Assume $V' = \{v_{i_1}, v_{i_2}, \dots, v_{i_k}\}$ is a subset of vertices and its index subset I' is $I' = \{i_1, i_2, \dots, i_k\}$. Now we start to prove that, if $|V'| = k > 4$, V' is an independent set if and only if

$$\sum_{i \in I'} (a_{ij} - \frac{1}{|I'|} \sum_{l \in I'} a_{lj})^2 \leq 1 \quad (2)$$

for every column j in A .

If V' is an independent set, then, there is at most one 1-entry among a_{ij} 's, $i \in I'$, for every column j . Therefore, for every j , $\sum_{i \in I'} (a_{ij} - \frac{1}{|I'|} \sum_{l \in I'} a_{lj})^2 = 0$ if all a_{ij} 's ($i \in I'$) are zero. Otherwise, $\frac{1}{|I'|} \sum_{l \in I'} a_{lj} = \frac{1}{k}$ since $|I'| = k$ and there is one 1-entry among a_{ij} 's ($i \in I'$); furthermore,

$$\sum_{i \in I'} (a_{ij} - \frac{1}{|I'|} \sum_{l \in I'} a_{lj})^2 = (1 - \frac{1}{k})^2 + (k-1) \cdot (0 - \frac{1}{k})^2 = 1 - \frac{1}{k} < 1.$$

Conversely, if (2) is true for I' , then there is at most one 1-entry among a_{ij} 's ($i \in I'$) for every column j . Assume the fact is not true for some column j' . Since

the j' -th column has exactly two 1's, all these two 1-entries are among $a_{ij'}$'s, $i \in I'$. Hence, since $|I'| = k$, $\frac{1}{|I'|} \sum_{l \in I'} a_{lj'} = \frac{2}{k}$; since $k > 4$,

$$\sum_{i \in I'} (a_{ij'} - \frac{1}{|I'|} \sum_{l \in I'} a_{lj'})^2 = 2(1 - \frac{2}{k})^2 + (k-2)(0 - \frac{2}{k})^2 = \frac{2(k-2)}{k} > 1.$$

This contradicts to (2). Therefore, there is at most one 1-entry among $a_{ij'}$'s ($i \in I'$) for every column j and hence V' is an independent set.

We have proved that V' is an independent set of G if and only if its index subset I' is a solution for the instance $(A, 1)$ of the Square Smooth Clustering problem if $|V'| = |I'| > 4$. This implies that if we can find a solution of size k for $(A, 1)$, we can easily find an independent set of the same size in G . Since the INDEPENDENT SET problem is NP-complete, the Square Smooth Clustering problem is NP-hard. This finishes the proof.

The Smooth Biclustering problem can be thought as a generalization of finding a largest balanced complete bipartite subgraph of a bipartite graph. Thus it is NP-hard as proved below.

Theorem 3. *The Smooth Biclustering problem is NP-hard.*

Proof. The problem is equivalent to find a largest ϵ -smooth square submatrix $A(I, J)$, i.e. $|I| = |J|$. We proved the theorem by using a reduction from the Balanced Complete Bipartite Subgraph problem. Recalled that this problem is to, given a bipartite graph $G = (V, E)$ and a positive integer $k \leq |V|$, find two disjoint subsets $V_1, V_2 \subseteq V$ such that $|V_1| = |V_2| = k$ and such that $v_1 \in V_1$ and $v_2 \in V_2$ implies that $(u, v) \in E$. Such a problem is NP-complete (listed as GT24 in [6]).

Given a bipartite graph (V, E) , we construct an instance of the Smooth Biclustering problem as follows. Without loss of generality, we may assume that $V = \{1, 2, \dots, n\}$, where $n = |V|$. We define an $n \times n$ matrix $A = (a_{ij})$ by assigning $a_{ij} = a_{ji} = 0$ if $(i, j) \in V$, and $a_{ij} = i$ and $a_{ji} = j$ otherwise. Hence, each row/column of A corresponds to a vertex of the graph. Let $\epsilon < 1/2$. Then, we claim that, for any subsets $I, J \subset V$ such that $|I| = |J| \geq 2$, the square submatrix $A(I, J)$ is ϵ -smooth if and only if $I \times J \subseteq E$ and thus the induced subgraph on $I \cup J$ is a balanced complete bipartite graph. This implies that Smooth Biclustering problem is NP-hard.

Now we start to prove the claim. Assume that $I \times J \subseteq E$. By definition, for any $i \in I, j \in J$, $a_{ij} = 0$. Thus, $A(I, J)$ is a zero matrix and hence ϵ -smooth. Conversely, if $A(I, J)$ is ϵ -smooth, where $|I| = |J|$, it has to be a zero matrix and hence, $I \cup J$ induces a balanced complete bipartite graph. Otherwise, let the j th column have non-zero entries for some $j \in J$. Assume that $a_{lj} = \min_{i \in I} a_{ij}$ and $a_{mj} = \max_{i \in I} a_{ij}$. Since $|I| \geq 2$, by definition, $a_{lj} < a_{mj}$. Since all entries are integers, $|a_{lj} - \frac{1}{|I|} \sum_{i \in I} a_{ij}| \geq 1/2$, or $|a_{mj} - \frac{1}{|I|} \sum_{i \in I} a_{ij}| \geq 1/2$. This contradicts to the fact that $A(I, J)$ is ϵ -smooth.

4 Approximation Results

Since the Smooth Clustering problem is NP-hard, it is desirable to develop efficient approximating algorithms for it. However, this task seems difficult in general. In this section, we shall focus on matrices with only one column. In this case, the Smooth Clustering problem is equivalent to

Smooth Subset problem: Given a finite set S , a weight $w(s) \geq 0$ for each $s \in S$, and a positive number ϵ , find a largest ϵ -smooth subset $S' \subseteq S$, i.e. $|w(s) - \frac{1}{|S'|} \sum_{t \in S'} w(t)| \leq \epsilon$ for every $s \in S'$.

Theorem 4. *Let $k(S, \epsilon)$ be the size of a largest ϵ -smooth subset of S for a weighted set S and $\epsilon > 0$. There is a polynomial-time algorithm that always outputs an ϵ -smooth subset of size at least $k(S, \epsilon)/2$ on any input S .*

Proof. Let (S, ϵ) be an instance of the Smooth Subset problem. For simplicity, we assume that $S = \{a_1, a_2, \dots, a_n\}$, where $w(a_i) \leq w(a_{i+1})$ for $1 \leq i \leq n-1$. Otherwise, we can sort S in terms of its weight function in polynomial time. Let S' be a largest ϵ -smooth subset of S . Then, by assumption, $k(S, \epsilon) = |S'|$. For any $a, b \in S'$, by triangle inequality,

$$|w(a) - w(b)| \leq |w(a) - m| + |w(b) - m| \leq 2\epsilon,$$

where $m = \frac{1}{|S'|} \sum_{x \in S'} w(x)$. This concludes that S' is contained in an interval of length 2ϵ . Hence,

$$k(S, \epsilon) = |S'| \leq \max_{1 \leq i \leq n} |\{a \in S \mid w(a) \in [w(a_i), w(a_i) + 2\epsilon]\}|.$$

Let $X_i = \{a \in S \mid w(a) \in [w(a_i), w(a_i) + \epsilon]\}$ for $i = 1, 2, \dots, n$. Clearly, each X_i is an ϵ -smooth subset of S since $|w(a) - \frac{1}{|X_i|} \sum_{x \in X_i} w(x)| \leq w(a_i) + \epsilon - w(a_i) = \epsilon$ for every $a \in X_i$. We choose the largest subset X over all X_i 's, $1 \leq i \leq n$. Since

$$k(S, \epsilon) \leq \max_{1 \leq i \leq n} |\{a \in S \mid w(a) \in [w(a_i), w(a_i) + 2\epsilon]\}| \leq 2|X|,$$

X is an ϵ -smooth subset of size at least $\frac{1}{2}k(S, \epsilon)$. Moreover, X can be found in polynomial time.

Now we study a restricted version of the Smooth Subset problem:

Restricted Smooth Subset problem: given a finite weighted set S , a number $\epsilon > 0$, and two elements $a, b \in S$ such that $w(a) < w(b)$, find a largest ϵ -smooth subset S' 'between a and b ' that satisfies

$$\{a, b\} \subseteq S' \subseteq \{s \in S \mid w(a) \leq w(s) \leq w(b)\},$$

where $w(x)$ denotes the weight of $x \in S$.

It is easy to see that any efficient algorithm for the Restricted Smooth Subset problem can be used to solve the general problem efficiently by considering

all possible pairs (a, b) . Moreover, the Restricted Smooth Subset problem can be formulated into an integer program. Let $(S, (a, b))$ be an instance of the Restricted Smooth Subset problem, where $w(a) < w(b)$. If $w(b) - w(a) > 2\epsilon$, then, there is no ϵ -smooth subset containing a and b . Hence, we assume that $w(b) - w(a) \leq 2\epsilon$. We assign a 0-1 variable y_s to each $s \in S$ satisfying $w(a) \leq w(s) \leq w(b)$. Then, finding a largest ϵ -smooth subset ‘between a and b ’ can be formulated into the following integer program:

$$\begin{aligned} & \text{Maximize } \sum_{s \in S} y_s, \\ & \text{subject to} \\ & \quad \sum_{s \in S} (w(s) - \epsilon - w(a)) y_s \leq 0, \\ & \quad \sum_{s \in S} (w(b) - \epsilon - w(s)) y_s \leq 0, \\ & \quad y_a = y_b = 1, y_s = 0 \text{ or } 1, s \in S. \end{aligned}$$

Using this formulation, we obtain the following simple results.

Theorem 5. *Let S be a weighted finite set, and let $\epsilon > 0$. For any $a, b \in S$ such that $0 \leq w(b) - w(a) \leq 2\epsilon$, the following facts hold:*

(1). *For any largest ϵ -smooth subset S' of S such that $\{a, b\} \subseteq S' \subseteq \{x \mid w(a) \leq w(x) \leq w(b)\}$,*

$$\{s \in S \mid w(b) - \epsilon \leq w(s) \leq w(a) + \epsilon\} \subseteq S'.$$

(2). *If $w(b) = 2\epsilon + w(a)$, for any ϵ -smooth subset S' of S such that $\{a, b\} \subseteq S' \subseteq \{x \mid w(a) \leq w(x) \leq w(b)\}$,*

$$\sum_{s \in S} (w(s) - \epsilon - w(a)) y_s = 0$$

where y_s is 0 if $s \notin S'$ and 1 otherwise.

Using Theorem 5 (2), we are able to prove the following theorem.

Theorem 6. *Let $k(S, \epsilon)$ is the size of a largest ϵ -smooth subset of S for any weighted set S and $\epsilon > 0$. For any small constant $\delta > 0$, there is no polynomial-time algorithm that always outputs an ϵ -smooth subset of size at least $(0.8 + \delta)k(S, \epsilon)$ on input S unless $\text{NP} = \text{P}$.*

Proof. Let δ be a small positive constant. Suppose $\mathcal{A}$ is a polynomial time approximation algorithm with approximation factor $0.8 + \delta$ for the Smooth Subset problem. We will show that $\mathcal{A}$ can be used to derive a polynomial time algorithm for the Partition problem, contradicting its NP-completeness [6]. Recall that the Partition problem is to, given a finite set B and an integer size $s(b) > 0$ for each $b \in B$, decide if there is a subset $B' \subseteq B$ such that $\sum_{b \in B'} s(b) = \sum_{b \in B - B'} s(b)$. Observe that if B' is a solution subset to the Partition instance B , $B - B'$ is also a solution subset. Since δ is a constant, we only focus on instance sets B with size no less than $\frac{2}{5}(\frac{1}{5\delta} - 1)$.

For such a weighted set B as an instance of the Partition problem, we let $v = \frac{1}{2} \sum_{b \in B} b$. If $\max_{b \in B} s(b) > v$, obviously, there is no solution to the instance B . Without loss of generality, we may assume that $\max_{b \in B} s(b) \leq v$. Let $\epsilon = 2v + 1$. We construct an instance (D, ϵ) of the Smooth Subset problem from B as follows. First, D contains $|B|$ elements x_i of weight 0 and $|B|$ elements y_j of weight 2ϵ ; For each $b \in B$, D contains a unique element u_b of weight $\epsilon - s(b) > 0$; finally, D contains an element z of weight $\epsilon + v$. In total, D contains $3|B| + 1$ elements.

Fact *If there is a solution to the Partition instance B , then D has an ϵ -smooth subset of size at least $\frac{5}{2}|B| + 1$; Otherwise, any ϵ -smooth subset of D has size at most $2|B| + 1$.*

Proof. Suppose B has a subset B' such that $\sum_{b \in B'} s(b) = \sum_{b \in B-B'} s(b) = v$. Without loss of generality, we assume that $|B'| \geq \frac{1}{2}|B|$. (Otherwise, we choose $B - B'$ instead.) Define

$$D' = \{x_i, y_i \mid 1 \leq i \leq |B|\} \cup \{u_b \mid b \in B'\} \cup \{z\}.$$

Then, D' contains $2|B| + |B'| + 1$ elements, which is at least $\frac{5}{2}|B| + 1$ since $|B'| \geq \frac{1}{2}|B|$ by assumption. Noting that

$$\frac{1}{|D'|} \sum_{d \in D'} w(d) = \frac{1}{|D'|} \left[\sum_{i=1}^{|B|} w(x_i) + \sum_{i=1}^{|B|} w(y_i) + w(z) + \sum_{b \in B'} (\epsilon - s(b)) \right] = \epsilon,$$

we conclude that D' is ϵ -smooth. This has proved the first part of the fact.

If there is no solution to the Partition instance B , then, $\sum_{b \in B'} s(b) \neq v$ for any subset $B' \subset B$. Let D'' be a largest ϵ -smooth subset of D . Recall that, for each $d \in D$, we use $w(d)$ to denote its weight. Let $m = \frac{1}{|D''|} \sum_{d \in D''} w(d)$. If $m < \epsilon$, then, D'' does not contain $|B|$ elements of weight 2ϵ since $2\epsilon - m > \epsilon$ and thus $|D''| \leq 2|B| + 1$. Similarly, if $m > \epsilon$, D'' does not contain $|B|$ elements of weight 0 and again $|D''| \leq 2|B| + 1$. If $m = \epsilon$, we prove that $|D''| \leq 2|B| + 1$ by considering the following two cases.

Case 1: D'' does not contain either all the elements of weight 0 or of weight 2ϵ . Obviously, $|D''| \leq 2|B| + 1$.

Case 2: D'' contains r elements of weight 0 and s elements of weight 2ϵ , where $r, s > 0$. We let $D_1 = D'' \cap \{z, u_b \mid b \in B\}$. If D_1 is empty, then $|D''| \leq 2|B|$. Otherwise, we divide Case 2 into three subcases.

Subcase 2.1: $r = s$. Since D_1 is non-empty, that $m = \epsilon$ implies that $\sum_{d \in D_1} w(d) = |D_1|\epsilon$. Then, D_1 must contain the element z , which has weight $\epsilon + v$; hence, it induces a solution subset to the Partition instance B , a contradiction.

Subcase 2.2: $r > s$. Then, $\sum_{d \in D_1} w(d) = ((r - s) + |D_1|)\epsilon$ since $m = \epsilon$. Again, D_1 must contain the element z . On the other hand, for any element

$d \in D_1 - \{z\}$, $w(d) \leq \epsilon$. Since $\epsilon = 2v + 1$,

$$\begin{aligned} \sum_{d \in D_1} w(d) &\leq (|D_1| - 1)\epsilon + v + \epsilon \\ &< (|D_1| - 1)\epsilon + 2\epsilon \leq ((r - s) + |D_1|)\epsilon, \end{aligned}$$

a contradiction.

Subcase 2.3: $r < s$. Then, as Subcase 2.1, we have that

$$(s - r)\epsilon + \sum_{d \in D_1} w(d) = |D_1|\epsilon$$

since $m = \epsilon$. This is impossible since

$$\sum_{d \in D_1} w(d) \geq |D_1|\epsilon - \sum_{b \in B} s(b)$$

and hence

$$\begin{aligned} (s - r)\epsilon + \sum_{d \in D_1} w(d) &\geq \epsilon + \sum_{d \in D_1} w(d) \\ &= \sum_{b \in B} s(b) + 1 + \sum_{d \in D_1} w(d) > |D_1|\epsilon. \end{aligned}$$

This has proved the fact.

By assumption, $\mathcal{A}$ is a polynomial time algorithm with approximation factor $0.8 + \delta$ for the Smooth Subset problem. Now, we apply $\mathcal{A}$ to the instance D . If $\mathcal{A}$ outputs an ϵ -smooth subset has size at most $2|B| + 1$, then, the largest ϵ -smooth subset has size at most

$$\frac{2|B| + 1}{0.8 + \delta} = \frac{5}{2}|B| + \frac{5}{4 + 5\delta} - \frac{25|B|\delta}{8 + 10\delta} < \frac{5}{2}|B| + 1$$

since $\mathcal{A}$ is a $(0.8 + \delta)$ -approximation algorithm and $|B| > \frac{2}{5}(\frac{1}{5\delta} - 1)$. Thus, we conclude that there is no solution to the Partition instance B by the fact proved above. If $\mathcal{A}$ outputs an ϵ -smooth subset of length at least $2|B| + 2$, then, by the fact, there is a solution to the Partition instance B . Therefore, we derive a polynomial time algorithm for the Partition problem using $\mathcal{A}$, contradicting NP-completeness of the Partition problem.

5 Conclusions

We have proved that the Smooth Clustering and Biclustering problems are NP-hard. We also provide a simple polynomial time $\frac{1}{2}$ -approximation algorithm for the Smooth Subset problem, a special case of the Smooth Clustering problem. It is interesting whether there is a better approximation algorithm for the Smooth Subset problem or not.

References

1. A. Alizadeh *et al.* Distinct types of diffuse large B-cell lymphoma identified by gene expression profiling, *Nature* 403(2000), 503-510. 660
2. A. Ben-Dor, Z. Yakhini. Clustering gene expression patterns. In *Proceedings of Recomb'99*, 33-42. 660
3. Y. Chen, E. Dougherty, M. Bitter. Ratio-based decisions and the quantitative analysis of cDNA microarray images. *J. Biomed. Optics* 2 (1997), 364-374. 660
4. Y. Cheng and G. Church. Biclustering of expression data, In *Proceedings of ISMB'2000*, 93-103. 660, 661, 662
5. M. B. Eisen, P. T. Spellman, P. O. Browns, D. Botstein. Clustering Analysis and display of genome-wide expression pattern. *PNAS* 95(1998), 14863-68. 660
6. M. R. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. San Francisco: Freeman, 1979. 662, 664, 666
7. E. Hartuv, A. Schmitt, J. Lange, S. Meier-Ewert, H. Lehrach, and R. Shamir. An algorithm for clustering cDNAs for gene expression analysis, In *Proceedings of Recomb'99*, 188-197. 660
8. P. Tamayo, D. Slonim, J. Mesirov, Q. Zhu, S. Kitareewan, E. Dmitrovskay, E. S. Lander, T. R. Gloub. Interpreting patterns of gene expression with self-organizing maps: methods and application to hematopoietic differentiation. *PNAS* 96(1999), 2907-12. 660
9. O. Troyanskaya, M. Cantor, O. Alter, G. Sherlock, P. Brown, D. Botstein, R. Tibshirani, T. Hastie, and R. Altamn. Missing value estimation methods for DNA microarrays. *Bioinformatics* 17(2001), 520-525. 660
10. L. Zhang and S. Zhu. A new approach to clustering gene expression data. Submitted for publication. 660, 661, 662

A Modified Greedy Algorithm for the Set Cover Problem with Weights 1 and 2^{*}

Toshihiro Fujito and Tsuyoshi Okumura

Department of Electronics, Nagoya University
Furo, Chikusa, Nagoya 464-8603 Japan
fujito@nuee.nagoya-u.ac.jp

Abstract. The set cover problem is that of computing, given a family of weighted subsets of a base set U , a minimum weight subfamily $\mathcal{F}'$ such that every element of U is covered by some subset in $\mathcal{F}'$. The k -set cover problem is a variant in which every subset is of size bounded by k . It has been long known that the problem can be approximated within a factor of $H(k) = \sum_{i=1}^k (1/i)$ by the greedy heuristic, but no better bound has been shown except for the case of unweighted subsets. In this paper we consider approximation of a restricted version of the weighted k -set cover problem, as a first step towards better approximation of general k -set cover problem, where subset costs are limited to either 1 or 2. It will be shown, via LP duality, that improved approximation bounds of $H(3) - 1/6$ for 3-set cover and $H(k) - 1/12$ for k -set cover can be attained, when the greedy heuristic is suitably modified for this case.

1 Introduction

The *set cover* problem (SC) is a typical combinatorial optimization problem with many practical applications, and it is defined as follows: Given a base set U of n elements, a family $\mathcal{F}$ of subsets of U , and a nonnegative cost c_S associated with each $S \in \mathcal{F}$, it is required to find a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of minimum total weight such that $\cup_{S \in \mathcal{F}'} S = U$. The *k -set cover* problem (k -SC) is a variant of SC in which every subset is of size bounded from above by a constant k . The problem SC, or even k -SC for $k \geq 3$, is known to be NP-hard [15] as well as MAX SNP-hard [18].

An intuitively most natural and simple heuristic for SC is the greedy algorithm, which iteratively picks a most “cost-effective” subset until every element of U is covered by some picked subset; here, the cost effectiveness of a subset S is measured by its cost c_S divided by the number of elements, “yet to be covered”, in S . For the case of unit-costs it was first shown by Johnson [14] that its performance ratio is bounded by the n th Harmonic number $H(n) = \sum_{i=1}^n (1/i)$ for SC, of which value is between $\ln(n+1)$ and $1 + \ln n$, or $H(k)$ for k -SC, and it was Lovász who obtained the same results by making use of *fractional covers* [16]. While the same performance ratio of $H(k)$ was later shown to hold

^{*} Research partially supported by a grant from the Nitto Foundation.

even for the case of general costs [4] by extension of these analysis via the linear program duality, Slavík proved that it is exactly $\ln n - \ln \ln n + \Theta(1)$ for unit-cost SC [19]. It turns out, moreover, that the greedy bound of $H(n)$ is almost the best possible one for SC because the interactive proof based hardness result of Feige [7] says that SC is not approximable within a factor of $(1 - \epsilon) \ln n$ for any fixed $\epsilon > 0$ unless $NP \subset DTIME(n^{O(\log \log n)})$.

An alternative view of the greedy algorithm for *unweighted* k -SC is that it computes a “maximal” set packing P^k of k -sets, reduces to an instance I of $(k-1)$ -SC by removing all the k -sets in P^k , and recurses on I . Although k -SC is NP-hard for $k \geq 3$ as already stated, 2-SC is nothing but the *edge cover* problem which can be solved in time complexity of maximum matching. Thus, when I is reduced to the one for 2-SC, we may finish up the entire procedure with an optimal solution for I , instead of a maximal set packing of 2-sets plus whatever remains. This is the observation used by Goldschmidt, Hochbaum, and Yu [10], and they proved that such a modification to the standard greedy heuristic leads to the performance ratio of $H(k) - 1/6$ for k -SC. Further improvements over the greedy bound have been obtained more recently, by additionally applying various local search techniques to ordinary greedy, in the order of $H(k) - 11/42$ [11], $H(k) - 1/3$ [12], and $H(k) - 1/2$ [5], which is the best bound known to date for unweighted k -SC.

The packing problem as a counterpart of SC is the *maximum set packing* problem, another fundamental set optimization problem, and in the *k-set packing* problem, it is required, given a weighted set system as in k -SC, to find a subfamily of disjoint subsets of “maximum” total weight. For this problem, the tight greedy bound is k , whether subsets are weighted or not, while a local search heuristic yields an approximation ratio of $k/2 + \epsilon$ if subsets are unweighted [13, 11]. Unlike *weighted* k -SC, however, the performance ratio for *weighted* k -set packing has been improved from k of greedy, by combination of greedy and local search techniques, to $k - 1 + \epsilon$ [2, 1] first, and then to $2(k+1)/3$ [3].

In this paper we consider approximation of a severely restricted version of *weighted* k -SC, as a first step towards better approximation of general k -SC, where subset costs are limited to either 1 or 2, and show that improved approximation bounds of $H(3) - 1/6$ for 3-SC and $H(k) - 1/12$ for k -SC can be attained when the greedy heuristic is suitably modified for this case. This algorithm is a generalization of the modified greedy algorithm of Goldschmidt et al., being identical to theirs when all the costs are unit (i.e., unweighted case), and the approximation bound of $H(k) - 1/6$ was also shown by them to be tight for $k \geq 3$ [10]. Although it may thus appear that such improvements result from straightforward extension of their approach for the unweighted case, we base our analysis on the LP relaxation of k -SC and its dual program, following the Chvátal’s approach in [4], unlike their analysis, or any other in [11, 12, 5], which are all based on purely combinatorial arguments. In Sect. 3 we first present an alternative proof that the modified greedy algorithm delivers a solution of which size is within a factor of $H(k) - 1/6$ from optimal (i.e., unweighted case), where a lower bound for the optimal size is now provided by the LP relaxation. This

new proof serves for the purposes of elucidating our strategy in a simpler setting of only unit costs, and providing bases of algorithmic and accounting scheme to be extended later. A key to our accounting scheme in this analysis is the decomposition theorem of Gallai [8,9] and Edmonds [6]. In Sect. 4 the algorithm is extended to deal with subsets of costs 1 and 2, first in the setting of 3-SC and then in that of k -SC. Interestingly, the Gallai-Edmonds decomposition will be seen needed here in the algorithm design as well to guide us in picking subsets.

2 Preliminaries

An instance of the set cover problem is a weighted set system $(U, \mathcal{F})$, where $\mathcal{F} \subseteq 2^U$, $\cup_{S \in \mathcal{F}} S = U$, and each $S \in \mathcal{F}$ is associated with a nonnegative cost c_S . For any $\mathcal{F}' \subseteq \mathcal{F}$, we write $\overline{\mathcal{F}'}$ as a shorthand of $\cup_{S \in \mathcal{F}'} S$, and a set of size i will be called i -set.

In the greedy-type algorithms considered in this paper, once $S \in \mathcal{F}$ is picked, it will never be discarded from a solution. Also, once it happens and S is picked, we assume that any subset $S' \in \mathcal{F}$ is represented by $S' - S$ from now onward, and if $S' - S$ is later picked by the algorithm, it will be understood that the one actually selected into a solution is S' . More formally, if all the subsets in $\mathcal{F}'$ are already picked at some point, the current state of a given instance is represented by the set system $(V, \mathcal{F}[V])$, where $V = U - \overline{\mathcal{F}'}$ and $\mathcal{F}[V] = \{S \cap V \mid S \in \mathcal{F}\}$ is the collection of subsets *induced by V in $\mathcal{F}$* , and the cost of $S \cap V \in \mathcal{F}[V]$ is c_S .

2.1 LP Relaxation

The set cover problem for an instance of $(U, \mathcal{F})$, with a nonnegative cost c_S for each subset S , can be formulated by the following integer program:

$$\begin{aligned}
 & \text{Min } \sum_{S \in \mathcal{F}} c_S \cdot x_S \\
 & \text{subject to:} \\
 \text{(IP)} \quad & \sum_{S: u \in S} x_S \geq 1 \quad \forall u \in U \\
 & x_S \in \{0, 1\} \quad \forall S \in \mathcal{F}
 \end{aligned}$$

where $x_S = 1$ iff S is chosen in a solution. The LP relaxation of (IP), denoted (LP), is then obtained by replacing the integral constraints $x_S \in \{0, 1\}$ in (IP) by linear constraints $x_S \geq 0$ for all $S \in \mathcal{F}$. Let OPT denote the optimal value of (LP), with which the cost of our solution will be compared.

We also make use of the dual of (LP), denoted (D), and it is given by:

$$\begin{aligned}
 & \text{Max } \sum_{u \in U} y_u \\
 & \text{subject to:} \\
 \text{(D)} \quad & \sum_{u \in S} y_u \leq c_S \quad \forall S \in \mathcal{F} \\
 & y_u \geq 0 \quad \forall u \in U
 \end{aligned}$$

Suppose now that we have a set cover $\mathcal{C} \subseteq \mathcal{F}$ and dual variables $y \in \mathbb{Q}^U$ satisfying that

1. $c(\mathcal{C}) = \sum_{S \in \mathcal{C}} c_S = y(U) = \sum_{u \in U} y_u$, and
2. $y(S) = \sum_{u \in S} y_u \leq \alpha \cdot c_S$ for each $S \in \mathcal{F}$,

for some $\alpha \in \mathbb{R}^+$. Then, since $(1/\alpha)y(S) \leq c_S, \forall S \in \mathcal{F}$, $(1/\alpha)y$ is feasible to (D), with the objective value of $(1/\alpha)y(U) = (1/\alpha) \sum_{u \in U} y_u$. The LP duality theorem says an objective value of (D) is always a lower bound for OPT, implying that the cost of $\mathcal{C}$, $y(U)$, is bounded by $\alpha \cdot \text{OPT}$:

Proposition 1. *If a set cover $\mathcal{C}$ and dual variables $y \in \mathbb{Q}^U$ satisfy the two conditions given above, $c(\mathcal{C}) \leq \alpha \cdot \text{OPT}$.*

(This is the approach taken by Chvátal [4] in establishing the greedy bound of $H(n)$ (or $H(k)$) for the *weighted* set cover problem)

2.2 Decomposition Theorems

Gallai [8,9] and Edmonds [6] independently found a “canonical” decomposition of a graph determined by maximum matchings in it. For any graph G denote by D the set of all vertices in G which are not covered by at least one maximum matching of G . Let A denote the set of vertices in $V - D$ adjacent to at least one vertex in D , and let $C = V - A - D$. A graph G is called *factor-critical* if removal of any vertex from G results in a graph having a perfect matching in it. Clearly, any factor-critical graph contains an odd number of vertices. A *near-perfect* matching in G is one covering all but exactly one vertex of G . This decomposition, which can be computed in polynomial time via the Edmonds matching algorithm, provides important information concerning all the maximum matchings in G :

Theorem 2 (the Gallai-Edmonds structure theorem).

1. *The components of the subgraph induced by D are factor-critical.*
2. *The subgraph induced by C has a perfect matching.*
3. *If M is any maximum matching of G , it contains a near-perfect matching of each component of D , a perfect matching of each component of C and matches all vertices of A with vertices in distinct components of D .*

3 Unweighted Case

In this section we treat the modified greedy algorithm for unweighted k -SC presented in [10], which we call MG-Unit, and analyze its performance using the LP duality, as a precursor to the later analysis for a more general SC with weights. When all the subsets are unit-costed, the ordinary greedy algorithm repeatedly picks one covering the maximum number of uncovered elements, and this has the same effect as computing a maximal set packing consisting of subsets

of size i , when i is the size of a largest subset remaining in the current set system (and choosing all the subsets in the packing). This operation is iterated after the set system is updated by considering any subset to be the one consisting of only uncovered elements in it. The modification was made to this algorithm by noticing that, when no subsets of size larger than 2 exist in the set system after larger ones are already taken, the remainder can be optimally covered since such a set system can be identified with a graph G_1 , and a minimum edge cover for G_1 gives an optimal cover for the system. A minimum cardinality edge cover can be computed in a graph G , by first computing a maximum cardinality matching M in G , and then, choosing any edge incident to u for each vertex u left uncovered by M . It is possible that G_1 contains a singleton component $\{u\}$ with no edge to cover u , and u itself can be selected in this case.

The modified greedy algorithm MG-Unit for unweighted k -SC is thus described as follows:

1. For $i = k$ downto 3 do
 - (a) Construct a maximal i -set packing P_1^i in $(U, \mathcal{F}[U])$.
 - (b) Set $U \leftarrow U - \overline{P_1^i}$.
2. Letting $V_1 = U$ and $E_1 = \mathcal{F}[V_1]$, compute a maximum matching M_1 in a graph $G_1 = (V_1, E_1)$.
3. For each vertex u left uncovered by M_1 in G_1 , choose an extra edge (or vertex) to cover u , and add it to Z .
4. Output $(\cup_{i=3}^k P_1^i) \cup M_1 \cup Z$.

Recall that every vertex left uncovered by a maximum matching M occurs in D , and for any component X of $G[D]$, we say X is *unmatched (by M)* if it contains such a vertex while X is *matched* otherwise. This is equivalent to saying that X is matched iff there exists an edge of M between X and A . Thus, there exist exactly $|A|$ many matched components in $G[D]$, and all the rest are unmatched.

For the sake of analysis of this algorithm, divide V_1 , the vertex set of G_1 , into C_1, A_1 , and D_1 according to Theorem 2. For each edge $\{u, v\}$ taken in M_1 , we distribute its cost to u and v by setting

$$\begin{aligned}
 y_u = y_v = 1/2 & \quad \text{if } \{u, v\} \subseteq C_1, \\
 y_u = y_v = 1/2 & \quad \text{if } \{u, v\} \subseteq D_1 \text{ and } \{u, v\} \text{ lies in a component matched} \\
 & \quad \text{by } M_1, \\
 y_u = 1/3, \quad y_v = 2/3 & \quad \text{if } u \in A_1, v \in D_1.
 \end{aligned}$$

To account for total cost of the edge cover for G_1 , the vertices yet to be assigned are those in unmatched components of $G_1[D_1]$, and let X be such a component. Since X contains exactly one vertex left uncovered by M_1 , X is covered by $\lfloor |X|/2 \rfloor$ many edges of M_1 and just one more. Averaging the total cost of these subsets over all the vertices of X , the covering cost for X can be accounted for by setting

$$y_u = \frac{\lfloor |X|/2 \rfloor + 1}{|X|}$$

for each $u \in X$. Notice that $y_u = 1$ if X is a singleton set $\{u\}$, but otherwise, $y_u = \frac{\lfloor |X|/2 \rfloor + 1}{|X|} \leq \frac{\lfloor 3/2 \rfloor + 1}{3} = 2/3$ since $|X| \geq 3$.

Lemma 3. *Let $y : V_1 \rightarrow \mathbb{Q}$ be the dual assignments on the vertices of $G_1 = (V_1, E_1)$, as given above. Then,*

$$\begin{aligned} y_u &\leq 1 \text{ for each } u \in V_1, \\ y_u + y_v &\leq 4/3 \text{ for each } \{u, v\} \in E_1. \end{aligned}$$

Proof. These inequalities can be easily verified to hold, and $y_u = 1$ exactly when u is an unmatched singleton component of $G_1[D_1]$, and $y_u + y_v = 4/3$ exactly when either $\{u, v\}$ is an edge of an unmatched triangle component of $G_1[D_1]$ with $y_u = y_v = 2/3$, or $y_u = 1$ as in the previous case and $y_v = 1/3$ with $v \in A_1$ (recall that there exist no edges between D_1 and C_1). $\square$

Theorem 4. *The algorithm MG-Unit computes a set cover $\mathcal{C}$ such that $|\mathcal{C}| \leq (H(k) - 1/6)OPT$, where k is the size of the largest subset in $\mathcal{F}$.*

Proof. For any subset $S \in \mathcal{F}$ selected in an i -set packing P_1^i during Step 1 (that is, S was an i -set S^i when it was picked), set $y_u = 1/i$ for each $u \in S^i$. For any element u left uncovered after Step 1, assign y_u as above. Then clearly, $|\mathcal{C}| = y(U)$, and it suffices to show that for any i -set $S \in \mathcal{F}$, $y(S) = \sum_{u \in S} y_u \leq H(i) - 1/6$ (by Proposition 1).

Number the elements $\{u_1, \dots, u_i\}$ of an i -set S in the order they are covered. Observe first that $y_{u_l} \leq 1/(i - l + 1)$ for any u_l covered during Step 1, due to the greedy selection rule, since S was of size at least $i - l + 1$ when u_l was covered for the first time, and $y_{u_l} \leq 1/3$ in any case. Therefore, if S becomes a j -set S^j , $0 \leq j \leq 2$, after Step 1, $y(S - S^j) = \sum_{u \in S - S^j} y_u \leq \sum_{l=3}^i (1/l) + (2 - j)(1/3)$. Thus, if S^j is a 0-set, $y(S) \leq \sum_{l=3}^i (1/l) + 2/3 = H(i) - 5/6$, and if $S^j = \{u\}$ is a 1-set, $y_u \leq 1$ by Lemma 3, and hence, $y(S) = y(S - S^j) + y_u \leq (\sum_{l=3}^i (1/l) + 1/3) + 1 = H(i) - 1/6$. Similarly, if $S^j = \{u, v\}$ is a 2-set, it will appear in G_1 as an edge, and $y(S) = y(S - S^j) + y(\{u, v\}) \leq \sum_{l=3}^i (1/l) + 4/3 = H(i) - 1/6$ since $y_u + y_v \leq 4/3$ by Lemma 3. $\square$

4 With Weights 1 and 2

Let us first consider 3-SC, where the cost of any subset is either 1 or 2. In fact, if any cost is either 1 or d with $d \geq 3$, instead of 2, it is easier to design a modified greedy algorithm with performance guarantee of $H(3) - 1/6$; run MG-Unit first on the set system induced by cost-1 subsets only, and then, run MG-Unit again to cover the remainder using this time cost- d subsets only. The reason why such simply concatenated runs of MG-Unit can yield the same approximation ratio as the one for unit cost case is because the effect of the second run can be analyzed independently from the first run; no price assigned in the first run exceeds $d/3$ (≥ 1), which is the least price to be assigned in the second run, and

thus, the first run leaves nothing to hinder the same argument repeated in the second run.

We need to be much more careful, however, when the costs are either 1 or 2, because they are too close to each other, causing more interactions between subsets of different costs, and it could happen, for instance, to be more beneficial to cover the same elements by cost-2 subsets than by cost-1 subsets. To resolve such conflicts we need to delay our commitment in the first run to a certain group of elements until the very end of the second run. Moreover, although the Gallai-Edmonds decomposition played an important role in the analysis of the modified greedy algorithm for the unit-cost k -SC in Sect. 3, here it will be needed in the algorithm itself as well to ensure that a maximum matching computed in the second run possesses a certain property (thus, we need to actually compute it).

4.1 Algorithm for 3-Set Cover

Let $\mathcal{F}_i = \{S \in \mathcal{F} \mid c_S = i\}$, and $U_1 = \overline{\mathcal{F}}_1$, the part of the base set coverable by cost-1 subsets only. If we run the modified greedy algorithm of Sect. 3 on $(U_1, \mathcal{F}_1)$, U_1 will be divided into the subbases covered by a maximal 3-set packing P_1 , and V_1 in which every subset of $\mathcal{F}_1$ is of size bounded by 2. Then, a maximum matching M_1 is computed, and V_1 is further divided into C_1, A_1 , and D_1 , as before. We pay special attention to a singleton components X of $G_1[D_1]$ such that it is unmatched by M_1 yet it can be covered by some cost-2 subset (recall that we had to assign $y_u = 1$ for such a component $\{u\}$ before); so, we designate the set of such singletons as B_1 (i.e., $B_1 = \{u \in D_1 \mid \{u\} \text{ is a component of } G_1[D_1], \text{ unmatched by } M_1, \text{ and } u \in S \text{ for some } S \in \mathcal{F}_2\}$).

Let us consider now 3-SC on $(U_2, \mathcal{F}_2[U_2])$, where $U_2 = U - (U_1 - B_1)$. By first taking any maximal 3-set packing P_2 in it, the system of cost-2 subsets is reduced to a graph $G_2 = (V_2, E_2)$, where $V_2 = U_2 - \overline{P}_2$ and $E_2 = \mathcal{F}_2[V_2]$. Let M_2 be a maximum matching in G_2 , and C_2, A_2 , and D_2 be the Gallai-Edmonds decomposition of V_2 , corresponding to M_2 . We say that a component X of $G_2[D_2]$ is *hit* (by B_1) if $X \cap B_1 \neq \emptyset$, and it is *free* otherwise.

The first part of the modified greedy algorithm, called MG-3SC, for 3-SC with weights 1 and 2 simulates MG-Unit on the set system induced by cost-1 subsets only, as described below, except for non-commitment to those in B_1 :

1. Initialize $U_1 = \overline{\mathcal{F}}_1$, and compute a maximal 3-set packing P_1 of cost-1 subsets in $(U_1, \mathcal{F}_1)$.
2. Set $V_1 \leftarrow U_1 - \overline{P}_1$, and let $G_1 = (V_1, E_1)$ be a graph representing the set system $(V_1, \mathcal{F}_1[V_1])$ of all the remaining cost-1 subsets.
3. Compute a maximum matching M_1 in G_1 as well as the Gallai-Edmonds decomposition, C_1, A_1 , and D_1 , of V_1 .
4. Set $B_1 = \{u \in D_1 \mid \{u\} \text{ is a component of } G_1[D_1], \text{ unmatched by } M_1, \text{ and } u \in S \text{ for some } S \in \mathcal{F}_2\}$.
5. Pick any subset to cover each singleton component, unmatched by M_1 , remaining in $G_1[D_1 - B_1]$ (such a subset must be of cost-1), and store it in Z .

6. From each non-singleton component X , unmatched by M_1 , of $G_1[D_1]$, choose one extra edge to cover the vertex left uncovered by M_1 in X , and store it in Z .

Notice that it is not yet determined at this point which subsets are to cover the elements in B_1 ; we cannot simply use a cost-1 subset to cover each $u \in B_1$ here for, if we do, and if a cost-2 subset $S = \{u, v, w\}$ exists such that $S \cap B_1 = \{u, v\}$ and S is the only one covering w , we need to assign $y_u = y_v = 1, y_w = 2$ resulting in $y(S) = 4 > 2(H(3) - 1/6)$.

Therefore, the second part begins the second run of MG-Unit on the set system induced by $U - (U_1 - B_1)$ as if cost-1 subsets covering those in B_1 do not exist:

7. Set $U_2 \leftarrow U - (U_1 - B_1)$, and construct a maximal 3-set packing P_2 of cost-2 subsets in $(U_2, \mathcal{F}_2[U_2])$.
8. Set $V_2 \leftarrow U_2 - \bar{P}_2$, and let $G_2 = (V_2, E_2)$ be a graph representing the set system $(V_2, \mathcal{F}_2[V_2])$ of all the remaining cost-2 subsets.
9. Compute a maximum matching M_2 in G_2 as well as the Gallai-Edmonds decomposition, C_2, A_2 , and D_2 of V_2 .

Let $\Gamma(u)$ denote the set of vertices adjacent to a vertex u in G_2 . For any vertex u covered by a matching M , there is a unique edge $\{u, v\}$ in M , and v is denoted by $\text{mate}(u)$. Here, we modify the structure of the maximum matching M_2 in G_2 :

10. Set $B_2 \leftarrow \{u \in D_2 \mid \{u\} \text{ is a free singleton component of } G_2[D_2], \text{ unmatched by } M_2\}$.
11. For each $u \in B_2$, test if, for some $v \in \Gamma(u) \subseteq A_2$, the component containing $\text{mate}(v)$ is hit (by B_1), and if so, replace $\{v, \text{mate}(v)\}$ with $\{u, v\}$ in M_2 (the component $\{u\}$ becomes matched as a result).

Observe that, every time M_2 is updated here, an edge of M_2 is flipped from a hit component to a free component; thus, this operation does not cycle, and M_2 is updated at most $|B_2|$ times. This operation ensures M_2 to possess some key property to be used in later analysis, but it is also an intuitively reasonable thing to do. Imagine the situation where the test condition above is satisfied. The component X containing $\text{mate}(v)$ is hit, and so, it can be covered entirely with cost of $2(|X| - 1)/2 + 1 = |X|$ *without* using an edge $\{v, \text{mate}(v)\}$ from M_2 ; thus, both u and X can be covered, by flipping the edge of M_2 with cost of $|X| + 2$, while, if we do not flip, it will cost 2 (for u) plus $2 + 2(|X| - 1)/2 = |X| + 1$ (for X).

In the final stage of subset selection, we recall the existence of cost-1 subsets by which vertices in $V_2 \cap B_1$ can be covered, and we choose them whenever it is a clear plus for us:

12. For each unmatched (by M_2) component X of $G_2[D_2]$,
 - (a) If X is free, choose one extra edge, as in Step 6, to cover the vertex left uncovered by M_2 in X , and store it in Z .

- (b) Otherwise (i.e., hit by B_1), let $u \in B_1$ be any one of those hitting X , and replace edges of M_2 in X , if necessary, by those of a perfect matching in $G_2[X - u]$. This way, X is covered by u and M_2 . Store u in Z .
13. Output $P_1 \cup M_1 \cup P_2 \cup M_2 \cup Z$.

4.2 Analysis

Basically, we extend the analysis of Sect. 3 for the unweighted case to the algorithm above, and to do so, we use the same dual assignment as given in Sect. 3 for the elements covered by cost-1 subsets in the first part of the algorithm, except for the treatment of those in B_1 ; if $u \in B_1$ is after all covered by a cost-1 subset in Step 12b, its cost is accounted for by setting $y_u = 1$, but otherwise, the value of y_u depends on where a cost-2 set covers it in G_2 , though $y_u \leq 1$ in any case, as will be seen next.

The dual assignment corresponding to cost-2 subsets imitates the twice of the one for cost-1 subsets as was given in Sect. 3, yet it is here necessary to distinguish those edges e between A_2 and D_2 according to whether the component of $G_2[D_2]$ into which e is incident is hit or free: For each edge $\{u, v\}$ taken in M_2 , we set

$$\begin{aligned}
 y_u = y_v = 1 & \quad \text{if } \{u, v\} \subseteq C_2, \\
 y_u = y_v = 1 & \quad \text{if } \{u, v\} \subseteq D_2 \text{ and } \{u, v\} \text{ lies in a component matched} \\
 & \quad \text{by } M_2, \\
 y_u = 2/3, \quad y_v = 4/3 & \quad \text{if } u \in A_2, v \in D_2, \text{ and } v \text{ is in a component free} \\
 & \quad \text{from } B_1, \\
 y_u = y_v = 1 & \quad \text{if } u \in A_2, v \in D_2, \text{ and } v \text{ is in a component hit by } B_1.
 \end{aligned}$$

(That is, the cost of $\{u, v\}$ is distributed evenly to u and v except for the case when $\{u, v\} \in A_2 \times D_2$ and v belongs to a free component.)

The remaining vertices to be assigned are again those in unmatched components of $G_2[D_2]$, and let X be such a component. Once again we distinguish the cases according to whether X is hit or free. If it is hit by $v \in B_1$, X is covered by v (or any subset containing it) of cost-1, together with the edges of M_2 perfectly matching $X - u$. Since the total cost is $1 + 2((|X| - 1)/2) = |X|$, we may account for it by setting

$$y_u = 1 \quad \text{if } u \in X \subseteq D_2 \text{ and } X \cap B_1 \neq \emptyset.$$

On the other hand, if X is free from B_1 , the same argument as before tells us that the costs of covering X can be accounted for by setting $y_u = 2(\lfloor |X|/2 \rfloor + 1)/|X|$ for each $u \in X$. Therefore, $y_u = 2$ if X is a singleton set, and $y_u = \frac{2(\lfloor |X|/2 \rfloor + 1)}{|X|} \leq \frac{2(\lfloor 3/2 \rfloor + 1)}{3} = 4/3$ when X is not a singleton set.

The following auxiliary lemma follows easily from the operation of Step 11 in the above algorithm.

Lemma 5. *The matching M_2 computed by the above algorithm satisfies the property that, if $\{u\}$ is a free (from B_1) and unmatched (by M_2) singleton*

component in $G_2[D_2]$, $\text{mate}(v) \in D_2$ belongs to a free component for each $v \in \Gamma(u) \subseteq A_2$.

Now the counterpart of Lemma 3 is:

Lemma 6. *Let $y : V_2 \rightarrow \mathbb{Q}$ be the dual assignments on the vertices of $G_2 = (V_2, E_2)$, as given above. Then,*

$$\begin{aligned} y_u &\leq 2 \text{ for each } u \in V_2 - B_1, \\ y_u &\leq 1 \text{ for each } u \in V_2 \cap B_1, \\ y_u + y_v &\leq 8/3 \text{ for each } \{u, v\} \in E_2. \end{aligned}$$

Proof. Clearly the first inequality holds since $y_u \leq 2, \forall u \in V_2$. For the second inequality, observe that the value of y_u could exceed 1 only when u lies in a component of $G_2[D_2]$, whether it is matched or not, which is free from B_1 (thus, this cannot happen if $u \in B_1$).

Suppose $\{u, v\} \in E_2$ and w.l.o.g. $y_u \geq y_v$. The third inequality clearly holds when $y_u \leq 4/3$, so assume $y_u > 4/3$. Then, it must be the case that $y_u = 2$ and $\{u\}$ is a free and unmatched singleton component of $G_2[D_2]$. Therefore, $\text{mate}(w) \in D_2$ belongs to a free component for each $w \in \Gamma(u) \subseteq A_2$, according to Lemma 5. This then implies that $y_w \leq 2/3, \forall w \in \Gamma(u)$, and this is why $y_u + y_v \leq 2 + 2/3 = 8/3$. $\square$

Lemma 7. *For any $S \in \mathcal{F}_1$ of cost-1, $y(S) \leq H(3) - 1/6$.*

Proof. As far as $(U_1 = \overline{\mathcal{F}}_1, \mathcal{F}_1)$ is concerned, the possible difference from the dual assignments given in Sect. 3 can arise only in B_1 as pointed out already. Yet, B_1 consists of singleton components of $G_1[D_1]$ when seen from inside G_1 , and $y_u \leq 1, \forall u \in B_1 \cap V_2$ by Lemma 6 (or $y_u = 2/3$ if $u \in B_1 - V_2$). Thus, the inequalities in Lemma 3 still hold, and the analysis for Theorem 4 still works. $\square$

Finally, we have

Theorem 8. *The performance guarantee of MG-3SC for 3-SC is at most $H(3) - 1/6$.*

Proof. To prove the performance ratio of $H(3) - 1/6$ (using Proposition 1), it remains, due to Lemma 7, only to show that $y(S) \leq 2(H(3) - 1/6) = 10/3$ for any $S \in \mathcal{F}_2$ of cost-2. Recall from Sect. 3 that $y_u \leq 2/3$ for each $u \in U_1 - B_1$ (Note: For $u \in U_1 \cap \overline{\mathcal{F}}_2$, y_u could exceed $2/3$ exactly when $u \in B_1$). Also, $y_u = 2/3, \forall u \in \overline{P}_2$. Since U is partitioned into $U_1 - B_1, \overline{P}_2$, and V_2 , $y(S - V_2) \leq 2|S - V_2|/3$. So, $y(S) \leq 2$ if $S \cap V_2 = \emptyset$, and, by Lemma 6, $y(S) \leq 4/3 + 2 = 10/3$ if $|S \cap V_2| = 1$ while $y(S) \leq 2/3 + 8/3$ if $|S \cap V_2| = 2$. The claims thus follows in either case. $\square$

4.3 Algorithm for k -Set Cover

One simple strategy for handling i -sets with $i > 3$ is to extend the construction of maximal 3-set packings in Steps 1 and 7 of MG-3SC to that of maximal i -set packings. To do so, we run the standard greedy procedure to process larger

subsets, and eventually switch into the operation mode of MG-3SC. When do we switch then? Suppose we let the standard greedy algorithm to pick first all and *only* the cost-1 i -sets with $i \geq 3$ and the cost-2 j -sets with $j \geq 5$. We then initiate MG-3SC to cover the remainder of elements, but a maximal cost-1 3-set packing is already taken at this point, and thus Step 1 is to be skipped. Moreover, there could remain 4-sets of cost-2, and we modify Step 7 of MG-3SC to incorporate them, so that a maximal 4-set packing of cost-2 subsets is constructed before a maximal 3-set packing; let P_2 denote now the union of these set packings.

How large could $y(S)$ become now? Let S be a cost-1 k -set first, and suppose $S^j \subseteq S$ is a j -set remaining uncovered after the standard greedy part, where $0 \leq j \leq 2$. If $S^j = \emptyset$, $y(S) \leq (\sum_{i=3}^k 1/i) + 2(2/5) = H(k) - 7/10$, since $y_u \leq 2/5$ for any u covered by the standard greedy part. If $j = 2$, S^j becomes an edge of G_1 as before, and hence, $y(S^j) \leq 4/3$, even after the modification in Step 7 of MG-3SC (the existence of cost-2 4-sets can only possibly lower its value). Meanwhile, $y(S - S^j) \leq \sum_{i=3}^k 1/i$, and thus, $y(S) \leq (\sum_{i=3}^k 1/i) + 4/3 \leq H(k) - 1/6$. Similarly when $j = 1$, $y(S - S^j) \leq (\sum_{i=3}^k 1/i) + 2/5$, and $y(S^j) \leq 1$, totaling to $y(S) \leq H(k) - 1/10$.

Suppose next S is a cost-2 k -set, and in this case, S^j could be a j -set for $0 \leq j \leq 4$, when the standard greedy procedure terminates. It is straightforward though to verify that, if $j \leq 3$, $y(S - S^j) \leq (\sum_{i=5}^k 2/i) + 2/5 \leq 2(H(k) - H(3))$ and $y(S^j) \leq 2(H(3) - 1/6)$ as before, resulting in $y(S) \leq 2(H(k) - 1/6)$. So assume that S^j is a 4-set, and observe that, as soon as S^j becomes a 3-set S^3 during the MG-3SC part, the analysis of Sect. 4.2 applies and $y(S^3) \leq 2(H(3) - 1/6)$. So, let $u \in S^j$ be the first element covered by MG-3SC. Then, if u remains uncovered at the initiation of the maximal cost-2 4-set packing construction in Step 7, it receives $y_u = 2/4$. On the other hand, if u is already covered as a vertex of G_1 by a cost-1 set, y_u could be as large as $2/3$. Therefore, $y(S^j) \leq 2/3 + 2(H(3) - 1/6) = 2(H(4) - 1/12)$, and $y(S) \leq 2(\sum_{i=5}^k 1/i) + 2(H(4) - 1/12) = 2(H(k) - 1/12)$.

The standard greedy algorithm can be implemented to run in time $O(\sum_{S \in \mathcal{F}} |S|)$. The running time for computing a maximum *cardinality* matching (and the Gallai-Edmonds decomposition) is known to be $O(|E|\sqrt{|V|})$ for a graph $G = (V, E)$ [17]. After all, we have shown that

Theorem 9. *The modified greedy algorithm, given above, computes a set cover for k -SC of cost bounded by $H(k) - 1/12$ times the optimal cost in time $O(\sum_{S \in \mathcal{F}} |S| + \min\{|\mathcal{F}|, |U|^2\} \cdot \sqrt{U})$.*

The *integrality gap* of (LP) is the maximum ratio, over all instances of k -SC, of the optimal value of (IP) to that of (LP). Since our analysis is throughout based on the LP duality, we additionally have

Corollary 10. *The integrality gap of (LP) with $c_S \in \{1, 2\}, \forall S \in \mathcal{F}$, is bounded by $\frac{5}{3}$ for 3-SC and by $H(k) - 1/12$ for k -SC with $k \geq 4$.*

References

1. Arkin, E. M., Hassin, R.: On local search for weighted k -set packing. In: Proc. ESA '97. LNCS #1284. Springer-Verlag (1997) 13–22 671
2. Bafna, V., Narayanan, B., Ravi, R.: Nonoverlapping local alignments (Weighted independent sets of axis-parallel rectangles). Discrete Appl. Math. **71** (1996) 41–53 671
3. Chandra, B., Halldórsson, M. M.: Greedy local improvement and weighted set packing approximation. In: Proc. 10th Annual ACM-SIAM Symp. Discrete Algorithms. ACM-SIAM (1999) 169–176 671
4. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3) (1979) 233–235 671, 673
5. Duh, R., Fürer, M.: Approximation of k -set cover by semi-local optimization. In: Proc. 29th Annual ACM Symp. Theory of Computing. ACM (1997) 256–264 671
6. Edmonds, J.: Paths, trees and flowers. Canad. J. Math. **17** (1965) 449–467 672, 673
7. Feige, U.: A threshold of $\ln n$ for approximating set cover. In: Proc. 28th Annual ACM Symp. Theory of Computing. ACM (1996) 314–318 671
8. Gallai, T.: Kritische Graphen II. Magyar Tud. Akad. Mat. Kutató Int. Közl. **8** (1963) 373–395 672, 673
9. Gallai, T.: Maximale Systeme unabhängiger Kanten. Magyar Tud. Akad. Mat. Kutató Int. Közl. **9** (1964) 401–413 672, 673
10. Goldschmidt, O., Hochbaum, D. S., Yu, G.: A modified greedy heuristic for the Set Covering problem with improved worst case bound. Inform. Process. Lett. **48** (1993) 305–310 671, 673
11. Halldórsson, M. M.: Approximating discrete collections via local improvements. In: Proc. 6th Annual ACM-SIAM Symp. Discrete Algorithms. ACM-SIAM (1995) 160–169 671
12. Halldórsson, M. M.: Approximating k -set cover and complementary graph coloring. In: Proc. IPCO V. LNCS #1084. Springer-Verlag (1996) 118–131 671
13. Hurkens, C. A. J., Schrijver, A.: On the size of systems of sets every t of which have an SDR, with an application to the worst-case ratio of heuristics for packing problems. SIAM J. Discrete Math. **2**(1) (1989) 68–72 671
14. Johnson, D. S.: Approximation algorithms for combinatorial problems. J. Comput. System Sci. **9** (1974) 256–278 670
15. Karp, R. M.: Reducibility among combinatorial problems. In: Miller, R. E., Thatcher, J. W. (eds.): Complexity of Computer Computations. Plenum Press, New York (1972) 85–103 670
16. Lovász, L.: On the ratio of optimal integral and fractional covers. Discrete Math. **13** (1975) 383–390 670
17. Micali, S. and Vazirani, V. V.: An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In: Proc. 21st Annual Symp. Foundations of Computer Science. IEEE (1980) 17–27 680
18. Papadimitriou, C., Yannakakis, M.: Optimization, approximation and complexity classes. J. Comput. System Sci. **43** (1991) 425–440 670
19. Slavík, P.: A tight analysis of the greedy algorithm for set cover. In: Proc. 28th Annual ACM Symp. Theory of Computing. ACM (1996) 435–441 671

A Unified Framework for Approximating Multiway Partition Problems (Extended Abstract)

Liang Zhao¹, Hiroshi Nagamochi², and Toshihide Ibaraki¹

¹ Kyoto University
Kyoto, Japan

{zhao, ibaraki}@amp.i.kyoto-u.ac.jp

² Toyohashi University of Technology
Toyohashi, Japan
naga@ics.tut.ac.jp

Abstract. We present a simple and unified framework for developing and analyzing approximation algorithms for some multiway partition problems (with or without terminals), including the *k-way cut* (or *k-cut*), *multiterminal cut* (or *multiway cut*), *hypergraph partition* and *target split*.

1 Introduction

Let V and $f: 2^V \rightarrow \mathbf{R}$ be a finite set and a set function respectively. Function f is *submodular* if $f(A) + f(B) \geq f(A \cap B) + f(A \cup B)$ for all subsets A and B of V . It is *symmetric* if $f(S) = f(V - S)$ for all $S \subseteq V$. A family $\mathcal{P} = \{V_1, \dots, V_k\}$ of pairwise disjoint nonempty subsets of V whose union is V is called a *k-partition* of V . The cost of $\mathcal{P}$ (with respect to f) is defined as $f(\mathcal{P}) = \sum_{i=1}^k f(V_i)$. Given a submodular system (V, f) where f is nonnegative, the *k-partition problem in submodular systems* (*k-PPSS*) is to find a *k-partition* of V with the minimum cost ($2 \leq k \leq |V| - 1$). The *k-partition problem in symmetric submodular systems* (*k-PP3S*) is the *k-PPSS* with symmetric f . In this paper, we assume that f is given by an oracle that computes $f(S)$ in at most θ time for any $S \subseteq V$.

The *k-partition problem in hypergraphs* (*k-PPH*) is a special case of the *k-PP3S* in which V and f are respectively the vertex set and the *cut function* of a hypergraph with nonnegative hyperedge costs (i.e., for any $S \subseteq V$, $f(S)$ is the sum of costs of hyperedges that have at least one but not all endpoints in S . It is nonnegative, symmetric and can be easily seen to be submodular).

Our study starts from the *k-partition problem in graphs* (*k-PPG*). Given an undirected graph with nonnegative edge costs, the *k-PPG* asks to find a minimum cost edge subset whose removal leaves the graph with at least k connected components. The problem is also called the *k-way cut* or *k-cut problem*. Goldschmidt and Hochbaum [4] have shown that the *k-PPG* is NP-hard for arbitrary k even for unit edge costs, while it is solvable for any fixed k in $O(n^{k^2})$ time, where n is the number of vertices. Faster algorithms can be found in [9, 13, 14].

Saran and Vazirani [19] and Kapoor [7] showed that the k -PPG problem (for arbitrary k) can be approximated within factor $2 - \frac{2}{k}$ in polynomial time. Recently, the authors [20] have given an approximation algorithm with improved performance guarantee about $2 - \frac{3}{k}$.

It is easy to see that the inclusion among the above problem classes is k -PPG $\subset k$ -PPH $\subset k$ -PP3S $\subset k$ -PPSS. Hence all of these are also NP-hard for arbitrary k . Queyranne [17] has shown that for any fixed k the k -PP3S is solvable in $O(|V|^{k^2}\theta)$ time. A faster algorithm for the 3-PP3S can be found in [13]. On the other hand, Queyranne [17] extends the greedy algorithm in [7,19] to show that the k -PP3S can be approximated within factor $2 - \frac{2}{k}$ in polynomial time. We note that his proof and the proofs of [7,19] all use lower bounds derived from the so-called cut tree (or Gomory-Hu tree) for f (or for undirected graph), and are rather complicated and work only for symmetric submodular systems. As will be seen in the following, our approach in this paper works for any submodular system and gives a much simpler proof to show the same results.

We first show that the 2-PPSS is solvable in $O(|V|^3\theta)$ time, while we leave it open whether the k -PPSS can be solved in polynomial time for fixed $k \geq 3$. We then extend the greedy algorithm in [7,17,19] to the k -PPSS. It finds a k -partition of V by greedily “splitting” V via minimum 2-partition computations. We will give a simple proof to show that the performance guarantee is no worse than $(1 + \alpha)(1 - \frac{1}{k})$, where α is any number that satisfies $\sum_{i=1}^k f(V - V_i) \leq \alpha \sum_{i=1}^k f(V_i)$ for all k -partitions $\{V_1, \dots, V_k\}$ of V . We show that in general we can let $\alpha = k - 1$, which implies the performance guarantee $k - 1$. This is the first approximation algorithm for the k -PPSS. Furthermore, it is clear that we can let $\alpha = 1$ if f is symmetric, which implies the results of [7,17,19]. Several more applications of the results will also be given.

We next consider to approximate the k -PPSS via minimum 2, 3-partition computations. We will show some properties on the performance and use them to approximate the k -PPH by factor about $2 - \frac{3}{k}$. This extends our result [20] for the k -PPG and improves the previous best bound $2 - \frac{2}{k}$ (implied by the result for the k -PP3S due to Queyranne [17]).

Finally we extend our results to the *target split problem in submodular systems* (TSPSS), which for an additional given *target set* $T \subseteq V$ ($|T| \geq k$) asks to find a minimum k -partition $\{V_1, V_2, \dots, V_k\}$ such that each V_i contains at least one target in T . A special case in which $|T| = k$, V and f are respectively the vertex set and the cut function of graphs is called the *multiterminal* (or previously *multiway*) *cut problem*, which is NP-hard even for $k = 3$ [3], and can be approximated within factor $1.5 - \frac{1}{k}$ [1], 1.3438 [8]. Clearly the TSPSS is a generalization of the k -PPSS and the multiterminal cut problem. We note that Maeda, Nagamochi and Ibaraki [12] have considered the target split problem in graphs and shown that it can be approximated within factor $2 - \frac{2}{k}$ in polynomial time. Our result will also give a simpler proof to show their result.

2 k -PPSS and Greedy Splitting Algorithm

We first observe that the 2-PPSS is solvable in $O(|V|^3\theta)$ time.

Theorem 1. (Queyranne [16]) *Given a symmetric submodular function $g: 2^V \rightarrow \mathbf{R}$, a nonempty proper subset S^* of V ($|V| \geq 2$) such that $g(S^*)$ is minimum can be found in $O(|V|^3\theta_g)$ time where θ_g is the time bound of the oracle for g . $\square$*

Theorem 2. *Given a submodular function $f: 2^V \rightarrow \mathbf{R}$ and a $W \subseteq V$ ($|W| \geq 2$), a nonempty proper subset S^* of W such that $f(S^*) + f(W - S^*)$ is minimum can be found in $O(|W|^3\theta)$ time where θ is the time bound of the oracle for f .*

Proof. Let $g: 2^W \rightarrow \mathbf{R}$ be defined by $g(S) = f(S) + f(W - S)$ for all $S \subseteq W$. Notice that g is symmetric, submodular and for any $S \subseteq W$ we can compute $g(S)$ in at most 2θ time. Theorem 1 shows that such an S^* can be found in $O(|W|^3\theta)$ time. $\square$

(Notice that f is not needed to be nonnegative in Theorem 2.) We next present a *greedy splitting* approximation algorithm (GSA) for the k -PPSS in Table 1.

Table 1. Greedy splitting algorithm (GSA) for the k -PPSS.

```

1  $\mathcal{P}_1 \leftarrow \{V\}$ ;
2 for  $i = 1, \dots, k - 1$  do
3    $(S_i, W_i) \leftarrow \operatorname{argmin}\{f(S) + f(W - S) - f(W) \mid \emptyset \neq S \subset W, W \in \mathcal{P}_i\}$ ;
4    $\mathcal{P}_{i+1} \leftarrow (\mathcal{P}_i - \{W_i\}) \cup \{S_i, W_i - S_i\}$ ;
5 end /* for */
6 Output  $\mathcal{P}_k$ .
```

GSA contains $k - 1$ rounds and the i -th round computes an $(i + 1)$ -partition $\mathcal{P}_{i+1}$ of V , where $\mathcal{P}_1 = \{V\}$ and $\mathcal{P}_{i+1}$ is obtained by greedily “splitting” some member in $\mathcal{P}_i$ into two nonempty parts at the minimum cost. Formally, in the i -th round we compute a pair (S_i, W_i) that minimizes $f(S) + f(W - S) - f(W)$ (called the *splitting cost*) over all S and W such that $\emptyset \neq S \subset W$ and $W \in \mathcal{P}_i$. We get $\mathcal{P}_{i+1}$ from $\mathcal{P}_i$ by replacing W_i with S_i and $W_i - S_i$. Thus, for $\ell = 1, 2, \dots, k$, it holds

$$f(\mathcal{P}_\ell) = f(V) + \sum_{i=1}^{\ell-1} (f(S_i) + f(W_i - S_i) - f(W_i)). \quad (1)$$

Clearly the output $\mathcal{P}_k$ of GSA is a k -partition of V . For any (fixed) $W \subseteq V$, Theorem 2 shows that we can find in $O(|W|^3\theta)$ time a nonempty proper subset S^* of W such that $f(S^*) + f(W - S^*)$ (hence $f(S^*) + f(W - S^*) - f(W)$) is minimum. Thus we can execute step 3 in $\sum_{W \in \mathcal{P}_i} O(|W|^3\theta) = O(\sum_{W \in \mathcal{P}_i} |W|^3\theta) = O(|V|^3\theta)$ time. Hence the running time of GSA is $O(k|V|^3\theta)$.

To analyze the performance guarantee, we first go through a technical lemma.

Lemma 1. *For an $\ell \in \{1, \dots, k\}$, let $\mathcal{P}_\ell$ be the ℓ -partition of V found by GSA in the $(\ell - 1)$ -th round. Then for any ordered ℓ -partition $\{V_1, V_2, \dots, V_\ell\}$ of V , it holds that*

$$f(\mathcal{P}_\ell) \leq \sum_{i=1}^{\ell-1} (f(V_i) + f(V - V_i)) - (\ell - 2)f(V). \quad (2)$$

Proof. We proceed by induction on ℓ . It is trivial for $\ell = 1$. Suppose that it holds for $\ell - 1$. Consider an ordered ℓ -partition $\mathcal{P} = \{V_1, V_2, \dots, V_\ell\}$ of V . Since $\mathcal{P}_{\ell-1}$ is an $(\ell - 1)$ -partition of V , there exist a $W \in \mathcal{P}_{\ell-1}$ and two distinct $V_j, V_h \in \mathcal{P}$ with $j < h$ such that $W \cap V_j \neq \emptyset \neq W \cap V_h$. We here consider the ordered $(\ell - 1)$ -partition $\mathcal{P}' = \{V_1, \dots, V_{j-1}, V_{j+1}, \dots, V_{\ell-1}, V_j \cup V_h\}$ where the order is the same as $\mathcal{P}$ except for that V_j is merged with the last member V_ℓ (notice $j < \ell$). By the induction hypothesis on $\ell - 1$, (2) holds for $\mathcal{P}_{\ell-1}$ and $\mathcal{P}'$, i.e.,

$$f(\mathcal{P}_{\ell-1}) \leq \sum_{1 \leq i \leq \ell-1, i \neq j} (f(V_i) + f(V - V_i)) - (\ell - 3)f(V). \quad (3)$$

Thus by (3) it suffices to show that

$$f(\mathcal{P}_\ell) - f(\mathcal{P}_{\ell-1}) \leq f(V_j) + f(V - V_j) - f(V). \quad (4)$$

Notice that $W \cap V_j$ is a nonempty proper subset of W . Thus $(W \cap V_j, W)$ is a candidate for step 3 of GSA. Hence by the optimality of $(S_{\ell-1}, W_{\ell-1})$,

$$f(S_{\ell-1}) + f(W_{\ell-1} - S_{\ell-1}) - f(W_{\ell-1}) \leq f(W \cap V_j) + f(W - V_j) - f(W). \quad (5)$$

The submodularity of f implies that the right hand of (5) is at most

$$f(V_j) + f(W - V_j) - f(W \cup V_j) \leq f(V_j) + f(V - V_j) - f(V),$$

proving (4). $\square$

Theorem 3. *Given a nonnegative submodular system (V, f) , GSA finds a k -partition of V of cost at most $(1 + \alpha)(1 - \frac{1}{k})$ times the optimum, where α is any number that satisfies $\sum_{i=1}^k f(V - V_i) \leq \alpha \sum_{i=1}^k f(V_i)$ for all k -partitions $\{V_1, \dots, V_k\}$ of V .*

Proof. Let $\mathcal{P}^* = \{V_1^*, V_2^*, \dots, V_k^*\}$ be an optimal k -partition of V with the order such that $f(V_k^*) + f(V - V_k^*) = \max_{1 \leq i \leq k} \{f(V_i^*) + f(V - V_i^*)\}$. Then

$$\begin{aligned} \sum_{i=1}^{k-1} (f(V_i^*) + f(V - V_i^*)) &\leq (1 - \frac{1}{k}) \sum_{i=1}^k (f(V_i^*) + f(V - V_i^*)) \\ &\leq (1 + \alpha)(1 - \frac{1}{k}) \sum_{i=1}^k f(V_i^*). \end{aligned}$$

On the other hand, by Lemma 1 GSA finds a k -partition of cost at most $\sum_{i=1}^{k-1} (f(V_i^*) + f(V - V_i^*))$ (note $f(V) \geq 0$). Hence the proof goes because $\sum_{i=1}^k f(V_i^*)$ is the optimum. $\square$

For symmetric f , we can let $\alpha = 1$ and obtain the following corollaries.

Corollary 1. (Queyranne [17]) *The k -PP3S can be approximated within factor $2 - \frac{2}{k}$ in polynomial time.* $\square$

Corollary 2. (Saran and Vazirani [19], Kapoor [7]) *The k -PPG problem can be approximated within factor $2 - \frac{2}{k}$ in polynomial time.* $\square$

In general we cannot let $\alpha = 1$. Nevertheless, we show that $\alpha = k - 1$ is enough.

Lemma 2. $\sum_{i=1}^k f(V - V_i) \leq (k - 1) \sum_{i=1}^k f(V_i) - k(k - 2)f(\emptyset)$ holds for any k -partition $\{V_1, \dots, V_k\}$ of a submodular system (V, f) .

Proof. For any two disjoint subsets $A, B \subseteq V$, $f(A \cup B) \leq f(A) + f(B) - f(\emptyset)$ holds by the submodularity of f . Thus $f(V - V_i) = f(\bigcup_{j \neq i} V_j) \leq \sum_{j \neq i} f(V_j) - (k - 2)f(\emptyset)$ for $i = 1, \dots, k$. Hence the lemma goes. $\square$

Notice that $f(\emptyset) \geq 0$ in the k -PPSS, which implies that $\alpha = k - 1$ is enough. Thus the performance guarantee of GSA for the k -PPSS is no worse than $k - 1$. We remark that the bound is also tight (a tight example will be given in the full paper). By summarizing the arguments so far we establish the next theorem.

Theorem 4. *The k -PPSS can be approximated within factor $k - 1$ in $O(k|V|^3\theta)$ time for any nonnegative submodular system (V, f) , where θ is the time bound of the oracle for f .* $\square$

Our proof is not only very simple but also allows us to plug some approximate algorithms into GSA. Suppose that a ρ -approximation algorithm for 2-PPSS is used. It is easy to see that the cost of the obtained k -partition is bounded by $\rho(1 + \alpha)(1 - \frac{1}{k})$ times the optimum.

Theorem 5. *The variant of GSA that uses a ρ -approximation algorithm for 2-PPSS to compute 2-partitions is a $\rho(1 + \alpha)(1 - \frac{1}{k})$ approximation algorithm for the k -PPSS, where α is any number that satisfies $\sum_{i=1}^k f(V - V_i) \leq \alpha \sum_{i=1}^k f(V_i)$ for all k -partitions $\{V_1, \dots, V_k\}$ of V .* $\square$

As a result, we obtain the next corollary by using the linear time $(2 + \epsilon)$ -approximation algorithm [11] for minimum cut problem in graphs with unit edge costs, where $\epsilon \in (0, 1)$ is an arbitrary number.

Corollary 3. *The k -PPG in graphs with unit edge costs can be approximated within factor $(4 + \epsilon)(1 - \frac{1}{k})$ in $O(k(n + m))$ time, where $\epsilon \in (0, 1)$ is a fixed number, and n and m are the numbers of vertices and edges respectively.* $\square$

Before closing this section, we show important applications of our results to two variants of the k -PPH that arise from VLSI design [2, 10]. Let $H = (V, E)$ be a hypergraph with vertex set V and hyperedge set E . Let $c: E \rightarrow \mathbf{R}^+$ be a nonnegative hyperedge cost function. For a k -partition $\mathcal{P}$ of V , two types of cost to be minimized, $cost_1(\mathcal{P})$ and $cost_2(\mathcal{P})$, are introduced: $cost_1(\mathcal{P})$ counts

the cost $c(e)$ of each hyperedge e $p - 1$ times if its endpoints of e belong to p distinct members in $\mathcal{P}$, while $\text{cost}_2(\mathcal{P})$ counts $c(e)$ once if its endpoints e belong to at least two distinct members in $\mathcal{P}$. (Recall that $f(\mathcal{P})$ in the k -PPH counts $c(e)$ p times if its endpoints of e belong to $p \geq 2$ distinct members in $\mathcal{P}$.)

For the k -PPH with cost functions cost_1 and cost_2 , the previous best approximation guarantees are $2 - \frac{2}{n}$ and $d_{\max}(1 - \frac{1}{n})$ respectively [15], where $d_{\max}$ is the maximum degree of hyperedges. We here show that better guarantees can be obtained by a simpler proof than [15]. For this, we define three set functions f_{ex} , f_{in} and $f: 2^V \rightarrow \mathbf{R}^+$ as follows. Let f_{ex} be the cut function of H . For any $S \subseteq V$, let $f_{\text{in}}(S)$ be the sum of costs of hyperedges whose endpoints are all in S , and $f(S) = f_{\text{ex}}(S) + f_{\text{in}}(S)$. Observe that the k -PPH with cost function cost_1 asks to find a k -partition $\mathcal{P} = \{V_1, V_2, \dots, V_k\}$ of V that minimizes $\sum_{i=1}^k f(V_i) - f(V) = \sum_{i=1}^k (f(V_i) - \frac{f(V)}{k})$, while the k -PPH with cost function cost_2 asks to minimize $f_{\text{in}}(V) - \sum_{i=1}^k f_{\text{in}}(V_i) = \sum_{i=1}^k (\frac{f_{\text{in}}(V)}{k} - f_{\text{in}}(V_i))$. It is easy to see that both functions $g_1 = f - \frac{f(V)}{k}$ and $g_2 = \frac{f_{\text{in}}(V)}{k} - f_{\text{in}}$ are submodular, but may not be nonnegative or symmetric. Nevertheless, since both Theorem 2 and Lemma 1 do not require the function to be nonnegative or symmetric, we can still use GSA to find a k -partition and use Lemma 1 to estimate the performance. By easy calculations, we can enjoy the next performance guarantees.

Corollary 4. *The k -PPH with cost function cost_1 (resp., cost_2) can be approximated within factor $2 - \frac{2}{k}$ (resp., $\min(k, d_{\max}^+)(1 - \frac{1}{k})$) in polynomial time, where $d_{\max}^+$ is the maximum degree of hyperedges with positive cost. $\square$*

3 Greedy Splitting via Minimum 2,3-Partitions

We have seen the GSA that increases the number of partitions one by one via minimum 2-partition computations. In this section we consider to increase the number of partitions two by two greedily. Let $k = 2m + 1 \geq 3$ be an odd number. (The case that k is an even number will be treated later.) We consider the next approximation algorithm for the k -PPSS. GSA2 (Table 2) contains m rounds and the i -th round constructs an $(2i + 1)$ -partition $\mathcal{P}_{i+1}$ of V , where $\mathcal{P}_1 = \{V\}$, and the $(2i + 1)$ -partition $\mathcal{P}_{i+1}$ is obtained by greedily “splitting” some member(s) in $\mathcal{P}_i$ at the minimum cost. There are two ways of such splitting. One is to split two members into four, which is considered by step 3. Another is to split one member into three, which is considered by step 4. We choose the one with the minimum cost to get $\mathcal{P}_{i+1}$ from $\mathcal{P}_i$ (step 5 – 8).

Clearly the output $\mathcal{P}_{m+1}$ is a k -partition of V . Let us consider the running time. In step 3 the objective is minimized by the least two minimum 2-partitions of members in $\mathcal{P}_i$. Thus by Theorem 2 step 3 can be done in $\sum_{W \in \mathcal{P}_i} O(|W|^3 \theta) = O(|V|^3 \theta)$ time. However, by now we do not know how to find a minimum 3-partition in submodular systems, which means that the time complexity of step 4 is still unknown in general. Therefore we suppose that the input (V, f) satisfies the next condition, which ensures that GSA2 runs in polynomial time.

Table 2. Greedy splitting algorithm 2 (GSA2) for the k -PPSS with odd $k = 2m + 1$.

```

1  $\mathcal{P}_1 \leftarrow \{V\};$ 
2 for  $i = 1, \dots, m$  do
3    $(S_i^1, W_i^1, S_i^2, W_i^2) \leftarrow \operatorname{argmin}\{\sum_{j=1}^2 (f(S_i^j) + f(W_i^j - S_i^j) - f(W_i^j)) \mid$ 
    $\emptyset \neq S_i^j \subset W_i^j, j = 1, 2, \text{ for distinct } W^1, W^2 \in \mathcal{P}_i\};$ 
4    $(T_i^1, T_i^2, W_i^3) \leftarrow \operatorname{argmin}\{f(T_i^1) + f(T_i^2) + f(W_i^3 - T_i^1 - T_i^2) - f(W_i^3) \mid$ 
    $\{T_i^1, T_i^2, W_i^3 - T_i^1 - T_i^2\} \text{ is a 3-partition of some } W \in \mathcal{P}_i\};$ 
5   if  $\sum_{j=1}^2 (f(S_i^j) + f(W_i^j - S_i^j) - f(W_i^j))$ 
    $< f(T_i^1) + f(T_i^2) + f(W_i^3 - T_i^1 - T_i^2) - f(W_i^3)$  then
6      $\mathcal{P}_{i+1} \leftarrow (\mathcal{P}_i - \{W_i^1, W_i^2\}) \cup \{S_i^1, W_i^1 - S_i^1, S_i^2, W_i^2 - S_i^2\};$ 
7   else
8      $\mathcal{P}_{i+1} \leftarrow (\mathcal{P}_i - \{W_i^3\}) \cup \{T_i^1, T_i^2, W_i^3 - T_i^1 - T_i^2\};$ 
9   end /* if */
10 end /* for */
11 Output  $\mathcal{P}_{m+1}$ .
```

Condition 1. For any $W \subseteq V$, a 3-partition $\{T^1, T^2, W - T^1 - T^2\}$ of W that minimizes $f(T^1) + f(T^2) + f(W - T^1 - T^2)$ can be found in polynomial time.

To analyze the performance of GSA2, we show a lemma analogous with Lemma 1.

Lemma 3. For an $\ell \in \{0, 1, \dots, m\}$, let $\mathcal{P}_{\ell+1}$ be the $(2\ell+1)$ -partition of V found by GSA2 in the ℓ -th round. Then for any ordered $(2\ell+1)$ -partition $\{V_1, V_2, \dots, V_{2\ell+1}\}$ of V , it holds that

$$f(\mathcal{P}_{\ell+1}) \leq \sum_{i=1}^{\ell} (f(V_{2i-1}) + f(V_{2i}) + f(V - V_{2i-1} - V_{2i})) - (\ell - 1)f(V). \quad (6)$$

Proof. We proceed by induction on ℓ . It is trivial when $\ell = 0$. Suppose that it holds for $\ell - 1$. Consider an ordered $(2\ell + 1)$ -partition $\mathcal{P} = \{V_1, V_2, \dots, V_{2\ell+1}\}$ of V . Since $\mathcal{P}_\ell$ is a $(2\ell - 1)$ -partition of V , we see that at least one of the next two cases occurs for $\mathcal{P}$ and $\mathcal{P}_\ell$.

1. There is a $W^1 \in \mathcal{P}_\ell$ and three distinct $V_r, V_s, V_t \in \mathcal{P}$ ($r < s < t$) such that $W^1 \cap V_r \neq \emptyset$, $W^1 \cap V_s \neq \emptyset$, and $W^1 \cap V_t \neq \emptyset$.
2. There are two distinct $W^1, W^2 \in \mathcal{P}_\ell$ and four distinct $V_a, V_b, V_p, V_q \in \mathcal{P}$ ($a < b, p < q, a < p$) such that $W^1 \subseteq V_a \cup V_b$, $W^2 \subseteq V_p \cup V_q$, $W^1 \cap V_a \neq \emptyset \neq W^1 \cap V_b$ and $W^2 \cap V_p \neq \emptyset \neq W^2 \cap V_q$.

In case 1, we further consider the following two sub-cases.

- 1a. There is an $h \in \{1, \dots, \ell\}$ such that $r = 2h - 1$ and $s = 2h$.
- 1b. Otherwise $r \in \{2h - 1, 2h\}$ and $s \in \{2h' - 1, 2h'\}$ for some $1 \leq h < h' \leq \ell$.

Similarly, we consider the next two sub-cases in case 2.

- 2a. There is an $h \in \{1, \dots, \ell\}$ such that $a = 2h - 1$ and $p = 2h$.
2b. Otherwise $a \in \{2h - 1, 2h\}$ and $p \in \{2h' - 1, 2h'\}$ for some $1 \leq h < h' \leq \ell$.

We will show that in each sub-case of 1a, 1b, 2a, 2b, there is a “nice splitting” which is a candidate for step 3 or 4 of GSA2. (Recall that the cost of any “nice splitting” is an upper bound on $f(\mathcal{P}_{\ell+1}) - f(\mathcal{P}_\ell)$.) We show that we can construct an ordered $(2\ell - 1)$ -partition $\mathcal{P}' = \{V'_1, \dots, V'_{2\ell-1}\}$ of V from $\mathcal{P}$ such that $\sum_{i=1}^{\ell-1} (f(V'_{2i-1}) + f(V'_{2i}) + f(V - V'_{2i-1} - V'_{2i})) - (\ell - 2)f(V)$ plus the cost of the “nice splitting” is at most the right hand of (6). This will prove the lemma by the induction hypothesis on $\mathcal{P}'$.

In what follows, we only consider sub-case 2a due to space limitation (the other cases can be shown analogously). Let $\mathcal{P}'$ be the ordered $(2\ell - 1)$ -partition $\{V_1, \dots, V_{2h-2}, V_{2h+1}, \dots, V_{2\ell}, V_{2h-1} \cup V_{2h} \cup V_{2\ell+1}\}$ of V , which has the same order as $\mathcal{P}$ except for that V_{2h-1} and V_{2h} are merged with the last member $V_{2\ell+1}$ (notice $2h - 1 < 2h < 2\ell + 1$). By the induction hypothesis on $\ell - 1$, (6) holds for $\mathcal{P}_\ell$ and $\mathcal{P}'$, i.e.,

$$f(\mathcal{P}_\ell) \leq \sum_{1 \leq i \leq \ell, i \neq h} (f(V_{2i-1}) + f(V_{2i}) + f(V - V_{2i-1} - V_{2i})) - (\ell - 2)f(V).$$

Thus, it suffices to show

$$f(\mathcal{P}_{\ell+1}) - f(\mathcal{P}_\ell) \leq f(V_{2h-1}) + f(V_{2h}) + f(V - V_{2h-1} - V_{2h}) - f(V). \quad (7)$$

For this, we choose $(W^1 \cap V_{2h-1}, W^1, W^2 \cap V_{2h}, W^2)$ as the “nice splitting”, i.e., split W^1 and W^2 into $W^1 \cap V_{2h-1}$, $W^1 - V_{2h-1}$ and $W^2 \cap V_{2h}$, $W^2 - V_{2h}$ respectively. Clearly it is a candidate for $(S_\ell^1, W_\ell^1, S_\ell^2, W_\ell^2)$ in step 3 of GSA2 (see Table 2). Therefore,

$$\begin{aligned} f(\mathcal{P}_{\ell+1}) - f(\mathcal{P}_\ell) &\leq f(W^1 \cap V_{2h-1}) + f(W^1 - V_{2h-1}) - f(W^1) \\ &\quad + f(W^2 \cap V_{2h}) + f(W^2 - V_{2h}) - f(W^2). \end{aligned} \quad (8)$$

By the submodularity, the right hand of (8) is at most

$$\begin{aligned} &f(W^1 \cap V_{2h-1}) - f(W^1) + f(W^2 \cap V_{2h}) + f(W^2 \cup V_{2h}) - f(W^2) - f(V) \\ &\quad + [f(W^1 - V_{2h-1}) + f(V - V_{2h})] \\ &\leq [f(W^1 \cap V_{2h-1}) + f(W^1 \cup V_{2h-1}) - f(W^1)] \\ &\quad + [f(W^2 \cap V_{2h}) + f(W^2 \cup V_{2h}) - f(W^2)] + f(V - V_{2h-1} - V_{2h}) - f(V) \\ &\leq f(V_{2h-1}) + f(V_{2h}) + f(V - V_{2h-1} - V_{2h}) - f(V), \end{aligned}$$

proving (7). $\square$

For an even $k = 2m \geq 2$, we start with a minimum 2-partition of V before increasing the number of partitions two by two greedily. It is described in Table 3, where the same code as in Table 2 are abbreviated. Clearly the output $\mathcal{P}_m$ is a k -partition of V . In order to be a polynomial time algorithm, it is again assumed that Condition 1 is satisfied. We give a lemma on the performance, where the proof can be done in a similar way as Lemma 3 and is omitted.

Table 3. Greedy splitting algorithm 2 (GSA2) for the k -PPSS with even $k = 2m$.

```

1  $\mathcal{P}_1 \leftarrow$  a minimum 2-partition of  $V$ ;
2 for  $i = 1, \dots, m - 1$  do
3 – 10 (The same as step 3 – 10 in Table 2);
11 Output  $\mathcal{P}_m$ .
```

Lemma 4. For an $\ell \in \{1, 2, \dots, m\}$, let $\mathcal{P}_\ell$ be the 2ℓ -partition of V found by GSA2 in the ℓ -th round. Then for any ordered 2ℓ -partition $\{V_1, V_2, \dots, V_{2\ell}\}$ of V , it holds that

$$f(\mathcal{P}_\ell) \leq f(V_1) + f(V - V_1) + \sum_{i=1}^{\ell-1} (f(V_{2i}) + f(V_{2i+1}) + f(V - V_{2i} - V_{2i+1})) - (\ell - 1)f(V). \quad (9)$$

□

We note that, not surprisingly, GSA2 does no worse than GSA in any cases. This can be seen by comparing the right hand of (2) and (6) or (9). Notice that $f(V - A - B) + f(V) \leq f(V - A) + f(V - B)$ for any disjoint subsets A and B of V . In fact, using Lemma 3 and Lemma 4, we have the next result.

Theorem 6. The performance guarantee of GSA2 is $\frac{k-1}{2}$ for the k -PPSS and $2 - \frac{2}{k}$ for the k -PP3S. There are examples that indicate these bounds are tight.

□

However, we know that GSA2 can do better for the k -PPG [20]. A question is, what can it guarantee to problem classes lying between the k -PPG and the k -PP3S e.g., the k -PPH. In the following, we show that GSA2 achieves a guarantee better than $2 - \frac{2}{k}$ for the k -PPH, extending the result for the k -PPG by [20].

Theorem 7. The k -PPH can be approximated in polynomial time within factor $2 - \frac{3}{k}$ for any odd $k \geq 3$ and factor $2 - \frac{3}{k} + \frac{1}{k^2 - k}$ for any even $k \geq 2$.

Proof. Let V and f be respectively the vertex set and the cut function of a hypergraph H . It is easy to see that Condition 1 is satisfied by considering the reduced hypergraph of H for any vertex subset $W \subseteq V$, where for each hyperedge e , the endpoints of e that are not in W are removed and e is present if it has at least two endpoints in W . Thus GSA2 (Table 2, 3) is a polynomial time approximation algorithm for the k -PPH. We next show the claimed performance guarantee. Let $\mathcal{P}^* = \{V_1^*, V_2^*, \dots, V_k^*\}$ be a minimum k -partition of V . Let π denote a numbering of $\{1, \dots, k\}$, and let $\pi(i)$ be the number of i .

First consider an odd number $k = 2m + 1 \geq 3$. By applying Lemma 3 to $\mathcal{P}^*$, we see that GSA2 finds a k -partition of V with cost at most $f_\pi = \sum_{i=1}^m (f(V_{\pi(2i-1)}^*) + f(V_{\pi(2i)}^*) + f(V - V_{\pi(2i-1)}^* - V_{\pi(2i)}^*))$ for any numbering π . We want to show that there is a numbering π^* such that f_{π^*} is no more than $2 - \frac{3}{k}$

times the optimum $f(\mathcal{P}^*) = \sum_{i=1}^k f(V_i^*)$. This can be done by considering all numberings and showing that the average value of f_π is at most $(2 - \frac{3}{k})f(\mathcal{P}^*)$.

Let us rewrite f_π as $2f(P^*) - \Delta_\pi$ where $\Delta_\pi = 2f(V_{\pi(k)}^*) + \sum_{i=1}^m (f(V_{\pi(2i-1)}^*) + f(V_{\pi(2i)}^*) - f(V - V_{\pi(2i-1)}^* - V_{\pi(2i)}^*))$. Thus we only need to show that the average value of Δ_π is at least $\frac{3}{k}f(\mathcal{P}^*)$. For each hyperedge e , we consider the average number that e is counted in Δ_π . For simplicity, let us contract each $V_i^* \in \mathcal{P}^*$ to a single node v_i (this may decrease the degree of e). Let $H|_{\mathcal{P}^\square}$ denote the contracted hypergraph. To avoid confusing we use the word “node” in $H|_{\mathcal{P}^\square}$ to denote the contracted vertex subsets. We assume without loss of generality that $H|_{\mathcal{P}^\square}$ is simple and complete. Otherwise we can meet this by merging the hyperedges with the same endpoints and adding zero cost hyperedges. Suppose that after contraction e has degree $d \geq 2$ (otherwise e is not counted in Δ_π).

Recall that $f(S)$ is the sum of costs of hyperedges that has at least one but not all endpoints in S for $S \subseteq V$. Thus due to the $2f(V_{\pi(k)}^*)$ term in Δ_π , e is counted twice if one endpoint of e is numbered k . Since $H|_{\mathcal{P}^\square}$ has k nodes and e has d endpoints, we see that the average number (expected value) that e is counted due to the $2f(V_{\pi(k)}^*)$ term is $\frac{2d}{k}$. On the other hand, due to the other term $\sum_{i=1}^m (f(V_{\pi(2i-1)}^*) + f(V_{\pi(2i)}^*) - f(V - V_{\pi(2i-1)}^* - V_{\pi(2i)}^*))$ in Δ_π , e is counted twice if $d = 2$ and the two endpoints of e are numbered $\pi(2i - 1)$ and $\pi(2i)$ for some $i \in \{1, 2, \dots, m\}$, otherwise e is counted p times if $d \geq 3$ and the endpoints of e contains p pairs of nodes that are numbered $\pi(2i_p - 1)$ and $\pi(2i_p)$ for some distinct $i_p \in \{1, 2, \dots, m\}$. Notice that for each fixed pair of indices $2i - 1$ and $2i$, the average number (probability) that both nodes v_{2i-1} and v_{2i} become endpoints of e is $\binom{k-2}{d-2} / \binom{k}{d} = \frac{d(d-1)}{k(k-1)}$. Thus the average number that e is counted due to the $\sum_{i=1}^m (f(V_{\pi(2i-1)}^*) + f(V_{\pi(2i)}^*) - f(V - V_{\pi(2i-1)}^* - V_{\pi(2i)}^*))$ term is $2 \cdot m \cdot \frac{2 \cdot (2-1)}{k(k-1)} = \frac{2}{k}$ if $d = 2$, or $m \cdot \frac{d(d-1)}{k(k-1)} = \frac{d(d-1)}{2k}$ if $d \geq 3$. Since e is counted d times in the optimum $f(\mathcal{P}^*)$, we see that the contribution of e to the average value of Δ_π is $\frac{1}{2}(\frac{4}{k} + \frac{2}{k}) = \frac{3}{k}$ ($d = 2$) or $\frac{1}{d}(\frac{2d}{k} + \frac{d(d-1)}{2k}) \geq \frac{3}{k}$ ($d \geq 3$) times the contribution to $f(\mathcal{P}^*)$. Thus we see that the average value of Δ_π is at least $\frac{3}{k}$ times $f(\mathcal{P}^*)$, which finishes the proof of the theorem for an odd k .

Similarly, we can prove the theorem for an even k . We note that the bounds are tight, see [20]. $\square$

4 Target Split Problem in Submodular Systems

Given a *target set* $T \subseteq V$ ($|T| \geq k$) as an additional input, the *target split problem in submodular systems* (TSPSS) is to find a minimum k -partition $\{V_1, V_2, \dots, V_k\}$ such that each V_i contains at least one target. By considering only those “valid” k -partitions (i.e., a target split of T), we extend our algorithms to the TSPSS.

Let us first consider algorithm GSA. In step 3 of GSA, we need to compute a valid 2-partition for some W in the current solution $\mathcal{P}_i$ at the minimum cost. This can be done if we can compute a minimum valid 2-partition for each $W \in \mathcal{P}_i$. We do this in the next way.

We do nothing with $W \in \mathcal{P}_i$ such that $|T \cap W| \leq 1$. For each $W \in \mathcal{P}_i$ with $|T \cap W| \geq 2$, we choose a target $s \in T \cap W$, compute a minimum 2-partition of W that separates s and t for each target $t \in T \cap W - \{s\}$, and choose the one with the minimum cost. We see that a minimum 2-partition of W that separates specified vertices s and t can be found in polynomial time.

Lemma 5. *Given a submodular system (V, f) and a $W \subseteq V$, for any $s, t \in W$ ($s \neq t$), a subset S^* of W such that $s \in S^*$, $t \notin S^*$ and $g(S^*) + g(W - S^*)$ is minimum can be found in polynomial time.*

Proof. Consider a submodular system $(W - \{s, t\}, g)$ where $g(S) = f(S \cup \{s\}) + f(W - S - \{s\})$ for all $S \subseteq W - \{s, t\}$. Clearly we need only to find a subset S' of $W - \{s, t\}$ that $g(S')$ is minimum by letting $S^* = S' \cup \{s\}$. Since g is submodular, it can be minimized in polynomial time [5,6,18]. $\square$

Hence we have seen that GSA can be extended to the TSPSS and runs in polynomial time. Furthermore, the performance can still be shown in a straightforward manner as Lemma 1 and Theorem 3. We summarize this as the next theorem.

Theorem 8. *Given a nonnegative submodular system (V, f) with a target set $T \subseteq V$, the TSPSS can be approximated within factor $(1+\alpha)(1-\frac{1}{k})$ in polynomial time, where α is any number that satisfies $\sum_{i=1}^k f(V - V_i) \leq \alpha \sum_{i=1}^k f(V_i)$ for all k -partitions $\{V_1, \dots, V_k\}$ of V that is a target split of T , where we can let $\alpha = k - 1$ in general, and let $\alpha = 1$ for symmetric f .* $\square$

Let us consider GSA2. Since the multiterminal cut problem is NP-hard even for $k = 3$, we cannot expect a polynomial time algorithm to compute a minimum 3-partition that is a target split in general (unless $P=NP$). Nevertheless, we note that Lemma 3, 4 can be extended to the TSPSS in a straightforward manner.

5 Conclusion and Remark

In this paper, we have presented a simple and unified approach for developing and analyzing approximation algorithms for some multiway partition related minimization problems. The main idea is a greedy splitting approach to unified problems k -PPSS (k -partition problem in submodular systems) and TSPSS (target split problem in submodular systems). Several important and interesting results are shown in this paper.

We note that it is still open whether the k -PPSS can be solved in polynomial time for any $k \geq 3$ (the 2-PPSS is shown to be solvable in polynomial time). Finally, we remark that it seems not so easy as in this paper to show the performance guarantee of greedy algorithms that increase the number of partitions three (or more) by three (or more). This is because analogous properties that we have shown in Lemma 1, 3 and 4 no longer hold even for k -PPG [20].

Acknowledgment

This research was partially supported by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan.

References

1. G. Călinescu, H. Karloff and Y. Rabani. An Improved Approximation Algorithm for Multiway Cut. In Proc. STOC 1998, pp. 48–52, 1998. 683
2. S. Chopra and J. H. Owen. A Note on Formulations for the A -Partition Problem on Hypergraphs. Discrete Applied Mathematics 90, pp. 115–133, 1999. 686
3. E. Dalhaus, D. S. Johnson, C. H. Papadimitriou, P. Seymour and M. Yannakakis. The Complexity of the Multiterminal Cuts. SIAM J. Comput. Vol. 23, No. 4, pp. 864–894, 1994. 683
4. O. Goldschmidt and D. S. Hochbaum. A Polynomial Algorithm for the k -cut Problem for fixed k . Math. Oper. Res. 19, No. 1, pp. 24–37, 1994. 682
5. M. Grötschel, L. Lovász and A. Schrijver. Geometric Algorithms and Combinatorial Optimization. Springer, Berlin, 1998. 692
6. S. Iwata, L. L. Fleischer and S. Fujishige. A Combinatorial, Strongly Polynomial-Time Algorithm for Minimizing Submodular Functions. In Proc. STOC 2000, pp. 97–106, 2000. 692
7. S. Kapoor. On Minimum 3-cuts and Approximating k -cuts Using Cut Trees. IPCO 1996, LNCS 1084, pp. 132–146, 1996. 683, 686
8. D. R. Karger, P. Klein, C. Stein, M. Thorup and N. E. Young. Rounding Algorithms for a Geometric Embedding of Minimum Multiway Cut. In Proc. STOC 1999, pp. 668–678, 1999. 683
9. D. R. Karger and C. Stein. A New Approach to the Minimum Cut Problems. J. of ACM 43, No. 4, pp. 601–640, 1996. 682
10. T. Lengauer. Combinatorial Algorithms for Integrated Circuit Layout. Wiley, New York, 1990. 686
11. D. W. Matula, A Linear Time $2 + \epsilon$ Approximation Algorithm for Edge Connectivity. In Proc. SODA 1993, pp. 500–504, 1993. 686
12. N. Maeda, H. Nagamochi, T. Ibaraki. Approximate Algorithms for Multiway Objective Point Split Problems of Graphs (Japanese). Computing Devices and Algorithms (Japanese) (Kyoto, 1993). Surikaiseikikenkyusho Kokyuroku No. 833, pp. 98–109, 1993. 683
13. H. Nagamochi and T. Ibaraki. A Fast Algorithm for Computing Minimum 3-way and 4-way Cuts. Mathematical Programming 88, No. 3, pp. 507–520, 2000. 682, 683
14. H. Nagamochi, S. Katayama and T. Ibaraki. A Faster Algorithm for Computing Minimum 5-way and 6-way Cuts in Graphs. J. Combinatorial Optimization 4, pp. 35–78, 2000. 682
15. H. Narayanan, S. Roy and S. Patkar. Approximation Algorithms for Min- k -Overlap Problems Using the Principal Lattice of Partitions Approach. J. Algorithms 21, pp. 306–330, 1996. 687
16. M. Queyranne. Minimizing Symmetric Submodular Functions. Mathematical Programming B 82, pp. 3–12, 1998. 684
17. M. Queyranne. On Optimum Size-Constrained Set Partitions. AUSSOIS 1999, France, March 8–12, 1999. 683, 686

18. A. Schrijver. A Combinatorial Algorithm Minimizing Submodular Functions in Strongly Polynomial Time. *J. Combin. Theory B* 80, No. 2, pp. 346–355, 2000. [692](#)
19. H. Saran and V. V. Vazirani. Finding k -cuts within Twice the Optimal. *SIAM J. Comput.* 24, No. 1, pp. 101–108, 1995. [683](#), [686](#)
20. L. Zhao, H. Nagamochi and T. Ibaraki. Approximating the Minimum k -way Cut in a Graph via Minimum 3-way Cuts. *ISAAC 1999, LNCS 1741*, pp. 373–382, 1999. Full paper will appear in *J. Combinatorial Optimization*. [683](#), [690](#), [691](#), [692](#)

On-Line Algorithms for Cardinality Constrained Bin Packing Problems

Luitpold Babel^{1*}, Bo Chen^{2**}, Hans Kellerer³, and Vladimir Kotov⁴

¹ Institut für Mathematik, Technische Universität München
D-80290 München, Germany, babel@statistik.tu-muenchen.de

² Warwick Business School, University of Warwick
Coventry CV4 7AL, United Kingdom
B.Chen@warwick.ac.uk

³ Institut für Statistik und Operations Research, Universität Graz
Universitätsstraße 15, A-8010 Graz, Austria
hans.kellerer@kfunigraz.ac.at

⁴ Faculty of Applied Mathematics and Computer Science, University of Minsk
Minsk, 220080, Byelaruscia
Kotov@fpm.bsu.unibel.by

Abstract. The bin packing problem asks for a packing of a list of items from $(0, 1]$ into the smallest possible number of bins having unit capacity. The k -item bin packing problem additionally imposes the constraint that at most k items are allowed in one bin. We present two efficient approximation algorithms for the on-line version of this problem. We show that, for increasing values of k , the asymptotic worst-case performance ratio of the first algorithm tends towards 2 and that the second algorithm has an asymptotic worst-case performance ratio of 2. Both heuristics considerably improve upon the best known result 2.7 of Krause, Shen and Schwetman. Moreover, we present algorithms for $k = 2$ and $k = 3$, where the result for $k = 2$ is best possible.

Keywords: bin packing, on-line, cardinality constraint

1 Introduction

In the classical *bin packing problem* (*BP*) we are given a list $L = (a_1, a_2, \dots, a_m)$ consisting of real numbers from $(0, 1]$, called *items*, and arbitrary many bins of unit capacity. The task is to find a packing of the items into as few bins as possible. In the *on-line* version (*oBP*) of this problem, the items have to be packed into the bins in the order they arrive. A new item a_n is packed solely on the basis of the sizes of the previous items $a_1, \dots, a_{n-1}$ and their packing. There is no information about subsequent items nor is it allowed to move or remove an already packed item.

^{*} Supported by the Deutsche Forschungsgemeinschaft

^{**} Supported by the ESRC Management Research Fellowship.

We study a variant of (*oBP*) in which an upper bound is additionally imposed on the number of items that can be packed together into one bin. The resulting problem is known as the *on-line k-item bin packing problem (okBP)*. It is derived directly from (*oBP*) by adding the constraint that at most k items can be packed into every bin.

The problem (*okBP*) first appeared in [3] in the context of task-scheduling on a multiprogramming computer system. In such a system there are k processors that share a common memory of fixed capacity. A sequence of tasks with unit processing times have to be executed on these processors. Each task has a certain memory requirement. The goal is to execute all tasks within the shortest possible time. We can represent the tasks by items and each unit of time by one bin. The memory requirements of the tasks correspond to the sizes of the items. All tasks that are performed in parallel correspond to items in the same bin. In particular, each bin has capacity 1 and contains not more than k items. The total execution time of the schedule equals the number of bins used in the packing.

Since bin packing is well known to be NP-complete, we are interested in the worst-case performance of approximation algorithms. For a given bin packing heuristic H and a list L of items, let $C^H(L)$ denote the number of bins used in a solution generated by H . Let $C^{OPT}(L)$ denote the smallest possible number of bins. (If L is clear from the context, then we omit it from the notation.) Then, the *asymptotic worst-case performance ratio* R_H of a heuristic H is defined by

$$R_H = \lim_{N \rightarrow \infty} \sup_L \left\{ \frac{C^H(L)}{C^{OPT}(L)} \mid C^{OPT}(L) = N \right\}.$$

The *d-dimensional vector packing problem* is a generalization of the one-dimensional bin packing problem. In such a problem, each item i is characterized by d numbers $a_i^1, a_i^2, \dots, a_i^d$. The task is to find a packing of items into as few bins as possible such that, for every j with $1 \leq j \leq d$, the sum of the numbers a_i^j of all items i in one bin is at most 1. The best known on-line heuristic for this problem is the generalization of First-Fit to the d -dimensional case, for which Garey et al. [2] prove that $R_{FF} = d + 0.7$. Clearly, (*okBP*) can be seen as a special case where $d = 2$ and the second number associated with each item is $1/k$. This provides an immediate bound $R_{FF} \leq 2.7$.

While bin-packing without cardinality constraints is well investigated, not much is known so far about the k -item bin packing problem. Krause et al. [3] investigated in 1975 an adaptation of (*FF*) to the cardinality constrained problem (hereinafter denoted by (*kFF*)), which packs a new item into the first possible bin that contains less than k items. They prove a bound $2.7 - \frac{12}{5k}$ for the asymptotic worst-case performance ratio of (*kFF*). As pointed out in [1], no improvement of this result has been obtained since 1975. While (*kFF*) behaves sufficiently well if k is small, it turns out that for large values of k the corresponding bounds are considerably worse than the bound obtained for First-Fit in the unconstrained case. For that reason we are particularly interested in algorithms that have a better worst-case performance ratio when k is not too small.

In our paper we study approaches more sophisticated than First-Fit, which enable us to improve the previous results. We present two efficient heuristics A_1

and A_2 and show that the asymptotic worst-case performance ratio of A_1 tends towards 2 as k goes to infinity and that the asymptotic worst-case performance ratio of A_2 is actually 2. Clearly this is an important progress from the bound 2.7 obtained for (kFF) . Moreover, we will present algorithms for $k = 2$ and $k = 3$, where the result for $k = 2$ is best possible.

2 Algorithm A_1

The main feature of our algorithm A_1 is that a new item can be packed into a bin only if the bin contains either *few items* or, if this is not the case, the bin together with this new item is *sufficiently filled*. In this way we want to avoid that bins with plenty of empty space contain too many items, a situation which could lead to a very bad solution.

Let $\ell(B)$ denote the *load* of bin B , i.e. the sum of the items in B , and $c(B)$ the number of items in B . We say that a bin B is *active* if $1 \leq c(B) \leq k - 1$. If $c(B) = k$ then B is called *full*. Furthermore, a bin B is termed *available* for item a_n , if $\ell(B) + a_n \leq 1$ holds.

In our algorithm A_1 we require that a bin containing k items has load at least $1/2$, a bin with $k - 1$ items has load at least $1/3$, a bin with $k - 2$ items has load at least $1/4$ etc. In this sense, for $1 \leq l < p$ where p is a not yet specified integer with

$$1 < p < k, \quad (1)$$

we say that a bin B is $(k - l)$ -*blocked* if $c(B) = k - l$ and $\ell(B) < 1/(l + 1)$. For convenience, we also briefly say that B is *blocked*. (Note that according to (1) for blocking at least two items are necessary.) On the other side, a bin B is called *unblocked* if either $c(B) \leq k - p$, or $c(B) = k - l$ and $\ell(B) \geq 1/(l + 1)$. Thus, an available bin is a candidate for a new item a_n if it is either unblocked, or it is blocked and fulfills the *threshold condition*

$$\frac{1}{l + 1} \leq \ell(B) + a_n. \quad (2)$$

The newly arriving items are packed into the bins according to the following strategy: First it is checked whether there is an available blocked bin whose load, along with the actual item, exceeds the associated threshold value $\frac{1}{l+1}$. If there is more than one such bin then we pack the item into a bin with largest load. If no such bin exists then it is checked whether there is an available unblocked bin that has been blocked in the past. If there is more than one such bin then the algorithm again chooses a bin with largest load. If also no such bin exists then it is checked whether, among the remaining active bins, there is an available one. In case of a tie again a bin with largest load is selected. Finally, if no active bin could be found that fits for the actual item, then the algorithm opens a new bin for it.

Roughly speaking, A_1 follows a Best-Fit strategy which is first applied to all blocked bins which satisfy the threshold condition (2), then to all formerly blocked and now unblocked bins, and finally to the remaining bins.

A more formal description of algorithm A_1 is given below:

ALGORITHM A_1

While the list L of items is nonempty **do**

Remove the next item a_n from L

Let $\mathcal{B}_1 := \{B \mid \exists l \in \{1, \dots, p-1\} : c(B) = k-l \text{ and } \ell(B) < \frac{1}{l+1} \leq \ell(B) + a_n \leq 1\}$

If $\mathcal{B}_1 \neq \emptyset$ **then choose** $B \in \mathcal{B}_1$ with $\ell(B)$ maximal **else**

Let $\mathcal{B} := \{B \mid (1 \leq c(B) \leq k-p) \text{ or } \exists l \in \{1, \dots, p-1\} :$

$(c(B) = k-l \text{ and } \frac{1}{l+1} \leq \ell(B) \leq 1-a_n)\}$

and $\mathcal{B}_2 := \mathcal{B} \cap \{B \mid B \text{ marked}\}, \mathcal{B}_3 := \mathcal{B} - \mathcal{B}_2$

If $\mathcal{B}_2 \neq \emptyset$ **then choose** $B \in \mathcal{B}_2$ with $\ell(B)$ maximal **else**

If $\mathcal{B}_3 \neq \emptyset$ **then choose** $B \in \mathcal{B}_3$ with $\ell(B)$ maximal **else**

Choose a bin B with $c(B) = 0$

Pack item a_n into the selected bin B

If $\exists l \in \{1, \dots, p-1\} : (c(B) = k-l \text{ and } \ell(B) < \frac{1}{l+1})$ **then mark** B .

Algorithm A_1 contains a parameter, namely p , which determines the smallest number of items in a bin to become blocked. We first analyze the worst-case performance ratio of algorithm A_1 depending on p . Then we choose the most suitable value for p .

Note that, during the execution of the algorithm, a bin can repeatedly become blocked and unblocked. We first point out that in every stage of the algorithm all bins with small load, except at most one, are blocked or have been blocked in the past. More precisely, we claim:

Proposition 1 *In each stage of the algorithm A_1 there are no two active bins with load smaller than $\frac{1}{2}$ which never have been blocked.* $\square$

If after the execution of algorithm A_1 there is no bin with load smaller than $1/2$ which is blocked or has been blocked in the past then, by the previous fact, all bins except at most one have load at least $1/2$. This immediately implies that

$$C^{A_1} \leq 2C^{OPT} + 1. \quad (3)$$

In the following we can assume that there exists at least one bin of small load that is blocked or has been blocked.

Among all the bins which ever become blocked, let B^* denote the one which becomes marked at last (i.e. when B^* becomes blocked for the first time, all

other bins have already been blocked). Let further a^* be the smallest item which is packed into B^* until B^* is blocked for the first time.

We consider first the structure of the packing **at time t_1 when item a^* appears**. Till then we distinguish between three types of bins. The bins of *type 1* are the active bins with load smaller than $1/2$ at time t_1 . By the previous proposition, all these bins except at most one are blocked or have been blocked. All active bins which are not of type 1 have load at least $1/2$. The bins of *type 2a* are the full bins at time t_1 . We can state:

Observation 2 *All bins B of type 2a fulfill $c(B) = k$ and $\ell(B) \geq \frac{1}{2}$.* $\square$

The remaining active bins at time t_1 are of *type 3*. Clearly, all these bins are unblocked. We can show:

Proposition 3 *All bins B of type 3 fulfill $\ell(B) > 1 - \frac{1}{2(k-1)}$.* $\square$

Let us now study the structure of the solution **after having packed the last item**. The items which are treated after a^* are packed into bins of type 1 or of type 3, or into bins which are opened after the appearance of a^* (note that bins of type 1 can now also have load at least $1/2$ and can also be full). By Proposition 1, all bins of type 1 except one extra-bin are blocked or have been blocked. Let $\ell_{\min}$ denote the smallest load of all bins of type 1 (except the extra-bin). For each of these bins consider further its load when it has been blocked for the last time. We denote with $\ell_{\min}^*$ the smallest of all these loads. Since we assumed that there is at least one bin with load smaller than $1/2$, we have $\ell_{\min}^* \leq \ell_{\min} < 1/2$. These definitions immediately imply:

Observation 4 *All bins B of type 1 - except at most one - fulfill $\ell(B) \geq \ell_{\min}$. If $\frac{1}{l+1} \leq \ell_{\min}^* < \frac{1}{l}$ with $1 < l \leq p$ then $c(B) \geq k - l + 1$. If $\ell_{\min}^* < \frac{1}{p}$ then $c(B) \geq k - p + 1$.* $\square$

For bins of type 2a and 3 clearly Observation 2 and Proposition 3 are still valid. It is easy to see that bins which are opened after the appearance of a^* can never become blocked: If such a bin becomes blocked before B^* then, since no blocked bins with a single item exist, in contradiction to Proposition 1, at some stage there are two active bins with load smaller than $1/2$ which never have been blocked in the past. On the other side, due to the special choice of B^* , such a bin cannot become blocked after B^* .

What can we say about the items which appear after a^* and which are packed into these new bins? First, since such an item is not packed into a blocked bin of type 1, it might be too small in order to exceed the associated threshold value for the bin. If $1/(l+1) \leq \ell_{\min}^* < 1/l$ then it must be smaller than $1/l - 1/(l+1) = 1/(l(l+1))$, if $\ell_{\min}^* < 1/p$ then it must be smaller than $1/p$. Second, the item might be too large in order to fit into a blocked bin or into an unblocked (and formerly blocked) bin of type 1. This means that it must be larger than $1 - \ell_{\min}$.

The new bins which contain an item larger than $1 - \ell_{\min}$ are said to be of *type 4*.

Observation 5 *All bins B of type 4 fulfill*

$$\ell(B) > 1 - \ell_{\min}.$$

□

The remaining bins contain only items smaller than $1/(l(l+1))$ resp. $1/p$. If they are active, they are of *type 5*. Otherwise, they are of *type 2b*. Of course, Observation 2 holds also for bins of type 2b. Bins which are either of type 2a or of type 2b are of *type 2*.

Lemma 6 *All bins B of type 5 - except at most one - fulfill*

$$c(B) \geq l^2 + l \quad \text{resp.} \quad c(B) \geq p.$$

With the exception of at most two bins we furthermore have

$$\ell(B) \geq \frac{l^2 + l}{l^2 + l + 1} \quad \text{resp.} \quad \ell(B) \geq \frac{p}{p + 1}.$$

□

Theorem 7

$$C^{A_1} \leq \left(2 + \frac{k + p(p-3)}{(k-p+1)p}\right) C^{OPT} + 2.$$

□

Let $p = k^\alpha$ with $0 < \alpha < 1$ fixed. As k goes to infinity, we obtain that R_{A_1} tends towards 2. If we replace p by the nearest integer $p^* = \text{round}\left(\frac{-k + \sqrt{k^3 - 2k}}{k-2}\right)$ then the above bound becomes as small as possible.

3 Algorithm A_2

For the construction of the algorithm A_2 we distinguish between *closed* and *open* bins. Closed bins are bins with $\ell(B) \geq \frac{1}{2}$ and $c(B) \geq \frac{k}{2}$ or pairs of bins B_1, B_2 with $\ell(B_1) + \ell(B_2) \geq 1$ and $c(B_1) + c(B_2) \geq k$. All the other bins are called open. The open bins are partitioned into three different types:

1. Bins of type 1: These are bins with $\ell(B) \geq \frac{1}{2}$ and $c(B) < \frac{k}{2}$.
2. Bins of type 2: These are bins with $\ell(B) < \frac{1}{2}$ and $c(B) < k - 1$.
3. Bins of type 3: These are bins with $\ell(B) < \frac{1}{2}$ and $c(B) = k - 1$.

The current number of bins of type i ($i = 1, 2, 3$) is denoted by c_i .

Our algorithm A_2 works as follows: First we try to pack an incoming item a into a bin of type 1. Then we try to put a into a bin of type 3 if the total load with a would be greater than $\frac{1}{2}$ or if there exists a bin of type 1. Finally try to put a into bins of type 2. If it fits in none of these bins, a new bin is opened.

Lemma 8 *In each stage of algorithm A_2 the following two properties hold:*

$$c_2 \leq 1 \quad (4)$$

and

$$c_3 = 0 \text{ or } c_1 + c_2 \leq 1. \quad (5)$$

Proof. The proof will be done by induction on the number of items assigned. Assume that before item a is taken from the list both (4) and (5) hold. We distinguish several cases:

a) Item a is packed into a bin of type 1:

By adding item a a bin of type 1 either becomes closed or remains of type 1. Thus, the number of bins of type 1 does not increase and (4) and (5) hold.

b) Item a is packed into bin of type 3:

Let B be a bin of type 3 in which a fits. By adding a either B becomes closed or B forms together with an arbitrary bin $\bar{B}$ of type 1 a pair of closed bins since a did not fit in $\bar{B}$.

c) Item a is packed into a bin of type 2:

After packing item a there will be no increase of c_2 . Thus, (4) holds. Now let B be the unique bin of type 2. If B becomes closed or remains of type 2, the numbers c_1 , c_2 , c_3 are still the same. If B turns into a bin of type 1, then $c_1 + c_2$ and c_3 remain unchanged and (5) holds. Finally, assume that B turns into a bin of type 3. If $c_1 = 0$, then from (4) we know that $c_1 + c_2 \leq 1$ still holds. If $c_1 > 0$, there was at least one bin of type 1, denoted by $\bar{B}$, in which item a did not fit. But then $\ell(\bar{B}) + \ell(B) > 1$. Consequently, B and $\bar{B}$ form a pair of closed bins.

d) A new bin is opened by item a :

If there is no bin of type 3 before adding a , then (5) holds. In the case that there is no bin of type 2 also (4) is valid. Otherwise, a did not fit into the bin of type 2 and has size greater than $\frac{1}{2}$. Therefore, the new bin is of type 1 and (4) is valid again.

If there is a bin of type 3, in which item a does not fit, this bin and a form a pair of closed bins. Thus, assume a fits in all bins of type 3. Since a new bin is opened by item a , item a has size smaller than $\frac{1}{2}$ and no bins of type 1 exist. Consequently, also no bin of type 2 could exist before adding a . We get $c_1 + c_2 \leq 1$ after packing a and both (4) and (5) hold. $\square$

Theorem 9 *Algorithm A_2 has asymptotic worst-case ratio 2.*

Proof. The number of items divided by k or the total sum of the items are two obvious lower bounds for the number of bins in an optimal solution. Lemma 8 guarantees that A_2 packs either more than $\frac{k}{2}(C^{A_2} - 1)$ items or has total load greater than $\frac{1}{2}(C^{A_2} - 1)$. The claim follows. $\square$

4 A Best Possible Algorithm for (o2BP)

This section contains the algorithm A for (o2BP). Before we start the description of the algorithm, some notations are introduced. An item is said to be *small* if its size is no more than $\frac{1}{2}$. Otherwise, it is called *big*. Consider a packing configuration right after item a_i is packed. A non-empty bin is said to be of *type* X^i if it contains exactly one small item, of *type* Y^i if it contains a big item and a small item, of *type* Z^i if it contains two small items, and of *type* U^i if it contains exactly one big item. As we will see, packing a big item is relatively simple. Therefore, bins of small items, i.e., of types X , Y and Z , will be our main concern. Let the numbers of bins of these types be x^i , y^i , z^i and u^i , respectively. If there is no confusion, we will omit the superscript i from the above notation. Denote $\rho = \sqrt{2} - 1$.

4.1 Algorithm A and Its Analysis

At any stage of packing, the algorithm tries to make the following condition satisfied if all possible:

$$\lfloor \rho z \rfloor \leq x + y \leq \lfloor \rho z \rfloor + 2. \quad (6)$$

In the following description of algorithm A , an item is said to be *A-packed* (*packed*, respectively) into a set S of bins, if it is packed into one of these bins according to the rule of *Best Fit* when $|S| \geq 2$ (when $S \neq \emptyset$, respectively), and a new bin is opened if it fits in none of these bins or $|S| \leq 1$ ($S = \emptyset$, respectively).

Algorithm A. Consider how algorithm A packs an item a_{i+1} .

- Step 1.** If the item is the first small item, then pack it into a new bin.
- Step 2.** If the item is big, then A -pack the item into a bin of type X^i .
- Step 3.** If the item is small and if $x^i + y^i < \lfloor \rho z^i \rfloor + 2$, then pack the item into a bin of type U^i .
- Step 4.** If the item is small and if $x^i + y^i \geq \lfloor \rho z^i \rfloor + 2$, then pack the item into a bin of type X^i if $x^i \geq 2$ and into a bin of type U^i otherwise.

The following two lemmas are evident.

Lemma 10 For all i , $x^i \geq 1$. The first small item in any bin of type Z^i is at least as big as the smallest item among the x^i items in bins of type X^i . Similarly, the first item a_k in any bin of type Y^i is at least as big as the smallest item among the x^i items in bins of type X^i if a_k is small, and of type U^i if a_k is big.

Lemma 11 Violation of condition (6) is necessarily equivalent to $x + y > \lfloor \rho z \rfloor + 2$. Packing at Step 3 will never result in such situation.

Lemma 12 If $x \geq 3$, then condition (6) is satisfied.

Proof. Let $a_{i_1}, \dots, a_{i_x}$ be the small items contained in the bins of type X , where $i_1 < \dots < i_x$. If condition (6) is not satisfied, then a_{i_x} is packed at Step 4 according to Lemma 11. However, according to the algorithm, item a_{i_x} would have been packed into a bin of type X^{i_x-1} , such as those of items a_{i_1} and a_{i_2} . This is a contradiction.

Theorem 13 *Algorithm A satisfies $C^H \leq (1 + \rho)C^{OPT} + 3$.*

Proof. Since half of the total number of packed items is a lower bound on C^{OPT} , we first obtain

$$C^{OPT} \geq \frac{u+x}{2} + y + z. \quad (7)$$

Let us consider two cases. Case 1: $x \geq 3$. According to the algorithm, each of the items, except possibly one, in bins of type X will not fit into a single bin with the item in any bin of type U . According to Lemma 10, there are at least $x + z - 1$ small items, each of which will not fit into a single bin with any big items in the bins of type U . Taking into account of the y big items in bins of type Y , we obtain

$$C^{OPT} \geq u + \frac{x+y+z}{2} = (u+x) + \frac{y+z}{2} - \frac{1+x}{2},$$

which, together with the fact that $x+y \leq \rho z + 2$ according to Lemma 12, implies that

$$\begin{aligned} C^{OPT} &\geq (u+x) + \frac{y+z}{2} - \frac{1+x}{2} \geq (u+x) + \frac{y+z}{2} + \frac{y-\rho z}{2} - \frac{3}{2} \\ &\geq (u+x) + \frac{1-\rho}{2}(y+z) - 2. \end{aligned} \quad (8)$$

Inequality (8) plus $(1 + \rho)$ times (7) gives

$$(2 + \rho)C^{OPT} \geq \frac{3+\rho}{2}(u+x+y+z) - 3 = \frac{3+\rho}{2}C^A - 2,$$

which is equivalent to $C^A \leq (1 + \rho)C^{OPT} + 2(1 - \rho)$.

Case 2: $x \leq 2$. Since $x + y \geq \lfloor \rho z \rfloor > \rho z - 1$ according to condition (6) and Lemma 11, from (7) we obtain

$$C^{OPT} \geq \frac{u+x+y}{2} + \frac{1}{2}y + z \geq \frac{u+x+y}{2} + \frac{2+\rho}{2}z - \frac{x+1}{2}. \quad (9)$$

On the other hand, since all the $u + y$ big items have to be packed separated in any packing, we obtain

$$C^{OPT} \geq u + y = (u + x + y) - x. \quad (10)$$

Sum of 2 times (9) and $(1 + \rho)$ times (10) gives

$$(3 + \rho)C^{OPT} \geq (2 + \rho)(u + x + y + z) - (2 + \rho)x - 1 \geq (2 + \rho)C^A - (5 + 2\rho),$$

which is equivalent to $C^A \leq (1 + \rho)C^{OPT} + (2 + \rho)$.

4.2 A Matching Lower Bound

Given any on-line algorithm H , according to the definition, there exists a constant $N_H \geq 1$, such that for any item list L ,

$$C^H(L) \leq R_H C^{OPT}(L), \quad (11)$$

where $C^{OPT}(L) \geq N_H$.

Theorem 14 *Any on-line algorithm for (o2BP) has an asymptotic competitive ratio at least $\sqrt{2}$.*

Proof. Consider any on-line algorithm H . Let $N = 2N_H$, where N_H is chosen as above. Let us release *one-by-one* a list of k_1 ($1 \leq k_1 \leq N$) small jobs of sizes

$$a_{1,k} = \frac{1}{2} - \frac{k}{4N}, \quad k = 1, \dots, k_1,$$

where $k_1 = \max \{k: 1 \leq k \leq N \text{ and } z^k = 0\}$. It is evident that k_1 is well defined. In general, suppose, for some $s \geq 1$, we have released one-by-one $n_s = \sum_{i=1}^s k_i$ small jobs of sizes $a_{i,k}$ ($k = 1, \dots, k_i$, $i = 1, \dots, s$), which are defined iteratively as follows: for $i = 0, \dots, s-1$,

$$a_{i+1,k} = a_{i,k_i-1} - \frac{k}{4N^{i+1}}, \quad k = 1, \dots, k_{i+1}, \quad (12)$$

where

$$k_{i+1} = n_i + \max \{k: 1 \leq k \leq N - n_i \text{ and } z^{n_i+k-1} = z^{n_i}\}, \quad (13)$$

and $k_0 = 1$, $a_{0,0} = 1/2$. If $n_s = N$, we stop. Otherwise, we further release one-by-one k_{s+1} small jobs of sizes $a_{s+1,k}$, $k = 1, \dots, k_{s+1}$, which are defined as in (12) and (13) by setting $i = s$. Therefore, with the inductive definition, we assume without loss of generality that $n_s = N$. By construction, we observe that the list L_0 of the N small items has the following two properties: (a) The sizes of all items satisfy $1/4 \leq a_{i,k} \leq 1/2 - 1/(4N)$ and (b) the sizes of second items in all bins of type Z satisfy $a_{1,k_1} < a_{2,k_2} < \dots < a_{s,k_s}$ and they are all strictly smaller than any other $N - s$ items. Now we are ready to give a contradiction. If we further release a list L' of s big jobs, each of size $1 - a_{s,k_s} > 1/2$, then any optimal packing of the concatenated list $L_0 L'$ has $C^{OPT}(L_0 L') = s + (N - s)/2 = (N + s)/2$, while $C^H = N$. Hence we have

$$\frac{C^H(L_0 L')}{C^{OPT}(L_0 L')} = \frac{2N}{N + s}. \quad (14)$$

On the other hand, instead of L' , if we further release a list L'' of N big jobs, each of size $1/2 + 1/(4N)$, then any optimal packing has $C^{OPT} = N$, while $C^H = N + s$. Hence we also have

$$\frac{C^H(L_0 L'')}{C^{OPT}(L_0 L'')} = \frac{N + s}{N}. \quad (15)$$

Combining (14) and (15) and taking (11) into account, we obtain

$$R_H \geq \max \left\{ \frac{2N}{N+s}, \frac{N+s}{N} \right\}.$$

The right-hand side of above is minimized to $\sqrt{2}$ when $s = (\sqrt{2} - 1)N$, which proves our theorem.

5 A Simple Heuristic for (o3BP)

The harmonic approach was introduced by Lee and Lee [5] in 1985 for constructing efficient algorithms for on-line bin packing without cardinality constraints problem. For $k = 3$ we apply a simple harmonic-type algorithm.

Our Heuristic H for (o3BP) works as follows: The items are partitioned into three intervals $A =]0, \frac{1}{3}]$, $B =]\frac{1}{3}, \frac{1}{2}]$ and $C =]\frac{1}{2}, 1]$. Heuristic H always keeps three *active* bins B_A , B_B , and B_C . Items of interval A (respectively B and C) are packed by Next-Fit into bin B_A (respectively bin B_B and bin B_C). As soon as an active bin B_A (respectively bin B_B and bin B_C) has received 3 (respectively 2 and 1) items, it is *closed* and a new bin becomes active.

We show that H has asymptotic worst-case ratio $\frac{5}{3}$ by using the weighting function technique. Assign to every A -item x a weight $w(x) = \frac{1}{3}$, to every B -item x a weight $w(x) = \frac{1}{2}$, and to every C -item x a weight $w(x) = 1$. The weight of an item set shall be equal to the sum of the weights of the items in this set. Now consider a list L of items. By definition, H assigns a weight of 1 to every closed bin. There are at most two non-empty open bins with minimum total weight at least $\frac{5}{6}$. Hence,

$$w(L) \geq C^H(L) - 2 + \frac{5}{6} \geq C^H(L) - \frac{7}{6}. \quad (16)$$

It can be easily seen that the maximum weight of a bin is $\frac{5}{3}$ (one A -item, one B -item and one C -item). Hence,

$$w(L) \leq \frac{5}{3} C^{OPT}(L). \quad (17)$$

Combining the inequalities (16) and (17) yields the desired asymptotic competitive ratio of $\frac{5}{3}$.

A lower bound of $\frac{3}{2}$ for (o3BP) can easily be established by using three lists with n items of sizes ε , $\frac{1}{3} + \varepsilon$ and $\frac{1}{2} + \varepsilon$, respectively as introduced by Yao [7] for on-line bin packing without cardinality constraints. (For a detailed description of the proof for the lower bound we refer to [7].) Our results are summarized in the following theorem.

Theorem 15 *There is an on-line algorithm for (o3BP) with asymptotic competitive ratio $\frac{5}{3}$. Any on-line algorithm for (o3BP) has an asymptotic competitive ratio at least $\frac{3}{2}$.*

6 Conclusions

We have studied a variant of the one-dimensional bin packing problem where each bin may contain not more than k items. We introduced two on-line algorithms with much better competitive ratio than the previously best known bound 2.7 obtained by Krause, Shen and Schwetman. While algorithm A_2 shows a slightly better worst-case performance than A_1 , it performs much worse in the average case, even worse than the adapted version of (FF) by Krause et al., as it has been confirmed in extensive numerical testings. Due to the construction of A_2 the number of bins used by A_2 tends towards double the maximum of the total sum of the items and the total number of the items divided by k .

It is a challenging open problem to find out whether there are on-line algorithms with competitive ratio strictly better than 2 (i.e. $\leq 2 - \varepsilon$ independently from k). A further object of research is to search for reasonable lower bounds on the worst-case ratio of any algorithm for the on-line k -item bin packing problem. It is obvious that the lower bounds for the unconstrained case by Van Vliet [6] also hold if k tends to infinity. Thus, it is interesting whether bounds can be found which are stronger than the lower bounds by Van Vliet.

References

1. Galambos G., Woeginger G. J.: On-line bin packing - a restricted survey. *ZOR - Mathematical Methods of Operations Research* **42** (1995) 25–45 [696](#)
2. Garey M. R., Graham R. L., Johnson D. S., Yao A. C. C.: Resource constrained scheduling as generalized bin packing. *J. Combinatorial Theory A* **21** (1976) 257–298 [696](#)
3. Krause K. L., Shen V. Y., Schwetman H. D.: Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *Journal of the ACM* **22** (1975) 522–550 [696](#)
4. Krause K. L., Shen V. Y., Schwetman H. D.: Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems, Errata. *Journal of the ACM* **24** (1977) 527
5. Lee C. C., Lee D. T.: A simple on-line bin packing algorithm. *Journal of the ACM* **32** (1985) 562–572 [705](#)
6. Van Vliet A.: An improved lower bound for on-line bin packing algorithms. *Information Processing Letters* **43** (1992) 277–284 [706](#)
7. Yao A. C. C.: New algorithms for bin packing. *Journal of the ACM* **27** (1980) 207–227 [705](#)

Suffix Vector: A Space-Efficient Suffix Tree Representation

Krisztián Monostori¹, Arkady Zaslavsky¹, and István Vajk²

¹ School of Computer Science and Software Engineering, Monash University,
Melbourne, 900 Dandendong Rd, Caulfield East VIC3145, Australia
{Krisztian.Monostori, Arkady.Zaslavsky}@csse.monash.edu.au

² Department of Automation and Applied Informatics,
Budapest University of Technology and Economics
1111 Budapest, Goldmann György tér 3. IV.em.433., Hungary
vajk@aut.bme.hu

Abstract. This paper introduces a new way of representing suffix trees. The basic idea behind the representation is that we are storing the nodes of the tree along with the string itself, thus edge labels can directly be read from the string. The new representation occupies less space than the best-known representation to date in case of English text and program files, though it requires slightly more space in case of DNA sequences. We also believe that our representation is clearer and thus implementing algorithms on it is easier. We also show that our representation is not only better in terms of space but it is also faster to retrieve information from the tree. We theoretically compare the running time of the matching statistics algorithm on both representations.

1 Introduction

Suffix tree is a data structure that stores all possible suffixes of a string. It is one of the most versatile data structures in the area of string matching. Apostolico [2] lists over 40 applications of suffix trees and Gusfield [5] has a list of more than 20 applications.

Our main research area is identifying overlapping documents on the Internet [10] and we use a suffix tree structure to find the exact matching chunks among documents. Our application also shows that the applications of suffix trees are not limited to DNA-matching and basic string-matching problems but they can also be applied in other areas.

If we stored the suffix tree in a naive way it would occupy $O(n^2)$ space because the overall length of suffixes is $n*(n-1)/2$. Instead of storing the actual characters of an edge in the tree we store a start pointer and an end pointer. Since the number of edges and nodes are proportional to n , the overall space requirement of a suffix tree is $O(n)$. There are three basic algorithms to build a suffix tree: McCreight's [9], Weiner's [12], and Ukkonen's [11]. For a unifying view of these algorithms, see Giegerich and Kurtz's paper [4].

One of the main arguments against suffix trees is the space requirement of the structure. There has been quite a work done in this area, though most of the implementations are tailored to a specific problem. The original implementation, as it was suggested by McCreight [9], occupies 28 bytes for each input character in the worst case. Most of the implementations that aimed to improve the space-efficiency are not as versatile as the original suffix tree. None of the following alternative structures keep suffix link information in the tree, which is heavily utilised in the matching statistics [3] algorithm that we use to compare texts:

- suffix arrays [8] occupy 9 bytes for each input symbol
- level compressed tries [1] take 12 bytes for each input character
- suffix cactuses [6] require $10n$ bytes

Kurtz [7] proposes a data structure that requires 10.1 bytes per input character on average for a collection of 42 files. This data structure retains all features of the suffix tree including suffix links. Kurtz compares his representation to many other competing representations and finds that his implementation is the most space-efficient for the collection of 42 files he used. Later in this paper we will have a more detailed comparison of our representation to Kurtz's.

In the next section we introduce suffix vectors: we show the basic idea abstracted from the actual implementation. In Section 3 we show how the new representation can be stored efficiently. Section 4 compares our representation to Kurtz's representation in terms of space and usability. Section 5 gives a high-level description of how the suffix vector can be constructed from a suffix tree. In Section 6 we summarize our results and look at future work.

2 Suffix Vector

Suffix vectors are proposed in this paper as an alternative representation of suffix trees and directed acyclic graphs (DAG) derived from suffix trees. The basic idea of suffix vectors is based on the observation that we waste too much space on edge indices, so we store node information aligned with the original string. Hence, edge labels can be read directly from the string. This section describes the proposed alternative structure and in later sections we analyse worst-case space requirements.

First we give a sample string and the suffix tree representation for that string. Let the string be $S = \text{'abcdabdbcdabb\$'}$. '\$' is the unique termination symbol, which is needed, otherwise the suffix tree cannot be constructed. The suffix tree for that string is depicted in *figure 1*.

Firstly we introduce a high-level suffix vector representation that is abstracted from the actual storage method. We show how the traversal of the tree works using this representation and later we show how we can efficiently represent this structure in memory. As we have already mentioned, the basic idea of our new representation is based on storing nodes and edges aligned with the string. *Figure 2* shows the new representation.

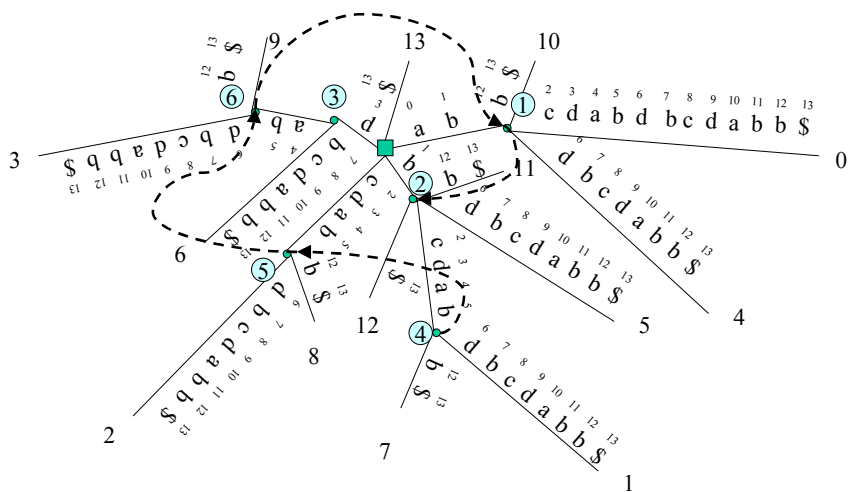


Fig. 1. Suffix Tree of 'abcdabdbcdabb\$'

The root node is represented as a linked list and it shows where to start searching for a string. It has one pointer for each possible character in the string (a,b,c,d,\$). Nodes in the original tree are represented as linked lists in the vector aligned with the string. For example node 3 in the original tree is represented by the box between position 3 and 4. Each node has a “natural edge”, that is the continuation of the string, so the edge pointing from node 3 to node 6 is character 4 and 5 in the string. The first number in bold is the depth of the node, so in case of node 3, ‘7 x’ means that after matching one character (position 3 ‘d’) we can either follow the string itself (this is the edge pointing from node 3 to node 6), or we can jump to position 7 (this is leaf 6). The ‘x’ means that if we jump to position 7 there are no more nodes, that is a leaf. The second number in bold (5) says that if we reached this position after matching one character (‘d’) and we would follow matching ‘a’ (the “natural edge”), the next node is at position 5 (between 5 and 6). In the original tree the next node is 6, which is depicted by the third row of the box between 5 and 6. We need to be able to jump from one node to the next one for some algorithms. There are situations, which do not require this information. For example if we only need to find one occurrence of a pattern in the string we can find it without this information.

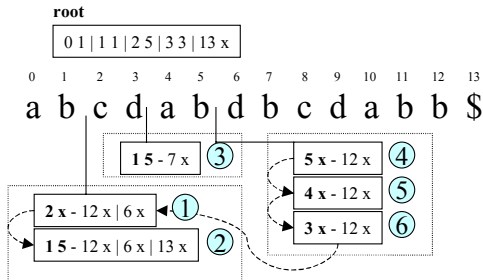


Fig. 2. Suffix Vector

As one can see, each node has one corresponding row in one of the boxes. Node 1 is the first row in the box at position 1, node 2 is the second row in the box at position 1, node 3 (as discussed earlier) is the only row in the box at position 3, node 4 is the first row in the box at position 5, node 5 is the second row in the box at position 5, node 6 is the third row in the box at position 5. Every node is stored at the smallest possible index that is at the first occurrence of the string running into that node.

To see how the algorithm finds a string, let us follow the matching of ‘dabb’ in the string. We start from the root and find that we have to start at position 3. It is equivalent to analysing the edges running out of the root in the tree. After having matched ‘d’, we try to match ‘a’. In the tree we have to check whether there is an edge starting with ‘a’ running out of node 3, in the suffix vector we match the next character. In this case it matches ‘a’ but if it did not match we should check the possible followings after having matched one character. We find this information in the first row of the box at position 3. After ‘a’ we have to match ‘b’ on the edge in the tree and in the string in the suffix vector. They match, so we have to match the second ‘b’. They do not match. We have followed 3 characters up to now, so we have to check the possible followings from here. We can see that having matched 3 characters we could follow at position 12, that is leaf 9 in the tree. The ‘x’ depicts that this is a leaf node, so that is the only possible match. We have matched 4 characters up to position 12, so the start position is 9.

3 Space-Efficient Representation of a Suffix Vector

A naive representation of the suffix vector would store a pointer at each position in the string. These pointers may or may not be filled in. Each pointer would point to a box structure that may store multiple nodes at each position. We store the depth of the deepest node and the number of nodes in each box. We can have a pointer each to an array of next node pointers, suffix links, and first edges. Knowing the actual depth of the node we can calculate its position in the array from the deepest node value. The deepest node is stored at position 0. The edges may be represented as linked lists. Obviously this storage method is far from ideal. In the following subsection we propose a more space-efficient representation. In Section 3.2 we introduce the concept of large nodes and reduced nodes, which allow further space reduction.

3.1 Suffix Vector Physical Representation

We start with a few observations and then present the implementation details that utilise those observations.

Observation 1. The node depth of a deepest node can usually fit into 7 bits.

Large node depth values represent long repetitions in the string. These are very rare in English texts and also very unlikely in random texts. The representation does not limit the node depth but it stores it in one byte whenever it is possible. We use 7 bits because in the actual implementation the first bit is used for some other purposes.

Observation 2. The number of nodes at a given position (in a given box) can usually fit into 7 bits.

This observation is a direct consequence of Observation 1 because we cannot have more nodes in a given box than the depth of the deepest node since nodes with the same node depths are stored at different positions.

Observation 3. The length of an edge can usually fit into 1 byte.

This observation follows the reasoning of the previous ones. Long edges mean long overlaps in the text and you cannot have many long overlaps. If you have many long overlaps it means that you have a long text, so the ratio of long edges is still small.

Theorem 1. The suffix link of a node always points to another node at the same position except for the last node (the node with the smallest node depth).

Let the label of *node v* be *aw* where *a* is a single character and *w* is a sequence of characters. If there is a suffix link between *node v* and *node z* then *node z* has a label *w*. Let *x* be the depth of *node v*. Then *node v* is listed at the given position (let this position be *i*). If *node v* is not the node with the smallest node depth then there is a *node y*, which has a node depth *x-1* and its label is *w* by definition. The suffix link of *node v* points to a node with label *w* and there is only one such node in the suffix tree because node labels are unique. Therefore, *node v* must point to *node y*, thus the equality relation *node z* $\equiv$ *node y* follows .

Figure 3 depicts the representation that utilises the observations and the theorem above.

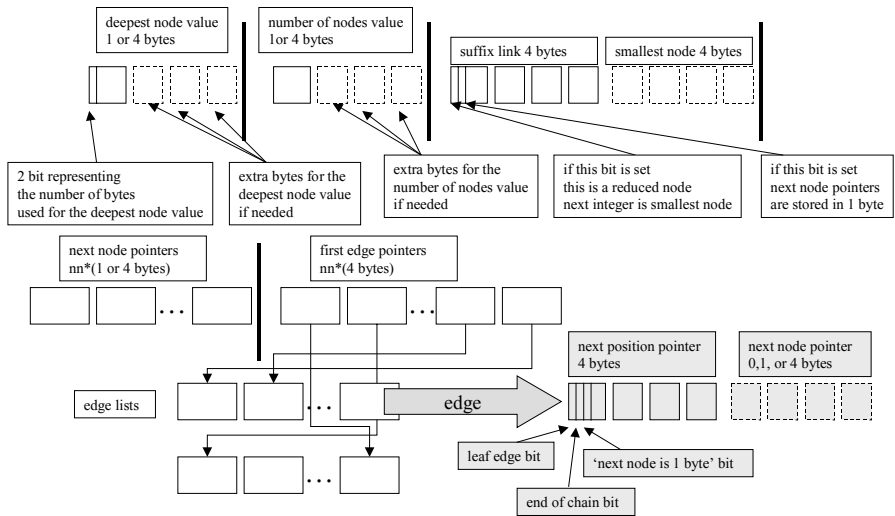


Fig. 3. Space-efficient storage of a suffix vector

In each box we have to store three pieces of information that characterize the box rather than individual nodes, so this information must be stored once per box. The first value that we store is the deepest node value representing the depth of the deepest node stored at this position. From Observation 1 we know that the depth of the deepest

node is usually very small, so storing it constantly in 4 bytes is a waste of storage space. We use the first bit to denote the number of bytes we need to store the deepest node value (1 or 4 bytes). Let us denote this value by l . The best case for us is when the depth is under 128 because then it fits into the first byte (note that the first bit is used to flag the length of the field). It is very rare that chunks greater than 128 characters are repeated in any text. The number of nodes value uses the same number of bytes (l) based on Observation 2. It is possible that the number of nodes value is less than the deepest node value, thus it fits into one byte when the deepest node value does not fit into one byte but using another bit to flag this situation would unnecessarily complicate retrieval of data and would only save space rarely.

The next piece of information stored in the box is the suffix link value. From Theorem 1 it follows that every box needs to store at most one suffix link value. If the number of nodes value equals to the deepest node value it means that the depth of shortest node is one character. One-character-deep nodes do not need to store a suffix link. In this case it is not necessary to store a suffix link for the shortest node but we store one anyway because we use the first bit of the suffix link to flag whether this is a reduced node and the second bit to flag whether we have small next node pointers (1 byte) or large next node pointers (4 bytes). Reduced nodes are discussed later. For reduced nodes we need one more integer beside the suffix link, which tells us the depth of the shortest node. If the first bit is not set the suffix link is stored in one integer (the offset of the start position of the suffix link is $2 * l$). If the second bit is set it means that all next node pointers can be stored in 1 byte, so the following pointers for next nodes occupy one byte. Let s be 1 if this is a reduced node, so we know that an extra integer is used for the smallest node and let n denote the length of the fields used for storing next node pointers (1 or 4 bytes).

The next thing that we are storing is the next node pointers. Next node pointers point to the node following from this position. Note that depending on the depth of the node stored at this position the next node value may vary, thus we have to store a next node pointer for each node represented at this position. From Observation 3 we know that edges are usually short, which means that we can save space by storing the length of an edge rather than the actual position of the next node. From the length of the edge we can calculate the actual position. As you will see in the ‘Performance Analysis’ section it is very rare to have edges longer than 256 characters, so we only consider two cases. If all the edges are short then next node pointers are stored in one byte, if any of the edges is long next node pointers are stored in 4 bytes. In the array we have as many pointers as the number of nodes and the size of the pointers depends on the size of edges as discussed above.

The next piece of information that we have to store in the array is the pointers to the list of first edges. The first of these pointers is located $(2 * l + 4 + s * 4 + n * \text{number\_of\_nodes})$ bytes from the start position of the box. These are physical pointers to a given memory address. We need as many pointers as the number of nodes stored at this position. A pointer points to the address space where the list of edges running out of that node is stored. It is possible that a pointer points to an area and another pointer addresses edges within that area. These cases will be discussed in detail in the following subsection.

Each first edge pointer points to an area where the list of edges is stored. In the following we will discuss how a list of edges is represented. Each edge must store a next position pointer, which tells the next position in the string where we can follow the matching of a pattern. We store this information in an integer (4 bytes). The 3 most significant bits of this value are saved for some additional information. The first bit flags whether this edge is a leaf or an intermediate edge. If it is a leaf there is no need to store a next node pointer, so in this case the edge is stored in one integer. The next bit flags whether this edge is the last one in the list or there are more edges to follow. Using this technique we do not need to store edges as a linked list connected by pointers, rather we can have a fix array and we check for each edge whether this is the last edge in the list. If it is not we know that the next integer stores another edge. The third bit flags the number of bytes used to store the next node pointer. We follow the same reasoning that we followed in case of the next node pointers for the box. Edges are usually short, so if they are shorter than 256 we store the length of the edge in one byte if they are longer then we store them in an integer (4bytes). The difference here is that we can decide for each next node pointer whether we need one or four bytes but in case of the next node pointers for the box it is determined for the whole array. By using this technique we can always determine the address of the next edge in the list in constant time from the first 3 bits or we learn that this is the last edge in the list. Let l be 0 if this is a leaf and 1 if it is not a leaf. Let n denote the number of bytes used for the next node pointer for the given edge. The number of bytes needed to store the given edge can be calculated using the following formula: $4+l*n$.

3.2 Reduced Nodes and Large Nodes

There are two observations we can make when we analyse the suffix vector of figure 2.

- the nodes at position 5 contain the same information
- some edges at node 1 have the same information

We can store the information of only one node at position 5, thus reducing the number of nodes we have to store. In order to retain memory integrity we set the number of nodes value to 1 at these nodes. It means that we have to store the actual smallest node value somewhere else in the data structure. It is stored after the suffix link (see section 3.1).

We can utilise the second observation by eliminating redundant edge information. It can be proven that the number of edges running out of a node monotonically increases as the node depth decreases for nodes stored at the same position. There are three cases:

- **Rule 1:** the node following in the list has as many edges as the previous one. In this case we simply set their first edge pointers to the same position.
- **Rule 2:** the node following in the list has the same edges as the previous node but it also has some extra edges. In this case the pointer of the previous node will point to the edge in the list of the second node where its own edges start.

- **Rule 3:** the node following in the list does not have the same edges as the previous node. In this case all the edges must be represented in a separate list. We call these nodes large nodes.
- *Figure 4* illustrates this concept. Reduced nodes and large nodes are very important in the matching statistics algorithm [3]. In case a reduced node is found we can save as many steps as the number of nodes represented because we do not need to analyse redundant information. In case Rule 1 is applied when a suffix link is followed (note that this is the case when the suffix link is the next node in the list) we can save comparison again and when Rule 2 applies we only have to check until the beginning of the previous edge because after that all the edges have been checked in the previous step.

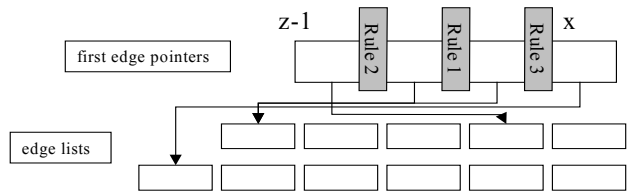


Fig. 4. Large nodes

4 Performance Analysis

The most space-efficient suffix tree representation so far has been developed by Kurtz [7]. He uses a collection of 42 files of different types to compare his representation to others. We compare our representation to his representation in this section but we only show the result to one file in each file group because of the space limits of this paper. Other files in the same group have similar results. In Kurtz’s collection there are English text files (see book2 as an example), program code files (progl), and DNA sequences (K02402). We do not consider binary files because they are not commonly used in suffix tree applications. The following tables show the space requirement of our and his representation as well as some statistical data about our representation that explain why our representation is better in some situations and why his representation is better than ours in other cases.

Table 1. Comparison of the total space requirement

File name:	File size	Vector size	Bytes/symbol Suffix Vector	Bytes/symbol Kurtz
book2	610856	5454903	8.9299328	9.67
Progl	71646	593135	8.2786897	10.22
K02402	38095	488504	12.82331	12.59

Table 1 shows the total space requirement of the test files. For English texts our representation gives better results. The difference varies between 0.11 and 0.74 bytes per symbol. For program source code files our representation is significantly better than

Kurtz’s. The difference varies between 0.67 and 1.95 bytes per symbol. For K02402, which is a DNA sequence Kurtz’s representation is better than ours by 0.23 bytes per symbol.

Table 2. Statistical Data on Suffix Vectors

File name:	File size	Number of nodes	Number of large nodes	Number of reduced nodes	Number of boxes	Number of edges in the tree	Number of edges in the vector	Number of long edges - vector	Number of long depths
Book2	610856	328825	81238	27928	85812	939682	505567	0	220
Progl	71646	46512	6693	2242	6911	118159	66321	742	201
K02402	38095	24364	16574	1422	12706	62460	55614	0	0

In table 2 we show some statistical data that explains why our representation is better in some cases and why Kurtz’s representation is better in other cases. Firstly, let us analyse that part of the data that supports our observations. Observation 1 says that the node depth of the deepest node can usually be stored in one byte. If you consider the last column of the table you can see that this assumption is very true in practice. Observation 2 follows from Observation 1.

The results also support Observation3. The number of long edges is 0 for most files. The largest number of long edges can be found in progl, which means that this file contains long overlaps. This is a program source code, so it is not a surprise that it has long overlaps within itself. The ratio of long edges is still small.

The main advantage of our algorithm is that redundant information is eliminated. If you compare the number of edges in the tree to the number of edges in the suffix vector you can see how many edge-representations can be saved by this technique. The number of reduced nodes also tells us that many nodes represent the same information and they are only stored once in our representation. The number of large nodes value is some way related to the number of edges value because the less the number of large nodes we have the less edges that we have to represent. Note that only edges of large nodes are explicitly represented.

Now we explain why we get much better results for program code. As you can see from the data in the table for program source code we have many nodes represented at each position (compare the number of nodes value to the number of boxes value). These nodes share some information and give a chance for longer sequence of small nodes.

For the DNA sequence we can see that we hardly save any edges and the number of large nodes are very close to the total number of nodes. It means that we have to represent most of the nodes and most of the edges. DNA sequences have more complicated suffix tree structures than natural English texts or program source codes. This complex structure can be slightly more efficiently represented by Kurtz’s implementation.

5 Retrieving Information from the Suffix Vector

The efficiency of storing the tree is only one issue that we have to consider. Retrieving information from the tree is at least as important as the space requirement of the representation. In this section we compare the number of operations needed to retrieve the information on nodes and edges in both representations. There are three basic operations on a suffix tree:

- getting the first edge running out of a node
- getting the next edge from the current edge in the list of edges
- following a suffix link

We divide operations into three categories:

- **Masking.** It is when we have a value (an integer or one byte) that stores multiple pieces of information and we have to mask some bits to retrieve the information we need. We denote the masking operation by **M**.
- **Comparison.** It is when we have to compare two values (two integers or two bytes) and based on the result we choose different execution paths. We denote the comparison operation by **C**.
- **Addition (Subtraction).** It is when we have to add (or subtract) two values. We denote the addition (subtraction) operation by **A**.

Due to space limitations here we are unable to analyse all operations, so we only give the actual number of steps needed in both representations for all operations. These figures may be verified by actually analysing each step. For each operation we give both the worst-case and best-case scenarios.

Table 3. Number of Steps Required to Obtain Information

	Kurtz’s representation		Suffix Vector	
	Worst case	Best case	Worst case	Best case
first edge	10M 5C 8A	6M 3C 2A	7M 5C 4A	6M 4C 3A
next edge	6M 4C 5A	1M 1C	5M 5C 2A	1C
suffix link	Alphabet size dependent	1C 1A	1C 1A	1C 1A

Table 3 shows that both worst-case and best-case data are better in case of the suffix vector representation except for one situation. The best case of getting the first edge is better in case of Kurtz’s representation. The worst case of getting the suffix link information in Kurtz’s representation is when the suffix link is stored at the end of the edge list. It is only necessary if there is a node, whose depth is greater than $2^{10}-1$. This is a very rare situation. Neglecting this scenario the worst case is 3 maskings and 2 additions.

6 Building a Suffix Vector from a Suffix Tree

In this section we show that a suffix vector can be built from a suffix tree. It is not clear yet whether a direct construction algorithm can be used to build suffix vectors. Suffix vectors may still have their application, for example in case of our document overlap program. Firstly, the suffix vector representation uses fewer steps to retrieve the same information (see previous section). We have also analysed the number of steps saved by not analysing redundant information (see section 3.2), which showed that over 20% of the steps could be saved by using suffix vectors.

Before we give a high-level algorithm let us define some basic rules. Our first observation is that a suffix tree built by Ukkonen's algorithm will always have the lowest possible edge labels (note that edges are labelled by the beginning and end position). It is true because in the case of a given substring appearing multiple times in the string it will be inserted into the tree when the first occurrence is encountered (the tree is built from left to right) and later occurrences will already be in the tree, so they will not be added. When a node has to be represented in the vector it is inserted into the position that is defined by the end position of the edge running into the node. Let us define node-depth as the number of characters followed from the root to the node (note that this definition is the same as Kurtz's [7] but it is different from Gusfield's node-depth definition [5]).

The problem we have to face is that we do not want to recreate those arrays that contain next node pointers and point to the first edges. It would be good to know how many nodes we will have at a given position. This can be found in $O(n)$ time by a depth first search of the tree. If a node is found we have two options. This is the first node at a given position: a new box object is created in the position identified by the edge running into that node, the node counter is set to 1, and the deepest node value is set to the node-depth of the given node. Other information in the box is not filled at this stage. The other option is that the box object has been created in some previous step and in this case we only update its information. By the end of the traversal each box object will have the correct value for the number of nodes, and the depth of the deepest node. In the same run large nodes and reduced nodes can also be identified. With another $O(n)$ run we can go through the vector and create the actual boxes when every node is actually inserted: its position in the array is defined by `deepest_node_depth-current_node_depth` and the list of the edges is created. The next node values of edges are simply the difference between the end position and start position value of the edge. Note that we store the difference rather than the actual end position.

7 Conclusion and Future Work

In this paper we have proposed an alternative data structure for representing and storing suffix trees. We compared this representation to the best-known representation to date. We have found that the suffix vector representation is more space-efficient in case of semi-structured textual documents and program code while it is slightly worse in case of DNA sequences. This structure similarly to Kurtz's representation retains the versatility of suffix trees.

Future work will include analysis of how our representation can directly be constructed from scratch in $O(n)$ time. Our data structure, at this stage, has the advantage of simplicity over Kurtz's representation besides space efficiency, which is a very important argument. This advantage is demonstrated in the number of steps needed to obtain information from the tree. Our representation also eliminates some redundant information of a tree, which also saves some time on algorithms that run on the proposed data structure.

Acknowledgement

Support from Distributed Systems Technology Centre (DSTC Pty Ltd) for this project is thankfully acknowledged.

References

1. Anderson A., Nilson S. Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters*, 46:295-300, 1993.
2. Apostolico A. (1985). The Myriad Virtues of Subword Trees, in A. Apostolico and Z.Galli. *Combinatorial Algorithms on Words*. (Springer-Verlag Heidelberg) pp. 85-96.
3. Chang W.I., Lawler. E.L. Sublinear Approximate String Matching and Biological Applications. *Algorithmica* 12, pp. 327-344, 1994.
4. Giegerich R., Kurtz S. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19:331-353, 1997.
5. Gusfield D. Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, 1997.
6. Kärkkäinen J. Suffix Cactus: A Cross between Suffix Tree and Suffix Array. In Proceedings of the Annual Symposium on combinatorial Pattern Matching (CPM'95), LNCS 937, pages 191-204, 1995.
7. Kurtz S.: Reducing the Space Requirement of Suffix Trees. *Software-Practice and Experience*, 29(13), pages 1149-1171, 1999.
8. Manber U., Myers E.W. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935-948, 1993.
9. McCreight. E. M. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262-272, 1976.
10. Monostori K., Schmidt H., Zaslavsky A. Document Overlap Detection System for Distributed Digital Libraries. *ACM Digital Libraries 2000 (DL00)*, 2-7 June, 2000 in San Antonio, Texas, USA.
11. Ukkonen. E. On-Line Construction of Suffix Trees. *Algorithmica* 14. pp 249-260, 1995
12. Weiner P. Linear Pattern Matching Algorithms. *Proceedings of the 14th IEEE Annual Symposium on switching and Automata Theory*, p1-11, The University of Iowa, 15-17 October, 1973.

Fragmentary Pattern Matching: Complexity, Algorithms and Applications for Analyzing Classic Literary Works

Hideaki Hori¹, Shinichi Shimozono^{2*}, Masayuki Takeda^{3,4}, and Ayumi
Shinohara³

¹ Graduate School of Computer Science and Systems Engineering
Kyushu Institute of Technology, Iizuka 820-8502, Japan

hori@daisy.ai.kyutech.ac.jp

² Department of Artificial Intelligence, Kyushu Institute of Technology
Iizuka 820-8502, Japan

sin@ai.kyutech.ac.jp

³ Department of Informatics

Kyushu University 33, Fukuoka 812-8581, Japan

{takeda, ayumi}@i.kyushu-u.ac.jp

⁴ PRESTO, Japan Science and Technology Corporation, Japan

Abstract. A fragmentary pattern is a multiset of non-empty strings, and it matches a string w if all the strings in it occur within w without any overlaps. We study some fundamental issues on computational complexity related to the matching of fragmentary patterns. We show that the fragmentary pattern matching problem is NP-complete, and the problem to find a fragmentary pattern common to two strings that maximizes the pattern score is NP-hard. Moreover, we propose a polynomial-time approximation algorithm for the fragmentary pattern matching, and show that it achieves a constant worst-case approximation ratio if either the strings in a pattern have the same length, or the importance weights of strings in a pattern are proportional to their lengths.

Keywords: fragmentary pattern, string resemblance, string matching, NP-completeness, polynomial-time approximation

1 Introduction

Waka is a form of traditional Japanese poetry with 1300-year history. A *Waka* poem has five lines and thirty-one syllables, arranged thus: 5-7-5-7-7. Since one syllable is represented by one Kana character in Japanese, a *Waka* poem consists of thirty-one Kana characters. In [13], we attempted to discover similar poems semi-automatically from an accumulation of about 450,000 *Waka* poems in a machine-readable form. One of the aims is to find unheeded instances of *Honkadori*, a technique based on specific allusion to earlier famous poems. The

[□] This research is partially supported by Grants-in-Aid for Encouragement of Young Scientists, Japan Society for the Promotion of Science, No. 12780286.

approach we took is very simple: Arrange all possible pairs of poems in decreasing order of their similarity, and scholarly scrutinize a first part.

The key to success in this approach would be how to develop an appropriate similarity measure. Traditionally, the scheme of weighted edit distance with a weight matrix may have been used to quantify affinities between strings (see e.g. [10]). This scheme, however, requires a fine tuning of quadratically many weights in a matrix with the size of alphabet, by a hand-coding or a heuristic criterion. As an alternative idea, we introduced a new framework called *string resemblance systems* (SRSs for short) [13]. In this framework, similarity of two strings is evaluated via a pattern that matches both of them, with the support by an appropriate function that associates the quantity of resemblance to candidate patterns. This scheme bridges a gap among optimal pattern discovery (e.g. [12]), machine learning (e.g. [2,3]) and similarity computation (e.g. [6,10]).

An SRS is specified by (1) a *pattern set* to which common patterns belong, and (2) a *pattern score function* that maps each pattern in the set to the quantity of resemblance. For example, if we choose the set of patterns with *variable-length don't-cares* (VLDC's) and define the pattern score to be the number of non-variable symbols in a pattern, then we obtain one of the traditional measures, the longest common subsequence (LCS): a common pattern $a\star d\star a\star$ for both *acdeba* and *abdac*, whose score is three. With this framework researchers can easily design and modify their measures not only for generic purposes but also for definite usages. In fact, we designed several similarity measures as combinations of a pattern set and a score function along with this framework, and reported successful results in discovering instances of Honkadori [13].

Some of the similarity measures employed in [13] base upon a class of *fragmentary patterns*, or *order-free patterns*. A fragmentary pattern is formally a multiset of non-empty strings. It matches a string w if all the strings in it occur within w without any overlaps. Although the computational complexity of matching a fragmentary pattern had not been clarified, the potential intractability to deal with it could be ignored for comparing Waka poems, since the lengths of the poems are only approximately 31.

However, the computational complexity is crucial and must be paid attention to when comparing longer texts by a fragmentary pattern. For example, searching for a fragmentary pattern in long texts arises in detecting instances of *Hikiuta*. *Hikiuta* is a rhetorical device used in *Monogatari* (tales), which is based on a specific allusion to a famous poem and appears in the narrative, conversation, and letters. A prose passage of the tale and the poem, therefore, share a phrase or part of phrase when this device is used. Other possible applications in molecular biology require that methods can process efficiently for huge size of sequences.

The purpose of this paper is to settle some fundamental issues on computational complexity related to the matching of fragmentary patterns and the string resemblance system adopting them. Firstly, we show that a matching decision of a fragmentary pattern is NP-complete. This indicates that if a pattern contains strings whose suffices and prefixes can overlap, then finding a set of non-overlapping occurrences of the strings becomes intractable. Also, we prove that

the problem to find a fragmentary pattern that is common to two strings and maximizes the pattern score is NP-hard. Furthermore we present a polynomial-time approximation algorithm for the maximization version of the fragmentary pattern matching, and show that the algorithm achieves a constant worst-case approximation ratio if (i) the strings in a pattern have the same length, or (ii) the importance weights of strings in a pattern are the lengths of them.

The rest of this paper is organized as follows. Section 2 gives a brief sketch of the framework of string resemblance systems. Section 3 defines the class of fragmentary patterns and then proves that the pattern matching problem for this class is NP-complete. Section 4 discusses the complexity required for computing similarity between two strings for SRSs with the fragmentary patterns. Section 5 considers combinatorial optimization versions of the fragmentary pattern matching and gives an approximation algorithm. Section 6 describes applications to two typical problems arisen in analysis of classic Japanese literary works.

2 A Unifying Framework for String Similarity

This section briefly sketches the framework of string resemblance systems according to [13]. Gusfield [10] pointed out that in dealing with string similarity the language of alignments is often more convenient than the language of edit operations. Our framework is a generalization of the alignment based scheme and is based on the notion of *common patterns*.

Before describing our scheme, we introduce some notations. The set of all strings over a finite alphabet Σ is denoted by Σ^* . The length of a string $s \in \Sigma^*$ is denoted by $|s|$. The *empty string* ε is the string of length zero. The set $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ thus denotes the set of all non-empty strings.

A *pattern system* is a triple $\langle \Sigma, \Pi, L \rangle$ of a finite alphabet Σ , a set Π of descriptions called *patterns*, and a function L that maps a pattern $\pi \in \Pi$ to a *language* $L(\pi) \subseteq \Sigma^*$. A pattern $\pi \in \Pi$ *matches* $w \in \Sigma^*$ if w belongs to $L(\pi)$. Also, π is a *common pattern* of w and u for strings $w, u \in \Sigma^*$, if π matches both of them. Usually, a set Π of patterns is expressed as a set of strings over an alphabet $\Sigma \cup X$, where X is a finite alphabet which is disjoint to Σ .

Definition 1. A string resemblance system (SRS) is a quadruple $\langle \Sigma, \Pi, L, \text{Score} \rangle$, where $\langle \Sigma, \Pi, L \rangle$ is a pattern system and *Score* is a pattern score function that maps a pattern in Π to a real number.

The similarity $\text{SIM}(x, y)$ between strings x and y with respect to an SRS $\langle \Sigma, \Pi, L, \text{Score} \rangle$ is defined by

$$\text{SIM}(x, y) = \max\{\text{Score}(\pi) \mid \pi \in \Pi \text{ and } x, y \in L(\pi)\}.$$

When the set $\{\text{Score}(\pi) \mid \pi \in \Pi \text{ and } x, y \in L(\pi)\}$ is empty or the maximum does not exist, $\text{SIM}(x, y)$ is undefined.

The definition given above regards the similarity computation as *optimal pattern discovery*. In this sense, our framework bridges a gap between similarity

computation and pattern discovery. In [13], the class of homomorphic SRSs was defined, and it was shown that the class covers most of the well-known and well-studied similarity (dissimilarity) measures, including the edit distance, the weighted edit distance, the Hamming distance, the LCS measure. Also this class was extended to the semi-homomorphic SRSs in [13], into which for example the similarity measures for musical sequence comparison developed in [11] falls.

Interestingly, membership problems of homomorphic and semi-homomorphic pattern systems are assumed reasonably to be polynomial-time solvable, while membership problems of non-homomorphic pattern systems include NP-complete one, e.g. the Angluin pattern system [1]. The similarity computation for homomorphic and semi-homomorphic SRSs can be performed in polynomial time [13] by the idea of weighted edit graph (see, e.g., [10]) under the above assumption, while the similarity computation via the Angluin pattern system is NP-hard in general [14]. We emphasize that the fragmentary pattern system is included in the class of non-homomorphic pattern systems.

3 Fragmentary Patterns and Complexity of Their Matching

We focus on the class of fragmentary patterns in this section, and discuss the computational complexity of a matching or a searching of an arbitrary large fragmentary pattern, before looking into SRSs adopting this class.

A *fragmentary pattern* over Σ is a multiset $\{p_1, \dots, p_\ell\}$ of $\ell > 0$ non-empty strings $p_1, \dots, p_\ell \in \Sigma^+$, and is denoted by $\pi[p_1, \dots, p_\ell]$. The size of a fragmentary pattern $\pi[p_1, \dots, p_\ell]$ is the total length of strings $p_1, \dots, p_\ell$, and denoted by $\|\pi\|$.

Definition 2 (Fragmentary pattern system). *The fragmentary pattern system on Σ is a pattern system $\langle \Sigma, \Pi, L \rangle$ such that (i) Π is the set of all fragmentary patterns over Σ , and (ii) L is the function that maps $\pi[p_1, \dots, p_\ell] \in \Pi$ to the language $L(\pi[p_1, \dots, p_\ell])$ that contains all strings expressed by*

$$s_0 \cdot p_{\sigma(1)} \cdot s_1 \cdot p_{\sigma(2)} \cdot s_2 \cdots s_{\ell-1} \cdot p_{\sigma(\ell)} \cdot s_\ell,$$

where $s_0, s_1, \dots, s_\ell$ are arbitrary strings in Σ^* and $\langle \sigma(1), \dots, \sigma(\ell) \rangle$ is an arbitrary permutation of integers $1, \dots, \ell$.

For example, the language of the pattern $\pi[abc, de]$ is denoted by a regular expression

$$L(\pi[abc, de]) = \Sigma^* abc \Sigma^* de \Sigma^* \cup \Sigma^* de \Sigma^* abc \Sigma^*.$$

In the context of a string pattern matching, the following notions are convenient. Let p and t be strings over Σ^* . An *occurrence position* i of p in t is an integer such that $p = t[i] \cdots t[i + |p| - 1]$. The range $[i, i + |p| - 1]$ on t represents the substring $t[i] \cdots t[i + |p| - 1]$ and is said to be an *occurrence* of p in t . A fragmentary pattern $\pi[p_1, \dots, p_\ell]$ *matches* $t \in \Sigma^*$ if there is a sequence $\langle k_1, \dots, k_\ell \rangle$ of integers such that (i) every k_i for $1 \leq i \leq \ell$ is an occurrence position of p_i

in t , and (ii) $k_i + |p_i| - 1 < k_j$ holds for any $k_i < k_j$, i.e. any pair of occurrences never overlap. We say such a sequence $\langle k_1, \dots, k_\ell \rangle$ an occurrence of π in t .

Then the following is a fundamental problem for a fragmentary pattern system $\langle \Sigma, \Pi, L \rangle$ on Σ .

Definition 3. FRAGMENTARY PATTERN MATCHING (FRAG-MATCHING)

Given a fragmentary pattern $\pi \in \Pi$ and a string $w \in \Sigma^*$, determine whether w belongs to $L(\pi)$.

This may rather seem to be tractable. Actually, if no pair of strings in a fragmentary pattern shares a common string as a prefix and a suffix, then strings in a pattern cannot overlap and thus this problem is solvable in polynomial time. It is a simple ‘AND’ query of multiple string patterns. However, in general, the following theorem holds.

Theorem 1. FRAGMENTARY PATTERN MATCHING is NP-complete.

Firstly, we prove this theorem by a reduction from 3SAT to FRAG-MATCHING, with which a reduced instance requires an alphabet whose size depends on the size of a given 3CNF formula. After showing it, we briefly discuss how those symbols can be expressed over an alphabet of fixed size. The problem 3SAT (e.g. [8]) is, given a set $C = \{c_1, \dots, c_m\}$ of 3 literal clauses over a set $X = \{x_1, \dots, x_n\}$ of Boolean variables, to determine whether C is satisfiable.

Proof. In the following we show a logspace algorithm that builds an instance (t_C, P_C) of FRAGMENTARY PATTERN MATCHING over an alphabet

$$\Sigma_C = \{x_1, \dots, x_n, c_1, \dots, c_m, \#\}$$

for a 3SAT instance (X, C) .

We introduce some gadgets utilized to construct t_C and P_C . For each $1 \leq i \leq n$, we define $t_1^i = x_i x_i c_1 x_i c_2 x_i \dots c_m x_i x_i \#$, and $t_2^i = t_2^i(1) \dots t_2^i(m)$ where

$$t_2^i(j) = \begin{cases} c_j c_j x_i c_j \# & \text{if } c_j \text{ contains } x_i, \\ c_j x_i c_j c_j \# & \text{if } c_j \text{ contains } \neg x_i, \\ c_j x_i c_j \# & \text{if neither } x_i \text{ nor } \neg x_i \text{ is in } c_j, \end{cases}$$

for $1 \leq j \leq m$. With these gadgets, we define $t_1 = t_1^1 \dots t_1^n$ and $t_2 = t_2^1 \dots t_2^n$, and as the concatenation $t_C = t_1 \cdot t_2$. The pattern P_C is defined by the union of $P_1 = \cup_{i=1}^n \{x_i x_i\}$, $P_2 = \cup_{j=1}^m \{c_j c_j\}$ and $P_3 = \cup_{i=1}^n \{c_1 x_i, x_i c_1, \dots, c_m x_i, x_i c_m\}$. Note that P_C contains only strings of the length two. Clearly, this algorithm runs with logarithmic space.

The gadgets defined above have the following properties: (i) P_1 matches t_1 , while any string in it does not match t_2 ; (ii) for each $1 \leq i \leq n$, the string $x_i x_i \in P_1$ is either the prefix of t_1^i , or the suffix of t_1^i ; and (iii) P_2 matches t_2 , while any string in it does not match t_1 . Also, for each $1 \leq i \leq n$ and $1 \leq j \leq m$, either $c_j x_i$ or $x_i c_j$ in P_3 matches t_1^i , and the remaining one matches t_2^i .

Now, we prove that (X, C) is satisfiable if and only if P_C matches t_C . Firstly, we show that if there is a truth assignment $f : X \rightarrow \{\mathbf{true}, \mathbf{false}\}$ that satisfies (C, X) , then an occurrence of P_C in t_C exists.

According to the assignment f , we split $P_1 \cup P_3$ into two sets: we define, for each $1 \leq i \leq n$,

$$Q_1^i = \{x_i x_i, c_1 x_i, \dots, c_m x_i\}, \quad Q_2^i = \{x_i c_1, \dots, x_i c_m\}$$

if $f(x_i)$ is **true**, and otherwise (if $f(x_i) = \mathbf{false}$) define

$$Q_1^i = \{x_i x_i, x_i c_1, \dots, x_i c_m\}, \quad Q_2^i = \{c_1 x_i, \dots, c_m x_i\}.$$

Note that Q_1^i and Q_2^i matches t_1^i and t_2^i , respectively, without depending on whether $f(x_i)$ is **true** or **false**. Then, since f satisfies C , for each $1 \leq j \leq m$, there must be an index $1 \leq i \leq n$ such that either x_i or $\neg x_i$ satisfies c_j .

This can be interpreted with the above definition that for each $1 \leq j \leq m$ there is a variable index $1 \leq i \leq n$ such that either (a) $c_j c_j x_i c_j \#$ occurs in t_2^i and $x_i c_j$ is in Q_2^i , or (b) $c_j x_i c_j c_j \#$ occurs in t_2^i and $c_j x_i$ is in Q_2^i . Then, in t_2^i there remains a substring $c_j c_j$ to which a string $c_j c_j$ in P_2 of the pattern matches. This guarantees that P_2 can, with all Q_2^i 's, match t_2 , and thus the whole fragmentary pattern P_C matches t_C .

Next we show that if P_C matches t_C then a truth assignment associated with an occurrence of P_C satisfies C .

By the construction of (t_C, P_C) , for each $1 \leq i \leq n$, either the pattern

$$\{x_i x_i, c_1 x_i, \dots, c_m x_i\}, \text{ or } \{x_i x_i, x_i c_1, \dots, x_i c_m\}$$

must match t_1^i ; otherwise we lose all the possible places where $x_i x_i$ in P_1 occurs. With respect to this choice, we define the set $P_T^i \subseteq P_3$ as either $\{x_i c_1, \dots, x_i c_m\}$ or $\{c_1 x_i, \dots, c_m x_i\}$ for each $1 \leq i \leq n$. Also, we define $P_T = \bigcup_{i=1}^n P_T^i$ and $P_F = P_3 - P_T$. Then, $P_1 \cup P_T$ matches t_1 , and this requires that $P_2 \cup P_F$ matches t_2 . For each $1 \leq i \leq m$, there is an index $1 \leq i \leq n$ such that either (a) t_2 contains $c_j c_j x_i c_j \#$ and $x_i c_j$ is in P_F , or (b) t_2 contains $c_j x_i c_j c_j \#$ and $c_j x_i$ is in P_F . Otherwise we have no positions to which $c_j c_j$ matches without overlaps.

According to the occurrence of P_C in t_C inspected as above, we define a truth assignment f as follows: $f(x_i) = \mathbf{true}$ if P_T^i includes $c_j x_i$ ($1 \leq j \leq m$); $f(x_i) = \mathbf{false}$ if P_T^i includes $x_i c_j$ ($1 \leq j \leq m$). Then, since P_F and P_2 must match t_2 , like the discussion on Q_2^i 's and P_2 in above, the assignment f implies that for each clause in C there is at least one literal having **true**. Therefore, C is satisfiable if P_C matches t_C .

The above two properties complete this proof. $\square$

The reduction presented here can be easily modified to one that reduces to an instance of FRAG-MATCHING over an alphabet consisting of a fixed number of symbols. For example, an alphabet $\Sigma = \{0, 1, \$\}$ could be used to represent finitely many symbols in Σ_C by distinguished binary strings of the same length, followed with the separator symbol '\$.' The coding sizes of P_C and t_C is expanded only $\log |\Sigma_C|$ times the original represented with Σ_C . Even the unary coding scheme can be applied.

Corollary 1. *FRAGMENTARY PATTERN MATCHING is NP-complete even if either (i) the size of the alphabet is fixed, or (ii) strings in a pattern are of the same length, or both.*

4 Complexity of Similarity Computation by Fragmentary Patterns

We now consider the computation of similarity between two strings and its computational complexity. In the following, we assume the values of score function are integers.

Definition 4. *SIMILARITY COMPUTATION WITH SRS $\langle \Sigma, \Pi, L, \text{Score} \rangle$.*

Given two strings $w_1, w_2 \in \Sigma^*$, find a pattern $\pi \in \Pi$ with $\{w_1, w_2\} \subseteq L(\pi)$ that maximizes $\text{Score}(\pi)$.

Let $\#$ be a symbol not in Σ , and π a fragmentary pattern $\pi[u_1, \dots, u_\ell]$ over Σ . For a fragmentary pattern π' over Σ , we write $\pi' \preceq \pi$ if π' matches the string $u_1\# \dots u_\ell\#$ in $(\Sigma \cup \{\#\})^*$. Here, the function L is naturally extended to one that maps a pattern to the language $L(\pi_1)$ over $\Sigma \cup \{\#\}$. We write as $\pi_1 \prec \pi_2$ if $\pi_1 \preceq \pi_2$ and the two multisets π_1 and π_2 are not identical. A pattern score function Score is *strictly increasing with respect to* $\prec$ if $\pi_1 \prec \pi_2$ implies $\text{Score}(\pi_1) < \text{Score}(\pi_2)$. For example, let $\text{Score}_1(\pi) = \|\pi\|$ and $\text{Score}_2(\pi[u_1, \dots, u_\ell]) = \sum_{i=1}^{\ell} |u_i|^2$. Then, Score_2 is strictly increasing, while Score_1 is not.

Theorem 2. *SIMILARITY COMPUTATION WITH SRS with the fragmentary pattern system is NP-hard in general.*

Proof. We show the NP-completeness of a decision version of SIMILARITY COMPUTATION with the class of pattern score functions that are strictly increasing: Given two strings $w_1, w_2 \in \Sigma^*$ and a nonnegative integer k , determine whether a pattern $\pi \in \Pi$ satisfying $\{w_1, w_2\} \subseteq L(\pi)$ and $\text{Score}(\pi) \geq k$ exists.

We give a reduction from FRAGMENTARY PATTERN MATCHING $\langle \Sigma, \Pi, L \rangle$ to SIMILARITY COMPUTATION WITH SRS $\langle \Sigma', \Pi', L', \text{Score} \rangle$. A triple $\langle \Sigma', \Pi', L' \rangle$ is the fragmentary pattern system on $\Sigma' = \Sigma \cup \{\#\}$, and Score is a pattern score function defined on the set of fragmentary patterns Π' over Σ' , whose limitation to $\Pi \subseteq \Pi'$ is strictly increasing with respect to $\prec$.

For a given instance $\pi = \pi[u_1, \dots, u_\ell] \in \Pi$ and $w \in \Sigma^*$ of FRAGMENTARY PATTERN MATCHING, we construct an instance (w'_1, w'_2, k) of SIMILARITY COMPUTATION by letting $w'_1 = u_1\# \dots u_\ell\#$, $w'_2 = w$, and $k = \text{Score}(\pi)$. Since $\#$ does not occur in w'_2 , there is a pattern $\pi' \in \Pi'$ with $\{w'_1, w'_2\} \subseteq L'(\pi')$ and $\text{Score}(\pi') \geq k$ if and only if $w \in L(\pi)$. This completes the proof. $\square$

On the other hand, there are pattern score functions that are not trivial and with which similarity can be efficiently computed. For example, with the pattern score function that can be considered as an *order-free version* of LCS, we can readily show that:

Theorem 3. SIMILARITY COMPUTATION WITH RESPECT TO SRS with the fragmentary pattern system is solvable in linear time using $O(|\Sigma|)$ space for the pattern score function $\text{Score}(\pi) = \|\pi\|$.

5 Maximization of Fragmentary Pattern Matching

More than a powerful pattern class for the similarity computation, fragmentary patterns can be used as a conjunction of queries for texts in which word-boundaries are not evident. By viewing the matching problem as a combinatorial optimization problem, a fragmentary pattern can be thus applied like an at-least- k -of- m rule. It is regarded as a generalization of the membership problem of fragmentary patterns, to classify noisy inputs with a specified robustness.

So now we consider the problem to find a maximal subset of a given set of strings that matches a text as a fragmentary pattern. Firstly, we introduce some notions of combinatorial optimization problems. In the following we only deal with and thus define ‘maximization versions’ of combinatorial optimization problems. (See e.g. [4,5] for details.)

A *maximization problem* P is specified by (i) the set I_P of instances, (ii) the set $S_P(x)$ of *solutions* of each instance $x \in I$, and (iii) the *measure* $m_P(x, s)$ that maps a pair of an instance x and a solution s of x to a nonnegative integer. The ultimate goal of a maximization problem is to find an *optimum solution*, that is, a solution whose measure is maximum. An *approximation algorithm* A for P is an algorithm that produces for any instance $x \in I_P$ a solution $s \in S_P(x)$. Furthermore, for a rational number $r > 1$, A is said to be an *r -approximation algorithm for P* if A always produces a solution whose measure is no less than $1/r$ times the measure of an optimum solution. A maximization problem P is in class APX if there is a polynomial-time r -approximation algorithm for P with some constant r .

A maximization version of our pattern matching problem is formalized as follows.

Definition 5. MAXIMUM FRAGMENTARY PATTERN MATCHING (MAX FRAG-MATCHING)

Given a weighted instance of FRAG-MATCHING, i.e. a triple (π, w, t) of a fragmentary pattern $\pi \in \Pi$, a weight $w : \pi \rightarrow \mathbb{Z}^+$ and a string $t \in \Sigma^*$, find a fragmentary pattern $\pi' \subseteq \pi$ that matches t and maximizes the total weight $\sum_{u \in \pi'} w(u)$ in π' .

For this maximization problem, let us consider the following simple polynomial-time algorithm.

Algorithm Greedy

Input: An instance triple (π, w, t) ;

Output: A fragmentary pattern $\pi' \subseteq \pi$ that matches t .

1. Let $\pi' = \emptyset$, and let I be an empty list of occurrences.
2. For each $u \in \pi$, in the weight-descending order with respect to w , do the following:

- a. Find an occurrence of u in t , say $[k, \ell]$, which does not overlap any occurrences in I ; If no such an occurrence can be found, then continue to the next iteration to proceed to the next string in π .
 - b. Add u to π' , and add the occurrence $[k, \ell]$ to I .
3. Output π' .

This algorithm runs in $O(n \log n + m)$ time with the number n of strings in π and the length m of string t , by employing appropriate sorting, set managing and string matching algorithms. Furthermore, with certain kinds of restrictions on input strings or weight functions, the following lemmas hold:

Lemma 1. *If all the strings in π have the same length, then the algorithm Greedy is a 3-approximation algorithm, i.e. guarantees an output whose total weight is at least $1/3$ times the total weight of an optimum solution.*

Proof. Let $\pi^* \subseteq \pi$ be an optimum fragmentary pattern for t . An addition of string u to π' with some occurrence, in an iteration at the step 2-b, can interfere at most two strings in π^* matching t . For these two strings, there are following three cases: (i) each of the two strings has the weight less than $w(u)$, (ii) the two strings are already chosen in π' , or (iii) the two strings are interfered by some string already chosen in π' . Therefore the addition of u disables the contributions of weights from π^* no more than $2w(u)$, while in π^* the two strings and u may contribute totally at most $3w(u)$. By repeating this process, we finally obtain a solution whose total weight is at least $\frac{1}{3}$ times the optimum. $\square$

Lemma 2. *If the weight of each string is the length of it, then the algorithm Greedy is a 4-approximation algorithm.*

Proof. This can be shown by a discussion similar to the previous proof. An addition of u to π' at each iteration of the Step 2-b may block some strings in π^* occurring in the text. Since $|u| = w(u)$ contiguous symbols are occupied by the occurrence of u , the total weight of those blocked strings is at most $w(u) - 2 + 2w(u) < 3w(u)$. The string u may also be included in π^* , so the algorithm is guaranteed to choose a fragmentary pattern whose total weight is no less than $\frac{1}{4} = \frac{w(u)}{w(u)+3w(u)}$ times the optimum. $\square$

Note that the restricted subproblem considered in lemma 1 includes instances constructed in the reduction presented in Section 3. Also the case dealt with lemma 2 seems likely to occur in practical applications, since shorter strings may have less meaning in general, and in automated pattern discovery some automatic weighting scheme will be requested.

Corollary 2. MAX FRAGMENTARY PATTERN MATCHING is in the class APX [5] if strings in a fragmentary pattern have the same length. Also the problem is in APX if the weight function is equal to or stronger than the length of string.

6 Applications for Classic Literary Works

Honkadori is a technique of composing a Waka poem as an allusive-variation of a model poem. In [13], we developed a similarity measure appropriate for finding instances of Honkadori, based on a measure to quantify affinities between two lines which falls into the class of semi-homomorphic SRSs mentioned in Section 2. With this measure we have succeeded to discover instances of Honkadori which have never been pointed out in the long research history of Waka poetry. In [13] we also showed two similarity measures, which are defined as SRSs with fragmentary pattern systems. The difference of the two measures lies in the pattern score functions. Each of the pattern score functions can be described as

$$\text{Score}(\pi[u_1, \dots, u_\ell]) = \sum_{i=1}^{\ell} f(u_i) \quad (1)$$

with a function f that maps a string in Σ to a real number. One measure is obtained by letting

$$f(u) = \begin{cases} |u|, & \text{if } |u| > \ell; \\ 0, & \text{otherwise,} \end{cases} \quad (2)$$

where ℓ is a threshold in ignoring short fragments in a common pattern. In [13], we set $\ell = 1$. This measure is suitable for discovering instances of Honkadori with word-order alternations, as shown in Fig. 1.

Poem alluded to. (Kokin-Shū #125)

KA-HA-TSU-NA-KU/I-TE-NO-YA-MA-FU-KI/CHI-RI-NI-KE-RI
HA-NA-NO-SA-KA-RI-NI/A-HA-MA-SHI-MO-NO-WO

Allusive-variation. (Shin-Kokin-Shū #1162)

A-SHI-HI-KI-NO/YA-MA-FU-KI-NO-HA-NA/CHI-RI-NI-KE-RI
I-TE-NO-KA-HA-TSU-HA/I-MA-YA-NA-KU-RA-MU

Fig. 1. An instance of Honkadori with word-order alternations

Although SIMILARITY COMPUTATION for this score function is NP-hard, the length of Waka poems we dealt with was approximately 31. Thus we could have performed the computation in feasible time.

The other measure is obtained by letting $f(u)$ be the *rarity* of string u , that is, $f(u)$ is the logarithm of inverse of the probability of occurring u in database. The idea of rarity was shown to be effective in identifying only close affinities which are hardly seen elsewhere, possibly excluding known stereotype expressions [13].

Hikiuta is a poetic device used in tales, which is based on a specific allusion to a famous poem. We wish to find a portion of a tale which alludes to a poem. We use an SRS with fragmentary pattern system to quantify the affinities between

a substring of a tale and a poem. For this purpose, the length of a substring to be compared to a poem has to be limited by an appropriate threshold called *window size*, as in the *episode matching* (e.g. [9]). Our problem is then formalized as follows:

Given a short string, called *poem*, a long string, called *tale*, a window size $k > 0$, and a threshold t , to find all substrings of the tale that are of length k and resemble the poem with a similarity value higher than t .

Preliminary experimental results suggest that the pattern score function defined by Eq. 1 and Eq. 2 with a relatively large value of ℓ might be suitable for effectively detecting instances of Hikiuta within a tale. A practically efficient approach would be a filtering technique based on searching of fragments of the poem that are of length greater than the threshold ℓ within the tale, in which such index structures as the directed acyclic word graphs (e.g. [7]) will play a key role, and verification of candidate areas of the tale.

Acknowledgments

The authors would be grateful to the anonymous referees for their careful reading of the draft and useful comments.

References

1. D. Angluin. Finding patterns common to a set of strings. *J. Comput. Sys. Sci.*, 21:46–62, 1980. 722
2. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987. 720
3. D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988. 720
4. G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation*. Springer-Verlag, Berlin, 1999. 726
5. G. Ausiello, P. Crescenzi, and M. Protasi. Approximate solution of NP optimization problems. *Theor. Comput. Sci.*, 150: 1–55, 1995. 726, 727
6. A. Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of Sequences (SEQUENCES'97)*, pages 21–29, 1997. 720
7. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994. 729
8. M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman & Co., New York, 1979. 723
9. G. Das, R. Fleischer, L. Gasieniec, D. Gunopulos, and J. Karkkainen. *Episode Matching*. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM'97)*, pages 12–27, 1997. 729
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, 1997. 720, 721, 722

11. T. Kadota, M. Hirao, A. Ishino, M. Takeda, A. Shinohara, and F. Matsuo. Musical sequence comparison for melodic and rhythmic similarities. In *Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE2001)*, 2001, to appear. 722
12. S. Shimozono, H. Arimura, and S. Arikawa. Efficient discovery of optimal word-association patterns in large databases. *New Gener. Comput.*, 18(1):49–60, 2000. 720
13. M. Takeda, T. Fukuda, I. Nanri, M. Yamasaki, and K. Tamari. Discovering instances of poetic allusion from anthologies of classical Japanese poems. *Theor. Comput. Sci.*, 2001, to appear. 719, 720, 721, 722, 728
14. K. Yamamoto, M. Takeda, A. Shinohara, T. Fukuda, and I. Nanri. Discovering repetitive expressions and affinities from anthologies of classical Japanese poems. In *Proc. 4th International Conference on Discovery Science (DS2001)*, 2001, to appear. 722

Computing the Quartet Distance between Evolutionary Trees in Time $O(n \log^2 n)$ [★]

Gerth Stølting Brodal, Rolf Fagerberg, and Christian N. S. Pedersen

BRICS – Basic Research in Computer Science^{□□}, Department of Computer Science,
University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark
{gerth,rolf,cstorm}@brics.dk www.brics.dk

Abstract. Evolutionary trees describing the relationship for a set of species are central in evolutionary biology, and quantifying differences between evolutionary trees is an important task. One previously proposed measure for this is the quartet distance. The quartet distance between two unrooted evolutionary trees is the number of quartet topology differences between the two trees, where a quartet topology is the topological subtree induced by four species. In this paper, we present an algorithm for computing the quartet distance between two unrooted evolutionary trees of n species in time $O(n \log^2 n)$. The previous best algorithm runs in time $O(n^2)$.

1 Introduction

The evolutionary relationship for a set of species is commonly described by an evolutionary tree. This is a rooted tree where the leaves correspond to the species, and the internal nodes correspond to speciation events, i.e. the points in time where the evolution has diverged in different directions. The direction of the evolution is described by the location of the root, which corresponds to the most recent common ancestor for all the species, and the rate of evolution is described by assigning lengths to the edges. The true evolutionary tree for a set of species is rarely known, hence estimating it from obtainable information about the species, e.g. genomic data, is of great interest. The problem of computationally estimating aspects of the true evolutionary tree requires a model describing how to use the available information about the species in question. Given a model, the problem of estimating certain aspects of the true evolutionary tree is often referred to as constructing the evolutionary tree in that model. Many models and methods for constructing evolutionary trees have been presented, see [10, Chap. 17] for an overview.

An important aspect of the true evolutionary tree is the undirected tree topology induced by ignoring the location of root and the length of the edges. Many models and methods are concerned with estimating this tree topology,

[□] Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

^{□□} funded by the Danish National Research Foundation.

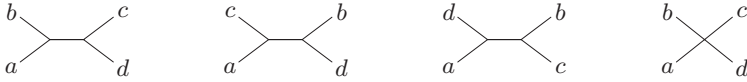


Fig. 1. The four possible quartet topologies of species a , b , c , and d

usually under the further assumption that all internal nodes have degree three. We say that such models and methods are concerned with constructing the unrooted evolutionary tree of degree three for a set of species. For the remainder of this paper an evolutionary tree denotes an unrooted evolutionary tree of degree three.

Different models and methods often yield different estimates of the evolutionary tree for the same set of species. The same model and method can also give rise to different evolutionary trees for the same set of species when applied to different information about the species, e.g. different genes. To study such differences in a systematic manner, one must be able to quantify differences between evolutionary trees using well-defined and efficient methods.

One approach for comparing two evolutionary trees is to determine a consensus tree (or forest) that reflects common traits of the two trees, e.g. the maximum agreement subtree. Much work has been concerned with developing efficient methods for computing the maximum agreement subtree of two or more evolutionary trees, see e.g. [2]. Another approach for comparing two evolutionary trees is to define a distance measure between two trees and compare the two trees by computing the distance. Several distance measures have been proposed, e.g. the symmetric difference metric [12], the nearest-neighbor interchange metric [16], the subtree transfer distance [1], the Robinson and Foulds metric [13], and the quartet metric [8]. Each distance measure has different properties and reflects different aspects of biology, e.g. the subtree transfer distance is related to the number of recombinations between the two sets of species. The quartet metric has several attractive properties. Bryant *et al.* in [5] discuss the properties of the quartet metric and conclude that it does not suffer from drawbacks of the other distance measures. For example, measures based on transformation operations, e.g. the subtree transfer distance, do not distinguish between transformations that affect a large number of leaves and transformations that affect a small number of leaves.

In this paper, we study the quartet metric. For an evolutionary tree, the *quartet topology* of four species is the topological subtree induced by these species. In general, the possible quartet topologies of four species are the four shown in Fig. 1. Of these, the right-most cannot occur if we assume that all internal nodes have degree three. It is well-known that the complete set of quartet topologies is unique for a given tree and that the tree can be uniquely recovered from its set of quartet topologies in polynomial time [6]. If the tree has degree three, then, as observed in [11], it can be recovered from its set of quartet topologies in time $O(n \log n)$ using methods [4,9,11] for constructing an evolutionary tree in the experiment model in time $O(n \log n)$.

Given two evolutionary trees on the same set of n species, the *quartet distance* between them is the number of sets of four species for which the quartet topologies differ in the two trees. Since there are $\binom{n}{4}$ sets of four species, the quartet distance can be calculated in time $O(n^4)$ by examining the sets one by one. Steel and Penny in [14] present an algorithm for computing the quartet distance in time $O(n^3)$. Bryant *et al.* in [5] present an algorithm that computes the quartet distance in time $O(n^2)$. In this paper, we present an algorithm that computes the quartet distance in time $O(n \log^2 n)$, making it possible to compare much larger evolutionary trees. Our solution is based on two techniques: the smaller-half trick, also used by methods for finding tandem repeats in strings, see e.g. [15], and a data structure related to the data structure for dynamic expression trees [7].

The rest of the paper is organized as follows. In Sect. 2, we introduce quartets and present our algorithm for computing the quartet distance between two unrooted evolutionary trees. In Sect. 3, we describe a hierarchical decomposition of unrooted trees which is an essential part of the data structure used by our algorithm. In Sect. 4, we present the details of our data structure.

2 The Algorithm

As mentioned, we in this paper by an *evolutionary tree* mean an unrooted tree where all nodes are either leaves (i.e. have degree one) or have degree three, and where the leaves are uniquely labeled by the elements of a set S of species. Let n denote the size of S .

For an evolutionary tree T , the *quartet topology* of four species a, b, c , and d is the topological subtree of T induced by these species. In general, the possible quartet topologies for species a, b, c, d are the four shown in Fig. 1. Of these, the right-most does not occur in our setting, due to the assumption about all internal nodes having degree tree. Hence, the quartet topology is a pairing of the four species into two pairs, defined by letting a and b be a pair if among the three paths in T from a to respectively b, c , and d , the path to b is the first to separate from the others.

Given two evolutionary trees T_1 and T_2 on the same set S of species, the *quartet distance* between the two trees is the number of four-sets $\{a, b, c, d\} \subseteq S$, for which the quartet topologies in T_1 and T_2 differ. As there are $\binom{n}{4}$ different four-sets in S , the quartet distance can also be calculated as $\binom{n}{4}$ minus the number of four-sets for which the quartet topologies in T_1 and T_2 are identical. In this paper, we give an algorithm for finding this number in time $O(n \log^2 n)$.

To facilitate the counting of identical quartet topologies in the two trees, we view the quartet topology of a four-set $\{a, b, c, d\}$ as two *oriented* quartet topologies given by the two possible orientations of the “middle edge” of the topology. Figure 2 shows the two oriented quartet topologies arising from one unoriented quartet topology.

Clearly, the number of identical oriented quartet topologies between the trees T_1 and T_2 is twice the number of identical unoriented quartet topologies.

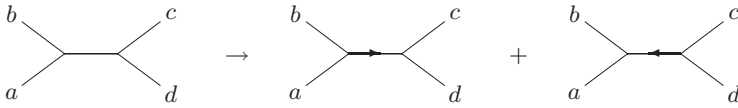


Fig. 2. The two orientations of a quartet topology

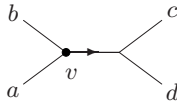


Fig. 3. A generic quartet

The goal of our algorithm is to count identical oriented quartet topologies. For brevity, we in the rest of this paper let the word *quartet* denote an oriented quartet topology of a four-set.

We associate quartets to internal nodes in T_1 as follows: Consider the generic quartet in Fig. 3, where the orientation is from the pair $\{a, b\}$ to the pair $\{c, d\}$. There is a unique node v in T_1 where the paths from a and b to c (and d) meet. We associate the quartet of Fig. 3 with the node v . This partitions the $2^{\binom{n}{4}}$ quartets into $n - 2$ disjoint sets, as there are $n - 2$ internal nodes in a tree of n leaves, when all internal nodes have degree three.

For an internal node v in T_1 , we by the subtrees *incident* to v mean the three subtrees which arise if v and its three incident edges are removed from T_1 . These are shown in Fig 4, denoted by A , B , and C . The number of quartets associated with v is given by the expression

$$\binom{|A|}{2} \cdot |B| \cdot |C| + \binom{|B|}{2} \cdot |C| \cdot |A| + \binom{|C|}{2} \cdot |A| \cdot |B|,$$

where $|T|$ denotes the number of leaves in subtree T .

The strategy of the algorithm is for each internal node v in T_1 to count how many of the quartets associated with v which also are quartets of T_2 . The sum over all nodes in T_1 of these counts then gives the required number of identical quartets in T_1 and T_2 .

To do this, the algorithm colors the elements of S using the three colors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. The coloring is maintained via the data structure described in Sect. 4. When v is an internal node in T_1 , we say that the elements of S are colored *according* to v if the labels of the leaves of one of the three subtrees incident to v all have color $\mathcal{A}$, the labels of the leaves of another of the subtrees all have color $\mathcal{B}$, and the labels of the leaves of the remaining subtree all have color $\mathcal{C}$.

The central feature of the data structure is that if the elements of S are colored according to a node v in T_1 , then it can return in constant time the number of quartets associated with v which also are quartets in T_2 . The data

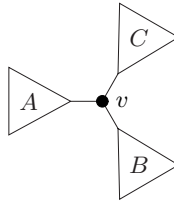


Fig. 4. Subtrees incident to an internal node v

structure also allows the color of an element to be changed in time $O(\log n)$, given a pointer to the element.

The algorithm starts by rooting T_1 at an arbitrary leaf. It then calculates the size $|v|$ of each node v in T_1 during a postorder traversal starting at the root, where $|v|$ denotes the number of leaves below v , and stores this information in the nodes. It also colors all elements of S by the color $\mathcal{C}$.

The algorithm then calculates the desired sum of the counts for all internal nodes of T_1 in a recursive fashion, starting at the single child of the root of T_1 . To achieve the claimed complexity, the algorithm at a node v will recurse first on its smaller child, then on its larger child, and finally add the count for v to the sum calculated so far.

In Fig. 5, the algorithm is described in pseudo-code as a recursive procedure $\text{Count}(v)$. A call to $\text{Count}(v)$ returns the sum of the counts for the internal nodes of T_1 which are below v . Initially, it is called with v set to the single child of the root of T_1 . The two routines $\text{Small}(v)$ and $\text{Large}(v)$ return the child of v having smallest, respectively largest, size. The routine $\text{NodeCount}(v)$ is a call to the data structure of Sect. 4, returning the count for the node v . The routine $\text{ColorLeaves}(v, \mathcal{X})$ colors by the color $\mathcal{X}$ all elements in the data structure which are labels of leaves below v in T_1 . This is done by a traversal of the subtree in T_1 rooted at v . By maintaining bi-directional pointers between elements of S in the data structure and the leaves in T_1 and T_2 which they label, this takes time $O(|v| \cdot \log n)$.

Theorem 1. *Let T_1 and T_2 be two unrooted evolutionary trees on the same set S of species, and let all internal nodes in the trees have degree three. Then the quartet distance between T_1 and T_2 can be found in time $O(n \log^2 n)$.*

Proof. We here assume the existence of the data structure discussed above. This existence is proven in Sect. 4. By induction on the number of calls to $\text{Count}(v)$, it follows that the algorithm above maintains the invariants:

1. At the *beginning* of the execution of an instance of $\text{Count}(v)$, all elements in S are colored by the color $\mathcal{C}$.
2. At the *end* of the execution of an instance of $\text{Count}(v)$, all elements in S which are labels of leaves *below* v in T_1 are colored by the color $\mathcal{A}$, and all other elements in S are colored by the color $\mathcal{C}$.

```

Procedure Count( $v$ )
  if  $v$  is a leaf then
    color  $v$  by the color  $\mathcal{A}$ 
    return 0
  else
     $x = \text{Count}(\text{Small}(v))$ 
    ColorLeaves( $\text{Small}(v)$ ,  $\mathcal{C}$ )
     $y = \text{Count}(\text{Large}(v))$ 
    ColorLeaves( $\text{Small}(v)$ ,  $\mathcal{B}$ )
     $z = \text{NodeCount}(v)$ 
    ColorLeaves( $\text{Small}(v)$ ,  $\mathcal{A}$ )
    return  $x + y + z$ 

```

Fig. 5. The algorithm

The invariants imply that when a call to $\text{NodeCount}(v)$ takes place, labels of leaves in the subtree of $\text{Small}(v)$ are labeled by the color $\mathcal{B}$, labels of leaves in the subtree of $\text{Large}(v)$ are labeled by the color $\mathcal{A}$, and the remaining elements are labeled by the color $\mathcal{C}$. In other words, the elements of S are colored according to v . Correctness of the algorithm follows.

For complexity, note that the work incurred by an instance of $\text{Count}(v)$, not counting recursive calls made during this instance, is $O(|\text{Small}(v)| \cdot \log n)$. Let this work be accounted for by charging each leaf below $\text{Small}(v)$ in T_1 (or v itself, if it is a leaf) an amount $O(\log n)$ of work. For a given leaf, this charging can only happen at nodes v on the path from the leaf to the root where the path goes from $\text{Small}(v)$ to v . As the size of v is at least twice as large as the size of $\text{Small}(v)$, this can only happen $\log n$ times. Hence, each leaf is at most charged $O(\log^2 n)$ work in total, and the result follows. $\square$

3 Hierarchical Decomposition

An essential part of the data structure in Sect. 4 is a *hierarchical decomposition* of the evolutionary tree T_2 . Given an unrooted tree T where all nodes have degree at most three, we in the following describe how to obtain a hierarchical decomposition of T with logarithmic height. Our decomposition is very similar to the decompositions used for solving the parallel and dynamic expression tree evaluation problems [3,7], but in our setting the underlying tree is considered to be unrooted.

We base our hierarchical decomposition on the notion of *components*. We define a component C in T to be one of the following:

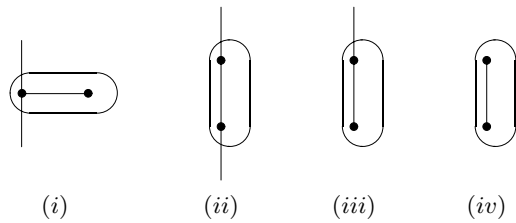


Fig. 6. The four possible types of compositions of components

1. A set consisting of a single node of T .
2. A connected subset of the nodes of T , such that at most two nodes in C are connected by an edge to nodes in $T \setminus C$.

In other words, a component is either a set consisting of a single node, or a connected subset of nodes such that the cut defined by the subset is of size at most two. The *external edges* of a component C of T are the edges in T connecting nodes in C and $T \setminus C$. The *degree* of a component is the number of external edges of the component. By the second condition above, a component with two or more nodes can have degree at most two.

Each node of T (including leaves) constitutes a component of type 1. Components of type 2 are formed as the union of two adjacent components C' and C'' , where C' and C'' are said to be adjacent if there exist an edge (u, v) in T such that $u \in C'$ and $v \in C''$. We call such a union a *composition*. We only allow the four compositions depicted in Fig. 6. Nodes represent contracted components and ovals represent component compositions. Types (i), (iii), and (iv) are the cases where a component with degree one is composed with a component of degree three, two, and one respectively. Type (ii) is the case where two components with degree two are composed into a new component with degree two. Note that each composition of two components corresponds to a unique edge in the tree T , namely the edge connecting the two components.

A hierarchical decomposition of an unrooted tree T is a rooted binary tree, in the following denoted $H(T)$. Each node of $H(T)$ represents a component in T . Leaves of $H(T)$ represent components of type 1, and there is a one-to-one mapping between these components and the leaves of $H(T)$. An internal node v of $H(T)$ represent a component of type 2 formed by the composition of the two components represented by the children of v .

Lemma 1. *For every unrooted tree with n nodes and all nodes having degree at most three, there exists a hierarchical decomposition tree with height $O(\log n)$. The decomposition can be computed in time $O(n)$.*

Proof. Given a tree with n nodes, we construct a hierarchical decomposition bottom-up in $O(\log n)$ steps. Initially we start with each node being a component

by itself. In each step we greedily select an arbitrary maximal set of independent compositions, using time linear in the number of remaining components.

Let n denote the number of components at the beginning of a step. A composition of type (iv) will occur if and only if $n = 2$. If $n \geq 3$, let n_1 , n_2 , and n_3 denote the number of components of degree one, two and three respectively. We have $n = n_1 + n_2 + n_3$ and $n_3 = n_1 - 2$. Since $n \geq 3$, there are n_1 possible compositions of types (i) and (iii). We observe that the only edges not corresponding to legal compositions are edges connecting a component of degree three with a component of degree two or three. Since there are at most $3n_3$ such edges, the number of possible compositions is at least $n - 1 - 3n_3 = n - 3n_1 + 5$. If $n_1 < n/4$, then this bound is at least $n/4$. It follows that there are always at least $n/4$ possible compositions. Since each possible composition can conflict with at most two other compositions, any maximal set of non-conflicting compositions has size at least $n/12$.

After k steps, at most $n(11/12)^k$ components will remain. In particular, at most one component will remain after at most $\lceil \log_{12/11} n \rceil$ steps, so the height of the hierarchical decomposition tree is bounded by $\lceil \log_{12/11} n \rceil$. Since the number of components decreases geometrically for each step, the total time becomes $O(n)$. $\square$

4 Counting Quartets in Components

Given a coloring of the elements in S with the colors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, and given a quartet oriented as in Fig. 3 from the pair $\{a, b\}$ to the pair $\{c, d\}$, we say that the quartet is *compatible* with the coloring if a and b have different colors, and c and d both have the remaining color. Let T be an evolutionary tree for S , and let $H(T)$ be the hierarchical decomposition tree for T , as defined in Sect. 3.

Lemma 2. *When S is colored according to a choice of v in T , then the set of quartets compatible with the coloring is exactly the quartets associated with v .*

Proof. Follows from the definitions of quartets being compatible with a coloring and quartets being associated with a node. $\square$

We now describe how to decorate the nodes of $H(T)$ with information such that the number of quartets of T which are compatible with a given coloring of S can be returned in constant time. Furthermore, for a given coloring, the information can be generated in $O(n)$ time, and if one element of S changes color, the information can be updated in time $O(\log n)$.

For each node of $H(T)$, we store a tuple (a, b, c) of integers and a function F . Recall that a node in $H(T)$ represents a component in T . The integers a , b , and c of the tuple are the number of elements at the leaves contained in this component which are colored $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively. A component has k external edges for k between zero and three (the case of zero external edges occurs only at the root of $H(T)$). The function F has three variables for each of the external edges of the component. For a component with at least one external edge, we number

these edges arbitrarily from 1 to k and denote the three variables corresponding to edge i by $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{c}_i$. If an external edge were removed from T , two subtrees of T would arise, of which one does not contain the component in question. We call this subtree the subtree *induced* by the external edge. The variables $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{c}_i$ denote the number of elements in leaves from the subtree induced by edge i which are colored $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, respectively. Finally, F states, as a function of the variables $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{c}_i$ for $1 \leq i \leq k$, the number of the quartets which are both associated (in the sense defined in Sect. 2) with nodes in the component *and* are compatible with the given coloring. It will turn out that F is actually a polynomial of total degree at most four.

The root of $H(T)$ represents a component which comprises the entire tree T , i.e. the component has no external nodes, so the function F stored there is actually a constant. Hence, the number of quartets of T which are compatible with a given coloring of S is part of the information stored at the root.

Lemma 3. *The tree $H(T)$ can be decorated with the information described above in time $O(n)$.*

Proof. The information is computed in a bottom up fashion during a traversal of $H(T)$. We first describe how the information for leaves in $H(T)$ is generated, i.e. for nodes representing single node components. Recall that a node in T is either a leaf and has degree one, or is an internal node and has degree three.

For a component consisting of a single leaf with an element colored $\mathcal{A}$, $\mathcal{B}$, or $\mathcal{C}$, the tuple is $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$, respectively. The function F is identically zero, as quartets are only associated with internal nodes of T , not with leaves of T .

For a component consisting of a single degree three node u , the tuple is $(0, 0, 0)$, as no leaves of T are contained in the component. The function F should count the number of quartets which are both compatible with the coloring and associated with u in T . A quartet oriented from the pair $\{a, b\}$ to the pair $\{c, d\}$ fulfills this requirement precisely when c and d are contained in one of the three subtrees induced by the external edges of the component, and they have the same color, and a and b each are in one of the remaining two induced subtrees and each have one of the remaining two colors. For the case that c and d are in the subtree induced by edge number one and have color $\mathcal{A}$, the number of quartets fulfilling this is

$$\binom{\mathbf{a}_1}{2} \cdot (\mathbf{b}_2 \mathbf{c}_3 + \mathbf{b}_3 \mathbf{c}_2).$$

Summing over all $3 \cdot 3 = 9$ choices of the induced subtree and color for c and d , we get:

$$\begin{aligned}
& F(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1, \mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2, \mathbf{a}_3, \mathbf{b}_3, \mathbf{c}_3) \\
&= \binom{\mathbf{a}_1}{2} \cdot (\mathbf{b}_2 \mathbf{c}_3 + \mathbf{b}_3 \mathbf{c}_2) + \binom{\mathbf{a}_2}{2} \cdot (\mathbf{b}_1 \mathbf{c}_3 + \mathbf{b}_3 \mathbf{c}_1) + \binom{\mathbf{a}_3}{2} \cdot (\mathbf{b}_2 \mathbf{c}_1 + \mathbf{b}_1 \mathbf{c}_2) \\
&+ \binom{\mathbf{b}_1}{2} \cdot (\mathbf{a}_2 \mathbf{c}_3 + \mathbf{a}_3 \mathbf{c}_2) + \binom{\mathbf{b}_2}{2} \cdot (\mathbf{a}_1 \mathbf{c}_3 + \mathbf{a}_3 \mathbf{c}_1) + \binom{\mathbf{b}_3}{2} \cdot (\mathbf{a}_2 \mathbf{c}_1 + \mathbf{a}_1 \mathbf{c}_2) \\
&+ \binom{\mathbf{c}_1}{2} \cdot (\mathbf{b}_2 \mathbf{a}_3 + \mathbf{b}_3 \mathbf{a}_2) + \binom{\mathbf{c}_2}{2} \cdot (\mathbf{b}_1 \mathbf{a}_3 + \mathbf{b}_3 \mathbf{a}_1) + \binom{\mathbf{c}_3}{2} \cdot (\mathbf{b}_2 \mathbf{a}_1 + \mathbf{b}_1 \mathbf{a}_2)
\end{aligned}$$

We now turn to the generation of the information stored in the internal nodes of $H(T)$. Consider the composition of two components C' and C'' . Let (a', b', c') and F' , and (a'', b'', c'') and F'' be the information stored at the nodes representing the components C' and C'' . The information stored at the node representing the composition C of C' and C'' is $(a' + a'', b' + b'', c' + c'')$ and F , where F depends on the type of composition. If the component composition is of type (ii), we consider the case where the numbering of external edges of components is such that the first external edge of C' and C'' is the edge connecting C' and C'' , and the second external edge of C' is the first external edge of C , and the second external edge of C'' is the second external edge of C . The remaining cases of numbering of external edges are obtained by appropriate changes of the arguments to F' and F'' .

$$\begin{aligned}
& F(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1, \mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2) \\
&= F'(\mathbf{a}_2 + \mathbf{a}'', \mathbf{b}_2 + \mathbf{b}'', \mathbf{c}_2 + \mathbf{c}'', \mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) \\
&+ F''(\mathbf{a}_1 + \mathbf{a}', \mathbf{b}_1 + \mathbf{b}', \mathbf{c}_1 + \mathbf{c}', \mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)
\end{aligned}$$

Component compositions of type (iii) and (iv) are identical to type (ii), except that the definition of F is simpler. For type (iii) we have (assuming that C'' is the component of degree one)

$$F(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) = F'(\mathbf{a}'', \mathbf{b}'', \mathbf{c}'', \mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1) + F''(\mathbf{a}_1 + \mathbf{a}', \mathbf{b}_1 + \mathbf{b}', \mathbf{c}_1 + \mathbf{c}') ,$$

and for type (iv) we have

$$F = F'(\mathbf{a}'', \mathbf{b}'', \mathbf{c}'') + F''(\mathbf{a}', \mathbf{b}', \mathbf{c}') .$$

Note that for type (iv) compositions, F is a constant.

Finally, we for type (i) compositions get the following expression for F , assuming C' has degree one and the first and second external edges of C are the second and third external edges of C'' , respectively.

$$\begin{aligned}
& F(\mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1, \mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2) \\
&= F'(\mathbf{a}_1 + \mathbf{a}_2 + \mathbf{a}'', \mathbf{b}_1 + \mathbf{b}_2 + \mathbf{b}'', \mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}'') \\
&+ F''(\mathbf{a}', \mathbf{b}', \mathbf{c}', \mathbf{a}_1, \mathbf{b}_1, \mathbf{c}_1, \mathbf{a}_2, \mathbf{b}_2, \mathbf{c}_2)
\end{aligned}$$

By structural induction on the definition of the F functions stored at components, it follows that F is always a polynomial of total degree at most four.

Polynomials with total degree at most four and at most nine variables can be stored in constant space by storing the coefficients of the polynomials, and they can be manipulated in constant time, e.g. when adding or composing two polynomials. We conclude that for a component C which is the composition of two components C' and C'' , the information to be stored at C can be computed in constant time, provided that the information stored at C' and C'' is known. It follows that $H(T)$ can be decorated in time $O(n)$. $\square$

Lemma 4. *The decoration of $H(T)$ can be updated in $O(\log n)$ time when the color of an element in S changes.*

Proof. From the proof of Lemma 3 we know that the decoration of a node in $H(T)$ only depends on the decoration of the children of the node in $H(T)$, i.e. the only decorations that need to be updated in $H(T)$ while changing the color of an element in S are the ancestors of the leaf in $H(T)$ corresponding to the element. Since $H(T)$ has height $O(\log n)$ and the decoration of a node takes constant time to compute knowing the decoration of the children, it follows that the decoration of $H(T)$ can be updated in time $O(\log n)$. $\square$

Lemma 5. *When S is colored according to a choice of v in T_1 , then the set of quartets compatible with the coloring is exactly the quartets associated with v .*

Proof. Follows from the definitions of the colors and compatible quartets. $\square$

Corollary 1. *If the above construction is done with T_2 for T , and the coloring of S is according to a choice of v in T_1 , then the quartets in T_2 compatible with the coloring are exactly the quartets which are in both T_1 and T_2 . Furthermore, the number of such quartets is exactly the value of the constant function F stored at the root of $H(T_2)$.*

References

1. B. L. Allen and M. Steel. Subtree transfer operations and their induced metrics on evolutionary trees. *Annals of Combinatorics*, 5:1–13, 2001. 732
2. A. Amir and D. Keselman. Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1997. 732
3. R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, Apr. 1974. 736
4. G. S. Brodal, R. Fagerberg, C. N. S. Pedersen, and A. Östlin. The complexity of constructing evolutionary trees using experiments. In *Proc. 28th International Colloquium on Automata, Languages, and Programming*, volume 2076 of *Lecture Notes in Computer Science*, pages 140–151. Springer-Verlag, 2001. 732
5. D. Bryant, J. Tsang, P. E. Kearney, and M. Li. Computing the quartet distance between evolutionary trees. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 285–286, N. Y., Jan. 9–11 2000. ACM Press. 732, 733

6. P. Buneman. The recovery of trees from measures of dissimilarity. *Mathematics in Archeological and Historical Sciences*, pages 387–395, 1971. 732
7. R. F. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13(3):245–265, 1995. 733, 736
8. G. Estabrook, F. McMorris, and C. Meacham. Comparison of undirected phylogenetic trees based on subtrees of four evolutionary units. *Syst. Zool.*, 34(2):193–200, 1985. 732
9. M. Farach, S. Kannan, and T. J. Warnow. A robust model for finding optimal evolutionary trees. *Algorithmica*, 13(1/2):155–179, 1995. 732
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. 731
11. A. Lingas, H. Olsson, and A. Östlin. Efficient merging, construction, and maintenance of evolutionary trees. In *Proc. 26th Int. Colloquium on Automata, Languages and Programming*, volume 1644 of *Lecture Notes in Computer Science*, pages 544–553. Springer-Verlag, 1999. 732
12. D. F. Robinson and L. R. Foulds. Comparison of weighted labelled trees. In *Combinatorial mathematics, VI (Proc. Sixth Austral. Conf., Univ. New England, Armidale, 1978)*, *Lecture Notes in Mathematics*, pages 119–126. Springer, Berlin, 1979. 732
13. D. F. Robinson and L. R. Foulds. Comparison of phylogenetic trees. *Math. Biosci.*, 53(1-2):131–147, 1981. 732
14. M. Steel and D. Penny. Distribution of tree comparison metrics—some new results syst. *Syst. Biol.*, 42(2):126–141, 1993. 733
15. J. Stoye and D. Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Computer Science*, pages 140–152. Springer-Verlag, 1998. 733
16. M. S. Waterman and T. F. Smith. On the similarity of dendrograms. *Journal of Theoretical Biology*, 73:789–800, 1978. 732

The *Cent-dian* Path Problem on Tree Networks

Ronald I. Becker¹, Yen-I Chiang¹, Isabella Lari², and Andrea Scozzari²

¹ Department of Mathematics, University of Cape Town
Rondebosh 7700, South Africa
`{rib,yeni}@maths.uct.ac.za`

² Dipartimento di Statistica, Probabilità e Statistiche Applicate, Università di Roma
“La Sapienza”

P.le A. Moro 5, 00185 Roma, Italy
`isabella.lari@uniroma1.it`
`scozzar@rosd.sta.uniroma1.it`

Abstract. In a network, the **DISTSUM** of a path is the sum of the distances of all vertices to the path, and the **ECCENTRICITY** is the maximum distance of any vertex to the path. The *Cent-dian* problem is the constrained optimization problem which seeks to locate on a network a path which has minimal value of the **DISTSUM** over all paths whose **ECCENTRICITY** is bounded by a fixed constant. We consider this problem for trees, and we also consider the problem where an additional constraint is required, namely that the optimal path has **LENGTH** bounded by a fixed constant. The first problem has already been considered in the literature. We give another linear time algorithm for this problem which is considerably simpler than the previous one. The second problem does not seem to have been considered elsewhere, and we give an $O(n \log^2 n)$ divide-and-conquer algorithm for its solution.

Keywords: facility location, median path, centre path

1 Introduction

Network facility location is concerned with the optimal selection of a site or of a set of sites in a network. Several authors extended the theory to include sites that are not merely single points but paths or trees. The objective is either minimizing the distance from the furthest vertex of the network to the facility or of minimizing the sum of the distances from the vertices to the selected facility. Some authors have also questioned the pertinence of the median and the centre criterion for these problems. Indeed, using the median tends to favor clients who are clustered in population centres to the detriment of clients who are spatially dispersed [9]. However, the centre criterion may yield a significant increase in the total distance. This had led Halpern [5,?] to model the corresponding trade off as a bicriterion problem in which a combination of total distance and maximal distance is minimized. It is also important in practical situations to consider the cost of the facility to be located. In case of a path-shaped facility this can be done by introducing a bound on the cost (or length) of the path [3,11]. Let P be

a path, let $d(P)$ be the sum of the distances of the vertices in the network to P , let $E(P)$ be the maximum distance of a vertex to P , and let $L(P)$ be the length of P (these quantities are defined precisely below). In this paper we consider the following two problems on tree networks:

Problem 1: Find a path P^* which minimizes $d(P)$ over all paths P satisfying $E(P) \leq R$.

Problem 2: Find a path P^* which minimizes $d(P)$ over all paths P satisfying $E(P) \leq R$ and $L(P) \leq \ell$.

We call a path which is a solution to Problem 1 a *Cent-dian* path of the tree [5]. We call a path which is a solution to Problem 2 a *Bounded Cent-dian* path. The first problem has already been considered in the literature in [1], where the authors give an $O(n)$ time algorithm for solving it (n is the number of vertices in the tree). They start by considering the problem as a bicriterion path problem and propose a procedure that finds a superset M which contains the set of Pareto-Optimal paths, and having cardinality at most n . In this paper we propose a procedure that solves this problem also in linear time, by visiting *bottom up* and *top down* the tree rooted at some vertex. This enables the rapid computation at a given vertex of all the quantities that are needed for solving the problem (see [2,8,13]) and avoids finding the set M . For the second problem we use another type of recursion. A “central” vertex in the tree is computed; then a path P^* of minimum $d(P)$ through this vertex, which has $E(P) \leq R$ and $L(P) \leq \ell$, is found. If P^* is not the path that minimizes the sum of the distances, then the best path must lie entirely in one of the subtrees rooted at the adjacent vertices of the “central” vertex. The algorithm is recursively applied to these subtrees. An appropriate choice of the “central” vertex ensures that the depth of the recursion is $O(\log n)$. This type of recursion was used by Peng and Lo [11] for solving another location problem.

In Section 2, we provide notation and definitions, as well as an account of a preprocessing phase which calculates several quantities needed in the algorithm. Section 3 gives an algorithm for Problem 1 with time complexity $O(n)$, and Section 4 gives an algorithm for Problem 2 with time complexity $O(n \log^2 n)$. At the end of the paper there is an Appendix containing a numerical example for an instance of Problem 1.

2 Notation and Definitions

Given a tree $T = (V, E)$, with $|V| = n$, let $a(e)$ be a positive weight (length) associated with each edge $e = (v, w) \in E$. Suppose also that to each vertex v is assigned a nonnegative weight $h(v)$. Let P be a path in T . The *length* of P is $L(P) = \sum_{e \in P} a(e)$. Given two vertices v and u , we denote the unique path from v to u as P_{vu} . We define the distance $d(v, u)$ between two vertices v and u of V as the length of P_{vu} . A vertex v is a *leaf* if the number of edges incident

with v is 1. We denote by $p(v)$ the parent of v in T with respect to its current root. Given a path P in T , the sum of the distances from P to all the vertices $v \in V$ is $d(P) = \sum_{v \in V} h(v)d(v, P)$, where $d(v, P)$ is the minimum distance from $v \in V$ to a vertex of P (see [10]). We call $d(P)$ the DISTSUM of P . If $P = \{v\}$ then we write $d(v)$ instead of $d(\{v\})$. A path P which minimizes DISTSUM in T is called a median path (see [10,12]). We define $E(P)$ the ECCENTRICITY of P where $E(P) = \max_{v \in V} \{d(v, P)\}$. The shortest path P which minimizes ECCENTRICITY is the path centre of T (see [12]).

Definition 1. A path P is said to be R -feasible if $E(P) \leq R$. It is said to be ℓ -feasible if $L(P) \leq \ell$. It is said to be ℓ -R-feasible if it is both ℓ -feasible and R-feasible.

For both the problems presented in this paper, we need a preprocessing phase that allows us to compute the quantities that will be used in the two algorithms. In the following we give an overview of the recursive formulas calculated in this preprocessing phase.

Let T be rooted at some vertex z and call it T^z . (For the first algorithm the root of the tree will not be given explicitly in the notation, but will always be understood from the context). Denote by T_v^B the subtree rooted at vertex v and by T_v^U the subtree generated by vertices $(V - T_v^B) \cup \{v\}$. We denote by $deg_B(v)$ the degree of vertex v in T_v^B . Let $d_B(v)$ be the sum of the distances of the vertices in T_v^B to vertex v . By using the standard *bottom up* approach, and proceeding from the leaves to the root, we compute the sum of the weights of the vertices in T_v^B , say $sum_B(v)$ (in an unweighted tree this would be the cardinality of T_v^B) and $d_B(v)$ as follows (see [2,3,8,13]).

$$\begin{cases} sum_B(v) = h(v) & \text{if } v \text{ is a leaf of } T^z \\ sum_B(v) = h(v) + \sum_{w \text{ a son of } v} sum_B(w). \end{cases} \quad (1)$$

$$\begin{cases} d_B(v) = 0 & \text{if } v \text{ is a leaf of } T^z \\ d_B(v) = \sum_{w \text{ a son of } v} [d_B(w) + sum_B(w)a(v, w)]. \end{cases} \quad (2)$$

The time needed for the computation of (1) and (2) is $O(n)$. However, in the following algorithm we need to calculate $\forall v \in V$, $d(v)$, that is, the sum of the distances of all the vertices in T to v . Then, $d(z) = d_B(z)$ and by proceeding from the root z to the leaves we have:

$$d(v) = d(p(v)) + a(v, p(v))[H - 2sum_B(v)] \quad (3)$$

where $H = \sum_{v \in V} h(v)$ (see also [8]). Given a vertex v we denote by $E_B(v)$ and $E_U(v)$ the *eccentricity* of v in T_v^B and in T_v^U respectively. We proceed bottom up and top down to compute these quantities efficiently at each vertex of T . In particular, in a bottom up visit of the tree we associate to the vertices three values of eccentricity $E_B^1(v)$, $E_B^2(v)$ and $E_B^3(v)$ that represent the maximum, the second maximum and the third maximum eccentricity of v in T_v^B respectively.

$$E_B^1(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max_{w \text{ a son of } v} \{E_B^1(w) + a(v, w)\} & \text{otherwise.} \end{cases} \quad (4)$$

$$E_B^2(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf or } \deg_B(v) < 2 \\ \max_w \text{ a son of } v \{E_B^1(w) + a(v, w) | w \neq u_B^1(v)\} & \text{otherwise.} \end{cases} \quad (5)$$

where $u_B^1(v)$ is a son of v that gives the value of $E_B^1(v)$.

$$E_B^3(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf or } \deg_B(v) < 3 \\ \max_w \text{ a son of } v \{E_B^1(w) + a(v, w) | w \neq u_B^1(v) \text{ and } w \neq u_B^2(v)\} & \text{otherwise.} \end{cases} \quad (6)$$

where $u_B^2(v)$ is a son of v that gives $E_B^2(v)$. The above three formulas can be computed in $O(n)$ time for all the vertices of T . Let us consider the eccentricity of a vertex v in T_v^U , then by proceeding from the root toward its leaves we distinguish:

if $v = z$ then $E_U(v) = 0$

if vertex v is adjacent to the root z :

$$E_U(v) = \begin{cases} E_B^2(z) + a(z, v) & \text{if } v = u_B^1(z) \\ E_B^1(z) + a(z, v) & \text{otherwise.} \end{cases} \quad (7)$$

if vertex v is not adjacent to z :

$$E_U(v) = \begin{cases} \max\{(E_U(p(v)) + a(p(v), v)), (E_B^1(p(v)) + a(p(v), v))\} & \text{if } v \neq u_B^1(p(v)) \\ \max\{(E_U(p(v)) + a(p(v), v)), (E_B^2(p(v)) + a(p(v), v))\} & \text{if } v = u_B^1(p(v)) \end{cases} \quad (8)$$

In a top down visit of the tree the above formulas can be computed in $O(n)$ time. We now introduce a definition relevant to the formulation of the algorithms of the following sections.

Definition 2. Given a path P_{vu} and a path P_{uw} with edges disjoint from P_{vu} , the *distance saving* of P_{uw} with respect to P_{vu} , is the reduction of DISTSUM obtained by adding P_{uw} to P_{vu} (see [10]), that is:

$$sav(P_{vu}, P_{uw}) = d(P_{vu}) - d(P_{vw}) \quad (9)$$

If the first path consists of only one vertex v , we simply write $sav(v, P_{vw})$.

Referring to Definition 2, we will now define two quantities $sav_B^1(v)$ and $sav_B^2(v)$ whose meaning and use will be described in the following lemma.

$$\begin{cases} sav_B^1(v) = 0 & \text{if } v \text{ is a leaf} \\ sav_B^1(v) = \max_{u \text{ a son of } v} \{sav_B^1(u) + sum_B(u)a(v, u)\} & \text{if } E_B^1(v) \leq R \\ sav_B^1(v) = sav_B^1(u_B^1(v)) + sum_B(u_B^1(v))a(v, u_B^1(v)) & \text{if } E_B^1(v) > R \end{cases} \quad (10)$$

$$\begin{cases} sav_B^2(v) = 0 & \text{if } v \text{ is a leaf or } \deg_B(v) < 2 \\ sav_B^2(v) = \max_{u \neq u_B^1(v) \text{ a son of } v} \{sav_B^1(u) + sum_B(u)a(v, u)\} & \text{if } E_B^2(v) \leq R \\ sav_B^2(v) = sav_B^1(u_B^2(v)) + sum_B(u_B^2(v))a(v, u_B^2(v)) & \text{if } E_B^2(v) > R \end{cases} \quad (11)$$

where we define:

$$\begin{cases} w_B^1(v) = \text{a son of } v \text{ that gives } sav_B^1(v) \\ w_B^2(v) = \text{a son of } v \text{ that gives } sav_B^2(v) \end{cases} \quad (12)$$

(10) and (11) can be computed in $O(n)$ time for all v with a bottom up scan of T .

Lemma 1. Let $deg_B(v) \geq 2$. If there is an R-feasible path passing through v in T_v^B , then $sav_B^1(v) + sav_B^2(v)$ is the maximum value that can be saved from $d(v)$ by putting an R-feasible path through v into T_v^B .

Proof. Note that if there is an R-feasible path through v in T_v^B , then for all $x \in T_v^B \setminus \{v\}$:

1. $E_B^2(x) \leq R$
2. $sav_B^1(x)$ is either the maximum distance saving from x to a leaf if $E_B^1(x) \leq R$ or since 1. holds, it is the maximum distance saving of an R-feasible path from x to a leaf that must pass through the edge $(x, u_B^1(x))$

Suppose there is an R-feasible path through v in T_v^B , there are 3 possibilities at v : (i) $E_B^1(v) \leq R$; (ii) $E_B^2(v) > R$ and $E_B^3(v) \leq R$; (iii) $E_B^1(v) > R$ and $E_B^2(v) \leq R$ (in all cases, we must have $E_B^3(v) \leq R$). In case (i) all vertices $x \in T_v^B$ have $E_B^1(x) \leq R$. Then any path from v into T_v^B extends to a feasible path, so, the second lines of equations 10 and 11 are used, and the Lemma holds. In case (ii) the only possible feasible path through v in T_v^B must pass through $u_B^1(v)$ and $u_B^2(v)$. Then the third lines of equations 10 and 11 are used and referring to 1. and 2. the Lemma holds. In case (iii) there must be a path containing $(v, u_B^1(v))$, but any other path through v can be used to complete it, so, the third line of 10 and the second line of 11 are used and by 1. and 2. the Lemma holds. $\square$

Corollary. If there is an R-feasible path passing through v in T_v^B , then there is a feasible path P such that

$$d(P) = d(v) - (sav_B^1(v) + sav_B^2(v))$$

and this value is the minimal value of $d(P)$ over all such paths.

Proof. Follows from Lemma 1. $\square$

Define $d'(v)$ for all $v \in V$ by $d'(v) = d(v) - (sav_B^1(v) + sav_B^2(v))$. See the above corollary for the meaning of $d'(v)$.

Remark 1. For the algorithms that follow, we will need to know for each $v \in V$ the values $E_B^1(v), E_B^2(v), E_B^3(v), E_U(v), d(v), d'(v), sav_B^1(v), sav_B^2(v), w_B^1(v), w_B^2(v)$.

3 The Algorithm for the Cent-dian Path Problem

We recall that Problem 1 consists in finding a path P which minimizes $d(P)$ over all *R-feasible* paths P . The idea of the algorithm is to hang the tree from

an arbitrary root and then proceeding bottom up, to calculate for each vertex v the R-feasible path P_v in T_v^B through v which minimizes $d(P)$ over all R-feasible paths P in T_v^B . We then find the path with minimum $d(P_v)$ over all $v \in V$. (In practice, we find the path with minimum $d(P_v)$ over all vertices v considered up to the present time, and we stop when it is clear that no further vertices w can have a *R-feasible* path in T_w^B). It is clear that for any R-feasible path P , there is a v such that P lies in T_v^B and passes through v . Hence, the procedure outlined above will successfully find an optimal solution to Problem 1.

ALGORITHM Cent-dian:

Input: a weighted tree T^z rooted at some vertex z

Output: a *Cent-dian* path P^* of T and its DISTSUM d^*

begin

$d^* := +\infty$

$P^* := \emptyset$

Proceed bottom up level by level until the root is reached

for each vertex v visited **do**

if $E_B^3(v) > R$ **or** $(E_U(v) > R$ **and** $E_B^2(v) > R)$ **then Stop**

/* There is no feasible solution*/

else if $E_U(v) > R$ **then** process the next vertex

else

if $d'(v) < d^*$ **then** $d^* := d(P)$ and $P^* := P$

if $E_B^2(v) > R$ **Stop** visiting the tree

Using ALGORITHM FINDPATH starting at v^* , find P^* .

P^* is the *Cent-dian* path of T and d^* is the optimal value of $d(P)$

end

ALGORITHM FINDPATH

Input: v with $\deg_B(v) \geq 2$ and such that there is an R-feasible path through v in T_v^B .

Output: An R-feasible path P^* through v in T_v^B with minimal value of $d(P)$ over all such paths P .

Find paths P_1, P_2 as follows: /*possibly $P_2 = \phi^*$ */

The first edge of P_1 is $(v, w_B^1(v))$, and the k^{th} edge is $((w_B^1)^{(k-1)}(v), (w_B^1)^{(k)}(v))$
 $k > 1$ and $(w_B^1)^{(1)}(v) = w_B^1(v)$, $k = 1$, where $(w_B^1)^{(k)}$, $k > 1$ is the k -fold composition of w_B^1 and we add edges until a leaf is reached.

The first edge of P_2 is $(v, w_B^2(v))$, and the k^{th} edge is $((w_B^1)^{(k-1)}(v), (w_B^1)^{(k)}(v))$
 $k > 1$ and $(w_B^1)^{(1)}(v) = w_B^2(v)$, $k = 1$ and we add edges until a leaf is reached.

$P := P_1 \cup P_2$

Notice that if $\deg_B(v) = 1$, then the best path in T_v^B passing through v has only one leaf as endpoint. If $v = z$ this could be an optimal path.

Theorem 1. Algorithm *Cent-dian* finds an optimal solution for Problem 1, if one exists, and stops in failure if no optimal solution exists.

Proof. Suppose that the algorithm stops at the first **STOP**. If $E_B^3(v) > R$, then three paths incident from v have eccentricity greater than R . A path P through v can only reduce the eccentricity in two of those paths. Hence for any such path $E(P) > R$, and there is no feasible solution. If $E_U(v) > R$ and $E_B^2(v) > R$, similar reasoning shows that there is no feasible solution. If vertex v is being processed and $E_U(v) > R$, then any path P in T_v^B has $E(P) > R$ and so there is no feasible path through v that extends in T_v^B . If $E_B^2(v) > R$, then no vertex on the same level or above v can have a feasible solution through it, and we can quit our visit after processing v (at the second **Stop**) and calculate P^* . If a vertex v gets its value of $d'(v)$ compared by the algorithm to d^* , then we must have $E_B^3(v) \leq R$ and $E_U(v) \leq R$. We show that there is an R-feasible path in T_v^B passing through v . Since $E_B^3(v) \leq R$, if there were no such R-feasible path, there would be a vertex $w \neq v$ in T_v^B such that $E_B^2(w) > R$. But this is impossible, since by the bottom up procedure, we would already have visited w and we would have stopped the visit because $E_B^2(v) > R$. By Lemma 1., $d'(v)$ is the value of $d(P_v)$ for some R-feasible P which has minimal $d(P)$ over all R-feasible paths through v in T_v^B . Hence the algorithm calculates the minimal value of $d(P_v)$ over all vertices v which have an R-feasible solution through v in T_v^B . The optimal solution, if it exists, must be one of these P_v for some v . Hence d^* at the end of the algorithm is optimal, if an R-feasible solution exists, and v^* is the associated vertex. If no R-feasible solution exists, the algorithm finishes at the first **STOP**. It follows from the proof of Lemma 1. that ALGORITHM FINDPATH finds an R-feasible path P^* through v^* in $T_{v^*}^B$ which has minimal value of $d(P)$. $\square$

Theorem 2. Algorithm *Cent-dian* finds an optimal solution in $O(n)$ time.

Proof. All the quantities that are needed for the preprocessing and for the algorithm can be computed for all v in $O(n)$ time (see [2,3]) by making a bottom up pass and a top down pass, level by level, of the tree. To execute the algorithms, we need to remember the values of $E_B^1(v), E_B^2(v), E_B^3(v), E_U(v), d(v), d'(v), w_B^1(v), w_B^2(v)$. The algorithms themselves are then clearly $O(n)$ in time and space.

4 The Algorithm for the Bounded Cent-dian Problem

In this section we consider the above problem along with an additional constraint, namely that of finding a path P which minimizes DISTSUM with ECCENTRICITY at most R and with length less than or equal to ℓ . The procedure is based on the following comments.

Remark 2. Given a tree $T = (V, E)$ and a vertex $v \in V$ we have two cases:

- the *Bounded Cent-dian path* contains v ;
- the *Bounded Cent-dian path* is fully contained in one of the subtrees obtained by removing v from T .

Definition 3. Given a weighted tree T , a *central vertex* v of T is the *centroid* of the corresponding unweighted tree T , that is a vertex that minimizes the maximum of the number of vertices of the subtrees obtained by removing v [12].

Remark 3. A central vertex has maximum subtree cardinality less than or equal to $n/2$ and an (unweighted) median of a tree is a central vertex [7]. Computing the central vertex requires $O(n)$ time with the algorithm in [4].

We will now outline the preprocessing phase for this algorithm. We use a slightly different notation from the previous section. Given a vertex $v \in V$, consider v as the root of T , call it T^v and denote by T_u^v the subtree of T^v rooted at u . Then, by using the recursive formulas of Section 2, we can compute in $O(n)$ time the sum of the weights of the vertices in each subtree of $T_u^v \forall u \in T^v$, say $sum_v(u)$; the DISTSUM $d(v)$; the values, $\forall u \in T_u^v$, of the eccentricity $E_B^i(u)$ with $i = 1, 2, 3$; $E_U(u)$ needs to be carried along as well and the values of:

$$sav(v, P_{vu}) = sav(v, P_{vp(u)}) + sum_{p(u)}(u)a(p(u), u) \quad (13)$$

where if $v = p(v)$ then $sav(v, P_{vp(v)}) = sav(v, v) = 0$. The DISTSUM of the path $P_{vu} \forall u \in V \setminus \{v\}$ is therefore:

$$d(P_{vu}) = d(v) - sav(v, P_{vu}) \quad (14)$$

Finally, by a top down scan of T^v , we can find the lengths of the paths P_{vu} from v to each $u \in V \setminus \{v\}$ as follows:

$$L(P_{vu}) = L(P_{vp(u)}) + a(p(u), u) \quad (15)$$

This complete the preprocessing. The idea of the algorithm is that we root the tree at a “central” vertex m of T , giving T^m , and then we first search for the paths P passing through m which are feasible with respect to the *eccentricity*. As in the previous section, what we have to do is to check the feasibility of all the paths passing through the “central” vertex. We then have the problem of finding a path P passing through m with minimum DISTSUM and length less than or equal to ℓ . See [3] for a discussion and for an algorithm for this problem. We then do this recursively for each subtree obtained by removing m .

ALGORITHM Bounded Cent-dian:

Input: a weighted tree T

Output: a *Bounded Cent-dian path* P^* of T and its DISTSUM d^*

begin

$d^* := +\infty$

$P^* := \emptyset$

SUBTREE(T')

end

Procedure SUBTREE(T'):

Input: a subtree $T' = (V', E')$ of T with $|V'| = n'$, the best current DISTSUM d^*

Output: if the best path P in T' has DISTSUM less than d^* , the best path P^* in T' and its DISTSUM d^*

begin

Update the preprocessed data structures

find a central vertex m of T' , call the tree T'^m

compute the values of $E_B^i(u)$ for $i = 1, 2, 3$ and $E_U(u) \forall u \in T'^m$

if there exists $u \in T'^m$ such that $E_B^3(u) > R$ **or** there exist at least two

vertices u and w with $u \neq w$ such that $E_B^2(u) > R$ and $E_B^2(w) > R$

then Stop the Algorithm The problem is infeasible

else CASE 1

if there exists exactly one vertex $w \in T'^m$ such that $E_B^2(w) > R$

if $E_U(w) > R$ **Stop the Algorithm** /*No feasible path in T^* /*

else /*the only feasible path in T could be the one passing through w^* /*

consider the two sons $u = u_B^1(w)$ and $u' = u_B^2(w)$ of w in T'^m

let P_{wx} and $P_{wx'}$ be the two paths from w to the leaves x and x'

passing through u and u'

/* P_{wx} and $P_{wx'}$ give the values of the eccentricity $E_B^1(w)$ and $E_B^2(w)^*$ /*

proceed down on P_{wx} until the first vertex t with $E_B^1(t) \leq R$ is found

proceed down on $P_{wx'}$ until the first vertex t' with $E_B^1(t') \leq R$ is found

if $L(P_{wt} \cup P_{wt'}) > \ell$ **Stop the Algorithm** /*No feasible path in T^* /*

else

contract to a point $P_{wt} \cup P_{wt'}$ and denote by w' the new root of T'_w

Apply the algorithm in [3] to find an ℓ -feasible path P

of minimal DISTSUM through w

Stop the Algorithm /*this is the only feasible path in T^* /*

else CASE 2 $E_B^1(m) > R$ and $E_B^2(m) \leq R$

/*There could be a path passing through m^* /*

consider the son $u = u_B^1(m)$ of m

consider the path P_{mx} from m to a leaf x , passing through

the vertex u , which gives the value of the eccentricity $E_B^1(m)$

proceed down on P_{mx} until the first vertex t with $E_B^1(t) \leq R$ is found

if $L(P_{mt}) > \ell$ **Stop** /*No feasible path in T'^m /*

else

contract the path P_{mt} to a point and let m' be the new root of T'

Apply the algorithm in [3] to find an ℓ -feasible path P

of minimal DISTSUM through m

if $d(P) < d^*$ **then** $d^* := d(P)$ and $P^* := P$

if $E_U(u_B^1(m)) \leq R$ **then** SUBTREE($T'_{u_B^1(m)}$)

```

else Stop the Algorithm
else CASE 3    $E_B^1(m) \leq R$ 
/*In  $T^m$  all the paths passing through  $m$  are R-feasible*/
Apply the algorithm in [3] to find the path  $P$  passing through  $m$ 
with minimum DISTSUM and length at most  $\ell$ 
if  $d(P) < d^*$  then  $d^* := d(P)$  and  $P^* := P$ 
for each son  $s$  of  $m$  do
    if  $E_U(s) \leq R$  then SUBTREE( $T_s^m$ )
end

```

Remark 4. The Algorithm in [3] referred to in the above algorithm needs minor modification to apply to the case where part of the path has been contracted.

4.1 Complexity Analysis and Correctness

Remark 5. Given a rooted tree T , the procedure described in [3] finds a path passing through the root of T with minimum DISTSUM and with length less than or equal to ℓ in $O(n \log n)$ time.

Once we apply the recursion to the subtrees rooted at sons of m we have to recompute all the quantities calculated during the preprocessing phase (see the Update command on the first line of procedure SUBTREE(T')). The following Remark allows us to update such quantities without recomputing them from scratch.

Remark 6. Given is a tree T^v rooted at a vertex v . Denote by $sum(v)$ the sum of the weights of T^v . Let (v, u) be an edge incident to v and $T' = (V', E')$ be the subtree T_u^v rooted at u with $|V'| = n'$. Moreover let $d_v(u)$ be the sum of the distances from all the vertices in T' to u and let $sum_v(u)$ be the sum of the weights of the vertices in T' . Now we consider a vertex v' in T_u^v as the new root of T and we call $T^{v'}$ the rooted tree so obtained. We have:

$$sum_{v'}(v) = sum(v) - sum_v(u);$$

$$d_{v'}(v) = d(v) - (d_v(u) + sum_v(u)a(v, u)).$$

Let $E_B^i(v)$ with $i = 1, 2, 3$ be the three values of eccentricity at the root v in T^v . We show how to update these values in $T^{v'}$ with v' the new root. In what follows let us assume that $deg(v) > 3$. Suppose u is the vertex that gives neither values of eccentricity at v in T^v (i.e. $E_B^i(v)$ $i = 1, 2, 3$). Then, these three values remain unchanged at vertex v in $T^{v'}$. Suppose without loss of generality that $u = u_B^1(v)$ in T^v then, in $T^{v'}$ there are at least three edges two of which give $E_B^2(v)$ and $E_B^3(v)$. Now in $T^{v'}$ $E_B^2(v)$ becomes $E_B^1(v)$ and $E_B^3(v)$ becomes $E_B^2(v)$. For finding the new value of $E_B^3(v)$ we have:

$$E_B^3(v) = \max_w \text{ a son of } v \{E_B^1(w) + a(v, w) | w \neq u_B^2(v) \text{ and } w \neq u_B^3(v)\}$$

where $u_B^2(v)$ and $u_B^3(v)$ are the vertices that give $E_B^2(v)$ and $E_B^3(v)$ in T^v , respectively. Similar arguments apply if $u = u_B^2(v)$ or $u = u_B^3(v)$ in T^v . Hence, having already computed in T^v the quantities $sum_v(u)$, and the DISTSUM $d_v(u)$, $\forall u \neq v$, we can compute the corresponding quantities in $T^{v'}$, that is $sum_{v'}(v)$, $d_{v'}(v)$ and $E_B^i(v)$ $i = 1, 2, 3$ in $O(n')$ time. $\square$

Theorem 3. Algorithm *Bounded Cent-dian* correctly finds an optimal solution, if it exists, in $O(n \log^2 n)$ time.

Proof. At the beginning of the procedure SUBTREE(T'), if there exists a vertex u such that $E_B^3(u) > R$ or there exist at least two vertices u and w with $E_B^2(u) > R$ and $E_B^2(w) > R$ there is not possible to find an R -feasible path passing through any vertex of T . In CASE 1 one can check only if there is a path passing through w in T_w^m since any path passing through vertices different from w and not containing w , clearly has eccentricity greater than R . Similar arguments apply in CASE 2. Indeed, either there exists a path passing through m or it must lie in the subtree rooted at the son s of m which gives the value of $E_B^1(m) > R$. However, the recursion is applied only if $E_U(s) \leq R$ otherwise any path passing through vertices in T_s^m is not R -feasible. Thus, the algorithm correctly finds an R -feasible path if it exists or it stops. Finding the vertex t in CASE 2 (t and t' in CASE 1) requires $O(n')$ time in a subtree T' since there exists only one path P_{mx} that gives $E_B^1(m)$ (only two paths in CASE 1) otherwise there will not exist any feasible path in T^m . Hence, using Remark 5 and 6, the complexity of procedure SUBTREE(T') is $O(n \log n)$. Since the recursion is performed on the central vertex of a subtree, Remark 3 implies that the depth of the recursion tree is $O(\log n)$ so, the overall time complexity of the algorithm *Bounded Cent-dian* is $O(n \log^2 n)$. $\square$

Notice that if the tree is unweighted, the paths starting from m are already ordered by length and in this case the algorithm has an overall time complexity of $O(n \log n)$.

Appendix

We provide an example of the algorithm for Problem 1 with $R = 65$. We refer to the same weighted tree as in the example in [1] where all the vertices have weights equal to 1. The tree is rooted at vertex v_6 . To each vertex v are associated the values $E_B^i(v)$ $i = 1, 2, 3$ and $E_U(v)$ (the ones in the squared brackets). In the following table, for each vertex there are the values $d_B(v)$, $d(v)$ and $d'(v)$. The optimal solution is given by $P = \{v_7, v_6, v_2, v_{10}, v_1\}$ with $d(P) = 85$ and $E(P) = 20$. Notice that, in the table “-” means that there is no feasible path passing through v .

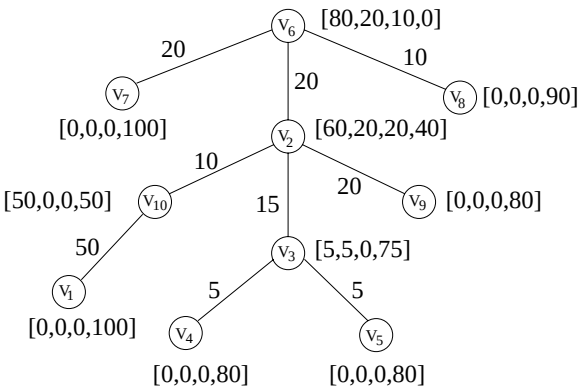


Fig. 1. An example

Table 1. Values of the vertices

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}
$d_B(v)$	0	145	10	0	0	315	0	0	0	50
$d(v)$	695	235	295	335	335	315	475	395	395	295
$d^B(v)$	-	115	-	-	-	85	-	-	-	245

References

1. I. Averbakh and O. Berman.: Algorithms for path medi-center of a tree. Computers and Operations Research, **26** (1999) 1395-1409. 744, 753

2. R. I. Becker.: Inductive algorithms on finite trees. Quaestiones Mathematicae, **13** (1990) 165-181. 744, 745

3. R. I. Becker, Y. Chiang, I. Lari, A. Scozzari, and G. Storchi.: Finding the ℓ -core of a tree. Quaderni del Dipartimento di Statistica, Probabilità e Statistiche Applicate, **15** (1999). To appear, Discrete Applied Mathematics. 743, 745, 750, 751, 752

4. A. J. Goldman.: Optimum center location in simple networks. Transportation Science, **5** (1971) 212-221. 750

5. J. Halpern.: The location of a center-median convex combination on an undirected tree. Journal of Regional Science, **16** (1976) 237-245. 743, 744

6. J. Halpern.: Finding minimal center-median convex combination (cent-dian) of a graph. Management Science, **24** (1978) 534-544. 743

7. O. Kariv and S. L. Hakimi.: An algorithmic approach to network location problems. II: The p -medians. SIAM journal of Applied Mathematics, **37** (1979) 539-560. 750

8. T. U. Kim, T. J. Lowe, A. Tamir, and J. E. Ward.: On the location of a tree-shaped facility. Networks, **28** (1996) 167-175. 744, 745

9. M. Labbé, D. Peeters, and J. F. Thisse.: Location on networks. Handbooks in O. R. and M. S., **8** (1995) 551-624. 743

10. C. A. Morgan and J. P. Slater.: A linear algorithm for a core of a tree. Journal of Algorithms, **1** (1980) 247-258. 745, 746

11. S. Peng and W. Lo.: Efficient algorithms for finding a core of a tree with a specified length. *Journal of Algorithms*, **20** (1996) 445-458. 743, 744
12. P. J. Slater.: Locating central paths in a network. *Transportation Science*, **16** (1982) 1-18. 745, 750
13. A. Tamir.: An $O(pn^2)$ algorithm for the p -median and related problems on tree graphs. *Operation Research Letters*, **19** (1996) 59-64. 744, 745

Approximate Hotlink Assignment[★]

Evangelos Kranakis¹, Danny Krizanc², and Sunil Shende³

¹ Carleton University, School of Computer Science
Ottawa, ON, K1S 5B6, Canada.
`kranakis@scs.carleton.ca`

² Department of Mathematics, Wesleyan University
Middletown CT 06459, USA
`dkrizanc@wesleyan.edu`

³ Department of Computer Science, Rutgers University
Camden, NJ, 08102, USA.
`shende@crab.rutgers.edu`

Abstract. Consider a directed rooted tree $T = (V, E)$ of maximal degree d representing a collection V of web pages connected via a set E of links all reachable from a source home page, represented by the root of T . Each leaf web page carries a weight representative of the frequency with which it is visited. By adding hotlinks, shortcuts from a node to one of its descendants, we are interested in minimizing the expected number of steps needed to visit the leaf pages from the home page. We give an $O(N^2)$ time algorithm for assigning hotlinks so that the expected number of steps to reach the leaves from the root of the tree is at most $\frac{H(p)}{\log(d+1) - (d/(d+1)) \log d} + \frac{d+1}{d}$, where $H(p)$ is the entropy of the probability (frequency) distribution $p = \langle p_1, p_2, \dots, p_N \rangle$ on the N leaves of the given tree, i.e., p_i is the weight on the i th leaf. The best known lower bound for this problem is $\frac{H(p)}{\log(d+1)}$. Thus our algorithm approximates the optimal hotlink assignment to within a constant for any fixed d .

1 Introduction

In an attempt to enhance the experience and reduce the latency of the average user, a number of authors have suggested ways of improving the design of web-sites, such as promoting and demoting pages, highlighting links, and clustering related pages in an adaptive fashion depending on user access patterns [4,7]. In this paper we consider the strategy of adding “hotlinks”, i.e., shortcuts from web pages at or near the home page of a site to popular pages a number of levels down in the (generally directed) network of pages. The idea of “hotlinks” was suggested by Perkowitz and Etzioni [7] and studied earlier by Bose et al. [2] for the special case of a website represented by a complete binary tree. Experimental results showing the validity of this approach are given in [3].

[★] Research supported in part by NSERC (Natural Sciences and Engineering Research Council) of Canada and MITACS (Mathematics of Information Technology and Complex Systems) grants.

We model a website as a rooted directed tree $T = (V, E)$ where V is a collection of webpages connected by a set E of links. We assume that all webpages are reached starting from the root of the tree, representing the home page of the site and that users are interested in accessing information stored at the leaf webpages. Each leaf carries a weight representative of the frequency with which it is visited, i.e., its popularity. Our goal in adding hotlinks (directed edges from a node to one of its descendents) is to minimize the expected number of pages a user would visit when starting at the root and attempting to reach a leaf. We assume a user will always follow a hotlink (u, v) if after reaching u he or she wishes to reach a leaf that is a descendent of v . Note that this implies that adding hotlinks to a tree results in a new tree, not a general directed graph. We restrict ourselves to the case where at most one hotlink is added per node, but our results can be extended to the case where more than one hotlink can be added per node.

Consider a rooted directed tree T with N leaves and of maximal degree d . Let T^A be the tree resulting from an assignment A of hotlinks. The expected number of steps from the root to find a leaf web page is

$$E[T^A, p] = \sum_{i \text{ is a leaf}} d_A(i) p_i,$$

where $d_A(i)$ is the distance of the node i from the root in T^A , and $p = \langle p_i : i = 1, \dots, N \rangle$ is the probability distribution on the leaves of the original tree T . We are interested in finding an assignment A which minimizes $E[T^A, p]$.

An lower bound on $E[T^A, p]$ was given in [2] using information theory. Let $H(p)$ be the Entropy (see [1]) of the probability distribution $p = \langle p_i : i = 1, \dots, N \rangle$, which is defined by the formula¹

$$H(p) = \sum_{i=1}^N p_i \log(1/p_i). \quad (1)$$

A tree of maximal degree d can be thought of as the encoding of the leaves with the d -symbol alphabet $0, 1, \dots, d-1$. Adding a hotlink increments the alphabet by a single symbol to form an alphabet with $d+1$ symbols. Using this and the theory of prefix codes the following result can be proved (see [1,2]):

Theorem 1 ([2]). *Consider an arbitrary rooted tree of maximal degree d . For any probability distribution $p = \langle p_1, p_2, \dots, p_N \rangle$ on the leaves of the tree and any assignment of at most one hotlink per node the expected number of steps to reach a web page located at a leaf from the root of the tree is at least $\frac{H(p)}{\log(d+1)}$. $\square$*

Our main result is to show that the above lower bound can be matched to within a constant for any constant d .

Theorem 2. *Consider an arbitrary rooted tree of maximal degree d . There is an algorithm, quadratic in the number of vertices of the tree, which for any*

¹ Throughout this paper $\log$ denotes logarithm in base 2 and $\ln$ logarithm in base e .

probability distribution $p = \langle p_1, p_2, \dots, p_N \rangle$ on the leaves of the tree assigns one hotlink per node in such a way that the expected number of steps to reach a leaf of the tree from the root is at most $\frac{H(p)}{\log(d+1) - (d/(d+1)) \log d} + \frac{d+1}{d}$. $\square$

Section 2 provides the proof of the main theorem for the case of binary trees. Subsection 2.1 provides the proof of a useful lemma concerning entropies. In Section 3 we extend the proof to arbitrary trees of maximum degree d . In Section 4 we discuss an improved analysis of our algorithm on complete trees and in Section 5 we conclude with some open problems.

2 Hotlink Assignments for Binary Trees

In this section we give the hotlink assignment that achieves the upper bound of the main theorem. For the sake of simplicity we first consider the case of binary trees. Later we adapt the result to the case of trees of maximum degree d .

2.1 A Useful Lemma on Entropies

Before giving the hotlink assignment algorithm and its analysis we present a useful lemma concerning entropy.

Consider a probability distribution $p = \langle p_1, p_2, \dots, p_N \rangle$ and a partition $A_1, A_2, \dots, A_k$ of the index set $\{1, 2, \dots, N\}$ into k non-empty subsets. Define

$$S_i = \sum_{j \in A_i} p_j, \text{ for } i = 1, 2, \dots, k. \quad (2)$$

Consider the new distributions:

$$p^{(i)} = \langle p_j^{(i)} := \frac{p_j}{S_i} : j \in A_i \rangle, \text{ for } i = 1, 2, \dots, k. \quad (3)$$

Lemma 1. *For any partition $A_1, A_2, \dots, A_k$ of the index set of the probability distribution we have the identity*

$$H(p) = \sum_{i=1}^k S_i H(p^{(i)}) - \sum_{i=1}^k S_i \log S_i, \quad (4)$$

where S_i and $p^{(i)}$ are defined in Equations (2, 3).

PROOF The proof is a straightforward application of the definition of the entropy. We have

$$H(p) = \sum_{j=1}^N -p_j \log p_j$$

$$\begin{aligned}
&= \sum_{i=1}^k - \sum_{j \in A_i} p_j \log p_j \\
&= \sum_{i=1}^k -S_i \sum_{j \in A_i} p_j^{(i)} \log p_j \\
&= \sum_{i=1}^k -S_i \sum_{j \in A_i} p_j^{(i)} (\log p_j^{(i)} + \log S_i) \\
&= \sum_{i=1}^k S_i H(p^{(i)}) - \sum_{i=1}^k S_i \log S_i,
\end{aligned}$$

which proves the Lemma. $\square$

2.2 Algorithm for Binary Trees

Before we proceed any further we need to show how to assign weights to all the nodes of the tree. In $O(N)$ time we can propagate the original weights on the leaves of the tree through the entire tree using a bottom-up process. This is done inductively as follows. The weight of the i th leaf is equal to p_i . The weight of a node u is equal to the sum of the weights of its children. Finally, we define the weight of a subtree to be equal to the weight of the root of this subtree. We present the following well-known lemma for completeness.

Lemma 2. *Consider a probability distribution $p = \langle p_1, p_2, \dots, p_N \rangle$ on the N leaves of a binary tree. Then either there is a tree node whose weight is between $1/3$ and $2/3$ or else there is a leaf whose weight is $> 2/3$.*

PROOF Assume there is no tree node whose weight is in the range $[1/3, 2/3]$. We will show that there is a leaf whose weight is $> 2/3$. Start from the root. All its descendants have weight outside the range $[1/3, 2/3]$. Take a descendant c that has weight $> 2/3$. If this is a leaf then we are done. So assume it is not a leaf. All descendants of c have weight outside the range $[1/3, 2/3]$. Hence, c has a descendant whose weight is $> 2/3$. If this is a leaf then we are done. So assume it is not a leaf. Iterate this procedure and we see that there must exist a leaf of weight $> 2/3$. This completes the proof of the Lemma. $\square$

Theorem 3. *Consider a rooted binary tree. There is an algorithm, quadratic in the number of vertices of the tree, which for any probability distribution on the leaves of the tree assigns a hotlink per node in such a way that the expected number of steps to reach a leaf of the tree is at most $aH(p) + b$, where $a = \frac{1}{\log 3 - 2/3}$ and $b = \frac{3}{2}$.*

PROOF As mentioned before in $O(N)$ time we can propagate the original weights on the leaves of the tree through the entire tree. Once all these internal node weights are assigned we use a top-down method to assign hotlinks.

The assignment of hotlinks is done recursively. We partition the tree T into three subtrees T_1, T_2, T_3 . The partition is determined as follows. Find a node u that determines a subtree T_2 rooted at u such that the weight of u is bounded from below by $1/3$ and from above by $2/3$. I.e.,

$$\frac{1}{3} \cdot \text{weight}(T) \leq \text{weight}(u) \leq \frac{2}{3} \cdot \text{weight}(T). \quad (5)$$

If u is not a child of c then do the following. Without loss of generality assume T_2 is contained entirely inside the subtree rooted at the left child of the root. Then T_1 is the tree rooted at the left child of the root minus the tree T_2 and T_3 is the tree rooted at the right child of the root. The recursive process assigns a hotlink to the root u of the subtree T_2 . If however, the only node u satisfying Inequalities (5) must be a child of c then we select u to be the heaviest grandchild of c , which must have weight at least $1/4$ of the weight of T (this is because the tree is binary). If no such node exists then we choose for u the heaviest leaf of the tree which is guaranteed to have weight greater than $2/3$. The recursive process assigns a hotlink to this new node u . The trees T_1, T_2, T_3 are defined exactly as before, and moreover none of them has weight bigger than $2/3$ of the weight of T . Then we recursively assign hotlinks to the subtrees T_1, T_2, T_3 . The idea is illustrated in Figure 1.

The precise algorithm **HotlinkAssign** which determines the hotlink assignment is defined as follows.

HotlinkAssign(T)

Initialize: $c := \text{root of } T$, $l := \text{left } (r := \text{right}) \text{ child of } c$;

if c has grandchildren **do**;

1a. **find** u descendant of c such that

$$\text{weight}(T)/3 \leq \text{weight}(u) \leq 2\text{weight}(T)/3$$

1b. if no such descendant exists let u be a max weight leaf;

2a. **if** distance from c to u is ≥ 2

then add a hotlink from c to u

2b. **else** let u be the (any) grandchild of c with heaviest weight;

add a hotlink from c to u

3. $T_2 := \text{tree rooted at } u$;

4. Let v ancestor of u that is child of c ;

w.l.o.g. assume $v = l$;

5. $T_1 := \text{tree rooted at left child of } c \text{ minus } T_2$;

6. $T_3 := \text{tree rooted at right child of } c$;

7. **HotlinkAssign** (T_1),

8. **HotlinkAssign** (T_2),

9. **HotlinkAssign** (T_3)

end;

We will prove by induction on the depth of the tree T that there exist constants a, b such that for the hotlink assignment A described above

$$E[T^A, p] \leq aH(p) + b.$$

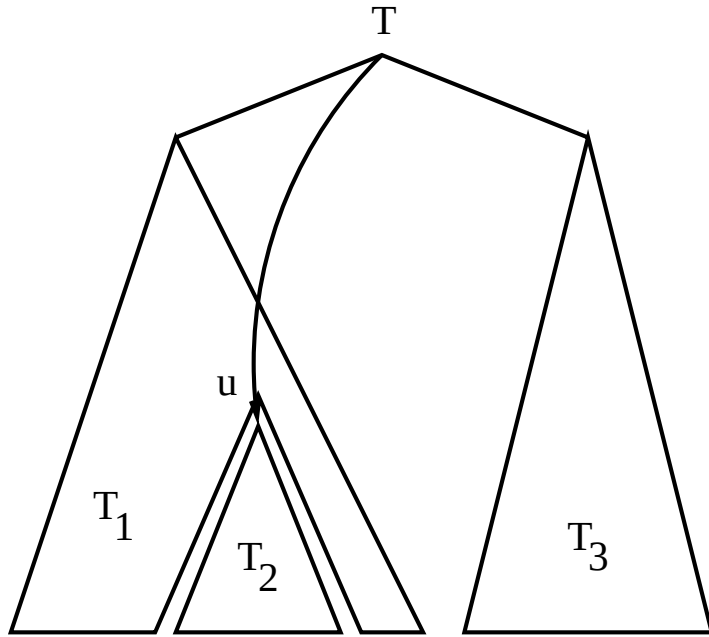


Fig. 1. Assignment of hotlinks: we assign a hotlink from the root to a “heavy” node u and iterate the recursive assignment to the subtrees T_1, T_2, T_3

It will become apparent from the proof below how to select a and b (see Inequalities (6) and (7)).

The initial step, when the depth of the tree is 1, is trivial because we will choose b so that $b \geq 1$. Assume the induction hypothesis is valid for the subtrees of T . We calculate costs of the resulting hotlink assignments. According to Lemma 2 the hotlink assigned from the root to a node partitions the leaves of the tree into three subsets A_1, A_2, A_3 with corresponding weights S_1, S_2, S_3 . If Lemma 2 chooses a node that has weight between $1/3$ and $2/3$ (or if it is the child of the root and a grandchild is chosen with weight greater than or equal to $1/4$) then it is easy to see that all three S_i ’s have weight $\leq 2/3$. If Lemma 2 chooses a leaf then $S_2 > 2/3$ and $S_1 + S_3 < 1/3$.

In the first case, using the notation of Lemma 1 we obtain

$$\begin{aligned} E[T^A, p] &= \sum_{i=1}^3 S_i (1 + E[T_i^A, p^{(i)}]) \\ &= 1 + \sum_{i=1}^3 S_i E[T_i^A, p^{(i)}] \end{aligned}$$

$$\begin{aligned}
&\leq 1 + \sum_{i=1}^3 S_i(aH(p^{(i)}) + b) \\
&= 1 + b + a \sum_{i=1}^3 S_i H(p^{(i)}) \\
&= 1 + b + aH(p) + a \sum_{i=1}^3 S_i \log S_i \\
&\leq aH(p) + b.
\end{aligned}$$

The last inequality being valid because we will choose the constant a such that

$$1 + a \sum_{i=1}^3 S_i \log S_i \leq 0. \quad (6)$$

If A_2 is a leaf of weight greater than $2/3$ then T_2 is the empty tree and using the notation of Lemma 1 we obtain

$$\begin{aligned}
E[T^A, p] &= S_2 + \sum_{i=1,3} S_i(1 + E[T_i^A, p^{(i)}]) \\
&= 1 + \sum_{i=1,3} S_i E[T_i^A, p^{(i)}] \\
&\leq 1 + \sum_{i=1,3} S_i(aH(p^{(i)}) + b) \\
&= 1 + a \sum_{i=1,2,3} S_i H(p^{(i)}) + (S_1 + S_3)b \\
&= 1 + aH(p) + a \sum_{i=1}^3 S_i \log S_i + (S_1 + S_3)b \\
&\leq 1 + aH(p) + (S_1 + S_3)b \\
&\leq 1 + aH(p) + (1/3)b \\
&\leq aH(p) + b
\end{aligned}$$

The last inequality being valid because we will choose the constant b such that

$$1 + (1/3)b \leq b. \quad (7)$$

We now consider the value of the constants a and b . Clearly, $b = 3/2$ satisfies Inequality (7). To choose a we first prove the following lemma.

Lemma 3. *The solutions of the optimization problem*

$$\begin{aligned}
&\text{maximize } f(x, y) = x \ln x + y \ln y + (1 - x - y) \ln(1 - x - y) \\
&\text{subject to } 0 \leq x, y \leq 2/3, 1/3 \leq x + y \leq 1
\end{aligned}$$

are such that $\{x, y, 1 - x - y\} = \{0, 1/3, 2/3\}$.

PROOF The partial derivative of f with respect to x is equal to $\ln x - \ln(1-x-y)$, which is increasing as a function of x . This means that the maximum values of f (as a function only of x parametrized with respect to y) are obtained at the endpoints of the interval on which it is defined, i.e., $\max\{0, 1/3 - y\} \leq x \leq \min\{2/3, 1 - y\}$. This gives rise to two cases depending on the value of y .

Case 1: $y \geq 1/3$.

In this case we have that the endpoints are $0 \leq x \leq 1 - y$ and the value of f at the endpoints is equal to

$$\begin{aligned} f(0, y) = f(1 - y, y) &= y \ln y + (1 - y) \ln(1 - y) \\ &\text{subject to } 1/3 \leq y \leq 2/3. \end{aligned} \quad (8)$$

Case 2: $y \leq 1/3$.

In this case we have that the endpoints are $1/3 - y \leq x \leq 2/3$ and the value of f at the endpoints is equal to

$$\begin{aligned} f(1/3 - y, y) = f(2/3, y) &= (1/3 - y) \ln(1/3 - y) + y \ln y + (2/3) \ln(2/3) \\ &\text{subject to } 0 \leq y \leq 1/3. \end{aligned} \quad (9)$$

In particular, the maximum value of f is attained at the maximum values of Cases 1 and 2 above. The functions in Equations (8, 9) depend only on the variable y and their maxima are obtained at the endpoints of the interval for y on which the function is defined. In Case 1, this is $1/3 \leq y \leq 2/3$ and in Case 2, this is $0 \leq y \leq 1/3$. It follows that for Case 1 we have that our function obtains its maximum value when $y = 1/3, 2/3$ and in Case 2 when $y = 0, 1/3$, i.e., $y = 0, 1/3, 2/3$. Consequently, when $y = 0$ we have $x = 1/3, 2/3$, when $y = 1/3$ we have $x = 0, 2/3$, and when $y = 2/3$ we have $x = 0, 1/3$. This proves the lemma. $\square$

Lemma 3 implies that $\sum_{i=1}^3 S_i \log S_i \leq (2/3) \log(2/3) + (1/3) \log(1/3)$. Hence, Inequality (6) is satisfied when $a = \frac{1}{\log 3 - 2/3}$.

Concerning the running time of the algorithm we note that in linear time we can assign weights to the nodes of the tree bottom up. For each recursive call of the algorithm **HotlinkAssign** we must update the weights. Since the number of recursive calls does not exceed the height of the tree, we see that the running time is worst-case quadratic. The proof of Theorem 3 is now complete. $\square$

3 Trees of Maximum Degree d

Lemma 2 has a natural generalization for the case of trees with maximum degree d . Namely the following result is known.

Lemma 4. *Consider a probability distribution $p = \langle p_1, p_2, \dots, p_N \rangle$ on the N leaves of a tree of maximum degree d . Then either there is a tree node whose weight is between $1/(d+1)$ and $d/(d+1)$ or else there is a leaf whose weight is $> d/(d+1)$.* $\square$

3.1 Algorithm for Trees of Maximum Degree d

Now we can prove our main result in Theorem 2. In the sequel we indicate only the necessary changes.

PROOF (OUTLINE) of Theorem 2. As with Theorem 3 we assign weights to all nodes of the tree in a bottom-up fashion. The assignment of hotlinks is done recursively in a top-down fashion. Let c be the root of T . Indeed, we partition the tree T into at most $d + 1$ subtrees as follows. By Lemma 4, find a node u that determines a subtree T'_i rooted at u such that the sum of the weights of the leaves of T'_i is bounded from below by $1/(d + 1)$ and from above by $d/(d + 1)$. Without loss of generality assume this tree is a descendant of the i th child of the root. Then T_i is defined to be the tree rooted at the i th child of the root minus the tree T'_i . Also T_j , for $j \neq i$, is defined to be the tree rooted at the j -th child of the root. Now, T'_i and the sequence $T_1, T_2, \dots, T_i, \dots$ is the desired partition of the subtrees. As before, if only children of c are the only nodes whose weight is bounded from below by $1/(d + 1)$ and from above by $d/(d + 1)$ then we select u to be the (any) heaviest grandchild of c . The recursive process assigns a hotlink from c to the root u of the subtree T'_i . Then we recursively assign hotlinks to the subtrees $T_1, T_2, \dots, T_d$.

Using Lemma 4 and an analysis similar to that of Theorem 3 we obtain the main result. As before we prove by induction on the depth of the tree T that there exist constants a, b such that for the hotlink assignment A described above

$$E[T^A, p] \leq aH(p) + b.$$

Inequality (6) is transformed into

$$1 + a \sum_{i=1}^{d+1} S_i \log S_i \leq 0, \quad (10)$$

and Inequality (7) into

$$1 + (1/(d + 1))b \leq b. \quad (11)$$

Here, a and b are selected so as to satisfy Inequalities (10) and (11). The value of a follows immediately from the following lemma.

Lemma 5. *The solutions of the optimization problem*

$$\begin{aligned} \text{maximize } f(s_1, s_2, \dots, s_d) &= \sum_{i=1}^d s_i \ln s_i + \left(1 - \sum_{i=1}^d s_i\right) \ln \left(1 - \sum_{i=1}^d s_i\right) \\ \text{subject to } 0 \leq s_1, s_2, \dots, s_d &\leq \frac{d}{d+1}, \frac{1}{d+1} \leq \sum_{i=1}^d s_i \leq 1 \end{aligned}$$

are obtained when one among the quantities $s_1, \dots, s_d, 1 - \sum_{i=1}^d s_i$ attains the value $d/(d + 1)$, another the value $1/(d + 1)$ and all the rest are equal to 0.

PROOF The proof is by induction on the number of variables. For $d = 2$ this is just Lemma 3. Assume the lemma is true for $d - 1 \geq 2$. We will prove it for d . Set $x := s_d$ and $y := s_1 + \dots + s_{d-1}$. The partial derivative of f with respect to x is equal to $\ln x - \ln(1 - x - y)$, which is increasing as a function of x . This means that the maximum values of f (as a function only of x parametrized with respect to $s_1, \dots, s_{d-1}$) are obtained at the endpoints of the interval on which it is defined, i.e., $\max\{0, 1/(d+1) - y\} \leq x \leq \min\{d/(d+1), 1 - y\}$. This gives rise to two cases depending on the value of y .

Case 1: $y \geq 1/(d+1)$.

In this case we have that the endpoints are $0 \leq x \leq 1 - y$ and the value of f at the endpoints is equal to

$$\begin{aligned} f(s_1, \dots, s_{d-1}, 0) &= f(s_1, \dots, s_{d-1}, 1 - y) \\ &= (1 - y) \ln(1 - y) \\ &\quad + s_1 \ln s_1 + \dots + s_{d-1} \ln s_{d-1} \\ &\text{subject to } 1/(d+1) \leq y \leq 1, \text{ and } s_1, \dots, s_{d-1} \leq d/(d+1). \end{aligned} \quad (12)$$

Case 2: $y \leq 1/(d+1)$.

In this case we have that the endpoints are $1/(d+1) - y \leq x \leq d/(d+1)$ and the value of f at the endpoints is equal to

$$\begin{aligned} f(s_1, \dots, s_{d-1}, 1/(d+1) - y) &= f(s_1, \dots, s_{d-1}, d/(d+1)) \\ &= (1/(d+1) - y) \ln(1/(d+1) - y) \\ &\quad + s_1 \ln s_1 + \dots + s_d \ln s_{d-1} \\ &\quad + (d/(d+1)) \ln(d/(d+1)) \\ &\text{subject to } 0 \leq y \leq 1/(d+1). \end{aligned} \quad (13)$$

Thus, we have reduced the original optimization problem to the two optimization problems described in Problems (12, 13) which have only $d - 1$ variables (i.e., one variable less). It is trivial that in Problem (13) the optimal solution is obtained when $s_1 = \dots = s_{d-1} = 0$. In this case $s_d = 1/(d+1)$ or $s_d = d/(d+1)$ and the inductive hypothesis is valid for d . In Case 1 we reduce to the same optimization problem on $d - 1$ variables. Hence, by the induction hypothesis the optimal solutions are obtained when one among the quantities $s_1, \dots, s_{d-1}, 1 - \sum_{i=1}^{d-1} s_i$ attains the value $d/(d+1)$, another the value $1/(d+1)$ and all the rest are equal to 0. Hence the result follows for d variables because in this case $s_d = 0$ or $s_d = 1 - \sum_{i=1}^{d-1} s_i$. The proof of the lemma is now complete. $\square$

The rest of the details of the proof can now be left to the reader. The proof of Theorem 2 is complete. $\square$

4 Analysis for Special Trees and Distributions

Our present analysis of the hotlink assignment problem focused on using the entropy of the distribution in order to bound the expected number of steps. In fact, our analysis still leaves a gap between the lower bound of Theorem 1 and the upper bound of Theorem 2. Can we improve the upper bound any further?

In the sequel we indicate that our algorithm still performs close to the lower bound for the uniform distribution on complete trees of degree d . We also indicate how to adapt our algorithm **HotlinkAssign** in the case of arbitrary distributions on complete trees.

First, consider the uniform distribution p on the leaves of the complete d -ary tree with $N \approx d^n$ leaves. The entropy of this distribution is $H(p) = n \log d$. Theorem 1 implies that

$$\frac{H(p)}{\log(d+1)} = \frac{\log d}{\log(d+1)} \cdot n \quad (14)$$

is a lower bound, while Theorem 2 implies that

$$\frac{H(p)}{\log(d+1) - (d/(d+1)) \log d} + \frac{d+1}{d} = \frac{\log d}{\log(d+1) - (d/(d+1)) \log d} \cdot n + \frac{d+1}{d}$$

is an upper bound on the expected number of steps.

However, in this case it is easy to see that the **HotlinkAssign** algorithm always picks a hotlink that is a grandchild of the current root. This observation can be used to give a different analysis of the algorithm. Using the method employed in [2][Theorem 3] we can show directly that the expected number of steps to reach a leaf is at most

$$\left(1 - \frac{1}{d^2}\right) \cdot n \quad (15)$$

plus an additive constant.

More generally, on a complete tree of degree d with an arbitrary distribution on the leaves we can change our algorithm **HotlinkAssign** so that a hotlink is placed always at the heaviest grandchild of the current root, i.e., we omit step 2a in algorithm **HotlinkAssign**. The analysis employed in [2][Theorem 3] as well as the resulting upper bound given in (15) is still valid. Moreover, it is easy to see that the lower bound in (14) and the upper bound in (15) are asymptotically identical, as $d \rightarrow \infty$.

5 Conclusions and Open Problems

In this paper we have considered the problem of assigning hotlinks to the nodes of a tree so as to minimize the expected number of steps from the root to the leaves under an arbitrary probability distribution. Our main result is an approximation algorithm for the case of bounded degree trees. A significant gap remains between the upper and lower bounds and further improvements would be of interest. It is expected that experimental results like the ones in [3] will provide additional insight on this problem. While it is known that the problem is NP-complete for DAGs, the complexity of the case of trees is still open. In this paper, we restricted ourselves to at most one hotlink added per node. Fuhrmann et al. [5] report results on the case of adding k links per node to a d -regular complete

tree. Our results can be extended to the case of a fixed number, k , added per node (using known generalizations of lemma 2) but the gap between upper and lower bounds increases with k . Perhaps another approach will not suffer from this weakness. The variation where the total number of hotlinks does not exceed a certain fixed budget could be explored. Additional interesting problems include finding further improvements for special distributions, such as Zipf's distribution (e.g., see [8,6]) which is especially relevant to this application.

References

1. N. Abramson, "Information Theory and Coding", McGraw Hill, 1963. 757
2. P. Bose, J. Czyzowicz, L. Gasienicz, E. Kranakis, D. Krizanc, A. Pelc, M. Vargas Martin, "Strategies for Hotlink Assignments," in Proceedings of ISAAC 2000, held in Taipei, Taiwan, Dec. 2000, D. T. Lee and Shang-Hua Teng, eds., SVLNCS, Vol. 1969, pp. 23-34. 756, 757, 766
3. J. Czyzowicz, E. Kranakis, D. Krizanc, A. Pelc, and M. Vargas Martin, "Evaluation of Hotlink Assignment Heuristics for Improving Web Access," in proceedings of the 2nd International Conference on Internet Computing (IC'2001), June 25th - 28th, 2001, Monte Carlo Resort, Las Vegas, Nevada, USA. 756, 766
4. M. C. Drott, "Using Web Server Logs to Improve Site Design," Proceedings of ACM Conference on Computer Documentation, 1998, pp. 43-50. 756
5. S. Fuhrmann, S. O. Krumke and H.-C. Wirth, "Multiple Hotlink Assignment," Proceedings of Workshop on Graph-Theoretic Concepts in Computer Science, 2001, SVLNCS, to appear. 766
6. D. Knuth, "The Art of Computer Programming: Volume 3, Sorting and Searching", Addison Wesley, 3rd ed., 1998. 767
7. M. Perkowitz and O. Etzioni, "Towards Adaptive Web Sites: Conceptual Framework and Case Study", Proceedings of Eight WWW Conference, 1999. 756
8. C. K. Zipf. "Human Behavior and the Principle of Least Effort", Reading Mass., Addison Wesley, 1949. 767

Efficient Algorithms for Two Generalized 2-Median Problems on Trees

Shan-Chyun Ku¹, Chi-Jen Lu², Biing-Feng Wang¹, and Tzu-Chin Lin¹

¹ Department of Computer Science, National Tsing Hua University,
Taiwan, ROC

² Institute of Information Science, Academia Sinica,
Taiwan, ROC

Abstract. The p -median problem on a tree T is to find a set S of p vertices on T that minimize the sum of distances from T 's vertices to S . For this problem, Tamir [12] had an $O(pn^2)$ -time algorithm, while Gavish and Sridhar [1] had an $O(n \log n)$ -time algorithm for the case of $p=2$. Wang *et al.* [13] introduced two generalizations by imposing constraints on the 2-median: one is to limit their distance while the other is to limit their eccentricity, and they had $O(n^2)$ -time algorithms for both. We solve both generalizations in $O(n \log n)$ time, matching even the fastest algorithm currently known for the 2-median problem. We also study cases when linear time algorithms exist for the 2-median problem and the two generalizations. For example, we solve all three in linear time when edge lengths and vertex weights are all polynomially bounded integers. Finally, we consider the relaxation of the two generalized problems by allowing 2-medians on any position of edges, instead of just on vertices, and we give $O(n \log n)$ -time algorithms for them.

1 Introduction

Optimally locating a set of facilities on a network is an important problem in the fields of transportation and communication [1, 2, 6, 11, 15]. One classical problem is the p -median problem: Given a graph $G=(V, E)$, in which each edge has a nonnegative length, and a number p , find a set S of p vertices to minimize some distance-sum function on S . This models the scenario of placing p facilities to minimize the average access cost from the network. Let $d(v, u)$ be the distance between vertices v and u and let $d(v, S)=\min_{u \in S} d(v, u)$. Kariv and Hakimi [8] considered the distance-sum function $Sum_G(S)=\sum_{v \in V} d(v, S)$ and showed that the problem is NP-hard. So one may only hope to find efficient algorithms for special classes of graphs. For trees, they provided in that paper an $O(p^2n^2)$ -time algorithm, while Tamir [12] had an $O(pn^2)$ -time algorithm with respect to a generalized distance-sum function. From now on, trees will be the only subject of our discussion and all our results are stated with respect to trees. Some

attention has been given to the cases of $p=1$ and $p=2$. Goldman [5] had a linear time algorithm for $p=1$. Gavish and Sridhar [1] had an $O(n \log n)$ -time algorithm for $p=2$ using the distance-sum function defined in [10]:

$$Sum_T(S) = \sum_{v \in V} d(v, S) \times w(v),$$

where each vertex v of the tree $T=(V, E)$ has a nonnegative weight $w(v)$. As an extension of Gavish and Sridhar's work, Wang *et al.* [13] introduced two generalizations of the 2-median problem by imposing constraints on the 2-median, from some practical point of view. One is to limit the distance between the pair of a 2-median. The other is to limit the *eccentricity* of a 2-median, with the eccentricity of a subset $S \subseteq V$ defined as $Ecc_T(S) = \max_{v \in V} d(v, S)$. The 2-median problem is their special case with large enough distance or eccentricity limit. Wang *et al.* [13] had $O(n^2)$ -time algorithms for both generalizations.

These two generalized problems of 2-median on trees are the focus of this paper. We solve both in $O(n \log n)$ time, matching even the fastest algorithm currently known for the 2-median problem. We can even do better for some natural cases. In linear time, we can solve the 2-median problem with constant edge lengths or polynomial vertex weights, the first generalization with polynomial vertex weights and polynomial edge lengths, and the second generalization with polynomial vertex weights. Finally, we consider the relaxation that allows 2-medians on any position of edges, instead of just on vertices. We give $O(n \log n)$ -time algorithms for the relaxed version of the two generalized problems.

Along the way, we solve a problem, named the tree marker problem, in $O(n \log n)$ time, which may be of independent interest. In linear time, we can solve the case in which all edge lengths are polynomially bounded integers. A special case of the tree marker problem is the tree bisector problem of Wang *et al.* [13, 14], for which they had an $O(n \log n)$ -time algorithm.

The rest of this paper is organized as follows. Notation and preliminaries are given in the next section. We solve the generalization with distance constraint in Section 3, and the generalization with eccentricity constraint in Section 4. In Section 5, we study cases when linear time algorithms exist. We solve the relaxed problems in Section 6. Finally, future work is discussed in Section 7.

2 Notation and Preliminaries

Given a tree T , let $V(T)$ denote its vertex set and let $E(T)$ denote its edge set. Each edge e has a nonnegative length $d(e)$ and each vertex v has a nonnegative weight $w(v)$. Define the weight of a subtree H as $w(H) = \sum_{v \in V(H)} w(v)$. For two vertices u and v , let $path(u, v)$ denote the path between them, let their distance $d(u, v)$ be the length of this path, and let $LCA(u, v)$ denote their least common ancestor when T is rooted.

We will use Gavish and Sridhar's distance-sum function defined in the previous section. For notational convenience, we write $Sum_T(v)$ for $Sum_T(\{v\})$ and $Sum_T(u, v)$ for $Sum_T(\{u, v\})$. A set S of p vertices is called a p -median of T if $Sum_T(S) \leq Sum_T(S')$ for any S' of p vertices. A median of a tree is just its 1-median.

Consider a rooted tree T and a vertex v . An ancestor of v is a vertex on the path from v to the root, so v is considered an ancestor of itself. Let $par(v)$ denote v 's parent in T . Let T_v denote the subtree rooted at v , and define

$$\delta_T(v) = w(T_v) - w(T \setminus T_v)$$

for the weight difference between T_v and its complement. For a child u of v , T_u is called a subtree of v . The following states an important property of medians on trees.

Lemma 2.1 On a rooted tree T , the vertex v with the smallest $\delta_T(v) > 0$ is a median.

Proof: Let v be the vertex with the smallest $\delta_T(v) > 0$. Imagine moving v to a vertex $r \neq v$. If r is not on T_v , vertices of T_v have their distances increased by $d(r, v)$ while others have distances decreased by at most $d(r, v)$, and the distance-sum increases by at least $\delta_T(v)d(r, v) > 0$. Otherwise, r is on a subtree T_u of v , and vertices outside of T_u have distances increased by $d(r, v)$ while others have distances decreased by at most $d(r, v)$. The distance-sum now increases by at least $-\delta_T(u)d(r, v)$, which is nonnegative as $\delta_T(u) < \delta_T(v)$ and $\delta_T(v)$ is the smallest positive one. Moving v elsewhere does not decrease the distance-sum, so v is a median. $\square$

A rooted tree T may have more than one median but it has a unique vertex v with the smallest $\delta_T(v) > 0$, and we call v the median, denoted as $m(T)$. This lemma basically says that the median gives the most balanced partition of the tree into two parts. We can find $m(T)$ by starting from the root and going to the heaviest subtree each time, as vertices elsewhere have negative δ_T values. If a vertex v has at least two heaviest subtrees, then all its children have negative δ_T values, so we don't need to go down further, and v is the median. Let $P(T)$ denote such a path on T . Clearly, $m(T)$ can be found this way in linear time.

Next, consider the 2-median problem on trees. We will root T at $m = m(T)$ throughout the paper unless mentioned otherwise. The idea is that with (m_1, m_2) being a 2-median of a tree T , those vertices closer to m_1 and those others divide T into two subtrees with m_1 and m_2 being their respective 1-median. Let $X(e)$ and $Y(e)$ denote the two resulting subtrees when edge e is deleted, with $X(e)$ being the one containing m . This motivates the following useful strategy called the *link-deletion method* [1, 10]:

Step 1. Find $m_Y(e) = m(Y(e))$ for every $e \in E(T)$.

Step 2. Find $m_X(e) = m(X(e))$ for every $e \in E(T)$.

Step 3. Output the pair $(m_X(e), m_Y(e))$ with smallest $Sum_{Y(e)}(m_Y(e)) + Sum_{X(e)}(m_X(e))$

Gavish and Sridhar had an $O(n \log n)$ -time algorithm, which runs Step 1 and 3 in $O(n)$ time but needs $\Theta(n \log n)$ time for Step 2. Let T_α and T_β be m 's two heaviest subtrees, with a tie broken arbitrarily. They observed a property that $m_X(e) \in P(T_\beta)$ if $e \in T_\alpha$ and $m_X(e) \in P(T_\alpha)$ otherwise. Let $J_T(e)$ denote that path containing $m_X(e)$.

For vertices x and y , define their bisector $BS(x, y)$ as the edge containing the position with equal distance to x and y , which has the following property.

Property 2.1 [13] For $e = BS(x, y)$ and $x \in X(e)$, $Sum_T(x, y) = Sum_{X(e)}(x) + Sum_{Y(e)}(y)$.

Sometimes it makes sense to consider arbitrary positions, called points, on edges, instead of just vertices. Some definitions above can be easily extended, such as the distance of two points and the distance-sum function. For a point $r \notin V(T)$, $\delta_T(r)$ is defined as the weight difference between the two subtrees separated by r . This relaxation does not reduce the distance-sum of a 1-median, as moving it to some vertex on that edge still preserves the distance-sum. This is also true for a 2-median, as the two vertices are the respective 1-medians of $X(e)$ and $Y(e)$ for some edge e . However, this is not the case for the generalized 2-median problems.

Later we will encounter some tasks which can be formulated as the following *tree marker problem*. The input is a tree T and a set of $O(n)$ 3-tuples (v, l, u) with $v, u \in V(T)$ and $0 \leq l \leq d(v, u)$, and for each 3-tuple (v, l, u) , we output the point p on $path(v, u)$ with $d(v, p) = l$. We have the following, which is proved in Appendix A, assuming that sorting $O(n)$ edge lengths takes t_e time.

Theorem 2.1 The tree marker problem can be solved in $O(n+t_e) = O(n \log n)$ time.

3 The First Generalization

The problem studied in this section is to find vertices r_1 and r_2 on T with $d(r_1, r_2) \leq l_d$, for a given l_d , that minimize $Sum_T(r_1, r_2)$. We will follow the framework of the link-deletion method. For an edge e , let $r_X(e)$ and $r_Y(e)$ denote the respective vertices on $X(e)$ and $Y(e)$ with $d(r_X(e), r_Y(e)) \leq l_d$ that achieve the smallest distance-sum. Root T at $m = m(T)$. The following lemma helps cut down the search space for $r_X(e)$ and $r_Y(e)$.

Lemma 3.1 For $e = (v, par(v))$, $r_Y(e) \in path(v, m_Y(e))$ and $r_X(e) \in path(m_X(e), par(v))$.

Proof: Moving a vertex r on $path(v, m_Y(e))$ away from the path to a descendant r' increases its distance to v and increases the distance-sum. So $r_Y(e) \in path(v, m_Y(e))$. Similarly, one can show that $r_X(e) \in path(m_X(e), par(v))$. $\square$

For an edge e , let y_e denote the vertex farthest from $m_X(e)$ on $path(v, m_Y(e))$ with $d(m_X(e), y_e) \leq l_d$. Note that for any edge $e = (v, par(v))$ and vertex $y \in Y(e)$, $path(par(v), m_X(e)) \subseteq path(y, m) \cup J_T(y, par(y))$. So for a vertex $y \neq m$, let x_y denote the vertex farthest from y on $path(y, m) \cup J_T(y, par(y))$ with $d(y, x_y) \leq l_d$. Instead of finding all those $(r_X(e), r_Y(e))$, we produce the following two sets of pairs:

$$S_1 = \{(m_X(e), y_e) \mid e \in E(T)\} \text{ and } S_2 = \{(x_y, y) \mid y \in V(T)\}.$$

The following lemma shows that they are good enough for our purpose.

Lemma 3.2 For every $(r_X(e), r_Y(e))$, there is a pair $(x, y) \in S_1 \cup S_2$ such that $Sum_T(x, y) \leq Sum_{X(e)}(r_X(e)) + Sum_{Y(e)}(r_Y(e))$.

Proof: Let $e = (v, par(v))$. If $d(m_X(e), r_Y(e)) \leq l_d$, y_e is the nearest vertex to $m_Y(e)$ on $path(v, m_Y(e))$, so $Sum_T(m_X(e), y_e) \leq Sum_{X(e)}(m_X(e)) + Sum_{Y(e)}(y_e) \leq Sum_{X(e)}(r_X(e)) +$

$Sum_{Y(e)}(r_Y(e))$. Otherwise, for $y=r_Y(e)$, x_y is the nearest vertex to $m_X(e)$ on $path(m_X(e), par(v))$, so $Sum_T(x_y, y) \leq Sum_{X(e)}(x_y) + Sum_{Y(e)}(y) \leq Sum_{X(e)}(r_X(e)) + Sum_{Y(e)}(r_Y(e))$. $\square$

$S_1 \cup S_2$ has at most $2n$ pairs and finding them can be reduced to the tree marker problem. For each $(x_y, y) \in S_2$, we also need to find $e=BS(x_y, y)$ in order to compute $Sum_T(x_y, y) = Sum_{X(e)}(x_y) + Sum_{Y(e)}(y)$, according to Property 2.1. Finding all such bisectors can again be reduced to the tree marker problem. So we have following, assuming that getting all $m_X(e)$ takes t_x time and sorting n edge lengths takes t_e time.

Theorem 3.1 The first generalization can be solved in $O(n+t_x+t_e) = O(n \log n)$ time.

4 The Second Generalization

Recall that the eccentricity of a set S is defined as $Ecc_T(S) = \max_{v \in V(T)} d(v, S)$. For notational convenience, we write $Ecc_T(v)$ for $Ecc_T(\{v\})$ and $Ecc_T(u, v)$ for $Ecc_T(\{u, v\})$. The diameter of T , denoted as $dm(T)$, is the longest path on T . The center of T , denoted as $c(T)$, is a vertex u such that $Ecc_T(u) \leq Ecc_T(v)$ for every vertex v on T . It is not hard to verify that $c(T)$ is also a center of $dm(T)$, and $dm(T)$ must end at a farthest leaf of T , with $c(T)$ being its ancestor.

The problem to study now is finding two vertices $r_1, r_2 \in V(T)$ with $Ecc_T(r_1, r_2) \leq l_e$, for a given l_e , that minimize $Sum_T(r_1, r_2)$. According to Property 2.1, $Sum_T(r_1, r_2) = Sum_{X(e)}(r_1) + Sum_{Y(e)}(r_2)$ for $e=BS(r_1, r_2)$ with $r_1 \in X(e)$ and $r_2 \in Y(e)$, and now $Ecc(r_1, r_2) = \max\{Ecc_{X(e)}(r_1), Ecc_{Y(e)}(r_2)\}$. So again we can follow the link-deletion method. For an edge e , define $r_X(e)$ as a vertex on $X(e)$ with $Ecc_{X(e)}(r_X(e)) \leq l_e$ that minimizes $Sum_{X(e)}(r_X(e))$, and define $r_Y(e)$ on $Y(e)$ accordingly. We will find all such $(r_X(e), r_Y(e))$ and output the pair with the smallest distance-sum. The following lemma tells us where to find $r_Y(e)$ and $r_X(e)$.

Lemma 4.1 $r_X(e) \in path(m_X(e), C(X(e)))$ and $r_Y(e) \in path(m_Y(e), C(Y(e)))$.

This lemma holds as moving away from either path never decreases the eccentricity or the distancesum. Then $r_X(e)$ is the vertex on $path(m_X(e), C(X(e)))$ nearest to $m_X(e)$ with $Ecc_{X(e)}(r_X(e)) \leq l_e$, and similarly for $r_Y(e)$. We obtain all those $m_X(e)$ and $m_Y(e)$ first.

Then, we will find every $dm(Y(e))$ and $c(Y(e))$, or equivalently $dm(T_v)$ and $c(T_v)$ for $e=(v, par(v))$. There are three possibilities for $dm(T_v)$. If $dm(T_v)=dm(T_u)$ for some child u of v , then $c(T_v)=c(T_u)$. If $dm(T_v)$ has one end at v , then v has only one child u and $dm(T_v)$ must ends at u 's farthest leaf, so $c(T_v) \in path(v, c(T_u))$. Otherwise, $dm(T_v)$ is the path through v connecting v 's two farthest leaves, then $c(T_v) \in path(v, c(T_u))$ with T_u being v 's subtree containing v 's farthest leaf. So $c(T_v)$ either stays on or moves up from the center of some subtree T_u of v . In a bottom-up way, we can determine which case happens and find $dm(T_v)$ and $c(T_v)$ for every v in linear time.

Next, let's find $r_Y(e)$ from $path(m_Y(e), c(Y(e)))$ for every $e=(v, par(v))$. Let u be the child of v with largest $w(T_u)$. Let $e'=(u, v)$ and $r=r_Y(e')$. If r is not $m_Y(e')$'s ancestor,

there must be some leaf l of T_u dragging r from moving up with $d(\text{par}(r), l) > l_e$, so $r_Y(e)$ must stay at r . Checking this condition for every e in a bottom-up way takes linear time. Otherwise, r is $m_Y(e')$'s ancestor. Since $m_Y(e)$ is $m_Y(e')$'s ancestor, $r_Y(e)$ is the first vertex r' with $\text{Ecc}_{Y(e)}(r') \leq l_e$ on the path from r to $c(Y(e))$. If $r_Y(e)$ is an ancestor of r , then $r_Y(e)$ moves up from $r = r_Y(e')$. Otherwise, $r_Y(e)$ is not an ancestor of r , and thus not an ancestor of $m_Y(e)$. According to the same argument above, $r_Y(e'') = r_Y(e)$ for every e'' above e , so the path from $\text{LCA}(r, c(Y(e)))$ to $c(Y(e))$ need not be checked again later. Since each vertex is checked only once, the total time is linear.

Finally, let's find $c(X(e))$, and then $r_X(e) \in \text{path}(m_X(e), c(X(e)))$, for every e . Let $F = \text{path}(m, l_1) \cup \text{path}(m, l_2)$, where $dm(T) = \text{path}(l_1, l_2)$. First, consider those $e \notin F$. Clearly, $dm(X(e)) = dm(T)$ and $c(X(e)) = c(T)$. If $\text{Ecc}_{X(e)}(m_X(e)) \leq l_e$, then $r_X(e) = m_X(e)$. Otherwise, $r_X(e)$ is the vertex $r \in J_T(e) \cup \text{path}(c(T), m)$ nearest to $m_X(e)$ with $\text{Ecc}_T(r) \leq l_e$, which is always one of the at most two (only one when $c(T) \notin J_T(e)$) vertices $r \in J_T(e) \cup \text{path}(c(T), m)$ farthest from $c(T)$ with $\text{Ecc}_T(r) \leq l_e$. Finding all such $r_X(e)$ takes linear time. Next we consider those $e \in F$. We can change T 's root to one end of $dm(T)$ so that each $X(e)$ appears as a subtree T_v with $e = (v, \text{par}(v))$. Then all such $r_X(e)$ can be found using the same bottom-up way as that for $r_Y(e)$. Hence, we have the following theorem, assuming that finding all $m_X(e)$ takes t_x time.

Theorem 4.1 The second generalization can be solved in $O(n + t_x) = O(n \log n)$ time.

5 Linear Time Algorithms

Now we study cases when linear-time algorithms exist for the 2-median problem and the two generalizations. A common obstacle is to find all $m_X(e)$'s, which has the following lower bound, proved in Appendix B.

Theorem 5.1 Without assumption on vertex weights, finding all those $m_X(e)$ requires $\Omega(n \log n)$ time in the comparison model, even when every edge has a unit length.

This suggests two possibilities for a more efficient algorithm. The first is to avoid finding all $m_X(e)$'s, and we consider this assuming that edge lengths are at most $k = n^{O(1)}$. Suppose that we have obtained every $m_Y(e)$. According to Property 2.1, if $(m_X(e), m_Y(e))$ is a 2-median, then $e = BS(m_X(e), m_Y(e))$. So to find $m_X(e)$, only the interval of vertices v on $J_T(e)$ satisfying $e = BS(v, m_Y(e))$ needs to be considered. Sometimes we may fail to find $m_X(e)$ there, but it doesn't matter as such $(m_X(e), m_Y(e))$ is not a 2-median. Let $\text{path}(\text{start}(e), \text{end}(e))$ denote this interval with $\text{start}(e)$ being the one closer to e . After knowing the distance of every vertex to the root, finding all $(\text{start}(e), \text{end}(e))$'s takes $O(n)$ time by an integer sorting as $k = n^{O(1)}$ is assumed. One can check that each interval has length at most $2d(e) \leq 2k$ and contains at most $O(k)$ vertices, so $m_X(e)$ can be found in $O(\log k)$ time using a binary search. So the 2-median problem takes $O(n \log k)$ time. A similar idea works for the generalized problem in Section 4 too. The bottleneck there is to find $m_X(e)$ and then $r_X(e)$ for those $e \notin F$. Now we don't need $m_X(e)$. From Property 2.1, if $(r_X(e), r_Y(e))$ is a solution, then

$e=BS(r_X(e), r_Y(e))$, so we search $r_X(e)$ from the interval on $path(m, c(T)) \cup J_T(e)$ satisfying this condition. Thus, we have the following.

Theorem 5.2 Both the 2-median problem and the generalization in Section 4 can be solved in $O(n \log k)$ time when edge lengths are at most $k=n^{O(1)}$. In particular, it takes linear time for constant k .

The second possibility is having some restriction on vertex weights, and we consider the case with vertex weights being polynomially bounded integers. Again, we root T at $m(T)$. For each $e=(v, par(v))$, $m_X(e)$ is the vertex $u \in X(e)$ with the smallest $\delta_{X(e)}(u)=\delta_T(u)+w(T_v)>0$, or equivalently the smallest $\delta_T(u)>-w(T_v)$, according to Lemma 2.1. For those $e \notin P(T_\alpha)$, $m_X(e) \in P(T_\alpha)$, and we can find all such $m_X(e)$ by sorting the set $\{-w(T_v) \mid v \in P(T_\alpha)\} \cup \{\delta_T(u) \mid u \in P(T_\alpha)\}$. For those $e \in P(T_\alpha)$, $m_X(e) \in P(T_\beta)$, and we can find all such $m_X(e)$ by sorting the set $\{-w(T_v) \mid v \in P(T_\alpha)\} \cup \{\delta_T(u) \mid u \in P(T_\beta)\}$. All those $\delta_T(u)$ and $w(T_v)$ can be computed in linear time via a bottom-up way. So we have the following.

Theorem 5.3 Both the 2-median problem and the generalization in Section 4 can be solved in linear time when vertex weights are polynomially bounded integers.

Section 3's algorithm also sorts edge lengths, so we only have the following.

Theorem 5.4 The generalized problem in Section 3 can be solved in linear time when all edge lengths and vertex weights are polynomially bounded integers.

6 Relaxation from Vertices to Points

In this section, we consider the relaxation by allowing 2-medians on points instead of just on vertices. We know that without constraints on 2-medians, this relaxation does not give smaller distance-sum. So we consider the two generalized problems discussed in Section 3 and 4.

All the properties in Section 4 still hold with respect to such relaxation, so the same algorithm also works here, and we have the following theorem.

Theorem 6.1 The relaxed problem of the second generalization needs $O(n \log n)$ time.

Some work is needed for the first generalization. Our algorithm again is based on the link-deletion method, but now there is a new possibility that the pair of a 2-median lie on the same edge. So we first find two points $p_1(e)$ and $p_2(e)$ on each edge e with $d(p_1(e), p_2(e)) \leq l_d$ that minimize the distance-sum. After that, for each edge e , we want to find a point $r_X(e)$ on $X(e)$ and a point $r_Y(e)$ on $Y(e)$ with $d(r_X(e), r_Y(e)) \leq l_d$ that achieve the smallest distance-sum. For the reason similar to that of Lemma 3.1, we know that for any $e=(v, par(v))$, $r_X(e)$ is on $path(m_X(e), par(v))$ and $r_Y(e)$ is on $path(v, m_Y(e))$.

We can assume $d(e) \leq l_d$, as $(r_X(e), r_Y(e))$ does not exist otherwise, and $d(m_X(e), m_Y(e)) > l_d$, as $(r_X(e), r_Y(e)) = (m_X(e), m_Y(e))$ otherwise. We can also assume that $d(r_X(e), r_Y(e)) = l_d$ as otherwise moving $r_X(e)$ and $r_Y(e)$ apart further decreases the distance-sum. For convenience, let's say that $m_X(e)$, $par(v)$, v , and $m_Y(e)$ are lined up from left to right. Consider what happens when moving on $path(m_X(e), par(v))$ from a point r_1 to a point r_1' with a small fixed distance Δ . Moving r_1 left decreases $Sum_{X(e)}$ by a positive amount $dec(r_1)$, and $dec(r_1)$ gets smaller as r_1 comes closer to $m_X(e)$ because the partition of $X(e)$ becomes more balanced. Moving r_1 right increases $Sum_{X(e)}$ by a positive amount $inc(r_1)$, and $inc(r_1)$ gets larger as r_1 moves closer to $par(v)$. When moving a point r_2 on $path(v, m_Y(e))$, similar phenomena occur with the difference that moving r_2 left now increases $Sum_{Y(e)}$. Note that $dec(r) \leq inc(r)$ for any point r .

We will do a binary search for $(r_X(e), r_Y(e))$ in iterations, each time updating two intervals I_X and I_Y for $r_X(e)$ and $r_Y(e)$ respectively, with $I_X = path(m_X(e), par(v))$ and $I_Y = path(v, m_Y(e))$ initially. After I_X and I_Y both reduce to one edge each, $r_X(e)$ and $r_Y(e)$ can then be found easily. In each iteration, we select vertices r_1 and r_2 to cut I_X and I_Y each into two halves of almost equal number of vertices. Two cases to consider:

$dec(r_1) \leq inc(r_2)$. Suppose $d(r_1, r_2) \geq l_d$. Let r_2' denote the point on $path(v, m_Y(e))$ with $d(r_1, r_2') = l_d$, and note that $inc(r_2') \geq inc(r_2)$. Moving (r_1, r_2') left for a distance b increases the distance-sum by at least $(inc(r_2') - dec(r_1))b/\Delta \geq 0$, so we can ignore the half to the left of r_1 for $r_X(e)$, and update I_X to be the remaining half. Otherwise suppose $d(r_1, r_2) < l_d$. Let r_1' denote the point on $path(m_X(e), u)$ with $d(r_1', r_2) = l_d$, and note that $dec(r_1') \leq dec(r_1)$. Moving (r_1', r_2) left for a distance b increases the distance-sum by at least $(inc(r_2) - dec(r_1'))b/\Delta \geq 0$, so we can ignore the half to left of r_2 for $r_Y(e)$, and update I_Y to be the remaining half.

$dec(r_1) > inc(r_2)$. Note that $dec(r_2) \leq inc(r_2) < dec(r_1) \leq inc(r_1)$. Thus, this is just case 1 with r_1 and r_2 switched, and a similar action can be taken.

So we first find all $(p_1(e), p_2(e))$ in $O(n)$ time, and all $(m_X(e), m_Y(e))$ in $O(n \log n)$ time. Then we use binary searches as described above to find $r_X(e)$ and $r_Y(e)$ for every edge e , but with all the $n-1$ binary searches carried out in parallel. During each iteration, we find all $n-1$ middle vertices at once by reducing the task to the tree maker problem with unit edge length, which only takes $O(n)$ time. There are at most $\log n$ iterations, so $O(n \log n)$ time suffices. Finally we choose the pair $(p_1(e), p_2(e))$ or $(r_X(e), r_Y(e))$ with the smallest distance-sum. So we have the following theorem.

Theorem 6.2 The relaxed problem of the first generalization needs $O(n \log n)$ time.

7 Future Work

For the p -median problem on trees, Tamir [12] has an $O(pn^2)$ -time algorithm. Can it be solved in $o(n^2)$ time for any $p > 2$? We have an $\Omega(n \log n)$ -time lower bound for the link-deletion method, but we are more interested in a lower bound for the 2-median problem and its generalizations. Can they be solved in $o(n \log n)$ time? Wang *et al.* [13] solved both generalizations in $O(\log n)$ parallel time but with $O(n^2)$ work. Can one design more efficient parallel algorithms for them?

References

1. B. Gavish and S. Sridhar, Computing the 2-median on tree networks in $O(n \lg n)$ time, *Networks*, vol. 26, pp. 305-317, 1995.
2. S. L. Hakimi, Optimal distribution of switching centers in communication networks and some related graph theoretical problems, *Operations Research*, vol. 13, pp. 462-475, 1965.
3. H. N. Gabow and R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, *Journal of Computer and System Sciences*, vol. 30, pp. 209-221, 1985.
4. Z. Galil and G. F. Italiano, Data structures and algorithms for disjoint set union problems, *ACM Computing Surveys*, vol. 23, no. 3, pp. 319-344, 1991.
5. A. J. Goldman, Optimal center location in simple networks, *Transportation science*, vol. 5, pp. 212-221, 1971.
6. G. Y. Handler and P. Mirchandani, *Location on Networks*, MIT Press, Cambridge, MA, 1979.
7. J. JáJá, *An Introduction to Parallel Algorithm*, Addison-Wesley, Reading, MA, 1992.
8. O. Kariv and S. L. Hakimi, An algorithmic approach to network location problems, Part II: The p -medians, *SIAM Journal on Applied Mathematics*, vol. 37, pp. 539-560, 1979.
9. S.-C. Ku, *Algorithms for location problems on trees*, Ph.D. thesis, Computer Science Department, National Tsing Hua University, 2001.
10. P. B. Mirchandani and A. Oudjit, Localizing 2-medians on probabilistic and deterministic tree networks, *Networks*, vol. 10, pp. 329-350, 1980.
11. B. C. Tansel, R. L. Francis and T. J. Lowe, Location of networks: A survey, *Management Science*, vol. 29, pp. 482-511, 1983.
12. A. Tarmir, An $O(pn^2)$ algorithm for the p -median and related problems on tree graphs, *Operations Research Letters*, vol. 19, pp. 59-64, 1996.
13. B.-F. Wang, S.-C. Ku, and K.-H. Shi, Cost-optimal parallel algorithms for the tree bisector problem and applications, *IEEE Transactions on Parallel and Distributed Systems*, accepted.
14. B.-F. Wang, S.-C. Ku, K.-H. Shi, T.-K. Hung, and P.-S. Liu, Parallel algorithms for the tree bisector problem and applications, in *Proceedings of the 1999 International Conference on Parallel Processing*, pp. 192-199, 1999.
15. B.-F. Wang, Efficient parallel algorithms for optimally locating a path and a tree of a specified length in a weighted tree network, *Journal of Algorithms*, Vol. 34, pp. 90-108, 2000.

Appendix A. Proof of Theorem 2.1

Let's do some preprocessing first. Root the tree at an arbitrary vertex r , and find the least common ancestor for every vertex pair, which takes only linear time. Then, replace each 3-tuple (x, l, y) by the pair $(v, t(v))$ which is $(x, d(x, r)-l)$ if $l \leq d(x,$

$LCA(x,y)$) and $(y, d(y, r)-l)$ otherwise, indicating that the point we want is above v with distance $t(v)$ to r . For simplicity, we assume that no vertex v appears in two pairs; otherwise, we can simply duplicate the vertex v on T .

Now we borrow the idea from Wang *et al.*'s algorithm for the tree bisector problem [12, 13], which in fact is a special case of the tree marker problem. Consider the Euler tour U of T , and let $U[i]$ denote the i -th vertex in the tour. For a pair $(v, t(v))$, we want to find v 's highest ancestor u with $d(u, r) > t(v)$. Let $w = \text{par}(u)$. Note that $d(w, r) \leq t(v)$ and $d(u', r) > t(v)$ for every vertex u' in the subtree T_u . Also note that in U , there is an appearance of w right before the Euler tour of the subtree T_u . If we go backward from the first appearance of v in U , w is first vertex we encounter with $d(w, r) \leq t(v)$. Let F and X be the arrays with $F[i] = t(U[i])$ and $X[i] = d(r, U[i])$. Now the remaining task can be easily reduced to the following *f-left-match problem*, introduced by Wang *et al.* [13]. The input is two arrays X and F , each of n numbers. The output is an array M such that $M[k]$ is the largest index $l < k$ with $X[l] \leq F[k]$ and $M[k] = \emptyset$ if no such index exists. $X[M[k]]$ is called $F[k]$'s *target element*.

Given input X and F , we first stable-sort their concatenation increasingly into a new array W . Let Z be the array of $2n$ elements with $Z[i] = 1$ if $W[i]$ is from X and $Z[i] = 0$ otherwise. Let D be the array of $2n$ elements with $D[i]$ being the index of $W[i]$ in its original array, X or F . An example is shown in Figure A.1. Using some data structure S , we do the following for index i from 0 to $2n-1$. If $Z[i] = 1$, insert $D[i]$ into S . Otherwise, search the largest number $l < D[i]$ in S and set $M[D[i]] = l$. This assignment is correct because at this point, S contains exactly those indices l such that $X[l] < F[D[i]]$, or $X[l] = F[D[i]]$ but $l < D[i]$, due to the stable sorting.

	0	1	2	3	4	5	6	7	8
X	1	3	8	5	7				
F	*	2	4	2	6				
M	*	0	1	0	3				
W	1	2	2	3	4	5	6	7	8
Z	1	0	0	1	0	1	0	1	1
D	0	1	3	1	2	3	4	4	2

*: Don't care

Fig. A.1. An illustration of the arrays M , W , Z , and D from the arrays X and F

The complexity depends on how S is implemented. This is related to the *interval split-find problem* introduced in [4], as the set of inserted numbers is in fact $I_n = \{0, 1, \dots, n-1\}$. An *interval* of I_n is a set of consecutive numbers in I_n . The problem is to maintain a data structure that represents some partition of I_n and supports two types of operations: *find*(x), which returns the interval containing x , and *split*(x), which splits the interval of x into two, one for those less than x and one for the rest. Gabow and Tarjan [3] had an $O(n)$ -time algorithm for $O(n)$ operations, which we will build our data structure S upon. Initially we have only one interval I_n . The idea is that if we insert each $D[j]$ in S by calling *split*($D[j]$), then given $D[i]$, the largest $D[j] < D[i]$ in S is just the smallest number in the interval containing $D[i]$. The only thing we need to take care is to record the smallest number of each interval, using an array SS updated

in the following way. Whenever a $D[j]$ is to be inserted, let $s = SS[find(D[j])]$, call $split(D[j])$, set $SS[find(D[j]-1)] = s$, and set $SS[find(D[j])] = D[j]$. There are $O(n)$ operations and only linear time is needed. Thus, we have the following lemma, assuming t_a is the time for stable-sorting the concatenation of X and F , which immediately implies Theorem 2.1.

Lemma A.2 The f -left-match problem can be solved in $O(n+t_a)$ time.

Appendix B. Proof of Theorem 5.1

Consider the *multi-search problem* defined by Wang *et al.* [12]. The input is an array P of p unsorted numbers and an array Q of q distinct sorted numbers. The output is an array R of p integers, with $R[i]$, for $1 \leq i \leq p$, recording the number of elements in Q smaller than $P[i]$. Wang *et al.* proved an $\Omega(p \log q)$ -time lower bound in the comparison model for this problem. We will reduce this problem in linear time to the link-deletion method.

Let $s = \sum_{1 \leq i \leq p} P[i]$, and we can assume w.l.o.g. that $s > Q[q]$, as otherwise an entry of a large value can be added to P . Construct a tree T of $p+q+2$ vertices with unit edge lengths in the following way. T has a root z with $w(z) = s$. For $1 \leq i \leq p$, there is an edge e_i connecting z to a leaf v_i with $w(v_i) = s - P[i]$. There is a path from a leaf u_0 to z , passing through vertices $u_1, u_2, \dots, u_q$. Let $w(u_0) = (p-1)s + Q[q]/2$, $w(u_1) = Q[1]/2$, and $w(u_i) = (Q[i] - Q[i-1])/2$ for $2 \leq i \leq q$.

Let H denote the subtree of z rooted at u_q . Note that $w(T \setminus H) = w(z) + \sum_{1 \leq i \leq p} w(v_i) = s + (p-1)s = ps$, and $w(H) = \sum_{0 \leq i \leq q} w(u_i) = (p-1)s + Q[q] < ps$; so the median of T must lie on $T \setminus H$. As each v_i has negative $\delta_T(v_i)$, z is the median of T . H is the heaviest subtree of T , so for $1 \leq i \leq p$, $m_X(e_i)$ must lie on H if edge e_i is deleted. Now $rem(e_i) = w(z) - w(v_i) + \sum_{1 \leq j \leq p} w(v_j) = s - (s - P[i]) + (p-1)s = (p-1)s + P[i]$, for $1 \leq i \leq p$, and $\delta_H(u_k) = \sum_{0 \leq j \leq k} w(u_j) - \sum_{k < j \leq q} w(u_j) = ((p-1)s + Q[q]/2 + Q[k]/2) - (Q[q]/2 - Q[k]/2) = (p-1)s + Q[k]$, for $1 \leq k \leq q$. As $m_X(e_i)$ is the vertex u_k on H with the smallest nonnegative $\delta_H(u_k) - rem(e_i) = Q[k] - P[i]$, it gives the smallest k with $Q[k] \geq P[i]$. So finding $m_X(e_i)$ for every edge e_i solves the multi-search problem on the input P and Q .

Given any two arrays P and Q , we can construct the corresponding tree T in linear time. As the multi-search problem with $p=q=n/2$ has an $\Omega(n \log n)$ -time lower bound, so does the link deletion method.

Author Index

Albert, Michael H.	355	Goldberg, Andrew V.	502
Aldred, Robert E. L.	355	Gramm, Jens	441
Asano, Tetsuo	636		
Aspnes, James	403	Hartling, Julia	403
Atkinson, Mike D.	355	Hashimoto, Kenta	99
		Holton, Derek A.	355
Babel, Luitpold	695	Hori, Hideaki	719
Becker, Ronald I.	743	Horiyama, Takashi	231
Biedl, Therese	308	Huang, Jingui	48
Bodlaender, Hans L.	550	Hürlimann, Daniel	390
Bremner, David	623	Hurtado, Ferran	623
Breutzmann, Josef M.	219		
Brodal, Gerth Stølting	731	Ibaraki, Toshihide	231, 279, 682
Brodnik, Andrej	599	Ibarra, Oscar H.	244
		Ishii, Toshimasa	124
Chen, Jianer	48	Iturriaga, Claudia	649
Chen, Bo	380, 695		
Chen, Danny Z.	332, 454, 467	Jakoby, Andreas	562
Chiang, Yen-I	743	Juedes, David W.	219
Chin, Francis Y. L.	172		
Chwa, Kyung-Yong	74, 480	Kao, Ming-Yang	403, 429
		Karlsson, Johan	599
Dang, Zhe	244	Karuno, Yoshiyuki	36
Demaine, Erik D.	308	Kellerer, Hans	695
Deng, Xiaotie	26, 380	Khoussainov, Bakhadyr	550
Dinneen, Michael J.	550	Kim, Jae-Hoon	74
Duncan, Christian A.	308	Kim, Junhyong	403
		Kobourov, Stephen G.	308
Erlebach, Thomas	390	Koga, Hisashi	61
Fagerberg, Rolf	731	Koshiba, Takeshi	268
Fellows, Michael R.	291	Kotov, Vladimir	695
Feng, Haodi	26	Kovaleva, Sofia	493
Fiala, Jiří	537	Kranakis, Evangelos	756
Fleischer, Rudolf	308	Kratochvíl, Jan	537
Fujito, Toshihiro	670	Krizanc, Danny	756
Fung, Stanley P. Y.	172	Ku, Shan-Chyun	768
		Kurosawa, Kaoru	208
Galbiati, Giulia	116		
Galluccio, Anna	345	Lari, Isabella	743
Gantenbein, Martin	390	Lee, Hyunyoung	587
Garrido, Mari Ángeles	649	Lee, Jae-Ha	480
		Lin, Tzu-Chin	768

- Lin, Xuemin 575
 Liśkiewicz, Maciej 562
 Lu, Chi-Jen 768
 Luan, Shuang 454
 Lutz, Jack H. 219
 Lyngsø, Rune B. 416

 Márquez, Alberto 649
 Mizuki, Takaaki 196
 Monostori, Krisztián 707
 Murota, Kazuo 14

 Nagamochi, Hiroshi ... 36, 124, 682
 Nagoya, Takayuki 136
 Nakano, Shin-ichi 107
 Neyer, Gabriele 390
 Niedermeier, Rolf 441
 Nikolopoulos, Stavros D. 320
 Nishizeki, Takao 196, 514
 Nomura, Kumiko 86

 Okumura, Tsuyoshi 670
 Ono, Hirotaka 279

 Pagourtzis, Aris 390
 Palios, Leonidas 320
 Park, Sang-Min 480
 Pedersen, Christian N. S. . 416, 731
 Peinado, Marcus 525
 Penna, Paolo 390
 Poon, Sheung-Hung 610
 Portillo, José Ramón 649
 Proietti, Guido 345

 Ramachandran, Vijay 429
 Ramaswami, Suneeta 623
 Reif, John 160
 Reyes, Pedro 649
 Rossmanith, Peter 441

 Sacristán, Vera 623
 Sadakane, Kunihiko 148
 Schindelhauer, Christian 185
 Schlude, Konrad 390
 Scozzari, Andrea 743

 Shah, Gauri 403
 Shende, Sunil 756
 Shimozono, Shinichi 719
 Shin, Chan-Su 610
 Shinohara, Ayumi 719
 Spieksma, Frits C. R. 493
 Steinhöfel, Kathleen 390
 Strijk, Tycho 610
 Sugawara, Norito 148
 Sun, Zheng 160

 Takeda, Masayuki 719
 Tamura, Akihisa 14
 Taylor, David Scot 390
 Tokuyama, Takeshi 148, 636

 Ueno, Shuichi 86, 99
 Uno, Takeaki 367

 Vajk, István 707

 Wang, Biing-Feng 768
 Wang, Jie 467
 Weber, Birgitta 185
 Welch, Jennifer L. 587
 Whitesides, Sue 1
 Widmayer, Peter 390
 Wolff, Alexander 610, 649
 Wu, Xiaodong 332, 467

 Xu, Jinhui 454

 Yagiura, Mutsunori 279
 Yamada, Toshinori 86, 99
 Yamamoto, Hiroaki 257
 Yamamura, Akihiro 208

 Zang, Wenan 380
 Zaslavsky, Arkady 707
 Zhang, Louxin 660
 Zhang, Pixing 26
 Zhao, Liang 682
 Zhou, Xiao 514
 Zhu, Hong 26
 Zhu, Song 660