Lecture Notes in Computer Science 1741
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
*Berlin*
*Heidelberg*
*New York*
*Barcelona*
*Hong Kong*
*London*
*Milan*
*Paris*
*Singapore*
*Tokyo*

Alok Aggarwal C. Pandu Rangan (Eds.)

# Algorithms and Computation

10th International Symposium, ISAAC'99
Chennai, India, December 16-18, 1999
Proceedings

Springer

Series Editors

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

Volume Editors

Alok Aggarwal
IBM, Solutions Research Center, Block 1
Indian Institute of Technology
New Delhi, 110016, India
E-mail: aggarwa@in.ibm.com

C. Pandu Rangan
Department of Computer Science and Engineering
Indian Institute of Technology
Madras, Chennai – 600 036, India
E-mail: rangan@iitm.ernet.in

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

**Algorithms and computation** : 10th international symposium ; proceedings / ISAAC '99, Chennai, India, December 16 - 18, 1999. Alok Aggarwal ; C. Pandu Rangan (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1999
(Lecture notes in computer science ; Vol. 1741)
ISBN 3-540-66916-7

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

ISSN 0302-9743
ISBN 3-540-66916-7 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 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10749965 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

# Preface

This volume contains the proceedings of the $10^{th}$ ISAAC conference (Tenth Annual International Symposium on Algorithms And Computation) held in Chennai, India. This year's conference attracted 71 submissions from as many as 17 different countries. Each submission was reviewed by at least three independent referees. After a week-long e-mail discussion, the program committee agreed to include 40 papers in the conference program. The high acceptance rate is clearly an indication of the quality of the papers received. We thank the program committee members and the reviewers for their sincere efforts.

We were fortunate to have three invited speakers this year, providing for a very attractive program: Kurt Mehlhorn (MPI, Saarbrücken, Germany), Eva Tardos (Cornell University, U.S.A.), and Kokichi Sugihara (Univ. of Tokyo, Japan). Moreover, the conference was preceded by a tutorial on a cutting-edge area, Web Algorithmics by Monika Henzinger (Compaq Systems Research Center, Palo Alto, U.S.A.) as a joint event with FST&TCS 99 (Foundations of Software Technology and Theoretical Computer Science, December 13-15, 1999, Chennai). As a post conference event, a two-day workshop on Approximate Algorithms by R.Ravi (CMU, U.S.A.) and Naveen Garg (IIT, Delhi) was organized. We thank all the invited speakers and special event speakers for agreeing to participate in ISAAC'99.

This year is a special year for the organizing institute Indian Institute of Technology, Madras and for the Computer Science and Engineering department at IIT, Madras; IIT is celebrating its $40^{\text{th}}$ anniversary and it is the Silver Jubilee year for the CSE department. The CSE department at IIT is proud to host this prestigious conference and the allied events. We thank the Director of IIT, Madras, Prof. R. Natarajan, for making available the facilities of the institute for ISAAC related events. Several private companies and educational trusts came forward to extend financial support for ISAAC'99. The generous support from IBM Solutions Research Centre, New Delhi, Jeppiar Educational Trust, Chennai, Jaya Educational Trust, Chennai, Philips Software Centre Pvt. Ltd, Bangalore, Digital and Analog Computing Services (DACS), Bangalore are greatfully acknowledged. The IIT Madras Alumni Association in North America has contributed towards subsidized stay and travel expenses for several Indian participants, enabling them to attend ISAAC'99. We thank the members of organizing committee for making it happen. Special thanks go to the staff of our Institute and to Alfred Hoffman of Springer-Verlag for help with the proceedings.

October 1999

Alok Aggarwal
C. Pandu Rangan

# Tenth Annual International Symposium on Algorithms And Computation

*ISAAC'99*

**December 16-18, 1999, Chennai, India**

## Symposium Chair

- Pandu Rangan C *(IIT, Madras)*

## Program Committee Co-chairs

- Alok Aggarwal *(IBM, India)*
- Pandu Rangan C *(IIT, Madras)*

## Program Committee

- Antonios Symvonis *(U. Sydney, Australia)*
- Aravind Srinivasan *(Bell Labs, U.S.A.)*
- Cai Mao-cheng *(CAS, R.O.C.)*
- Hisao Tamaki *(U. Meiji, Japan)*
- Jung-Heum Park *(Catholic U. Korea, South Korea)*
- Lusheng Wang *(City U. Hong Kong, R.O.C.)*
- Mark Keil *(U. Saskatchewan, Canada)*
- Prabhakar Raghavan *(IBM, U.S.A.)*
- Ramesh Kumar Sitaraman *(U. Massachusetts, U.S.A.)*
- Seinosuke Toda *(U. Nihon, Japan)*
- Tadao Takaoka *(U. Canterbury, New Zealand)*
- Tak-wah Lam *(U. Hong Kong, R.O.C.)*
- Wen-Lian Hsu *(Academia Scinica, Taiwan, R.O.C.)*

Alok Aggarwal *(IBM, India)*

Pandu Rangan C *(IIT, Madras, India)*

## Organizing Committee

- E. Boopal *(IIT, Madras)*
- V. Kamakoti *(ATI Research, Chennai)*
- R. Rama *(IIT, Madras)*
- K. Rangarajan *(MCC, Chennai)*

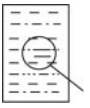
## Advisary Committee

- Francis Chin (U. Hong Kong, R.O.C.)
- Peter Eades (U. Newcastle, Australia)
- Ding-Zhu Du (U. Minnesota, U.S.A.)
- Wen-Lian Hsu (Academia Sinica, Taiwan)
- Toshihide Ibaraki (U. Kyoto, Japan)
- D.T. Lee (Academia Sinica, Taiwan, R.O.C.)
- Takao Nishizeki (Tohoku U., Japan, Chair)

## Sponsors

- Indian Institute of Technology, Madras
- IBM Solutions Research Center, New Delhi
- Jeppiar Educational Trust, Chennai
- Jaya Educational Trust, Thiruninravur
- Philips Software Center Pvt. Ltd., Bangalore
- Digital & Analog Computing Services (DACS), Bangalore

# List of Reviewers



Alexander Russell
Alok Aggarwal
Antonios Symvonis
Aravind Srinivasan
Cai Mao-Cheng
C. Pandu Rangan
Chan-Su Shin
Hee-Chul Kim
Hema A Murthy
Hisao Tamaki
Hwan Gue Cho
Hyeong-Seok Lim
Igor Shparlinski
Jeff Kingston
Jik Hyun Chang
Jung-Heum Park
Kamala Krithivasan
Kamesh Munagala
Kasturi Varadarajan
K.G. Subramanian
Kunsoo Park
K.V. Subrahmanyam
Kyubum Wee
Lusheng Wang
Mahnhoon Lee
Mark Keil
Moses Charikar
Naveen Garg
Peter Schachte
Prabhakar Raghavan
P.S. Subramanian
R. Uehara
Ramesh K. Sitaraman
Richard Brent
S.A. Choudum
S. Arun Kumar
Seinosuke Toda
Soo-Hwan Kim
S.S. Ravi
Stephen Hirst
Sung Kwon Kim
T. Hasunuma
T. Tokuyama
Tadao Takaoka
Tae-Cheon Yang
Tak-Wah Lam
Te. Asano
Trevor Dix
V.S. Subrahmanian
Wen-Lian Hsu
Y. Takenaga

# Table of Contents

**Invited Talk**

The Engineering of Some Bipartite Matching Programs ..... 1
*Kurt Mehlhorn*

**Session 1(a) - Data Structure I**

General Splay: A Basic Theory and Calculus ..... 4
*G.F. Georgakopoulos, D.J. McClurkin*

Static Dictionaries Supporting Rank ..... 18
*Venkatesh Raman, S. Srinivasa Rao*

**Session 1(b) - Parallel & Distributed Computing I**

Multiple Spin-Block Decisions ..... 27
*Peter Damaschke*

Asynchronous Random Polling Dynamic Load Balancing ..... 37
*Peter Sanders*

**Session 2(a) - Approximate Algorithm I**

Simple Approximation Algorithms for MAXNAESP and Hypergraph *2-colarability* ..... 49
*Daya Ram Gaur, Ramesh Krishnamurti*

Hardness of Approximating Independent Domination in Circle Graphs .... 56
*Mirela Damian-Iordache, Sriram V. Pemmaraju*

Constant-Factor Approximation Algorithms for Domination Problems on Circle Graphs ..... 70
*Mirela Damian-Iordache, Sriram V. Pemmaraju*

**Session 2(b) - Computational Intelligence**

Ordered Binary Decision Diagrams as Knowledge-Bases ..... 83
*Takashi Horiyama, Toshihide Ibaraki*

Hard Tasks for Weak Robots: The Role of Common Knowledge in Pattern Formation by Autonomous Mobile Robots ..... 93
*Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, Peter Widmayer*

**Session 3(a) - Online Algorithm**

On-Line Load Balancing of Temporary Tasks Revisited ..... 103
*Kar-Keung To, Wai-Ha Wong*

Online Routing in Triangulations ..... 113
*Prosenjit Bose, Pat Morin*

**Session 3(b) - Complexity Theory I**

The Query Complexity of Program Checking by Constant-Depth Circuits 123
*V. Arvind, K.V. Subrahmanyam, N.V. Vinodchandran*

Tree-Like Resolution Is Superpolynomially Slower Than DAG-Like Resolution for the Pigeonhole Principle ..... 133
*Kazuo Iwama, Shuichi Miyazaki*

**Session 4(a) - Approximate Algorithm II**

Efficient Approximation Algorithms for Multi-label Map Labeling ..... 143
*Binhai Zhu, C.K. Poon*

Approximation Algorithms in Batch Processing ..... 153
*Xiaotie Deng, Chung Keung Poon, Yuzhong Zhang*

**Session 4(b) - Graph Algorithm I**

LexBFS-Ordering in Asteroidal Triple-Free Graphs ..... 163
*Jou-Ming Chang, Chin-Wen Ho, Ming-Tat Ko*

Parallel Algorithms for Shortest Paths and Related Problems on Trapezoid Graphs ..... 173
*F.R. Hsu, Yaw-Ling Lin, Yin-Te Tsai*

**Invited Talk**

Approximation Algorithms for Some Clustering and Classification Problems 183
*Eva Tardos*

**Session 5(a) - Computational Geometry I**

How Many People Can Hide in a Terrain? ..... 184
*Stephan Eidenbenz*

Carrying Umbrellas: An Online Relocation Problem on Graphs ..... 195
*Jae-Ha Lee, Chong-Dae Park, Kyung-Yong Chwa*

**Session 5(b) - Parallel & Distributed Computing II**

Survivable Networks with Bounded Delay: The Edge Failure Case ..... 205
*Serafino Cicerone, Gabriele Di Stefano, Dagmar Handke*

Energy-Efficient Initialization Protocols for Ad-hoc Radio Networks ..... 215
*J.L. Bordim, J. Cui, T. Hayashi, K. Nakano, S. Olariu*

**Session 6(a) - Data Structure II**

Constructing the Suffix Tree of a Tree with a Large Alphabet ..... 225
*Tetsuo Shibuya*

An O(1) Time Algorithm for Generating Multiset Permutations ..... 237
*Tadao Takaoka*

**Session 6(b) - Complexity Theory II**

Upper Bounds for MaxSat: Further Improved ..... 247
*Nikhil Bansal, Venkatesh Raman*

A Linear Time Algorithm for Recognizing Regular Boolean Functions .... 259
*Kazuhisa Makino*

**Session 7(a) - Computational Geometry II**

Station Layouts in the Presence of Location Constraints ..... 269
*Prosenjit Bose, Christos Kaklamanis, Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, David Peleg*

Reverse Center Location Problem ..... 279
*Jianzhong Zhang, Xiaoguang Yang, Mao-cheng Cai*

**Session 7(b) - Algorithms in Practice**

Performance Comparison of Linear Sieve and Cubic Sieve Algorithms for Discrete Logarithms over Prime Fields ..... 295
*Abhijit Das, C.E. Veni Madhavan*

External Memory Algorithms for Outerplanar Graphs ..... 307
*Anil Maheshwari, Norbert Zeh*

**Session 8(a) - Approximate Algorithm III**

A New Approximation Algorithm for the Capacitated Vehicle Routing Problem on a Tree ..... 317
*Tetsuo Asano, Naoki Katoh, Kazuhiro Kawashima*

Approximation Algorithms for Channel Assignment with Constraints ..... 327
*Jeannette Janssen, Lata Narayanan*

**Session 8(b) - Graph Algorithm II**

Algorithms for Finding Noncrossing Steiner Forests in Plane Graphs ..... 337
*Yoshiyuki Kusakari, Daisuke Masubuchi, Takao Nishizeki*

A Linear Algorithm for Finding Total Colorings of Partial $k$-Trees ..... 347
*Shuji Isobe, Xiao Zhou, Takao Nishizeki*

**Invited Talk**

Topology-Oriented Approach to Robust Geometric Computation ..... 357
*Kokichi Sugihara*

**Session 9(a) - Approximate Algorithm IV**

Approximating Multicast Congestion ..... 367
*Santosh Vempala, Berthold Vöcking*

Approximating the Minimum $k$-way Cut in a Graph via Minimum 3-way Cuts ..... 373
*Liang Zhao, Hiroshi Nagamochi, Toshihide Ibaraki*

**Session 9(b) - Parallel & Distributed Computing III**

Online Scheduling of Parallel Communications with Individual Deadlines . 383
*Jae-Ha Lee, Kyung-Yong Chwa*

A Faster Algorithm for Finding Disjoint Paths in Grids ..... 393
*Wun-Tat Chan, Francis Y.L. Chin, Hing-Fung Ting*

**Session 10(a) - Computational Geometry III**

Output-Sensitive Algorithms for Uniform Partitions of Points ..... 403
*Pankaj K. Agarwal, Binay K. Bhattacharya, Sandeep Sen*

Convexifying Monotone Polygons ..... 415
*Therese C. Biedl, Erik D. Demaine, Sylvain Lazard, Steven M. Robbins, Michael A. Soss*

**Session 10(b) - Graph Algorithm III**

Bisecting Two Subsets in 3-Connected Graphs ..... 425
*Hiroshi Nagamochi, Tibor Jordán, Yoshitaka Nakao, Toshihide Ibaraki*

Generalized Maximum Independent Sets for Trees in Subquadratic Time . 435
*B.K. Bhattacharya, M.E. Houle*

**Author Index** ..... 447

# The Engineering of Some Bipartite Matching Programs

Kurt Mehlhorn

Max-Planck-Institut fur Informatik,
Im Stadtwald,
66123 Saarbrucken, Germany,
http://www.mpi-sb.mpg.de/~mehlhorn

Over the past years my research group was involved in the development of three algorithm libraries:

- LEDA, a library of efficient data types and algorithms [LED]
- CGAL, a computational geometry algorithms library [CGA], and
- AGD, a library for automatic graph drawing [AGD].

In this talk I will discuss some of the lessons learned from this work. I will do so on the basis of the LEDA-implementations of bipartite cardinality matching algorithms. The talk is based on Section 7.6, pages 360–392, of [MN99]. In this book Stefan Näher and I give a comprehensive treatment of the LEDA system and its use. We treat the architecture of the system, we discuss the functionality of the data types and algorithms available in the system, we discuss the implementation of many modules of the system, and we give many examples for the use of LEDA.

My personal level of involvement was very different in the three projects: The LEDA project was started in 89 by Stefan Näher and myself and I have been involved as a designer and system architect, implementer of algorithms and tools, writer of documentation and tutorials, and user of the system. For CGAL I acted as an advisor and for AGD my involvement was marginal.

The bipartite cardinality matching problem asks for the computation of a maximum cardinality matching $M$ in a bipartite graph $(A \dot\cup B, E)$. A *matching* in a graph $G$ is a subset $M$ of the edges of $G$ such that no two share an endpoint.

I will discuss the following points:

- Specification: We discuss several specifications of the problem and discuss their relative merits, in particular, with respect to verification and flexibility.
- Checking and verification: We discuss how a matching algorithm can justify its answers and how answers can be checked.
- Representations of matchings: We discuss how matchings can be represented and what the relative merits of the representations are.
- Reinitialization in iterative algorithms: Most matching algorithms work in phases. We discuss how to reinitialize data structures in a cost-effective way.
- Search for augmenting paths by depth-first search or by breadth-first search: We discuss the relative merits of the two methods.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 1–3, 1999.
© Springer-Verlag Berlin Heidelberg 1999

| $n$ | $m$ | $k$ | FFB | dfs– | dfs+ | bfs– | bfs+ | HK– | HK+ | AB– | AB+ | Check |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 4 | 1 | 311.1 | 1.63 | 1.14 | 1.08 | 0.93 | 1.5 | 1.42 | 0.94 | 0.96 | 0.09 |
| 2 | 4 | 50 | 319.2 | 1.24 | 0.65 | 0.71 | 0.58 | 1.09 | 0.99 | 0.7599 | 0.77 | 0.07001 |
| 2 | 4 | 2500 | 316.1 | 0.35 | 0.32 | 0.32 | 0.3 | 1.01 | 0.92 | 0.69 | 0.68 | 0.07001 |
| 2 | 6 | 1 | 404 | 24.06 | 6.72 | 18.97 | 7.1 | 2.35 | 2.24 | 1.29 | 1.26 | 0.1001 |
| 2 | 6 | 50 | 397 | 71 | 15.22 | 12.29 | 7.57 | 1.76 | 1.67 | 0.97 | 0.95 | 0.08997 |
| 2 | 6 | 2500 | 313.9 | 3.15 | 1.12 | 0.6902 | 0.6299 | 2.37 | 2.29 | 0.78 | 0.76 | 0.08008 |
| 2 | 8 | 1 | 364.8 | 7.42 | 2.61 | 7.56 | 3.71 | 2.71 | 2.55 | 2.84 | 2.95 | 0.1199 |
| 2 | 8 | 50 | 360 | 34.15 | 12.47 | 9.35 | 7.59 | 2.19 | 2.1 | 1.68 | 1.64 | 0.09985 |
| 2 | 8 | 2500 | 360.2 | 42.8 | 10.47 | 1.9 | 1.76 | 2.9 | 2.74 | 1.23 | 1.22 | 0.08984 |
| 4 | 8 | 1 | – | 4.43 | 3.23 | 3.03 | 2.64 | 3.3 | 3.05 | 2.47 | 2.47 | 0.1699 |
| 4 | 8 | 50 | – | 2.95 | 1.84 | 1.76 | 1.52 | 2.84 | 2.62 | 1.92 | 1.91 | 0.1501 |
| 4 | 8 | 2500 | – | 0.9202 | 0.6599 | 0.71 | 0.6599 | 2.38 | 2.18 | 1.69 | 1.66 | 0.1501 |
| 4 | 12 | 1 | – | 108 | 27.77 | 87.78 | 29.59 | 5.49 | 5.22 | 3.44 | 3.43 | 0.21 |
| 4 | 12 | 50 | – | 317.5 | 67.39 | 57.71 | 34.76 | 3.86 | 3.65 | 2.62 | 2.53 | 0.1699 |
| 4 | 12 | 2500 | – | 291.2 | 77.17 | 29.31 | 22.05 | 4.19 | 3.92 | 2.09 | 2.05 | 0.1699 |
| 4 | 16 | 1 | – | 23.81 | 9.01 | 26.91 | 10.93 | 5.3 | 4.93 | 2.66 | 2.66 | 0.25 |
| 4 | 16 | 50 | – | 205.2 | 59.7 | 46.32 | 38.94 | 4.92 | 4.62 | 2.27 | 2.31 | 0.2002 |
| 4 | 16 | 2500 | – | 470.3 | 105.3 | 16.62 | 14.71 | 4.46 | 4.23 | 2.09 | 2.1 | 0.1699 |
| 8 | 16 | 1 | – | – | – | 6.27 | 7.57 | 8.28 | 7.75 | 6.76 | 6.61 | 0.37 |
| 8 | 16 | 50 | – | – | – | 4.5 | 4.22 | 5.8 | 5.55 | 5.13 | 5.13 | 0.31 |
| 8 | 16 | 2500 | – | – | – | 1.66 | 1.4 | 4.69 | 4.42 | 4.33 | 4.26 | 0.28 |
| 8 | 24 | 1 | – | – | – | 378.3 | 116.5 | 12.54 | 12.01 | 9.77 | 9.52 | 0.45 |
| 8 | 24 | 50 | – | – | – | 248.2 | 152.6 | 9.94 | 9.51 | 7.39 | 7.32 | 0.36 |
| 8 | 24 | 2500 | – | – | – | 118.2 | 82.81 | 6.8 | 6.41 | 5.48 | 5.42 | 0.3301 |
| 8 | 32 | 1 | – | – | – | 109.6 | 39.3 | 12.2 | 11.63 | 9.87 | 9.84 | 0.39 |
| 8 | 32 | 50 | – | – | – | 181.9 | 157.6 | 10.47 | 10.08 | 7.4 | 7.36 | 0.39 |
| 8 | 32 | 2500 | – | – | – | 63.56 | 50.97 | 9.9 | 9.54 | 5.53 | 5.48 | 0.37 |

**Table 1.** Running times of our matching algorithms. The first columns show the values of $n/10^4$ and $m/10^4$, respectively. The meaning of the other columns is explained in the text. A dash indicates that the program was not run on the instance.

- The use of heuristics to find an initial solution: Matching algorithms can either start from an empty matching or can use a heuristic to construct an initial matching.
- Simultaneous search for augmenting paths: The fastest matching algorithms search for augmenting paths in order of increasing length.
- Documentation: We discuss the merits of literate programming for documentation and why we use it document our implementations.

Figure 1 shows the running times of our bipartite matching algorithms; the source code of all our implementations can be found in [MN99]. A plus sign indicates the use of the greedy heuristic for finding an initial matching and a minus sign indicates that the algorithm starts with the empty matching. The algorithms HK [HK73] and AB [ABMP91] have a worst case running time of $O(\sqrt{n}m)$ and the other algorithms have a worst case running time of $O(nm)$. FFB stands for the basic version of the Ford and Fulkerson algorithm [FF63]. It runs in $n$ phases, uses depth-first-search for finding augmenting paths and uses $\Theta(n)$ time at the beginning of each phase for initialization. Its best case running time is $\Theta(n^2)$. The algorithms dfs and bfs are variants of the Ford and Fulkerson algorithm. They avoid the costly initialization at the beginning of each

phase and use depth-first and breadth-first search, respectively. The algorithms HK and AB use breadth-first search and search for augmenting paths in order of increasing length. The last column shows the time to check the result of the computation.

We used bipartite group graphs $G_{n,m,k}$, as suggested by [CGM+97] in their experimental study of bipartite matching algorithms, for our experiments. A graph $G_{n,m,k}$ has $n$ nodes on each side. On each side the nodes are divided into $k$ groups of size $n/k$ each (this assumes that $k$ divides $n$). Each node in $A$ has degree $d = m/n$ and the edges out of a node in group $i$ of $A$ go to random nodes in groups $i + 1$ and $i - 1$ of $B$.

The running times our algorithms differ widely. We observe (the book attempts to explain the observations, but we will not do so here) that the program with the quadratic best case running time is much slower than the other implementations, dfs is almost always slower than bfs and frequently much slower, that the use of the heuristic helps and the advantage is more prominent for the slower algorithms, and that the asymptotically better algorithms are never much slower than the asymptotically slower algorithms and sometimes much better. We also see that the time for checking the result of the computation is negligable.

Table 1 is a strong case for algorithm engineering and its interplay with the theoretical investigation of algorithms. We have algorithms with the same asymptotic bounds and widely differing observed behavior. The differences can be explained, sometimes analytically and sometimes heuristically, coined into implementation principles, and applied to other algorithms. See Sections 7.7 on maximum cardinality matching in general graphs, 7.8 on weighted matchings in bipartite graphs, and 7.9 on weighted matchings in general graphs of [MN99] to see how we applied the lessons learned from bipartite cardinality matchings to other matching problems.

## References


ABMP91. H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Information Processing Letters*, 37:237–240, 1991.

AGD. The AGD graph drawing library. `http://www.mpi-sb.mpg.de/AGD/`.

CGA. CGAL (Computational Geometry Algorithms Library). `www.cs.ruu.nl/CGAL`.

CGM+97. B. Cherkassky, A. Goldberg, P. Martin, J. Setubal, and J. Stolfi. Augment or relabel? A computational study of bipartite matching and unit capacity maximum flow algorithms. Technical Report TR 97-127, NEC Research Institute, 1997.

FF63. L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1963.

HK73. J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.

LED. LEDA (Library of Efficient Data Types and Algorithms). `www.mpi-sb.mpg.de/LEDA/leda.html`.

MN99. K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

# General Splay: A Basic Theory and Calculus

G.F. Georgakopoulos and D.J.McClurkin*

Dept. of Computer Science,
University of Crete, Heraklion, Greece
{ggeo, davidmcc}@csd.uch.gr

**Abstract.** For storage and retrieval applications where access frequencies are biased, uniformly-balanced search trees may be suboptimal. Splay trees address this issue, providing a means for searching which is statically optimum and conjectured to be dynamically optimum. Subramanian explored the reasons for their success, expressing local transformations as templates and giving sufficient criteria for a template family to exhibit amortized $O(\log N)$ performance. We present a different formulation of the potential function, based on progress factors along edges. Its decomposition w.r.t. a template enables us to relax all of Subramanian's conditions. Moreover it illustrates the reasons why template-based self-adjustment schemes work, and provides a straightforward way of evaluating the efficiency of such schemes.

## 1 Introduction

The *directory* problem, that of handling an on-line series of INSERT, DELETE and FIND requests, is for most applications addressed satisfactorily by balanced search trees (BST) (see e.g., [9] for a trace of BST's from [2] onwards), which guarantee $O(\log N)$ worst time for each of these operations. For applications however in which the access frequencies are highly biased the "overbalancedness" of these trees yields suboptimal performance. This was addressed first, statistically, in the 1970's with biased search trees (see [8], [7], [5]), and later dynamically in the 1980's with *splay trees* (see [11]). Splay trees achieve a logarithmic amortized cost for searching and other operations, and are *competitive w.r.t. any other static binary search tree* on the same set of elements (see also [15]).

However [15] left some important issues open, among which the most intriguing is the *dynamic optimality conjecture*: Are splay trees competitive even when compared to dynamically maintained search trees? To the best of the authors' knowledge this issue remains open, so we still do not know which is the best, at least in this sense, solution to the directory problem. Most probably this can be attributed to the lack of a *general self-adjustment theory*. In the 1990's Subramanian ([13]) made a step towards this by describing *splaying* as a set of rules, called *templates*. Templates are comprised of two trees, the *before-*

* This work was supported in part by grant TACIT 312304 at the Foundation of Reasearch and Technology, Hellas (FORTH).

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 4–17, 1999.
© Springer-Verlag Berlin Heidelberg 1999

and *after-schema*, each with a special *current* node. Templates specify a *splay-step* in which the before-schema is replaced by the after-schema; this is repeated continuously upwards along a path, until the current node reaches the root. Subramanian proved that for such rules to have logarithmic amortized cost, the following conditions are *sufficient for binary search trees*: (1) "strict growth": the set of nodes below the current node is strictly augmented at each stage; (2) "depth reduction": the (two) subtrees of the current node are linked strictly nearer the root; (3) "progress": descendants of template nodes must be moved below the current node within a constant number of steps.

In this paper we adopt the template framework of [13] and employ a *potential function* analysis (i.e., we define for a tree $T$ a potential function $\Phi(T)$, and for each operation $q$ which transforms $T$ to $T'$ we bound the change in potential $\Phi(T)-\Phi(T') \geq -a(q)+c(q)$ where $c(q)$ is the *actual* and $a(q)$ the *amortized* cost of $q$). Thereafter we pursue a different line of analysis, succeeding in extending substantially [13]: (a) we define the *progress factor* along a path (edge) $u \to v$ as $\log(w(u)/w(v))$ where $w(x)$ is the weight of the subtree rooted at node $x$; (b) we introduce the *multiplicity* $\mu(e)$ of a schema-edge $e$, defined as the number of paths in the schema which pass through it, minus one; (c) we define the potential function $\Phi(T)$, somewhat differently than in [11], as the sum of all progress factors along edges of $T$.

All these enable us to express $\Phi$, when applying a template, as three partial sums: (I) over non-schema edges, (II) over the paths of the schemata applied, and (III) a compensating addend for (II), summing the progress factors of each schema edge $e$, $\mu(e)$ times. We show that, for before-schemata which are *paths*, part III of $\Phi(T')$ can be summed telescopically to $O(\log|T|)$. To achieve logarithmic amortized cost it suffices to guarantee further that part III of $\Phi(T)$ is $\Omega(1)$. We show that if an after-shema contains a size-$d$ branching, all edges of which have multiplicity at least $\mu$, part III is $\Omega(\mu d \log d)$ $(= \Omega(1)$ for $\mu > 0)$. Even when a branching does not appear explicitly, we prove that many templates—quite easily recognizable by our method—offer sufficient gain because a branching appears in an *implicit* intermediate splay-step. Moreover, since multiplicities are easily visualizable, we obtain a handy calculus to assess the effiency of any splay rule-set.

Thus all conditions of [13] are relaxed: logarithmic amortized-cost splaying can be achieved in *any tree*, while neither "strict growth" nor "depth reduction" nor any non-trivial "progress" need to be enforced: mere *branching* (explicit or implicit) is what makes splay efficient. By our work the vast majority of template-based self-adjustment schemes turn out to have, quite unexpectedly, logarithmic amortized cost!

In sections 2 and 3 we give basic definitions and some metrics for templates. In sections 4 and 5 we present our calculus for estimating the cost of a splay operation and prove sufficient conditions for logarithmic amortized cost "splaying". In section 6 we give characteristic applications. Section 7 is an epilogue with further open issues towards a self-adjustment theory.

## 2 Templates: Schemata, Rules and Rule-Sets

We assume the reader to be familiar with the usual tree terminology: root, node, edge, children, parent, sibling, path, degree. We shall denote the node-set of a tree $T$ with $V$ or $V(T)$ and the edge-set with $E$ or $E(T)$. Our trees will have four characteristics: (1) they will be *rooted*: we shall imagine edges as oriented from the root towards the leaves; (2) children will be ordered, pictorially from left to right; (3) they will have a single distinguished node, cursor($T$); and (4) they will have no nodes of degree one. The node-set $V(T)$ will consist of internal($T$) nodes (those with degree $\geq 2$), and external($T$) (those with degree 0). Edges between internal nodes will be called *internal* edges and their set will be denoted by $I(T)$.

Expanding on [13], a *schema* is a cursored tree $\rho$, where cursor($\rho$) is one of its *internal* nodes. Given a cursored tree $T$ and a schema $\rho$, a *matching of* $\rho$ *into* $T$, $f : \rho \to T$, is a mapping which "preserves" the cursor, the edges, the degrees and the order of the children. (So $f$ is a tree endomorphism for trees as described above.) If a matching exists it will be unique and we shall say that $\rho$ *matches into* $T$ and denote the matching by $\rho \to T$. When referring to a schema $\rho$ matched into a tree $T$, we shall denote the image of a node, an edge, a set of edges, the cursor, etc., of $\rho$ in $T$ by appending only the "$\to T$" part. E.g., external($\rho$) $\to T$ is the set $(\rho \to T)$(external($\rho$)). We do the same for the inverse image of a node, edge, etc., of $T$ back to $\rho$: e.g., for $e \in T$, $e \to \rho$ is the inverse image $(\rho \to T)^{-1}(e)$ (when defined).

A *template-rule* $\sigma$ is a triple $\sigma = \langle \sigma^-, \sigma^+, \sigma^* \rangle$ where $\sigma^-$ and $\sigma^+$ are schemata *with the same number of external nodes* (but not necessarily the same number of internal nodes), and $\sigma^*$ is a bijection external($\sigma^-$) $\to$ external($\sigma^+$), determining the rearrangement of these leaves. In the case that we are dealing with search trees, $\sigma^*$ simply follows the left-to-right ordering of the external nodes of $\sigma^-$ and $\sigma^+$. Our approach however allows us to address the general case in which nodes may be ordered by other criteria.

We shall call $\sigma^-$ the "before"-schema and $\sigma^+$ the "after"-schema. A rule $\sigma$ is *applicable* in $T$ if and only if $\sigma^-$ matches in $T$. In this case we define the *application of* $\sigma$ *to* $T$ as the (unique) tree $T^{[\sigma]}$ with node-set $V(T) - (I(\sigma^-) \to T) \cup I(\sigma^+)$, such that the edges outside $I(\sigma^-) \to T$ remain unaffected, $\sigma^+$ matches in $T^{[\sigma]}$, and the nodes external($\sigma^-$) $\to T$ correspond to external($\sigma^+$) $\to T^{[\sigma]}$ according to $\sigma^*$. Although $T^{[\sigma]}$ is a re-linking of parts of $T$ with $\sigma^+$, we shall imagine it as a new "copy". (See figure 1. Cursor positions are indicated by arrows, and the external-nodes' mapping $\sigma^*$ by uniquely labelling the respective nodes.)

Template-rules are intended to be applied iteratively along the path from some *initial* node to some other *final* node (usually the root of our tree), selecting at each stage a matching template-rule. For this purpose we shall need a whole *set of rules*. It is assumed that we possess a *complete* rule-set, i.e., a set $S$ of rules sufficiently rich in order that given any cursor position (other than the final one) within any cursored tree $T$ there exists in $S$ at least one applicable rule for $T$. A rule-set may be given either explicitly or implicitly. Given a complete rule-set $S$ and an initial tree $T$ with an initial position for cursor($T$), we get a

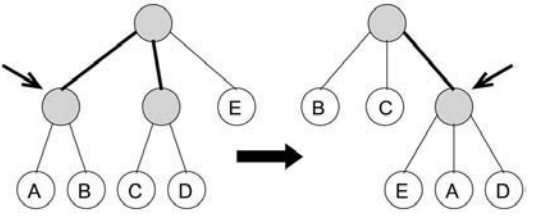



**Fig. 1.** A template rule: "before" and "after" schema (shaded nodes are internal)

sequence of cursored trees,

$$T = \underbrace{T_0 \xrightarrow{\sigma_1} T_1 \xrightarrow{\sigma_2} T_2 \longrightarrow \cdots \longrightarrow \overbrace{T_{k-1} \xrightarrow{\sigma_k \in S} T_k}^{\text{splay step}} \longrightarrow \cdots \xrightarrow{\sigma_L} T_L}_{\text{splay operation}} = S(T)$$

where each $T_k$ is obtained by applying an applicable rule from $S$, until cursor$(T)$ reaches its final position (e.g., the root of $T$). Following the traditional—by now—terminology, we shall call $T_{k-1} \to T_k$ a *splay-step*, and the whole transformation $T_0 \to \ldots \to T_L$ a *splay-operation*.

At this level of generality some splay-steps may be, in some sense, *ineffective*. Consider two rules, $\sigma_\alpha$ and $\sigma_\beta$: if $\sigma_\beta^-$ matches in $\sigma_\alpha^+$ and $\sigma_\beta$ is applied immediately after $\sigma_\alpha$, all relinking due to $\sigma_\beta$ takes place within $\sigma_\alpha^+$. In this case $\sigma_\alpha$ and $\sigma_\beta$ combine to form one rule, of size *equal* to that of $\sigma_\alpha$. Such "combined" rules should be *already present* in $S$ and *selected immediately* for application. Since certainly we should not pay for just "splaying around", we shall assume that $S$ is not only complete but also *effective*, i.e., for any tree $T$ and any rule $\sigma_\alpha$ matched in $T$, there exists some rule $\sigma_\beta$ which matches in $T^{[\alpha]}$ (thus it is applicable) but which does not match in $\sigma_\alpha^+$ (thus it is "effective"). Effective rule-sets should be, of course, also *applied* accordingly: usually there will be a general prescribed "rule-selection policy" which will enforce only effective applications of rules. In many cases such a policy may be "select the *largest* applicable rule".

## 3 Splay Histories and the Amortized Cost of Splaying

We now turn our attention to the computational cost of splaying. We define some reasonable metrics for templates, rules and rule-sets. The *size* of a tree $T$ is defined to be the number of external nodes of $T$. The size of a rule $\sigma = \langle \sigma^-, \sigma^+, \sigma^* \rangle$ is the size of either $\sigma^-$ or $\sigma^+$ (they are equal). For a rule-set $S$, MaxSize$(S)$ is the *maximum* size of any rule in $S$. Implicit in this definition is the fact that we are dealing with bounded-height and bounded-degree templates.

Consider now a schema $\rho$ and its internal edges, $I(\rho)$. Let us denote by branch$(\rho)$ the maximum number of *internal* nodes which are siblings of each other. If branch$(\rho) = 1$ then all internal edges form a path, in which case we shall

denote by bottom($\rho$) the internal node of $\rho$ farthest from the root. If branch($\rho$) $\geq$ 2 then the edges $I(\rho)$ form a proper tree, in which case bottom($\rho$) is undefined.

The branching $B_S$ of a rule-set $S$ is the minimum branch($\sigma^+$) of any any "after"-schema $\sigma^+$ of rules in $S$. A rule $\sigma$ is called *path-oriented* if branch($\sigma^-$) = 1 and cursor($\sigma^-$) = bottom($\sigma^-$); it is *(explicitly) branching* if branch($\sigma^+$) $\geq 2$.

We assume that the worst-case cost of a splay step is $O(\text{MaxSize}(S))$; this cost includes: (1) the cost of inspecting $T$ (around cursor($T$)) and selecting an applicable rule from $S$; (2) performing the corresponding splay-step; and (3) performing any further computation related to whatever information is stored in the nodes. A splay (operation) on $T$ can have a cost even $\Omega(\text{size}(T))$, so we will be interested in obtaining an amortized cost analysis. We assume that we start from an initial tree $T^{(0)}$ and we perform a series of $M$ splay operations, $s_k$, according to rule-set $S$:

$$T = \underbrace{T^{(0)} \xrightarrow{s_1} T^{(1)} \xrightarrow{s_2} T^{(2)} \longrightarrow \cdots \longrightarrow \overbrace{T^{(k-1)} \xrightarrow{s_k} T^{(k)}}^{\text{splay operation}} \longrightarrow \cdots \xrightarrow{s_M} T^{(M)}}_{\text{splay history}}$$

Let each splay operation $s_k$ applied to $T$ cost $C(s_k, T_{k-1})$. We shall define a non-negative function, $\Phi(\cdot)$, on trees with weights, bounding the cost in the following sense:

$$C(s_k, T_{k-1}) \leq l(T_k) + (\Phi(T_{k-1}) - \Phi(T_k))$$

where $l(T_k)$ is a convenient cost-function defined on $T_k$. Since in such an expression potential differences cancel telescopically, the positivity of $\Phi$ guarantees that $\sum_k C(s_k, T_{k-1})$ is less than $\sum_k l(T_k)$, and so $l(T)$ can be taken to be the *amortized cost* of a splay operation. We shall examine—among other related issues—under what general conditions this cost is logarithmic as a function of size($T$).

## 4 Our Logistics Scheme: Progress and Edge-Multiplicities

We assume there to be a *weight function*, $w : V(T) \to [1, \infty)$, defined on nodes $u$ of $T$, such that (i) $w(u)$ is super-additive, i.e., if node $u$ has children $v_1, \ldots, v_d$ then $w(u) \geq \sum_{k=1,\ldots,d} w(v_k)$; (ii) $w(u)$ depends only on the set of elements in the subtree rooted at $u$, and not on their structure. The standard example of a weight function is $w(u) = \text{size}(T_u)$, where $T_u$ is the subtree with root $u$. (We assume that $w(u) \geq 1$ because we shall only deal with quotients $w(u)/w(v)$.) Since along an edge $e = (u, v)$ one essentially passes from a subtree of size $w(u)$ to one with size $w(v)$, we define the *progress factor* along $e$ as

$$\phi(e) = \phi((u, v)) = \log(w(u)/w(v)).$$

Analogously for a path $P = \langle u_0, u_n \rangle$, i.e., a set of edges $(u_{k-1}, u_k)$, $k = 1, \ldots, n$, we define the progress factor of $P$ as $\phi(P) = \log(w(u_0)/w(u_n))$. We define as

the potential of $T$, $\Phi(T)$, the sum of all progress factors:

$$\Phi(T) = \sum_{e \in E(T)} \phi(e)$$

In the case of full binary trees this corresponds closely to the Sleator-Tarjan potential function, $\Phi_{ST}(T) = \sum_{u \in V(T)} \log(w(u))$. In particular $\Phi(T) = \Phi_{ST}(T) + \log(w(\text{root}(T)))$.

For most edges $e$, $\phi(e)$ does not change when applying a rule $\sigma$, so many terms cancel out in the expression for $\Delta\Phi = \Phi(T) - \Phi(T^{[\sigma]})$. To capture these cancellations we rewrite the expression for $\Phi(T)$ in a way that depends on the rule we apply. Given a tree $\rho$, the *multiplicity* $\mu(e)$ of an edge $e$ in $\rho$, is the total number of paths $\langle \text{root}(\rho), u \rangle$, for $u \in \text{external}(\rho)$ passing through $e$, minus one (we subtract one to simplify expressions throughout):

$$\text{For } e \in \rho,\ \mu(e) = |\{\langle \text{root}(\rho), u \rangle : u \in \text{external}(\rho)\}|$$

We rewrite $\Phi(T)$ w.r.t. a schema $\sigma$ as $\Phi(T) = \Phi_\sigma(T) + \Phi'_\sigma(T) + \Phi''_\sigma(T)$, where

$$\Phi_\sigma = -\sum_{e \in I(\sigma) \to T} \mu(e \to \sigma)\phi(e),\ \Phi'_\sigma = \sum_{u \in \text{external}(\sigma) \to T} \phi(\langle \text{root}(\sigma) \to T, u \rangle),$$

$$\Phi''_\sigma = \sum_{e \in E(T) - (E(\sigma) \to T)} \phi(e)$$

The $\Phi''_\sigma$-sum is over "non-schema" edges; the $\Phi'_\sigma$-sum is over the paths from the "root" of $\sigma$ to its external nodes and the $\Phi_\sigma$-sum compensates for this by subtracting the progress factor for internal edges the appropriate number of times, i.e., their multiplicity w.r.t. $\sigma$. We write

$$\Delta\Phi = (\Phi_{\sigma^-}(T) - \Phi_{\sigma^+}(T^{[\sigma]})) + (\Phi'_{\sigma^-}(T) - \Phi'_{\sigma^+}(T^{[\sigma]})) + (\Phi''_{\sigma^-}(T) - \Phi''_{\sigma^+}(T^{[\sigma]})).$$

Edges outside $\sigma$ remain unaffected, i.e., $\Phi''_{\sigma^-}(T) = \Phi''_{\sigma^+}(T^{[\sigma]})$. Each subtree hanging from an external node, $u$, of $\sigma^-$ although repositioned is again hanging from an external node of $\sigma^+$, so the total progress along $\langle \text{root}(\sigma^-), u \rangle$ or $\langle \text{root}(\sigma^+), u \rangle$ is the same, so $\Phi'_{\sigma^-}(T) = \Phi'_{\sigma^+}(T^{[\sigma]})$. (Here we use property (b) of the weight function $w(u)$.) We therefore get an exact expression for $\Delta\Phi$ by $\Phi_\sigma$-sums only:

$$\Delta\Phi = -\underbrace{\left(\sum_{e \in I(\sigma^-) \to T} (\mu(e) \to \sigma^-)\phi(e)\right)}_{\text{"loss" part of potential change}} + \underbrace{\left(\sum_{e \in I(\sigma^+) \to T^{[\sigma]}} (\mu(e) \to \sigma^+)\phi(e)\right)}_{\text{"gain" part of potential change}}$$

So $\Phi$ changes by "losing" progress factors along $e \in I(\sigma^-) \to T$ and "gaining" along $e \in I(\sigma^+) \to T^{[\sigma]}$, each as many times as their multiplicity. We visualize this situation, thinking of each edge $e \in I(\sigma^-) \to T$ as being labeled by $\mu(e \to \sigma^-)$ "minuses" and each $e \in I(\sigma^+) \to T^{[\sigma]}$ by $\mu(e \to \sigma^+)$ "pluses". See figure 2. Two simple and crucial lemmata follow:

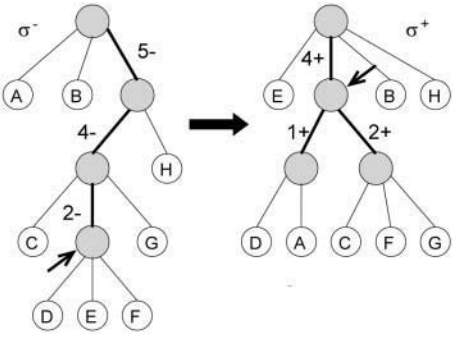



**Fig. 2.** Template Rules: '+' and '−'

**Lemma 1 ("Vertical" Lemma):** Let $P$ be a path in tree $T$ with weight $w(\cdot)$, and $w_T = w(\text{root}(T))$. Let $c(e)$ be a constant for each edge $e$ in $P$, and $c_{\max} = \max_{e \in P}\{c(e)\}$. Then

$$\sum_{e \in P} c(e)\phi(e) \leq c_{\max}\phi(P) \leq c_{\max}\log(w_T).$$

Proof: Easily obtained by a "telescopic" product (sum of logarithms). □

**Lemma 2 ("Horizontal" Lemma):** Let $C \subseteq \text{children}(u)$ for $u$ a node of tree $T$. Then

$$\sum_{v \in C} \phi((u,v)) \geq |C| \log |C|.$$

Proof: An easy consequence of the super-additivity of $w(\cdot)$ and the well-known fact that for positive numbers $x_k$, $k = 1, \ldots, n$, their arithmetic mean is larger than or equal to the geometric mean. □

The following third lemma expresses the fact that the minimum for lemma 2 cannot always be achieved. Lemma 3 will not be useful for our main theorem, but it will be for an interesting application.

**Lemma 3 ("Ladder" Lemma):** Let $\rho$ be a path-oriented schema, with length $h_p$ and total weight $w_p$. Then

$$\sum_{e \in P} \phi(e) \;\geq\; \max\left(h_\rho \log \frac{h_\rho}{w_\rho}, h_\rho\right).$$

Proof: (A slight modification—to fit our working framework—of the proof in [4]. □

If during a splay operation of $L$ splay-steps, the "loss"-part is accumulated along a path, and each "gain"-part is along branchings then the total "loss" will be less than $\text{MaxSize}(S)O(\log(w_T))$ (by the "vertical" lemma) and the total "gain" will be greater than $L\Omega(1)$ (by the horizontal lemma). The derived relation:

$$L\Omega(1) \leq \text{MaxSize}(S)O(\log(w_T)) - \Delta\Phi \tag{1}$$

(where $\Delta\Phi \geq 0$) will prove a logarithmic amortized cost for a splay operation. We can guarantee such a bound under very mild conditions.

## 5 Productive and Progressive Schemata

To guarantee that all "−" will accumulate along a path we consider only *path-oriented* rules. To guarantee that some "+" will appear along at least two edges leading to siblings, we consider (temporarily) only *explicitly branching* rules.

There is still a third thing we must guarantee: that "minuses" gathered along a path do not overlap—at least not in an unrestricted manner. When we apply each "next" rule $\sigma_\beta$, we lose potential along $I(\sigma_\beta^-) \to T$, but along this path we find "gain" produced by the previous rule, $\sigma_\beta$. If the new multiplicities for $e \in T$, $\mu(e \to \sigma_\beta^-)$, are less than the old, $\mu(e \to \sigma_\alpha^+)$, then previous "gain" compensates for next "loss", and "loss" appears only along path $P = (I(\sigma_\beta^-) \to T) - (E(\sigma_\alpha^+) \to T)$. Since $I(\sigma_\beta^- \to T)$ is a path, and $S$ is applied effectively, paths like $P$ extend continuously upward in a non-overlapping fashion, so the "vertical" lemma applies. A simple criterion is sufficient to guarantee that $\mu(e \to \sigma_\alpha^+$ is strictly greater than $\mu(e \to \sigma_\beta^-$ for all edges $e$ in the overlapping region: We shall call a rule $\sigma$ *progressive* if in its after-schema $\sigma^+$ there exists at least one internal node lying strictly below cursor$(\sigma^+)$. Let us combine all this in the following basic theorem:

**Theorem 1:** If $S$ is an *effective* rule-set with *path-oriented*, *explicitly branching* and *progressive* rules, then splay operations on $T$ have an amortized cost of

$$O\left(\frac{\text{MaxSize}(S)}{B_S \log B_S} \log(w_T)\right).$$

Proof: let $\sigma_k$, $k = 1, \ldots, L$, be the sequence of rules applied: $T_{k-1} \xrightarrow{s_k} T_k$. We get $\Delta\Phi$:

$$\Delta\Phi = \sum_{k=1}^{L} \left[ -\left( \sum_{e \in I(\sigma_k^-) \to T_{k-1}} \mu(e \to \sigma_k^-)\phi(e) \right) + \left( \sum_{e \in I(\sigma_k^+) \to T_k} \mu(e \to \sigma_k^+)\phi(e) \right) \right]$$

Within each $T_k$, $k = 1, \ldots, L$, we define the edge-sets $P_k$, $Q_k$ and $R_k$ based on $\sigma_k^+$ and $\sigma_{k+1}^-$:

$$Q_k = (I(\sigma_{k+1}^-) \to T_k) \cap (I(\sigma_k^+) \to T_k), \quad P_k = (I(\sigma_{k+1}^-) \to T_k) - Q_k,$$

$$R_k = (I(\sigma_k^+) \to T_k) - Q_k$$

Recalling that $\sigma_{k+1}^-$ is a path, we see that $P_k$ is a path "above" $\sigma_k^+$, $Q_k$ is a path inside schema $\sigma_k^+$ (part of $\sigma_{k+1}^-$) and $R_k$ consists of the remaining internal edges of $\sigma_k^+$. See figure 3. Since the multiplicities are bounded by MaxSize$(S)$ (in fact by MaxSize$(S) - 1$) we get:

$$\sum_{e \in I(\sigma_{k+1}^-) \to T_k = Q_k \cup P_k} \mu(e \to \sigma_{k+1}^-)\phi(e) \leq \sum_{e \in Q_k} \mu(e \to \sigma_{k+1}^-)\phi(e) + \text{MaxSize}(S)\phi(P_k)$$

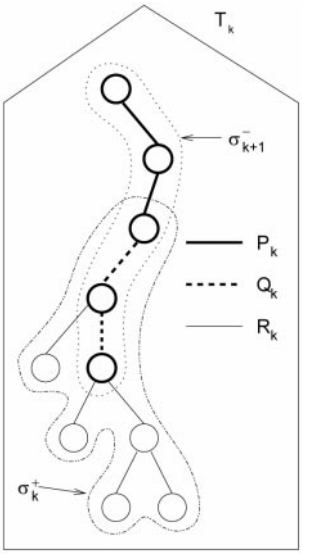



**Fig. 3.** +/−-canellation along $\mathrm{Q_k}$

Moreover, by progressiveness we get $\mu(e \to \sigma_k^+) \geq \mu(e \to \sigma_{k+1}^-) + 1$ for $e \in Q_k$, and since internal edges have positive multiplicity, we get:

$$\sum_{e \in I(\sigma_k^+) \to T_k = Q_k \cup R_k} \mu(e \to \sigma_k^+)\phi(e) \geq \sum_{e \in Q_k} \mu(e \to \sigma_{k+1}^-)\phi(e) + \sum_{Q_k \cup R_k} \phi(e)$$

The important thing now is that all paths $P_k$, $k = 1, \ldots, L$, even $P_0 = I(\sigma_1^-) \to T_0$, are *disjoint paths* of the initial tree $T = T_0$: they consist of edges not matched by any before-schema up to the k-th step (a fact easy either to visualize or to prove inductively). Moreover their progress factors in $T_k$ are equal to their corresponding progress factors in $T$, since the weights of their subtrees are left untouched up to the k-th step. Summing for $\Delta\Phi$, for $k = 1, \ldots, L$, the partial sums over $Q_k$ cancel out, and we are left with:

$$\Delta\Phi = \Phi(T_0) - \Phi(T_L) \geq -\mathrm{MaxSize}(S) \underbrace{\left( \sum_{k=0}^{L-1} \phi(P_k) \right)}_{\text{part I}} + \sum_{k=1}^{L} \underbrace{\left( \sum_{e \in I(\sigma_k^+) \to T_k} \phi(e) \right)}_{\text{part II}}$$

Since $\mathrm{branch}(\sigma_k^+) \geq B_S$, the theorem follows by applying the vertical lemma to part I and the horizontal lemma to part II (see equation (1)). □

Notice that splay can "pay" also for any other operation with an *actual* cost of $O(L)$. If splay starts at depth $h$, then we are permitted to perform $L = \Omega(h/\mathrm{MaxSize}(S))$ splay-steps. So splay can pay not only for itself, but also

for the steps needed to reach the initial cursor-position starting from the root (e.g., during a FIND operation).

Theorem 1 does not address *path-to-path* rules, i.e., non-branching rules, in which *both* the "before"- and "after"-schemata are paths. However we can relax the explicit branching condition: most path-to-path rules are—under mild conditions—branching rules in disquise. The next lemma, although not in the strongest possible form, suffices to reveal this fact:

**Lemma 4: (a)** (see also [13]) Let $\sigma$ be a path-to-path rule, and let *all* nodes in children(bottom($\sigma_k^-$)) be linked *strictly* above bottom($\sigma_k^+$). Then the gain-part of $\sigma$ is $\Omega(1)$, in an amortized sense (specifically every two splay-steps). **(b)** Let $\sigma$ be a path-to-path rule, and let plus($\sigma_k^-$) $\subseteq$ children(bottom($\sigma_k^+$)) denote those external nodes $u$ of $\sigma_k^-$ for which the edge $(u, \text{bottom}(\sigma_k^-)) \in I(\sigma_{k-1}^+)$ was an *internal* edge of $\sigma_{k-1}^+$ (i.e., carries a "+"). If *any* node $u$ in plus($\sigma_k^-$) is linked *strictly* above bottom($\sigma_k^+$), then the gain-part of $\sigma$ is $\Omega(1)$ in an amortized sense (specifically every two splay-steps).

Proof: after presenting our $\pm$-calculus in theorem 1, we can prove lemma 4 easily: (a) Rules $\sigma$ satisfying the stated condition can be considered to hide a branching obtained by an intermediate phase: We *factor* $\sigma_k$ into two rules $\alpha = \langle \alpha^-, \gamma_\alpha, \alpha^* \rangle$ and $\beta = \langle \gamma^\beta, \beta^+, \beta^* \rangle$, where $\alpha^- = \sigma_k^-$ and $\beta^+ = \sigma_k^+$ (see figure 4). The first intermediate schema $\gamma_\alpha$ consists of three nodes, $r$ as root and $u$, $v$ as two children, where $u$ is cursor($\gamma_\alpha$). children(bottom($\sigma_k^-$)) are the children of $u$, and children(bottom($\sigma_k^+$)) are the children of $v$. The remaining external nodes of $\sigma_k^-$ hang arbitrarily from root $r$. The second intermediate schema $\gamma_\beta$ consists of nodes $u$ and $r$ as internal nodes and its external nodes are those of $\gamma_\alpha$ linked to $u$ and $r$. All this is possible since no node in children(bottom($\sigma_k^-$)) will be a child of bottom($\sigma_k^+$). Schema $\gamma_\alpha$ is *explicitly* branching, so it has positive gain.

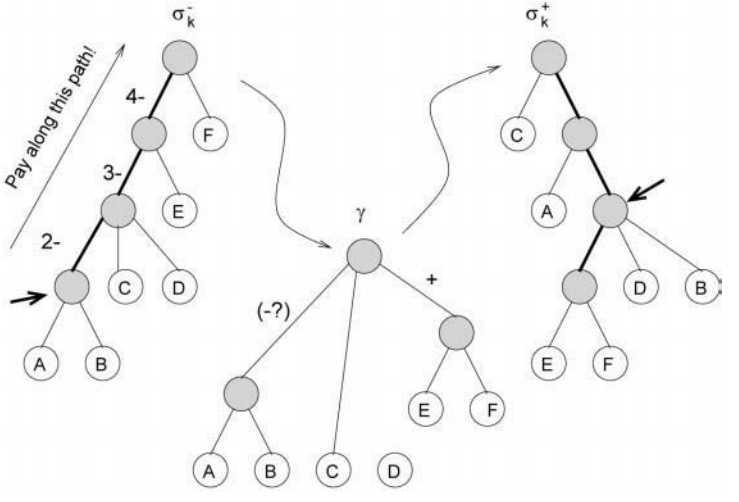



**Fig. 4.** A branching in disguise (a)

The "loss" in potential associated with passing from $\gamma_\alpha$ to $\gamma_\beta$ (a "$-$" along edge $(r, u)$) is transferred to $\sigma_k^-$ by adding a series of "$-$" along the path of $\sigma_k^-$ and this is why we keep children(bottom($\sigma_k^-$)) at their positions: the progress factor along the path from root($\gamma_k$) to their parent then is equal to the progress along the path from root($\sigma_k^-$) to them. In doing this we *may* lose all gain of step $k$, but in total the gain from steps $k$ and $k+1$ will be $\Omega(1)$—whatever these steps are.
(b) The second case is somewhat similar: Charging the path of schema $\sigma_k^-$ another "minus" including the edge below cursor($\sigma_k^-$) with a "plus", we are free to assign one "plus" to this edge in the next schema, $\sigma_k^+$, thus gaining $\Omega(1)$ for our potential. Again these extra "minuses" may cause no gain at step $k$, but in total the gain from steps $k$ and $k+1$ will be $\Omega(1)$—whatever these steps are. (Alternatively one can observe that such rules produce an explicit branching if applied twice.) □

Let us refer to path-to-path rules as in the above lemma as *implicitly branching* rules, and let $S^{(r)}$ denote the set of "composite" rules obtained from $r$ consecutive applications of rules from $S$. Our final theorem fulfills our promises:
**Theorem 2:** Let $S$ be a rule-set for which, for some $r$, $S^{(r)}$ is *effective* with *path-oriented*, *progressive* and, either *explicitly* or *implicitly*, *branching* rules. Splay operations on tree $T$ with weight $w(\cdot)$ will have amortized cost (here $B_S$ takes into consideration any implicitly branching schemata):

$$O\left(r\frac{\text{MaxSize}(S)}{B_S \log B_S}\log(w_T)\right).$$

Proof: Straightforward, according to the discussion above. □

## 6 Examples: Classic Splay Trees, Median-Split Trees and Path Balance

Consider the rule set for classic splay trees. The "zig-zag" case is explicitly branching, and the "zig-zig" case is implicitly branching according to lemma 3(a). The "zig" case guarantees no gain, but if we apply each time the large rule, the "zig" case will be applied at most once during the last splay-step. Note that our "$+/-$" calculus gives easily an amortized number of $4\log N + 2$ nodes visited for classic splay.

In figure 5 we show a template for a variety of trees of our design—we call them "median-split" trees. These trees are B-tree-like because except for the root all nodes have degree ranging from $\delta$ to $2\delta$. From the pluses and minuses in the picture we obtain immediately that the "loss" part has a constant factor of $\Theta(\delta)$ and the gain part, also, $\Theta(\delta)$ (recall the previous section). Therefore during splay we visit an amortized number, $h$, of nodes, where $h \in O(\log N)$, and where the constant is independent of $\delta$. This result compares with that of Sherk (see [10]); but here we have a simpler rule-set, with rules of size $\Theta(\delta)$ (in [10] templates were of size $\Theta(\delta^2)$), we have variable degrees, and a proof of

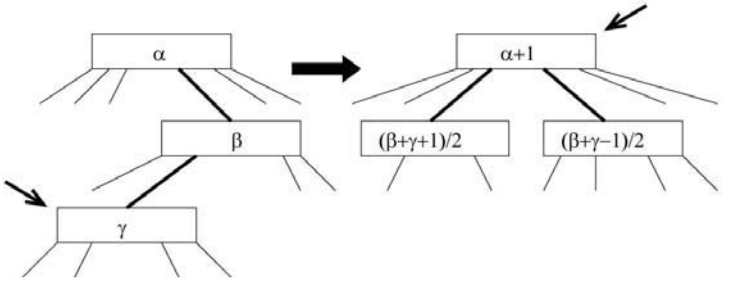



**Fig. 5.** Main Rule for "Median-Split" splay trees ($\alpha, \beta, \gamma$ = degrees of nodes, all no less than d)

just a few lines (given theorem 1) instead of many pages. (Moreover we have strong reasons to conjecture that even templates like that of [10] cannot offer a $O(\log_\delta N)$ amortized number of nodes visited.)

In figure 6 we give a rule-set which guarantees logarithmic amortized cost but does not fulfil the criteria of [13]: one of the lowermost subtrees is not repositioned nearer to the root.

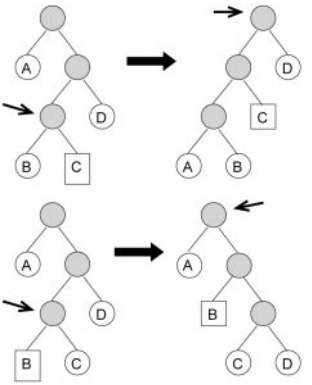



**Fig. 6.** Non-Subramanian Rules Square nodes hang from + edges

Our final application is a simplification of the analysis of the "path-balance" heuristic given in [4]. In figure 7(a) we see the simplest branching template-rule. If applied along a "straight" path of length $L$ (figure 7(b)) it halves the path, similar to a repeated application of the "zig-zig" rule. Applying this halving $\log h_L$ times, we obtain the "path-balance" heuristic discussed in [13], [4]. "Minuses" can be transferred along the initial path, thus accumulating only $\leq 2 \log h_L$ "minuses"

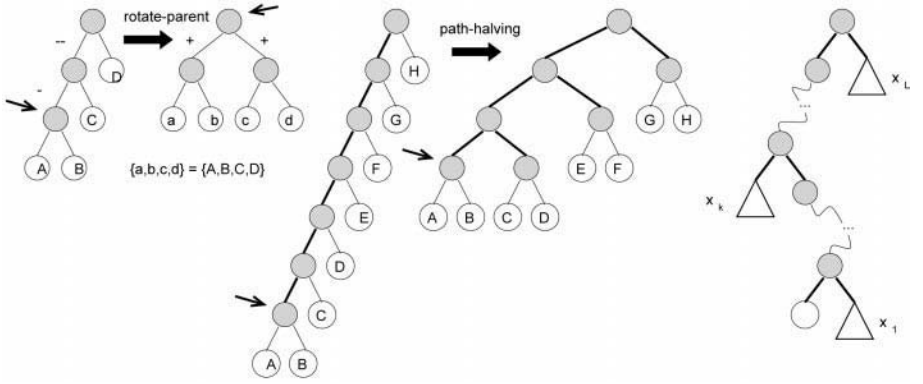



**Fig. 7.** (a) Rotate-Parent; (b) "Path-Halving": multiple application of (a); (c) "Ladder lemma"

per edge. By the "ladder" lemma and our "+/−" analysis we get:

$$\Delta\Phi \geq -\log N \log h_L \; + \; \max\left(h_L \log \frac{h_L}{\log N}, h_L\right).$$

(The case for a non straight path is treated quite similarly.) The obscure way of proving this in [4] can thus be avoided by our "+/−" calculus, while an interesting issue is illuminated: the gain and loss are again separated (a fact which is not so clear in [4]): the gain comes from the after-schema and the loss is attributed to the before-schema. Moreover our approach suggests a splay-rule better than "path-balance" in the sense that it has $O(\log N)$ amortized complexity, instead of $O(\log N \log\log N/\log\log\log N)$: apply "path-halving" only as long as the length of the path we work upon is over $2\log N$. The interesting fact here is that the depth of all nodes along the path is reduced to $O(\log N)$—a result not achievable by other template-rules (so far).

## 7 Epiloge and Further Open Issues

We have proved that under very mild and natural conditions "almost all" local bounded transformations along a path of a tree are splay operations, i.e., with logarithmic amortized cost. Our result extends previous results in all directions, and offers a handy calculus for estimating the amortized cost of a candidate set of rules for splay.

Nonetheless, quite a few improvements are possible and many interesting issues are left open. Among them we mention the following: We would like to extend our theory to templates of unbounded degree and/or unbounded height. To prove stronger forms of the lemma concerning implicitly branching rules. To formulate a similar theory for top-down splay (if possible). To obtain "gain" not

only from after-schemata but from correlating an after-schema with its before-schema. To handle non-path-oriented rules. To provide tools for accurate appraisal of which templates are best—in any possible and reasonable sense. To check whether improved, or otherwise special, results can be obtained by other potential functions or "edge factors", say by discretising our progress factors, or by height-originated, or even by asymmetric (e.g., by distinguishing "left" from "right") edge factors (see for example [14]). To provide a systematic basis for probabilistic analysis of the efficiency of template-rules. And finally to provide a basis for proving lower bounds as well.

## Acknowledgements

We would like to thank the anonymous reviewer for his/her helpful observations and suggestions.

## References

1. Allen, B., Munro, J. I., "Self Organizing Binary Search Trees", J. of the A.C.M. **25(4)** (1978), 526–535.
2. Adel'son-Vel'ski, G. M., Landis, E. M., "An Algorithm for the Organization of Information", Soviet Mathematics Doklady **3** (1962), 1259–1263.
3. Bayer, R., McCreight, E., "Organization and Maintenance of Large Ordered Indices", Acta Informatica **1(3)** (1972), 173–189.
4. Balasubramanian, R., Raman, V., "Path Balance Heuristic for Self-Adjusting Binary Search Trees", Foundations of Software Technology and Theoretical Computer Science **15** (1995), 338–348.
5. Bent, S. W., Sleator, D. D., Tarjan, R. E., "Biased Search Trees", J. on Computing **14(3)** (1985), 545–568.
6. Grinberg, D., Rajagopalan, S., Venkatesan, R., Wei, V. K., "Splay Trees for Data Compression", Proc. 6th Symp. on Discrete Algorithms (1995), 522–530.
7. Hu, T. C., Tucher, A. C., "Optimal Computer Search Trees and Variable-Length Alphabetic Codes", J. of Applied Mathematics **21(4)** (1971), 514–532.
8. Knuth, D. E., "Optimum Binary Search Trees", Acta Informatica **1(1)** (1971), 14–25.
9. Overmars, M. H., "The Design of Dynamic Data Structures", Lecture Notes on Computer Science **156**, Springer Verlag, 1983.
10. Sherk, M., "Self Adjusting k-ary Search Trees", J. of Algorithms **19(1)** (1995), 25–44.
11. Sleator, D. D., Tarjan, R. E., "Self Adjusting Binary Search Trees", J. of the A.C.M. **32(3)** (1985), 652–686.
12. Sleator, D. D., Tarjan, R. E., "Amortized Efficiency of List Updates and Paging Rules", C. of the A.C.M. **28(2)** (1985), 202–208.
13. Subramanian, A., "An Explanation of Splaying", J. of Algorithms **20(3)** (1996), 512–525.
14. Sundar, R., "Twists, Turns, Cascades, Dequeue Conjecture and Scanning Theorem", 30th FOCS (1989), 555–559.
15. Tarjan, R. E., "Amortized Computational Complexity", J. of Applied and Discrete Mathematics **6(2)** (1985), 306–318.

# Static Dictionaries Supporting Rank

Venkatesh Raman and S. Srinivasa Rao

The Institute of Mathematical Sciences, C. I. T. Campus,
Chennai 600 113. India.
{vraman, ssrao}@imsc.ernet.in

**Abstract.** A static dictionary is a data structure for storing a subset $S$ of a finite universe $U$ so that membership queries can be answered efficiently. We explore space efficient structures to also find the rank of an element if found. We first give a representation of a static dictionary that takes $n \lg m + O(\lg \lg m)$ bits of space and supports membership and rank (of an element present in $S$) queries in constant time, where $n = |S|$ and $m = |U|$. Using our structure we also give a representation of a $m$-ary cardinal tree with $n$ nodes using $n\lceil \lg m \rceil + 2n + o(n)$ bits of space that supports the tree navigational operations in $O(1)$ time, when $m$ is $o(2^{\lg n / \lg \lg n})$. For arbitrary $m$, we give a structure that takes the same space and supports all the navigational operations, except finding the child labeled $i$ (for any $i$), in $O(1)$ time. Finding the child labeled $i$ in this structure takes $O(\lg \lg \lg m)$ time.

## 1 Introduction and Motivation

A static dictionary is a data structure for storing a subset $S$ of a finite universe $U$ so that membership queries can be answered efficiently. This problem has been widely studied and various structures have been proposed to support membership in constant time [8,7,4,12] in slightly different models. There are many situations where one is interested in finding the rank of an element found (say when the elements are marks of an exam, and one is interested in the relative place of a mark in the ranking). Our focus in this paper is on space efficient data structures to support the rank operation, which asks for the number of elements in the set less than or equal to the given element. Our model of computation is an extended RAM machine model that permits constant time arithmetic and boolean bitwise operations.

Another motivation for studying rank operation comes from the recent succinct representation of $m$-ary cardinal trees[3]. A *cardinal tree* of degree $k$ is a rooted tree in which each node has $k$ positions for an edge to a child. A binary tree is a cardinal tree of degree 2. An *ordinal tree* is a rooted tree of arbitrary degree in which the children of each node are ordered. The representation [3] of an $m$-ary cardinal tree essentially has two parts: one part giving the ordinal information of the tree using $2n + o(n)$ bits and the other part storing the children information of each node in the tree using $n \lg m$ bits where $n$ is the number of nodes. To navigate around the tree, in particular, to find the child labeled $i$ of a

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 18–26, 1999.
© Springer-Verlag Berlin Heidelberg 1999

node, we need to find the rank of the element $i$ in the ordinal part information of the node. The representation of Benoit et. al.[3] supports this operation in $O(\lg\lg m)$ time. Clearly, to perform this operation in constant time, a structure for static dictionary taking $n \lg m + o(n)$ bits supporting the rank operation in constant time suffices. Though we could achieve this when $m$ is $o(2^{\lg n/\lg\lg n})$, for the general case we have a structure that supports rank and membership in $O(\lg\lg\lg m)$ time.

If we want to support the rank operation for every element in the universe, there is a lower bound of $\Omega(\lg\lg m)$ time per query even if the space used is polynomial in $n$ [1]. Willard [14] gave a structure that answers rank (and hence membership) queries in $O(\lg\lg m)$ time using $O(n \lg m)$ bits of space. Fiat et. al.[7] gave a structure that answers membership queries in constant time and rank queries in $O(\lg n)$ time using $n \lg m + O(\lg\lg m + \lg n)$ bits of space. The structure given by Pagh [12] answers membership and rank queries in constant time when the size of the set $n$ is $\omega(m \lg\lg m/\lg m)$, using $n \lg(m/n) + O(n)$ bits. Our focus here is to support rank queries for only the elements that are present in the given set.

The only structure we know of to support membership and rank for those elements found is due to Benoit et. al.[3] en route to their efficient cardinal tree representation. Their structure supports membership and rank in $O(\lg\lg m)$ time using at most $n \lg m$ bits. We give an alternate structure that supports both these operations in $O(1)$ time using $n \lg m + O(\lg\lg m) - \Theta(n)$ bits. As a matter of fact, the structure due to Benoit et. al. supports both these operations in constant time using at most $n \lg m$ bits as long as $n > \lg m$ whereas our structure supports both these operations using the same time and space as long as $n > \lg\lg m$. In the smaller range, both these structures take $O(\lg n)$ time if only $n \lg m$ bits are allowed.

In Section 2, we give a space efficient static dictionary structure that answers membership and rank queries in constant time. This structure builds up on the recent enhancement of Pagh[12] of the FKS[8] dictionary and uses $n \lg m + O(n + \lg\lg m)$ bits. In Section 3, we use an interesting idea to remove the $O(n)$ term in the space complexity of the structure. In this section, we also outline space efficient structures to support the select operation (find the $j$-th smallest element in the given set). In Section 4, we outline the $m$-ary cardinal tree representation of Benoit et. al.[3] and explain how our rank dictionary structure can be used to improve the running time from $O(\lg\lg m)$ to $O(\lg\lg\lg m)$ for finding the child labeled $i$, if exists, for any $i$. We also illustrate another improvement to the structure so that all the navigational operations can be supported in constant time if $m$ is $o(2^{\lg n/\lg\lg n})$.

## 2 A Rank Structure Taking $n \lg m + O(n + \lg\lg m)$ Bits

Fredman et. al.[8] have given a structure that takes $n \lg m + O(\lg\lg m + n\sqrt{\lg n})$ bits and supports membership in $O(1)$ time. Schmidt and Seigel [13] have im-

proved this space complexity to $n \lg m + O(\lg \lg m + n)$ bits. We refer to this structure as the FKS dictionary in the later sections.

The original FKS construction to store a set $S$, as described in [13], has four basic steps:

- A function $h_{k,p}(x)$ is found that maps $S$ into $[0, n^2 - 1]$ without collisions. It suffices to choose $h_{k,p}(x) = (kx \ mod \ p) \ mod \ n^2$ with suitable $k < p < n^2 \lg m$ where $p$ is a prime. Here, the values $k$ and $p$ depend on the set $S$.
- Next, a function $h_{\kappa,r}(z)$ is found that maps $h_{k,p}(S)$ into $[0, n-1]$ so that the sum of the squares of the collision sizes is not too large. Again, it suffices to choose $h_{\kappa,r}(z) = (\kappa z \ mod \ r) \ mod \ n$, where $r$ is any prime greater than $n^2$ and $\kappa \in [0, r]$ so that $\sum_{0 \leq j < n} |h^{-1}_{\kappa,r}(j) \cap h_{k,p}(S)|^2 < 3n$.
- For each non-empty bucket $i$, a secondary hash function $h_i$ is found that is one-to-one on the collision set. We choose $h_i(z) = (k_i z \ mod \ r) \ mod \ {c_i}^2$, where $k_i \in [0, r-1]$ and $c_i$ is the size of the collision set. The element $x \in S$, is stored in location $C_i + h_i(h_{\kappa,r}(h_{k,p}(x)))$, where $C_i = {c_0}^2 + {c_1}^2 + \cdots + {c_{i-1}}^2$. This locates all $n$ items within a table of size $3n$, say $A^\star[1 \ldots 3n]$.
- Finally the table is stored without any vacant locations in an array $A[1...n]$ in the same order.

The composite hash function requires the parameters $k$, $p$, $\kappa$ and $r$ for $h_{k,p}$ and $h_{\kappa,r}$, a table $K[0 \ldots n]$ storing the parameters $k_i$ for secondary hash functions $h_i$, a table $C[0 \ldots n]$ listing the locations $C_i$ and finally a compression table $D[1 \ldots 3n]$, where $D[j]$ gives the index, within $A$, of the item (if any) that hashes to the value $j$ in $A^\star$. Thus this composite hash function description requires $O(n \lg n + \lg \lg m)$ bits of space.

Schmidt and Siegel [13] first observe that up to $\lfloor \lg n \rfloor + 1$ secondary hash functions are sufficient to store the elements of the set. They also show how to represent this composite hash function using $O(n + \lg \lg m)$ bits of space. We briefly describe their representation below. Parameters $k$ and $p$ require $O(\lg n + \lg \lg m)$ bits each and $\kappa$ and $r$ take $O(\lg n)$ bits each. The parameters for the secondary hash functions $k_1, k_2, \ldots, k_{(\lfloor \lg n \rfloor + 1)}$ are stored in an array, which takes $O((\lg n)^2)$ bits. The table $K$ contains, for its $i$th sequence of bits, the integer $\alpha_i$ in unary, if $k_{\alpha_i}$ is the multiplier (the secondary hash key) associated to hash the bucket $i$, $0 \leq i < n$. This table is an $O(n)$ bit string. We store a $o(n)$ bit auxiliary structure along with this bit string to support rank and select operations[9,5,10] on it in constant time. Using this and the array of multipliers, given an $i$, we can find the multiplier associated with the bucket $i$ in constant time. The table $C$, which contains the values $C_i$ $(= {c_0}^2 + {c_1}^2 + \cdots + {c_{i-1}}^2)$, is encoded as follows. First the values $c_i^2$ are stored in a table $T_0$ in unary notation (in order of appearance, separated by 0's), which is of length at most $4n$. We also store an auxiliary structure of $o(n)$ bits to support rank and select on both the bits, in constant time. Now, given an $i$, $C_i$ is nothing but the rank of the $i$th 0 (i.e. $C_i = rank_1(select_0(i))$ ), which can be found in constant time. For the compression table $D$, we store a bit string of length $3n$ where the $i$th bit is a 0 if $A^\star[i]$ is empty and 1 otherwise. We also store a $o(n)$ bit auxiliary structure to

support rank and select operations on this in constant time. When an element is hashed to a location in $D$, the rank of the bit in that location in the bit vector representation of $D$ gives the location of the element in the array $A$.

Now, to obtain rank for the element in the set, we could simply store the rank with each element in the FKS table. However this takes $n \lg m + n \lg n + O(n + \lg \lg m)$ bits. In the rest of this section, we describe how we can get rid of the $n \lg n$ term.

Pagh [12] has observed that each bucket $j$ of the hash table may be resolved with respect to the part of the universe hashing to bucket $j$. Thus we can save space by compressing the hash table part (i.e. table $A$ above) of the data structure, storing in each location not the element itself, but only a *quotient* information that distinguishes it from the part of $U$ that hashes to this location. The quotient function, slightly modified from that of Pagh is as follows:

$$q_{k,p}(x) = ((x \ div \ p).\lceil p/r \rceil + (k.x \ mod \ p) \ div \ n^2).\lceil r/n \rceil + (\kappa.z \ mod \ r) \ div \ n$$

where $z = (k.x \ mod \ p) \ mod \ n^2$ and the parameters $k$, $p$, $\kappa$ and $r$ are as defined in the FKS perfect hash function. It is easy to see that $q_{k,p}(x)$ for $x \in U$ is $O(m/n)$ (as $(\kappa.z \ mod \ r) \ div \ n < r/n$ and $(k.x \ mod \ p) \ div \ n^2 < p/r$).

Thus the total space to store all the quotient values along with the hash function will be $n \lg(m/n) + O(n + \lg \lg m)$ bits. To find an element, we compute its quotient value, apply the composite hash function to determine a location and check whether the quotient value appears in that location. Now, with each element we also store the rank of the element in the set for an extra space of $n\lceil \lg n \rceil$ bits. Thus if the element is found, we can get its rank from the rank information stored in its location.

Thus we have

**Theorem 1.** *A static dictionary for a subset $S$ of size $n$ of a finite universe $U = \{1, \ldots, m\}$ can be constructed using $n \lg m + O(n + \lg \lg m)$ bits of space so that membership and rank queries can be answered in $O(1)$ time.*

## 3 A Rank Structure Taking $n \lg m + O(\lg \lg m)$ Bits

In this section we illustrate a method by which the space used by the structure in the last section can be reduced by $cn$ bits for any parameter $c < (1 - \epsilon) \lg n$, $0 < \epsilon < 1$. The trick is to store only the last $(\lg n - c)$ bits of the rank instead of storing the entire value of the rank along with each element. Suppose the sorted list of the elements of the set is divided into $2^c$ blocks of size roughly $n/2^c$ each. Then the information stored with each element is precisely its rank within its block. In another array, we store the index of the $(in/2^c)$th element in the sorted order of the elements (i.e. the first element of the $i$-th block), for $1 \leq i \leq 2^c$.

Given an element, the membership proceeds as in the case of our modified FKS strategy (as mentioned in the last section). Once an element is found, the block to which it belongs (in the sorted order) can be found by doing a binary search (using $c$ steps) on the first elements of each block stored in the separate

array. The rank of an element within its block is stored with the element in the FKS dictionary. From these two information, we can obtain the rank of the element.

The space required, in addition to the $n \lg m/n + O(n + \lg\lg m)$ bits used to store the quotient values and the hash function information, is $n \lg n - cn + 2^c \lg n$ bits. Let the space occupied by the hash function and the auxiliary storage for the FKS dictionary be $d(n + \lg\lg m)$ bits. Choose $c$ such that $cn > 2^c \lg n + dn$. Then the total space requirement will be $n \lg m + O(\lg\lg m) - \Theta(n)$ bits. If $n$ is $\Omega(\lg\lg m)$, we can choose $c$ such that $cn > 2^c \lg n + d(n + \lg\lg m)$ so that the total space will be $n \lg m - \Theta(n)$ bits.

Note that we actually don't store the elements in the array locations, but store only the quotient value of the element in the location to which it hashes to. So we describe below, how given a location, we can actually find the element of the set, whose quotient is stored in that location, in constant time.

For this purpose, we store the values of $k^{-1}$ and $\kappa^{-1}$ along with other parameters, which require $O(\lg\lg m + \lg n)$ bits of extra space. Now, given a location $l$, let $q$ be the quotient value stored in that location and let $x$ be the actual element of the given set that hashes to that location.

Table $D$ (given in the last section) can be used to find the location $l^\star$ in the virtual array $A^\star$ in which the element should have been stored, using a select operation on the bit representation of $D$. Now, using the table $T_0$ (i.e. the compressed form of table $C$), we can find the bucket into which the element has hashed to, which is nothing but the value of $(\kappa.z \ mod \ r) \ mod \ n$. Also $q \ mod \ r/n$ gives us the value $(\kappa.z \ mod \ r) \ div \ n$. From these two values, we can find the value of $\kappa.z \ mod \ r$ from which, using $\kappa^{-1}$ we can find $z$. Note that $z$ is nothing but $(k.x \ mod \ p) \ mod \ n^2$. Now, $(q \ mod \ r/n) \ mod \ p/r$ gives the value of $(k.x \ mod \ p) \ div \ n^2$. Using these two values, we can find $(k.x \ mod \ p)$ from which the value of $x \ mod \ p$ can be found using the value of $k^{-1}$. Again, $(q \ mod \ r/n) \ div \ p/r$ gives the value of $x \ div \ p$. Using these two values, we can find the value $x$.

Thus we have,

**Theorem 2.** *There exists a static dictionary for a subset $S$ of size $n$ of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg\lg m) - \Theta(cn)$ bits of space and answers membership queries in constant time and rank queries in $O(c)$ time where $1 \leq c \leq (1 - \epsilon) \lg n$ for any positive constant $\epsilon < 1$.*

**Corollary 1.** *There exists a static dictionary for a subset $S$ of size $n$ of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg\lg m) - \Theta(n)$ bits of space and answers membership and rank queries in $O(1)$ time.*

**Corollary 2.** *There exists a static dictionary for a subset $S$ of size $n$ of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m - \Theta(n)$ bits of space and answers membership and rank queries in $O(1)$ time when $n = \Omega(\lg\lg m)$.*

Note the time-space tradeoff in the main theorem above. In particular, if we are willing to support the rank operation in $O(\lg \lg n)$ time, then space complexity comes down to $n \lg m + O(\lg \lg m) - \Theta(n \lg \lg n)$ bits.

### 3.1 Static Dictionary Supporting Select

Suppose we want to support only membership and select operations efficiently. To support select, besides the modified FKS dictionary to support membership, we can store in an array, the pointer to the $i$th smallest element of the set, for $1 \leq i \leq n$. This requires an additional $n \lg n$ bits of space.

To further reduce space, we again store only the last $\lg n - c$ bits (i.e. the position of the $j$th element within a block of size $n/2^c$) for some parameter $c$ (to be determined) and in a separate array store the index of the $(ni/2^c)$th element in the sorted order of the elements, for $1 \leq i \leq 2^c$. Given a $j$, to find the $j$th element, we do the following. Find the last $\lg n - c$ bits of the position of the $j$th element from the first array. Now for each choice of the first $c$ bits, find the element stored in the location given by the $\lg n$ bits. If that element lies between the elements ranked $n(j-1)/2^c$ and $nj/2^c$ (which can be found using the pointers stored in the second array), output that element as the $j$th element. Clearly, there will be a unique choice of the first $c$ bits, which gives the location of the $j$-th smallest element.

As in the last section, $c$ can be chosen in such a way that $cn > 2^c \lg n + dn$.

Thus we get

**Theorem 3.** *There exists a static dictionary for a subset $S$ of size $n$ of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg \lg m) - \Theta(cn)$ bits of space and answers membership queries in constant time and select queries in $O(2^c)$ time for any parameter $c < \lg n$.*

**Corollary 3.** *There exists a static dictionary for a subset $S$ of size $n$ of a finite universe $U = \{1, \ldots, m\}$ that uses $n \lg m + O(\lg \lg m) - \Theta(n)$ bits of space and answers membership and select queries in $O(1)$ time. When $n = \Omega(\lg \lg m)$, the space used is simply $n \lg m - \Theta(n)$ bits.*

### 3.2 Static Dictionary Supporting Rank and Select

When $n$ is a constant, we can support membership, rank and select using $n \lg m$ bits by storing the elements in a sorted array. Also when $n > \frac{m}{\lg m} + \frac{m \lg \lg m}{(\lg m)^2}$, we can store a bit vector of the subset ($m$ bits) and some auxiliary structures ($o(m)$ bits) to support membership, rank and select in constant time[5,3].

Fiat et. al.[7] have given a structure to store multi key records where search can be performed under any key in constant time. By storing an element and its rank as a two key record, using this structure, one can support membership, rank and select queries in constant time. This structure takes $n \lg mn + O(\lg \lg m + \lg n)$ bits of space.

Another obvious way to support both rank and select operations is to take either of the previous two structures (supporting rank or select) given in the last subsections and augment it with an array (of $n \lg n$) bits to support the other operation also in constant time. Thus we can support membership, rank and select in constant time using a structure that takes $n \lg mn + O(\lg \lg m) - \Theta(n)$ bits of space.

One can also find the rank by performing a binary search in a structure that supports membership and select. Thus we can support membership and select in constant time and rank in $\lg n$ time using a structure that takes $n \lg m + O(\lg \lg m) - O(n)$ bits of space.

It would be interesting to know whether we can support all these operations in constant time using $n \lg m + O(\lg \lg m) + o(n)$ bits.

## 4 Representing $m$-ary Cardinal Trees

In this section, we look at the problem of representing an $m$-ary cardinal tree. In this tree, each node has $m$ positions for an edge to a child, some of which can be empty. Benoit et. al.[3] have given an optimal representation of a cardinal tree that takes $n\lceil \lg m \rceil + 2n + o(n)$ bits and supports all navigational operations in constant time, except finding a child labeled $i$, which takes at most $O(\lg \lg m)$ time. This encoding has two parts. The first one uses the succinct encoding of ordinal trees [11,3] which takes $2n + o(n)$ bits to store an ordinal tree of $n$ nodes that supports all navigational operations (on ordinal trees) in constant time. In the second part, the $n\lceil \lg m \rceil$ bits of storage is used to store, for each node, $d\lceil \lg m \rceil$ bits to encode which children are present, where $d$ is the number of children at that node.

In this structure, when the given subset is very sparse (namely when $n \leq \lg m$), they store the elements in sorted order, so that a search for an element or finding the rank of it takes $O(\lg n)$ time. When $n > \lg m$ the universe is split into equal sized buckets and the values that fall into each bucket are stored using perfect hash functions. By choosing the number of buckets appropriately, one can make the space occupied by this structure to be at most $n \lg m$ bits.

We observe that by using a static dictionary that supports rank and membership in constant time that requires at most $n \lg m$ bits of space, we can construct a $m$-ary cardinal tree structure that supports all navigational operations in constant time.

If the number of children of a node is less than $\lg \lg m$, we store them in a sorted array. Membership and rank queries in this array can be answered in $O(\lg \lg \lg m)$ time. When $n$ is at least $\lg \lg m$, we store it using the structure of Corollary 4.

Thus we have

**Theorem 4.** *There exists an $n\lceil \lg m \rceil + 2n + o(n)$ bit representation of $m$-ary cardinal trees on $n$ nodes that supports the operations of finding the parent of a node or the size of the subtree rooted at any node in constant time and supports finding the child with label $j$ in $O(\lg \lg \lg m)$ time.*

When $n$ is $\omega(m^{\lg\lg m+1+o(1)})$, we propose an alternate structure. The idea is very similar to the one described in [2,3,4]. We follow the above encoding except for vertices whose degree is at most $\lg\lg m$. We construct a table in which each entry represents a set of size at most $\lg\lg m$ which is stored as an $m$ bit vector. Along each entry, we also store an auxiliary structure which takes $o(m)$ bits to support rank operation on the $m$ bit characteristic vector in constant time. We will have a two level ordering of the table. In the first level, we order the sets based on their cardinalities. In the second level, we order sets with the same cardinality lexicographically (in the bit vector representation). Now in the cardinal representation, when a node has degree at most $\lg\lg m$ instead of storing them in sorted order, we simply keep the position of the set in the second level of the table (since we can compute the cardinality of the set, which is the same as the degree of the node, from the ordinal information, we obtain the position in the first level of the table).

The space occupied by the table is $O(m^{\lg\lg m})(m+o(m))$ which is $o(n)$. The space used by the index at each small degree node is $\lg\binom{m}{d}$ which is at most $d\lg m$ bits, where $d$ is the degree of the node. Now for these nodes, to search for a child or to find its rank, we first find the subtree size from the ordinal information of the node and using the index stored with the node in the cardinal part, find the bitmap of the subset, which can be used to search for the element or find its rank (using the $o(m)$ auxiliary structure) in constant time.

Thus we have,

**Theorem 5.** *There exists an $n\lceil\lg m\rceil + 2n + o(n)$ bit representation of $m$-ary cardinal trees on $n$ nodes that supports all the navigational operations in constant time when $n$ is $\Omega(m^{\lg\lg m+1})$.*

## 5 Conclusions and Open Problems

We have given representations of static dictionaries that support rank (or select) and membership queries in constant time using $n\lg m + O(\lg\lg m)$ bits of space. This gives us a structure that supports rank, select and membership queries in constant time using $n\lg mn + O(\lg\lg m)$ bits of space. We also gave a representation of a $m$-ary cardinal tree that supports all navigational operations in constant time except finding a child with label $j$ which takes at most $O(\lg\lg\lg m)$ time using $\lceil n\lg m\rceil + 2n + o(n)$ bits of space.

Some open problems that arise/remain are:

- Find the space optimal structures for supporting membership, rank (for elements in the set) and/or select queries in constant time.
- Find a representation of $m$-ary cardinal trees that takes $\lceil n\lg m\rceil + 2n + o(n)$ bits of space and supports all navigational operations in constant time for all values of $n$.

## References


1. M. Ajtai, "A lower bound for finding predecessors in Yao's cell probe model", *Combinatorica* **8** (1988) 235-247.
2. D. Benoit, " Compact Tree Representations", Master's Thesis, Department of Computer Science, University of Waterloo, Canada (1998).
3. D. Benoit, E. D. Demaine, J. I. Munro and V. Raman "Representing Trees of Higher Degree", *The Proceedings of the 6th International Workshop on Algorithms and Data Structures*, Springer Verlag Lecture Notes in Computer Science **1663** (1999) 169-180.
4. A. Brodnik and J. I. Munro, "Membership in constant time and almost minimum space", to appear in *SIAM Journal on Computing*.
5. D. R. Clark, "Compact Pat Trees", Ph.D. Thesis, University of Waterloo, 1996.
6. D. R. Clark and J. I. Munro, "Efficient Suffix Trees on Secondary Storage", *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms* (1996) 383-391.
7. A. Fiat, M. Noar, J. P. Schmidt and A. Siegel, "Non-oblivious hashing", *Journal of the Association for Computing Machinery*, **39**(4) (1992) 764-782.
8. M. L. Fredman, J. Komlós and E. Szemerédi, "Storing a sparse table with $O(1)$ access time", *Journal of the Association for Computing Machinery*, **31** (1984) 538-544.
9. G. Jacobson, "Space-efficient Static Trees and Graphs", *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1989) 549-554.
10. J. I. Munro, "Tables", *Proceedings of the 16th FST & TCS conference*, Springer Verlag Lecture Notes in Computer Science **1180** (1996) 37-42.
11. J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs", *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1997) 118-126.
12. Rasmus Pagh, "Low redundancy in dictionaries with O(1) worst case lookup time", to appear in *Proceedings of the International Colloquium on Automata, Languages and Programming* (1999).
13. J. P. Schmidt and A. Siegel, "The spatial complexity of oblivious -probe hash functions", *SIAM Journal on Computing* **19**(5) (1990) 775-786.
14. D. E. Willard, "Log-Logarithmic worst case range queries are possible in space $\Theta(n)$", *Information Processing Letters* **17** (1983) 81-89.

# Multiple Spin-Block Decisions

Peter Damaschke

FernUniversität, Theoretische Informatik II
58084 Hagen, Germany
Peter.Damaschke@fernuni-hagen.de

**Abstract.** We study the online problem of holding a number of idle threads on an application server, which we have ready for processing new requests. The problem stems from the fact that both creating/deleting and holding threads is costly, but future requests and completion times are unpredictable. We propose a practical scheme of barely random discrete algorithms with competitive ratio arbitrarily close to $e/(e-1)$.

## 1 Introduction

A server in a network has to execute a large number of jobs due to requests from several clients. Multithreading is an approach to serve many requests simultaneously (either on parallel processors or scheduled by a multitasking operating system), such that short requests do not have to wait for completion of other time consuming jobs, which would be frustrating or even unacceptable. Each arriving job is assigned to some currently idle thread which is responsible for completing the job. If no thread is idle then a new thread is created. A busy thread that has finished a job becomes idle again.

The workload of the server, i.e. the number of parallel jobs, can fluctuate over time very much, but creating and deleting a thread is a costly operation. So we should have enough idle threads ready for future requests. On the other hand, running many threads on the spot over long periods would be a waste of processor cycles which could be better used by other applications residing on the same machine. Hence one has to observe a suitable strategy for deleting idle threads, without knowledge of future jobs.

Note that such an online policy is concerned with the *number* of jobs and idle threads only. In contrast to online scheduling problems (cf. [7]), it also makes no essential difference whether the completion times of current jobs are known or unknown. (The latter assumption is suitable e.g. if the jobs include select queries to databases, such that completion time depends on the previously unknown size of the answer). An adversary may both send arbitrarily short requests and extend completed job immediately by new ones, so the online player cannot exploit knowledge of execution times.

We assume that our idle threads do not retard progress of our busy threads (but take time from other concurrent applications instead), so the workload, i.e. number of busy threads, is *given* to the online player at any time and is beyond his influence. It is natural to assume fixed costs $C$ and $D$ for each create and

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 27–36, 1999.
© Springer-Verlag Berlin Heidelberg 1999

delete operation, respectively, and cost 1 for holding a thread one time unit long. W.l.o.g. let be $C + D = 1$ throughout the paper, that means, the time unit is chosen such that running an idle thread for 1 time unit is as expensive as a creation-deletion pair. We will find this choice to be very convenient.

Our online problem can be stated as follows: How should we assign new jobs to idle threads and which of the idle threads should be deleted at what time, in order to minimize the total costs? For heavily loaded servers, savings on this front may have a measurable effect [2]. In view of the practical relevance, the strategy should not only be competitive, but also easy to implement and, most importantly, computationally simple. (It would be foolish to save thread costs, but to compensate this by too large amount of additional data structures for thread administration.)

As we shall explain in Section 2, our problem is a generalization of the single rent-to-buy problem, where we need a resource for an unknown time interval and are allowed to rent it at price 1 per time unit or to buy it at an arbitrary moment at price 1. This problem and some of its variants are well-known under different names: leasing, spin-block, ski rental problem etc. Therefore we call *our* problem *multiple spin-block*.

Trivially, the best deterministic competitive ratio for single rent-to-buy is 2. In contrast, there is an $e/(e-1)$-competitive randomized algorithm against an oblivious adversary [5]. Throughout the paper this is called the KMMO algorithm. Note that any randomized rent-to-buy algorithm is nothing else than a probability distribution on the points $b$ in time at which we shall buy the resource. In the KMMO algorithm, the probability to buy before $b$ is a continuous function, namely $\int_0^b (e-1)^{-1} e^t dt$. Under the assumption that evenly distributed random reals $X$ from $[0, 1]$ are available, we may set $b = \ln(1 + (e-1)X)$.

In [6] a sequence of isolated rent-to-buy decisions is studied under the assumption that requests follow an unknown but fixed probability distribution, and the goal is to adapt the online player's strategy to this distribution; cf. this paper for further motivations and for pointers to empirical studies. In contrast, we consider concurrent threads which overlap in time, and we do not make probabilistic assumptions. The TCP acknowledgment delay problem [3] is of similar flavour as ours. The difference is that each acknowledgment has unit cost regardless the number of acknowledged packets, whereas our operations create or delete only one thread each. The call admission problem is also different, as requests may be rejected due to limited capacities, and the goal is to maximize the throughput (see e.g. [4]). For a general introduction to competitive analysis of online problems we refer to [1].

We believe that the main contribution of the present paper is a scheme of $r$-competitive multiple spin-block algorithms, with $r$ arbitrarily close to $e/(e-1)$. It is based on KMMO, but barely random and computationally simple. The proofs in this extended abstract are only sketched.

## 2 Decomposition of the Multiple Spin-Block Problem

The only relevant information from the input is a staircase function from the reals (time) into the nonnegative integers (number of running jobs), called the workload function $f$. Note that $f = 0$ outside the finite interval from the arrival of the first job until completion of the last job. Similarly, the outcome of a deletion algorithm is a function $g \geq f$ indicating the number of threads at each time.

We say that a staircase function $g$ has a downwards step at $t$ if $g$ decreases at $t$. Similarly, $g$ has an upwards step at $t$ if increases at $t$. There may be several upwards or downwards steps at the same $t$. The ordinate of an upwards/downwards step is the function value before/after the step. A down-up pair is a downwards step together with the next upwards step *on the same ordinate*. (Think of matching open and close parentheses in an arithmetic expression.) The width of a down-up pair is the time distance between the downwards and upwards step. Note that $\max_t f(t)$ upwards and downwards steps, respectively, are not involved in down-up pairs. We may consider them as down-up pairs of infinite width.

The cost of $g$ obviously consists of the following summands: $C$ times the number of upwards steps, $D$ times the number of downwards steps, and the area bounded by the graph of $g$ and the time axis. (Since the costs of busy threads must be paid anyhow, we may replace the last term with the area between the graphs of $g$ and $f$. In this case we only consider the overhead for create/delete operations and idle threads.) The optimal offline cost for a load function $f$ is the minimum cost of some $g$ with $g \geq f$.

The following offline algorithm is called BRIDGES: Consider any workload function $f$. We fix $g$ along the time axis. Start with $g = 0$ before the left endpoint of $I$. Whenever $g = f$ and $f$ increases then, clearly, we further keep $g = f$. But if $g = f$ and $f$ decreases at $t$ then $g$ changes to value $\min\{f(t), \max_{0<h<1} f(t+h)\}$. (Finally $g$ becomes 0.) Figuratively speaking, $g$ builds bridges over all valleys of $f$ shorter than 1, and $g = f$ elsewhere. Note that the lookahead of BRIDGES is bounded by $C + D = 1$.

**Lemma 1.** *For every workload function, BRIDGES yields the unique optimal solution.*

*Proof.* Consider an arbitrary algorithm. It necessarily incurs cost $C$ for every upwards step which is the first on its ordinate. Similarly, it incurs cost $D$ for every downwards step which is the last on its ordinate. For every down-up pair of width $w$, it pays either $h + 1$ if it deletes a thread $h$ time units after the downwards step and opens a new thread with the corresponding upwards step, or it pays $w$ if it lets the thread spin. Hence the best is to delete one thread immediately if $w \geq 1$ and to simply continue if $w < 1$. This is exactly what BRIDGES does, so optimality follows. A more formal proof would use induction on the number of steps. □

Next we present two straightforward but different generalizations of the 2-competitive deterministic solution to the rent-to-buy problem. The first one is:

EXPIRY DATE STACK
Maintain a stack of threads.
(1) When a thread becomes idle at time $t$, assign expiry date $t+1$ to it, and add it to the stack.
(2) When a new job arrives, assign it to the last idle thread on the stack and remove the now busy thread from the stack.
(3) When an expiry date is reached, remove the thread from the bottom (!) of the stack and delete it.

Note that the expiry dates are monotone on the stack. Although (3) is not a stack operation, we use the term "stack" to stress the fact that idle threads are added and assigned to jobs on a last-in-first-out basis. If threads are deleted by a central supervisor, it doesn't matter which thread is deleted or receives a new job. Then only the expiry dates must form a stack.

The function $s$ produced by this algorithm mimics the monotone decreasing parts of $f$ with delay 1, thereby keeping $s \geq f$.

**Theorem 1.** *EXPIRY DATE STACK is* 2*-competitive.*

*Proof.* EXPIRY DATE STACK incurs cost 2 for each pair of an upwards and downwards step which is the first and last, respectively, on its ordinate. This is clear since it waits for 1 time unit after every such downwards step before deleting one thread. The optimum would be 1. Additionally, for every down-up pair of width $w$, EXPIRY DATE STACK obviously pays $w$ if $w < 1$, and 2 else, whereas the optimum cost is $\min\{w, 1\}$. Together this implies the result. A more formal proof might be given by induction. □

EXPIRY DATE STACK is easy enough to implement. However it might be more convenient to maintain only constantly many numbers in order to fix the next expiry date, instead of a stack. This suggests a simpler algorithm:

CUMULATIVE IDLE COSTS
Start with $H = 0$. Whenever the total cost $H$ of holding the idle threads reaches 1, delete one thread and reset $H$ to zero.

Clearly, the next expiry date can be computed in advance from the current $H$ and the number of idle threads, and can easily be recomputed whenever the number of idle threads changes. So one has always to store only two numbers.

**Theorem 2.** *CUMULATIVE IDLE COSTS is also* 2*-competitive.*

*Proof.* Let $f$ be the given load function, $g$ the function that BRIDGES would produce, and $h$ the function produced by CUMULATIVE IDLE COSTS. We charge every downwards step of $h$ with costs at most 2, namely $D$ for the downwards step itself, 1 for the idle threads until deletion of the next thread, and $C$ for the next upwards step on the same ordinate (if existing). Note that every downwards step of $h$ which is not below the graph of $g$ can be considered as a delayed downwards step of $g$ at the same ordinate. Since BRIDGES pays $C$ and

$D$ for every upwards and downwards step of $g$, CUMULATIVE IDLE COSTS pays at most twice the optimum as long as $h \geq g$. For time intervals with $h < g$, the argument for ratio 2 is a bit different: The total cost incurred by BRIDGES on such intervals is the area between the graphs of $g$ and $f$. The payments of 2 made by CUMULATIVE IDLE COSTS correspond to disjoint subareas of the mentioned area, each of size 1. □

**Remarks:**

(1) Since 2 is the optimal competitive ratio already for the special case of the single rent-to-buy problem, these algorithms are strongly competitive.

(2) Competitive ratios are understood with respect to the costs of create/delete operations and idle threads only. As an immediate corollary, the algorithms are also 2-competitive with respect to the costs of create/delete operations, idle *and* busy threads.

A competitive ratio below 2 can be obtained by randomization. Consider an arbitrary but fixed randomized algorithm $R$ for rent-to-buy, with expected competitive ratio $r$ against an oblivious adversary. For example, take the KMMO algorithm with $r = e/(e-1) \approx 1.58$.

An obvious randomized version of CUMULATIVE IDLE COSTS might come first into mind:

CUMULATIVE IDLE COSTS ($R$)
Start with $H = 0$. Proceed with the total cost $H$ of holding the idle threads as $R$ would do with the rent cost. When R buys then delete one thread and reset $H$ to zero.

Unfortunately this attempt fails, as the following heuristic consideration shows. Let $h$ be the (random) function produced by CUMULATIVE IDLE COSTS ($R$). Consider a "canyon" in $f$, consisting of $n$ downwards steps at the same time, followed by $n$ upwards steps $t$ time units later, where $t < 1$. BRIDGES pays $tn$ there. Since R has expected competitive ratio $r$, the expected area below $h$ until the $(k+1)$-th deletion inside the canyon is $\sum_{i=n-k}^{n}(r-1)/i$. For large enough $n$ we may assume that the number $k$ of deletions satisfies $\sum_{i=n-k}^{n}(r-1)/i \approx t$. Since the harmonic numbers grow as the ln function, this gives $k \approx n(e^y - 1)/e^y$, where $y = t/(r-1)$. Since CUMULATIVE IDLE COSTS ($R$) pays $kr$ ($C + D = 1$ for every deletion and creation, and $k(r-1)$ for the area below $h$), we would obtain a competitive ratio $kr/tn \approx r/(r-1)$ for $f$ consisting of a sequence of such canyons with small $t$. Ironically, this is larger than 2 just because of $r < 2$. This example suggests to pay attention to the moments when the threads became idle. So we argue that randomization of EXPIRY DATE STACK is the proper way.

EXPIRY DATE STACK ($R$)
Maintain a stack of threads.
(1) When a thread becomes idle, add it to the stack.
(2) When a new job arrives, assign it to the last idle thread in the stack and remove the now busy thread from the stack.

(3) Simultaneously delete idle threads (at any position!) according to $R$, and always retain the ordering of surviving threads on the stack.

**Theorem 3.** *EXPIRY DATE STACK ($R$) is $r$-competitive. More precisely, the expected cost is at most $r$ times the optimum, in each down-up pair of the workload function.*

*Proof.* Consider a "stubborn" version of EXPIRY DATE STACK ($R$) where the list elements do not move after deletion of middle elements. Instead we leave the gaps on the stack, standing for idle threads which have already expired, and when such a thread is requested from the end of the list then we create a new one. For this algorithm, $r$-competitiveness follows quite easily from linearity of expectation, if we decompose the workload function into down-up pairs. It is fairly obvious that the original algorithm is not more costly: It picks up the next idle thread from the stack as long as there is one. Hence creations are postponed or even avoided, and idle costs are saved. From this the assertion follows. □

EXPIRY DATE STACK ($R$) is highly random if we apply $R$ independently to all threads that become idle. On the other hand, the proof of $r$-competitiveness does not rely on independent applications of $R$ at all. (Linearity of expectation holds for arbitrary random variables.) So we might even fix a single $b$ at random, and then use it everywhere! The obvious drawback is that certain workload functions can foil such a solution, i.e. produce an actual competitive ratio significantly larger than $r$. (For example, consider a 0,1-valued $f$ where the $f = 0$ intervals have length $b + \varepsilon$.) On the other hand, independent applications of $R$ make the competitive ratio sharply concentrated around $r$ for any large enough input. So we have a trade-off between randomness and variation. Nicely, we can remove most randomness from EXPIRY DATE STACK ($R$) yet keeping the competitive ratio sharply concentrated at $r$. For this however $R$ must be a discrete rent-to-buy algorithm as we introduce next.

## 3 Discrete Randomized Rent-to-Buy

We return to the single rent-to-buy problem.

**Definition 1.** *A discrete rent-to-buy algorithm $R$ with denominator $n$ is a probability distribution on a set of $n$ points $0 \le t_1 \le \ldots \le t_n$, such that $R$ buys, for all $k$, after $t_k$ time units with probability $1/n$. Let $r$ denote the expected competitive ratio of $R$.*

The question is to find points $t_k$ so as to minimize $r$, for fixed denominator $n$. This is also interesting for its own, since a discrete strategy does not need real computations, unlike the continuous KMMO algorithm. The $t_k$ can be computed once and stored in a table. Note that our problem is "orthogonal" to that of randomized snoopy caching solved in [5] where $t_k = k/n$ are fixed and the probabilities are the variables.

Let $t$ denote the duration (unknown to the online player) the resource is needed for. For convenience let $t_0 = 0$ and $t_{n+1} = \infty$. We express $r$ by $r = \max r_k$ where $r_k$ is the worst-case competitive ratio if $t_k \le t < t_{k+1}$. Note that $r_0 = 1$ is redundant. Define $s_k = \sum_{i=1}^{k} t_i$. The expected cost $R$ incurs is $(s_k + k + (n-k)t)/n$. If $t_{k+1} \le 1$ then $r_k$ is the cost divided by $t$, which is maximized if $t = t_k$. Hence $r_k = ((s_k+k)/t_k+n-k)/n$ in this case. If $1 < t_k$ then $r_k$ is maximized if $t = t_{k+1}$, hence $r_k = (s_k + k + (n-k)t_{k+1})/n$, particularly $r_n = (s_n + n)/n$. Finally, if $t_k \le 1 < t_{k+1}$ then $r_k$ is the maximum of both expressions.

**Lemma 2.** *In an optimal $R$ with denominator $n$ we have $t_n = 1$, and $r = 1 + n^{-1} \max_k((s_k + k)/t_k - k)$.*

*Proof.* Assuming $t_n > 1$, let $u$ be that index with $t_u \le 1 < t_{u+1}$. The $t_k$, $k > u$ do not appear in the $r_k$, $k \le u$, and the $r_k$, $k > u$ are monotone increasing in all $t_i$. Hence for any fixed $t_1, \ldots, t_u$ we get minimum $r$ if $t_{u+1} = \ldots = t_n$ instead of the given values. This shows $t_n \le 1$. We conclude that $r_k = ((s_k + k)/t_k + n - k)/n$ for all $k$. Now assume $t_n < 1$. If we multiply all $t_i$ by the same factor $a > 1$ then all $r_k$ decrease. Hence $r$ is minimized if we choose $a$ possibly large, but this means $t_n = 1$. □

Since $r_k$ is monotone decreasing in $t_k$ and monotone increasing in all previous $t_i$, we get minimum $r$ if $r_1 = \ldots = r_n$. So Lemma 2 yields $n - 1$ algebraic equations in $n - 1$ variables $t_k$, $k < n$. We illustrate the application in case $n = 2$:

**Proposition 1.** *The best $R$ with denominator 2, given by $t_1 = (\sqrt{5}-1)/2 \approx .62$, has competitive ratio $(5 + \sqrt{5})/4 \approx 1.81$.*

*Proof.* By Lemma 2, $r_1 = 1+1/2t_1$ and $r_2 = 1+(t_1+1)/2$. Thus $t_1^2+t_1-1 = 0$. □

Denominators $n > 2$ lead to higher-order algebraic equations that may be solved numerically. For $n = 3$ we get $t_1 \approx .45$, $t_2 \approx .78$, and $r \approx 1.75$, etc. It is important to estimate the competitive ratio for fixed $n$. Due to the following result, some $n$ between 5 and 10 should be satisfactory in practice:

**Theorem 4.** $e/(e-1) + 1/2n - 1/n^2 < r < e/(e-1) + 1/2n$.

*Proof.* Choose $R$ with $t_k = \ln(1 + (e-1)k/n)$. Recall the following properties of any $R$ with $t_n \le 1$: We have $r = \max r_k$ where the $r_k$ are the expressions from Lemma 2, and the worst-case ratio of expected cost and $t$ occurs at some $t = t_k$. Now we compare the expected costs of $R$ and KMMO at any fixed $t = t_k$. By our choice of $t_k$, the probability to buy until $t_k$ is $k/n$ in both algorithms. It remains to compare the expected rent time. The probability to buy the resource in interval $(t_i, t_{i+1}]$ is $1/n$, but $R$ defers the buy decisions of KMMO until $t_{i+1}$. Hence the contribution of every such interval (for $0 \le i < k$) to the expected

rent time of $R$ exceeds that of KMMO by at most $(t_{i+1} - t_i)/2n$. (Since the density function $e^t/(e-1)$ used in KMMO is monotone, the average delay is at most the half interval length.) Hence the total excess is at most $t_k/2n$. Since in $r_k$ the costs are divided by $t_k$, the assertion follows.

For the lower bound, consider any discrete rent-to-buy algorithm $R'$. By Lemma 2 we may w.l.o.g. assume $t'_n = 1$. Let $k$ be the first index with $t'_k \leq t_k$. Clearly, $r' \geq r'_k \geq r_k$. Hence it suffices to prove the asserted lower bound for each $r_k$ of our particular $R$. Similarly as above, it is not hard to observe geometrically that the contribution of interval $(t_i, t_{i+1}]$ to the excess of expected rent time is at least $a(t_{i+1} - t_i)/2n$, where $a \geq 2n/(2n+e-1)$. (Figuratively speaking, since the density function is monotone, at least a $2e^{t_i}/(e^{t_{i+1}} + e^{t_i})$ fraction of the probability mass has an average delay of exactly the half interval length. Since $e^{t_i} = 1 + (e-1)i/n$, the worst case is with $i = 0$, which yields our factor $a$.) From this we obviously get the square term. □

A tighter analysis may improve the lower bound.

## 4 Barely Random Multiple Spin-Block Algorithms

Let $R$ be a fixed discrete rent-to-buy algorithms with denominator $n$ and $t_n \leq 1$. Our scheme works as follows. Choose a fixed permutation $\pi$ of the first $n$ positive integers, and a real parameter $h \leq t_1$, may be $h = t_1$. Divide the time axis by lattice points into slices of length $h$. Let $T(l)$ be the set of $m$ threads that became idle between $l - h$ and $l$, and are still idle at $l$ (that means, are not removed again from the stack by new jobs). Note that $R$ would not delete any jobs from $T(l)$ before $l$, since they have been idle for less than $t_1$ time units. So we may fix the expiry dates of all idle threads in $T(l)$ still at time $l$. This is done in the following way. Put the threads of $T(l)$ in a round robin fashion into $n$ subsets, in the ordering they became idle. In other words, any $n$ consecutive threads of $T(l)$ belong to different subsets. Choose a random integer $x \in \{1, \ldots, n\}$. Assign idle time $v = t_{\pi(k)+x} \bmod n+1$ to all threads in the $k$-th subset. (That means, the expiry date of a thread is $u + v$ if it became idle at $u$.)

Thus every idle thread in $T(l)$ is deleted according to $R$, just as in EXPIRY DATE STACK ($R$). Due to Theorem 3 and linearity of expectation, this version of EXPIRY DATE STACK ($R$) is also $r$-competitive. Moreover it uses a fixed number of independent random integers per time unit, regardless of the workload function $f$. Most pleasantly, $r$ is not only the expected competitive ratio. We can give a stronger guarantee if the workload is high:

**Proposition 2.** *In every time unit $U$, the total cost of all but constantly many down-up pairs of $f$ starting in $U$ is surely (!) at most $r$ times the optimum.*

*Proof.* For simplicity consider the stubborn version of the algorithm, as in Theorem 3. So every thread is assigned to a fixed down-up pair of $f$, as soon as it becomes idle. Partition every $T(l)$ into $T(l)_k$ $(0 \leq k \leq n)$, the sets of threads in $T(l)$ being responsible for down-up pairs of $f$ of width between $t_k$ and $t_{k+1}$.

These are contiguous subsets of threads on the stack. Further partition each $T(l)_k$ into contiguous blocks of exactly $n$ threads. There remain less than $n$ threads in each $T(l)_k$, at most $n^2$ threads in each $T(l)$, and $O(n^3)$ threads in each time unit, since $t_1 \approx (e-1)/n$ in an optimum $R$. (The latter is not hard to see from previous section.) Due to the "modulo construction", each block contains exactly one thread with idle time $t_j$ $(1 \leq j \leq n)$. From this, the formula in Lemma 2 implies that the cost of each block in any $T(l)_k$ is within $r_k$ times the optimum. Finally note $r_k \leq r$. □

Hence, though the amount of randomness per time is constant, the competitive ratio concentrates around $r$ the better, the more down-up pairs $f$ has. Roughly speaking, this follows by standard arguments (Chernoff bounds) in the "horizontal direction" (time), and from Proposition 2 in the "vertical direction" (workload).

It is worthwhile to mention some implementation details:

(1) Note that Proposition 2 does not rely on the fixed cyclic ordering $\pi$ of the $t_k$ values. Thus we are free to choose $\pi$ such that the $t_k$ are "well mixed" in the following sense: For each $k$ and $m$, all segments of length $m$ in the cyclic ordering should contain almost the same number of $t_i$ values with $i \leq k$. It is intuitively clear that this little effort improves the variance of costs on the $O(n^3)$ remaining threads mentioned in Proposition 2. The following is a good choice for any $n$: Let $\phi = (3-\sqrt{5})/2$ (thus $1-\phi$ is the golden ratio). Define a total ordering $\prec$ by $t_i \prec t_j$ iff $i\phi - \lfloor i\phi \rfloor < j\phi - \lfloor j\phi \rfloor$, and use the cyclic ordering obtained from $\prec$.

(2) In the proof it was convenient to assume that idle times are assigned to all threads in $T(l)$ at time $l$, after a random cyclic shift in $\pi$, but we may also assign idle times according to $\pi$ immediately whenever a thread becomes idle, and apply a random shift to $\pi$ after every $h$ time units. This simplifies the code.

(3) A crucial point in a real implementation is to handle the deletion of threads. Using a timer for each thread is out of the question. It is too expensive and may even cause a server crash [2]. Instead we should use $n$ timers, one for each set $T_k$ of idle threads whose idle time is fixed to be $t_k$. The $k$-th timer notifies the program if the earliest expiry date in $T_k$ is reached. Then we delete one thread and set the alarm for the next expiry date in $T_k$ which is found in $n$ (i.e. constantly many) expected steps: Since the $t_k$ are drawn from a cyclic ordering, interrupted by random shifts, we may simply search the stack bottom-up, until we meet a thread from $T_k$. To appreciate this property, note that in the basic version of EXPIRY DATE STACK ($R$) (cf. Theorem 3) we have no such guarantee, so we would need an additional data structure to quickly find the next expiry date in the stack.

## References

1. A. Borodin, R. El-Yaniv: *Online Computation and Competitive Analysis*, Cambridge Univ. Press 1998
2. T. Damaschke: private communication on the business component software project DamBuKo
3. D.R. Dooly, S.A. Goldman, S.D. Scott: TCP dynamic acknowledgment delay: theory and practice, *30th STOC'98*, 389-398
4. J.A. Garay, I.S. Gopal, S. Kutten, Y. Mansour, M. Yung: Efficient on-line call control algorithms, *J. of Algorithms* 23 (1997), 180-194
5. A.R. Karlin, M.S. Manasse, L.A. McGeoch, S. Owicki: Competitive randomized algorithms for non-uniform problems, *Algorithmica* 11 (1994), 542-571
6. P. Krishnan, P.M. Long, J.S. Vitter: Adaptive disk spindown via optimal rent-to-buy in probabilistic environments, *Algorithmica*, to appear. Extended abstract in: *12th Int. Conf. on Machine Learning*, Morgan Kaufmann 1995
7. R. Motwani, S. Phillips, E. Torng: Nonclairvoyant scheduling, *Theor. Comp. Sc.* 130 (1994), 17-47; extended abstract in: *4th SODA'93*, 422-431

# Asynchronous Random Polling Dynamic Load Balancing

Peter Sanders

Max-Planck-Institut für Informatik,
Im Stadtwald, 66123 Saarbrücken, Germany.
E-mail: sanders@mpi-sb.mpg.de,
WWW: http://www.mpi-sb.mpg.de/~sanders

**Abstract.** Many applications in parallel processing have to traverse large, implicitly defined trees with irregular shape. The receiver initiated load balancing algorithm *random polling* has long been known to be very efficient for these problems in practice. For any $\epsilon > 0$, we prove that its parallel execution time is at most $(1+\epsilon)T_{\text{seq}}/P + \mathcal{O}(T_{\text{atomic}} + h(\frac{1}{\epsilon} + T_{\text{rout}} + T_{\text{split}}))$ with high probability, where $T_{\text{rout}}$, $T_{\text{split}}$ and $T_{\text{atomic}}$ bound the time for sending a message, splitting a subproblem and finishing a small unsplittable subproblem respectively. The *maximum splitting depth* $h$ is related to the depth of the computation tree. Previous work did not prove efficiency close to one and used less accurate models. In particular, our machine model allows asynchronous communication with nonconstant message delays and does not assume that communication takes place in rounds. This model is compatible with the LogP model.

## 1 Introduction

Many algorithms in operations research and artificial intelligence are based on the backtracking principle for traversing large irregularly shaped trees that are only defined implicitly by the computation [3,4,6,9,12,13,14,19,17,21,35]. Similar problems also play a role in parallel programming languages [1,16]. Even for loop scheduling and some numerical problems [7,24] like adaptive numerical integration [25] it can be useful to view the computations as an implicitly defined tree (refer to [34] for a more detailed discussion of examples).

For parallelizing tree shaped computations, a load balancing scheme is needed that is able to evenly distribute the parts of an irregularly shaped tree over the processors. It should work with minimal interprocessor communication and without knowledge of the shape of the tree. Load balancers often suffer from the dilemma that subtrees which are not subdivided turn out to be too large for proper load balancing whereas excessive communication is necessary if the tree is shredded into too many pieces.

We consider *random polling* dynamic load balancing [19] (also known as *randomized work stealing* [5,10,2,11]), a simple algorithm that avoids both problems: Every processing element (PE) handles at most one piece of work (which may represent a part of a backtracking tree) at any point in time. If a PE runs out of

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 37–48, 1999.
© Springer-Verlag Berlin Heidelberg 1999

work, it sends requests to randomly chosen PEs until a busy one is found which splits its piece of work and transmits one to the requestor.

We continue this introduction by explaining the machine model in Section 1.1 and the problem model *tree shaped computations* in Section 1.2. Section 1.3 reviews related work and summarizes the new contributions. The main body of the paper begins with a more detailed description of random polling in Section 2. In Section 3 we then give expected time bounds and show in Section 4 that they also hold with high probability (using additional measures). Finally, Section 5 summarizes the paper and discusses some possible future research.

### 1.1 Machine Model

We basically adopt the LogP model [8] due to its simplicity and genericity. There are $P$ PEs numbered 0 through $P-1$. We assume a word length of $\Omega(\log P)$ bits.[1] Arithmetics on numbers of word length – including random number generation – is assumed to require constant time. All messages delivered to a PE are first put into a single FIFO *message queue*. In the full LogP model, three parameters for "latency" $L$, "overhead" $o$ and "gap" $g$ contribute to the cost of message transfer. We make the more conservative assumption that sending and receiving messages always costs $T_{\text{rout}} := L + o + g$ units of time. So the analysis also applies to the widespread messaging protocols that block until a message has been copied into the message queue of the recipient.

### 1.2 Tree Shaped Computations

We now abstract from the applications mentioned in the introduction by introducing *tree shaped computations* which expose just enough of their common properties in order to parallelize them efficiently. All the work to be done is initially subsumed in a single *root problem* $I_{\text{root}}$. $I_{\text{root}}$ is initially located on PE 0. All other PEs start idle, i.e., they only have an *empty problem* $I_\emptyset$.

What makes parallelization attractive, is the property that problem instances can be subdivided into *subproblems* that can be solved independently by different PEs. For example, a subproblem could be "search this subtree by backtracking" or "integrate function $f$ over that subinterval". We model this property by a *splitting operation* split($I$) that splits a given (sub)problem $I$ into two new subproblems subsuming the parent problem. Let $T_{\text{split}}$ denote a bound on the time required for the split operation. For example, in backtracking applications a subproblem is usually represented by a stack and splitting can be implemented by copying the stack and manipulating the copies in such a way that they represent disjoint search spaces covering the original search space [26].

The operation work($I, t$) transforms a given subproblem $I$ by performing sequential work on it for $t$ time units. The operation also returns when the subproblem is exhausted.

[1] Throughout this paper $\log x$ stands for $\log_2 x$.

What makes parallelization difficult, is that the *size*, i.e., the execution time $T(I) := \min\{t : \text{work}(I,t) = I_\emptyset\}$, of a subproblem cannot be predicted. In addition, the splitting operation will rarely produce subproblems of equal size. For the analysis we assume however that $\forall I : \text{split}(I) = (I_1, I_2) \implies T(I) = T(I_1)+T(I_2)$ regardless when and where $I_1$ and $I_2$ are worked on. For a discussion when this assumption is strictly warranted and when it is a good approximation, refer to Section 5 and to [32,34].

Next we quantify some guaranteed "progress" made by splitting subproblems. Every subproblem $I$ belongs to a *generation* $\text{gen}(I)$ recursively defined by $\text{gen}(I_{\text{root}}) := 0$ and $\text{split}(I) = (I_1, I_2) \implies \text{gen}(I_1) = \text{gen}(I_2) = \text{gen}(I) + 1$. For many applications, it is easy to give a bound on a *maximum splitting depth* $h$ which guarantees that the size of subproblems with $\text{gen}(I) \geq h$ cannot exceed some *atomic grain size* $T_{\text{atomic}}$. For example, a backtracking search tree of depth $d$ and maximum branching factor $b$ is easy to split in such a way that $h \leq d\lceil \log b \rceil$. We want to exclude problem instances with very little parallelism and therefore assume $h \geq \log P$. Otherwise, we might quickly end up with less than $P$ atomic pieces of work that cannot be split any more. Since $h$ is the only factor that constrains the shape of the emerging "subproblem splitting tree", it can be viewed as a measure for the irregularity of the problem instance. (Obviously, very regular instances with large $h$ are possible. But in applications where this is frequently the case, one should perhaps look for a splitting function exploiting these regularities to decrease $h$.)

Finally, subproblems can be moved to other PEs by sending a single message. If problem descriptions are long, the parameters of the LogP model must be adapted to reflect the cost of such a long message. The resulting time bounds will be conservative since many messages are much shorter.

The task of the algorithm analysis is now to bound the parallel execution time $T_{\text{par}}$ required to solve a problem instance of size $T_{\text{seq}} := T(I_{\text{root}})$ given the problem parameters $h$, $T_{\text{split}}$ and $T_{\text{atomic}}$ and the machine parameters $P$ and $T_{\text{rout}}$. The bound is represented in the form

$$T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{seq}}}{P} + T_{\text{rest}}(P, T_{\text{rout}}, \epsilon, h, \dots) \tag{1}$$

where $\epsilon > 0$ represents some small value we are free to choose. So, for situations with $T_{\text{rest}} \ll T_{\text{seq}}/P$ we have a highly efficient parallel execution.

### 1.3 Related Work and New Results

There is a quite large body of related research so that we can only give a rough outline. Many algorithms use a simpler approach regarding tree decomposition by requiring all "splits" to occur before calls to "work" (in our terminology). However, this is only efficient for some applications since in the worst case a huge number of subproblems may have to be generated or communicated (e.g. [18,7,27]).

Random polling belongs to a family of *receiver initiated* load balancing algorithms which have the advantage to split subproblems only on demand by

idle PEs. This adaptive approach has been used successfully for a variety of purposes such as parallel functional [1] and logic programming [16] or game tree search [12]. Randomized partner selection goes at least back to [13]. The partner selection strategy turns out to be crucial. The apparently economic option to poll neighbors in the interconnection network can be extremely inefficient since it leads to a buildup of "clusters" of busy PEs shielding large subproblems from being split [26]. Polling PEs in a "global round robin" fashion [18] avoids this because no large subproblems can "hide". Execution times $T_{\text{par}} \in \mathcal{O}(\frac{T_{\text{seq}}}{P} + hT_{\text{count}})$ can be achieved where $T_{\text{count}}$ is the time for incrementing a global counter. However, even sophisticated distributed counting algorithms have $T_{\text{count}} \in \Omega(T_{\text{rout}} \log P / \log\log P)$ [36]. It was long known that random polling performs better than global round robin in practice although the first analytical treatments could only prove an asymptotically weaker bound $\mathbf{E}T_{\text{par}} \in \mathcal{O}(\frac{T_{\text{seq}}}{P} + hT_{\text{rout}} \log P)$ [18]. Tree shaped computations are a generalization of the $\alpha$-splitting model used in [18]. The gap between analysis and practical experience was closed in [28,29] by showing that $T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{seq}}}{P} + \mathcal{O}(hT_{\text{rout}})$ with high probability using synchronous random polling.

Slightly later, random polling (also called randomized work stealing) was found to be very efficient for scheduling multithreaded computations [5]. For many underlying applications, the two models can be translated into each other. The critical path length $T_\infty$ in multithreaded computations then becomes $hT_{\text{split}} + T_{\text{atomic}}$ for tree shaped computations. Multithreading can model predictable dependencies between subproblems while tree shaped computations allow for different splitting strategies which may significantly decrease $h$ [26]. Multi-threaded computations are most easy to use with programming language support, while tree shaped computations are directly useful for a portable and reusable library [31,34]. In the following, we concentrate on tree shaped computations. Adapting these results to multithreading or some more general model encompassing both approaches is an interesting area for future work however.

All the analytical results above (including [28,29]) make simplifying assumptions that are unrealistic for large systems, difficult to implement or detrimental to practical performance. The most common assumption is that communication takes place in synchronized communication rounds. This is undesirable since idle PEs have to wait for the next communication round and the network capacity is left unexploited most of the time. In fact, actual implementations are usually asynchronous. Arora et al. allow small speed fluctuations ($2C$–$3C$ instructions per round) but even that may be diffucult to attain since the number of clock cycles needed per instruction can be highly data dependent on modern processors (e.g., cache faults for large inputs). They also assume that polling and splitting take constant time ($T_{\text{rout}} + T_{\text{split}} \in \mathcal{O}(1)$ in our terminology). This is a viable assumption for moderate size shared memory machines and the thread stack of a multithreaded language. But we want an algorithm that scales to large distributed memory machines and allows more sophisticated application specific splitting functions. Using an even simpler stochastic model, Mitzenmacher was able to analyze many variants of work stealing [23].

Unfortunately, we cannot fully transfer an analysis for the above "round models" to a realistic asynchronous model since subproblems that are "in transit" cannot be split and long request queues can build up around PEs that have "difficult to split" subproblems. In Section 3 we solve these analytical problems and show that $\mathbf{E}T_{\text{par}} \leq (1+\epsilon)T_{\text{seq}}/P + \mathcal{O}(T_{\text{atomic}} + h\,(1/\epsilon + T_{\text{rout}} + T_{\text{split}}))$. In Section 4 it turns out that this bound also holds with high probability although for some values of $h$ it may be necessary to actively trim long queues.

The time bound is not only tight for random polling but in [32] we also show a number of lower bounds which come very close: There are tree shaped computations for which a splitting overhead of $\Omega(hT_{\text{split}})$ is unavoidable so that we get an $\Omega(T_{\text{seq}}/P + T_{\text{atomic}} + hT_{\text{split}})$ lower bound. Furthermore, any receiver initiated load balancing algorithm not only needs $\Omega(h)$ communications on the critical path but also $\Omega(hP)$ full size messages overall so that the network bandwidth is fully utilized. Wu and Kung [37] show that a similar bound holds for all deterministic algorithms. Random polling *can* be slightly improved on certain networks by carefully increasing the average locality of communication [30]. At least up to constant factors, similar results can be achieved by dynamic tree embedding algorithms (e.g. [15]).

## 2 The Algorithm

Figure 1 gives pseudo-code for the basic random polling algorithm. PE 0 is initialized with the root problem as specified in the model. PEs in possession of nonempty subproblems do sequential work on them but poll the network for incoming messages at least every $\Delta t$ time units and at most every $\alpha\Delta t$ time units for any constant $\alpha < 1$.[2] When a request is received, the local subproblem is split and one of the new subproblem is sent to the requestor. Idle PEs send requests to randomly determined PEs and wait for a reply until they receive a nonempty subproblem. Requests received in the meantime are answered with an empty subproblem. Note that an empty subproblem can be coded by a short message equivalent to a rejection of the request.

Concurrently, a distributed termination detection protocol is run that recognizes when all PEs have run out of work. We have adapted the *four counter* method [22] for this purpose. Each PE counts the number of sent and received messages that contain nonempty subproblems. When the global sum over these two counts yields identical results over two global addition rounds, there cannot be any work left (not even in transit). Instead of the ring based summing scheme proposed in [22], we use a tree based asynchronous global reduction operation. This is a simple and portable way to bound the termination detection delay by $\mathcal{O}(T_{\text{rout}} \log P)$.

[2] If the machine supports it, explicit polling can be replaced by more efficient and more elegant interrupt mechanisms which (almost) only cost time when requests arrive.

```
var I, I' : Subproblem
I := if i_PE = 0 then I_root else I_∅
while no global termination yet do
  if T(I) = 0 then
    send a request to a random PE
    repeat
      receive any message M (blockingly)
      reply requests from PE j with I_∅
    until M is a reply to my request
    unpack I from M
  else I := work(I, Δt)
    if incoming request from PE j then (I, I') := split(I); send I' to PE j
```

**Fig. 1.** Basic algorithm for asynchronous random polling.

## 3 Expected Time Bounds

This Section is devoted to proving the following bound on the expected parallel execution time of asynchronous random polling dynamic load balancing:

**Theorem 1.** $\mathbf{E}T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{seq}}}{P} + \mathcal{O}\big(T_{\text{atomic}} + h\left(\frac{1}{\epsilon} + T_{\text{rout}} + T_{\text{split}}\right)\big)$ *for an appropriate choice of* $\Delta t$.

The basic idea for the proof is to partition the execution time of each individual PE into intervals of productive work on subproblems and intervals devoted to load balancing. We first tackle the more difficult part and show that a certain overall effort on load balancing suffices to split all remaining subproblems at least $h$ times. By definition of $h$ this implies that they are smaller than $T_{\text{atomic}}$. As a preparation, we assign a technical meaning to the terms "ancestor", "arrive" and "reach":

**Definition 1.** *The* ancestor *of a subproblem $I$ at time $t$ is the uniquely defined subproblem from which $I$ was derived by applying the operations "*work*" and "*split*". A load request* arrives *at the point of time $t$ when it is put into the message queue of a PE. A load request* reaches *a subproblem $I$ at time $t$ if it arrives at some PE at time $t$ and (later) leads to a splitting of $I$.*

We start the analysis by bounding the expense associated with sending and answering individual requests:

**Lemma 1.**

1. *The total amount of active CPU work expended for processing a request is bounded by* $T_{\text{split}} + \mathcal{O}(T_{\text{rout}})$.
2. *If any requests have arrived at a PE, at least one of the requests is answered every* $\Delta t + T_{\text{split}} + \mathcal{O}(T_{\text{rout}})$ *time units.*
3. *The expected elapsed time between the arrival of a message and sending the corresponding reply is in* $\mathcal{O}(\Delta t + T_{\text{split}} + T_{\text{rout}})$.

*Proof.* **1:** A request triggers at most one split. The total expense for sending and receiving is in $\mathcal{O}(T_{\text{rout}})$. **2:** An additional time of $\Delta t$ for sequential work can elapse until the message queue is checked the next time. **3:** Some queues might be long so that some request are delayed for a quite long time. However, there are at most $P$ active requests at any point in time. A request arriving at a random PE will therefore encounter an expected queue length bounded by $\sum_{i<P}$ "queue length at PE $i$"$/P \leq 1$. ∎

When a subproblem is split by one or more subsequent load request, there is a *dead time* interval during which it cannot be reached by any other request.

**Lemma 2.** *All dead times can be covered by associating a dead time* $T_{\text{dead}} = \Delta t + T_{\text{split}} + \mathcal{O}(T_{\text{rout}})$ *with each request reaching a subproblem.*

*Proof.* Let $I$ denote a subproblem that is reached by a request $R$ at time $t$ and at PE $i$. Let $k \geq 0$ denote the number of requests in the message queue of PE $i$ that reach $I$ before $R$. Only if $I$ is moved to another PE $j$ due to $R$, $I$ cannot be reached by any request arriving after $t$ until $I$ is put into the message queue of PE $j$. In the worst case, the dead time is $(k+1)(\Delta t + T_{\text{split}} + T_{\text{rout}})$. This is the case, when "work" has just been called for the ancestor of $I$. Then a time $\Delta t$ passes until the load balancer is next activated. Subsequently, the ancestor is split with an expense of $T_{\text{split}}$ and a subproblem is sent away. This cycle is repeated $k+1$ times. Then $I$ is reachable on PE $j$. The total dead time can be distributed over the $k+1$ requests involved. ∎

Now we know the various costs and delays associated with requests. If we could find out how many request are necessary to split all subproblems $h$ times with high probability, we were almost done. However, the question is stated too imprecisely yet. Requests that arrive during a dead time of a subproblem are "lost" for that subproblem. We therefore only consider a subset of all completed requests that has the property to be "sufficiently uniformly" distributed over time.

**Definition 2.** *A request may be colored* red *if there are at most $P$ other red requests during a time interval $T_{\text{dead}}$ after its arrival.*

**Lemma 3.** *Let $I\langle i\rangle$ denote the subproblem at PE $i$. For every $\beta > 0$ there is a constant $c > 0$, such that after processing $cPh$ red requests*

$$\mathbf{P}\left[\exists i : \text{gen}(I\langle i\rangle) < h\right] \leq P^{-\beta} \text{ (for sufficiently large } P\text{).}$$

*Proof.* For some fixed PE index $i$, we have

$$\mathbf{P}\left[\exists i : \text{gen}(I\langle i\rangle) < h\right] \leq P\mathbf{P}\left[\text{gen}(I\langle i\rangle) < h\right] \quad .$$

So it suffices to show that $\mathbf{P}\left[\text{gen}(I\langle i\rangle) < h\right] < P^{-\beta-1}$ for sufficiently large $P$. We can bound $\text{gen}(I\langle i\rangle)$ by the number of red requests that reach $I\langle i\rangle$. Uncolored requests can be ignored here w.l.o.g.: Although it may happen that an uncolored

request reaches $I\langle i\rangle$ and causes one or more subsequent red requests to miss $I\langle i\rangle$, this split will be accounted to the next following red request and its dead time suffices to explain that the subsequent red requests miss $I\langle i\rangle$. Using a combinatorial treatment, we now show that $\sum_{k<h} P_k \le P^{-\beta-1}$ where

$$P_k := \mathbf{P}\left[I\langle i\rangle \text{ is reached by } k \text{ red requests}\right] .$$

There are $\binom{chP}{k}$ ways, to choose $k$ red request that are to reach $I\langle i\rangle$. The probability that they are all heading for PE $i$ is $P^{-k}$. Since there are at most $P$ red requests in the dead time after a request, there are at least $chn - kP$ remaining red request that do not reach $I\langle i\rangle$. The probability of this event is

$$(1-1/P)^{chP-kP} \le e^{-(ch-k)} .$$

All in all, we have

$$P_k \le \binom{chP}{k} P^{-k} e^{-(ch-k)} \le \left(\frac{chPe}{k}\right)^k P^{-k} e^{-(ch-k)} = \left(\frac{che^2}{k}\right)^k e^{-ch}$$

using the Stirling approximation $\binom{m}{k} \le (me/k)^k$. Since $k < h$, it is easy to verify that the $k$-dependent part of the above expression is monotonously increasing with $k$ for $c > 1/e$ and can be bounded from above by setting $k = h$, i.e.,

$$P_k \le \left(ce^2\right)^h e^{-ch} = e^{-h(c-\ln c-2)} .$$

Now $\mathbf{P}\left[\text{gen}(I\langle i\rangle) < h\right]$ can be bounded by

$$he^{-h(c-\ln c-2)} = e^{-h\left(c-\ln c-2-\frac{\ln h}{h}\right)} \le e^{-h\left(c-\ln c-2-\frac{1}{e}\right)} .$$

Since we assume that $h \in \Omega(\log P)$ there is a $c'$ such that $h \ge c' \ln P$:

$$\mathbf{P}\left[\text{gen}(I\langle i\rangle) < h\right] \le P^{-c'\left(c-\ln c-2-\frac{1}{e}\right)} \le P^{-\beta-1}$$

for an appropriate $c$ and sufficiently large $P$. ∎

Now we bound the expense for all requests in order to have $cPh$ red ones among them.

**Lemma 4.** *Let $c > 0$ denote a constant. Requests can be colored in such a way that an expected work in $\mathcal{O}(hP(\Delta t + T_{\text{split}} + T_{\text{rout}}))$ for all request processing suffices to process $chP$ red requests.*

*Proof.* Let $R_1, \ldots, R_m$ denote all the requests processed and let $t(R_1) \le \cdots \le t(R_m))$ denote the arrival time of $R_i$. Going through this sequence of requests we color $P$ subsequent requests red and then skip the requests following in an interval of $T_{\text{dead}}$, etc. Since there can never be more than $P$ requests in transit there can be at most $2P$ uncolored requests whose executions overlaps an individual red interval. Therefore, the expense for $P$ red requests can be bounded by $PT_{\text{dead}}$ plus the expense for processing $3P$ requests. The expense for this is given in Lemma 1. ∎

By combining lemmata 3 and 4 we get a bound for the communication expense of random polling until only atomic subproblems are left.

**Lemma 5.** *The expected overall expense for communicating, splitting and waiting until there are no more subproblems with* $\text{gen}(I) < h$ *is in* $\mathcal{O}(hP(\Delta t + T_{\text{split}} + T_{\text{rout}}))$.

Bounding the expense for sequential work – i.e. calls of "work" – is easy. Let $T_{\text{poll}}$ denote the (constant) expense for probing the message queue unsuccessfully. It suffices to choose $\Delta t > T_{\text{poll}}/(\epsilon\alpha)$ to make sure that only $(1+\epsilon)T_{\text{seq}}$ time units are spent for those iterations of the main loop where the local subproblem is not exhausted and no requests arrive. All other loop iterations can be accounted to load balancing.

As the last component of our proof, we have to verify that atomic subproblems are disposed of quickly and that termination detection is no bottleneck.

**Lemma 6.** *If* $\Delta t \in \Omega\big(\min(\frac{T_{\text{atomic}}}{h}, T_{\text{rout}} + T_{\text{split}})\big)$ *and* $\text{gen}(I\langle i\rangle) \geq h$ *for all PEs then the remaining execution time is in*

$$\mathcal{O}(T_{\text{atomic}} + h(\Delta t + T_{\text{split}} + T_{\text{rout}})) \ .$$

*Proof.* From the definition of $h$ we can conclude that for all remaining subproblems $I$ we have $T(I) \leq T_{\text{atomic}}$. For $\frac{T_{\text{atomic}}}{h} \in \mathcal{O}(T_{\text{rout}} + T_{\text{split}})$, $\mathcal{O}(h)$ iterations (of each PE) with cost $\mathcal{O}(\Delta t + T_{\text{split}} + T_{\text{rout}})$ each suffice to finish up all subproblems. Otherwise, a busy PE spends at least a constant fraction of its time with productive work even if it constantly receives requests.[3] Therefore, after a time in $\mathcal{O}(T_{\text{atomic}})$ no nonempty subproblems will be left. After a time in $\mathcal{O}(T_{\text{rout}} \log P) \subseteq \mathcal{O}(hT_{\text{rout}})$, the termination detection protocol will notice this condition. ∎

The above building blocks can now be used to assemble a proof of Theorem 1. Choose some $\Delta t \in \mathcal{O}(T_{\text{rout}} + T_{\text{split}}) \cap \Omega\big(\min\big(\frac{T_{\text{atomic}}}{h}, T_{\text{rout}} + T_{\text{split}}\big)\big)$ such that $\Delta t > T_{\text{poll}}/(\epsilon\alpha)$ (where $T_{\text{poll}}$ is the constant time required to poll the network in the absence of messages). This is always possible and for the frequent case $T_{\text{atomic}}/h \ll T_{\text{rout}} + T_{\text{split}}$ there is also a very wide feasible interval for $\Delta t$. Every operation of Algorithm 1 is either devoted to working on a nonempty subproblem or to load balancing in the sense of Lemma 5. Therefore, after an expected time of $(1+\epsilon)\frac{T_{\text{seq}}}{P} + \mathcal{O}(h(1/\epsilon + T_{\text{rout}} + T_{\text{split}}))$ sufficiently many requests have been processed such that only subproblems with $\text{gen}(I) \geq h$ are left with high probability. The polynomially small fraction of cases where this number of requests is not sufficient cannot influence the expectation of the execution time since even a sequential solution of the problem instance takes only $\mathcal{O}(P)$ times as long as a parallel execution. According to Lemma 6, an additional time in $\mathcal{O}(T_{\text{atomic}} + h(1/\epsilon + T_{\text{split}} + T_{\text{rout}}))$ suffices to finish up the remaining subproblems and to detect termination. ∎

[3] In the full LogP model even $\Delta t \in \Omega\Big(\min(\frac{T_{\text{atomic}}}{h}, \max(T_{\text{split}} + o, g))\Big)$ suffices.

## 4 High Probability

In order to keep the algorithm and its analysis as simple as possible, Theorem 1 only bounds the expected parallel execution time. In [33] it is also shown how the same bounds can be obtained with high probability. The key observation is that Martingale tail bounds can be used to bound the sum of all queue lenghts encounterd by requests if the maximal queue length is not too large.

**Theorem 2.** *For $\Delta t$ and $\epsilon$ as in Theorem 1,*

$$T_{\text{par}} \leq (1+\epsilon)\frac{T_{\text{par}}}{P} + \tilde{\mathrm{O}}\left(T_{\text{atomic}} + h\left(\tfrac{1}{\epsilon} + T_{\text{split}} + T_{\text{rout}}\right)\right)$$

*if $h \in \Omega(P \log P)$ or queue lengths in $\Omega\left(\sqrt{P}\right)$ are avoided by algorithmic means.*[4]

## 5 Discussion

Tree shaped computations represent an extreme case for parallel computing in two respects. On the one hand, parallelism is very easy to expose since subproblems can be solved completely independently. Apart from that they are the worst case with respect to irregularity. Not only can splitting be arbitrarily uneven (only constrained by the maximum splitting depth $h$) but it is not even possible to estimate the size of a subproblem. Considering the simplicity of random polling and its almost optimal performance (both in theory and practice) the problem of load balancing tree shaped computations can largely be considered as solved.

Although tree shaped computations span a remarkably wide area of applications, an important area for future research is to generalize the analysis to models that cover dependencies between subproblems. The predictable dependencies modeled by multithreaded computations [2] are one step in this direction. But in many classic search problems the main difficulty are heuristics that prune the search tree in an unpredictable way.

## References

1. G. Aharoni, Amnon Barak, and Yaron Farber. An adaptive granularity control algorithm for the parallel execution of functional programs. *Future Generation Computing Systems*, 9:163–174, 1993.
2. N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

[4] Let $\tilde{\mathrm{O}}(\cdot)$ denote the following shorthand for asymptotic behavior with high probability [20]: A random variable $X$ is in $\tilde{\mathrm{O}}(g(P))$ iff $\forall \beta > 0 : \exists c > 0 : \exists P_0 : \forall P \geq P_0 : \mathbf{P}[X \leq cf(P)] \geq 1 - P^{-\beta}$

3. S. Arvindam, V. Kumar, V. N. Rao, and V. Singh. Automatic test pattern generator on parallel processors. Technical Report TR 90-20, University of Minnesota, 1990.
4. G. S. Bloom and S. W. Golomb. Applications of numbered undirected graphs. *Proceedings of the IEEE*, 65(4):562–570, April 1977.
5. R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foudations of Computer Science*, pages 356–368, Santa Fe, 1994.
6. M. Böhm and E. Speckenmeyer. A fast parallel SAT-solver – efficient workload balancing. *Annals of Mathematics and Artificial Intelligence*, 17:381–400, 1996.
7. S. Chakrabarti, A. Ranade, and K. Yelick. Randomized load balancing for tree-structured computation. In *Scalable High Performance Computing Conference*, pages 666–673, Knoxville, 1994.
8. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. v. Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, 1993.
9. W. Ertel. *Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen.* Dissertation, TU München, 1992.
10. P. Fatourou and P. Spirakis. Scheduling algorithms for strict multithreaded computations. In *ISAAC: 7th International Symposium on Algorithms and Computation*, number 1178 in LNCS, pages 407–416, 1996.
11. P. Fatourou and P. Spirakis. A new scheduling algorithm for general strict multithreaded computations. In *13rd International Symposium on DIStributed Computing (DISC'99)*, Bratislava, Slovakia, 1999. to appear.
12. R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *ACM Symposium on Parallel Architectures and Algorithms*, pages 94–103, 1994.
13. R. Finkel and U. Manber. DIB – A distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.
14. C. Goumopoulos, E. Housos, and O. Liljenzin. Parallel crew scheduling on workstation networks using PVM. In *EuroPVM-MPI*, number 1332 in LNCS, Cracow, Poland, 1997.
15. V. Heun and E. W. Mayr. Efficient dynamic embedding of arbitrary binary trees into hypercubes. In *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, number 1117 in LNCS, 1996.
16. J. C. Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.
17. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
18. V. Kumar and G. Y. Ananth. Scalable load balancing techniques for parallel computers. Technical Report TR 91-55, University of Minnesota, 1991.
19. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.
20. F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17:157–205, 1994.
21. S. Martello and P. Toth. *Knapsack Problems – Algorithms and Computer Implementations.* Wiley, 1990.
22. F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

23. M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *10th ACM Symposium on Parallel Algorithms and Architectures*, pages 212–221, 1998.
24. A. Nonnenmacher and D. A. Mlynski. Liquid crystal simulation using automatic differentiation and interval arithmetic. In G. Alefeld and A. Frommer, editors, *Scientific Computing and Validated Numerics*. Akademie Verlag, 1996.
25. W. H. Press, S.A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2. edition, 1992.
26. V. N. Rao and V. Kumar. Parallel depth first search. *International Journal of Parallel Programming*, 16(6):470–519, 1987.
27. A. Reinefeld. Scalability of massively parallel depth-first search. In *DIMACS Workshop*, 1994.
28. P. Sanders. Analysis of random polling dynamic load balancing. Technical Report IB 12/94, Universität Karlsruhe, Fakultät für Informatik, April 1994.
29. P. Sanders. A detailed analysis of random polling dynamic load balancing. In *International Symposium on Parallel Architectures, Algorithms and Networks*, pages 382–389, Kanazawa, Japan, 1994.
30. P. Sanders. Better algorithms for parallel backtracking. In *Workshop on Algorithms for Irregularly Structured Problems*, number 980 in LNCS, pages 333–347, 1995.
31. P. Sanders. A scalable parallel tree search library. In S. Ranka, editor, *2nd Workshop on Solving Irregular Problems on Distributed Memory Machines*, Honolulu, Hawaii, 1996.
32. P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. PhD thesis, University of Karlsruhe, 1997.
33. P. Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*. Number 463 in Fortschrittsberichte, Reihe 10. VDI Verlag, 1997.
34. P. Sanders. Tree shaped computations as a model for parallel applications. In *ALV'98 Workshop on application based load balancing*. SFB 342, TU München, Germany, March 1998. `http://www.mpi-sb.mpg.de/~sanders/papers/alv.ps.gz`.
35. E. Speckenmeyer, B. Monien, and O. Vornberger. Superlinear speedup for parallel backtracking. In C. D. Houstis, E. N.; Papatheodorou, T. S.; Polychronopoulos, editor, *Proceedings of the 1st International Conference on Supercomputing*, number 297 in LNCS, pages 985–993, Athens, Greece, June 1987. Springer.
36. R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. *Journal Parallel and Distributed Processing, Special Issue on Parallel and Distributed Data Structures*, 49:135–145, 1998.
37. I. C. Wu and H. T. Kung. Communication complexity of parallel divide-and-conquer. In *Foudations of Computer Science*, pages 151–162, 1991.

# Simple Approximation Algorithms for MAXNAESP and Hypergraph *2-colorability*

Daya Ram Gaur and Ramesh Krishnamurti

School of Computing Science,
Simon Fraser University, B.C, V5A 1S6.
{gaur,ramesh}@cs.sfu.ca

**Abstract.** Hypergraph *2-colorability*, also known as set splitting, is a widely studied problem in graph theory. In this paper we study the maximization version of the same. We recast the problem as a special type of satisfiability problem and give approximation algorithms for it. Our results are valid for hypergraph *2-colorability*, set splitting and MAX-CUT (which is a special case of hypergraph *2-colorability*) because the reductions are approximation preserving. Here we study the MAXNAESP problem, the optimal solution to which is a truth assignment of the literals that maximizes the number of clauses satisfied. As a main result of the paper, we show that any locally optimal solution (a solution is locally optimal if its value cannot be increased by complementing assignments to literals and pairs of literals) is guaranteed a performance ratio of $\frac{1}{2} + \epsilon$. This is an improvement over the ratio of $\frac{1}{2}$ attributed to another local improvement heuristic for MAX-CUT [6]. In fact we provide a bound of $\frac{k}{k+1}$ for this problem, where $\geq 3$ is the minimum number of literals in a clause. Such locally optimal algorithms appear to subsume typical greedy algorithms that have been suggested for problems in the general domain of satisfiability. It should be noted that the NAESP problem where each clause has exactly two literals, is equivalent to MAX-CUT. However, obtaining good approximation ratios using semi-definite programming techniques [3] appears difficult. Also, the randomized rounding algorithm as well as the simple randomized algorithm both [4] yield a bound of $\frac{1}{2}$ for the MAXNAESP problem. In contrast to this, the algorithm proposed in this paper obtains a bound of $\frac{1}{2} + \epsilon$ for this problem.

**Keywords:** *Approximation Algorithms, Hypergraph* 2-colorability*, Set Splitting, MAXNAESP, MAX-CUT.*

## 1 Introduction

Hypergraph *2-colorability*[1] is defined as follows:

*INSTANCE:* Given a collection $C$ of subsets of finite set $S$.

*QUESTION:* Is there a partition of $S$ into two subsets $S_1$ and $S_2$ such that every set in $C$ has elements from $S_1$ and $S_2$.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 49–55, 1999.
© Springer-Verlag Berlin Heidelberg 1999

The maximization version of this problem corresponds to finding two sets $S_1$ and $S_2$ such that the maximum number of sets in $C$ have elements from $S_1$ and $S_2$.

MAX-CUT is the problem of partitioning the nodes of an undirected graph $G = (V, E)$ into two sets $S$ and $V - S$ such that there are as many edges as possible between $S$ and $V - S$. Both hypergraph *2-colorability* and MAX-CUT are NP-Complete[6]. We recast the maximization version of hypergraph *2-colorability* as a maximization version of a special type of satisfiability problem called *MAXNAESP*.

*NAESP* is the not-all-equal satisfiability problem with only positive literals in the clauses where a clause is satisfied if it has at least one literal set to 0 and at least one literal set to 1. NAESP has a solution if every clause in the problem is satisfied. MAXNAESP is defined to be the problem of assigning boolean values to literals such that the maximum number of clauses are satisfied. MAXNAESP is a generalization of MAX-CUT which is evident from the following reduction. Given a graph $G = (V, E)$, for every edge $e = (u, v) \in E$ we have a clause $(u, v)$ in the corresponding MAXNAESP problem. It is easy to see that if there exists a cut of size $k$ then there exists a solution to the NAESP problem which satisfies at least $k$ clauses. Hence, we can regard MAXNAESP as a generalization of MAX-CUT and is equivalent to hypergraph *2-colorability*. The equivalence can be observed by associating each clause with an element (a subset of $S$) in set $C$ in the instance of hypergraph *2-colorability* and the assignment of boolean values to literals with the partition of $S$ into $S_1$ and $S_2$.

The technique of local improvement has been widely used as a heuristic for solving combinatorial optimization problems. A novel approximation algorithm for MAX-CUT is based on the idea of *local improvement* [6]. The idea is to start with some partition $S$ and $V - S$ and as long as we can improve the quality of the solution (the number of cross edges) by adding a single node to $S$ or by deleting a single node from $S$, we do so. It has been shown that this approximation algorithm [2] for MAX-CUT has a worst case performance ratio of $\frac{1}{2}$.

In this paper we describe another local improvement based approximation algorithm for solving MAXNAESP which yields a performance ratio of $\frac{k}{k+1}$ where each clause contains at least $k \geq 3$ variables. The algorithm described gives a bound of $\frac{1}{2} + \epsilon$ for MAX-CUT in contrast to the bound of $\frac{1}{2}$ achieved by the local search algorithm described in [2].

NP-Completeness of MAXNAESP follows from the fact that MAX-CUT is a special case of MAXNAESP. We give a polynomial reduction from 3NAES, in which each clause has exactly three literals, and the literals are not restricted to be positive [1], to NAESP thereby establishing the NP-Completeness of NAESP. In Section 2, we look at MAXNAESP and provide two approximation algorithms for it.

We begin with some definitions:

**Definition 1** NAES: *Given a set of clauses, find a satisfying truth assignment such that each clause contains at least one true literal and one false literal.*

**Definition 2** NAESP: *Given a set of clauses, where each clause contains only positive literals, find a satisfying truth assignment such that each clause contains at least one true literal and one false literal.*

**Theorem 1** *3NAES* $\alpha_p$ *3NAESP.*

*Proof:* Let $n$ be the number of clauses and $m$ be the number of literals in 3-NAES. Let us denote the literals in 3NAES by $x_1$, $\overline{x_1}$, $x_2$, $\overline{x_2}$, ... $x_m$, $\overline{x_m}$. We replace each $\overline{x_i}$ with a new literal $x_{m+i}$ in all the clauses. In addition, for each pair of literals $x_i$, $\overline{x_i}$, we add the following four clauses $(x_i, x_{m+i}, a), (x_i, x_{m+i}, b)$, $(x_i, x_{m+i}, c)$, $(a, b, c)$. We have a total of $(n + 4m)$ clauses in the instance of 3NAESP so generated.

$\Rightarrow$ If 3NAES is satisfiable then the set of clauses generated in the instance of 3NAESP generated is also satisfiable. The solution is obtained by assigning $x_{n+i}$ the same value as $\overline{x_i}$ in 3NAESP. In addition, the literal $a$ is set to 0 and the literals $b$,$c$ are set to 1.

$\Leftarrow$ If the clauses in the instance of 3NAESP generated is satisfiable then the solution to 3NAES is obtained by setting $\overline{x_i}$ to the same value that $x_{n+i}$ is set to. This assignment clearly satisfies all the clauses. For literals $x_i$, $\overline{x_i}$, the four clauses $(x_i, x_{m+i}, a)$, $(x_i, x_{m+i}, b)$, $(x_i, x_{m+i}, c)$, and $(a, b, c)$ guarantee that both $x_i$ and $\overline{x_i}$ are not set to 1 (or 0). □

## 2 Approximation Algorithms

In this section we study the optimization version of the NAESP problem called the MAXNAESP. The objective in MAXNAESP is to find a solution which maximizes the number of clauses satisfied. We first present a simple approximation algorithm for MAXNAESP which has a worst-case performance bound of $\frac{1}{2}$. We then examine a locally optimal approximation algorithm for the problem whose worst-case performance bound is $\frac{1}{2} + \epsilon$ for the problem, and in general has a bound of $\frac{k}{k+1}$, where $k \geq 3$ is the minimum number of literals in a clause.

$\frac{1}{2}$ **Approximation Algorithm:** Let the problem comprise $m$ literals and $n$ clauses. A literal $u$ is arbitrarily picked and set to 1 if it occurs in at least $\frac{n}{2}$ clauses. The remaining literals are set to 0 and the algorithm terminates. If however the literal $u$ occurs in only $k < \frac{n}{2}$ clauses, then the literal $u$ is set to 0 and the subproblem comprising the $n - k$ clauses not containing the literal $u$ is recursively solved (with the literal $u$ removed from the subproblem). If the solution $S$ to the subproblem satisfies more than $\frac{k}{2}$ clauses containing $u$ then the algorithm returns the solution $S$ for the remaining literals. Else the algorithm returns $\overline{S}$, the complement of the solution $S$ for the remaining literals (in the complement solution $\overline{S}$, all the literal settings are complemented).

**Theorem 2** *The above algorithm has a performance ratio of* $\frac{1}{2}$.

*Proof:* We prove by induction on the number of literals $l$ that the algorithm satisfies at least $\frac{n}{2}$ clauses.
*Base Case:* For $l = 2$ (the minimum number of literals required for NAESP) each clause contains both literals (else the clauses cannot be satisfied and can therefore be removed) and so the result follows trivially.
*Induction Hypothesis:* Assume the algorithm satisfies at least $\frac{1}{2}$ the total number of clauses for $l \leq p$ literals.
*Induction Step:* Let the number of clauses be $n$. Let the set of clauses satisfied by literal $u$ be denoted $U$. The algorithm either sets $u$ to 1 (if $u$ satisfies $k \geq \frac{n}{2}$ clauses), or sets $u$ to 0 and solves the subproblem with $p-1$ literals (and $n-k$ clauses) recursively. By the induction hypothesis, the algorithm derives a solution $S$ that satisfies at least $\frac{(n-k)}{2}$ clauses in the subproblem. If in addition $S$ also satisfies at least $\frac{k}{2}$ clauses among $U$, then the algorithm satisfies at least $\frac{n}{2}$ in total. If on the other hand $S$ satisfies less than $\frac{k}{2}$ clauses among $U$, then $\overline{S}$ satisfies at least $\frac{k}{2}$ clauses among $U$. This follows from the fact that the clauses that are unsatisfied in $U$ due to assignment $S$ have all their variables set to 0, implying that the assignment $\overline{S}$ satisfies all these clauses (because the variable $u$ in all these clauses is set to 0). In addition, both the assignment $S$ and its complement $\overline{S}$ continue to satisfy the same number of clauses in the subproblem. Thus the algorithm satisfies at least $\frac{n}{2}$ clauses in total. □

A trivial example where a literal satisfies exactly $\frac{1}{2}$ the total number of clauses suffices to show that the bound is tight.

In the next section we describe an approximation algorithm for MAXNAESP and show that the algorithm has a performance ratio of $\frac{k}{k+1}$, where $k$ is the minimum clause length.

### 2.1 $\frac{k}{k+1}$ Approximation Algorithm

This approximation algorithm relies on the notion of a local optimum. A solution $S$ to MAXNAESP, is *locally-1 optimal* if the number of clauses satisfied cannot be increased by complementing the value to which a literal is set in the solution $S$. A solution $S$ to MAXNAESP, is *locally-2 optimal* if the number of clauses satisfied cannot be increased by complementing the value of the literals in a satisfied clause of cardinality 2. A solution is said to be *locally optimal* if it is both *locally-1 optimal* and *locally-2 optimal*.

The algorithm starts with a random assignment of values to literals (say all literals are set to 0). It then complements the setting of a literal if this improves the solution. It then looks at all the satisfied clauses of cardinality 2 and checks if complementing both the literals increases the number of satisfied clauses. The algorithm continues in this manner until no further improvement results.

Let $S$ be the locally-optimal solution. With respect to this solution $S$, we divide the set of literals into two disjoint subsets, $X = \{x_1, x_2, \ldots, x_p\}$ comprising all literals which are set to 1, and $Y = \{y_1, y_2, \ldots, y_q\}$, comprising all literals which are set to 0. We divide the clauses of MAXNAESP into three sets, $A$,

$B$, and $C$. $A$ is the set of all satisfied clauses, $B$ ($C$) is the set of all unsatisfied clauses for which each literal in a clause belongs to $X$ ($Y$). We note that each clause in $A$ contains at least one literal that belongs to $X$, and one literal that belongs to $Y$ (else the clause will not be satisfied). Let $B(x_i), 1 \leq i \leq p$ ($C(y_j), 1 \leq j \leq q$) denote the number of clauses in $B$ ($C$) which contain the literal $x_i$ ($y_j$). Let $A(x)$ ($A(y)$) denote the number of clauses in $A$ that contain only one literal from the set $X$ ($Y$).

**Lemma 1** $A(x) \geq \sum_{i=1}^{p} B(x_i)$.

*Proof:* For the $B(x_i)$ clauses in $B$ containing the literal $x_i$, there should be $A(x_i) \geq B(x_i)$ clauses in $A$ which contain only literal $x_i$ from the set $X$ (and all other literals from the set $Y$). If this is not the case then by setting $x_i$ to 0 we can increase the number of clauses satisfied, violating the fact that $S$ is locally-optimal. Noting that a clause in $A$ which contains only literal $x_i$ is distinct from a clause in $A$ that contains only literal $x_j$, $i \neq j$, $1 \leq i, j \leq p$, it follows that the $A(x_i)$ clauses in $A$ containing only literal $x_i$ are distinct from the $A(x_j)$ clauses in $A$ containing only literal $x_j$. Thus, $A(x) = \sum_{i=1}^{p} A(x_i) \geq \sum_{i=1}^{p} B(x_i)$. □

The corollary below may be shown using similar reasoning as above.

**Corollary 1** $A(y) \geq \sum_{j=1}^{q} C(y_j)$.

Lemma 2 below derives an upper bound on $|B|$.

**Lemma 2** $|B| \leq \frac{\sum_{i=1}^{p} B(x_i)}{k}$.

*Proof:* Each clause $b_i \in B$ has $k_i \geq k$ literals in it. Counting the number of 1's occurring in the clauses in set $B$, we can write $\sum_{i=1}^{p} B(x_i) = \sum_{i=1}^{|B|} k_i$. The left hand side counts the occurrence of 1's in each literal, and the right hand side counts the occurrence of 1's in each clause. But $\sum_{i=1}^{|B|} k_i \geq \sum_{i=1}^{|B|} k = |B| \times k$, from which the result follows. □

By a similar reasoning, the corollary below follows.

**Corollary 2** $|C| \leq \frac{\sum_{j=1}^{q} C(y_j)}{k}$.

**Theorem 3** *Let $P$ be an algorithm which returns a locally-optimal solution to the MAXNAESP problem. The performance ratio of $P$ is given by $r_p \geq \frac{k}{k+1}$ for $k \geq 3$ for $k \geq 2$ the performance ratio is $\frac{1}{2} + \epsilon$.*

*Proof:*

($k \geq 3$) First we show that when the cardinality of each clause is $k \geq 3$ then the performance ratio of the algorithm is $\frac{k}{k+1}$. It should be noted that for this case we obtain the desired bound by considering only *locally-1 optimal* solutions. It follows from the definition that $|A| \geq A(x) + A(y)$,

since $A$ is the set of all clauses that are satisfied, and $A(x)$ $(A(y))$ is the set of clauses that are satisfied with exactly one literal from the set $X$ $(Y)$. From Lemmas 1 and 2, and Corollaries 1 and 2, it follows that $|A| \geq A(x) + A(y) \geq \sum_{i=1}^{p} B(x_i) + \sum_{j=1}^{q} C(y_j) \geq k \times (|B| + |C|)$. Hence the performance ratio $r_p$ is given by, $r_p = \frac{|A|}{(|A|+|B|+|C|)} \geq \frac{(|B|+|C|)k}{(|B|+|C|)k+|B|+|C|} \geq \frac{k}{k+1}$.

($k \geq 2$) When $k \geq 2$, the number of satisfied clauses $A$ is at most $\frac{A(x)+A(y)}{2}$ hence the previous case is not applicable. Here we use the fact that the solution under consideration is *locally-2 optimal*. Let $X(Y)$ be the set of variables which are set to 1(0) in the *locally optimal* solution. Without loss of generality we assume that $|X| \geq |Y|$. Since, the solution is locally-2 optimal, for every clause $(x_i, y_j)$ in the input

$$A(x_i) + A(y_j) \geq B(x_i) + C(y_j) + 2$$

else, interchanging the values of $x_i$ and $y_j$ would result in more clauses being satisfied. Also, $A(x_i) \geq B(x_i)$ and $A(y_j) \geq C(y_j)$ because the solution is locally-1 optimal. We can conclude that

$$\sum_{x_i \in X} A(x_i) + \sum_{y_j \in Y} A(y_j) > \sum_{x_i \in X} B(x_i) + \sum_{y_j \in Y} C(y_j) + 2$$

i.e., $(|B| + |C|) < \frac{2|A|-2}{2}$. Therefore the performance ratio is

$$\frac{|A|}{|A| + (|B| + |C|)} > \frac{|A|}{|A| + (|A| - 1)} = \frac{1}{2} + \epsilon$$

where $\epsilon = \frac{1}{4|A|}$. □

The following example illustrates that the above bound is tight when $k \geq 2$. Let the variables be $x_1, x_2, \ldots, x_m$ and $y_1, y_2, \ldots, y_m$. The first $m^2$ clauses are $(x_i, y_j)$ for $i = 1..m$ and $j = 1..m$. The next $\binom{m}{2}$ clauses are $(x_i, x_j)$ for $i = 1..m$ and $j = 1..m$ such that $i \neq j$. The last $\binom{m}{2}$ clauses are $(y_i, y_j)$ for $i = 1..m$ and $j = 1..m$ such that $i \neq j$. A *locally optimal* solution is all the $x's$ set to $1's$ and all the $y's$ set to $0's$. The ratio in this case is $\frac{m^2}{m^2+2*\binom{m}{2}} = \frac{1}{2} + \epsilon$.

## 3 Conclusion

As the main result in the paper, we propose a simple locally optimal approximation algorithm for the MAXNAESP problem whose performance ratio is $\frac{k}{k+1}$, where $k \geq 3$ is the minimum number of literals in a clause. For the case when $k \geq 2$ the performance ratio of the algorithm is $\frac{1}{2} + \epsilon$. Such an algorithm appears to be useful for a large class of problems in the general domain of satisfiability. Specifically, we note that this algorithm has a bound of $\frac{k}{k+1}$ for MAXSAT when

$k \geq 3$, and follows directly from Lemmas 1 and 2. This bound is identical to the bound derived by Johnson for a simple greedy algorithm [5].

We note that MAXNAESP is equivalent (in the sense that the approximation ratios are preserved) to the maximization version of hypergraph *2-colorability* and set splitting. Furthermore, the MAXNAESP problem where each clause has exactly 2 literals is equivalent to MAX-CUT. However, a simple adaptation of the semi-definite approximation algorithm for solving MAX-CUT [3] for MAXNAESP (where each clause has at least two literals) does not appear to work. In addition, a straight-forward extension of the randomized rounding algorithm (using the linear-programming solution as the probabilities) [4] yields a bound of $\frac{1}{2}$. Also, the simple randomized algorithm, where each literal is selected with probability $\frac{1}{2}$ has an expected bound of $1 - \frac{1}{2^{k-1}}$, where $k$ is the minimum clause length. In contrast to this, the algorithm proposed in this paper obtains a bound of $\frac{k}{k+1}$ for the MAXNAESP problem when $k \geq 3$. For the $k \geq 2$ the bound is $\frac{1}{2} + \epsilon$. Though the randomized algorithm has a better expected bound for $k \geq 4$, the algorithm proposed in this paper obtains a bound of $\frac{1}{2} + \epsilon$ (as compared to $\frac{1}{2}$ for the randomized algorithm), for the most general version of the MAXNAESP problem (and hyper graph *2-colorability*), obtained when $k = 2$.

**Acknowledgements:** The authors would like to thank an anonymous referee for pointing out an error in an earlier version of this paper.

## References

1. M. Garey and D. Johnson. *Computers and intractability: a guide to the theory of NP-Completeness*. W. H. Freeman, 1979.
2. M. R. Garey, D. S. Johnson, and L. J. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.
3. M. Goemans and D. P. Williamson. 0.878 approximation algorithms for MAX-CUT and max-2sat. In *Proceedings of the 26th. annual ACM symposium on theory of computing*, pages 422–431, 1994.
4. M. Goemans and D. P. Williamson. New 3/4 approximation algorithms for MAX-CUT. *SIAM J. Disc. Math.*, 7:656–666, 1994.
5. D. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and Systems Science*, 9:256–278, 1974.
6. C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

# Hardness of Approximating Independent Domination in Circle Graphs

Mirela Damian-Iordache[1] and Sriram V. Pemmaraju[2]

[1] Department of Computer Science, University of Iowa, Iowa City, IA 52242, U.S.A. damianjo@cs.uiowa.edu
[2] Department of Mathematics, Indian Institute of Technology, Bombay, Powai, Mumbai 400076, India. sriram@math.iitb.ernet.in

**Abstract.** A graph $G = (V, E)$ is called a *circle graph* if there is a one-to-one correspondence between vertices in $V$ and a set $C$ of chords in a circle such that two vertices in $V$ are adjacent if and only if the corresponding chords in $C$ intersect. A subset $V'$ of $V$ is a *dominating set* of $G$ if for all $u \in V$ either $u \in V'$ or $u$ has a neighbor in $V'$. In addition, if no two vertices in $V'$ are adjacent, then $V'$ is called an *independent dominating set*; if $G[V']$ is connected, then $V'$ is called a *connected dominating set*. Keil (*Discrete Applied Mathematics*, 42 (1993), 51-63) shows that the minimum dominating set problem and the minimum connected dominating set problem are both NP-complete even for circle graphs. He leaves open the complexity of the minimum independent dominating set problem. In this paper we show that the minimum independent dominating set problem on circle graphs is NP-complete. Furthermore we show that for any $\varepsilon$, $0 \leq \varepsilon < 1$, there does not exist an $n^{\varepsilon}$-approximation algorithm for the minimum independent dominating set problem on $n$-vertex circle graphs, unless P = NP. Several other related domination problems on circle graphs are also shown to be as hard to approximate.

## 1 Introduction

For a graph $G = (V, E)$, a subset $V'$ of $V$ is a *dominating set* of $G$ if for all $u \in V$ either $u \in V'$ or $u$ has a neighbor in $V'$. In addition, if no two vertices in $V'$ are adjacent, then $V'$ is called an *independent dominating set*; if the subgraph of $G$ induced by $V'$, denoted $G[V']$, is connected, then $V'$ is called a *connected dominating set*; if $G[V']$ has no isolated nodes, then $V'$ is called a *total dominating set*; and if $G[V']$ is a clique, then $V'$ is called a *dominating clique*. Garey and Johnson [18] mention that problems of finding a minimum cardinality dominating set (MDS), minimum cardinality independent dominating set (MIDS), minimum cardinality connected dominating set (MCDS), minimum cardinality total dominating set (MTDS), and minimum cardinality dominating clique (MDC) are all NP-complete for general graphs.

Restrictions of these problems to various classes of graphs have been studied extensively [11]. Table 1 shows the computational complexity of three of the problems mentioned above when restricted to different classes of graphs. P indicates

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 56–69, 1999.
© Springer-Verlag Berlin Heidelberg 1999

the existence of a polynomial time algorithm and NPc indicates that the problem is NP-complete. Some of the references mentioned in the table are to original papers where the corresponding result first appeared, while some are to secondary sources. It should be noted that this list is far from being comprehensive and that these problems have been studied for several other classes of graphs. In this paper we focus on the only "question mark" in the above table, corresponding to MIDS on circle graphs. Not only do we show that MIDS is NP-complete for circle graphs, we also show that it is extremely hard to approximate.

| Class of graphs | MDS | MIDS | MCDS |
|---|---|---|---|
| trees | P [18] | P [2] | P [11] |
| cographs | P [10] | P [10] | P [6] |
| interval graphs | P[4] | P [9] | P [17] |
| permutation graphs | P [10] | P [10] | P [5] |
| cocomparability graphs | P [11] | P [15] | P [15] |
| bipartite graphs | NPc [8] | NPc [6] | NPc [19] |
| comparability graphs | NPc [8] | NPc [6] | NPc [19] |
| chordal graphs | NPc [3] | P [9] | NPc [16] |
| circle graphs | NPc [14] | ? | NPc [14] |

**Fig. 1.** Complexity of three domination problems when restricted to different classes of graphs.

A graph $G = (V, E)$ is called a *circle graph* if there is a one-to-one correspondence between vertices in $V$ and a set $C$ of chords in a circle such that two vertices in $V$ are adjacent if and only if the corresponding chords in $C$ intersect. $C$ is called the *chord intersection model* for $G$. Equivalently, the vertices of a circle graph can be placed in one-to-one correspondence with the elements of a set $I$ of intervals such that two vertices are adjacent if and only if the corresponding intervals overlap, but neither contains the other. $I$ is called the *interval model* of the corresponding circle graph. Representations of a circle graph as a graph or as a set of chords or as a set of intervals are equivalent via polynomial time transformations. So, without loss of generality, in specifying instances of problems, we assume the availability of the representation that is most convenient.

The complexity of MDS on circle graphs was first mentioned as being unknown in Johnson's NP-completeness column [13]. Little progress was made towards solving this problem until Keil [14] resolved the complexity of MDS on circle graphs by showing that it is NP-complete. In the same paper, Keil showed that for circle graphs MCDS is NP-complete, MTDS is NP-complete, and MDC has a polynomial algorithm. He left the status of MIDS for circle graphs open. In this paper we show that MIDS is also NP-complete on circle graphs. We also attack the approximability of MIDS on circle graphs and show that this problem is extremely hard to approximate. An $\alpha$-*approximation* algorithm for a minimization problem is a polynomial time algorithm that guarantees that the

ratio of the cost of the solution to the optimal (over all instances of the problem) does not exceed $\alpha$. In Section 2 we show that for any $\varepsilon$, $0 \leq \varepsilon < 1$, there does not exist an $n^{\varepsilon}$-approximation algorithm for MIDS on an $n$-vertex circle graph, unless P = NP. In Section 3 we present hardness of approximation results for related problems — for example, we show that MIDS on bipartite graphs has no $n^{\varepsilon}$-approximation algorithm unless P = NP. This implies that MIDS, even when restricted to classes of graphs such as circle graphs and bipartite graphs, is as hard to approximate as the hardest problems — maximum clique and chromatic number, for example [1].

## 2 Intractability of MIDS on Circle Graphs

A first result in this section shows that the problem of finding a minimum independent dominating set on circle graphs is NP-complete. This was one of the open questions in [14]. In a second result we strengthen the NP-completeness proof to show that for any $\varepsilon$, $0 \leq \varepsilon < 1$, there does not exist an $n^{\varepsilon}$-approximation algorithm for MIDS on an $n$-vertex circle graph, unless P = NP. The decision problem formulation of the minimum independent dominating set problem on circle graphs (CMIDS) is as follows.

**Minimum Independent Dominating Set for Circle Graphs** (CMIDS)
INPUT: An interval representation of a circle graph $G = (V, E)$ and a positive integer $k$.
QUESTION: Is there an independent dominating set of $G$ of size at most $k$?

**Theorem 1.** *CMIDS is NP-complete.*

**Proof:** Clearly CMIDS is in NP. The proof that the problem is NP-hard is organized in two parts. In the first part we present a polynomial time reduction from 3SAT to CMIDS. Let $F$ be an arbitrary instance of 3SAT. Let $X = \{x_1, x_2, \cdots, x_q\}$ be the set of boolean variables and let $C = \{C_1, C_2, \cdots, C_m\}$ be the set of clauses in $F$. Let $I$ be the instance of CMIDS produced by the reduction from $F$ and let $G(I)$ be the overlap graph of $I$. In the second part we choose an integer $k$ and show that $G(I)$ has an independent dominating set of size $k$ or less if and only if $F$ is satisfiable.

In the following we describe the reduction from 3SAT to CMIDS. Without loss of generality we assume that no two literals involving the same variable appear more than once in a clause. We consider first each clause $C_j$ separately and create six intervals with types $a$, $b$, $f$, $t$, $p$ and $s$ associated with each literal that appears in $C_j$. Including the type $t$ interval in the independent dominating set will correspond to setting the associated literal to true and including the type $f$ interval in the dominating set will correspond to setting the associated literal to false.

For each clause $C_j$ we create three pairs of intervals of types $u$ and $w$. The purpose of each such pair is to dominate the type $a$ and type $b$ intervals associated with two out of the three literals that appear in $C_j$. The key idea of our proof

is as follows. Suppose that $C_j$ is satisfied by some assignment of truth values to variables. Then we can construct an independent dominating set that contains a type $t$ interval corresponding to a true literal in $C_j$ and a pair of type $u$, type $w$ intervals associated with $C_j$. The type $t$ interval dominates one pair of type $a$, type $b$ intervals while the other two pairs of type $a$, type $b$ intervals are dominated by the type $u$, type $w$ pair. In this manner all the type $a$ and type $b$ intervals associated with $C_j$ are dominated. Including the type $t$ interval in our independent dominating set will correspond to the clause $C_j$ being satisfied by a literal associated to this interval.

Finally we connect clauses together by $tt$, $ff$, $tf$ and $ft$ type intervals so as to maintain consistency of truth values of literals throughout all clauses. Figure 2 shows all intervals associated with a two clause and four variable formula in which $C_1 = \{x_1, x_2, \overline{x}_3\}$ and $C_2 = \{\overline{x}_1, x_2, \overline{x}_4\}$.

In the following we present our construction in detail. Let us consider an arbitrary clause $C_j$. For each variable $x_i$ that appears in $C_j$ we construct two independent intervals $a_j^i = [y_j + 12i, z_j + 5i]$ and $b_j^i = [y_j + 12i + 1, z_j + 5i - 1]$, where $y_j = 12(q+2)j$ and $z_j = y_j + 12(q+1)$. The purpose of the offset $y_j$ is to ensure that the intervals associated with different clauses are disjoint. Note that since $b_j^i \subset a_j^i$, these two intervals do not overlap. Next we construct truth setting intervals $t_j^i = [y_j + 12i - 1, y_j + 12i + 5]$ and $f_j^i = [y_j + 12i + 4, y_j + 12i + 10]$ associated with the variable $x_i$. The intervals $t_j^i$ and $f_j^i$ dominate each other, so at most one of them can appear in any independent dominating set. Furthermore $t_j^i$ dominates both $a_j^i$ and $b_j^i$ and $f_j^i$ does not dominate either. The truth setting intervals will help us in determining the correspondence between a satisfying truth assignment and an independent dominating set of size $k$ or less.

To ensure that exactly one of the intervals $t_j^i$ and $f_j^i$ associated with a literal involving $x_i$ appears in any "small enough" independent dominating set, we associate with $x_i$ two other independent intervals $p_j^i = [y_j + 12i + 3, y_j + 12i + 6]$ and $s_j^i = [y_j + 12i + 2, y_j + 12i + 7]$ adjacent to both $t_j^i$ and $f_j^i$ and to no other interval in $I$. Later we will argue that any "small enough" independent dominating set must contain either of the intervals $t_j^i$ or $f_j^i$ in order to dominate $p_j^i$ and $s_j^i$. Otherwise we would need to include both $p_j^i$ and $s_j^i$ in our independent dominating set and the set would become too large.

We now construct intervals associated to the clause $C_j$. The clause $C_j$ is satisfied if and only if at least one of the three literals involved in $C_j$ is true. This will correspond to at least one of the intervals $t_j^i$ appearing in the independent dominating set. Such an interval will dominate the intervals $a_j^i$ and $b_j^i$ associated to the literal involving $x_i$. In order to dominate the other four $a_j$ and $b_j$ type intervals associated to the other two literals in $C_j$ we add six more intervals of type $u$ and $w$ to $I$ as follows. Suppose that $C_j$ consists of three literals corresponding to three variables $x_i$, $x_k$ and $x_l$, with $i < k < l$. We create the intervals $u_j^{il} = [z_j, z_j + 5i + 3]$, $w_j^{il} = [z_j + 5k + 1, z_j + 5l + 3]$, $u_j^{ik} = [z_j + 1, z_j + 5k + 3]$, $w_j^{ik} = [z_j + 2, z_j + 5k + 2]$, $u_j^{kl} = [z_j + 5i + 1, z_j + 5l + 2]$ and $w_j^{kl} = [z_j + 5i + 2, z_j + 5l + 1]$, where $z_j$ is as defined above. Note that the

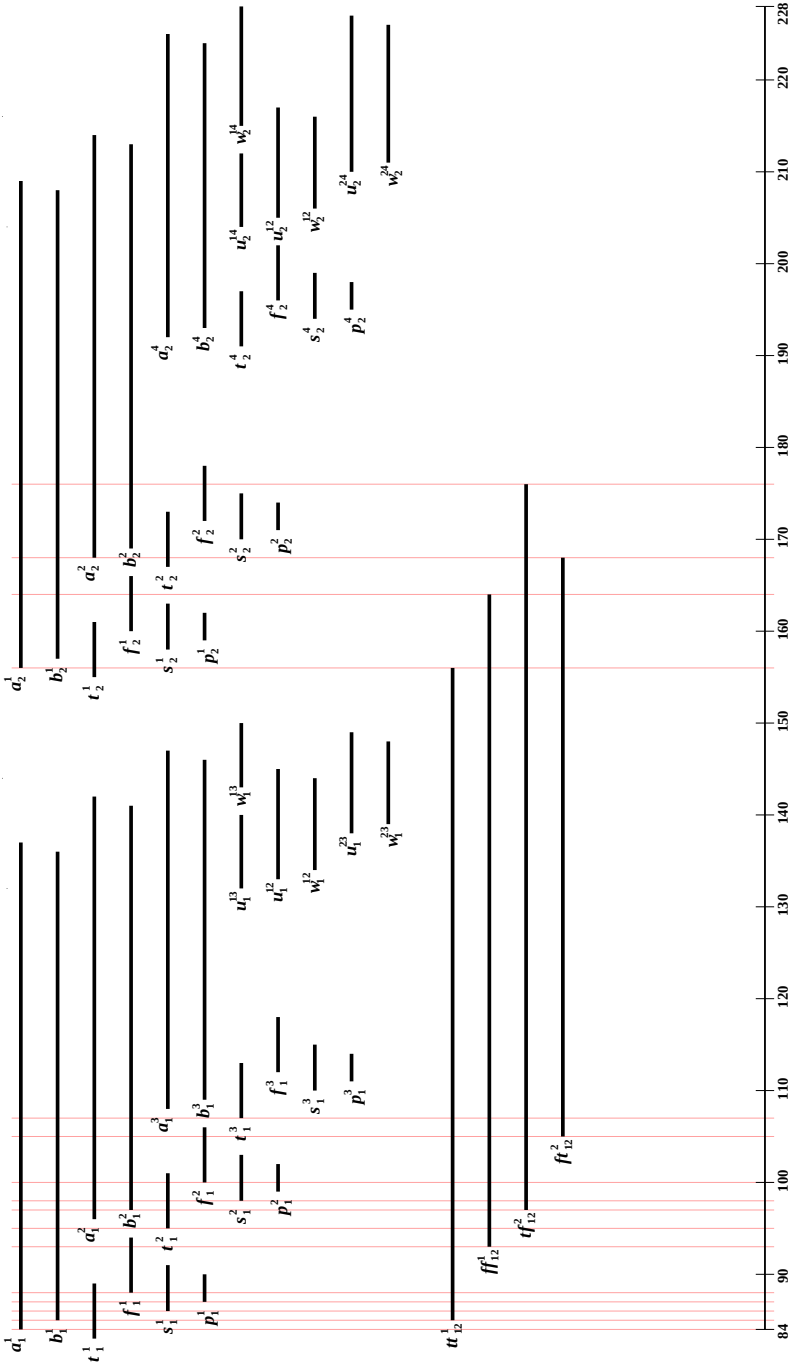



**Fig. 2.** The intervals constructed from the two clause and four variable formula $(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_4)$

two intervals $u_j^{il}$ and $w_j^{il}$ are independent and dominate the remaining 4 intervals of type $u_j$ and type $w_j$. Furthermore, $u_j^{il}$ and $w_j^{il}$ dominate the following 4 intervals of type $a_j$ and $b_j$: $a_j^i$, $b_j^i$, $a_j^l$, and $b_j^l$. The interval pair $u_j^{ik}$ and $w_j^{ik}$ and the interval pair $u_j^{kl}$ and $w_j^{kl}$ have similar properties.

If we repeat the construction described above for each clause $C_j$, $1 \leq j \leq m$, we obtain $m$ pairwise disjoint subsets of intervals, each subset associated to one clause. A last piece of our construction is a gadget that forces a variable to have the same truth value throughout all clauses. For this we create intervals of type $tt$, $ff$, $tf$ and $ft$ that connect clauses together as follows. For any $j$ and $k$, $1 \leq j < k \leq m$, if $x_i$ appears in two clauses $C_j$ and $C_k$ or if $\overline{x}_i$ appears in both $C_j$ and $C_k$ we create two intervals $tf_{jk}^i$ and $ft_{jk}^i$ such that $tf_{jk}^i$ overlaps $t_j^i$ and $f_k^i$ and $ft_{jk}^i$ overlaps $f_j^i$ and $t_k^i$. Furthermore, if $x_i$ appears in $C_j$ and $\overline{x}_i$ appears in $C_k$ or if $\overline{x}_i$ appears in $C_j$ and $x_i$ appears in $C_k$, we create two intervals $tt_{jk}^i$ and $ff_{jk}^i$ such that $tt_{jk}^i$ overlaps $t_j^i$ and $t_k^i$ and $ff_{jk}^i$ overlaps $f_j^i$ and $f_k^i$. We now specify the intervals of type $tt$, $ff$, $tf$, and $ft$. If $x_i$ appears in both $C_j$ and $C_k$ or $\overline{x}_i$ appears in both $C_j$ and $C_k$, then we add $tf_{jk}^i = [y_j + 12i + 1, y_k + 12i + 8]$, $ft_{jk}^i = [y_j + 12i + 9, y_k + 12i]$ to the set $I$. If $x_i$ appears in one of $C_j$ or $C_k$ and $\overline{x}_i$ appears in the other, then we add $tt_{jk}^i = [y_j + 12i + 1, y_k + 12i]$, $ff_{jk}^i = [y_j + 12i + 9, y_k + 12i + 8]$ to the set $I$.

This completes the construction of $I$. We now set $k = 5m$ and prove the following lemma, which completes the proof of the theorem.

**Lemma 1.** *$G$ has an independent dominating set of size $k$ if and only if $F$ is satisfiable.*

*Proof.* Given a satisfying truth assignment $A$ for $F$ we show how to construct a set $D$ of size $k$ of independent intervals that dominate all intervals in $I - D$. If $x_i$ is true in $A$ we include in $D$, $t_j^i$ for each clause $C_j$ in which $x_i$ appears and $f_j^i$ for each clause $C_j$ in which $\overline{x}_i$ appears. If $x_i$ is false in $A$ we include in $D$, $f_j^i$ for each clause $C_j$ in which $x_i$ appears and $t_j^i$ for each clause in which $\overline{x}_i$ appears. This ensures that all the $p^i$ and the $s^i$ type intervals have been dominated. Note that so far $3m$ intervals have been included in $D$.

We now show that all the $tf^i$, $ft^i$, $tt^i$ and $ff^i$ type intervals are also dominated by the intervals included in $D$ so far. Consider an interval $tf_{jk}^i$. The presence of this interval in $I$ implies one of four possibilities: (a) $x_i$ is true and $x_i$ appears in both $C_j$ and $C_k$, (b) $x_i$ is true and $\overline{x}_i$ appears in both $C_j$ and $C_k$, (c) $x_i$ is false and $x_i$ appears in both $C_j$ and $C_k$, and (d) $x_i$ is false and $\overline{x}_i$ appears in both $C_j$ and $C_k$. If $x_i$ is true and it appears in both $C_j$ and $C_k$, then $t_j^i$ dominates $tf_{jk}^i$. Otherwise if $x_i$ is true and $\overline{x}_i$ appears in both $C_j$ and $C_k$, then $f_k^i$ dominates $tf_{jk}^i$. If $x_i$ is false and it appears in both $C_j$ and $C_k$, then $f_k^i$ dominates $tf_{jk}^i$. Otherwise if $x_i$ is false and $\overline{x}_i$ appears in both $C_j$ and $C_k$, then $t_j^i$ dominates $tf_{jk}^i$. So $tf_{jk}^i$ is dominated by one of the intervals already included in $D$ in each of the four possible cases. In a similar manner it can be shown that the intervals of type $ft^i$, $tt^i$, and $ff^i$ are all dominated by the intervals already included in $D$.

Let us consider a clause $C_j$ that contains three literals corresponding to variables $x_i$, $x_k$ and $x_l$, where $i < k < l$. Since $A$ is a satisfying truth assignment, at least one of the literals involving $x_i$, $x_k$ and $x_l$ must be true. Let us assume without loss of generality that the literal involving $x_i$ is true and so we have included $t_j^i$ in $D$. In this case we add $u_j^{lk}$ and $w_j^{lk}$ to $D$, thus ensuring that all the $a_j$, $b_j$, $u_j$ and $w_j$ type intervals have been dominated. So we have added two more intervals associated with the clause $C_j$ to $D$. Repeating this for each clause will add $2m$ intervals to the independent set and will also ensure that all intervals in $G$ are dominated. Thus we have an independent dominating set of size $5m$ as required.

We now establish the other direction of Lemma 1: if there is a dominating independent set $D \subseteq I$ of size $k$, then there is a truth assignment to the variables in $X$ that satisfies $F$. Recall that associated with each variable $x_i$ that appears in a clause $C_j$ there is a pair of intervals: $s_j^i$ and $p_j^i$. Since no interval in $I$ is adjacent to two $p$ type intervals or two $s$ type intervals and since only $t_j^i$ and $f_j^i$ are adjacent to $p_j^i$ and $s_j^i$, $D$ must contain, for each $i$ and $j$, either $t_j^i$ or $f_j^i$ or both $s_j^i$ and $p_j^i$. If for each $i$ and $j$, $D$ contains $t_j^i$ or $f_j^i$ (but not both, since $t_j^i$ and $f_j^i$ are adjacent to each other), then exactly these $3m$ type $t$ and type $f$ intervals in $D$ suffice to dominate all the type $p$ and type $s$ intervals. This leaves exactly $2m$ intervals to dominate the rest of intervals in $I$. If for some $i$ and $j$, $D$ contains $p_j^i$ and $s_j^i$ then more than $3m$ intervals in $D$ are necessary to dominate the type $p$ and type $s$ intervals. This leaves fewer than $2m$ intervals to dominate the rest of the intervals in $I$.

There is no interval in $I$ adjacent to $u$ type intervals or $w$ type intervals associated with different clauses. In other words, for any $j \neq k$, no interval in $I$ is adjacent to two intervals of types $u_j$ and $w_k$, or $u_j$ and $u_k$, or $w_j$ and $w_k$. Thus the set of intervals in $D$ which dominate the $u$ and $w$ type intervals associated with different clauses are pairwise disjoint. Also no interval in $I$ is adjacent to all the six $u_j$ and $w_j$ type intervals associated with a clause $C_j$. Therefore $D$ must contain at least two intervals per clause in order to dominate all the $u$ and $w$ type intervals. Combined with the argument in the previous paragraph, this means exactly $3m$ intervals in $D$ dominate the type $p$ and type $s$ intervals and the remaining $2m$ intervals dominate the type $u$ and type $w$ intervals. This further implies that for each variable $x_i$ appearing in a clause $C_j$, either $t_j^i$ or $f_j^i$ (but, not both) belong to $I$.

We first show that for each clause $C_j$, $D$ contains at least one type $t_j$ interval, one type $u_j$ interval, and one type $w_j$ interval. Suppose that the three variables that appear in $C_j$ are $x_i$, $x_k$, and $x_l$, where $i < k < l$. As shown before, $D$ must contain two intervals which dominate all the $u_j$ and $w_j$ type intervals. The only intervals adjacent to a $u_j$ or a $w_j$ type interval are the $a_j$, $b_j$, $u_j$ and $w_j$ type intervals. No two $a_j$ type intervals or two $b_j$ type intervals can occur in $D$, since they are not independent. Suppose that $D$ contains an interval $a_j^i$. This interval is adjacent to at most four out of the six intervals of type $u_j$ or $v_j$. This leaves at least one type $u_j$, at least one type $w_j$ interval, and $b_j^i$ undominated. It is easy to see that independent of which type $t_j$ and which type $f_j$ intervals

are included in $D$, it is not possible for one interval to dominate all of these undominated intervals. Thus we have shown that $D$ cannot contain a type $a_j$ interval ( since the above argument can be used to show that $D$ cannot contain $a_j^k$ or $a_j^l$ as well). Likewise it can be shown that $D$ does not contain any of the $b_j$ type intervals. Hence $D$ must contain two independent intervals of type $u_j$ or $w_j$ in order to dominate all the $u_j$ and $w_j$ type intervals. ¿From the structure of $I$ it can be seen that such two intervals must be of the form $u_j^{ik}$ and $w_j^{ik}$. Besides dominating all the $u_j$ and $w_j$ type intervals, any pair $(u_j^{ik}, w_j^{ik})$ of intervals also dominates $a_j^i$, $b_j^i$, $a_j^k$ and $b_j^k$. The other two intervals of type $a_j$ and $b_j$ associated with the clause $C_j$ must be dominated by a type $t$ interval.

The type $t$ and type $f$ intervals have thus accounted for $3m$ of the intervals in $D$ and the two type $u$ and type $w$ intervals per clause account for the remaining $2m$ intervals in $D$. Since all $5m$ intervals in $D$ have been accounted for, we have to show that all intervals in $I - D$ are dominated. The above argument has established that all the type $p$, type $s$, type $t$, type $f$, type $a$, type $b$, type $u$, and type $w$ intervals have been dominated. This only leaves the type $tt$, type $tf$, type $ff$, and type $ft$ intervals. In the following we observe that these intervals are dominated if the truth values included in $D$ are "consistent" across variables and clauses.

(a) if $x_i$ (or $\overline{x}_i$) appears in two different clauses $C_j$ and $C_k$ and $D$ contains $t_j^i$ (respectively, $f_j^i$), then $D$ must also contain $t_k^i$ (respectively, $f_k^i$) in order to dominate $ft_{jk}^i$ (respectively, $tf_{jk}^i$).
(b) if $x_i$ appears in $C_j$, $\overline{x}_i$ appears in $C_k$ and $D$ contains $t_j^i$ (respectively, $f_j^i$), then $D$ must also contain $f_k^i$ (respectively, $t_k^i$) in order to dominate $ff_{jk}^i$ (respectively, $tt_{jk}^i$).

Now we can construct a satisfying truth assignment $A$ for $F$ as follows. If $D$ contains a type $t^i$ interval associated to a literal involving $x_i$, then we set this literal to true in $A$. Similarly, if $D$ contains a type $f^i$ interval associated with a literal involving $x_i$, then we set this literal to false in $A$. As mentioned above, $D$ must contain those type $t$ and type $f$ intervals that correspond to a literal having the same truth value throughout all clauses. Since for any clause $C_j$ there exists $i$ such that $t_j^i$ occurs in $D$, we conclude that any clause $C_j$ contains a true literal and therefore $A$ satisfies $F$ as required.

In the next result, we strengthen the above NP-completeness proof to show that the CMIDS problem is extremely hard even to approximate.

**Theorem 2.** *For any $\varepsilon$, $0 \leq \varepsilon < 1$, CMIDS does not have an $n^{\varepsilon}$-approximation algorithm, unless $P = NP$. Here $n$ is the number of intervals in the input to CMIDS.*

*Proof.* The proof of the theorem is organized in three parts. In the first part, we present a reduction from 3SAT to CMIDS similar to the one described in Theorem 1. Let $F$ be an arbitrary instance of 3SAT with $X = \{x_1, x_2, \cdots, x_q\}$

as the set of boolean variables and $C = \{C_1, C_2, \cdots, C_m\}$ as the set of clauses. Let $J$ be the instance of CMIDS produced by the reduction from $F$ and let $G(J)$ be the overlap graph of $J$. The reduction is parameterized by a positive integer $\alpha$ and the size of $J$ depends on the size of $F$ and on $\alpha$.

In the second part we show that the reduction has the following two properties. (i) if $F$ is satisfiable, then a minimum independent dominating set in $G(J)$ has size $5m$ or less and (ii) if $F$ is not satisfiable, then a minimum independent dominating set in $G(J)$ has size greater than $5\alpha m$

In the third part, we show that for any $\varepsilon$, $0 \leq \varepsilon < 1$, if $\alpha \leq |J|^{\varepsilon}$, then $|J|$ is a polynomial function of $m$. This ensures that the reduction described in the first part of the proof takes polynomial time.

These three pieces together prove the theorem. To see this suppose that for some $\varepsilon$, $0 \leq \varepsilon < 1$, we have an $n^{\varepsilon}$-approximation algorithm $\mathcal{A}$ for CMIDS. Then we could use $\mathcal{A}$ to solve an arbitrary instance $F$ of 3SAT in polynomial time as follows. Part 3 of the proof implies that given $F$, we construct $J$ in polynomial time. We then apply $\mathcal{A}$ to $G(J)$. If $F$ is satisfiable, then the size of a minimum independent dominating set in $G(J)$ is $5m$ or less and therefore the size of the independent dominating set found by $\mathcal{A}$ is $5n^{\varepsilon}m$ or less. If $F$ is not satisfiable, then the size of a minimum independent dominating set in $G(J)$ is greater than $5n^{\varepsilon}m$ and therefore the size of the independent dominating set found by $\mathcal{A}$ is greater than $5n^{\varepsilon}m$. Hence, comparing the size of the set found by $\mathcal{A}$ with the value $5n^{\varepsilon}m$ resolves the satisfiability of $F$ in polynomial time.

In the following we describe the reduction from 3SAT to CMIDS. The set of intervals $J$ is a simple extension of the set $I$ used in the proof of Theorem 1 and we only sketch the modifications here. The reduction is depicted in Figure 3. For an interval $a$, let $l(a)$ denote the left endpoint of $a$ and $r(a)$ denote the right endpoint of $a$. Consider an arbitrary clause $C_j$. For each variable $x_i$ that appears in $C_j$ the set $I$ contains two independent intervals $a_j^i$ and $b_j^i$. Recall that these two intervals overlap exactly the same set of intervals. Corresponding to these two intervals in $I$ we add $r = 5\alpha m$ independent intervals, named $a_j^{1i}$, $a_j^{2i}$ $\cdots$ $a_j^{ri}$ to $J$ such that (i) $a_j^{1i} = a_j^i$, $a_j^{ri} = b_j^i$ and (ii) $a_j^{l+1i} \subset a_j^{li}$ for $l = 1 \cdots r-1$. This ensures that the intervals $a_j^{li}$ for all $l$ overlap exactly the same set of intervals. Let us call $a_j^{1i}$ and $a_j^{ri}$ *originals* and $a_j^{\ell i}$ for all $\ell$, $1 < \ell < r$ *copies*. For each variable $x_i$ that appears in $C_j$, $I$ also contains four more intervals $t_j^i$, $f_j^i$, $s_j^i$ and $p_j^i$. We include in $J$ the truth setting intervals $t_j^i$ and $f_j^i$ as they appear in the set $I$. Corresponding to $s_j^i$ and $p_j^i$ in $I$ we add to $J$, $r$ independent intervals $s_j^{1i}$, $s_j^{2i}$ $\cdots$ $s_j^{ri}$ such that (i) $s_j^{1i} = s_j^i$, $s_j^{ri} = p_j^i$ and (ii) $s_j^{l+1i} \subset s_j^{li}$ for $l = 1 \cdots r-1$. This ensures that the intervals $s_j^{li}$ for all $l$, overlap exactly the same set of intervals. As before, let us call $s_j^{1i}$ and $s_j^{ri}$ *originals* and $s_j^{\ell i}$ for all $\ell$, $1 < \ell < r$ *copies*. The set $I$ also contains intervals of type $u_j$ and $w_j$ associated with each clause $C_j$. We add these to $J$ as they appear in $I$.

Finally, recall that in the construction used in Theorem 1, clauses that contain common variables are connected together by $tt$, $ff$, $tf$ and $ft$ type intervals in $I$. Corresponding to each of these intervals in $I$ we add $r$ new intervals to $J$

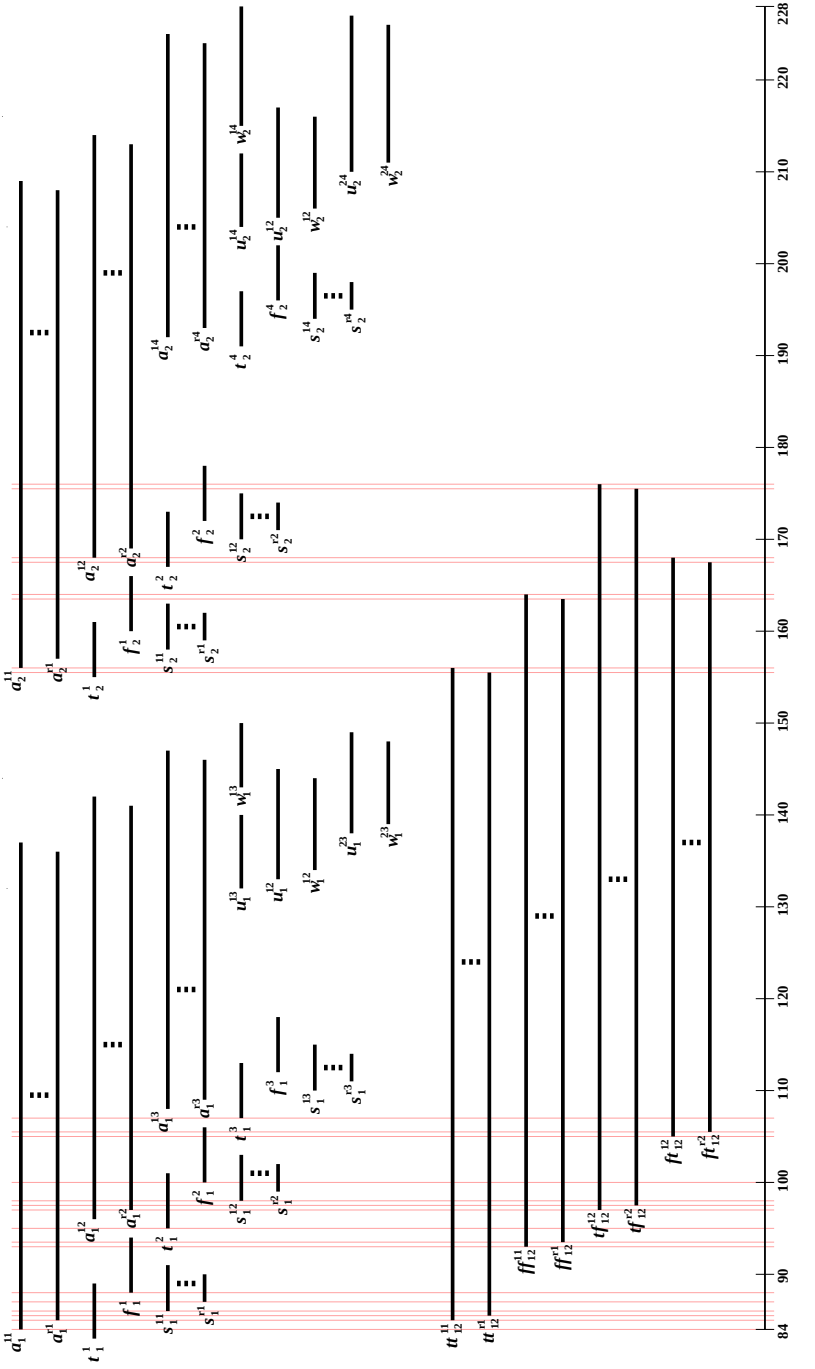



**Fig. 3.** The intervals constructed from the two clause and four variable formula $(x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_4)$

as follows. Without loss of generality we consider intervals of type $tt$. The construction is identical for intervals of type $ff$, $tf$, and $ft$. Suppose that $I$ contains the interval $tt^i_{jk}$. To $J$ we add $r$ independent intervals named $tt^{1i}_{jk}, tt^{2i}_{jk}, \cdots, tt^{ri}_{jk}$ such that (i) $tt^{1i}_{jk} = tt^i_{jk}$, (ii) $tt^{l+1i}_{jk} \subset tt^{li}_{jk}$, and (iii) the endpoints of the intervals $tt^{li}_{jk}$ are chosen close enough so that for all $l$, the intervals $tt^{li}_{jk}$ overlap exactly the same set of intervals. As before, we call $tt^{1i}_{jk}$ an *original* and the rest of the type $tt$ intervals *copies*. We use a similar nomenclature for the type $ft$, $tf$, and $ff$ intervals. This completes the construction of $J$. We are now ready to prove the following lemma.

**Lemma 2.** *If $F$ is satisfiable, then a minimum independent dominating set in $G(J)$ has size $5m$ or less. If $F$ is not satisfiable, then a minimum independent dominating set in $G(J)$ has size greater than $5\alpha m$.*

*Proof.* Suppose that $F$ is satisfiable. As shown in the proof of Theorem 1, we can construct an independent set $D$ of size $5m$ of type $t$, type $f$, type $u$ and type $w$ intervals that dominates all intervals in $I - D$. These intervals in $D$ appear in $J$ as well. Clearly, every interval in $J - D$ that appears in $I - D$ is dominated by some interval in $D$. The intervals in $J - D$ that do not appear in $I - D$ are copies. Since all the originals are dominated and the copies have exactly the same neighbors as the originals, and since none of the originals are in $D$, it follows that the copies are also dominated. This ensures that all intervals in $J - D$ are dominated by intervals in $D$.

Now suppose that $F$ is not satisfiable. Let $D$ be a smallest set of independent intervals in $J$ that dominates all intervals in $J - D$. Suppose that $D$ contains an interval of type $a$ - say $a^i_j$. Since intervals in $D$ are independent, $D$ cannot contain any interval that overlaps with $a^i_j$. Since the original and all the copies of type $a$ have exactly the same set of neighbors, to be dominated, all intervals of type $a$ must be in $D$. A similar argument can be made for intervals of type $tt$, $ft$, $tf$, $ff$, and $s$. This means that if $D$ contains any of these intervals, then its size is at least $r = 5\alpha m$. Now suppose that $D$ does not contain an interval of the following types: $a$, $tt$, $tf$, $ft$, $ff$, or $s$. The absence of the type $s$ intervals in $D$ implies that for each appropriate $i$, $j$ pair, $D$ contains $t^i_j$ or $f^i_j$. Thus we have an assignment of truth values to all the literals. The absence of type $tt$, $ff$, $tf$, and $ft$ intervals implies that these truth values are consistent with each other. This implies that for some clause $C_j$ and for all relevant $i$, $f^i_j \in D$. This leaves all the type $a_j$ intervals undominated and these cannot be dominated by the inclusion of the type $u$ and type $w$ intervals in $D$. This implies that at least one type $a_j$ interval has to be included in $D$. But, if one type $a_j$ interval is included in $D$, then all $r = 5\alpha m$ type $a_j$ intervals have to be included in $D$. This implies that if $F$ is not satisfiable, any dominating set of $J$ contains at least $5\alpha m$ intervals.

To complete the proof of this theorem, it remains to show that the transformation from $F$ to $J$ takes time that is polynomial in the size of $F$ when $\alpha \leq |J|^{\varepsilon}$. The following lemma states this claim formally. The proof is omitted due to space constraints and mainly consists of showing that for all $\alpha \leq |J|^{\varepsilon}$, the size of $J$ is bounded above by a polynomial in $m$.

**Lemma 3.** *For any $\varepsilon$, $0 \leq \varepsilon < 1$, and any $\alpha \leq |J|^{\varepsilon}$, $|J|$ is bounded above by a polynomial function of $m$.*

It is easy to see that the reduction described at the beginning of the proof takes time proportional to $|J|$. By showing that when $\alpha \leq |J|^{\varepsilon}$, $|J|$ is bounded above by a polynomial function of $m$, we have established that the reduction is a polynomial time reduction.

## 3 Related Hardness of Approximation Results

Let BMIDS be the minimum independent dominating set problem restricted to bipartite graphs. Irving [12] shows that for any fixed $\alpha > 1$, no polynomial time $\alpha$-approximation algorithm exists for BMIDS unless P=NP. We strengthen this result to show that BMIDS is not approximable within a factor of $n^{\varepsilon}$, for any $0 \leq \varepsilon < 1$, unless P = NP. Proofs of theorems in this section are omitted due to space constraints.

**Theorem 3.** *For any $\varepsilon$, $0 < \varepsilon < 1$, BMIDS does not have an $n^{\varepsilon}$-approximation algorithm unless P = NP. Here $n$ is the number of intervals in the input to BMIDS.*

We now determine the complexity of the "bichromatic" minimum independent dominating set problem and show that this problem is also hard to approximate. This problem is motivated by problems in polygon decomposition (see [7] for details). Let `TRUE`, `INDEP`, `CONNECT`, `TOTAL`, and `CLIQUE` refer to properties of a subset of vertices of a graph. In particular, for any graph $G = (V, E)$, any subset of $V$ has the property `TRUE`, any independent subset of $V$ has the property `INDEP`, any subset of $V$ that induces a connected subgraph has the property `CONNECT`, any subset of $V$ that induces a subgraph with no isolated vertices has the property `TOTAL`, and any subset of $V$ that induces a clique has the property `CLIQUE`. Letting $\pi \in \{\texttt{TRUE}, \texttt{INDEP}, \texttt{CONNECT}, \texttt{TOTAL}, \texttt{CLIQUE}\}$ we have the following classes of problems.

**Minimum $\pi$-Dominating Set on Circle Graphs** ($\pi$-CMDS)
`INPUT:` An interval representation of a circle graph $G = (V, E)$ and a natural number $k$.
`QUESTION:` Does $G$ have a dominating set of size no greater than $k$, having property $\pi$?

**Minimum $\pi$-Dominating Set on Bichromatic Circle Graphs** ($\pi$-BCMDS)
`INPUT:` An interval representation of a circle graph $G = (V, E)$, each of whose vertices is coloured `RED` or `BLUE`, and a natural number $k$.
`QUESTION:` Does $G$ have a `RED` set of vertices $R$ of size no greater than $k$ such that (i) $R$ has property $\pi$ and (ii) all the `BLUE` vertices are dominated by $R$.

**Theorem 4.** *For any $\pi \in \{\texttt{TRUE}, \texttt{INDEP}, \texttt{CONNECT}, \texttt{TOTAL}, \texttt{CLIQUE}\}$ there is a polynomial time reduction from $\pi$-CMDS to $\pi$-BCMDS.*

**Corollary 1.** *$\pi$-BCMDS is NP-complete for $\pi \in \{\texttt{TRUE}, \texttt{INDEP}, \texttt{CONNECT}, \texttt{TOTAL}\}$. Furthermore, for $\pi = \texttt{INDEP}$, there exists no $n^{\varepsilon}$-approximation algorithm for $\pi$-BCMDS on $n$-vertex bichromatic circle graphs, unless $P = NP$.*

## 4 Conclusions

The results in our paper show that independent domination problems on circle graphs are harder than expected. The topic of approximation algorithms for dominating set problems on restricted classes of graphs has not received much attention from researchers as yet. Our result in this paper — MIDS on circle graphs is extremely hard to approximate — along with the results on the existence of polynomial time constant-factor approximation algorithms for MDS, MTDS and MCDS problems (Damian-Iordache and Pemmaraju, *in this proceedings*), are probably the first of this kind.

*Acknowledgement:* We thank J. Mark Keil for sharing with us preliminary ideas on the NP-completeness proof of CMIDS

## References

1. Sanjeev Arora and Carsten Lund. Hardness of approximation. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-hard problems*. PWS Publishing Company, 1997.
2. T. A. Beyer, A. Proskurowski, S.T. Hedetniemi, and S. Mitchell. Independent domination in trees. *Congressus Numerantium*, 19:321–328, 1977.
3. K. S. Booth. Dominating sets in chordal graphs. Technical Report CS-80-34, Department of Computer Science, University of Waterloo, Waterloo, Ontario, 1980.
4. K. S. Booth and J. H. Johnson. Dominating sets in chordal graphs. *SIAM Journal on Computing*, 13:335–379, 1976.
5. C. J. Colbourn and L. K. Stewart. Permutation graphs: connected domination and Steiner trees. *Discrete Mathematics*, 86:179–189, 1990.
6. D. G. Corneil and Y. Perl. Clustering and domination in perfect graphs. *Discrete Applied Mathematics*, 9:27–39, 1984.
7. Mirela Damian-Iordache and Sriram V. Pemmaraju. Domination in circle graphs with applications to polygon decomposition. Technical Report 99-02, University of Iowa, 1999.
8. A. K. Dewdney. Fast turing reductions between problems in NP, chapter 4: Reductions between NP-complete problems. Technical Report 71, Department of Computer Science, University of Western Ontario, London, Ontario, 1983.
9. M. Farber. Independent domination in chordal graphs. *Operations Research Letters*, 1:134–138, 1982.
10. M. Farber and J. M. Keil. Domination in permutation graphs. *Journal of Algorithms*, 6:309–321, 1985.
11. T. W. Haynes, S. T. Hedetniemi, and P. J. Slater. *Fundamentals of Domination in Graphs*. Number 208 in Pure and Applied Mathematics: A series of monomgraphs and textbooks. Marcel Dekker Inc., New York, 1998.

12. Robert W. Irving. On approximating the minimum independent dominating set. *Information Processing Letters*, 37:197–200, 1991.
13. D. S. Johnson. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 6:434–451, 1985.
14. J. Mark Keil. The complexity of domination problems in circle graphs. *Discrete Applied Mathematics*, 42:51–63, 1991.
15. D. Kratsch and L. Stewart. Domination on cocomparability graphs. *SIAM Journal on Discrete Mathematics*, 6(3):400–417, 1993.
16. R. Laskar and J. Pfaff. Domination and irredundance in split graphs. Technical Report 430, Clemson University, 1983.
17. Y. D. Liang. Steiner set and connected domination in trapezoid graphs. *Information Processing Letters*, 56:101–108, 1995.
18. M.R.Garey and D.S.Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman and Company, New York, 1979.
19. J. Pfaff, R. Laskar, and S. T. Hedetniemi. NP-completeness of total and connected domination, and irredundance for bipartite graphs. Technical Report 428, Clemson University, 1983.

# Constant-Factor Approximation Algorithms for Domination Problems on Circle Graphs

Mirela Damian-Iordache[1] and Sriram V. Pemmaraju[2]

[1] Department of Computer Science, University of Iowa, Iowa City, IA 52242, U.S.A. damianjo@cs.uiowa.edu
[2] Department of Mathematics, Indian Institute of Technology, Bombay, Powai, Mumbai 400076, India. sriram@math.iitb.ernet.in

**Abstract.** A graph $G = (V, E)$ is called a *circle graph* if there is a one-to-one correspondence between vertices in $V$ and a set $C$ of chords in a circle such that two vertices in $V$ are adjacent if and only if the corresponding chords in $C$ intersect. A subset $V'$ of $V$ is a *dominating set* of $G$ if for all $u \in V$ either $u \in V'$ or $u$ has a neighbor in $V'$. In addition, if $G[V']$ is connected, then $V'$ is called a *connected dominating set*; if $G[V']$ has no isolated vertices, then $V'$ is called a *total dominating set*. Keil (*Discrete Applied Mathematics*, 42 (1993), 51-63) shows that the minimum dominating set problem (MDS), the minimum connected dominating set problem (MCDS) and the minimum total domination problem (MTDS) are all NP-complete even for circle graphs. He mentions designing approximation algorithms for these problems as being open. This paper presents $O(1)$-approximation algorithms for all three problems — MDS, MCDS, and MTDS on circle graphs. For any circle graph with $n$ vertices and $m$ edges, these algorithms take $O(n^2 + nm)$ time and $O(n^2)$ space. These results, along with the result on the hardness of approximating minimum independent dominating set on circle graphs (Damian-Iordache and Pemmaraju, *in this proceedings*) advance our understanding of domination problems on circle graphs significantly.

## 1 Introduction

A graph $G = (V, E)$ is called a *circle graph* if there is a one-to-one correspondence between vertices in $V$ and a set $C$ of chords in a circle such that two vertices in $V$ are adjacent if and only if the corresponding chords in $C$ intersect. $C$ is called the *chord intersection model* for $G$. Equivalently, the vertices of a circle graph can be placed in one-to-one correspondence with the elements of a set $I$ of intervals such that two vertices are adjacent if and only if the corresponding intervals overlap, but neither contains the other. $I$ is called the *interval model* of the corresponding circle graph. Representations of a circle graph as a graph or as a set of chords or as a set of intervals are equivalent via linear time transformations. So, without loss of generality, in specifying instances of problems, we assume the availability of the representation that is most convenient.

For a graph $G = (V, E)$, a subset $V'$ of $V$ is a *dominating set* of $G$ if for all $u \in V$ either $u \in V'$ or $u$ has a neighbor in $V'$. In addition, if no two

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 70–82, 1999.
© Springer-Verlag Berlin Heidelberg 1999

vertices in $V'$ are adjacent, then $V'$ is called an *independent dominating set*; if the subgraph of $G$ induced by $V'$, denoted $G[V']$, is connected, then $V'$ is called a *connected dominating set*; if $G[V']$ has no isolated nodes, then $V'$ is called a *total dominating set*; and if $G[V']$ is a clique, then $V'$ is called a *dominating clique*.

Garey and Johnson [4] mention that problems of finding a minimum cardinality dominating set (MDS), minimum cardinality independent dominating set (MIDS), minimum cardinality connected dominating set (MCDS), minimum cardinality total dominating set (MTDS), and minimum cardinality dominating clique (MDC) are all NP-complete for general graphs. Johnson [2] seems to be the first to identify MDS as an open problem for circle graphs. Little progress was made towards solving this problem until Elmallah, Stewart and Culberson defined the class of *k-polygon graphs* [1]. These are the intersection graphs of straight-line chords inside a convex $k$-sided polygon. Permutation graphs form a proper subset of the class of 3-polygon graphs and the class of circle graphs is the (infinite) union of the classes of $k$-polygon graphs for all $k \geq 3$. For fixed $k$, Elmallah et al. were able to provide a polynomial time algorithm for MDS. Finally, Keil [3] resolved the complexity of MDS on circle graphs by showing that it is NP-complete. In the same paper, he also showed that MCDS and MTDS are also NP-complete for circle graphs. In that paper, Keil does not investigate the question of approximation algorithms, however he mentions the construction of approximation algorithms for MDS, MCDS, and MTDS as being open.

An *$\alpha$-approximation* algorithm for a minimization problem is a polynomial time algorithm that guarantees that the ratio of the cost of the solution to the optimal (over all instances of the problem) does not exceed $\alpha$. In this paper we present an 8-approximation algorithm for MDS, a 14-approximation algorithm for MCDS, and a 10-approximation algorithm for MTDS on circle graphs. Our algorithms use $O(n^2)$ space and run in $O(n^2 + nm)$ time, where $n$ is the number of vertices and $m$ is the number of edges in the input circle graph. The algorithms for MCDS and MTDS are obtained by making simple modifications to the algorithm for MDS.

## 2 Approximating Dominating Set Problems on Circle Graphs

The minimum dominating set problem on circle graphs (CMDS) is as follows.

**Minimum Dominating Set for Circle Graphs** (CMDS)
INPUT: A set $T$ of chords in a circle.
OUTPUT: A smallest dominating set $U \subseteq T$.

Keil shows that CMDS is NP-complete [3]. In the following we present an approximation algorithm for CMDS that produces a dominating set of size within a factor of 8 of optimal.

Let $n = |T|$ and let $m$ be the number of pairwise intersections of chords in $T$. In other words, $n$ and $m$ are the number of vertices and number of edges, respectively of the corresponding circle graph. Let $1, 2, \ldots, 2n$ be the sequence of endpoints of chords in $T$ listed in counterclockwise order, starting at an arbitrary

endpoint. Without loss of generality, we assume that the endpoints of the chords are all distinct. For any chord $c \in T$, the endpoint with smaller (respectively, larger) label is called its *left endpoint* (respectively *right endpoint*), denoted $l(c)$ (respectively $r(c)$). For a set of chords $C \subseteq T$, let $E[C]$ denote the set of all endpoints of chords in $C$. For any points $i, j \in E[T]$ let $\texttt{sector}(i, j)$ denote the arc of the circle obtained by starting from $i$ and moving in counterclockwise direction until $j$ is reached. We assume that $\texttt{sector}(i, j)$ is an open set, that is, it does not include its endpoints. Occasionally, we will need sectors that contain their endpoints; we will use $\texttt{closure}(\texttt{sector}(i, j))$ to denote a sector that contains its endpoints $i$ and $j$. For any sector $s$, we call the endpoint with smaller (respectively, larger) label, the *left endpoint* (respectively, *right endpoint*) of $s$, denoted $l(s)$ (respectively, $r(s)$). A set of sectors $s_1, s_2, \cdots s_m$ is a *chain* if $r(s_i) = l(s_{i+1})$, $i = 1 \cdots m - 1$. Whenever we talk about a set of sectors, it is understood that the sectors are pairwise disjoint. For any chain $C$ let $l(C)$ denote the left endpoint of the first sector in $C$ and $r(C)$ denote the right endpoint of the last sector in $C$.

**Definition:** We say that $\texttt{sector}(i, j)$ is a *leaf sector* if and only if (i) there is no chord $c$ in $T$ with $i < l(c) < r(c) < j$ and (ii) any chord incident on a point in $\texttt{sector}(i, j)$ is cut by a chord with both endpoints outside $\texttt{sector}(i, j)$.

Let $LS$ denote the set of all leaf sectors. Note that the definition of a leaf sector is with respect to the given set of chords $T$ and therefore it may be more appropriate to denote the set of leaf sectors by $LS(T)$. Since $T$ is fixed, for convenience we will simply use $LS$. A chain of leaf sectors $C$ is a *leaf sector cover* if $\texttt{sector}(r(C), l(C))$ is also a leaf sector. The reason for defining a leaf sector cover will become clear from the following proposition, which is illustrated in Figure 1.

**Proposition 1.** *Let $D \subseteq T$ be a dominating set of $T$ of size $d$. Suppose that $i_1 < i_2 < \ldots < i_{2d}$ are the points in $E[D]$. Let $s_j = \texttt{sector}(i_j, i_{j+1})$, for all $j$, $1 \leq j < 2d$. Then $s_1, s_2, \ldots, s_{2d-1}$ is a leaf sector cover.*

Thus all dominating sets of $T$, including a minimum dominating set, induce a leaf sector cover that is roughly twice the size of the dominating set. Our goal is to compute an optimal leaf sector cover and then use it to compute a dominating set.

### 2.1 Computing an Optimal Leaf Sector Cover

For any pair of points $i, j \in E[T]$, $i < j$, we define $OC(i, j)$ to be a smallest chain $C$ of leaf sectors with $l(C) = i$ and $r(C) = j$. The following lemma establishes the optimal substructure property of $OC(i, j)$. Using this property, it is easy to compute $OC(i, j)$ using dynamic programming, given the set of leaf sectors $LS$.

**Lemma 1.** *For all $i, j \in E[T]$, $i < j$, $OC(i, j)$ satisfies the following recurrence:*

$$OC(i,j) = \begin{cases} \texttt{sector}(i,j) & \text{if } \texttt{sector}(i,j) \in LS \\ \min_{i<k<j}\{OC(i,k) \oplus \texttt{sector}(k,j) \mid \texttt{sector}(k,j) \in LS\} & \text{otherwise} \end{cases}$$

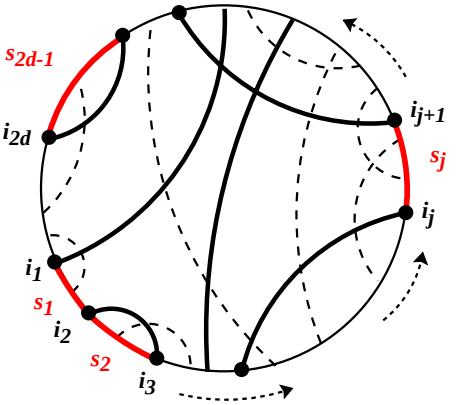



**Fig. 1.** $s_1, s_2, \ldots, s_{2d-1}$ is a leaf sector cover

*Here the* $\oplus$ *operator stands for concatenation and the* min *operator returns the chain with minimum cardinality.*

*Proof.* The proof is by induction on $(j - i)$. The base case is when $j - i = 1$ and the recurrence relation is trivially true. The inductive hypothesis is that for some natural number $s$ and for all $i$, $j$, such that $1 \leq j - i \leq s$, the above recurrence relation is true. To prove the inductive step we consider the leaf chain $OC(i, j)$ for some $i, j$ such that $j - i = s + 1$. Let $\texttt{sector}(q, j)$ be the last sector in $OC(i, j)$. In the recurrence above, the variable $k$ takes on the value $q$ also, therefore the size of the right hand side is no greater than that of the left hand side. We now show that the right hand side is no less than the left hand side. Suppose there existed a $k$ such that

$$|OC(i, j)| > |OC(i, k) \oplus \texttt{sector}(k, j)|. \tag{1}$$

But $C = OC(i, k) \oplus \texttt{sector}(k, j)$ is a chain of leaf sectors with the property that $l(C) = i$ and $r(C) = j$. By definition, this makes $C$ a candidate for $OC(i, j)$, therefore $|OC(i, j)| \leq |C|$. This contradicts inequality (1) and we are done.

It is obvious that $OC(1, 2n)$ is a leaf sector cover since $\texttt{sector}(2n, 1)$ is a leaf sector. The following proposition claims that this leaf sector cover is good enough for our purposes.

**Proposition 2.** *Let* $D^*$ *be a minimum dominating set of* $T$*. If* $|D^*| = d^*$*, then* $|OC(1, 2n)| \leq 2d^* + 2$.

So our goal is to compute $OC(1, 2n)$. As mentioned earlier, given the set of leaf sectors $LS$, we can compute $OC(1, 2n)$ using dynamic programming. To compute $OC(1, 2n)$ using the recurrence of Lemma 1, we need to compute $OC(1, j)$, for all $j$, $1 < j \leq 2n$. Computing $OC(1, j)$ given $OC(1, k)$, for all $k$, $1 < k < j$ takes $O(j)$ time for a total of $\sum_{j=1}^{2n} O(j) = O(n^2)$ time to compute $OC(1, 2n)$. In the next subsection we show how to compute the set $LS$ in $O(nm)$ time.

### 2.2 Identifying Leaf Sectors

For convenience we assume that the set $LS$ is stored in a $2n \times 2n$ boolean matrix called `leafsector`. For any $i$ and $j$, $1 \leq i < j \leq 2n$, `leafsector`$[i, j]$ is 1 if and only if `sector`$(i, j)$ is a leaf sector. Since we are only interested in `sector`$(i, j)$ for $i < j$, only the strict upper triangular portion of the matrix is relevant. Furthermore, if a sector `sector`$(i, j)$ is not a leaf sector, then sectors `sector`$(i, k)$ for all $k$, $j < k \leq 2n$ are not leaf sectors and similarly sectors `sector`$(k, j)$ for all $k$, $1 \leq k < i$ are not leaf sectors. This implies that if a certain entry `leafsector`$[i, j] = 0$ then all the entries in the submatrix to the northeast of entry $[i, j]$ are 0. This observation allows us to define an order for processing the sectors efficiently. We start with the leaf sector `sector`$(1, 2)$. Suppose that at a certain stage we have determined whether `sector`$(i, j)$ is a leaf sector. The next sector to process can be chosen as follows depending on whether `sector`$(i, j)$ is a leaf sector.

- Suppose that `sector`$(i, j)$ is a leaf sector. We are still looking for the first sector in row $i$ which is not a leaf sector. So we then process `sector`$(i, j+1)$.
- Suppose that `sector`$(i, j)$ is not a leaf sector. Then we know that for all $k > j$, `sector`$(i, k)$ is not a leaf sector and for all $k < j$, `sector`$(i, k)$ is a leaf sector. We know the latter because if `sector`$(i, k)$ was not a leaf sector for some $k < j$, we would have stopped processing the sectors in row $i$ earlier, that is, at column $k$. So we process `sector`$(i + 1, j)$ next.

For any $i \in E[T]$, let $T[i]$ denote the chord in $T$ incident on $i$. We will show that having determined that `sector`$(i, j)$ is a leaf sector, we can determine if `sector`$(i, j + 1)$ is a leaf sector in $O(\text{degree}(T[j]))$ time. Similarly, having determined that `sector`$(i, j)$ is not a leaf sector, we can determine if `sector`$(i + 1, j)$ is a leaf sector in $O(\text{degree}(T[j-1]))$ time. It is easy to see that only $O(n)$ sectors are considered for processing. From these observations, it follows that the total work done is $O(nm)$.

In the following we describe how to compute the value of `leafsector`$[i, j+1]$ or `leafsector`$[i + 1, j]$, having computed the value of `leafsector`$[i, j]$. First some definitions. For a given chord $c \in T$ and a point $i$, $1 \leq i \leq 2n$, let `closestpoint`$(i, c)$ denote the endpoint of $c$ first encountered in a counterclockwise walk starting from $i$. If $c = (i, j)$ is a chord in $T$, then `closestpoint`$(i, c) = i$ and `closestpoint`$(j, c) = j$. Define the *distance* between a point $i \in E[T]$ and a chord $c \in T$, denoted `distance`$(i, c)$ as the number of points in $E[T]$ encountered in the counterclockwise walk starting at $i$ and ending at `closestpoint`$(i, c)$. In computing `distance`$(i, c)$ we include $i$ but not `closestpoint`$(i, c)$. Thus `distance`$(i, c)$ measures the number of "hops" to get to the closer endpoint of $c$ from $i$ traveling in counterclockwise order. For any two points $i$ and $j$, $1 \leq i < j \leq 2n$, define `farthestcut`$(i, j)$ as the chord $c \in T$ that cuts $T[j]$ and maximizes `distance`$(i, c)$. Some of these definitions are illustrated in Figure 2. The following proposition is immediate from the definition of a leaf sector.

**Proposition 3.** *A sector* `sector`$(i, j)$ *is a leaf sector iff for all* $k$, $i < k < j$, `farthestcut`$(i, k)$ *has both endpoints outside* `sector`$(i, j)$.

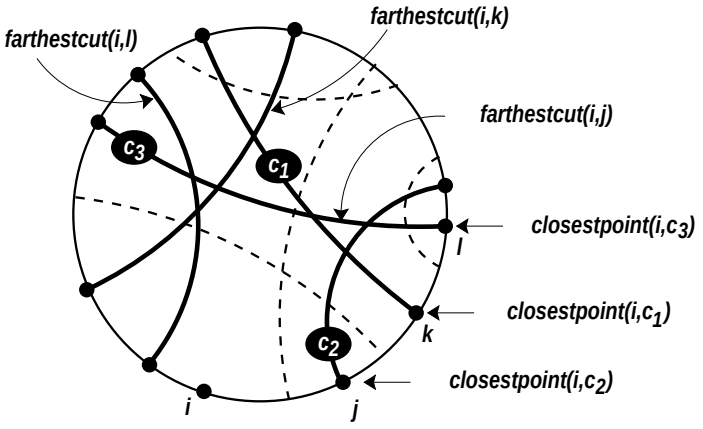



**Fig. 2.** An example illustrating the `closestpoint` and `farthestcut` definitions

We use the above proposition to determine if a sector is a leaf sector. Suppose that we have determined that $\texttt{sector}(i,j)$ is a leaf sector, assigned 1 to $\texttt{leafsector}[i,j]$ and in the process computed $\texttt{farthestcut}(i,k)$ for all $k$, $i < k < j$. To determine the status of $\texttt{sector}(i,j+1)$ we need to do the following:

1. Check if $T[j] = \texttt{farthestcut}(i,k)$ for any $k$, $i < k < j$. If this is the case, then $\texttt{sector}(i,j+1)$ is not a leaf sector. To check this condition examine each chord $c$ that cuts $T[j]$ and check if $c$ is incident on $k$ for some $k$, $i < k < j$. If so, check $\texttt{farthestcut}(i,k)$ and check if $\texttt{farthestcut}(i,k) = T[j]$. This takes $O(\text{degree}(T[j]))$.
2. Check if $\texttt{farthestcut}(i,j)$ lies outside $\texttt{sector}(i,j)$. It is easy to see that $\texttt{farthestcut}(i,j)$ can be computed in $O((\text{degree}(T[j]))$ time.

Suppose now that we have determined that $\texttt{sector}(i,j)$ is not a leaf sector, assigned 0 to $\texttt{leafsector}[i,j]$ and in the process computed $\texttt{farthestcut}(i,k)$ for all $k$, $i < k < j$. We now want to determine the status of $\texttt{sector}(i+1,j)$. In this case we know that $\texttt{sector}(i,j-1)$ is a leaf sector. This implies that $\texttt{sector}(i+1,j-1)$ is also a leaf sector. It is easy to see that if $\texttt{sector}(i+1,j-1)$ is a leaf sector, then $\texttt{farthestcut}(i+1,k) = \texttt{farthestcut}(i,k)$ for all $k$, $i+1 < k \leq j-1$. Using the fact that $\texttt{sector}(i+1,j-1)$ is a leaf sector and using information associated with it, we can determine if $\texttt{sector}(i+1,j)$ is a leaf sector, just as described above in time $O(\text{degree}(T[j-1]))$.

This completes our description of how $LS$ is computed. Note that we have assumed that $LS$ is stored in the `leafsector` matrix, only for convenience. Ignoring the lower triangular portion of the matrix, we see that each row contains a contiguous sequence of 1's followed by a contiguous sequence of 0's. Thus we need to only store the index of the last 1 in each row. This takes $O(n)$ space (as opposed to $O(n^2)$ space for the matrix) and provides us with an $O(1)$ time test for determining if $\texttt{sector}(i,j)$ is a leaf sector.

We now show how $OC(1, 2n)$ can be used to compute a small dominating set. For this we make use of a "structural" property of leaf sectors proved in the next subsection.

### 2.3 A Structural Property of Leaf Sectors

Two sectors $s_i$ and $s_j$ are said to be *connected* in $T$ if there is a chord $c \in T$ with one endpoint in $s_i$ and the other endpoint in $s_j$. In this case, $c$ is said to connect sectors $s_i$ and $s_j$. Let $T_{ij}$ be the set of chords that connect sectors $s_i$ and $s_j$.

**Lemma 2.** *Let $s_i$ and $s_j$ be two leaf sectors which are connected in $T$. Then $T_{ij}$ can be cut with at most two chords in $T$.*

*Proof.* Assume without loss of generality that $r(s_i) < l(s_j)$. By the definition of a leaf sector, any chord incident on a point in $s_i$ is cut by a chord with both endpoints outside $s_i$ (and similarly for $s_j$). Assume first that there exists a chord $c \in T$ with one endpoint in $q_1 = \texttt{closure}(\texttt{sector}(r(s_i), l(s_j)))$ and the other endpoint in $q_2 = \texttt{closure}(\texttt{sector}(r(s_j), l(s_i)))$. Then $c$ cuts all chords in $T_{ij}$ and the claim is true. So assume that there is no chord in $T$ with one endpoint in $q_1$ and the other endpoint in $q_2$. Let $U \subset T$ be the set of chords with one endpoint in $s_i$ and the other endpoint in $q_1$ or $q_2$. Since $s_j$ is a leaf chord, $U \neq \perp$ and cuts all chords in $T_{ij}$ (which are incident on points in $s_j$). We greedily pick a chord $c$ in $U$ that cuts most of the chords in $T_{ij}$. Assume without loss of generality that $c$ has one endpoint in $q_1$. If $c$ cuts all chords in $T_{ij}$, then the claim is true; otherwise let $B \subset T_{ij}$ be the set of chords in $T_{ij}$ not cut by $c$. Let $t$ be the chord in $B$ with the endpoint $l(t)$ closest to $l(c)$. Refer to Figure 3. Consider a chord $d \in U$ that cuts $t$. If $d$ has one endpoint in $q_1$, then $d$ cuts all chords cut by $c$ plus $t$, contradicting the fact that $c$ cuts most of the chords in $T_{ij}$. Hence $d$ must have one endpoint in $q_2$. Since $t$ is the chord with the rightmost left endpoint in $B$, it follows that $d$ cuts all chords in $B$. Thus $c$ and $d$ cut all chords in $T_{ij}$.

For any pair of leaf sectors $s_i$ and $s_j$, define $\texttt{cutset}(s_i, s_j)$ as a smallest set of chords satisfying the following properties: (i) every chord connecting $s_i$ and $s_j$ is cut by some chord in $\texttt{cutset}(s_i, s_j)$ and (ii) no chord in $\texttt{cutset}(s_i, s_j)$ connects $s_i$ and $s_j$. The above lemma tells us that $\texttt{cutset}(s_i, s_j)$ is well defined for any pair of leaf sectors $s_i$ and $s_j$ and has size at most 2. The proof of Lemma 2 yields an algorithm to compute $\texttt{cutset}(s_i, s_j)$ that takes time $O(\sum_{c \in T_{ij}} \text{degree}(c))$.

### 2.4 Computing a Dominating Set From a Leaf Sector Cover

We now show how using the structural property of leaf sectors proved in the previous section, we can compute a dominating set of $T$ that is within 4 times the size of $OC(1, 2n)$. Since the size of $OC(1, 2n)$ is roughly within twice the size of an optimal dominating set, we have an 8-approximation of a minimum dominating set.

Some definitions before we start describing the algorithm. Given a sequence of sectors $S = (s_1, s_2, \cdots s_k)$, the *connectivity graph* of $S$, denoted $K(S)$, is the

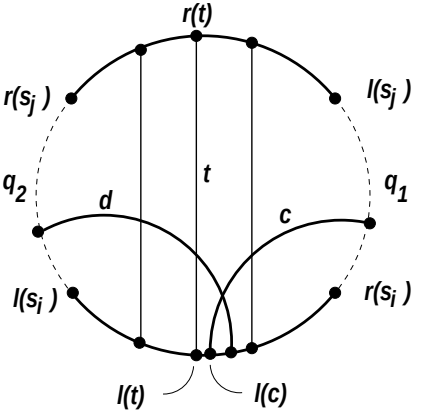



**Fig. 3.** Chords connecting $s_i$ and $s_j$ are cut with two chords $c$ and $d$ in $T$

graph with vertex set $S$ and with $s_i$ adjacent to $s_j$, $1 \leq i, j \leq k$, if and only if $i \neq j$ and there is a chord in $T$ that connects $s_i$ and $s_j$. Given a connectivity graph $G$, a *circle drawing* of $G$ is obtained by associating with each vertex in $G$ a distinct point on the circumference of a circle and by drawing the edges of $G$ as chords connecting points on the circle that are associated with end vertices of the edge. Furthermore, we require that in a circle drawing the points corresponding to sectors are placed in the same order on the circumference of the circle, as the sectors themselves. The chords in the circle drawing of a connectivity graph are called *drawing edges*.

In the following we describe the computation of a set $ADS \subseteq T$ of chords that dominates $T$. Suppose that initially $ADS = \emptyset$. Let $J \subseteq T$ be the set of chords incident on the endpoints of sectors in $OC(1, 2n)$. Note that it is not required that *both* endpoints of chords in $J$ coincide with endpoints of sectors in $OC(1, 2n)$. We start by adding $J$ to $ADS$. Let $C^* = OC(1, 2n) \oplus \texttt{sector}(2n, 1)$. We use the connectivity graph of $C^*$, namely $K(C^*)$, and in particular, a circle drawing $D$ of $K(C^*)$, to identify the chords to add to $ADS$. Let $P$ be a maximal set of pairwise non-intersecting chords in $D$. Remove from $P$ all chords that connect adjacent sectors (that is, sectors that share an endpoint) in $C^*$. Corresponding to each drawing edge $d = (s_i, s_j) \in P$, we add to $ADS$ a chord in $T$ connecting $s_i$ and $s_j$. By the definition of a drawing edge, such a chord always exists. Corresponding to each drawing edge $d = (s_i, s_j) \in P$, we add $\texttt{cutset}(s_i, s_j)$ to $ADS$. A summary of the entire approximation algorithm that computes a dominating set of the given circle graph is given in Figure 4.

### 2.5 Analysis of the Algorithm

The following theorems establish that the above algorithm does produce a dominating set, that the size of this dominating set is within a factor of 8 of the optimal, and that the running time of the algorithm is $O(n^2 + nm)$. Let $MDS$ be a minimum dominating set of the given circle graph.

APPROXIMATION ALGORITHM FOR CMDS
**Input:** a set of chords $T$
**Output:** a set of chords $ADS$ that dominates $T$

Step 1. compute the set $LS$ of leaf sectors
Step 2. compute the chain of leaf sectors $OC(1, 2n)$
Step 3. assign to $ADS$ the set of chords incident on sectors in $OC(1, 2n)$
let $C^*$ be $OC(1, 2n) \oplus \texttt{sector}(2n, 1)$
Step 4. find a maximal set of non-intersecting chords $P$ in the circle drawing $D$ of $K(C^*)$ remove from $P$ drawing edges that connect sectors adjacent in $C^*$
Step 5. **for** each drawing edge $d = (s_i, s_j) \in P$ **do**
pick an arbitrary chord connecting $s_i$ and $s_j$ and add it to $ADS$
add $\texttt{cutset}(s_i, s_j)$ to $ADS$
**return** $ADS$

**Fig. 4.** An outline of the approximation algorithm for CMDS

**Theorem 1.** *$ADS$ is a dominating set of $T$.*

*Proof.* Let $c$ be an arbitrary chord in $T - ADS$. We show that $c$ is cut by one of the chords in $ADS$. Recall that all chords incident on endpoints of sectors in $C^*$ are included in $ADS$. Therefore, there are sectors $s_i$ and $s_j$ in $C^*$ such that $c$ connects $s_i$ and $s_j$. Then there is a drawing edge $d = (s_i, s_j)$ in $K(C^*)$. We now consider two cases, depending on whether $d$ belongs to $P$ or not. Suppose first that $d$ belongs to $P$. In this case corresponding to $d$, we have added to $ADS$, the set of chords $\texttt{cutset}(s_i, s_j)$ that cut all chords connecting $s_i$ and $s_j$. Assume now that $d$ does not belong to $P$. In this case $d$ cuts at least one of the drawing edges in $P$. Let $f = (s_l, s_t)$ be a drawing edge in $P$ that cuts $d$. Corresponding to $f$, we have included in $ADS$ a chord $g \in T$ with one endpoint in $s_l$ and the other endpoint in $s_t$. It is easy to see that $g$ cuts all chords connecting $s_i$ and $s_j$. Since the choice of $c$ was arbitrary we conclude that $ADS$ is a dominating set of $T$.

**Theorem 2.** $|ADS| \leq 8|MDS| - 1$.

*Proof.* Let $q = |C^*|$. Proposition 2 implies that

$$q \leq 2|MDS| + 2 \tag{2}$$

$ADS$ contains chords in $J$, plus a chord corresponding to each chord in $P$, plus at most 2 chords in $\texttt{cutset}(s_i, s_j)$ for each chord $(s_i, s_j) \in P$. Thus

$$|ADS| \leq |J| + 3|P| \tag{3}$$

Clearly,

$$|J| \leq q \tag{4}$$

$P$ is an outerplanar embedding of a graph with $q$ vertices and such a graph has at most $2q-3$ edges. However, by removing edges between adjacent sectors, we leave at most $q-3$ edges in $P$. Hence

$$|P| \leq q-3 \tag{5}$$

Substituting inequalities (4) and (5) in (3) we get

$$|ADS| \leq q + 3(q-3) = 4q-9.$$

Substituting inequality (2) we get $|ADS| \leq 8|MDS| - 1$.

**Theorem 3.** *The running time of the algorithm is $O(n^2+nm)$.*

*Proof.* Steps 1 and 2 of the algorithm that are needed to compute $OC(1,2n)$ take $O(n^2+nm)$ time. Step 3 takes $O(n)$ time. Steps 4 and 5 take $O(n+m)$ time.

### 2.6 Approximating MTDS

A chord $c \in T$ is said to be *isolated* with respect to a set $U \subseteq T$ if and only if $c$ does not intersect any of the chords in $U - \{c\}$. We assume that $T$ contains no isolated chords with respect to $T$, otherwise $MTDS$ does not have a solution.

A simple algorithm that computes a total dominating set $ATDS$ of $T$ is as follows. Initially, $ATDS$ is empty. Start by adding $ADS$ to $ATDS$. Then, for each chord $c \in ADS$ which is isolated with respect to $ADS$, pick an arbitrary chord $d \in T$ that cuts $c$ and add it to $ATDS$. Clearly, $ATDS$ is a total dominating set of $T$.

**Lemma 3.** *For any chord $c \in ADS - J$, there is a chord in $ADS - \{c\}$ that cuts $c$.*

*Proof.* $ADS$ contains chords of three types:

Type (1) those that belong to $J$,
Type (2) those that correspond to edges in $P$, and
Type (3) those that belong to $\texttt{cutset}(s_i, s_j)$ for some $(s_i, s_j) \in P$.

Let $c$ be an arbitrary chord in $ADS$. If $c$ is of Type (2), that is, there is a chord $d = (s_i, s_j) \in P$ and $c$ is a chord connecting sectors $s_i$ and $s_j$, then $c$ is cut by an element in $\texttt{cutset}(s_i, s_j) \subseteq ADS$. Similarly, if $c$ is of Type (3), that is, $c \in \texttt{cutset}(s_i, s_j)$ for some drawing edge $d = (s_i, s_j) \in P$, then the chord we choose to add to $ADS$ corresponding to $d$, will cut $c$.

**Theorem 4.** $|ATDS| \leq 10|MDS| + 1$

*Proof.* If $c \in ADS - J$, then Lemma 3 tells us that $c$ is not isolated with respect to $ADS$. If $c \in J$, then $c$ may be isolated with respect to $ADS$, and in this case we have added to $ATDS$ a chord in $T$ that cuts $c$. Thus we have

$$|ATDS| \leq |ADS| + |J| \tag{6}$$

Substituting inequalities (4) and (2) in (6), we get $|ATDS| \leq |ADS| + 2|MDS| + 2$. Using the result of Theorem 2 we get $|ATDS| \leq 10|MDS| + 1$.

The result of the following theorem is immediate from Theorem 3.

**Theorem 5.** *The running time of the algorithm is* $O(n^2 + nm)$.

### 2.7 Approximating MCDS

We assume that $T$ is connected, otherwise MCDS does not have a solution. To compute a connected dominating set of $T$, we start with the dominating set $ADS$ and add to it a small set of chords that connects the elements of $ADS$.

One straightforward approach to connect the elements of $ADS$ using a smallest set of chords is to find a minimum Steiner tree for $ADS$ in $T$. Then the set of chords in $T$ corresponding to vertices of the minimum Steiner tree is a connected dominating set of $T$. Let $MCDS$ be a minimum connected dominating set of $T$. It is clear that $MCDS \cup ADS$ contains a Steiner tree for $ADS$ and therefore the number of vertices in a minimum Steiner tree for $ADS$ is no more than $|ADS| + |MCDS|$. This along with the result of Theorem 2 imply that the size of the connected dominating set found using this approach is within a factor of 9 of optimal. Johnson [2] mentions that the minimum Steiner tree problem has a polynomial time solution for circle graphs. However, this result does not appear explicitly in the literature.

In the following we present a second approach to connect the elements of $ADS$, which gives us a connected dominating set of size within a factor of 14 of optimal. For a given set of chords $S$, let $CC(S)$ denote the set of connected components in $S$.

**Lemma 4.** $|CC(ADS)| \leq 5|MDS| + 0.5$

*Proof.* Inequality (3) tells us that the number of chords in $ADS - J$ is at most $3|P|$. This along with Lemma 3 imply that the number of connected components in $ADS - J$ is at most $1.5|P|$. Hence the total number of connected components in $ADS$ is $|CC(ADS)| \leq |J| + 1.5|P|$. Substituting inequalities (4) and (5) we get $|CC(ADS)| \leq 2.5q - 4.5$. Substituting inequality (2) we get $|CC(ADS)| \leq 5|MDS| + 0.5$.

For any connected component $CC_i$, define the *neighborhood* of $CC_i$, denoted $\texttt{nbd}(CC_i)$, as the set of chords in $T$ adjacent to at least one chord in $CC_i$. For any pair of connected components $CC_i$ and $CC_j$, define $\texttt{connect}(CC_i, CC_j)$ as the smallest set of chords $U \subset T$ such that $CC_i \cup CC_j \cup U$ is connected.

**Lemma 5.** *Let $D$ be a dominating set of $T$. For any connected component $CC_i \in CC(D)$, there is a connected component $CC_j \in CC(D)$, $j \neq i$, such that $|\texttt{connect}(CC_i, CC_j)| \leq 2$.*

*Proof.* Let $i$ be fixed. Since we are assuming that the input graph is connected, there exists a chord $c \in \texttt{nbd}(CC_i)$ adjacent to some chord $d \in T - CC_i - \texttt{nbd}(CC_i)$. If $d \in CC_j$ for some $j \neq i$, then $CC_i \cup CC_j \cup \{c\}$ is connected and the lemma is true. Otherwise let $e \in D$ be a chord that cuts $d$ (such a chord always exists since $D$ is a dominating set) and let $CC_j$ be the connected component in $CC(D)$ that contains $e$. Since $c \in \texttt{nbd}(CC_i)$, $d \in \texttt{nbd}(CC_j)$ and $c$ is adjacent to $d$, we have that $CC_i \cup CC_j \cup \{c, d\}$ is connected.

We now use the above Lemma to compute a connected dominating set $ACDS$ of $T$ as follows.

Step 1. initialize $ACDS$ to $ADS$ and compute $CC(ACDS)$
**while** there exist $CC_i$ and $CC_j$ in $CC(ACDS)$
with $|\texttt{connect}(CC_i, CC_j)| = 1$ **do**
add $\texttt{connect}(CC_i, CC_j)$ to $ACDS$
Step 2. **while** $|CC(ACDS)| > 1$ **do**
pick an arbitrary connected component $CC_i$ in $CC(ACDS)$
find $CC_j \in CC(ACDS)$ such that $|\texttt{connect}(CC_i, CC_j)| = 2$
add $\texttt{connect}(CC_i, CC_j)$ to $ACDS$

Clearly $ACDS$ is a connected dominating set.

**Theorem 6.** $|ACDS| \leq 14|MCDS| - 5$.

*Proof.* Let $CC_1, CC_2, \cdots CC_p$ be the connected components in $CC(ACDS)$ after Step 1 above. At this point in the algorithm, for each pair of connected components $CC_i$ and $CC_j$, $1 \leq i \neq j \leq p$, we have that $\texttt{nbd}(CC_i) \cap \texttt{nbd}(CC_j) = \phi$. Thus the sets of chords $CC_i \cup \texttt{nbd}(CC_i)$ for all $i$, $1 \leq i \leq p$, are pairwise disjoint. This implies that $MCDS$ contains at least one chord in each of the sets $CC_i \cup \texttt{nbd}(CC_i)$, $i = 1 \cdots p$. If, for some $i$, $MCDS$ does not contain a chord in $CC_i \cup \texttt{nbd}(CC_i)$, then $MCDS$ does not cut any of the chords in $CC_i \cup \texttt{nbd}(CC_i)$, contradicting the fact that $MCDS$ is a dominating set. Hence we have that $p \leq |MCDS|$ and therefore at least $|CC(ADS)| - |MCDS| - 1$ pairs of connected components have been processed in Step 1 of the algorithm. Thus the total number of chords introduced in Steps 1 and 2 above is $|ACDS - ADS| \leq (|CC(ADS)| - |MCDS| - 1) + 2(|MCDS| - 1)$. Using the result of Lemma 4 we get $|ACDS - ADS| \leq 5|MDS| + |MCDS| - 2.5$. It is obvious that $|MDS| \leq |MCDS|$. Using this and the result of Theorem 2 we get $|ACDS| \leq 14|MCDS| - 3.5$.

**Theorem 7.** *The running time of the algorithm is $O(n^2 + nm)$.*

*Proof.* We assume that the algorithm dynamically maintains an $n \times n$ boolean matrix called `nbd`, whose rows correspond to chords in $T - ACDS$ and columns correspond to connected components in $CC(ACDS)$. For each chord $c \in T - ACDS$ and each connected component $K \in CC(ACDS)$, $\texttt{nbd}[c, K] = 1$ if and only if $c \in \texttt{nbd}(K)$. For each chord $c$, the last entry in row $c$ stores the number of connected components in $CC(ACDS)$ whose neighborhood contain $c$ (i.e, the number of entries in row $c$ equal to 1). A zero value for this entry indicates that the corresponding chord belongs to a connected component in $CC(ACDS)$. This allows us to perform the test for the existence of two connected components $CC_i$ and $CC_j$ in $CC(ACDS)$ such that $|\texttt{connect}(CC_i, CC_j)| = 1$, in Step 1 of the algorithm, in time $O(n)$. The computation of $CC(ACDS)$ in Step 1 takes time $O(n+m)$ (using a depth-first search technique, for instance). The initialization of `nbd` takes time $O(\Sigma_{c \in T-ADS} degree(c)) = O(m)$. It is also not hard to see that the matrix `nbd` can be updated in time $O(\Sigma_{c \in \{\texttt{nbd}(CC_i), \texttt{nbd}(CC_j)\}} degree(c)) = O(m)$, each time $\texttt{connect}(CC_i, CC_j)$ is added to $ACDS$. The search for $CC_j$ and the computation of $\texttt{connect}(CC_i, CC_j)$ in Step 2 of the algorithm takes time $O(\Sigma_{c \in \texttt{nbd}(CC_i)} degree(c)) = O(m)$. Since the total number of connected components in $CC(ADS)$ we start with is $O(n)$, Steps 1 and 2 above take time $O(nm)$. This along with Theorem 3 imply that the running time of the algorithm is $O(n^2 + mn)$.

## 3 Conclusions

The topic of approximation algorithms for dominating set problems on restricted classes of graphs has not received much attention from researchers as yet. Our results is this paper — MDS, MTDS and MCDS can be approximated within a constant factor of optimal in polynomial time — along with the result on the hardness of approximating minimum independent dominating set on circle graphs (Damian-Iordache and Pemmaraju, *in this proceedings*), are probably the first of this kind. Improving on the factors of approximation of the algorithms presented in this paper remains an open interesting problem.

## References

1. E. S. Elmallah, L. K. Stewart, and J. Culberson. Polynomial algorithms on -polygon graphs. In *Proceedings of the 21st Southeastern International Conference on Combinatorics, Graph Theory, and Computing*, Boca Raton (Florida), 1990.
2. D. S. Johnson. The NP-completeness column: an ongoing guide. *Journal of Algorithms*, 6:434–451, 1985.
3. J. Mark Keil. The complexity of domination problems in circle graphs. *Discrete Applied Mathematics*, 42:51–63, 1991.
4. M.R.Garey and D.S.Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. W. H. Freeman and Company, New York, 1979.

# Ordered Binary Decision Diagrams as Knowledge-Bases

Takashi Horiyama[1] and Toshihide Ibaraki[2]

[1] Graduate School of Information Science, Nara Institute of Science and Technology, Nara, 630-0101 Japan. horiyama@is.aist-nara.ac.jp
[2] Department of Applied Mathematics and Physics, Kyoto University, Kyoto, 606-8501 Japan. ibaraki@kuamp.kyoto-u.ac.jp

**Abstract.** We propose to make use of ordered binary decision diagrams (OBDDs) as a means of realizing knowledge-bases. We show that the OBDD-based representation is more efficient and suitable in some cases, compared with the traditional CNF-based and/or model-based representations in the sense of space requirement. We then consider two recognition problems of OBDDs, and present polynomial time algorithms for testing whether a given OBDD represents a unate Boolean function, and whether it represents a Horn function.

## 1 Introduction

Logical formulae are one of the traditional means of representing knowledge in AI [12]. However, it is known that deduction from a set of propositional clauses is co-NP-complete and abduction is NP-complete [13]. Recently, an alternative way of representing knowledge, i.e., by a subset of its models, which are called characteristic models, has been proposed (see e.g., [7,8,9,10]). Deduction from a knowledge-base in this model-based approach can be performed in linear time, and abduction is also performed in polynomial time [7].

In this paper, we propose yet another method of knowledge representation, i.e., the use of ordered binary decision diagrams (OBDDs) [1,2,3]. An OBDD is a directed acyclic graph representing a Boolean function, and can be considered as a variant of decision trees. By restricting the order of variable appearances and by sharing isomorphic subgraphs, OBDDs have the following useful properties: 1) When a variable ordering is given, an OBDD has a reduced canonical form for each Boolean function. 2) Many Boolean functions appearing in practice can be compactly represented. 3) There are efficient algorithms for many Boolean operations on OBDDs. 4) When an OBDD is given, satisfiability and tautology of the corresponding function can be easily checked in constant time. As a result of these properties, OBDDs are widely used for various applications, especially in computer-aided design and verification of digital systems (see e.g., [2,4,14]). The manipulation of knowledge-bases by OBDDs (e.g. deduction and abduction) was first discussed by Madre and Coudert [11].

We first compare the above three representations, i.e., formula-based, model-based, and OBDD-based, on the basis of their sizes. This will give a foundation

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 83–92, 1999.
© Springer-Verlag Berlin Heidelberg 1999

for analyzing and comparing time and space complexities of various operations. Comparisons between these representations have been attempted in different communities. In AI community, it was shown that formula-based and model-based representations are incomparable with respect to space requirement [7]. Namely, each of them sometimes allows exponentially smaller sizes than the other, depending on the functions. In theoretical science and VLSI design communities, formula-based and OBDD-based representations are shown to be incomparable [6]. However, the three representations have never been compared on the same ground. We show that, in some cases, OBDD-based representation requires exponentially smaller space than the other two, while there are also cases in which each of the other two requires exponentially smaller space than that of OBDDs. We also point out an unfortunate result that there exists a Horn function which requires exponential size for any of the three representations.

OBDDs are known to be efficient for knowledge-base operations such as deduction and abduction [11]. We investigate two fundamental recognition problems of OBDDs, that is, testing whether a given OBDD represents a unate Boolean function, and testing whether it represents a Horn function. We often encounter these recognition problems, since a knowledge-base representing some real phenomenon is sometimes required to be unate or Horn, from the hypothesis posed on the phenomenon and/or from the investigation of the mechanism causing the phenomenon. For example, if the knowledge-base represents the data set of test results on various physical measurements (e.g., body temperature, blood pressure, number of pulses and so on), it is often the case that the diagnosis of a certain disease is monotonically depending on each test result (we allow changing the polarities of variables if necessary). Also in artificial intelligence, it is common to consider Horn knowledge-bases as they can be processed efficiently in many respects (for example, deduction from a set of Horn clauses can be done in linear time [5]). We show that these recognition problems for OBDDs can be solved in polynomial time for both the unate and Horn cases.

The rest of this paper is organized as follows. The next section gives fundamental definitions and concepts. We compare the three representations in Section 3, and consider the problems of recognizing unate and Horn OBDDs in Sections 4 and 5, respectively.

## 2 Preliminaries

### 2.1 Notations and Fundamental Concepts

We consider a Boolean function $f : \{0,1\}^n \to \{0,1\}$. 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 of $f$, and the *theory* $\Sigma(f)$ *representing* $f$ is the set of all models of $f$. 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 $i = 1, 2, \ldots, n$, where $0 < 1$. Given a subset $E \subseteq \{1, 2, \ldots, n\}$, $\chi^E$ denotes the characteristic vector of $E$; the $i$-th coordinate $\chi^E_i$ equals 1 if $i \in E$ and 0 if $i \notin E$.

Let $x_1, x_2, \ldots, x_n$ be the $n$ variables of $f$. Negation of a variable $x_i$ is denoted by $\overline{x}_i$. Any Boolean function can be represented by some CNF (conjunctive normal form), which may not be unique. We sometimes do not make

a distinction among a function $f$, its theory $\Sigma(f)$, and a CNF $\varphi$ that represents $f$, unless confusion arises. 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, $f|_{x_i=a_i}(x_1,\ldots,x_n) = f(x_1,\ldots,x_{i-1},a_i,x_{i+1},\ldots,x_n)$ holds. Restriction may be applied to many variables. We also define $f \leq g$ (resp., $f < g$) by $\Sigma(f) \subseteq \Sigma(g)$ (resp., $\Sigma(f) \subset \Sigma(g)$).

For an assignment $p \in \{0,1\}^n$, we define $a \leq_p b$ if $(a \oplus_{\rm bit} p) \leq (b \oplus_{\rm bit} p)$ holds, where $\oplus_{\rm bit}$ denotes the bitwise exclusive-or operation. A Boolean function $f$ is *unate* with polarity $p$ if $f(a) \leq f(b)$ holds for all assignments $a$ and $b$ such that $a \leq_p b$. A theory $\Sigma$ is *unate* if $\Sigma$ represents a unate function. A clause is *unate* with polarity $p$ if $p_i = 0$ for all positive literals $x_i$ and $p_i = 1$ for all negative literals $\overline{x}_i$ in the clause. A CNF is *unate* with polarity $p$ if it contains only unate clauses with polarity $p$. It is known that a theory $\Sigma$ is unate if and only if $\Sigma$ can be represented by some unate CNF. A unate function is *positive* (resp., *negative*) if its polarity is $(00\cdots0)$ (resp., $(11\cdots1)$).

A theory $\Sigma$ is *Horn* if $\Sigma$ is closed under operation $\wedge_{\rm bit}$, where $a \wedge_{\rm bit} b$ is bitwise AND of models $a$ and $b$. For example, if $a = (0011)$ and $b = (0101)$, then $a \wedge_{\rm bit} b = (0001)$. The closure of a theory $\Sigma$ with respect to $\wedge_{\rm bit}$ is denoted by $Cl_{\wedge_{\rm bit}}(\Sigma)$. We also use the operation $\wedge_{\rm bit}$ as a set operation; $\Sigma(f) \wedge_{\rm bit} \Sigma(g) = \{a \mid a = b \wedge_{\rm bit} c$ holds for some $b \in \Sigma(f)$ and $c \in \Sigma(g)\}$. We often denotes $\Sigma(f) \wedge_{\rm bit} \Sigma(g)$ by $f \wedge_{\rm bit} g$, for convenience. Note that the two functions $f \wedge g$ and $f \wedge_{\rm bit} g$ are different.

A Boolean function $f$ is *Horn* if $\Sigma(f)$ is Horn; equivalently if $f \wedge_{\rm bit} f = f$ holds (as sets of models). 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. It is known that a theory $\Sigma$ is Horn if and only if $\Sigma$ can be represented by some Horn CNF.

For any Horn theory $\Sigma$, a model $a \in \Sigma$ is called *characteristic* if it cannot be produced by bitwise AND of other models in $\Sigma$; $a \notin Cl_{\wedge_{\rm bit}}(\Sigma - \{a\})$. The set of all characteristic models of a Horn theory $\Sigma$, which we call the *characteristic set of* $\Sigma$, is denoted by $Char(\Sigma)$. Note that every Horn theory $\Sigma$ has a unique characteristic set $Char(\Sigma)$, which satisfies $Cl_{\wedge_{\rm bit}}(Char(\Sigma)) = \Sigma$.

### 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 $x_1, x_2, \ldots, x_n$. Let $var(v)$ 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 $X = \{x_1, x_2, \ldots, x_n\}$ denote the set of $n$ variables. A *variable ordering* is a total ordering $(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$, associated with each OBDD, where $\pi$ is a permutation $\{1,2,\ldots,n\} \to \{1,2,\ldots,n\}$. The *level*[1] of a node $v$, denoted by $level(v)$, is defined by its label; if node $v$ has label $x_{\pi(i)}$,

[1] This definition of level may be different from its common use.

$level(v)$ is defined to be $n - i + 1$. 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 edges and nodes reachable from $v$. If node $v$ is a constant node, $f_v$ equals to its label. If node $v$ is a variable node, $f_v$ is defined as $\overline{var(v)}\, f_{0\text{-}succ(v)} \vee var(v)\, f_{1\text{-}succ(v)}$ by Shannon's expansion, where 0-$succ(v)$ and 1-$succ(v)$, 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 $a \in \{0,1\}^n$, the value of $f(a)$ is determined by following the corresponding path from the root node to a constant node in the following manner: at a variable node $v$, one of the outgoing edges is selected according to the assignment $a_{var(v)}$ to the variable $var(v)$. The value of the function is the label of the final constant node.

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 equivalent nodes and no redundant nodes is *reduced*. The *size* of an OBDD 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 minimum sizes of OBDDs representing a given Boolean function depends on the variable orderings [3]. In the following, we assume that all OBDDs are reduced.

## 3 Three Approaches for Knowledge-Base Representation

In this section, we compare three knowledge-base representations: CNF-based, model-based, and OBDD-based. We show that OBDD-based representation is incomparable to the other two with respect to space requirement.

**Lemma 3.1.** *There exists a negative theory on $n$ variables, for which OBDD and CNF both require size $O(n)$, while its characteristic set requires size $\Omega(2^{n/2})$.*

*Proof.* Consider a function $f_A = \bigwedge_{i=1}^{m} (\overline{x}_{2i-1} \vee \overline{x}_{2i})$, where $n = 2m$. □

**Lemma 3.2.** *There exists a negative theory on $n$ variables, for which OBDD requires size $O(n)$ and the characteristic set requires size $O(n^2)$, while CNF requires size $\Omega(2^{n/2})$.*

*Proof.* Consider a function $f_B = \bigvee_{i=1}^{m} (\overline{x}_{2i-1} \wedge \overline{x}_{2i})$, where $n = 2m$. □

**Theorem 3.1.** *There exists a negative theory on $n$ variables, for which OBDD requires size $O(n)$, while both of the characteristic set and CNF require sizes $\Omega(2^{n/4})$.*

*Proof.* Consider a function $f_C = (\bigwedge_{i=1}^{m} (\overline{x}_{2i-1} \vee \overline{x}_{2i})) \wedge (\bigvee_{i=m+1}^{2m} (\overline{x}_{2i-1} \wedge \overline{x}_{2i}))$, where $n = 4m$. This theorem can be obtained from Lemmas 3.1 and 3.2. □

**Theorem 3.2.** *There exists a Horn theory on $n$ variables, for which both of the CNF and the characteristic set require sizes $O(n)$, while the size of the smallest OBDD representation is $\Omega(2^{\sqrt{n}/\sqrt{2}})$.*

*Proof.* Consider a function $f_D = (\bigwedge_{i=1}^{m+1}(x_{i,m+1} \vee \bigvee_{j=1}^{m} \overline{x}_{i,j})) \wedge (\bigwedge_{j=1}^{m+1}(x_{m+1,j} \vee \bigvee_{i=1}^{m} \overline{x}_{i,j}))$ on $n$ variables $x_{i,j}$, $1 \leq i, j \leq m+1$, where $n = (m+1)^2$. □

The above results show that none of the three representations always dominate the other two. Therefore, OBDDs can find a place in knowledge-bases as they can represent some theories more efficiently than others.

Unfortunately, by combining Theorems 3.1 and 3.2, we can construct the following function, which is exponential for all representations.

**Corollary 3.1.** *There exists a Horn function on $n$ variables, for which both of the characteristic set and CNF require sizes $\Omega(2^{n/8})$ and the size of the smallest OBDD representation is $\Omega(2^{\sqrt{n}/2})$.*

*Proof.* Consider the conjunction $f_C \wedge f_D$, where $f_C$ (resp., $f_D$) is defined in the proof of Theorem 3.1 (resp., Theorem 3.2). Note that $f_C$ and $f_D$ both have $n/2$ variables, but share none of the variables. □

## 4 Checking Unateness of OBDD

In this section, we discuss the problem of checking whether a given OBDD is unate. We assume, without loss of generality, that the variable ordering is always $(x_n, x_{n-1}, \ldots, x_1)$. The following well-known property indicates that this problem can be solved in polynomial time.

*Property 4.1.* Let $f$ be a Boolean function on $n$ variables $x_1, x_2, \ldots, x_n$. Then, $f$ is unate if and only if $f|_{x_i=0} \leq f|_{x_i=1}$ or $f|_{x_i=0} \geq f|_{x_i=1}$ holds for all $i$'s. □

An OBDD representing $f|_{x_i=0}$ (resp., $f|_{x_i=1}$) can be obtained in $O(|f|)$ time from the OBDD representing $f$, where $|f|$ denotes its size [2]. The size does not increase by a restriction $f|_{x_i=0}$ or $f|_{x_i=1}$. Since property $g \leq h$ can be checked in $O(|g| \cdot |h|)$ time [3], the unateness of $f$ can be checked in $O(n|f|^2)$ time by checking conditions $f|_{x_i=0} \leq f|_{x_i=1}$ and $f|_{x_i=0} \geq f|_{x_i=1}$ for all $i = 1, 2, \ldots, n$.

The following well-known property is useful to reduce the computation time.

*Property 4.2.* Let $f$ be a Boolean function on $n$ variables $x_1, x_2, \ldots, x_n$. Then, $f$ is unate with polarity $p = (p_1, p_2, \ldots, p_n)$ if and only if both $f|_{x_n=0}$ and $f|_{x_n=1}$ are unate with same polarity $(p_1, p_2, \ldots, p_{n-1})$, $p_n = 0$ implies $f|_{x_n=0} \leq f|_{x_n=1}$ and $p_n = 1$ implies $f|_{x_n=0} \geq f|_{x_n=1}$. □

The unateness of functions $f|_{x_n=0}$ and $f|_{x_n=1}$ can be checked by applying Property 4.2 recursively, but we also have to check that $f|_{x_n=0}$ and $f|_{x_n=1}$ have the same polarity. Our algorithm is similar to the implementation of OBDD-manipulation-systems by Bryant [3], in the sense that we cache all intermediate computational results to avoid duplicate computation. In Bryant's idea, different

**Algorithm CHECK-UNATE**

**Input:** An OBDD representing $f$ with a variable ordering $(x_n, \ldots, x_1)$.
**Output:** "yes" and its polarity if $f$ is unate; otherwise, "no".

**Step 1 (initialize).** Set $\ell := 1$; $p[\,i\,] := *$ for all $i = 1, 2, \ldots, n$;

$$imp[u, v] := \begin{cases} \text{NO} & \text{if } (u, v) = (1, 0); \\ \text{YES} & \text{if } (u, v) = (0, 0), (0, 1), (1, 1); \\ * & \text{otherwise.} \end{cases}$$

**Step 2 (check unateness in level $\ell$ and compute $p[\,\ell\,]$).** For each node $v$ in level $\ell$ (i.e., labeled with $x_\ell$), apply Steps 2-1 and 2-2.

**Step 2-1.** Set $pol := 0$ if $imp[0\text{-}succ(v), 1\text{-}succ(v)] = \text{YES}$ holds; set $pol := 1$ if $imp[1\text{-}succ(v), 0\text{-}succ(v)] = \text{YES}$ holds; otherwise, output "no" and halt.
**Step 2-2.** If $p[\,\ell\,] = *$, then set $p[\,\ell\,] := pol$. If $p[\,\ell\,] \neq *$ and $p[\,\ell\,] \neq pol$ hold, then output "no" and halt.

**Step 3 (compute $imp$ in level $\ell$).** For each pair of nodes $u$ and $v$ such that $level(u) \leq \ell$ and $level(v) \leq \ell$, and at least one of $level(u)$ and $level(v)$ is equal to $\ell$, set $imp[u, v] := \text{YES}$ if both $imp[0\text{-}succ'(u), 0\text{-}succ'(v)]$ and $imp[1\text{-}succ'(u), 1\text{-}succ'(v)]$ are YES; otherwise, set $imp[u, v] := \text{NO}$.

**Step 4 (iterate).** If $\ell = n$, where $n$ is the level of the root node, then output "yes" and polarity $p = (p[1], p[2], \ldots, p[n])$, and halt. Otherwise set $\ell := \ell + 1$ and return to Step 2.

**Fig. 1.** An algorithm to check unateness of an OBDD.

computational results may be stored to the same memory in order to handle different operations, and hence the same computation may be repeated more than once. However, as our algorithm only aims to check the unateness, it can avoid such cache conflict by explicitly preparing memory for each result. This is a key to reduce the computation time.

We check the unateness of $f$ in the bottom-up manner by checking unateness of all nodes corresponding to intermediate results. Note that the property $f|_{x_n=0} \leq f|_{x_n=1}$ (resp., $f|_{x_n=0} \geq f|_{x_n=1}$) can be also checked in the bottom-up manner, since $g \leq h$ holds if and only if $g|_{x_i=0} \leq h|_{x_i=0}$ and $g|_{x_i=1} \leq h|_{x_i=1}$ hold for some $i$.

Algorithm CHECK-UNATE in Fig. 1 checks the unateness and the polarity of a given OBDD in the manner as described above. We use an array $p[\,\ell\,]$ to denote the polarity of $f$ with respect to $x_\ell$ in level $\ell$; each element stores 0, 1 or $*$ (not checked yet). We also use a two-dimensional array $imp[u, v]$ to denote whether $f_u \leq f_v$ holds or not; each element stores YES, NO or $*$. In Step 2, the unateness with the unique polarity is checked for the nodes in level $\ell$. More precisely, the unateness for them is checked in Step 2-1, and the uniqueness of their polarities is checked in Step 2-2. In Step 3, $imp[u, v]$ is computed for the nodes in level $\ell$. Namely, $f_u$ and $f_v$ are compared and the result is set to $imp[u, v]$. The comparison is performed easily, since the comparisons between $f_u|_{x_\ell=a_\ell}$ and $f_v|_{x_\ell=a_\ell}$ for

both $a_\ell = 0$ and 1 have already been completed. In Algorithm CHECK-UNATE, 0-$succ'(v)$ (resp., 1-$succ'(v)$) denotes 0-$succ(v)$ (resp., 1-$succ(v)$) if $level(v) = \ell$, but denotes $v$ itself if $level(v) < \ell$. This is because $f_v|_{x_\ell=0} = f_{0\text{-}succ(v)}$ and $f_v|_{x_\ell=1} = f_{1\text{-}succ(v)}$ hold if $level(v) = \ell$, and $f_v|_{x_\ell=0} = f_v|_{x_\ell=1} = f_v$ holds if $level(v) < \ell$. After Step 3 is done for some $\ell$, we know $imp[u, v]$ for all pairs of nodes $u$ and $v$ such that $level(u) \leq \ell$ and $level(v) \leq \ell$. We store all the results, although some of them may not be needed.

Next, we consider the computation time of this algorithm. In Step 2, checking unateness for each node is performed in a constant time from the data computed in the previous iteration. The unateness is checked for all nodes. In Step 3, the comparison between $f_u$ and $f_v$ for each pair of nodes $u$ and $v$ is also performed in a constant time. The number of pairs compared in Step 3 during the entire computation is $O\left(\binom{|f|}{2}\right) = O(|f|^2)$, and this requires $O(|f|^2)$ time.

**Theorem 4.1.** *Given an OBDD representing a Boolean function $f$ of size $|f|$, checking whether $f$ is unate can be done in $O(|f|^2)$ time.* □

If we start Algorithm CHECK-UNATE with initial condition $p[\,i\,] := 0$ (resp., $p[\,i\,] := 1$) for all $i$'s, we can check the positivity (resp. negativity) of $f$.

**Corollary 4.1.** *Given an OBDD representing a Boolean function $f$ of size $|f|$, checking whether $f$ is positive (resp., negative) can be done in $O(|f|^2)$ time.* □

## 5 Checking Horness of OBDD

### 5.1 Conditions for Horness

In this section, we discuss the problem of checking whether a given OBDD is Horn. Denoting $f|_{x_n=0}$ and $f|_{x_n=1}$ by $f_0$ and $f_1$ for simplicity, $f$ is given by $f = \overline{x}_n f_0 \vee x_n f_1$, where $f_0$ and $f_1$ are Boolean functions on $n-1$ variables $x_1, x_2, \ldots, x_{n-1}$. By definition, we can determine whether $f$ is Horn by checking the condition $f \wedge_{\rm bit} f = f$. For this, we may first construct an OBDD of $f \wedge_{\rm bit} f$, and then check the equivalence between $f \wedge_{\rm bit} f$ and $f$. However, the following theorem says that this approach may require exponential time and hence is intractable in general.

**Theorem 5.1.** *There exists a Boolean function $f$ on $n$ variables, for which OBDD requires size $O(n^2)$, while the OBDD representing $f \wedge_{\rm bit} f$ requires $\Omega(2^{n/4})$ for the same variable ordering.*

*Proof.* Consider a function $f_E = (\bigvee_{i=1}^{m}((\overline{x}_i \wedge \overline{x}_{i+m}) \wedge x_{i+3m} \wedge (\bigwedge_{j\in\{1,\ldots,2m\}-\{i+m\}} \overline{x}_{j+2m}))) \vee (\bigvee_{i=1}^{m}((\overline{x}_i \wedge \overline{x}_{i+m}) \wedge x_{i+2m} \wedge (\bigwedge_{j\in\{1,\ldots,2m\}-\{i\}} \overline{x}_{j+2m})))$, where $n = 4m$. □

Our main result however shows that $f \wedge_{\rm bit} f = f$ can be checked in polynomial time without explicitly constructing the OBDD of $f \wedge_{\rm bit} f$. For this goal, the following lemmas tell a key property that the problem can be divided into two subproblems, and hence can be solved by a divide-and-conquer approach.

**Algorithm CHECK-HORN**

**Input:** An OBDD representing $f$ with a variable ordering $(x_n, \ldots, x_1)$.
**Output:** "yes" if $f$ is Horn; otherwise, "no".

**Step 1 (initialize).** Set $\ell := 1$; $horn[v] := \begin{cases} \text{YES} & \text{if } v \in \{0,1\}; \\ * & \text{otherwise;} \end{cases}$

$bit\text{-}imp[u,v,w] := \begin{cases} \text{NO} & \text{if } (u,v,w) = (1,1,0); \\ \text{YES} & \text{if } u,v,w \in \{0,1\} \text{ and } (u,v,w) \neq (1,1,0); \\ * & \text{otherwise.} \end{cases}$

**Step 2 (check Horness in level $\ell$).** For each node $v$ in level $\ell$ (i.e., labeled with $x_\ell$), set $horn[v] :=$ YES if all of $horn[0\text{-}succ(v)]$, $horn[1\text{-}succ(v)]$ and $bit\text{-}imp[0\text{-}succ(v), 1\text{-}succ(v), 0\text{-}succ(v)]$ are YES; otherwise, output "no" and halt.

**Step 3 (compute *bit-imp* in level $\ell$).** For each triple $(u,v,w)$ of nodes such that $level(u) \leq \ell$, $level(v) \leq \ell$ and $level(w) \leq \ell$, and at least one of $level(u)$, $level(v)$ and $level(w)$ is equal to $\ell$, check whether $f_u \wedge_{\text{bit}} f_v \leq f_w$ holds according to Fig. 3. Set the result YES or NO to $bit\text{-}imp[u,v,w]$.

**Step 4 (iterate).** If $\ell = n$ then output "yes" and halt. Otherwise set $\ell := \ell + 1$ and return to Step 2.

**Fig. 2.** An algorithm to check Horness of an OBDD.

**Lemma 5.1.** *Let $f$ be a Boolean function on $n$ variables $x_1, x_2, \ldots, x_n$, which is expanded as $f = \overline{x}_n f_0 \vee x_n f_1$. Then, $f$ is Horn if and only if both $f_0$ and $f_1$ are Horn and $f_0 \wedge_{\text{bit}} f_1 \leq f_0$ holds.*

The Horness of $f_0$ and $f_1$ can be also checked by applying Lemma 5.1 recursively. The following lemma says that the condition $f_0 \wedge_{\text{bit}} f_1 \leq f_0$ can be also checked recursively. Note that the condition of type $f \wedge_{\text{bit}} g \leq f$ in Lemma 5.1 requires to check the condition of type $f_1 \wedge_{\text{bit}} g_0 \leq f_0$ (i.e., checking of type $f \wedge_{\text{bit}} g \leq h$ for three functions $f$, $g$ and $h$).

**Lemma 5.2.** *Let $f$, $g$ and $h$ be Boolean functions on $n$ variables, which are expanded as $f = \overline{x}_n f_0 \vee x_n f_1$, $g = \overline{x}_n g_0 \vee x_n g_1$ and $h = \overline{x}_n h_0 \vee x_n h_1$, respectively. Then, property $f \wedge_{\text{bit}} g \leq h$ holds if and only if $f_0 \wedge_{\text{bit}} g_0 \leq h_0$, $f_0 \wedge_{\text{bit}} g_1 \leq h_0$, $f_1 \wedge_{\text{bit}} g_0 \leq h_0$ and $f_1 \wedge_{\text{bit}} g_1 \leq h_1$ hold.* □

### 5.2 Algorithm to Check Horness

Algorithm CHECK-HORN in Fig. 2 checks the Horness of a given OBDD in the bottom-up manner by applying Lemmas 5.1 and 5.2 recursively. The bottom-up and caching techniques used there are similar to those of CHECK-UNATE. However, we emphasize here that, in the case of unateness, the naive algorithm to check the condition of Property 4.1 was already polynomial time, while the naive algorithm checking $f \wedge_{\text{bit}} f = f$ would require exponential time by Theorem 5.1. This CHECK-HORN is a first polynomial time algorithm, which is made possible by using both Lemmas 5.1 and 5.2.

YES if all of *bit-imp*[1-*succ*′(*u*), 1-*succ*′(*v*), 1-*succ*′(*w*)], *bit-imp* [0-*succ*′(*u*), 0-*succ*′(*v*), 0-*succ*′(*w*)], *bit-imp*[0-*succ*′(*u*), 1-*succ*′(*v*), 0-*succ*′(*w*)] and *bit-imp*[1-*succ*′(*u*), 0-*succ*′(*v*), 0-*succ*′(*w*)] are YES.

NO otherwise.

**Fig. 3.** Checking *bit-imp*$[u, v, w]$ (i.e., $f_u \wedge_{\rm bit} f_v \leq f_w$) for a triple of nodes $(u, v, w)$ in Step 3.

In Algorithm CHECK-HORN, we use an array *horn*[*v*] to denote whether node $v$ represents a Horn function or not, and a three-dimensional array *bit-imp*[*u*, *v*, *w*] to denote whether $f_u \wedge_{\rm bit} f_v \leq f_w$ holds or not; each element of the arrays stores YES, NO or $*$ (not checked yet). *horn*$[v]$ = YES implies that $f_v$ is Horn even if $f_v$ is treated as a Boolean function on more than *level*$(v)$ variables. (Recall that OBDD is reduced; all the added variables are redundant.) Similarly, *bit-imp*$[u, v, w]$ = YES implies that $f_u \wedge_{\rm bit} f_v \leq f_w$ holds even if $f_u$, $f_v$ and $f_w$ are treated as Boolean functions on $\ell$ $(\geq l_{max})$ variables, where $l_{max}$ denotes the maximum level of the nodes $u$, $v$ and $w$.

In Step 2 of Algorithm CHECK-HORN, *horn*$[v]$ can be easily computed according to Lemma 5.1. Note that every node $v$ in level $\ell$ satisfies $f_v|_{x_\ell=0} = f_{0\text{-}succ(v)}$ and $f_v|_{x_\ell=1} = f_{1\text{-}succ(v)}$. Also note that *horn*[0-*succ*$(v)$], *horn*[1-*succ*$(v)$] and *bit-imp*[0-*succ*$(v)$, 1-*succ*$(v)$, 0-*succ*$(v)$] have already been computed.

Similarly, *bit-imp*$[u, v, w]$ in Step 3 can be also computed easily by Fig. 3, corresponding to Lemma 5.2. Similar to the case of checking unateness, 0-*succ*′$(v)$ (resp., 1-*succ*′$(v)$) denotes 0-*succ*$(v)$ (resp., 1-*succ*$(v)$) if *level*$(v) = \ell$, but denotes $v$ itself if *level*$(v) < \ell$. After Step 3 is done for some $\ell$, we have the results for all triples $(u, v, w)$ of nodes such that *level*$(u) \leq \ell$, *level*$(v) \leq \ell$ and *level*$(w) \leq \ell$, which include all the information required in the next iteration.

Now, we consider the computation time of Algorithm CHECK-HORN. In Step 2, *horn*$[v]$ for each node $v$ is computed in a constant time. The Horness is checked for all nodes. In Step 3, *bit-imp*$[u, v, w]$ for each triple of nodes $(u, v, w)$ is also computed in a constant time. The number of triples to be checked in Step 3 during the entire computation is $O(|f|^3)$, where $|f|$ is the size of the given OBDD, and this requires $O(|f|^3)$ time. The time for the rest of computation is minor.

**Theorem 5.2.** *Given an OBDD representing a Boolean function $f$ of size $|f|$, checking whether $f$ is Horn can be done in $O(|f|^3)$ time.* □

## 6 Conclusion

In this paper, we considered to use OBDDs to represent knowledge-bases. We have shown that the conventional CNF-based and model-based representations, and the new OBDD representation are mutually incomparable with respect to space requirement. Thus, OBDDs can find their place in knowledge-bases, as they can represent some theories more efficiently than others.

We then considered the problem of recognizing whether a given OBDD represents a unate Boolean function, and whether it represents a Horn function. It turned out that checking unateness can be done in quadratic time of the size of OBDD, while checking Horness can be done in cubic time.

OBDDs are dominatingly used in the field of computer-aided design and verification of digital systems. The reason for this is that many Boolean functions which we encounter in practice can be compactly represented, and that many operations on OBDDs can be efficiently performed. We believe that OBDDs are also useful for manipulating knowledge-bases. Developing efficient algorithms for knowledge-base operations such as deduction and abduction should be addressed in the further work.

## Acknowledgement

The authors would like to thank Professor Endre Boros of Rutgers University for his valuable comments. This research was partially supported by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan.

## References

1. S.B. Akers, "Binary Decision Diagrams," *IEEE Trans. Comput.*, C-27, no.6, pp.509–516, 1978.
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.
3. R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, C-35, no.8, pp.677–691, 1986.
4. O. Coudert, "Doing Two-Level Logic Minimization 100 Times Faster," *Proc. of 6th ACM/SIAM SODA*, pp.112-118, 1995.
5. 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.
6. K. Hayase and H. Imai, "OBDDs of a Monotone Function and of Its Implicants," *Proc. of ISAAC-96*, LNCS 1178, Springer, pp. 136–145, 1996.
7. H.A. Kautz, M.J. Kearns, and B. Selman, "Reasoning with Characteristic Models," *Proc. of AAAI-93*, pp.34–39, 1993.
8. H.A. Kautz, M.J. Kearns, and B. Selman, "Horn Approximations of Empirical Data," *Artificial Intelligence*, 74, pp.129–245, 1995.
9. D. Kavvadias, C. Papadimitriou, and M. Sideri, "On Horn Envelopes and Hypergraph Transversals," *Proc. of ISAAC-93*, LNCS 762, Springer, pp. 399–405, 1993.
10. R. Khardon and D. Roth, "Reasoning with Models," *Artificial Intelligence*, 87, pp.187–213, 1996.
11. J.C. Madre and O. Coudert, "A Logically Complete Reasoning Maintenance System Based on a Logical Constraint Solver," *Proc. of IJCAI-91*, pp.294–299, 1991.
12. J. McCarthy and P.J. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence," *Machine Intelligence 4*, Edinburgh University Press, 1969.
13. B. Selman and H.J. Levesque, "Abductive and Default Reasoning: A Computational Core," *Proc. of AAAI-90*, pp.343–348, 1990.
14. N. Takahashi, N. Ishiura, and S. Yajima, "Fault Simulation for Multiple Faults Using BDD Representation of Fault Sets," *Proc. of ICCAD-91*, pp.550–553, 1991.

# Hard Tasks for Weak Robots: The Role of Common Knowledge in Pattern Formation by Autonomous Mobile Robots*

Paola Flocchini[1], Giuseppe Prencipe[2], Nicola Santoro[3], and Peter Widmayer[4]

[1] School of Information, Technology and Engineering, University of Ottawa, flocchin@site.uottawa.ca
[2] Dipartimento di Informatica - Università di Pisa, prencipe@di.unipi.it
[3] School of Computer Science, Carleton University, santoro@scs.carleton.ca
[4] Theoretische Informatik, ETH Zürich, pw@inf.ethz.ch

**Abstract.** In this paper we aim at an understanding of the fundamental algorithmic limitations on what a set of autonomous mobile robots can or cannot achieve. We study a hard task for a set of weak robots. The task is for the robots in the plane to form any arbitrary pattern that is given in advance. The robots are weak in several aspects. They are anonymous; they cannot explicitly communicate with each other, but only observe the positions of the others; they cannot remember the past; they operate in a very strong form of asynchronicity. We show that the tasks that such a system of robots can perform depend strongly on their common knowledge about their environment, i.e., the readings of their environment sensors.

## 1 Introduction, Definitions, and Overview

### 1.1 Autonomous Mobile Robots

We study the problem of coordinating a set of *autonomous*, mobile robots in the plane. The coordination mechanism must be totally *decentralized*, without any central control. The robots are *anonymous*, in the sense that a robot does not have an identity that it can use in a computation, and all robots execute the exact same algorithm. Each robot has its own, *local view* of the world. This view includes a local Cartesian coordinate system with origin, unit of length, and the *directions* of two coordinate axes, identified as $x$ axis and $y$ axis, together with their *orientations*, identified as the positive sides of the axes. The robots do not have a common understanding of the *handedness (chirality)* of the coordinate

* This work has been performed while the first and third authors have been visiting ETH Zürich and while the second and fourth authors have been visiting Carleton University. We also gratefully acknowledge the partial support of the Swiss National Science Foundation, the Natural Science and Engineering Research Council of Canada and the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche du Québec.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 93–102, 1999.
© Springer-Verlag Berlin Heidelberg 1999

system that allows them to consistently infer the orientation of the $y$ axis once the orientation of the $x$ axis is given; instead, knowing North does not distinguish East from West. The robots observe the environment and move; this is their only means of communication and of expressing a decision that they have taken. The only thing that a robot can do is make a *step*, where a step is a sequence of three actions. First, the robot observes the positions of all other robots with respect to its local coordinate system. Each robot is viewed as a point, and therefore the *observation* returns a set of points to the observing robot. The robot cannot distinguish between its fellow robots; they all look identical. In addition, the robot cannot detect whether there is more than one fellow robot on any of the observed points; we say, it cannot detect *multiplicity*. Second, the robot performs an arbitrary *local computation* according to its algorithm, based only on its *common knowledge* of the world (assumed e.g. to be stored in read-only-memory and to be read off from sensors of the environment) and the observed set of points. Since the robot does not memorize anything about the past, we call it *oblivious*. For simplicity, we assume that the algorithm is *deterministic*, but it will be obvious that all of our results hold for nondeterministic algorithms as well (randomization, however, makes things different). Third, as a result of the computation, the robot either stands still, or it moves (along any curve it likes). The *movement* is confined to some (potentially small) unpredictable, nonzero amount. Hence, the robot can only go towards its goal along a curve, but it cannot know how far it will come in the current step, because it can fall asleep anytime during its movement. While it is on its continuous move, a robot may be seen an arbitray number of times by other robots, even within one of its steps.

The system is totally *asynchronous*, in the sense that there is no common notion of time. Each robot makes steps at unpredictable time instants. The (global) time that passes between two successive steps of the same robot is finite; that is, any desired finite number of steps will have been made by any robot after some finite amount of time. In addition, we do not make any timing assumptions within a step: The time that passes after the robot has observed the positions of all others and before it starts moving is arbitrary, but finite. That is, the actual move of a robot may be based on a situation that lies arbitrarily far in the past, and therefore it may be totally different from the current situation. We feel that this assumption of *asynchronicity within a step* is important in a totally asynchronous environment, since we want to give each robot enough time to perform its local computation.

### 1.2 Pattern Formation

In this paper, we concentrate on the particular coordination problem that requires the robots to form a specific geometric pattern, the *pattern formation* problem. This problem has been investigated quite a bit in the literature, mostly as an initial step that gets the robots together and then lets them proceed in the desired formation (just like a flock of birds or a troupe of soldiers); it is interesting algorithmically, because if the robots can form any pattern, they can agree on their respective roles in a subsequent, coordinated action. We study this

problem for arbitrary geometric patterns, where a pattern is a set of points (given by their Cartesian coordinates) in the plane. The pattern is known initially by all robots in the system. For instance, we might require the robots to place themselves on the circumference of a circle, with equal spacing between any two adjacent robots, just like kids in the kindergarten are sometimes requested to do. We do not prescribe the position of the circle in the world, and we do not prescribe the size of the circle, just because the robots do not have a notion of the world coordinate system's origin or unit of length. The robots are said to *form the pattern*, if the actual positions of the robots coincide with the points of the pattern, where the pattern may be *translated*, *rotated*, *scaled*, and *flipped* into its mirror position in each local coordinate system. Initially, the robots are in arbitrary positions, with the only requirement that no two robots are in the same position, and that of course the number of points prescribed in the pattern and the number of robots are the same. Note that in our algorithms, we do not need to and we will not make use of the possibility of rotating the pattern.

The pattern formation problem for *arbitrary* patterns is quite a general member in the class of problems that are of interest for autonomous, mobile robots. It includes as special cases many coordination problems, such as leader election: We just define the pattern in such a way that the leader is represented uniquely by one point in the pattern. This reflects the general direction of the investigation in this paper: What coordination problems can be solved, and under what conditions? The only means for the robots to coordinate is the observation of the others' positions; therefore, the only means for a robot to send information to some other robot is to move and let the others observe (reminiscent of bees in a bee dance). For oblivious robots, even this sending of information is impossible, since the others will not remember previous positions. Hence, our study is at the extreme end in two ways: The problem is extremely hard, and the robots are extremely weak.

In an attempt to understand the power of common knowledge for the coordination of robots, we study the pattern formation problem under several assumptions. We give a complete characterization of what can and what cannot be achieved. First, we show that for an arbitrary number of robots that know the direction and the orientation of both axes, the pattern formation problem can be solved. Here, knowing the direction of the $x$ axis means that all robots know and use the fact that all the lines identifying their individual $x$ axes are parallel. Similarly, knowing the orientation of an axis means that the positive side of that axis in the coordinate system coincides for all robots. Second, we study the case of the robots knowing one axis direction and orientation. We show that the pattern formation problem can be solved whenever the number of robots is odd, and that it is in general unsolvable when the number of robots is even. Third, we show that the situation is the same, if one axis direction is known, but not the orientation of the axis. Fourth, we show that if no axis direction (and therefore also no orientation) is known, the problem cannot be solved in general. For brevity all proofs are omitted. The reader is referred to [4].

### 1.3 Related Work

The problem of controlling a set of autonomous, mobile robots in a distributed fashion has been studied extensively, but almost exclusively from an engineering and from an artificial intelligence point of view. In a number of remarkable studies (on social interaction leading to group behavior [5], on selfish behavior of cooperative agents in animal societies [6], on primitive animal behavior in pattern formation [1], to pick just a few), algorithmic aspects were somehow implicitly an issue, but clearly not a major concern, let alone the focus, of the study.

We aim at identifying the algorithmic limitations of what autonomous, mobile robots can or cannot do. An investigation with this flavor has been undertaken within the AI community by Durfee [2], who argues in favor of limiting the knowledge that an intelligent agent must possess in order to be able to coordinate its behavior with others. The work of Suzuki and Yamashita [7, 8, 9] is closest to our study (and, with this focus, a rarity in the mobile robots literature); it gives a nice and systematic account on the algorithmics of pattern formation for robots, under several assumptions on the power of the indivdual robot. The models that we use differ from those of [7, 8, 9] in the fact that our robots are as weak as possible in every single aspect of their behavior. The reason is that we want to identify the role of the robots' common knowledge of the world for performing a task. In contrast with [7, 8, 9], we do not assume that on a move, we know ahead of time the limited, but nonzero distance that a robot travels. We do not assume that the distance that a robot may travel in one step is so short that no other robot can see it while it is moving. We do not assume that the robots have a common handedness, called *sense of orientation* in [7, 9].

The most radical deviation from previous models may, however, be our assumption of *asynchronicity within one step*. In contrast, [7, 8, 9] assume the *atomicity of a step*: A robot moves immediately after it has observed the current situation, with all awake robots moving at the same clock tick (some robots may be asleep). This difference influences the power of the system of robots so drastically that in general, algorithms that make use of atomicity within one step do not work in our model; in particular, this is true for the work in [7, 8, 9].

## 2 Knowledge of Both Axis Directions and Orientations

For the case in which the directions and orientations of both axes are common knowledge, the robots can form an arbitrary given pattern when each robots executes the following algorithm in each step.

**Algorithm 1 (Both axis directions and orientations).**
**Input:** An arbitrary pattern $P$ described as a sequence of points $p_1, \ldots, p_n$, given in lexicographic order. The directions and orientations of the $x$ axis and the $y$ axis is common knowledge.

**Begin**
$\alpha :=$ `Angle(`$p_1$`,` $p_2$`)`;
Give a lexicographic order to all the robots in the system,
including myself, say from left to right and from bottom to top;
$A$ :=First robot in the order;
$B$ :=Second robot in the order;
$\beta :=$ `Angle(`$A$`,` $B$`)`;
**If** I am $B$ **Then** `Do_nothing()`
**Else If** I am $A$
**Then If** $\alpha = \beta$ **Then** `Do_nothing()`
**Else** `Go_Into_Position(`$A$`,` $B$`,` $\alpha$`)`
**Else** %I am neither $A$ nor $B$%
**If** $\alpha = \beta$
**Then** $Unit := \overline{AB}$;
% all the robots agree on a common unit distance %
$Final_Positions :=$ `Find_Final_Positions(`$A$`,`$B$`,`$Unit$`)`;
**If** I am on one of the $Final_Positions$
**Then** `Do_nothing()`
**Else** $Free_Robots := \{$*robots not on one of the* $Final_Positions\}$;
$Free_Points := \{$*Final_Positions with no robots on them*$\}$;
`Go_To_Points(`$Free_Robots$`,` $Free_Points$`)`;
**Else** `Do_nothing()`

**End**

`Angle(`$p$`,` $q$`)` computes the angle between the positive horizontal axis passing through $p$ and the segment $\overline{pq}$. `Do_nothing()` terminates the local computation and the current step of the calling robot.

`Go_Into_Position(`$A$`,` $B$`,` $\alpha$`)` orders $A$ to move so as to achieve angle $\alpha$ with $B$ while staying lexicographically first.

`Find_Final_Positions(`$A$`,` $B$`,` $Unit$`)` figures out the final positions of the robots according to the given pattern, and the positions of $A$ and $B$. The common scaling of the input pattern is defined by the common unit distance $Unit$.

`Go_To_Points(`$Free_Robots$`,` $Free_Points$`)` chooses the robot in $Free_Robots$ that is closest to a point in $Free_Points$ and moves it, as follows:

`Go_To_Points(`$Free_Robots$`,` $Free_Points$`)`
**Begin**
$(r, p) :=$ `Minimum(`$Free_Robots$`,` $Free_Points$`)`;
**If** I am $r$ **Then** `Move(`$p$`)`
**Else** `Do_nothing()`
**End**

`Minimum(`$Free_Robots$`,` $Free_Points$`)` finds one of the $Free_Robots$ that has the minimum Euclidean distance from one of the $Free_Points$ (i.e. with no robot

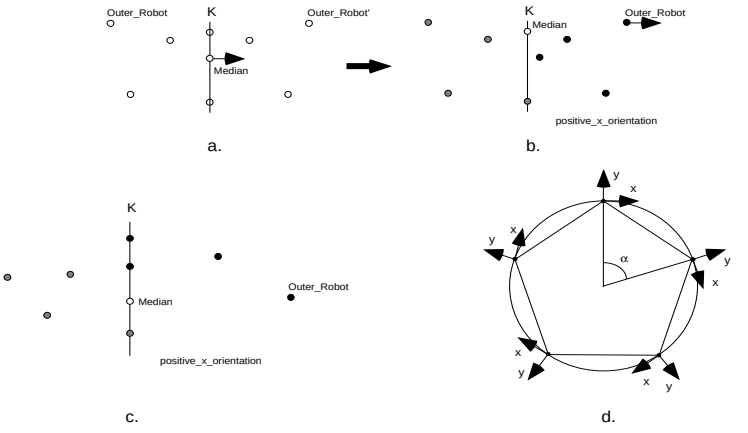



**Fig. 1.** Breaking Symmetry from (a) to (b); defining the sides (c); the unbreakable symmetry of a 5-gon (d)

on it). If more than one robot has minimum distance, the one smaller in the lexicographic order is chosen.

`Move(`$p$`)` terminates the local computation of the calling robot and moves it towards $p$.

**Theorem 1.** *With Algorithm 1, the robots correctly form the input pattern $P$.*

## 3 Knowledge of One Axis Direction and Orientation

Let us now look at the case when only one axis direction and orientation are known. As an aside, note that this case would trivially coincide with the first one, if the robots would have a common handedness (or sense of orientation, as Suzuki and Yamashita call it [7, 9]). We first show that in general, it is impossible to break the symmetry of a situation. We will then show that for the special case of an odd number of robots, symmetry can be broken and an arbitrary pattern can be formed.

**Theorem 2.** *In a system with $n$ anonymous robots that agree only on one axis direction and orientation, the pattern formation problem is unsolvable when $n$ is even.*

In contrast, we now show that for breaking the symmetry, it is enough to know that the number $n$ of robots is odd.

**Algorithm 2 (One axis direction and orientation).**
**Input:** An arbitrary pattern $P$ described as a sequence of points $p_1, \ldots, p_n$, given in lexicographic order. The direction and orientation of the $y$ axis is common knowledge.

**Begin**
**If** We are in a final configuration **Then** `Do_nothing()`;
$p_m$ := `Median_Pattern_Point(`$P$`)`;% median pattern point in $x$ direction %
$p$ := `Outermost_Pattern_Point(`$P$`)`; % outermost pattern point w.r.t. $p_m$ %
*Pattern_Unit_Length* := Horizontal distance in the pattern
between the vertical lines through $p_m$ and $p$;
$K$ := `Median_Robot_Line()`; % through the median robot position %
(*Outer_Robot*, *Outer_Line*, *positive_x_orientation*):=
`Outermost_Robot_Position(`$K$`)`;
*Median_Robot* := `Find_Median_Robot(`$K$`)`;
*Final_Positions* := `Find_Final_Positions(`*Median_Robot*,
*positive_x_orientation*, *Pattern_Unit_Length*,
`Distance(`$K$, *Outer_Line*`)`/2);
*Free_Points* := {*Final_Positions on my side with no robots on them*};
**If** I am the *Median_Robot* **Then** `Do_nothing()`
**Else If** I am the *Outer_Robot*
**Then If** *Free_Points* contains just one (last) free point
**Then** `Move_Towards(`*last free point*`)`
**Else** `Do_nothing()`
**Else If** I am on one of the *Final_Positions* **Then** `Do_nothing()`
**Else** *Free_Robots* := {*robots on my side not on Final_Positions*};
`Go_To_Points(`*Free_Robots*, *Free_Points*`)`

**End**

`Median_Pattern_Point(`$P$`)` finds the median point in direction of $x$ in the input pattern $P$ according to the local orientation of the axis. The ordering is given left-right, bottom-up.

`Outermost_Pattern_Point(`$P$`)` finds the point in the input pattern $P$ that lies on the vertical line farthest from the vertical line through the median. If more than one point exists, then the highest and rightmost is chosen according to the local orientation of the axis.

`Median_Robot_Line()` returns the vertical line through the median robot position. Note that the position of this line does not depend on the local orientations of the $x$ axes of the robots.

`Outermost_Robot_Position(`$K$`)` uniquely determines a robot that is outermost with respect to $K$. It does so by breaking symmetry of the situation, if necessary, in the following way (see Figure 1(a) and (b)):

`Outermost_Robot_Position(`$K$`)`
**Begin**
(*Outer_Robot*, *Outer_Robot*′, *unique*) := `Outer_Two_Robots(`$K$`)`;
**If** *not unique*
**Then If** `Symmetric(`$K$`)`
**Then If** I am the median of the points on $K$
**Then** `Move(`*to my right by* $\epsilon$`)`
**Else** `Do_nothing()`

**Else** *positive_x_orientation* := `Outermost_Asymmetry(`$K$`)`;
*Outer_Robot* := Choose between *Outer_Robot* and *Outer_Robot*′ the one that lies on the *positive_x_orientation*;
**If** I am *Outer_Robot* **Then** `Move(`*away from* $K$ *by* $\epsilon$`)`
**Else** `Do_nothing()`
**Else** *positive_x_orientation* := side on which *Outer_Robot* lies;
*Outer_Line* := vertical line on which *Outer_Robot* lies;
**Return** (*Outer_Robot*, *Outer_Line*, *positive_x_orientation*)
**End**

`Outer_Two_Robots(`$K$`)` finds either two or a unique single topmost robot(s) that lie(s) on the farthest vertical line(s) from $K$. The variable *unique* tells whether the robot found has been unique.

`Symmetric(`$K$`)` returns *True* if the current configuration of the robots in the system is symmetric with respect to $K$, in the following sense. The configuration is called *symmetric with respect to* $K$, if there is a perfect matching for all the robots not on $K$, such that any two matched robots lie on two vertical lines that are in symmetric position with respect to $K$ (see Figure 1(a)).

`Outermost_Asymmetry(`$K$`)` identifies, for an asymmetric configuration, the unique halfplane with respect to $K$ in which a robot $r$ lies that (in some matching) has no symmetric partner with respect to $K$, and that is on the farthest vertical line from $K$ among all robots with this property (see Figure 1(b)).

`Find_Median_Robot(`$K$`)` finds the median robot position in the current configuration of the robots. This median robot position splits the robot positions into two equal size subsets, of size $(n-1)/2$, defined as follows. One subset is in the halfplane of *positive_x_orientation*, including the points on $K$ that are *above* the median robot position, and the other subset is in the other halfplane of $K$, including the points on $K$ that are *below* the median robot position; from now on, we will call these the two *sides* (see Figure 1(c)). According to this definition, the median robot position is unique, and each robot (even if it lies on $K$) can decide to which side it belongs.

`Find_Final_Positions(`*Median_Robot*, *Side*, *Pattern_Unit_Length*, *World_Unit_Length*`)` returns the set of final positions of the robots according to the given pattern, based on the agreement on *Median_Robot* and on *positive_x_orientation*. The common scaling of the input pattern is defined by identifying *Pattern_Unit_Length* with *World_Unit_Length*.

`Distance(`$K, L$`)` returns the horizontal distance between the two vertical lines $K$ and $L$.

### 3.1 Correctness

To see that the above algorithm solves the pattern formation problem for an arbitrary pattern, we argue as follows. First, we show that the robots initially arrive at an agreement configuration, by breaking symmetry if necessary. Then, they translate and scale the pattern with respect to the median and the outermost point, and finally they move to their destinations. To present this

argument in more detail, we start with a brief definition. A *configuration (of the robots)* is a set of robot positions, one position per robot, with no position occupied by more than one robot. An *agreement configuration* is a configuration of the robots in which all robots agree on a unique *median* robot and a unique *outermost* robot, as defined in the above routines `Find_Median_Robot(`$K$`)` and `Outermost_Robot_Position(`$K$`)`. A *final configuration* is a configuration of the robots in which the robots form the desired pattern. Note that a final configuration might or might not be an agreement configuration.

**Theorem 3.** *With Algorithm 2, the robots correctly form the input pattern $P$.*

## 4 Knowledge of One Axis Direction

Note that the difference to the previous section is only the lack of knowledge about the axis orientation. For solving the pattern formation problem in this case, we can use an algorithm similar to the one used in Section 3. We easily observe that with slight modifications in Algorithm 2, the agreement on the orientation of the $x$ axis could have been achieved without using the knowledge of the orientation of the $y$ axis. More precisely, we can do the following:

1. Find the vertical line $K$ on which the median robot lies (as before, this is independent from the direction of $x$).
2. Find the *Outermost* robots with respect to $K$. Since we do not know the orientation of the given axis, let's say the $y$ axis, it is possible that we find more than one outermost robot; there are at most four of them, on both sides of $K$ to the top and to the bottom. In this case, we will detect an (outermost) asymmetry with respect to $K$ as before, or create it as follows. If the configuration is symmetric with respect to $K$, the median robot is uniquely identified, and it moves by some small amount $\epsilon > 0$ to its right, breaking the symmetry. So now, as in the previous section, all the robots agree on the positive direction of the $x$ axis (the side where the outermost asymmetry lies), and at most 2 outermost robots remain (the bottom and top ones on the positive $x$ side, say).
3. The same technique and argument now applies to the $x$ axis as the given one. In this way, an agreement also on the orientation of the $y$ axis can be reached. Now, we can select a unique *Outermost* robot out of the at most two that were remaining, and let it (for convenience) move by $\epsilon$ outwards.
4. The robots can compute their unique final positions and go towards them, in the same way as in Algorithm 2.

Using Theorem 2, we therefore conclude:

**Theorem 4.** *With common knowledge of one axis direction, an odd number of autonomous, anonymous, oblivious, mobile robots can form an arbitrary given pattern, while an even number cannot.*

## 5 No Knowledge

The following theorem states that giving up the common knowledge on at least one axis direction leads to the inability of the system to form an arbitrary pattern.

**Theorem 5.** *With no common knowledge, a set of autonomous, anonymous, oblivious, mobile robots cannot form an arbitrary given pattern.*

## 6 Discussion

We have shown that from an algorithmic point of view, only the most fundamental aspects of mobile robot coordination are being understood. In a forthcoming paper, we propose two algorithms for the point formation problem for oblivious robots; the first one does not need any common knowledge, and the second one works with limited visibility, when two axes are known [3]. There is a wealth of further questions that suggest themselves. For example, we have shown that an arbitrary pattern cannot always be formed; it is interesting to understand in more detail which patterns or classes of patterns can be formed under which conditions, because this indicates which types of agreement can be reached, and therefore which types of tasks can be performed. Slightly faulty snapshots, a limited range of visibility, obstacles that limit the visibility and that moving robots must avoid or push aside, as well as robots that appear and disappear from the scene clearly suggest that the algorithmic nature of distributed coordination of autonomous, mobile robots merits further investigation.

## References

[1] T. Balch and R. C. Arkin. Motor Schema-based Formation Control for Multiagent Robot Teams. *ICMAS*, pages 10–16, 1995.
[2] E. H. Durfee. Blissful Ignorance: Knowing Just Enough to Coordinate Well. *ICMAS*, pages 406–413, 1995.
[3] P. Flocchini, G. Prencipe, N. Santoro, E. Welzl, and P. Widmayer. Point Formation for Oblivious Robots. *manuscript, in preparation*, 1999.
[4] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer. Hard Tasks for Weak Robots. Technical Report TR-99-07, School of Comp. Sc. Carleton Univ., 1999.
[5] M. J. Mataric. Designing Emergent Behaviors: From Local Interactions to Collective Intelligence. *From Animals to Animats 2*, pages 423–441, 1993.
[6] L. E. Parker. Adaptive Action Selection for Cooperative Agent Teams. *Proc. Second Int'l. Conf. on Simulation of Adaptive Behavior*, pages 442–450, 1992.
[7] I. Suzuki and M. Yamashita. Formation and Agreement Problems for Anonymous Mobile Robots. *Proceedings of the 31st Annual Conference on Communication, Control and Computing, University of Illinois, Urbana, IL*, pages 93–102, 1993.
[8] I. Suzuki and M. Yamashita. Distributed Anonymous Mobile Robots - Formation and Agreement Problems. *Proc. Third Colloq. on Struc. Information and Communication Complexity (SIROCCO)*, pages 313–330, 1996.
[9] I. Suzuki and M. Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *Siam J. Comput.*, 28(4):1347–1363, 1999.

# On-Line Load Balancing of Temporary Tasks Revisited

Kar-Keung To and Wai-Ha Wong

Deptartment of Computer Science and Information Systems
The University of Hong Kong
{kkto,whwong}@csis.hku.hk

**Abstract.** We study load balancing problems with temporary jobs (i.e., jobs that arrive and depart at unpredictable time) in two different contexts, namely, machines and network paths. Such problems are known as machine load balancing and virtual circuit routing in the literature. We present new on-line algorithms and improved lower bounds.

## 1 Introduction

In this paper we study on-line algorithms for load balancing with temporary jobs (i.e., jobs that arrive and depart at unpredictable time) in two different contexts, namely, machines and network paths. Such problems are known as machine load balancing and virtual circuit routing in the literature (see [11, 4] for a survey). As for the former, we investigate a number of settings, namely, the list model, the interval model and the tree model. Our results show that these settings, though similar, cause the complexity of the load balancing problem to vary drastically, with competitive ratio jumping from $\Theta(1)$ to $\Theta(\log n)$ and to $\Theta(\sqrt{n})$. We also study these settings in the more general *cluster-based* model. Regarding the virtual circuit routing problem, we give the first algorithm with a sub-linear competitive ratio of $O(m^{2/3})$ when the network contains $m$ edges with identical capacity. We also improve the lower bound from $\Omega(m^{1/4})$ to $\Omega(m^{1/2})$. When edge capacities are not identical, our algorithm is $O(W^{2/3})$-competitive, where $W$ is the total edge capacity normalized to the minimum edge capacity.

### 1.1 On-Line Machine Load Balancing

We study the following on-line problem. There are $n$ machines with identical speed. Jobs arrive and depart at unpredictable time. Each job comes with a positive load. When a job arrives, it must be assigned immediately to a machine in a non-preemptive fashion, increasing the load of that machine by the job load until the job departs. The objective is to minimize the maximum load of any single machine over all time. As with previous work, we measure the performance of an on-line algorithm in terms of competitive ratio (see [11] for a survey), which is the worst-case ratio of the maximum load generated by the on-line algorithm to the maximum load generated by the optimal off-line algorithm.

The above on-line load balancing problem has been studied extensively in the literature (see e.g., [15, 5, 7, 6, 4]). Existing results are distinguished by the

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 103–112, 1999.
© Springer-Verlag Berlin Heidelberg 1999

**Table 1.** Competitive ratios in different settings of assignment restriction.

| Model | List | Interval | Two intervals, Tree, Arbitrary restriction |
|---|---|---|---|
| $n$ machines | $\Theta(1)$ | $\Theta(\log n)$ | $\Theta(\sqrt{n})$ |
| clusters | $\Theta(1)$ | $\Theta(\log \ )$ | $\Theta(\sqrt{K})$ |

presence of restrictions on machine assignment. In the simple case, every job can be assigned to any machine. It has been known for long that Graham's greedy algorithm is $(2-o(1))$-competitive [14, 5]. A matching lower bound was obtained recently by Azar and Epstein [6]. In the model with assignment restriction, each job specifies an arbitrary subset of the machines for possible assignment. For this model, Azar, Broder, and Karlin [5] proved that the competitive ratio of any on-line algorithm is $\Omega(\sqrt{n})$. An algorithm (called ROBIN-HOOD) with a matching upper bound is given in [7].

Notably, the allowance of arbitrary assignment restriction makes the problem significantly harder. It is interesting to investigate the complexities of the settings where the assignment restriction is allowed in a more controllable manner. In particular, Bar-Noy et al. [9] initiated the study of the following hierarchical model. The machines are related in the form of a tree. Each job specifies a machine $M$, so that the algorithm is restricted to choose a machine among the ancestors of $M$. Bar-Noy et al. showed that when the hierarchy is linear (i.e., the list model), we can achieve $O(1)$ competitive ratio. The complexity of the general tree model was left open. In this paper we show that the tree model actually admits an $\Omega(\sqrt{n})$ lower bound. In other words, the tree model, though more controllable, does not offer any advantage over arbitrary assignment restriction.

Intuitively, the list model orders the machines according to their capability, and a job specifies the least capable machine that can serve the job. A natural extension is that a job specifies both the least and the most capable machines (as in many applications, more capable machines would charge more). We call this model the interval model. The previous $O(1)$-competitive algorithm fails to work here. We find that there is indeed an $\Omega(\log n)$ lower bound, and we obtain an $O(\log n)$-competitive algorithm. On the other hand, if a job is allowed to request two or more intervals, we show that every algorithm is $\Omega(\sqrt{n})$-competitive.

This paper also initiates the study of cluster-based assignment restriction, which is a practical extension of machine-based assignment restriction. A *cluster* is a collection of machines with same functionality. More formally, the cluster model states that each machine belongs to one of $k$ clusters, and each job requests some clusters in which any machine can be used to serve the job. Similar to machine-based models, we also study clusters related in the form of lists, intervals and trees. The machine-based algorithms can be easily adapted to those settings, giving $O(1)$, $O(\log n)$ and $O(\sqrt{n})$ upper bounds respectively. However, it is more desirable to derive algorithms with competitive ratios depending on $k$ instead of $n$, the total number of machines, since in reality, $k$ is much smaller than $n$. For the list model and the interval model, we observe that the competitive ratios are $\Theta(1)$ and $\Theta(\log k)$, respectively. For the tree model, we have a more general

algorithm which actually works for the case where a job can request any clusters arbitrarily. Let $s_{\min}$ be the number of machines in the smallest cluster. Denote $K$ as $n/s_{\min}$, the total *normalized* number of machines. The competitive ratio of our algorithm is $O(\sqrt{K})$. Note that $k \leq K \leq n$. If the clusters are of roughly the same size, then $K$ is $O(k)$. We conjecture that this result can be further improved to $O(\sqrt{k})$ for general trees. To support this conjecture, we give an $O(\sqrt{k})$-competitive algorithm for the special case in which the clusters form a tree consisting of two levels. Table 1 shows a summary of these results.

*Related Work:* Two other variants of the machine load balancing problem have also been studied extensively in the literature. They include models in which jobs never depart, and in which jobs can be reassigned [2, 10, 12, 8, 1, 16, 13]. For details, readers can refer to the surveys of Azar [4] and Borodin and El-Yaniv [11].

### 1.2 On-Line Virtual Circuit Routing

The virtual circuit routing problem is a generalization of the machine load balancing problem to the context of high speed networks [2, 3]. The virtual circuit routing problem is defined as follows: We are given a directed graph with $m$ edges. Every edge $e$ is associated with a capacity $c_e$. Again, jobs can arrive and depart in an unpredictable fashion. Each job requests a route of a certain weight $w$ from a source to a destination. When a job arrives, an on-line algorithm assigns the job to a path connecting the source to the destination, thereby increasing the load of every edge $e$ along that path by $w/c_e$ until the job departs. The objective is to minimize the maximum load generated on any single edge over all time. The performance is again measured in terms of competitive ratio.

It is widely known that the $\Omega(\sqrt{n})$ lower bound on the competitive ratio of machine load balancing with assignment restriction [5] can lead to an $\Omega(m^{1/4})$ lower bound on the competitive ratio of the virtual circuit routing problem [3, 15]. This lower bound holds even when all edges have the same capacity. An interesting open problem in the literature is to determine the competitive ratio of the virtual circuit routing problem (see e.g., [4]). Prior to our work, the only related result is the work of Awerbuch et al. [3], who showed that if limited re-routing is allowed (i.e., the path to which a job is assigned can change dynamically), an $O(\log n)$-competitive algorithm exists.

In this paper we study the original virtual circuit routing problem and present an algorithm which is $O(m^{2/3})$-competitive when all edges have the same capacity. In addition, we improve the lower bound to $\Omega(\sqrt{m})$. For networks with general edge capacities, the competitive ratio of our algorithm becomes $O(W^{2/3})$, where $W$ is the total edge capacity normalized to the minimum edge capacity (i.e., $\sum_e c_e/c_{\min}$, where $c_{\min}$ is the minimum edge capacity).

## 2 Machine-Based Assignment Restriction

In this section we study the competitiveness in different settings of assignment restriction. In the tree model, machines are nodes of a tree. Each job specifies a machine, and the on-line assignment algorithm can assign the job to any ancestor of the specified machine in the tree. The list model is a special case where

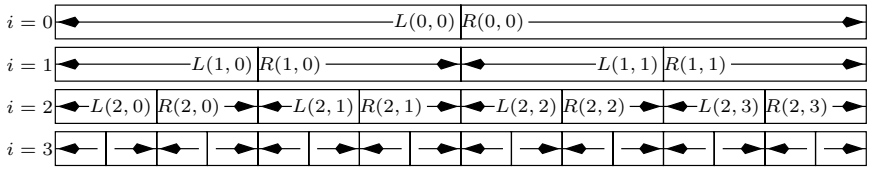



**Fig. 1.** Properly aligned sublists when $n = 16$.

machines are nodes of a list. Each job specifies a machine, and the algorithm can assign the job to any machine in the list between the list head and the requested machine. A 5-competitive algorithm for the list model has been known [9]. For the tree model, the best known algorithm is the $O(\sqrt{n})$-competitive algorithm inherited from arbitrary assignment restriction [7].

We define the interval model as an extension of the list model. Each job specifies two machines, and the algorithm may choose any machine in the list between these two machines to serve the job. In this section we show that this extension raises the competitive ratio to $\Theta(\log n)$. The lower bound result holds even if we add the assumption that the jobs never depart and all jobs generate the same load. We also show that all algorithms in the tree model is $\Omega(\sqrt{n})$-competitive. A similar result is obtained when we further extend the interval model to allow two intervals per request.

### 2.1 The Interval Model

We first show an $O(\log n)$-competitive algorithm (called INTERVAL) for the interval model. Then we state an $\Omega(\log n)$ lower bound on the competitive ratio, showing that INTERVAL is optimal. In this extended abstract we omit the proof of the latter. To ease our discussion, we assume $n$ is power of two.

For the list model, a 5-competitive algorithm was shown in [9]. This algorithm will be referred to as LINEAR. Our algorithm INTERVAL identifies $2n - 2$ special sublists, and run a copy of LINEAR on each of them. These sublists are called *properly aligned* sublists. For each $i$ between 0 and $\log n - 1$, we partition the list of machines evenly into $2^i$ intervals named $I(i, 0)$ to $I(i, 2^i - 1)$. Each of these intervals is further subdivided into two properly aligned sublists: the left one $L(i, j)$ is in reverse order, and the right one $R(i, j)$ is in the original order. Figure 1 shows an example. Note that each machine is contained in exactly $\log n$ properly aligned sublists.

Suppose a job arrives, requesting an interval $[l, r]$. INTERVAL finds the sublist $I(i, j)$ with the largest $i$ such that $I(i, j)$ contains the interval $[l, r]$. Since $I(0, 0)$ contains all machines, such a pair always exists. Let $\hat{l}$ and $\hat{r}$ be the number of machines in $L(i, j)$ and $R(i, j)$ respectively which are also in $[l, r]$. Note that, due to the maximum $i$ requirement, the head of $L(i, j)$ (respectively $R(i, j)$) is always included in $[l, r]$ whenever $\hat{l} > 0$ (respectively $\hat{r} > 0$). INTERVAL dispatches the job to the copy of LINEAR running on $L(i, j)$ if $\hat{l} \geq \hat{r}$, and to the copy of LINEAR running on $R(i, j)$ otherwise. INTERVAL would assign the job to the machine returned by the appropriate copy of LINEAR.

**Theorem 1.** INTERVAL *is* $(10 \log n)$*-competitive.*

*Proof.* Assume to the contrary that there is a sequence of jobs such that the optimal algorithm $\mathcal{O}$ generates a maximum load of $OPT$, and INTERVAL generates a maximum load of more than $10 \log n OPT$. Consider the time when the maximum load of INTERVAL occurs in machine $M$. Since there are only $\log n$ properly aligned sublists containing $M$, in at least one of them $M$ has a load of more than $10OPT$. Without loss of generality, suppose this sublist is $L(i_0, j_0)$.

Consider all jobs dispatched to the sublist $L(i_0, j_0)$. As scheduled by LINEAR, $M$ receives a load of more than $10OPT$. We show in the following paragraphs that it is possible to construct an off-line assignment $\mathcal{A}$ for $L(i_0, j_0)$ so that each machine receives a load of at most $2OPT$ at any time. Therefore, LINEAR is not 5-competitive, a contradiction occurs.

For each job, we find the machine $\sigma$ to which $\mathcal{O}$ assigned the job. Note that $\sigma$ is a machine in either $L(i_0, j_0)$ or $R(i_0, j_0)$, since they contain the requested interval. If $\sigma$ is in $L(i_0, j_0)$, $\mathcal{A}$ also assigns the job to $\sigma$. Otherwise, if $\sigma$ is the $r$-th machine in $R(i_0, j_0)$, $\mathcal{A}$ assigns the job to the $r$-th machine in $L(i_0, j_0)$. This is always allowed: for INTERVAL to dispatch the job to $L(i_0, j_0)$, the number of machines permissible in this sublist must be more than that in $R(i_0, j_0)$.

On the other hand, the jobs assigned to each machine by $\mathcal{A}$ is simply the union of jobs assigned to two machines in $\mathcal{O}$, each having a load of at most $OPT$. $\mathcal{A}$ thus gives rise to a load of at most $2OPT$ at any time. □

**Theorem 2.** *No on-line algorithm for the interval model has competitive ratio less than* $\log n/2$*. This holds even if jobs never depart, and even if all jobs generate the same load.*

### 2.2 The Tree Model

We show that the $O(\sqrt{n})$-competitive algorithm ROBIN-HOOD introduced in [7] is asymptotically optimal for the tree model. Note that if jobs never depart, $O(1)$ competitive ratio can be achieved [9].

**Theorem 3.** *No on-line algorithm for the tree model is* $(\sqrt{n} - 1)$*-competitive. This is true even if all jobs generate the same load.*

*Proof.* The proof is adapted from the lower bound proof for arbitrary assignment restriction presented in [15]. Consider the following tree of $r^2 + r$ nodes, with $r$ non-leaf nodes forming a list, and $r^2$ leaf nodes being children of the tail of this list. The following job sequence ensures that any on-line algorithm assigns to one of the nodes at least $r$ jobs. The job sequence consists of $r^2$ phases. In the $p$-th phase, $r$ jobs are released requesting an ancestor of the $p$-th leaf. If the on-line algorithm assigns all these jobs to the leaf, we are done. Otherwise, we retain a job assigned to a non-leaf node, and let all other jobs depart. After $r^2$ phases, the non-leaf nodes must be serving at least $r^2$ jobs, so one of them must be serving at least $r$ jobs.

In an off-line assignment, only non-departing jobs are assigned to leaf nodes. The maximum load created is 1. So the on-line algorithm is no better than $r$-competitive. Since $r > \sqrt{n} - 1$, the theorem follows. □

In the above argument, if we number the non-leaf nodes as 1 to $r$, and the leaf nodes as $r+1$ to $r^2+r$, all the jobs have assignment restriction in the form $\{1, 2, \cdots, r, j\}$. Therefore, if we extend the interval model so that each job can specify two intervals, the same lower bound holds.

**Corollary 4.** *No on-line algorithm for the two-interval model is* $(\sqrt{n}-1)$*-competitive. This is true even if all jobs generate the same load.*

## 3 Cluster-Based Assignment Restriction

In reality, assignment restriction is usually used to model the requirement of jobs for some discrete capabilities possessed only by some machines. The number of the distinct sets of capabilities possessed by the machines is usually much smaller than the number of machines. This motivates us to study cluster-based assignment restriction models, an extension in which each machine belongs to one of $k$ clusters. Each job specifies some clusters, so that only machines in these clusters can serve the job. We define the list, interval, 2-interval, tree and arbitrary restriction models analogous to the machine-based models in Sect. 2.

Here are some simple observations. In the extreme case in which each cluster contains only one machine, the cluster-based models reduce to the machine-based models. As a result, all the lower bounds in Sect. 2 still apply, replacing $n$ by $k$ in the respective bounds. As mentioned earlier, $k$ may be much smaller than $n$ and thus it is more interesting to see whether we can provide better upper bounds in terms of $k$.

For the list model, the algorithm of Bar-Noy et al. [9] is $O(1)$-competitive, independent of $n$ and $k$. For the interval model, the algorithm in Sect. 2.1 can be extended to produce an $O(\log k)$-competitive algorithm, matching the lower bound. For the tree model and the arbitrary restriction model, a trivial algorithm which always assign a job to a machine in the largest specified cluster is $(k+1)$-competitive. However, it is not clear how we can provide better upper bounds.

In this section we show that the algorithm of Azar et al. [7] can be generalized to the cluster-based arbitrary assignment restriction model, producing an $O(\sqrt{K})$-competitive algorithm, where $K = n/s_{\min}$ and $s_{\min}$ is the size of the smallest cluster. Thus in the case when all clusters are of similar sizes, it is $O(\sqrt{k})$-competitive. We also show an $O(\sqrt{k})$-competitive algorithm that works in the special case where the clusters are organized as a two-level tree.

### 3.1 Arbitrary Assignment Restriction Model

In this section we study the cluster-based arbitrary assignment restriction model in which each machine belongs to one of $k$ clusters and each job can request an arbitrary subset of clusters. Denote $s_{\min}$ as the size of the smallest cluster and let $K = n/s_{\min}$. The lower bound on competitive ratio given in [5] can be expressed as $\Omega(\sqrt{K})$. We extend the algorithm Robin-Hood introduced by Azar et al. [7] to work in this model, resulting in an $O(\sqrt{K})$-competitive algorithm Cluster.

To simplify our discussion, we assume that Cluster knows a value $OPT$ specifying the maximum load generated by the optimal off-line algorithm. With

the doubling technique [2], CLUSTER can be converted into an algorithm that does not know $OPT$ in advance (instead, it is approximated dynamically). This conversion incurs only a degradation factor of 4 to the competitive ratio.

At any time, a cluster is said to be *overloaded* if its average load is greater than $\sqrt{K}\,OPT$ under CLUSTER. For any overloaded cluster, define its *windfall time* to be the last moment it became overloaded.

When a job arrives, CLUSTER chooses a cluster for the job as follows. If possible, assign the job to a cluster that is not overloaded. Otherwise, assign it to the cluster with the greatest windfall time. Whenever a job is assigned to a cluster, the machine with the lowest load in that cluster is chosen.

**Theorem 5.** *Given an accurate $OPT$, the load of any machine under* CLUSTER *at any time is at most $(2\sqrt{K}+2)OPT$. Thus,* CLUSTER *is $(2\sqrt{K}+2)$-competitive.*

*Proof.* The proof is similar to that of ROBIN-HOOD. The detail is omitted.

### 3.2 Two-Level Trees

In this section we present a simple $O(\sqrt{k})$-competitive algorithm for the two-level cluster model, in which the clusters form a tree consisting of two levels. More precisely, the machines are partitioned into $k-1$ leaf clusters $S_i$ $(1 \le i \le k-1)$ and a root cluster $S_0$. Each job specifies one of $S_i$ containing machines which can be used to serve the job, but the algorithm may instead use a machine in $S_0$. The result in Sect. 2.2 can be adapted to this model, giving an $\Omega(\sqrt{k})$ lower bound on the competitive ratio. Here, we present an algorithm called TWOLEVEL with a matching upper bound.

We assume that TWOLEVEL knows $OPT$, the maximum load generated by the optimal off-line algorithm. As in Sect. 3.1, this assumption can easily be removed. If a job arrives which requests a leaf cluster containing a machine with less than $\sqrt{k}\,OPT$ load, TWOLEVEL assigns the job to that machine. Otherwise, TWOLEVEL assigns it to a machine in $S_0$ with the lowest current load.

**Theorem 6.** *Given an accurate $OPT$,* TWOLEVEL *is $(\sqrt{k}+2)$-competitive.*

*Proof.* Since no job creates a load of more than $OPT$, each machine in a leaf cluster has a load of at most $(\sqrt{k}+1)OPT$. The remainder of the proof establishes that no machine of the root cluster gets a load of more than $(\sqrt{k}+2)OPT$.

Let $s_i$ denote the number of machines in $S_i$. Consider any particular time $t$. Denote $n_i(t)$ and $f_i(t)$ as the total load of machines in $S_0$ at $t$ due to jobs requesting $S_i$ under TWOLEVEL and the optimal off-line algorithm respectively.

Note that, at the last time when $n_i(t)$ increases, all machines in $S_i$ must have load at least $\sqrt{k}\,OPT$. The off-line algorithm must accommodate the sum of load in these machines, i.e.,

$$s_i\sqrt{k}\,OPT + n_i(t) \le (s_0 + s_i)OPT \qquad \text{for } 1 \le i \le k-1. \tag{1}$$

On the other hand, the off-line algorithm assigns a load of at least $n_i(t)$ to machines in $S_0$ or $S_i$.

$$n_i(t) \le s_i OPT + f_i(t) \qquad \text{for } 1 \le i \le k-1. \tag{2}$$

Eliminating $s_i$, summing over $i$ and adding the equality $n_0(t) = f_0(t)$, we have

$$\sqrt{k} \sum_{i=0}^{k-1} n_i(t) \leq s_0(k-1)OPT + (\sqrt{k}-1) \sum_{i=0}^{k-1} f_i(t) \ . \tag{3}$$

Note that $\sum_{i=0}^{k-1} f_i(t) \leq s_0 OPT$. As a result, $\sqrt{k} \sum_{i=0}^{k-1} n_i(t) \leq (k + \sqrt{k} - 2)s_0 OPT$; thus, $\sum_{i=0}^{k-1} n_i(t) \leq (\sqrt{k}+1)s_0 OPT$. Since TwoLevel always assigns a job to a machine with the lowest load when using $S_0$, the load at $t$ is at most $(\sqrt{k}+2)OPT$ for any machine in $S_0$. □

## 4 Virtual Circuit Routing

In this section we study the virtual circuit routing problem. We are given a directed graph with edge set $E$ of $m$ edges. Each edge $e$ is associated with a capacity $c_e$. Jobs arrive at unpredictable time, each requesting a route from a source to a destination. We need to find a path from the source to the destination, thereby increasing the load of each edge $e$ in the path by $w/c_e$, where $w$ is the weight specified by the job. Our objective is to minimize, over all time, the maximum load on any edge.

In [2], Aspnes et al. studied the problem with the assumption that jobs never depart. With this assumption, they gave an $O(\log m)$-competitive on-line algorithm and proved that no on-line algorithm can have competitive ratio better than $O(\log m)$. We present an on-line algorithm called VC-Routing for the more general case where jobs may depart at unpredictable time. It is $O(W^{2/3})$-competitive, where $W = \sum_{e \in E} c_e / c_{\min}$ and $c_{\min}$ is the minimum capacity of the edges. When all edges have identical capacity, the competitive ratio is equivalent to $O(m^{2/3})$. We also prove that any on-line algorithm for the problem is $\Omega(\sqrt{m})$-competitive, even if all the edges have identical capacity.

### 4.1 The Algorithm

VC-Routing is a novel adaptation of Robin-Hood, the algorithm achieving optimal competitive ratio for the machine load balancing problem with arbitrary assignment restriction. The main challenge in the virtual circuit routing problem is that the length of the path chosen by the off-line algorithm may be different from that of the path chosen by the on-line algorithm. As a result, the aggregate load generated by the on-line algorithm can be much more than that generated by the optimal off-line algorithm. In order to control the difference of the aggregate load, VC-Routing takes into account the length of paths when assigning a job. Roughly speaking, it prefers relatively short paths, and applies a strategy similar to Robin-Hood only to those short paths. The details are as follows.

We assume that VC-Routing knows a value $OPT$ specifying the maximum load generated by the optimal off-line algorithm. As in Sect. 3.1, such an algorithm can be converted to one which does not need the value of $OPT$.

At any particular time, we say an edge is *overloaded* if its current load is greater than $W^{2/3}\,OPT$. An overloaded path is a path which contains an overloaded edge. For an overloaded edge, we define its *windfall time* as the last

moment it became overloaded. The windfall time of an overloaded path is the minimum windfall time of the overloaded edges on the path. We say a path is *short* if its length is no more than $W^{1/3}$; it is *medium* if its length is in the range $(W^{1/3}, W^{2/3}]$; otherwise, it is long. For every job $j$ with weight $w(j)$, we say a path is *eligible* if every edge on it satisfies $w(j)/c_e \leq OPT$.

When a new job arrives, VC-Routing selects a path among the eligible paths in the order of short, medium and finally long paths. Ties among short paths are broken as follows. VC-Routing selects one that is not overloaded if it exists; otherwise, it selects one with maximum windfall time. For medium and long paths, ties are broken arbitrarily.

We analyse the competitive ratio of VC-Routing by bounding the load on any edge. For edges which are not overloaded, this is trivially bounded by $W^{2/3}OPT$. To bound the load on an overloaded edge $e_{\rm o}$, we partition the jobs assigned to $e_{\rm o}$ into three sets, depending on the arrival time of the jobs and on how the off-line algorithm assigns the jobs. Let $\mathcal{J}_1(e_{\rm o})$ be the partition containing jobs which are assigned to $e_{\rm o}$ at or before the windfall time of $e_{\rm o}$, let $\mathcal{J}_2(e_{\rm o})$ be the partition containing those remaining jobs which the off-line algorithm assigns to either a medium or a long path, and let $\mathcal{J}_3(e_{\rm o})$ be the partition containing the other jobs. We show the competitive ratio of VC-Routing using the following lemma. The proofs will appear in the full version of this paper.

**Lemma 7.** *The total weight of jobs in $\mathcal{J}_2(e_{\rm o})$ and $\mathcal{J}_3(e_{\rm o})$ are both at most $W^{2/3}OPT\,c_{\min}$.*

**Theorem 8.** *Given an accurate $OPT$, the load of any edge under* VC-Routing *at any time is at most $(3W^{2/3}+1)OPT$. Thus* VC-Routing *is $(3W^{2/3}+1)$-competitive.*

### 4.2 Lower Bound

We show an $\Omega(\sqrt{m})$ lower bound on the competitive ratio of any on-line algorithm for the virtual circuit routing problem. Consider the graph in Fig. 2, which has $4r$ nodes and $3r^2 + r$ edges with identical capacity. The set of source nodes and the set of destination nodes form a complete bipartite graph. Each of these sets form a bipartite graph with a set of $r$ intermediate nodes. The two sets of intermediate nodes form a bipartite matching. Let $E'$ be the set of edges connecting the intermediate nodes.

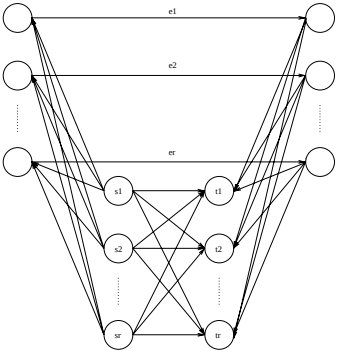



**Fig. 2.** An $\Omega(\sqrt{m})$ lower bound.

We construct a sequence of jobs with $r^2$ phases. In each phase, $r$ jobs of unit weight are released with a distinct pair of source and destination. If the on-line algorithm assigns all these jobs to the edge connecting the source and the destination, we are done. Otherwise, at least one of these jobs is assigned to a path containing an edge in $E'$. At the end of this phase, all jobs except this departs. After $r^2$ phases, $r^2$ jobs remain, each

increases the load of one edge in $E'$. Since there are $r$ edges in $E'$, at least one of them has a load of $r$. On the other hand, in an off-line assignment, the job that never departs is assigned to the edge connecting the source and the destination involved. The maximum load is 1 at all time. Therefore, we have the following theorem.

**Theorem 9.** *No on-line algorithm for the virtual circuit routing problem is* $(\sqrt{m/3} - 1)$*-competitive. This is true even if all edges have identical capacity.*

## References


[1] M. Andrews, M. X. Goemans, and L. Zhang. Improved bounds for on-line load balancing. In *Proceedings of the Second Annual International Computing and Combinatorics Conference*, pages 1–10, 1996.

[2] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line routing of virtual circuits with applications to load balancing and machine scheduling. *Journal of the ACM*, 44(3):486–504, 1997.

[3] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts. Competitiveness routing of virtual circuits with unknown duration. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 321–327, 1994.

[4] Y. Azar. On-line load balancing. In *Online Algorithms: the state of the art*, Lecture notes in computer science; 1442, Springer, pages 178–195, 1998.

[5] Y. Azar, A. Z. Broder, and A. R. Karlin. On-line load balancing. *In Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 218–225, 1992.

[6] Y. Azar and L. Epstein. On-line load balancing of temporary tasks on identical machines. In *5th Israeli Symposium on Theory of Computing and Systems*, pages 119–125, 1997.

[7] 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.

[8] Y. Azar, J. S. Naor, and R. Rom. Competitiveness of on-line assignments. *Journal of Algorithms*, 18:221–237, 1995.

[9] A. Bar-Noy, A. Freund, and J. Naor. Online load balancing in a hierarchical server topology. In *Proceedings of the Seventh Annual European Symposium on Algorithm*, pages 77–88, 1999.

[10] P. Berman, M. Charikar, and M. Karpinski. On-line load balancing for related machines. In *Workshop on Algorithms and Data Structures*, pages 116–125, 1997.

[11] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[12] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9(1):91–103, 1980.

[13] G. Galambos and B. J. Woeginger. An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling. *SIAM Journal on Computing*, 22(2):349–355, 1993.

[14] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[15] Y. Ma and S. Plotkin. An improved lower bound for load balancing of tasks with unknown duration. *Information Processing Letters*, 62(6):301–303, 1997.

[16] S. Phillips and J. Westbrook. Online load balancing and network flow. In *Proceedings of the Twenty-Fifth Annual Symposium on Theory of Computing*, pages 402–411, 1993.

# Online Routing in Triangulations*

Prosenjit Bose and Pat Morin

School of Computer Science, Carleton University
Ottawa, Canada, K1S 5B6
{jit,morin}@scs.carleton.ca

**Abstract.** We consider online routing strategies for routing between the vertices of embedded planar straight line graphs. Our results include (1) two deterministic memoryless routing strategies, one that works for all Delaunay triangulations and the other that works for all regular triangulations, (2) a randomized memoryless strategy that works for all triangulations, (3) an $O(1)$ memory strategy that works for all convex subdivisions, (4) an $O(1)$ memory strategy that approximates the shortest path in Delaunay triangulations, and (5) theoretical and experimental results on the competitiveness of these strategies.

## 1 Introduction

In this paper we consider online routing in the following abstract setting: The environment is a planar straight line graph [12], $T$, with $n$ vertices, whose edges are weighted by the Euclidean distance between their endpoints, the source $v_{src}$ and destination $v_{dst}$ are vertices of $T$, and a packet can only move on edges of $T$. Initially, a packet only knows $v_{src}$, $v_{dst}$, and $N(v_{src})$, where $N(v)$ denotes the set of vertices adjacent to $v$.

We classify online routing strategies based on their use of memory and/or randomization. Define $v_{cur}$ as the vertex at which the packet is currently stored. A routing strategy is called *memoryless* if the next step taken by a packet depends only on $v_{cur}$, $v_{dst}$, and $N(v_{cur})$. A strategy is *randomized* if the next step taken by a packet is chosen randomly from $N(v_{cur})$. A randomized strategy is memoryless if the distribution used to choose from $N(v_{cur})$ is a function only of $v_{cur}$, $v_{dst}$, and $N(v_{cur})$.

For a strategy $\mathcal{S}$ we say that a graph *defeats* $\mathcal{S}$ if there is a source/destination pair such that a packet never reaches the destination when beginning at the source. If $\mathcal{S}$ finds a path $P$ from $v_{src}$ to $v_{dst}$ we call $P$ the $\mathcal{S}$ path from $v_{src}$ to $v_{dst}$. Here we use the term *path* in an intuitive sense rather than a strict graph theoretic sense, since $P$ may visit the same vertex more than once.

In this paper we also consider, as a special case, a class of "well-behaved" triangulations. The *Voronoi diagram* [11] of $S$ is a partitioning of space into cells such that all points within a Voronoi cell are closer to the same element $p \in S$

* This research was supported by the Natural Sciences and Engineering Research Council of Canada.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 113-122, 1999
© Springer-Verlag Berlin Heidelberg 1999

than any other point in $S$. The *Delaunay triangulation* is the straight-line face dual of the Voronoi diagram, i.e., two points in $S$ have an edge between them in the Delaunay triangulation if their Voronoi regions have an edge in common.

In this paper we consider several different routing strategies and compare their performance empirically. In particular, we describe (1) a memoryless strategy that is not defeated by any Delaunay triangulation, (2) a memoryless strategy that is not defeated by any regular triangulation, (3) a memoryless randomized strategy that uses 1 random bit per step and is not defeated by any triangulation, (4) a strategy that uses $O(1)$ memory that is not defeated by any convex subdivision, (5) a strategy for Delaunay triangulations that uses $O(1)$ memory in which a packet never travels more than a constant times the Euclidean distance between $v_{src}$ and $v_{dst}$, and (6) a theoretical and empirical study of the quality (length) of the paths found by these strategies.

The first four routing strategies are described in Section 2. Section 3 presents theoretical and empirical results on the length of the paths found by these strategies and describes our strategy for Delaunay triangulations. A discussion of related previous work is provided in Section 4. Finally, Section 5 summarizes and describes directions for future research.

Due to space constraints, some proofs and figures are omitted from this extended abstract. The interested reader is referred to the full version of the paper [3].

## 2 Four Simple Strategies

In this section we describe four online routing strategies and prove theorems about which types of graphs never defeat them. We begin with the simplest (memoryless) strategies and proceed to the more complex strategies.

### 2.1 Greedy Routing

The *greedy routing* (GR) strategy always moves the packet to the neighbor $gdy(v_{cur})$ of $v_{cur}$ that minimizes $dist(gdy(v_{cur}), v_{dst})$, where $dist(p, q)$ denotes the Euclidean distance between $p$ and $q$. In the case of ties, one of the vertices is chosen arbitrarily. The greedy routing strategy can be defeated by a triangulation $T$ in two ways: (1) the packet can get trapped moving back and forth on an edge of the triangulation (Fig. 1 (a)), or (2) the packet can get trapped on a cycle of three or more vertices (Fig. 1 (b)). However, as the following theorem shows, neither of these situations can occur if $T$ is a Delaunay triangulation.

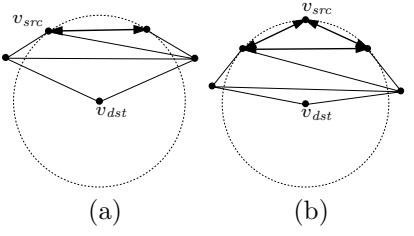



**Fig. 1.** Triangulations that defeat the greedy routing strategy.

**Theorem 1** *There is no point set whose Delaunay triangulation defeats the greedy routing strategy.*

### 2.2 Compass Routing

The *compass routing* (CR) strategy always moves the packet to the vertex $cmp(v_{cur})$ that minimizes the angle $\angle v_{dst}, v_{cur}, cmp(v_{cur})$ over all vertices adjacent to $v_{cur}$. Here the angle is taken to be the smaller of the two angles as measured in the clockwise and counterclockwise directions. In the case of ties, one of the (at most 2) vertices is chosen using some arbitrary deterministic rule.

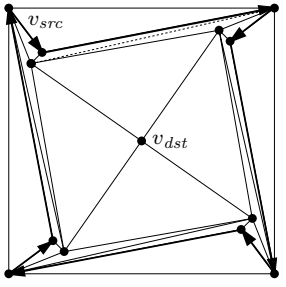



**Fig. 2.** A triangulation that defeats the compass routing strategy.

One might initially believe (as we did) that compass routing can always be used to find a path between any two vertices in a triangulation. However, the triangulation in Fig. 2 defeats compass routing. When starting from one of the vertices on the outer face of $T$, and routing to $v_{dst}$, the compass routing strategy gets trapped on the cycle shown in bold. The following lemma shows that any triangulation that defeats compass routing causes the packet to get trapped in a cycle.

**Lemma 1** *Let $T$ be a triangulation that defeats compass routing, and let $v_{dst}$ be a vertex such that compass routing fails to route a packet to $v_{dst}$ when given some other vertex as the source. Then there exists a cycle $C = v_0, \ldots, v_{k-1}$ ($k \geq 3$) in $T$ such that $cmp(v_i) = v_{i+1}$ for all $0 \leq i < k$.*[1]

We call such a cycle, $C$, a *trapping cycle* in $T$ for $v_{dst}$. Next we characterize trapping cycles in terms of a visibility property of triangulations. Let $t_1$ and $t_2$ be two triangles in $T$. Then we say that $t_1$ *obscures* $t_2$ if there exists a ray originating at $v_{dst}$ that strikes $t_1$ first and then $t_2$. Let $u$ and $v$ be any two vertices of $T$ such that $cmp(u) = v$. Then define $\triangle uv$ as the triangle of $T$ that is contained in the closed half-plane bounded by the line through $uv$ and that contains $v_{dst}$. We obtain the following useful characterization of trapping cycles.

**Lemma 2** *Let $T$ be a triangulation that defeats compass routing and let $C = v_0, \ldots, v_{k-1}$ be a trapping cycle in $T$ for vertex $v_{dst}$. Then $\triangle v_i v_{i+1}$ is either identical to, or obscures $\triangle v_{i-1} v_i$, for all $0 \leq i < k$.*

A *regular triangulation* [13] is a triangulation obtained by orthogonal projection of the faces of the lower hull of a 3-dimensional polytope onto the plane. Note that the Delaunay triangulation is a special case of a regular triangulation in which the vertices of the polytope all lie on a paraboloid. Edelsbrunner [7] showed that if $T$ is a regular triangulation, then $T$ has no set of triangles that

[1] Here and henceforth, all subscripts are assumed to be taken mod .

obscure each other cyclically from *any* viewpoint. This result, combined with Lemma 2, yields our main result on compass routing.

**Theorem 2** *There is no regular triangulation that defeats the compass routing strategy.*

### 2.3 Randomized Compass Routing

In this section, we consider a randomized routing strategy that is not defeated by any triangulation. Let $cw(v)$ be the vertex in $N(v)$ that minimizes the clockwise angle $\angle v_{dst}, v, cw(v)$ and let $ccw(v)$ be the vertex in $N(v)$ that minimizes the counterclockwise angle $\angle v_{dst}, v, ccw(v)$ . Then the *randomized compass routing* (RCR) strategy moves the packet to one of $\{cw(v_{cur}), ccw(v_{cur})\}$ with equal probability.

Before we can make statements about which triangulations defeat randomized compass routing, we must define what it means for a triangulation to defeat a randomized strategy. We say that a triangulation $T$ defeats a (randomized) routing strategy if there exists a pair of vertices $v_{src}$ and $v_{dst}$ of $T$ such that a packet originating at $v_{src}$ with destination $v_{dst}$ has probability 0 of reaching $v_{dst}$ in any finite number of steps.

Note that, since randomized compass routing is memoryless, proving that a triangulation $T$ does not defeat randomized compass routing implies that a packet reaches its destination with probability 1. The following theorem shows the versatility of randomized compass routing.

**Theorem 3** *There is no triangulation that defeats the randomized compass routing strategy.*

*Proof.* Assume, by way of contradiction that a triangulation $T$ exists that defeats the randomized compass routing strategy. Then there is a vertex $v_{dst}$ of $T$ and a minimal set $S$ of vertices such that: (1) $v_{dst} \notin S$, (2) the subgraph $H$ of $T$ induced by $S$ is connected, and (3) for every $v \in S$, $cw(v) \in S$ and $ccw(v) \in S$.

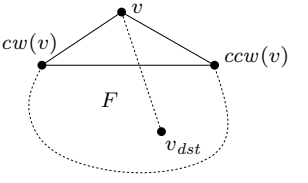



**Fig. 3.** The proof of Theorem 3.

Refer to Fig. 3 for what follows. The vertex $v_{dst}$ lies in some face $F$ of $H$. Let $v$ be a vertex on the boundary of $F$ such that the line segment $(v, v_{dst})$ is contained in $F$. Such a vertex is guaranteed to exist [5]. The two neighbours of $v$ on the boundary of $F$ must be $cw(v)$ and $ccw(v)$ and these cannot be the same vertex (since $F$ contains $(v, v_{dst})$ in its interior). Note that, by the definition of $cw(v)$ and $ccw(v)$, and by the fact that $T$ is a triangulation, the triangle $(cw(v), v, ccw(v))$ is in $T$. But this is a contradiction, since then $v$ is not on the boundary of $F$. □

### 2.4 Right-Hand Routing

The folklore "right-hand rule" for exploring a maze states that if a player in a maze walks around never lifting her right-hand from the wall, then she will eventually visit every wall in the maze. More specifically, if the maze is the face of a connected planar straight line graph, the player will visit every edge and vertex of the face [2].

Let $T$ be any convex subdivision. Consider the planar subdivision $T'$ obtained by deleting from $T$ all edges that properly intersect the line segment joining $v_{src}$ and $v_{dst}$. Because of convexity, $T'$ is connected, and $v_{src}$ and $v_{dst}$ are on the boundary of the same face $F$ of $T'$. The *right-hand routing* (RHR) strategy uses the right-hand rule on the face $F$ to route from $v_{src}$ to $v_{dst}$. Right-hand routing is easily implemented using only $O(1)$ additional memory by remembering $v_{src}$, $v_{dst}$, and the last vertex visited.

**Theorem 4** *There is no convex subdivision that defeats the right-hand routing strategy.*

## 3 Competitiveness of Paths

Thus far we have considered only the question of whether routing strategies can find *a* path between any two vertices in $T$. An obvious direction for research is to consider the length of the path found by a routing strategy. We say that a routing strategy is $c$-competitive for $T$ if for any pair $(v_{src}, v_{dst})$ in $T$, the length (sum of the edge lengths) of the path between $v_{src}$ and $v_{dst}$ found by the strategy is at most $c$ times the length of the shortest path between $v_{src}$ and $v_{dst}$ in $T$. In the case of randomized strategies, we use the expected length of the path. A strategy has a competitive ratio of $c$ if it is $c$-competitive.

This section addresses questions about the competitive ratio of the strategies described so far, as well as a new strategy specifically targeted for Delaunay triangulations. We present theoretical as well as experimental results.

### 3.1 Negative Results

It is not difficult to contrive triangulations for which none of our strategies are $c$-competitive for any constant $c$. Thus it is natural to restrict our attention to a well behaved class of triangulations. Unfortunately, even for Delaunay triangulations none of the strategies described so far are $c$-competitive.

**Theorem 5** *There exists Delaunay triangulations for which none of the greedy, compass, randomized compass, or right-hand routing strategies are c-competitive for any constant c.*

### 3.2 A $c$-Competitive Strategy for Delaunay Triangulations

Since none of the strategies described in Section 2 is competitive, even for Delaunay triangulations, an obvious question is whether there exists any strategy that is competitive for Delaunay triangulations. In this section we answer this question in the affirmative. Our strategy is based on the remarkable proof of Dobkin *et al* [6] that the Delaunay triangulation approximates the complete Euclidean graph to within a constant factor in terms of shortest path length. In the following we will use the notation $\mathrm{x}(p)$ (resp. $\mathrm{y}(p)$) to denote the $x$-coordinate (resp. $y$-coordinate) of the point $p$, and the notation $|X|$ to denote the Euclidean length of the path $X$.

Consider the directed line segment from $v_{src}$ to $v_{dst}$. This segment intersects regions of the Voronoi diagram in some order, say $R_0, \ldots, R_{m-1}$, where $R_0$ is the Voronoi region of $v_{src}$ and $R_{m-1}$ is the Voronoi region of $v_{dst}$. The *Voronoi routing* (VR) strategy for Delaunay triangulations moves the packet from $v_{src}$ to $v_{dst}$ along the path $v_0, \ldots, v_{m-1}$ where $v_i$ is the site defining $R_i$. An example of a path obtained by the Voronoi routing strategy is shown in Fig. 4. Since the Voronoi region of a vertex $v$ can be computed given only the neighbours of $v$ in the Delaunay triangulation, it follows that the Voronoi routing strategy is an $O(1)$ memory routing strategy.

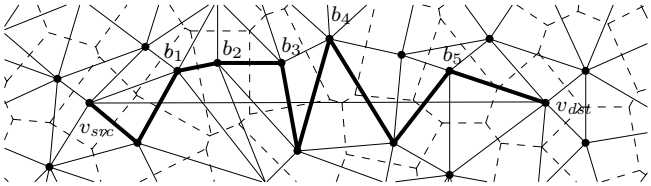



**Fig. 4.** A path obtained by the Voronoi routing strategy.

The Voronoi routing strategy on its own is not $c$-competitive for all Delaunay triangulations. However, it does have some properties that allow us to derive a $c$-competitive strategy. As with right-hand routing, let $T'$ be the graph obtained from $T$ by removing all edges of $T$ that properly intersect the segment $(v_{src}, v_{dst})$, and let $F$ be the face of $T'$ that contains both $v_{src}$ and $v_{dst}$. Assume wlog that $v_{src}$ and $v_{dst}$ both lie on the $x$-axis and that $\mathrm{x}(v_{src}) < \mathrm{x}(v_{dst})$. The following results follow from the work of Dobkin *et al* [6].

**Lemma 3** *The Voronoi path is $x$-monotone, i.e., $\mathrm{x}(v_i) < \mathrm{x}(v_j)$ for all $i < j$.*

**Lemma 4** *Let $P'$ be the collection of maximal subpaths of $v_0, \ldots, v_{m-1}$ that remain above the $x$-axis, i.e., $P' = \{v_i, \ldots, v_j : \mathrm{y}(v_{i-1}) < 0$ and $\mathrm{y}(v_{j+1}) < 0$ and $\mathrm{y}(v_k) \geq 0$ for all $i \leq k \leq j\}$. Then $\sum_{X \in P'} |X| \leq (\pi/2) dist(v_{src}, v_{dst})$.*

**Lemma 5** *Select any $i$ such that $y(v_i) \geq 0$ and $\mathrm{y}(v_{i+1}) < 0$. Let $j > i$ be the least value such that $\mathrm{y}(v_j) \geq 0$ and let $c_{dfs} = (1+\sqrt{5})\frac{\pi}{2}$.[2] Then the length of the*

[2] We call $c_{dfs}$ the Dobkin, Friedman and Supowit constant [6].

*path* $v_i, \ldots, v_j$ *is at most* $c_{dfs}(\mathrm{x}(v_j) - \mathrm{x}(v_i))$ *or the length of the upper boundary of* $F$ *between* $v_i$ *and* $v_j$ *is at most* $c_{dfs}(\mathrm{x}(v_j) - \mathrm{x}(v_i))$.

Our $c$-competitive routing strategy will visit the subsequence $b_0, \ldots, b_{l-1}$ of vertices $v_0, \ldots, v_{m-1}$ that are above or on the segment $(v_{src}, v_{dst})$. (Refer to Fig. 4.) If $b_i$ and $b_{i+1}$ are consecutive on the Voronoi path (i.e., $b_i = v_j$ and $b_{i+1} = v_{j+1}$ for some $j$) then our strategy will use the Voronoi path (i.e, the direct edge) from $b_i$ to $b_{i+1}$. On the other hand, if $b_i$ and $b_{i+1}$ are not consecutive on the Voronoi path then by Lemma 5, there exists a path from $b_i$ to $b_{i+1}$ of length at most $c_{dfs}(\mathrm{x}(b_{i+1}) - \mathrm{x}(b_i))$.

The difficulty occurs because the strategy does not know beforehand which path to take. The solution is to simulate exploring both paths "in parallel" and stopping when the first one reaches $b_{i+1}$.[3]

More formally, let $P_V = (p_0 = b_i, \ldots, p_n = b_{i+1})$ be the Voronoi path from $b_i$ to $b_{i+1}$ and let $P_F = (q_0 = b_i, \ldots, q_n = b_{i+1})$ be the path from $b_i$ to $b_{i+1}$ on the upper boundary of $F$. The strategy is described by the following algorithm.

1: $j \leftarrow 0$, $l_0 \leftarrow \min\{dist(p_0, p_1), dist(q_0, q_1)\}$.
2: **repeat**
3: Explore $P_F$ until reaching $b_{i+1}$ or until reaching a vertex $q_x$ such that $|q_0, \ldots, q_{x+1}| > 2l_j$. If $b_{i+1}$ is reached quit, otherwise return to $b_i$.
4: $j \leftarrow j + 1$, $l_j \leftarrow |q_0, \ldots, q_{y+1}|$.
5: Explore $P_V$ until reaching $b_{i+1}$ or until reaching a vertex $p_y$ such that $|p_0, \ldots, p_{y+1}| > 2l_j$. If $b_{i+1}$ is reached then quit, otherwise return to $b_i$.
6: $j \leftarrow j + 1$, $l_j \leftarrow |p_0, \ldots, p_{y+1}|$.
7: **until** $b_{i+1}$ is reached

**Lemma 6** *Using the parallel search strategy described above, a packet reaches* $b_{i+1}$ *after traveling a distance of at most* $9c_{dfs}(\mathrm{x}(b_{i+1}) - \mathrm{x}(b_i)) \sim 45.75(\mathrm{x}(b_{i+1}) - \mathrm{x}(b_i))$.

Given the positions of $v_{dst}$ and $v_{src}$ the parallel search strategy described above is easily implemented as part of an $O(1)$ memory routing strategy. We refer to the combination of the Voronoi routing strategy with the parallel search strategy described above as the *parallel Voronoi routing* (PVR) strategy.

**Theorem 6** *The parallel Voronoi routing strategy is* $(9c_{dfs} + \pi/2)$*-competitive for all Delaunay triangulations.*

*Proof.* The strategy incurs two costs: (1) the cost of traveling on subpaths of the Voronoi path that remain above the $y$-axis, and (2) the cost of applications of the parallel search strategy. By Lemma 4, the first cost is at most $(\pi/2) \cdot dist(v_{src}, v_{dst})$. By Lemma 6 and the fact that $b_0, \ldots, b_{l-1}$ is $x$-monotone (Lemma 3), the cost of the second is at most $9c_{dfs} \cdot dist(v_{src}, v_{dst})$. Since the Euclidean distance between two vertices of $T$ is certainly a lower bound on their shortest path distance in $T$ the theorem follows. □

[3] A similar strategy for finding an unknown target point on a line is given by Baeza-Yates *et al* [1]. See also Klein [8].

### 3.3 Empirical Results

While it is sometimes possible to come up with pathological examples of triangulations for which a strategy is not competitive, it is often more reasonable to use the competitive ratio of a strategy on average or random inputs as an indicator of how it will perform in practice. In this section we describe some experimental results about the competitiveness of our strategies. All experiments were performed on sets of points randomly distributed in the unit square, and each data point is the maximum of 50 independent trials.

The first set of experiments, shown in Fig. 5 (a), involved measuring the performance of all six routing strategies on Delaunay triangulations. Compass routing, greedy routing, and Voronoi routing consistently achieve better competitive ratios, with greedy routing slightly worse than the other two. Randomized

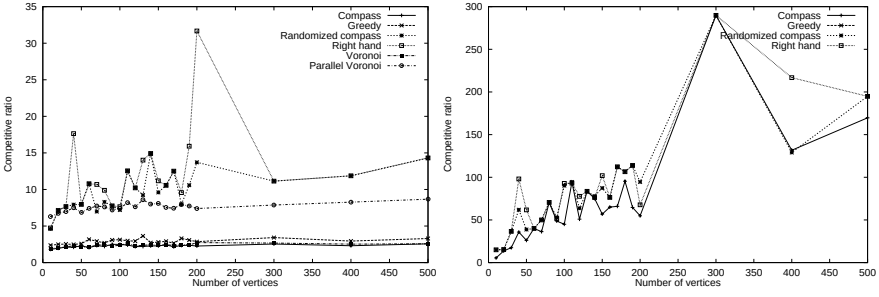



**Fig. 5.** Empirical competitive ratios for (a) Delaunay triangulations and (b) Graham triangulations.

compass routing, right-hand routing, and parallel Voronoi routing had significantly higher competitive ratios. The results for randomized compass routing and right-hand routing show a significant amount of jitter. This is due to the fact that relatively simple configurations[4] that can easily occur in random point sets, result in high competitive ratios for these strategies. On the other hand, parallel Voronoi routing seems much more stable, and achieves better competitive ratios in practice than its worst case analysis would indicate.

The most important conclusion drawn from these experiments is that there are no simple configurations (i.e., that occur often in random point sets) that result in extremely high competitive ratios for greedy, compass, Voronoi, or parallel Voronoi routing in Delaunay triangulations. This suggests that any of these strategies would work well in practice.

The four simple routing strategies of Section 2 were also tested on Graham triangulations. These are obtained by first sorting the points by $x$-coordinate and then triangulating the resulting monotone chain using a linear time algorithm for computing the convex hull of a monotone polygonal chain [12]. The

[4] These configurations come up in the proof of Theorem 5 [3].

results are shown in Fig. 5 (b). In these tests it was always the case that at least one of the 50 independent triangulations defeated greedy routing. Thus, there are no results shown for greedy routing. The relative performance of the compass, randomized compass, and right hand routing strategies was the same as for Delaunay triangulations. However, unlike the results for Delaunay triangulations, the competitive ratio appears to be increasing linearly with the number of vertices.

## 4 Comparison with Related Work

In this section we survey related work in the area of geometric online routing, and compare our results with this work. Due to space constraints, we can not provide a comprehensive list of references in this version of the paper. A more complete survey is available in the full version of the paper [3].

Kranakis *et al* [9] study compass routing, and provide a proof that no Delaunay triangulation defeats compass routing. The current paper makes use of a very different proof technique to show that compass routing works for a larger class of triangulations. They also describes an $O(1)$ memory routing strategy that is not defeated by any connected planar graph, thus proving a stronger result than Theorem 4.

Lin and Stojmenović [10] and Bose *et al* [4] consider online routing in the context of *ad hoc* wireless networks modeled by unit disk graphs. They provide simulation results for a variety of strategies that measuring success rates (how often a packet never reaches its destination) as well as hop-counts of these strategies on unit graphs of random point sets.

To the best of our knowledge, no literature currently exists on the competitiveness of geometric routing strategies in our abstract setting, and our parallel Voronoi routing strategy is the first theoretical result in this area.

## 5 Conclusions

We have studied the problem of online routing in geometric graphs. Our theoretical results show which types of graphs our strategies are guaranteed to work on, while our simulation results rank the performance of the strategies on two types of random triangulations. These results are summarized in the following table.

| Strategy | Memory | Randomized | Class of graphs | Rank 1 | Rank 2 | Competitive |
|---|---|---|---|---|---|---|
| GR | None | No | Delaunay △'s | 3 | – | No |
| CR | None | No | Regular △'s | 1 | 1 | No |
| RCR | None | Yes | All △'s | 5 | 2 | No |
| RHR | $O(1)$ | No | Convex subd. | 6 | 3 | No |
| VR | $O(1)$ | No | Delaunay △'s | 1 | – | No |
| PVR | $O(1)$ | No | Delaunay △'s | 4 | – | Yes |

## Acknowledgement

The authors would like to thank Silvia Götz for reading and commenting on an earlier version of this paper.

## References

[1] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier North-Holland, 1976.

[3] P. Bose and P. Morin. Online routing in triangulations. Available online at http://www.scs.carleton.ca/ morin, 1999.

[4] P. Bose, P. Morin, I. Stojmenović, and J. Urrutia. Routing with guaranteed delivery in *ad hoc* wireless networks. In *Proceedings of Discrete Algorithms and Methods for Mobility (DIALM'99)*, 1999. To appear.

[5] V. Chvátal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory B*, 18:39–41, 1975.

[6] D. Dobkin, S. J. Friedman, and K. J. Supowit. Delaunay graphs are almost as good as complete graphs. *Discrete and Computational Geometry*, 5:399–407, 1990. See also *28th Symp. Found. Comp. Sci.*, 1987, pp. 20–26.

[7] H. Edelsbrunner. An acyclicity theorem for cell complexes in $d$ dimension. *Combinatorica*, 10(3):251–260, 1988.

[8] R. Klein. Walking an unknown street with bounded detour. *Computational Geometry Theory and Applications*, 1:325–351, 1992.

[9] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry (CCCG'99)*, 1999. To appear.

[10] X. Lin and I. Stojmenović. Geographic distance routing in ad hoc wireless networks. Technical Report TR-98-10, SITE, University of Ottawa, December 1998.

[11] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tesselations: Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons, 1992.

[12] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[13] Günter M. Ziegler. *Lectures on Polytopes*. Number 154 in Graduate Texts in Mathematics. Springer-Verlag, New York, 1994.

# The Query Complexity of Program Checking by Constant-Depth Circuits

V. Arvind[1], K.V. Subrahmanyam[2], and N.V. Vinodchandran*[3]

[1] Institute of Mathematical Sciences, Chennai 600 113, India
arvind@imsc.ernet.in
[2] SPIC Mathematical Institute, Chennai 600017, India
kv@smi.ernet.in
[3] BRICS, Aarhus, Denmark.
vinod@daimi.aau.dk

**Abstract.** We study program result checking using $AC^0$ circuits as checkers. We focus on the number of queries made by the checker to the program being checked and we term this as the *query complexity* of the checker for the given problem. We study the query complexity of deterministic and randomized $AC^0$ checkers for certain P-complete and $NC^1$-complete problems. We show that for each $\epsilon > 0$, $\Omega(n^{1-\epsilon})$ is a lower bound to the query complexity of deterministic $AC^0$ checkers for the considered problems, for inputs of length $n$. On the other hand, we show that suitably encoded complete problems for P and $NC^1$ have randomized $AC^0$ checkers of *constant* query complexity. The latter results are proved using techniques from the PCP$(n^3, 1)$ protocol for 3-SAT in [4].

## 1 Introduction

In this paper we study program result checking (in the sense of Blum and Kannan [5]) with $AC^0$ circuits as checkers and we focus on the number of queries made by the checker to the checked program. We term this parameter as the *query complexity* of the checker for the given problem. The query complexity is an important parameter in the design of efficient program checkers because a large query complexity can be a serious bottleneck for a checker that may otherwise be efficient.

The seminal paper of Blum and Kannan [5] already initiates the study of parallel checkers. They give a deterministic CRCW PRAM constant-time (i.e. $AC^0$) program checker for the P-complete problem *LFMIS* (lex. first maximal independent set problem for graphs). Rubinfeld in [10] makes a comprehensive algorithmic study of parallel program checkers: parallel checkers are designed in [10] for various problems with emphasis on analyzing parallel time and processor efficiency. In particular, an $AC^0$ checker for the P-complete problem of evaluating straight-line programs is described in [10]. However, in the context of the present paper, we note that the above-mentioned $AC^0$ checkers for P-complete problems described in [5,10] have large query complexity (the number

* Part of the work done while at IMSc.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 123–132, 1999.
© Springer-Verlag Berlin Heidelberg 1999

of queries is proportional to the input size). Indeed the $AC^0$ checkers in [5,10] are deterministic, and as shown in the present paper deterministic $AC^0$ checkers for standard P-complete problems must necessarily have large query complexity.

Regarding query complexity, we note that constant query checkers are mentioned in [1] as a notion of practical significance. For instance, it is shown in [1] that GCD has a constant query checker. To the best of our knowledge, there is no other study or explicit mention of query complexity of program checkers.

Our motivation for studying $AC^0$ checkers is two-fold. First, in a complexity-theoretic sense $AC^0$ represents the easiest model of parallel computation, and one aspect of program checking is to try and make the checker as efficient as possible. In this sense $AC^0$ checkers can be seen as constant-time parallel checkers since they essentially correspond to constant-time CRCW PRAM algorithms. The second motivation rests on the main goal of this paper, namely, to study the query complexity of program checkers. It turns out that $AC^0$ yields a model of program checking that is amenable to lower bound techniques. The problems we consider are different suitably encoded versions of the *Circuit Value Problem* (henceforth CVP). Different versions of the CVP are known to be complete for different important complexity classes: the unrestricted version is P-complete and the CVP problem for circuits that are formulas (i.e. fanout of each gate is one) is $NC^1$-complete. In fact, using similar techniques we can also suitably encode an NL-complete problem that has a constant query randomized $AC^0$ checker.

We deduce nontrivial lower bounds on the query complexity of deterministic $AC^0$ checkers for these circuit value problems: we prove the lower bounds by first noting for the Parity problem that, by a result of Ajtai [2], $\Omega(n^{1-\epsilon})$ is a lower bound for the query complexity of deterministic $AC^0$ checkers for Parity. Since Parity is $AC^0$ many-one reducible to each considered circuit-value problem the same lower bounds on query complexity carry over. Thus, we get that for each $\epsilon > 0$, $\Omega(n^{1-\epsilon})$ is a lower bound to the query complexity of deterministic $AC^0$ checkers for these circuit-value problems.

In contrast, we design randomized $AC^0$ checkers of *constant* query complexity for these circuit-value problems. We outline the ideas involved: consider a deterministic $AC^0$ checker for the given circuit-value problem (e.g. the one described in [10] for general straight-line programs). The query complexity of this checker is roughly the number of gates in the input circuit. Our randomized *constant query* $AC^0$ checkers for the circuit-value problems use ideas from the $PCP(n^3, 1)$ protocol for satisfiability [4]. Intuitively, it turns out that the number of probes into an NP proof by a PCP protocol corresponds to the number of queries that the checker needs to make for a given instance of the circuit-value problem. A difficulty in the checker setting (which doesn't arise in the $PCP(n^3, 1)$ protocol) is that the $AC^0$ checker needs to compute unbounded GF(2) sums. It cannot directly do this because Parity is not computable in $AC^0$. But we can get around this difficulty by using Rubinfeld's parallel checker for Parity [10] which we note is already an $AC^0$ constant-query checker. Since Parity is $AC^0$ many-one reducible to the considered circuit-value problems, we can use the tested program

for CVP to compute Parity and use Rubinfeld's checker as subroutine to check that the returned answer is correct. Thus we are able to design a constant-query $AC^0$ checker for CVP.

As explained in [4], the PCP theorem has evolved from interactive proofs [7] and program checking [5,6]. In particular, there is a strong influence of ideas from self-correcting programs in the PCP$(n^3, 1)$ protocol for 3-SAT [4]. It is not surprising, therefore, that ingredients of the PCP$(n^3, 1)$ protocol find application in program checking. Our emphasis on the query complexity of program checkers leads naturally to ideas underlying probabilistically checkable proofs. More applications of ideas from the PCP theorem to other specific problems in program checking appears to be an attractive area worth exploring.

## 2 Preliminaries

We first formally define program checkers introduced in [5].

**Definition 1.** [5] *Let $A$ be a decision problem, a program checker for $A$, $C_A$, is a (probabilistic) oracle algorithm that for any program $P$ (supposedly for $A$) that halts on all instances, for any instance $x$ of $A$, and for any positive integer $k$ (the security parameter) presented in unary:*

1. *If $P$ is a correct program, that is, if $P(x) = A(x)$ for all instances $x$, then with probability 1, $C_A(x, P, k)$=Correct.*
2. *If $P(x) \neq A(x)$ then with probability $\geq 1 - 2^{-k}$, $C_A(x, P, k)$=Incorrect.*

*The probability is computed over the sequences of coin flips that $C_A$ could have tossed. Importantly, $C_A$ is allowed to make queries to the program $P$ on some instances.*

When we speak of $AC^0$ checkers we mean that the checker $C_A$ is described by a (uniform) family of $AC^0$ circuits, one for each input size. We will also consider the (stronger) notion of deterministic checkability. The decision problem $A$ is said to be *deterministically* checkable if $C_A$ in the above definition is a deterministic algorithm. Next we define the query complexity of $AC^0$ checkers.

**Definition 2.** *Let $L$ be a decision problem that is deterministically* $AC^0$ *checkable. The* $AC^0$ *checker defined by the circuit family $\{C_n\}_{n\geq 0}$ is said to have* query complexity $q(n)$ *if $q(n)$ bounds the number of queries made the checker circuit $C_n$ for any input $x \in \Sigma^n$.*

We now describe the CVP problems and the encodings of their instances. Let $C$ denote a boolean circuit over the standard base (of NOT, AND, and OR gates). We consider circuits of fanin bounded by two. We will encode the circuit $C$ as 4-tuples $(g_1, g_2, g_3, t)$ where $t$ is a constant number of bits to indicate the type of the gate labeled $g_1$, and $g_2$ and $g_3$ are the gates whose values feed into the gate labeled $g_1$. For uniformity, we can assume that NOT gates are also encoded as such 4-tuples, except that $g_2 = g_3$. Furthermore, we insist that

in the encoding, the gate labels $g_i$ be *topologically sorted* consistent with the *DAG* underlying the circuit $C$. Thus, in each 4-tuple $(g_1, g_2, g_3, t)$ present in the encoding of $C$, it will hold that $g_1 > g_2$ and $g_1 > g_3$. This stipulation ensures that checking whether an encoding indeed represents a circuit can be done in $\mathrm{AC}^0$. For a circuit $C$ with $n$ inputs let $C_g(x_1, x_2, \ldots, x_n)$ denote the value of the circuit $C$ at gate $g$ designated as output gate. We now define the *circuit value problem* which is the decision problem that we shall be mainly concerned with in this paper: $\{(C, g, x_1, x_2, \ldots, x_n) \mid C_g(x_1, x_2, \ldots, x_n) = 1\}$. The circuit $C$ is encoded as described above.

The above circuit value problem is known to be P-complete (under projection reducibility). We denote this by CVP. If the input circuit is a formula (i.e. each gate of the input circuit has fanout at most 1) the corresponding circuit value problem is known to be $\mathrm{NC}^1$-complete under projection reducibility; we denote this problem by FVP (for formula value problem). Notice that an $\mathrm{AC}^0$ circuit can check if a given circuit is a formula or not. Finally, we make another important stipulation on the circuits that are valid inputs for all the circuit value problems that we consider: we insist that the *fanout* of each gate is bounded by two. Notice that this last restriction on the input circuits does not affect the fact that CVP remains P-complete (in fact, such a restriction already holds for the CVP in the standard P-completeness proof by simulating polynomial-time Turing machines). Also, observe that this extra stipulation on the input circuits can be easily tested in $\mathrm{AC}^0$.

## 3 Deterministic $\mathrm{AC}^0$ checkers

We first recall deterministic $\mathrm{AC}^0$ checkers for Parity and the circuit value problem [5,10].

**Proposition 1.** *For each constant $k$, there is a deterministic* $\mathrm{AC}^0$ *checker for* $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$ *of query complexity* $n/\log^k n$.

*Proof.* Let $P$ be an alleged program for the Parity function. First observe that the $\mathrm{AC}^0$ checker can make parallel queries to $P$ for $\mathrm{Parity}(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n$. In order to verify that the program's value of $\mathrm{Parity}(x_1, x_2, \ldots, x_n)$ is correct the checker just has to verify that the answers to the queries for $\mathrm{Parity}(x_1, x_2, \ldots, x_i)$ are all locally consistent: $P(x_1, x_2, \ldots, x_{i+1}) = x_{i+1} \oplus P(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n-1$. The above verification can be easily done in parallel in $\mathrm{AC}^0$ since query answers $P(x_1, x_2, \ldots, x_i)$ for $2 \leq i \leq n$ are available. This yields an $\mathrm{AC}^0$ checker which makes $n-1$ queries. In order to design a checker with the number of queries scaled down to $n/\log^k n$ notice that we can compute the parity of $\log n$ boolean variables in $\mathrm{AC}^0$ by brute force. Thus, we can group the $n$ input variables into $n/\log n$ groups of $\log n$ variables each, compute the parity of each group again by brute force, and the problem boils down to checking the program's correctness for the parity of $n/\log n$ variables which we can do as before with $n/\log n$ queries to the program. Clearly, we can repeat the above strategy of grouping variables for a constant number of rounds, and

therefore achieve the query complexity of $n/\log^k n$ with an appropriate constant increase of depth of the checker circuit. This completes the proof.

**Remark**: Notice that the above result applies to checking iterated products over arbitrary finite monoids. The proof and construction of the checker is similar.

Indeed, the above idea of doing piece-wise "local testing" in order to check global correctness is already used in the deterministic CRCW PRAM checker in [5] for an encoding of the P-complete problem *LFMIS*, and also in [10] for checking straight-line programs. Notice that the checker for general straight-line programs includes checking the circuit-value problem as a special case. We include a proof sketch below since it is the starting point for the main results of this paper.

**Theorem 1.** [10] CVP *has a deterministic* $\mathrm{AC}^0$ *checker.*

*Proof.* Let $P$ be an alleged program for CVP and let $(C, g, x_1, \ldots, x_n)$ be an input instance for program $P$. The $\mathrm{AC}^0$ checker first queries in parallel the program for $P(C, g, x_1, \ldots, x_n)$ $\forall$ gates $g \in C$ Next, for each tuple $(g_1, g_2, g_3, t)$ in the circuit description $C$ the checker verifies that the program's answers are consistent with the gate type. This is again done in parallel for each tuple. The checker must also validate the input by verifying that the tuples that describe the circuit indeed describe an acyclic digraph. This is made sure as described in our encoding of the instances of the CVP. It suffices to check that $g_1 > g_2$ and $g_1 > g_3$ for each tuple $(g_1, g_2, g_3, t)$, which can be done in $\mathrm{AC}^0$. This completes the proof.

It can be shown similarly that FVP has a deterministic $\mathrm{AC}^0$ checker. We now turn to lower bounds on the query complexity of $\mathrm{AC}^0$ checkers for CVP and FVP. We first observe the following property of languages $L$ having deterministic $\mathrm{AC}^0$ checkers.

**Lemma 1.** *Let $L$ be a decision problem that is deterministically* $\mathrm{AC}^0$ *checkable and has an* $\mathrm{AC}^0$ *checker of query complexity $q(n)$. Then, for each $n > 0$, there is a nondeterministic* $\mathrm{AC}^0$ *circuit that takes $n$ input bits and $q(n)$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff $x \in L^{=n}$.*

Observe that, by symmetry, such nondeterministic $\mathrm{AC}^0$ circuits also exist for $\overline{L}$. The proof of the above lemma is a direct consequence of the definition of deterministic checkers and is a variation of a result on self-helping due to Schöning [11]: as observed e.g. in [3], polynomial-time deterministic checking coincides with self-helping defined by Schöning [11] who showed that languages that have self-helpers are already in NP∩co-NP. The above lemma is an extension of this fact to the setting where the checker is in $\mathrm{AC}^0$. The only extra observation made in Lemma 1 is that the number of queries made by the checker naturally translates into the number of nondeterministic bits used by the nondeterministic circuit. We next recall a result due to Ajtai [2] on lower bounds for $\mathrm{AC}^0$ circuits approximating Parity.

**Theorem 2.** [2] *For all constants $k, c$, and $\epsilon > 0$, there is no depth $k$ circuit of size $n^c$ that can compute* Parity$(x_1, x_2, \ldots, x_n)$ *for more than a $1/2 + 2^{-n^{1-\epsilon}}$ fraction of the inputs. Thus, no nondeterministic* $\mathrm{AC}^0$ *circuit with $O(n^{1-\epsilon})$ nondeterministic bits can compute* Parity$(x_1, x_2, \ldots, x_n)$.

**Lemma 2.** Parity *does not have deterministic* $\mathrm{AC}^0$ *checkers that make $O(n^{1-\epsilon})$ queries for any $\epsilon > 0$.*

*Proof.* Assume that Parity has $\mathrm{AC}^0$ checkers that makes $O(n^{1-\epsilon})$ queries for some $\epsilon > 0$. From Lemma 1 it follows that for each $n > 0$, there is a nondeterministic $\mathrm{AC}^0$ circuit that takes $n$ input bits and $O(n^{1-\epsilon})$ nondeterministic bits and accepts $x \in \Sigma^n$ iff $x$ has odd parity. This is impossible since it contradicts Theorem 2 of Ajtai.

**Theorem 3.** CVP *(likewise* FVP*) does not have deterministic* $\mathrm{AC}^0$ *checkers that make $O(n^{1-\epsilon})$ queries for any $\epsilon > 0$.*

*Proof.* We prove it just for CVP. Notice that we can design an $\mathrm{AC}^0$ circuit (call it $C'$) such that given an instance $x \in \Sigma^n$ of Parity the $\mathrm{AC}^0$ circuit produces an instance $(C, x_1, x_2, \ldots, x_n, g)$ of CVP such that Parity$(x_1, x_2, \ldots, x_n) = 1$ iff it holds that $(C, x_1, x_2, \ldots, x_n, g) \in$ CVP. Moreover, the size of $(C, x_1, x_2, \ldots, x_n, g)$ is $O(n \log n)$, since $C$ encodes the linear-sized circuit for Parity in the 4-tuple encoding we are using for CVP instances. Assume that CVP has an $\mathrm{AC}^0$ checker that makes $O(n^{1-\epsilon})$ queries for some $\epsilon > 0$. Combining the nondeterministic circuit given by Lemma 1 with the $\mathrm{AC}^0$ circuit $C'$, we get a nondeterministic $\mathrm{AC}^0$ circuit that takes $n$ input bits and $O(n^{1-\delta})$ nondeterministic bits and accepts an input $x \in \Sigma^n$ iff $x$ has odd parity, for some suitable $\delta > 0$. This contradicts Lemma 2 and hence completes the proof.

## 4 A constant query randomized $\mathrm{AC}^0$ checker for CVP

We first recall the relevant definition and results from [6] concerning the linearity test.

**Definition 3.** [6] *Let $F$ be* GF(2) *and $f, g$ be functions from $F^n$ to $F$. The relative distance $\Delta(f, g)$ between $f$ and $g$ is the fraction of points in $F^n$ on which they disagree. If $\Delta(f, g) \leq \delta$ then $f$ is said to be $\delta$-close to $g$.*

**Theorem 4.** [6] *Let $F$ be* GF(2) *and $f$ be a function from $F^n$ to $F$ such that when we pick $y, z$ randomly from $F^n$, Prob$[f(y) + f(z) = f(y + z)] \geq 1 - \delta$, where $\delta < 1/6$. Then $f$ is $3\delta$-close to some linear function.*

The theorem gives a linearity test that needs to evaluate $f$ at only a constant number of points in $F^n$, where the constant depends on $\delta$. If $f$ passes the test then the function is guaranteed to be $3\delta$-close to some linear function. Given a

function $f$ guaranteed to be $\delta$-close to a linear function $\hat{f}$, and $x$ in the domain of $f$, let us denote by SC-$f(x)$ the value $f(x+r)-f(r)$, for a randomly chosen $r$ in the domain of $f$. Then the above theorem guarantees that with high probability SC-$f(x)$ is equal to $\hat{f}(x)$.

Next we recall another lemma from [4] (as stated in [9, Lemma 7.13]). Given $\boldsymbol{a}$ in $\mathrm{GF}(2)^n$, the *outer product* $\boldsymbol{b} = \boldsymbol{a} \otimes \boldsymbol{a}$ is an $n \times n$ matrix over GF(2) such that $\boldsymbol{b}_{ij} = \boldsymbol{a}_i \boldsymbol{a}_j$.

**Lemma 3.** [4] *Let* $\boldsymbol{a} \in \mathrm{GF}(2)^n$ *and* $\boldsymbol{b}$ *be an* $n \times n$ *matrix over* GF(2)*. Suppose* $\boldsymbol{b} \neq \boldsymbol{a} \otimes \boldsymbol{a}$*, then* $\mathrm{Prob}[\boldsymbol{r}^t(\boldsymbol{a} \otimes \boldsymbol{a})\boldsymbol{s} \neq \boldsymbol{r}^t \boldsymbol{b} \boldsymbol{s}] \geq 1/4$ *where* $\boldsymbol{r}$ *and* $\boldsymbol{s}$ *are randomly chosen from* $\mathrm{GF}(2)^n$*.*

A randomized $\mathrm{AC}^0$ checker for Parity is described in [10] based on the fact that Parity is random self-reducible. Additionally, we observe that this checker has constant query complexity and we will use it as subroutine in the checker for CVP.

**Theorem 5.** [10] Parity *has a randomized* $\mathrm{AC}^0$ *checker of constant query complexity.*

We are now ready to design the constant query checker for the CVP problem (also for FVP). We make use of ideas in the $\mathrm{PCP}(n^3, 1)$ protocol for 3-SAT from [4]. A crucial point of departure from [4] is when the checker needs to compute the parity of a multiset of input variables and a product of input variables. To do this we use as subroutine the checker of Theorem 5. Another point to note is that all queries have to be valid instances of CVP (or FVP as the case may be), and they need to be generated in $\mathrm{AC}^0$. The starting point is the deterministic $\mathrm{AC}^0$ checker for CVP described in Theorem 1. Recall that given instance $(C, g, x_1, \ldots, x_n)$ the deterministic $\mathrm{AC}^0$ checker queries the program for $(C, g_i, x_1, \ldots, x_n)$, for each gate $g_i$ of $C$. Then it checks that the query answers are locally consistent for each gate. Let $y_1, y_2, \ldots, y_m$ be the query answers by $P$ for the queries $(C, g_i, x_1, \ldots, x_n)$, $1 \leq i \leq m$, where $C$ has $m$ gates. The *unique correct* vector $y_1, y_2, \ldots, y_m$ is a satisfying assignment to the collection of all the gate conditions (each of which is essentially a 3-literal formula). The idea is to avoid querying explicitly for $y_i$'s. Instead, using randomness the checker will make fewer queries for other inputs that encode the $y_i$'s. More precisely, we need to encode $y_1, y_2, \ldots, y_m$ in a way that making a constant number of queries to the program (which is similar to a constant number of probes into a proof by a PCP protocol) can convince the $\mathrm{AC}^0$ checker with high probability that $y_1, y_2, \ldots, y_m$ is consistent with all the gate conditions.

**Theorem 6.** *The* P*-complete problem* CVP *(likewise the* $\mathrm{NC}^1$*-complete problem* FVP*) has a randomized* $\mathrm{AC}^0$ *checker of constant query complexity.*

*Proof.* We describe the checker only for CVP (the checker for FVP is similar). Let $P$ be a program for CVP and $(C, g, x_1, \ldots, x_n)$ be an input. Suppose C has $m$ gates $g_1, \ldots, g_m$ w.l.o.g. assume $g_m = g$. The deterministic checker of Theorem 1 queries $P$ for $(C, g_i, x_1, \ldots, x_n)$, for each $g_i$. It then performs

a local consistency check to test the validity of $(C, g_m, x_1, \ldots, x_n)$. The new checker must avoid querying explicitly for each $y_i := (C, g_i, x_1, \ldots, x_n)$. Let $t_i$ denote a GF(2) polynomial corresponding to the $i$th gate of the circuit $C$, where $t_i$ is a polynomial in at most three variables (these three variables are from $\{x_1, \ldots, x_n\} \cup \{y_1, y_2, \ldots, y_m\}$). Define $t_i$ such that it is zero iff the corresponding variables are consistent with the gate type of $g_i$. More precisely, let $g$ be an AND gate with output $a$ and inputs $b$ and $c$ then the polynomial corresponding to $g$ is $a + bc$. Similarly, for an OR gate with output $a$ and inputs $b$ and $c$ the polynomial is $a + b + c + bc$, and for a NOT gate with output $a$ and input $b$ it is $a + b + 1$.

**The checker**

The checker first queries the program on input $(C, g, x_1, \ldots, x_n)$ and sets $y_m = P(C, g, x_1, \ldots, x_n)$. It suffices for the $AC^0$ checker to verify that the following linear function $f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) = \Sigma_{i=1}^{m} t_i(\boldsymbol{x}, \boldsymbol{y}) z_i$ in the new variables $z_i, 1 \leq i \leq m$ is the zero linear function.

Notice that if this function is nonzero then the linear function $f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z})$ evaluated at a randomly chosen $\boldsymbol{z} := \langle z_1, \ldots, z_m \rangle$ would be nonzero with probability 1/2. If it were the zero function then it must be zero with probability 1.

Recall that the checker must compute this value by asking a series of CVP queries that are generated in $AC^0$. Towards this end, we rewrite the above expression: $f(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) = p(\boldsymbol{x}, \boldsymbol{z}) + q(\boldsymbol{y}, \boldsymbol{z}) + r(\boldsymbol{y}, \boldsymbol{z})$, where $p(\boldsymbol{x}, \boldsymbol{z}) = \Sigma_{i=1}^{n} \Sigma_{k=1}^{m} \chi_i^k x_i + \Sigma_{i=1}^{n} \Sigma_{j=1}^{n} \Sigma_{k=1}^{m} \chi_{ij}^k x_i x_j$, $q(\boldsymbol{y}, \boldsymbol{z}) = \Sigma_{k=1}^{m} c_i * y_i$, and $r(\boldsymbol{y}, \boldsymbol{z}) = \Sigma_{(i,j) \in [m] \times [m]} c_{ij} * y_i y_j$.

In the above expression $\chi_i^k$ is 1 iff $x_i$ appears in the polynomial $p_k$ and if $z_k$ is 1. Likewise $\chi_{ij}^k$, is 1 iff $x_i x_j$ appears in the polynomial $p_k$ and $z_k$ is 1. From this it follows that an $nm + n^2 m$ length Boolean vector representing each term in $p$ can be obtained in $AC^0$. Notice that coefficients $c_i$ and $c_{ij}$ in $q$ and $r$ depend upon gates $g_i$ and $g_j$, the constant number of gates they feed into, the $z$ values corresponding to these gates and a constant number of input bits. So computing each of these coefficients involves computing the parity of a *constant* number of Boolean variables. This can also be done in $AC^0$.[1]

We describe below how the checker computes $\tilde{p}$, $\tilde{q}$, and $\tilde{r}$ with high probability. To complete the checking the checker evaluates $\tilde{p} + \tilde{q} + \tilde{r}$ and accepts $P(x)$ as correct only if this sum is zero.

**Computing** $p$ Note that $p$ is the parity of $nm + n^2 m$ Boolean variables. As noted above the value of these variables can be obtained in $AC^0$ given $\boldsymbol{x}$ and $\boldsymbol{z}$. The checker constructs a description of a canonical circuit for the parity of $nm + n^2 m$ variables, and queries the program on this input. Next the $AC^0$ checker checks the answer of $P$ using the checker of Theorem 5 as subroutine. If the answer is wrong then with high probability the subroutine checker will reject the program as incorrect. Thus the $AC^0$ checker computes a value $\tilde{p}$ which is $p$ with high (constant) probability. In the process only a constant number of queries are made to $P$.

Notice that, unlike in [4], we have to deal with both $y_1, \ldots, y_m$ as well as $x_1, x_2, \ldots, x_n$ which occur in the polynomials $t_i$. The crucial difference between

[1] It is easier to first conceive of a constant time CRCW PRAM algorithm for this task.

the $x_j$'s and $y_j$'s is that $x_1, x_2, \ldots, x_n$ are *bound* to the input values. Thus, computing the value of $p$ is a *parity computation* which the checker requires to get done. As explained above, this is done using the checker of Theorem 5 as subroutine.

**Computing $q$ and $r$** To compute $q$ and $r$ the checker goes through the following steps.

1. It builds a circuit $C_1$ with new inputs $r_1, r_2, \ldots, r_m$ that computes the function $\Sigma_{i=1}^{m} y_i r_i$. Recall that $y_i$ is the output of the $i$th gate of the input circuit $C$ on input $x_1, x_2, \ldots, x_n$. Clearly, the encoding of $C_1$ can be generated by an $\mathrm{AC}^0$ circuit from the encoding of $C$.
2. Similar to the above step, the checker builds a circuit $C_2$ with new inputs $r_{ij}, 1 \leq i, j \leq m$ that computes $\Sigma_{i=1}^{m} \Sigma_{j=1}^{m} y_i y_j r_{ij}$. It is clear that an encoding for $C_2$ can also be generated by an $\mathrm{AC}^0$ circuit.
3. The checker verifies that the program's behavior on $C_1$ is a function that is $\delta$-close to a linear function in the variables $r_i$, and the program's behavior on $C_2$ is a function that is $\delta$-close to a linear function in the variables $r_{ij}$. This can be done as described in the previous section using Theorem 4. If either of the tests fails, it rejects the program as being incorrect.
4. Like in the PCP protocol the checker now performs a consistency check. Let $\Sigma_{i=1}^{m} \Sigma_{j=1}^{m} b_{ij} r_{ij}$ be the linear function to which $C_2$ is $\delta$-close. The checker does a constant query test, and ensures with high probability that the matrix $b_{ij}$ is the tensor product of $\boldsymbol{y}$ with itself.
   To do this the checker employs the test given by Lemma 3. For two randomly random vectors $r_1, r_2$ of length $m$ it verifies that SC-$C_1(r_1) * $ SC-$C_1(r_2) =$ SC-$C_2(r_1 \otimes r_2)$
   Note that the tensor product can be computed in $\mathrm{AC}^0$ and the checker needs to ask 6 queries of $P$.

Having performed the linearity and consistency tests the checker evaluates $q$ and $r$ by self-correction. Let $\boldsymbol{c}$ denote the $m$-vector $c_1, c_2, \ldots, c_m$ and let $\boldsymbol{d}$ denote the $m \times m$-vector consisting of $c_{ij}, 1 \leq i, j \leq m$. The checker sets $\tilde{q}$ to SC-$C_1(\boldsymbol{c})$ and $\tilde{r}$ to SC-$C_2(\boldsymbol{d})$.

**Correctness.**

If $P$ is correct for all inputs then with probability 1 the checker will pass $P$ as correct. Suppose that $P$ is incorrect on $(C, g, x_1, \ldots, x_n)$ and let $P(C, g, x_1, \ldots, x_n) = b$. Let $F$ be the event that the checker fails to detect the program as incorrect. Let $T$ be the event that the checker passes the linearity and consistency tests done in the course of computing $q$ and $r$. Let $w$ dennote the concatenation of all random strings used by the checker. Notice that it suffices to bound $\mathrm{Prob}_w[F \mid T]$ as $\mathrm{Prob}_w[F] \leq \mathrm{Prob}_w[F \mid T]$. Therefore, we can assume that $C_1$ is $\delta$-close to a unique linear function of the $r_i$'s, $\Sigma_{i=1}^{m} y_i r_i$, wherein $y_m = b$. Likewise $C_2$ computes a function $\delta$-close to the linear function $\Sigma_{(i,j)\in[m]\times[m]} y_i y_j * c_{ij}$. Let $\hat{y} = \langle y_1, \ldots, y_m \rangle$ be this unique linear function. Given that $P(x)$ is incorrect, the function $f(\boldsymbol{x}\hat{y}, \boldsymbol{z}) = \Sigma_{i=1}^{m} t_i(\boldsymbol{x}, \hat{y}) z_i$ is a nonzero linear function of the $z_i$'s. Hence, $\mathrm{Prob}_w[f(\boldsymbol{x}, \hat{y}, \boldsymbol{z}) = 1 \mid T] = \frac{1}{2}$. Now,

$$\begin{aligned}\text{Prob}_w[F \mid T] &= \text{Prob}_w[F \ \&\ f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 1 \mid T] + \text{Prob}_w[F \ \&\ f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 0 \mid T] \\ &\leq \text{Prob}_w[F \ \&\ f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 1 \mid T] + 1/2 \\ &= \text{Prob}_w[f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 1 \mid T] * \text{Prob}_w[F \mid f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 1 \ \&\ T] + 1/2 \\ &= 1/2 * \text{Prob}_w[F \mid f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 1 \ \&\ T] + 1/2\end{aligned}$$

Since $f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = p(\boldsymbol{x},\boldsymbol{z}) + q(\hat{y},\boldsymbol{z}) + r(\hat{y},\boldsymbol{z})$ and $F$ is the event that $\tilde{p}+\tilde{q}+\tilde{r} = 0$, we observe: $\text{Prob}_w[F \mid f(\boldsymbol{x},\hat{y},\boldsymbol{z}) = 1 \ \&\ T] = \text{Prob}_w[\tilde{p} \neq p(\boldsymbol{x},z) \mid T] + \text{Prob}_w[\tilde{q} \neq q(\hat{y},z) \mid T] + \text{Prob}_w[\tilde{r} \neq r(\hat{y},z) \mid T]$.

From Theorem 5 the first term is bounded by $3/4$. Each of the other two terms is bounded by $2\delta$. Putting it together we get, $\text{Prob}_w[F \mid T] \leq 1/2*(3/4+4\delta)+1/2$. This is smaller than $15/16$ if we choose $\delta$ smaller than $1/32$.

Since the error in the checker is one-sided, we can easily make the error probability an arbitrarily small constant by repeating the checker a constant number of times in parallel and rejecting the program if in one such repetition the checker rejects. This completes the proof.

**Acknowledgments.** We thank Peter Bro Miltersen, Jaikumar Radhakrishnan, and S. Venkatesh for interesting discussions on $AC^0$ lower bounds for parity. For useful remarks we are grateful to Samik Sengupta and D. Sivakumar.

## References

1. L. A. Adleman, H. Huang, K. Kompella, Efficient checkers for number-theoretic computations, *Information and Computation,* **121**, 93–102, 1995.
2. M. Ajtai, $\Sigma_1^1$ formulas on finite structures, *Annals of Pure and Applied Logic,* **24**, (1983) 1-48.
3. V. Arvind, Constructivizing membership proofs in complexity classes, *International Journal of Foundations of Computer Science,* **8(4)** 433–442, 1997, World Scientific.
4. S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, Proof Verification and the intractability of approximation problems. In *Proceedings 33rd Symposium on the Foundations of Computer Science,* 14–23, IEEE Computer Society Press, 1992.
5. M. Blum and S. Kannan, Designing programs that check their work, *Journal of the ACM,* **42**: 269–291, 1995.
6. M. Blum, M. M. Luby, and R. Rubinfeld, Self-testing/correcting with applications to numerical problems, *J. Comput. Syst. Sciences,* **47(3)**: 549–595, 1993.
7. S. Goldwasser, S. Micali and C. Rackoff, The knowledge complexity of interactive proof systems. *SIAM Journal of Computing,* **18(1)**:186-208, 1989.
8. J. Hastad, Computational limitations for small depth circuits. M.I.T. press, Cambridge, MA, 1986.
9. R. Motwani and P. Raghavan, *Randomized Algorithms,* Cambridge University Press, 1995.
10. R. Rubinfeld, Designing checkers for programs that run in parallel. *Algorithmica,* **15(4)**:287-301, 1996.
11. U. Schöning, Robust algorithms: a different approach to oracles, *Theoretical Computer Science,* **40**: 57–66, 1985.

# Tree-Like Resolution Is Superpolynomially Slower Than DAG-Like Resolution for the Pigeonhole Principle

Kazuo Iwama* and Shuichi Miyazaki

School of Informatics, Kyoto University, Kyoto 606-8501, Japan
{iwama, shuichi}@kuis.kyoto-u.ac.jp

**Abstract.** Our main result shows that a shortest proof size of tree-like resolution for the pigeonhole principle is superpolynomially larger than that of DAG-like resolution. In the proof of a lower bound, we exploit a relationship between tree-like resolution and backtracking, which has long been recognized in this field but not been used before to give explicit results.

## 1 Introduction

A *proof system* is a nondeterministic procedure to prove the unsatisfiability of CNF formulas, which proceeds by applying (usually simple) rules each of which can be computed in polynomial time. Therefore, if there is a proof system which runs in a polynomial number of steps for every formula, then NP=coNP [5]. Since this is not likely, it has long been an attractive research topic to find exponential lower bounds for existing proof systems. There are still a number of well-known proof systems for which no exponential lower bounds have been found, such as Frege systems [3].

Resolution is one of the most popular and simplest proof systems. Even so, it took more than two decades before Haken [6] finally obtained an exponential lower bound for the pigeonhole principle. This settlement of the major open question, however, has stimulated continued research on the topic [1,4,8]. The reason is that Haken's lower bound is quite far from being tight and his proof, although based on an excellent idea later called *bottleneck counting*, is not so easy to read.

Tree-like resolution is a restricted resolution whose proof must be given as not directed acyclic graph (DAG) but a tree. It is a common perception that tree-like proof systems are exponentially weaker than their DAG counterparts. Again, however, proving this for resolution was not easy: In [2], Bonet et al. showed that there exists a formula whose tree-like resolution requires $2^{\Omega(n^\epsilon)}$ steps for some $\epsilon$, while $n^{O(1)}$ steps suffice for DAG-like resolution.

In this paper, we give such a separation between tree-like and DAG-like resolutions using the pigeonhole principle that is apparently the most famous

* Supported in part by Scientific Research Grant, Ministry of Japan, 10558044, 09480055 and 10205215.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 133–142, 1999.
© Springer-Verlag Berlin Heidelberg 1999

and well-studied formula. Our new lower bound for tree-like resolution is $(\frac{n}{4})^{\frac{n}{4}}$ steps for the $n+1$ by $n$ pigeonhole formula. The best previous lower bound is $2^n$ [4] which is not enough for such a (superpolynomial) separation since the best known upper bound of DAG-like resolution is $O(n^3 2^n)$ [4]. Our new lower bound shows that tree-like resolution is superpolynomially slower than DAG-like resolution for the pigeonhole principle.

Another contribution of this paper is that the new bound is obtained by fully exploiting the relationship between resolution and backtracking. This relationship has long been recognized in the community, but it was informal and no explicit research results have been reported. This paper is the first to formally claim a benefit of using this relationship. Our lower bound proof is completely based on backtracking, whose top-down structure makes the argument surprisingly simple and easy to follow.

In Sec. 2, we give basic definitions and notations of resolution, backtracking and the pigeonhole principle. We also show the relationship between resolution and backtracking. In Sec. 3, we prove an $(\frac{n}{4})^{\frac{n}{4}}$ lower bound of tree-like resolution for the pigeonhole principle. In Sec. 4, we give an upper bound $O(n^2 2^n)$ of the DAG-like resolution which is slightly better than $O(n^3 2^n)$ proved in [4]. It should be noted that our argument in this paper holds also for a generalized pigeonhole principle, called the *weak pigeonhole principle*, which is an $m$ by $n$ $(m > n)$ version of the pigeonhole principle. Finally, in Sec. 5, we mention future research topics related to this paper.

## 2 Preliminaries

A *variable* is a logic variable which takes the value *true* (1) or *false* (0). A *literal* is a variable $x$ or its negation $\overline{x}$. A *clause* is a sum of literals and a *CNF formula* is a product of clauses. A *truth assignment* for a CNF formula $f$ is a mapping from the set of variables in $f$ into $\{0,1\}$. If there is no truth assignment that satisfies $f$, we say that $f$ is *unsatisfiable*.

The pigeonhole principle is a tautology which states that there is no bijection from a set of $n+1$ elements into a set of $n$ elements. $PHP_n^{n+1}$ is a CNF formula that expresses a negation of the pigeonhole principle; hence $PHP_n^{n+1}$ is unsatisfiable. $PHP_n^{n+1}$ consists of $n(n+1)$ variables $x_{i,j}$ $(1 \le i \le n+1, 1 \le j \le n)$, and $x_{i,j} = 1$ means that $i$ is mapped to $j$. There are two sets of clauses. The first part consists of clauses $(x_{i,1} + x_{i,2} + \cdots + x_{i,n})$ for $1 \le i \le n+1$. The second part consists of clauses $(\overline{x_{i,k}} + \overline{x_{j,k}})$, where $1 \le k \le n$ and $1 \le i < j \le n+1$. Thus there are $(n+1) + \frac{1}{2}(n^2(n+1))$ clauses in total.

*Resolution* is a proof system for unsatisfiable CNF formulas. It consists of only one rule called an *inference rule*, which infers a clause $(A+B)$ from two clauses $(A+x)$ and $(B+\overline{x})$, where each of $A$ and $B$ denotes a sum of literals such that there is no variable $y$ that appears positively (negatively, resp.) in $A$ and negatively (positively, resp.) in $B$. We say that the variable $x$ is *deleted* by this inference. A *resolution refutation* for $f$ is a sequence of clauses $C_1, C_2, \cdots, C_t$, where each $C_i$ is a clause in $f$ or a clause inferred from clauses $C_j$ and $C_k$ $(j,k < i)$, and the last clause $C_t$ is the empty clause $(\emptyset)$. The size of a resolution refutation is the number of clauses in the sequence.

A resolution refutation can be represented by a directed acyclic graph. (In this sense, we sometimes use the term *DAG-like resolution* instead of "resolution.") If the graph is restricted to a tree, namely, if we can use each clause only once to infer other clauses, then the refutation is called *tree-like resolution refutation* and the tree is called a *resolution refutation tree* (*rrt*). More formally, an rrt for a formula $f$ is a binary rooted tree. Each vertex $v$ of an rrt corresponds to a clause, which we denote by $Cl(v)$. $Cl(v)$ must satisfy the following conditions: For each leaf $v$, $Cl(v)$ is a clause in $f$, and for each vertex $v$ other than leaves, $Cl(v)$ is a clause inferred from $Cl(v_1)$ and $Cl(v_2)$, where $v_1$ and $v_2$ are $v$'s children. For the root $v$, $Cl(v)$ is the empty clause ($\emptyset$). The size of an rrt is the number of vertices in the rrt.

*Backtracking* (e.g., [7]) determines whether a given CNF formula is satisfiable or not in the following way: For a formula $f$, variable $x$ and $a \in \{0, 1\}$, let $f_{x=a}$ be the formula obtained from $f$ by substituting value $a$ for $x$. In each step, we choose a variable $x$ and calculate $f_{x=0}$ and $f_{x=1}$ recursively. If $f$ is simplified to the constant true function at any point, then $f$ is satisfiable, and otherwise, $f$ is unsatisfiable.

Backtracking search is also represented by a tree. A *backtracking tree* (*btt*) for an unsatisfiable formula $f$ is a binary rooted tree satisfying the following three conditions: (i) Two edges $e_1$ and $e_2$ from a vertex $v$ are labeled as $(x = 0)$ and $(x = 1)$ for a variable $x$. (ii) For each leaf $v$ and each variable $x$, $x$ appears at most once (in the form of $(x = 0)$ or $(x = 1)$) in the path from the root to $v$. (iii) For each vertex $v$, let $As(v)$ be the (partial) truth assignment obtained by collecting labels of edges in the path from the root to $v$. Then, for each $v$, $v$ is a leaf iff $f$ becomes false by $As(v)$. (Recall that we consider only unsatisfiable formulas.) The size of a btt is the number of vertices in the btt.

**Proposition 1.** *Let $f$ be an unsatisfiable CNF formula. If there exists an rrt for $f$ whose size is $k$, then there exists a btt for $f$ whose size is at most $k$.*

*Proof.* Let $R$ be an rrt for $f$. It is known that a shortest tree-like resolution refutation is *regular*, i.e., for each path from the root to a leaf, each variable is deleted at most once [9]. Thus we can assume, without loss of generality, that $R$ is regular.

From $R$, we construct a btt $B$ which is isomorphic to $R$. What we actually do is to give a label to each edge in the following way: Let $v_i$ be a vertex of $R$ and let $v_{i_1}$ and $v_{i_2}$ be its children. Suppose $Cl(v_i) = (A + B)$, $Cl(v_{i_1}) = (A + x)$ and $Cl(v_{i_2}) = (B + \overline{x})$. Then the labels $(x = 0)$ and $(x = 1)$ are assigned to edges $(u_i, u_{i_1})$ and $(u_i, u_{i_2})$, respectively, where $u_i$ is a vertex of $B$ corresponding to $v_i$ of $R$. We shall show that this $B$ is a btt for $f$.

It is not hard to see that the conditions (i) and (ii) for btt are satisfied. In the following, we show that for any leaf $u$ of $B$, $f$ becomes false by $As(u)$. This is enough for the condition (iii) because if $f$ becomes false in some non-leaf node, then we can simply cut the tree at that point and can get a smaller one. To this end, we prove the following statement by induction: For each $i$, the clause $Cl(v_i)$ of $R$ becomes false by the partial assignment $As(u_i)$ of $B$. For the induction basis, it is not hard to see that the statement is true for the root. For the induction hypothesis, suppose that the statement is true for a vertex

$v_i$, i.e., $Cl(v_i)$ becomes false by $As(u_i)$. Now we show that the statement is also true for $v_i$'s children. Let $v_{i_1}$ and $v_{i_2}$ be $v_i$'s children, and let $Cl(v_i) = (A + B)$, $Cl(v_{i_1}) = (A + x)$ and $Cl(v_{i_2}) = (B + \overline{x})$. Then the label of the edge $(v_i, v_{i_1})$ is $(x = 0)$, and hence, $As(u_{i_1}) = As(u_i) \cup \{(x = 0)\}$. Since $As(u_i)$ makes $(A + B)$ false, $As(u_{i_1})$ makes $(A+x)$ false. The same argument shows that $As(u_{i_2})$ makes $Cl(v_{i_2})$ false. Now the above statement is proved, which immediately implies that $As(u_i)$ makes $f$ false for every leaf $u_i$ of $B$. □

Thus to show a lower bound of tree-like resolution, it suffices to show a lower bound on the size of btts.

## 3 A Lower Bound

In this section, we prove a lower bound on the size of tree-like resolution for $PHP_n^{n+1}$.

**Theorem 1.** *Any tree-like resolution refutation for $PHP_n^{n+1}$ requires $(\frac{n}{4})^{\frac{n}{4}}$ steps.*

*Proof.* By Proposition 1, it is enough to show that any btt for $PHP_n^{n+1}$ requires $(\frac{n}{4})^{\frac{n}{4}}$ vertices. For simplicity, we consider the case that $n$ is a multiple of 4. Let $B$ be an arbitrary btt for $PHP_n^{n+1}$. As we have seen before, each vertex $v$ of $B$ corresponds to a partial truth assignment $As(v)$. For a better exposition, we use an $n+1$ by $n$ array representation to express a partial assignment for $PHP_n^{n+1}$. Fig. 1 shows an example of $PHP_4^5$. A cell in column $i$ and row $j$ corresponds to the variable $x_{i,j}$. We consider that the value 1 (resp. 0) is assigned to the variable $x_{i,j}$ if the $(i, j)$ entry of the array is 1 (resp. 0). For example, Fig. 1 (b) expresses a partial assignment such that $x_{1,4} = x_{2,2} = x_{4,4} = x_{5,3} = 0$, $x_{3,3} = x_{4,1} = 1$. It should be noted that $PHP_n^{n+1}$ becomes false at a vertex $v$ iff (i) $As(v)$ contains a column filled with 0s or (ii) $As(v)$ contains a row in which two 1s exist.

Here are some notations. For a partial assignment $A$, a 0 in the $(i, j)$ entry of $A$ is called a *bad* 0 if neither the column $i$ nor the row $j$ contains a 1, and is called a *good* 0 otherwise. (The reason why we use terms "bad" and "good" will be seen later. Bad 0s make it difficult to count the number of vertices in $B$ in our analysis.) $\#BZ(A)$ denotes the number of bad 0s in $A$. A variable $x_{i,j}$ is called an *active variable* for $A$ if $x_{i,j}$ is not yet assigned (that is, $(i, j)$ entry of $A$ is blank) and neither the column $i$ nor the row $j$ contains a 1. For example, let $A_0$ be the assignment in Fig. 1 (b). Then $\#BZ(A_0) = 2$ (0s assigned to $x_{1,4}$ and $x_{2,2}$). For example, $x_{2,4}$ is an active variable for $A_0$. Let $v$ be a vertex of $B$ and $v_0$ and $v_1$ be its children. Suppose that labels of edges $(v, v_0)$ and $(v, v_1)$ are $(x = 0)$ and $(x = 1)$, respectively. Then we write $Var(v) = x$, namely, $Var(v)$ denotes the variable selected for substitution at the vertex $v$. We call $v_0$ a *false-child* of $v$ and write $F(v) = v_0$. Similarly we call $v_1$ a *true-child* of $v$ and write $T(v) = v_1$.

We want to show a lower bound on the number of vertices in $B$. To this end, we construct a tree $S$ from $B$. Before showing how to construct $S$, we show some properties of $S$: The set of vertices of $S$ is a subset of the set of vertices of $B$, so the number of vertices of $S$ gives a lower bound on the number of vertices of $B$. Each internal vertex of $S$ have either a single child or exactly $\frac{n}{4}$ children. The

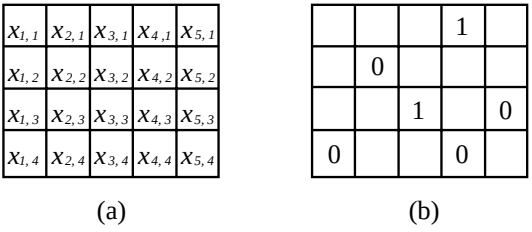



**Fig. 1.** An array representation of a partial assignment for $PHP_4^5$

height of $S$ is $\frac{n}{2}$; more precisely, the length of any path from the root to a leaf is exactly $\frac{n}{2}$.

Now we show how to construct $S$. The root of $S$ is the root of $B$. For each vertex $v$ of $S$, we select the set $CH(v)$ of $v$'s children in the following way: $CH(v)$ is initially empty and vertices are added to $CH(v)$ one by one while tracing the tree $B$ down from the vertex $v$. We look at vertices $T(v), T(F(v)), T(F(F(v)))(= T(F^2(v))), \cdots, T(F^l(v)), \cdots$ in this order. We add $T(F^l(v))$ $(l \geq 0)$ to $CH(v)$ if $Var(F^l(v))$ is an active variable for $As(F^l(v))$. Fig. 2 illustrates how to trace the tree $B$ when we construct the tree $S$. In this example, $T(F^2(v))$ is "skipped" because $Var(F^2(v))$ is not an active variable. We stop adding vertices if $|CH(v)|$ becomes $\frac{n}{4}$. In this case, $v$ has exactly $\frac{n}{4}$ children.

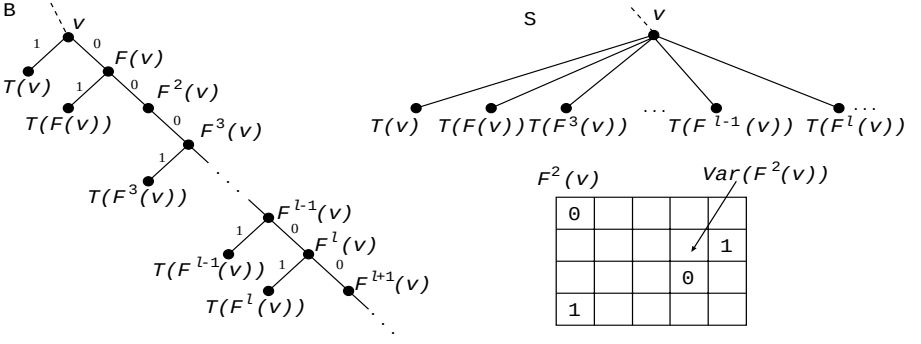



**Fig. 2.** A part of $S$ constructed from $B$

However, there is one exceptional condition to stop adding vertices to $CH(v)$ even if $|CH(v)|$ is less than $\frac{n}{4}$; when the number of bad 0s in some column reaches $\frac{n}{2}$, we stop adding vertices to $CH(v)$. More formally, let us consider a vertex $F^l(v)$. Suppose that the number of bad 0s in each column of $As(F^l(v))$ is at most $\frac{n}{2} - 1$. Also, suppose that $Var(F^l(v))$ is $x_{i,j}$ where the column $i$ of $As(F^l(v))$ contains exactly $\frac{n}{2} - 1$ bad 0s and there is no 1 in the row $j$ of $As(F^l(v))$. (See Fig. 3 for an example of the case that $n = 12$. There are eight 0s in the column $i$. Among them, five 0s are bad 0s.) Then $As(F^{l+1}(v))$

contains $\frac{n}{2}$ bad 0s in the column $i$, and hence we do not look for vertices any more, namely, $T(F^l(v))$ is the last vertex added to $CH(v)$. (Note that $T(F^l(v))$ is always selected since $x_{i,j}$ is an active variable for $As(F^l(v))$.) In this case, $|CH(v)|$ may be less than $\frac{n}{4}$. If so, we adopt only the last vertex as a child of $v$, i.e., only $T(F^l(v))$ is a child of $v$ in $S$. Thus, in this case, $v$ has only one child. It should be noted that, in the tree $S$, every assignment corresponding to a vertex of depth $i$ contains exactly $i$ 1s. We continue this procedure until the length of every path from the root to a leaf becomes $\frac{n}{2}$.

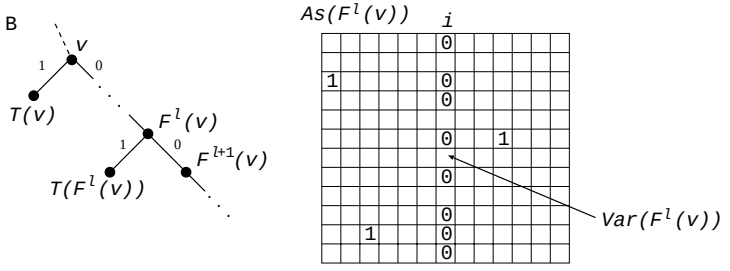



**Fig. 3.** A condition to stop tracing $B$

We then show that it is possible to construct such $S$ from any $B$. To see this, it suffices to show that we never reach a leaf of $B$ while tracing $B$ to construct $S$. Recall the definition of a backtracking tree: For any leaf $u$ of btt, $As(u)$ makes the CNF formula false. For $As(u)$ to make $PHP_n^{n+1}$ false, $As(u)$ must have a row containing two 1s or a column full of 0s. The former case does not happen in $S$ because we have skipped such vertices in constructing $S$. The latter case does not happen for the following reason: Recall that once the number of bad 0s in some column reaches $\frac{n}{2}$, we stop tracing the tree $B$. So, as long as we trace $B$ in constructing $S$, we never visit an assignment such that the number of bad 0s in a column exceeds $\frac{n}{2} - 1$. Also, recall that the number of 1s in $As(v)$ is at most $\frac{n}{2}$ since $v$'s depth in $S$ is at most $\frac{n}{2}$. So the number of good 0s in a column is at most $\frac{n}{2}$. Hence the number of 0s in each column is at most $n - 1$, and so, no column ever becomes filled with 0s. Now let us consider the following observation which helps to prove later lemmas.

**Observation 1.** Consider a vertex $v$ in $B$ and let $v' = F^l(v)$ for some $l$ (see Fig. 4). Suppose first that $T(v')$ is added to $CH(v)$ in constructing $S$. Then $Var(v')$ must be an active variable for $v'$, and hence $\#BZ(As(F(v'))) = \#BZ(As(v')) + 1$. On the other hand, suppose that $T(v')$ is not added to $CH(v)$ in constructing $S$. Then $Var(v')$ is not an active variable for $v'$. Therefore, $\#BZ(As(F(v'))) = \#BZ(As(v'))$.

Now we have a tree $S$ having the following properties: The length of any path from the root to a leaf is $\frac{n}{2}$. Every vertex in $S$ except for leaves has exactly $\frac{n}{4}$ children or one child. When a vertex $v$ has one child, we call the edge between $v$ and its child a *singleton*.

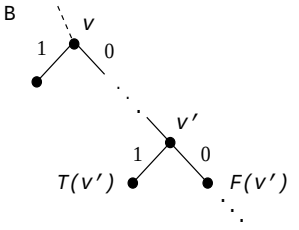



**Fig. 4.** An example for Observation 1

**Lemma 1.** *Consider an arbitrary path* $P = u_0u_1u_2\cdots u_{\frac{n}{2}}$ *in* $S$*, where* $u_0$ *is the root and* $u_{\frac{n}{2}}$ *is a leaf. For each* $k$*, if* $(u_{k-1}, u_k)$ *is not a singleton, then* $\#BZ(As(u_k)) \leq \#BZ(As(u_{k-1})) + \frac{n}{4} - 1$.

*Proof.* Consider $u_{k-1}$ and $u_k$ in the path $P$. Let $v$ be the vertex in the btt $B$ corresponding to $u_{k-1}$. Then there is some $l$ such that $T(F^l(v))$ corresponds to $u_k$. By Observation 1, $\#BZ(As(F^l(v))) - \#BZ(As(v))$ is equal to the number of vertices added to $CH(v)$ among $T(v), T(F(v)), T(F^2(v)), \cdots, T(F^{l-1}(v))$. So $\#BZ(As(F^l(v))) - \#BZ(As(v)) \leq \frac{n}{4} - 1$. Note that $As(T(F^l(v)))$ is the result of adding one 1 to some active variable of $As(F^l(v))$, so $\#BZ(As(T(F^l(v)))) \leq \#BZ(As(F^l(v)))$. Hence $\#BZ(As(T(F^l(v)))) \leq \#BZ(As(v)) + \frac{n}{4} - 1$, namely, $\#BZ(As(u_k)) \leq \#BZ(As(u_{k-1})) + \frac{n}{4} - 1$. □

**Lemma 2.** *Consider an arbitrary path* $P = u_0u_1u_2\cdots u_{\frac{n}{2}}$ *in* $S$*, where* $u_0$ *is the root and* $u_{\frac{n}{2}}$ *is a leaf. For each* $k$*, if* $(u_{k-1}, u_k)$ *is a singleton, then* $\#BZ(As(u_k)) \leq \#BZ(As(u_{k-1})) - \frac{n}{4}$.

*Proof.* Suppose that the edge $(u_{k-1}, u_k)$ is a singleton. Let $v$ and $T(F^l(v))$ be vertices in $B$ corresponding to $u_{k-1}$ and $u_k$, respectively. The same argument as in Lemma 1 shows that $\#BZ(As(F^l(v))) - \#BZ(As(v)) \leq \frac{n}{4} - 1$. Let $Var(F^l(v)) = x_{i,j}$. Since $(u_{k-1}, u_k)$ is a singleton, there are $\frac{n}{2} - 1$ bad 0s in the column $i$ of $As(F^l(v))$. Thus substituting the value 1 for the variable $x_{i,j}$ makes at least $\frac{n}{2} - 1$ bad 0s good, namely, $\#BZ(As(T(F^l))) \leq \#BZ(As(F^l(v))) - (\frac{n}{2} - 1)$. Hence $\#BZ(As(T(F^l(v)))) \leq \#BZ(As(v)) - \frac{n}{4}$, namely, $\#BZ(As(u_k)) \leq \#BZ(As(u_{k-1})) - \frac{n}{4}$. □

**Lemma 3.** *Consider an arbitrary path* $P = u_0u_1u_2\cdots u_{\frac{n}{2}}$ *in* $S$*, where* $u_0$ *is the root and* $u_{\frac{n}{2}}$ *is a leaf. The number of singletons in* $P$ *is at most* $\frac{n}{4}$.

*Proof.* Suppose that there exist more than $\frac{n}{4}$ singletons in the path $P$. We count the number of bad 0s of assignments along $P$. At the root, $\#BZ(As(u_0)) = 0$. Going down the path from the root $u_0$ to the leaf $u_{\frac{n}{2}}$, the number of bad 0s is increased at most $(\frac{n}{4} - 1)(\frac{n}{4} - 1) < \frac{n^2}{16}$ by Lemma 1, and is decreased at least $\frac{n}{4}(\frac{n}{4} + 1) > \frac{n^2}{16}$ by Lemma 2. This is a contradiction because the number of bad 0s becomes negative at the leaf. This completes the proof. □

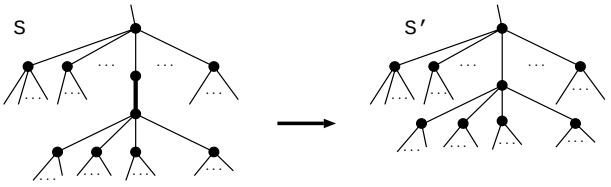



**Fig. 5.** Shrinking $S$ to obtain $S'$

Finally we "shrink" the tree $S$ by deleting all singletons from $S$. Let $S'$ be the resulting tree (see Fig. 5). Each vertex of $S'$ has exactly $\frac{n}{4}$ children and the length of every path from the root to a leaf is at least $\frac{n}{4}$ by Lemma 3. So there are at least $(\frac{n}{4})^{\frac{n}{4}}$ vertices in $S'$, and hence, the theorem follows. □

**Remark.** By slightly modifying the above proof, we can get a better lower bound of $(\frac{n}{4\log^2 n})^n$. For the tree $S$ in the above proof, we restrict the number of children of each node, the maximum number of bad 0s in each column, and the height of the tree with $\frac{n}{4}$, $\frac{n}{2}$, and $\frac{n}{2}$, respectively. To get a better lower bound, we let them be $\delta^2 n$, $\delta n$, and $(1-\delta)n$, respectively, with $\delta = \frac{1}{\log n}$. Then, the number of singletons in each path is at most $\delta(1-\delta)n$ and hence we can obtain a lower bound $(\delta^2 n)^{(1-\delta)^2 n} \geq (\frac{n}{4\log^2 n})^n$.

## 4 An Upper Bound

It is known that the size of a DAG-like resolution refutation for $PHP_n^{n+1}$ is $O(n^3 2^n)$[4]. Here we show a slightly better upper bound which is obtained by the similar argument as [4]. We can also obtain an upper bound of tree-like resolution refutation for $PHP_n^{n+1}$ as a corollary.

**Theorem 2.** *There is a DAG-like resolution refutation for $PHP_n^{n+1}$ whose size is $O(n^2 2^n)$.*

*Proof.* Let $Q$ and $R$ be subsets of $\{1, 2, \cdots, n+1\}$ and $\{1, 2, \cdots, n\}$, respectively. Then we denote by $P_{Q,R}$ the sum of positive literals $x_{i,j}$, where $i \in Q$ and $j \in R$. Let $[i, j]$ denote the set $\{i, i+1, \cdots, j-1, j\}$.

We first give a rough sketch of the refutation and then describe it in detail. The 0th level of the refutation has the single clause $P_{\{1\},[1,n]}$. The first level consists of $n$ clauses $P_{[1,2],R^{(n-1)}}$ for all sets $R^{(n-1)} \subset [1, n]$ of size $n-1$. The second level consists of ${}_nC_{n-2}$ clauses $P_{[1,3],R^{(n-2)}}$ for all sets $R^{(n-2)} \subset [1, n]$ of size $n-2$. Generally speaking, the $i$th level consists of ${}_nC_{n-i}$ clauses $P_{[1,i+1],R^{(n-i)}}$ for all $R^{(n-i)} \subset [1, n]$ of size $n-i$. At the $(n-1)$th level, we have ${}_nC_1 = n$ clauses $P_{[1,n],\{1\}}, P_{[1,n],\{2\}}, \cdots, P_{[1,n],\{n\}}$. Finally, at the $n$th level, we have the empty clause. We call the clauses described here *main clauses*. Note that there are $\Sigma_{i=0}^{i=n}({}_nC_i) = 2^n$ main clauses. Fig. 6 shows an example of the case when $n = 4$. A "+" sign in the $(i, j)$ entry means the existence of the literal $x_{i,j}$ in that clause.

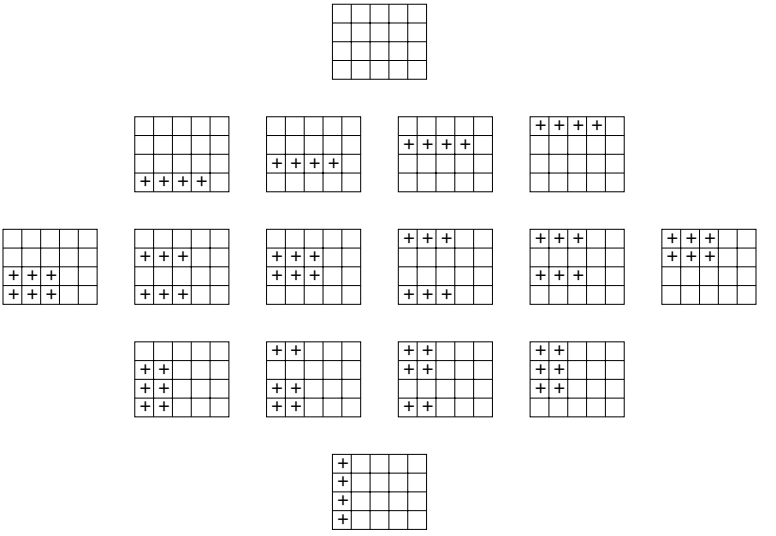


**Fig. 6.** Main clauses of the refutation

Then we describe the detail of the refutation. Each clause at the $i$th level is obtained by using $i$ clauses of the $(i-1)$th level and some initial clauses. To construct a clause $P_{[1,i+1],\{j_1,j_2,\cdots,j_{n-i}\}}$ in the $i$th level, we use $i$ clauses $P_{[1,i],\{j_1,j_2,\cdots,j_{n-i},k\}}$ for all $k \notin \{j_1, j_2, \cdots, j_{n-i}\}$. First, for each $k$, we construct a clause $P_{[1,i],\{j_1,j_2,\cdots,j_{n-i}\}} \cup \overline{x_{i+1,k}}$ using the clause $P_{[1,i],\{j_1,j_2,\cdots,j_{n-i},k\}}$ of the $(i-1)$th level and $i$ clauses $(\overline{x_{1,k}} + \overline{x_{i+1,k}})(\overline{x_{2,k}} + \overline{x_{i+1,k}}) \cdots (\overline{x_{i,k}} + \overline{x_{i+1,k}})$. Then we construct a target clause $P_{[1,i+1],\{j_1,j_2,\cdots,j_{n-i}\}}$ by using those $i$ clauses $P_{[1,i],\{j_1,j_2,\cdots,j_{n-i}\}} \cup \overline{x_{i+1,k}}$ and the initial clause $P_{\{i+1\},[1,n]}$. Fig. 7 illustrates an example of deriving $P_{[1,3],\{1,4\}}$ in the second level from clauses $P_{[1,2],\{1,2,4\}}$ and $P_{[1,2],\{1,3,4\}}$ in the first level. Similarly as the "+" sign, a "−" sign in the $(i,j)$ entry means the existence of the literal $\overline{x_{i,j}}$ in that clause.

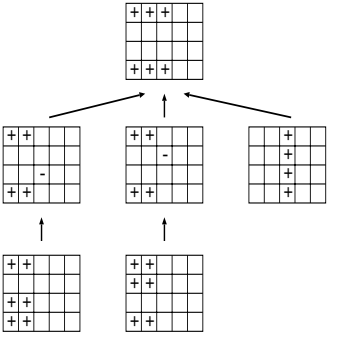


**Fig. 7.** Constructing a main clause

Thus to construct each main clause, we need $O(n^2)$ steps. Since there are $2^n$ main clauses, the size of the above refutation is bounded by $O(n^2 2^n)$. □

**Corollary 1.** *There is a tree-like resolution refutation for $PHP_n^{n+1}$ whose size is $O(n^2 n!)$.*

*Proof.* This is obtained by reforming the directed acyclic graph of the refutation obtained in Theorem 2 into a tree in a trivial manner. For main clauses, we have one level-$n$ clause, $n$ level-$(n-1)$ clauses, $n(n-1)$ level-$(n-2)$ clauses and so on. Generally speaking, we have $n(n-1)\cdots(i+1)$ level-$i$ clauses. Thus we have $1 + \Sigma_{i=0}^{i=n-1} n(n-1)\cdots(i+1) \leq 2n!$ main clauses. Each main clause is constructed in $O(n^2)$ steps and hence the size of the refutation is $O(n^2 n!)$. □

## 5 Concluding Remarks

By Theorems 1 and 2, we can see that the size of any tree-like resolution refutation is superpolynomially larger than the size of a shortest DAG-like resolution refutation. An interesting future research is to find a set of formulas that separates tree-like and DAG-like resolutions in the rate of $2^{cn}$ for some constant $c$ improving [2]. Another research topic is to find a tighter bound of the tree-like resolution for the pigeonhole principle. Note that an upper bound $O(n^2 n!)$ and a lower bound $(\frac{n}{4\log^2 n})^n$ obtained in this paper are tight in the sense that they both grow at the same rate of $n^{(1-o(1))n}$. An open question is whether we can get a tighter lower bound, e.g., $\Omega(n!)$.

**Acknowledgments.** The authors would like to thank Magnus M. Halldórsson for his valuable comments.

## References

1. P. Beame and T. Pitassi, "Simplified and improved resolution lower bounds," *Proc. FOCS'96*, pp. 274–282, 1996.
2. M. L. Bonet, J. L. Esteban, N. Galesi and J. Johannsen, "Exponential separations between restricted resolution and cutting planes proof systems," *Proc. FOCS'98*, pp. 638–647, 1998.
3. S. Buss, "Polynomial size proofs of the propositional pigeonhole principle," *Journal of Symbolic Logic*, 52, pp. 916–927, 1987.
4. S. Buss and T. Pitassi, "Resolution and the weak pigeonhole principle," *Proc. CSL'97*, LNCS 1414, pp.149–156, 1997.
5. S. A. Cook and R. A. Reckhow, "The relative efficiency of propositional proof systems," *J. Symbolic Logic*, 44(1), pp. 36–50, 1979.
6. A. Haken, "The intractability of resolution," *Theoretical Computer Science*, 39, pp. 297–308, 1985.
7. P. Purdom, "A survey of average time analysis of satisfiability algorithms," *Journal of Information Processing*, 13(4), pp.449–455, 1990.
8. A. A. Razborov, A. Wigderson and A. Yao, "Read-Once Branching programs, rectangular proofs of the pigeonhole principle and the transversal calculus," *Proc. STOC'97*, pp. 739–748, 1997.
9. A. Urquhart, "The complexity of propositional proofs," *The Bulletin of Symbolic Logic*, Vol. 1, No. 4, pp. 425-467, 1995.

# Efficient Approximation Algorithms for Multi-label Map Labeling

Binhai Zhu* and C.K. Poon**

Dept. of Computer Science, City University of Hong Kong, Kowloon, Hong Kong SAR, China.
{bhz,ckpoon}@cs.cityu.edu.hk.

**Abstract.** In this paper we study two practical variations of the map labeling problem: Given a set $S$ of $n$ distinct sites in the plane, one needs to place at each site: (1) a pair of uniform and non-intersecting squares of maximum possible size, (2) a pair of uniform and non-intersecting circles of maximum possible size. Almost nothing has been done before in this aspect, i.e., multi-label map labeling. We obtain constant-factor approximation algorithms for these problems. We also study bicriteria approximation schemes based on polynomial time approximation schemes (**PTAS**) for these problems.

## 1 Introduction

Map labeling is an old art in cartography which finds new applications in recent years in GIS, graphics and graph drawing. Extensive research have been conducted in this area [BC94,CMS93,CMS95,DF92,FW91,Imh75,Jon89] [KR92,PZC98,Wag94]. Recently much of the research are on generalizing the problems (models). One direction is to allow each site to have many, sometimes an infinite number of, possible labels (see [DMMMZ97,IL97,KSW98]). Another direction is to study the corresponding problem in a related area such as graph drawing [KT97,KT98a]. On the other hand, many of the realistic problems, like map labeling with multiple-labels and different label shapes, have not received much attention.

The multiple-label map labeling problem comes from our weather forecasting programs on TV where each city has two labels: its name and temperature. This naturally gives rise to two optimization problems: namely, that of maximizing label sizes and maximizing the number of sites labelled. This realistic example of map labeling, which everybody encounters almost daily nowadays, is largely ignored in the research of map labeling. The only result in this respect we know of is a very recent short note by Kakoulis and Tollis [KT98] which presents

* Binhai Zhu's research is supported by City University of Hong Kong, Laurentian University, NSERC of Canada and Hong Kong RGC CERG grant CityU1103/99E.
** C.K. Poon's research is supported by Hong Kong RGC Competitive Earmarked Research Grant 9040314.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 143–152, 1999.
© Springer-Verlag Berlin Heidelberg 1999

practical heuristic algorithm for solving this problem. However, no theoretical result is known.

Although the computational complexity of these problems are not yet known, it is very likely that they are all NP-hard. Thus, finding efficient approximate solutions is meaningful. We will mainly follow [DMMMZ97] to obtain efficient approximation solutions and approximation schemes for these problems. However, these extensions are non-trivial. In fact, in many ways we either simplify the methods in [DMMMZ97] or further generalize the results there.

This paper is organized as follows. Section 2 contains a brief summary of results. Section 3 formally defines the problems. Section 4 and Section 5 discusses the constant factor approximation algorithms for labeling a set of sites with uniform (axis-parallel) squares and uniform circle pairs. Section 6 contains a bicriteria approximation scheme for these problems. Section 7 concludes the paper.

## 2 A Summary of Results

In this paper, for the first time in literature, we study approximation algorithms for the general multi-label map labeling problem with uniform (axis-parallel) square pairs and uniform circle pairs. We obtain constant factor approximation algorithms for these problems. Recall that an approximation algorithm for a (maximization) optimization problem $\Pi$ provides a **performance guarantee** of $\rho$ if for every instance $I$ of $\Pi$, the solution value returned by the approximation algorithm is at least $1/\rho$ of the optimal value for $I$. (For the simplicity of description, we simply say that this is a factor $\rho$ approximation algorithm for $\Pi$.) Our main results are summarized as follows:

1. For map labeling with uniform square pairs, we design a polynomial time approximation algorithm with a performance guarantee of 4.
2. For map labeling with uniform circle pairs, we design a time-optimal approximation algorithm with a performance guarantee of 2.
3. We study bicriteria approximation schemes for the above two problems. Bicriteria approximation scheme for the map labeling problem is defined as that for any given $\epsilon$, $(1 - \epsilon)$ fraction of sites may be chosen in order to get a solution which is at least $(1 - c' \cdot \epsilon)$, where $c'$ is some constant, times the size of the optimal solution.

## 3 Preliminaries

In this section we formally define the problems to be studied. We also make some extra definitions related to our algorithms. To make our descriptions easily accessible, we restrict any square used in this paper to be axis-parallel — although by sacrificing the performance of the algorithms we can generalize the result to arbitrary uniform square pairs.

### 3.1 Map Labeling with Uniform Square Pairs (MLUSP)

The *decision version* of the MLUSP problem is defined as follows:

**Instance:** Given a set $S$ of points (sites) $p_1$, $p_2$, ..., $p_n$ in the plane and an integer $l$.
**Problem:** Does there exist a set of $n$ pairs of squares of size (i.e., length of a side) $l$ each of which are placed at each input site $p_i \in S$ such that no two squares intersect and no site is contained in any square.

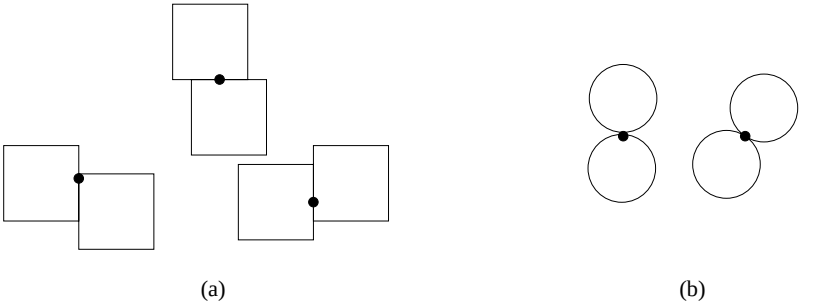



**Fig. 1.** Examples of labeling sites with square and circle pairs.

It should be noted that in this problem the site can be anywhere on the boundary of the two labeling squares (Figure 1 (a)). It is not known whether the MLUSP problem is NP-complete or not. Nevertheless, we will try to approximate the corresponding *maximization problem* in the subsequent sections. From now on, MLUSP will always refer to the maximization problem.

### 3.2 Map labeling with Uniform Circle Pairs(MLUCP)

The problem of MLUCP is defined as follows:

**Instance:** Given a set $S$ points (sites) $p_1$, $p_2$, ..., $p_n$ in the plane and an integer $k$.
**Problem:** Does there exist a set of $n$ uniform circle pairs of radius $l$ each of which are placed at each input site $p_i \in S$ such that no two circles intersect and no site is contained in any circle.

Because of the nature of this problem, the two circles labeling a site $p_i$ must be tangent to each other and $p_i$ is exactly the tangent point (Figure 1 (b)). Again, it is not known whether the MLUCP problem is NP-complete or not.

### 3.3 The Minimum 3-Diameter under $L_\infty$

Let $k \geq 2$ be an integer. Given a set $S$ of $k$ sites (sites) in the plane, the $k$-diameter of $S$ under the $L_\infty$ metric (or $L_\infty$ $k$-diameter in short) is defined as the maximum $L_\infty$ distance between any two points in $S$. Given a set $S$ of at least $k$ sites in the plane, the min-$k$-diameter of $S$ under $L_\infty$, denoted as $D_{k,\infty}(S)$, is the minimum $L_\infty$ $k$-diameter over all possible subsets of $S$ of size $k$. We can also define a similar set of notations on the $L_2$ (i.e., Euclidean) metric. In particular, we denote by $D_k(S)$ the minimum $k$-diameter (under $L_2$) over all possible subsets of $S$ of size $k$. Observe that $D_{k,\infty}(S) \leq D_k(S)$ for any $S$ and $k$.

The following two lemmas show why the 3-diameter, in particular, the 3-diameter under $L_\infty$ is relevant to the labeling of square pairs.

**Lemma 1.** *Given a set of three points with 3-diameter $D_{3,\infty}$ under $L_\infty$, the optimal labeling square pairs have size at most $D_{3,\infty}$.*

*Proof.* Let the three points be $p_1, p_2$ and $p_3$. Without loss of generality, let $D_{3,\infty}$ be determined by $p_1$ and $p_2$. Clearly when we look at the axis-parallel bounding box of the three points, its long edge (which is exactly $D_{3,\infty}$) is the maximum size for labeling a square pair on $p_3$. □

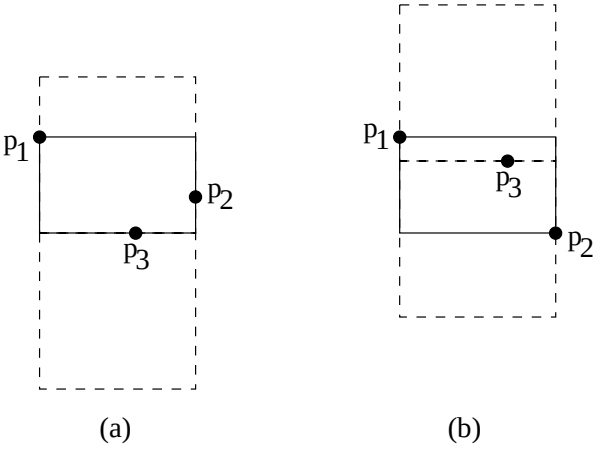



**Fig. 2.** Labeling 3 sites with maximum square pairs.

In Figure 2, we show how to label three points with maximum square pairs (the two dotted squares are for $p_3$ as $p_1, p_2$ can be easily labelled with pairs of squares of the same size). The following lemma extends the previous one to the case when $S$ has more than 3 sites.

**Lemma 2.** *Given a set $S$ of $n$ sites with minimum $L_\infty$ 3-diameter $D_{3,\infty}(S)$, the optimal labeling square pairs have size at most $D_{3,\infty}(S)$.*

In the following section we will present an approximation solution for labeling a set $S$ of $n$ sites using square pairs. For the ease of presentation, we will first use $D_3(S)$, the min-3-diameter of the same set of sites under $L_2$, to approximate $D_{3,\infty}(S)$ — and to obtain an approximation solution whose performance guarantee is slightly worse than 4. Then we show how to generalize the idea to use $D_{3,\infty}(S)$ to obtain a factor 4 approximation algorithm. Given a set of $n$ sites $S$, both $D_{3,\infty}(S)$ and $D_3(S)$ can be computed in $O(n \log n)$ time [EE94].

## 4 Map Labeling With Uniform Square Pairs (MLUSP)

In this section we present the details of an approximation algorithm for the MLUSP problem. Let $L^*$ denote the size of each square in the optimal solution of the problem MLUSP. By Lemma 2 and the fact that $D_{3,\infty}(S) \leq D_3(S)$, we have the following lemma.

**Lemma 3.** *If $|S| \leq 2$, then $L^*$ is unbounded and if $|S| \geq 3$, then $L^* \leq D_3(S)$.*

For any two points $p_i, p_j \in S$, let $d_{ij}$ denote the Euclidean distance between them. Let $C_i$ denote the circle centered at point $p_i \in S$ with radius $\frac{D_3(S)}{\alpha}$, where $\alpha \geq 4$. As shown in [DMMMZ97], the circle $C_i$ contains at most two points from the input set $S$, including its center. (Otherwise, the three points inside $C_i$ will have a diameter smaller than $D_3(S)$.) Let $p_j$, if exists, be the site in the circle $C_i$. Clearly $p_i$ is the only point contained in $C_j$. Since $\alpha \geq 4$, by the triangle inequality, no circle $C_i$ can intersect more than one circle. (Otherwise, if $C_i$ intersects $C_j, C_k$ then $d_{jk} \leq d_{ij} + d_{ik} < D_3(S)$ and the diameter of $p_i, p_j, p_k$ will be smaller than $D_3(S)$.)

Notice that as $C_i$ can intersect at most one circle, if $C_i$ is empty of other sites then trivially we can label a pair of squares with edge length $D_3(S)/\sqrt{2}\alpha$. Therefore, we only consider the (more difficult) case when $C_i$ contains another site $p_j$. By definition, $p_i, p_j \in C_i \cap C_j$. Therefore, we always have a half of $C_i$, bounded away from $p_j$ by a horizontal or a vertical line through $p_i$, being empty. Let $X_i$ be the square pair with edge length $\frac{D_3(S)}{\sqrt{2}\alpha}$ placed at point $p_i$. If $p_i$ is above $p_j$ then we place a square pair $X_i$ above $p_i$ and a square pair $X_j$ below $p_j$. If $p_i$ is below $p_j$ then we place a square pair $X_i$ below $p_i$ and a square pair $X_j$ above $p_j$. If $p_i$ and $p_j$ are on the same horizontal line then we place a square pair $X_i$ to the left of $p_i$ and a square pairs $X_j$ to the right of $p_j$.

Let $L'$ denote the edge length of each square generated by above procedure. We have the following lemma.

**Lemma 4.** *$L' \geq \frac{D_3(S)}{\sqrt{2}\alpha}$. Moreover $L' \geq \frac{L^*}{\sqrt{2}\alpha}$.*

The proof of this lemma is straightforward as $L' \geq \frac{D_3(S)}{\sqrt{2}\alpha} \geq \frac{D_{3,\infty}(S)}{\sqrt{2}\alpha} \geq \frac{L^*}{\sqrt{2}\alpha}$. Summarizing the above results, we have the following theorem.

**Theorem 5.** *For any given set of $n$ points in the plane, the above algorithm, which runs in $O(n \log n)$ time, produces a $4\sqrt{2} = 5.656$ approximation for the MLUSP problem.*

The algorithm involves first computing $D_3(S)$ and then labeling $S$ using the above constructive procedure. We hence omit the details.

Now it is quite obvious to generalize this algorithm by using $D_{3,\infty}(S)$ directly: Just generalize $C_i$ as the circle under the $L_\infty$ metric — which is geometrically a square, centered at point $p_i \in S$ with radius $\frac{D_{3,\infty}(S)}{\alpha}$. Again, since $\alpha \geq 4$, no circle can contain more than two sites and no circle can intersect more than one circle. The remaining steps are the same as shown in the above theorem and we are able to slice off the $\sqrt{2}$ factor. Therefore we have

**Theorem 6.** *For any given set of $n$ points in the plane, there exists a 4 approximation algorithm which runs in $O(n \log n)$ time for the MLUSP problem.*

## 5 Map Labeling With Uniform Circle Pairs (MLUCP)

In this section, we study the **MLUCP** problem. Let $R^*$ denote the radius of each circle in the optimal solution. Recall that $D_2(S)$ is the Euclidean distance between a closest pair of $S$. We have the following lemma.

**Lemma 7.** $R^* \leq D_2(S)/2$.

*Proof.* Consider the two points $p_i$ and $p_j$ which is a closest pair of $S$. Thus they are at distance $d_{ij} = D_2(S)$ apart. To avoid the intersection of the two pairs of circles and maximize the size of these circles, the best way is to place the two pairs of circles such that the lines through the centers of each pair are parallel to each other and perpendicular to $(p_i, p_j)$. In this way the maximum radius of these circles is $d_{ij}/2 = D_2(S)/2$. Clearly, the optimal solution for the problem **MLUCP**, $R^*$ must be bounded by $D_2(S)/2$. □

### 5.1 Algorithm

We are now ready to present an approximation algorithm for the **MLUCP** problem. We need an algorithm called ClosestPair to compute $D_2(S)$.

**Procedure** *PackCirclePair*($S$)
1. $D_2(S) :=$ ClosestPair($S$).
2. For each point $p_i$ in $S$, do the following:
   Place a pair of circles $Y_i$ of radius $D_2(S)/4$ at $p_i$ arbitrarily.

The correctness of the above algorithm rests on Step 2. For every site $p_i$, let $C_i$ be a circle of radius $D_2(S)/2$ centered at $p_i$. Then no two such circles can intersect; otherwise there would be a pair of sites whose Euclidean distance is less than $D_2(S)$, a contradiction. As the radius of each circle in the circle pair $Y_i$ is only $D_2(S)/4$, $Y_i$ is circumscribed by $C_i$ and hence will not overlap with any other circle pairs. Using the standard results of [PS85], we can find $D_2(S)$ in $O(n \log n)$ time. Therefore the whole algorithm takes $O(n \log n)$ time and we have the following theorem.

**Theorem 8.** *The above $O(n \log n)$ time algorithm finds an approximate solution with performance guarantee of 2.*

Clearly the $O(n \log n)$ time bound is optimal: in the degenerate case when all the $y$-coordinates of the points in $S$ are the same, any (approximate) solution for MLUCP returns a non-zero value if and only if all the $x$-coordinates of the points in $S$ are distinct. The latter is the famous *element uniqueness* problem, which has an $\Omega(n \log n)$ lower bound [BO83].

## 6 A Bicriteria Approximation Scheme

In reality, factor-4 or factor-2 approximations to the MLUSP and MLUCP problems could be far from satisfactory. In this section, we consider a bicriteria variant of the basic problem. We show that if we are allowed not to label a small subset of sites we can have an approximate solution which is almost optimal. We only focus on the MLUSP problem as the generalization to MLUCP is straightforward.

First of all, we need a few definitions. Define a polynomial time $(\alpha, \beta)$ approximation algorithm for the MLUSP problem as a polynomial time approximation algorithm that finds a placement of square pairs for at least an $\alpha$ fraction of sites such that the size of each square is at least $\beta$ times the size of a square in an optimal solution that places square pairs at each site.

An undirected graph is a **unit square graph** if and only if its vertices can be put in one-to-one correspondence with equal sized squares in the plane in such a way that two vertices are joined by an edge if and only if the corresponding squares intersect. (When dealing with this graph we assume that tangent squares intersect.) For any fixed $\lambda > 0$, we say that a square graph is a $\lambda$-*precision square graph* if the centers of any two squares are at least $\lambda$ distance apart.

Finally we recall some graph theoretic definitions for the sake of completeness.

**Maximum Independent Set (MIS):** Given a graph $G = (V, E)$, a maximum independent set for $G$ is a maximum cardinality subset $V'$ of $V$ such that for each $u, v \in V'$, $(u, v) \notin E$.

It is well known that the maximum independent set problem is NP-complete, even when restricted to square graphs [GJ79]. But as shown in [HM+94], if we are given a geometric specification of the squares the corresponding maximum independent set problem has a polynomial time approximation scheme (PTAS).

### 6.1 Overview

The basic idea of our $(\alpha, \beta)$ approximation algorithm is to reduce the MLUSP problem to the MLUS problem. Then following the approach in [DMMMZ97], we further reduce the MLUS problem to the problem of finding maximum independent sets in a collection of $\lambda$-precision unit square graphs.

A simple idea for the first reduction is to duplicate each site in the input to MLUSP and then solve for the MLUS problem. Unfortunately, the second reduction would then fail: As discussed in the above definition, if two squares touch each other, there will be an edge in the corresponding unit square graph. However, we need to label each site with a pair of squares, which certainly touch each other. Then there would be some problems in transforming these two squares into the vertices of the unit square graph — if possible, eventually we do not want to exclude one of these squares.

We deal with this problem by splitting a site $(x, y)$ into two 'dummy' sites: $(x - \delta, y - \delta)$ and $(x + \delta, y + \delta)$. We call the new problem of labeling these $2n$ sites with $2n$ uniform squares MLUS-2. We denote the new set of sites as $S_{2n}$.

**Theorem 9.** *Let $OPT_2$ be the optimal solution of MLUS-2 and let $OPT = L^*$ be the optimal solution of MLUSP. Then $OPT_2 \geq (OPT - 2\delta) \geq (1 - c \cdot \delta)OPT$, where $c$ is some constant.*

*Proof.* If we shrink $OPT$ by an additive factor of $2\delta$ then after we split (and translate) a site into two dummy sites any shrunk (and translated) square does not intersect with any other shrunk square. So $OPT - 2\delta$ gives a feasible solution to MLUS-2 and by the optimality of $OPT_2$ we have $OPT_2 \geq (OPT - 2\delta)$. By choosing a suitable constant $c$, $OPT_2 \geq (OPT - 2\delta) \geq (1 - c \cdot \delta)OPT$. □

Now having constructed an input for MLUS-2 from an input for MLUSP, all we need to do is to have a bicriteria approximation scheme for MLUS-2. As mentioned before, the basic idea is to reduce the MLUS-2 problem to finding maximum independent sets in a collection of unit square graphs as done similarly in [DMMMZ97]. Below, we outline the major idea in labeling at least $(1 - \epsilon)$ fraction of sites in $S_{2n}$ with squares of size at least $(1 - \epsilon)/(1 + \epsilon)$ of the optimal solution for MLUS-2, where $\epsilon$ is some small constant.

It was shown in [DMMMZ97] that the optimal solution for MLUS-2, $OPT_2$, is bounded above by $D_5(S_{2n})$ which is the min-5-diameter of $S_{2n}$ under $L_2$. Finding $D_5(S_{2n})$ takes only $O(n \log n)$ time by [EE94]. Therefore we can search for a good estimate of $OPT_2$ in the range $[0, D_5(S_{2n})]$ using powers of $(1 + \epsilon)$, where $\delta > \epsilon$. (Notice that we can scale this interval such that $OPT_2 > 1$ — this will only affect the running time of the algorithm by a constant factor. Therefore, without loss of generality, we assume that $OPT_2 > 1$.) At each stage we discretize the edge of a square into roughly $1/\epsilon$ points at which a (dummy) site can be located. Correspondingly, at each stage we change this problem into a maximum independent set problem in a $\delta\sqrt{2}$-precision unit square graph in which each vertex corresponds to a discretized unit square — a possible label for a site. Although this problem in NP-hard, it has been shown in [HM+94] that the maximum independent set problem for unit square graphs specified geometrically has a polynomial time approximation scheme. We use this as a subroutine to find a near optimal collection of non-overlapping squares to be placed at those feature points.

So we have successfully reduced the MLUS-2 problem into that of finding maximum independent sets of a number of square graphs. Specifically, given an

instance of $S_{2n}$, we construct $O(\log D_5(S_{2n}))$ square graphs, each of size polynomial in $2n$ and $1/\epsilon$. For each of these square graphs we obtain an approximate solution to the Maximum Independent Set problem — for which a polynomial-time approximation scheme is known [HM+94].

As $OPT_2 \leq D_5(S_{2n})$, there exists some iteration $k'$ such that $(1+\epsilon)^{k'} \leq OPT_2 \leq (1+\epsilon)^{k'+1}$. By [HM+94], in $O(\log D_5(S_{2n}))$ iterations we can find a placement of $(1-\epsilon)\cdot 2n$ squares whose size is at least $(1+\epsilon)^{k'} - \epsilon(1+\epsilon)^{k'}$. Clearly

$$(1+\epsilon)^{k'} - \epsilon(1+\epsilon)^{k'} \geq (1+\epsilon)^{k'}(1-\epsilon) \geq \frac{(1-\epsilon)}{(1+\epsilon)} \cdot OPT_2.$$

Consequently we have the following theorem.

**Theorem 10.** *For any fixed $\delta > \epsilon > 0$, given an instance of $n$ points of MLUSP, we can find a placement of at least $(1-2\epsilon)\cdot n$ square pairs with size of at least $(1-2\epsilon-c\delta+o(\epsilon\delta))OPT$ where $c$ is some constant and $OPT$ denotes the optimal solution.*

*Proof.* When approximating $OPT_2$, we fail to label at most $\epsilon\cdot 2n$ (dummy) sites. In the worst case this will destroy at most the same number of square pairs when we convert MLUS-2 back to MLUSP — which is simply done by translating the two squares associated with a pair of dummy sites back to the original site.

As we approximate $OPT_2$ with a factor of at least $\frac{(1-\epsilon)}{(1+\epsilon)}$, following Theorem 9, overall we approximate $OPT$ with a factor of $\frac{(1-\epsilon)}{(1+\epsilon)} \cdot (1 - c \cdot \delta)$, which is $(1-2\epsilon-c\cdot\delta+o(\epsilon\delta))$. In other words after we transform the approximate solution of MLUS-2 back to MLUSP we have an approximate scheme for MLUSP with performance guarantee of at least $(1-2\epsilon-c\delta+o(\epsilon\delta))OPT$. □

The running time of this algorithm is $O(n \log D_5(S_{2n}))$ as the algorithm for approximating the maximum independent set in a $\lambda$-precision unit square graph takes $O(n)$ time [HM+94]. The construction of input for MLUS-2 from input for MLUSP and the construction of output for MLUSP from output for MLUS-2 both takes linear time.

## 7 Concluding Remarks

One of the major questions regarding this paper is the computational complexity of the two problems we have just studied, namely the MLUSP and MLUCP. Are they NP-complete? The second question is whether the approximation constant factors we have achieved (i.e., 4 and 2) are optimal or not.

## References

BC94. D. Beus and D. Crockett. Automated production of 1:24,000 scale quadrangle maps. In *Proc. 1994 ASPRS/ACSM Annual Convention and Exposition*, volume 1, pages 94–99, 1994.

BO83. M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th ACM STOC*, pages 80–86, 1983.

CMS93. J. Christensen, J. Marks, and S. Shieber. Algorithms for cartographic label placement. In *Proc. 1993 ASPRS/ACSM Annual Convention and Exposition*, volume 1, pages 75–89, 1993.

CMS95. J. Christenson, J. Marks, and S. Shieber. An Empirical Study of Algorithms for Point-Feature Label Placement, *ACM Transactions on Graphics,*14:203-222, 1995.

DF92. J. Doerschler and H. Freeman. A rule-based system for cartographic name placement. *CACM*, 35:68–79, 1992.

DLSS. A. Datta, II.-P. Lenhof, C. Schwarz, and M. Smid. "Static and dynamic algorithms for k-point clustering problems," In *Proc. 3rd worksh. Algorithms and Data Structures,* springer-verlag, LNCS 709, pages 265-276, 1993.

DMMMZ97. S. Doddi, M. Marathe, A. Mirzaian, B. Moret and B. Zhu, Map labeling and its generalizations. In *Proc. 8th ACM-SIAM Symp on Discrete Algorithms (SODA'97)*, New Orleans, LA, Pages 148-157, Jan, 1997.

EE94. D. Eppstein and J. Erickson. "Iterated nearest neighbors and finding minimal polytopes," *Discrete & Comput. Geom.,* 11:321-350, 1994.

FW91. 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.

GJ79. M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, CA, 1979.

HM+94. H.B. Hunt III, M.V. Marathe, V. Radhakrishnan, S.S. Ravi, D.J. Rosenkrantz and R.E. Stearns, A Unified Approach to Approximation Schemes for NP- and PSPACE-Hard Problems for Geometric Graphs. *Proc. Second European Symp. on Algorithms (ESA'94)*, pages 468–477, 1994.

IL97. C. Iturriaga and A. Lubiw. Elastic labels: the two-axis case. In *Proc. Graph Drawing'97*, pages 181–192, 1997.

Imh75. E. Imhof. Positioning names on maps. *The American Cartographer*, 2:128–144, 1975.

Jon89. C. Jones. Cartographic name placement with Prolog. *Proc. IEEE Computer Graphics and Applications*, 5:36–47, 1989.

KSW98. M. van Kreveld, T. Strijk and A. Wolff. Point set labeling with sliding labels. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 337–346, 1998.

KT97. K. Kakoulis and I. Tollis. An algorithm for labeling edges of hierarchical drawings. In *Proc. Graph Drawing'97*, pages 169–180, 1997.

KT98a. K. Kakoulis and I. Tollis. A unified approach to labeling graphical features. *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 347–356, 1998.

KT98. K. Kakoulis and I. Tollis. On the multiple label placement problem. In *Proc. 10th Canadian Conf. on Comput. Geom.*, pages 66–67, 1998.

KR92. D. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM J. Disc. Math.*, 5:422–427, 1992.

PS85. F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

PZC98. C.K. Poon, B. Zhu and F. Chin, A polynomial time solution for labeling a rectilinear map. *Inform. Process. Lett.*, 65(4):201-207, Feb, 1998.

Wag94. F. Wagner. Approximate map labeling is in $\Omega(n \log n)$. *Inform. Process. Lett.*, 52:161–165, 1994.

# Approximation Algorithms in Batch Processing*

Xiaotie Deng[1], Chung Keung Poon[1], and Yuzhong Zhang[2]

[1] Department of Computer Science, City University of Hong Kong,
Hong Kong, P. R. China
{deng, ckpoon}@cs.cityu.edu.hk
[2] Institute of Operations Research, Qufu Normal University,
Qufu, Shandong, P. R. China
csyzzh@cityu.edu.hk

**Abstract.** We study the scheduling of a set of jobs, each characterised by a release (arrival) time and a processing time, for a batch processing machine capable of running (at most) a fixed number of jobs at a time. When the job release times and processing times are known *a-priori* and the inputs are integers, we obtained an algorithm for finding a schedule with the minimum makespan. The running time is pseudo-polynomial when the number of distinct job release times is constant. We also obtained a fully polynomial time approximation scheme when the number of distinct job release times is constant, and a polynomial time approximation scheme when that number is arbitrary. When nothing is known about a job until it arrives, i.e., the on-line setting, we proved a lower bound of $(\sqrt{5}+1)/2$ on the competitive ratio of any approximation algorithm. This bound is tight when the machine capacity is unbounded.

## 1 Introduction

Job scheduling on a batch processing system is an important issue in the manufacturing industry. As a motivating example, the manufacturing of very large scale integrated circuits requires a burn-in operation in which the integrated circuits are put into an oven in batches and heated for a prolong period (in terms of days) to bring out any latent defect. (See Lee [11] for the detailed process.) Since different circuits require different minimum baking time, the batching and scheduling of the integrated circuits is highly non-trivial and can greatly affect the production rate.

There are many variants of the batch processing problem. The basic setting of the problem we considered is as follows. We are given a set $J = \{1, \ldots, n\}$ of jobs to be processed. Each job $i$ is associated with a release time $r_i$, which specifies

* This research is partially supported by a grant from Hong Kong Research Grant Council and a grant from City University of Hong Kong as well as a grant from Natural Science Foundation of China and Shandong.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 153–162, 1999.
© Springer-Verlag Berlin Heidelberg 1999

when the job arrives, and a processing time $p_i$, which specifies the minimum time needed to process the job by a batch processor. A batch processor is capable of processing up to $B$ jobs simultaneously as a batch. The time, $\pi_I$, required to process a batch, $I$, is taken as the maximum processing time of an individual job in the batch, i.e., $\pi_I = \max_{i \in I} p_i$. Our goal is to find a schedule with the smallest *makespan*, $C_{max}$, which is the time to finish all the jobs in $J$. In this paper, we will consider the variant in which there is only one batch processor. We will adopt the notation of Graham et al. [8] and denote the problem as $1|r_j, B|C_{max}$. In comparison, the special case when all jobs arrive at the same time is denoted as $1|B|C_{max}$.

### 1.1 Previous Related Work

There has been a lot of research work ([9], [7], [6], [1], [11], [12], [5], [4], [13]) on similar or related scheduling problems. Lee and Uzsoy [10] advocated for the study of the $1|r_j, B|C_{max}$ problem. They gave an $O(n^2)$-time algorithm for the special case when $B = \infty$, extended it to polynomial time algorithms for other special cases, and proposed a pseudo-polynomial time algorithm when there are only two distinct release times. For the general $1|r_j, B|C_{max}$ problem, however, they only proposed a number of heuristics. The general problem was later shown to be strongly NP-hard by Brucker et al [3]. Along with other results, they presented an $O(n)$-time algorithm for $1|r_j, B = 1|C_{max}$, and improved Bartholdi's FBLPT algorithm [2] for the $1|B|C_{max}$ problem from $O(n \log n)$ to $\min\{O(n \log n), O(n^2/B)\}$.

### 1.2 Our Contributions

In this paper, we present several algorithms for the $1|r_j, B|C_{max}$ problem. First we gave an algorithm which computes, in time $O(mn(P_{max}P_{sum}B)^{m-1})$, a schedule with the minimum makespan, when there are $n$ jobs, $m$ distinct job release times and the maximum and sum of processing time(s) are $P_{max}$ and $P_{sum}$ respectively. For any constant $m$, the running time is pseudo-polynomial. This generalizes and improves an algorithm of Lee and Uzsoy [10] which takes pseudo-polynomial time when there are only two distinct release times. Note that both our algorithm and the Lee-and-Uzsoy algorithm only work for integer inputs. When there is no integral value restriction on the input, we obtain a fully polynomial time approximation scheme which computes a schedule with makespan at most $(1 + \epsilon)$ times the minimum makespan in time $O(m4^m n^{4m-3} B^{m-1}/\epsilon^{2m-2})$ for any $\epsilon > 0$. The time is thus polynomial in $n$ and $1/\epsilon$ for constant $m$. Finally, we extend this result to a polynomial approximation scheme which computes a $(1 + \epsilon)$-approximation in time $O(4^{2/\epsilon} n^{8/\epsilon+1} B^{2/\epsilon}/\epsilon^{4/\epsilon-2})$ for any $\epsilon > 0$. Hence the time is polynomial in $n$ (although exponentially in $1/\epsilon$).

In addition, the problem may be considered in the on-line setting where the job release times are unknown until they are released. We proved a lower bound of $(\sqrt{5}+1)/2$ on the competitive ratio of any approximation algorithm. We show that the lower bound is tight for the case when $B = \infty$ by exhibiting an approximation algorithm that achieves the same ratio.

The organisation of the rest of the paper is as follows. In section 2, we provide the basic concepts and the FBLPT algorithm. We then present the pseudo-polynomial time exact algorithm in section 3 and the two polynomial time approximation schemes in section 4. We turn to the on-line version of the problem in section 5. We present a lower bound for the problem and a matching upper bound for the special case when the batch machine has infinite capacity, i.e., $B = \infty$. Finally, in section 6, we discuss the future direction of research in this area. Due to space limitation, most of the proofs are omitted.

## 2 Terminology and the FBLPT Algorithm

An algorithm $A$ is a $(1+\epsilon)$-*approximation algorithm* for a minimization problem if it produces a solution which is at most $(1+\epsilon)$ times the optimal one. A family of algorithms $\{A_\epsilon\}$ is called a *polynomial time approximation scheme* if, for every $\epsilon > 0$, the algorithm $A_\epsilon$ is a $(1+\epsilon)$-approximation algorithm running in time polynomial in the input size when $\epsilon$ is treated as constant. It is called a *fully polynomial time approximation scheme* if the running time is also polynomial in $1/\epsilon$. Throughout this paper, we will denote by $\lfloor x \rfloor$ the largest integer smaller than or equal to $x$.

Given a set of jobs $J$, we define $P_{max}$ and $r_{max}$ as the maximum processing time and the maximum release time of a job respectively: $P_{max} = \max_{j\in J} p_j$ and $r_{max} = \max_{j\in J} r_j$. We define $P_{sum}$ as the total processing time: $P_{sum} = \sum_{j\in J} p_j$. A schedule for $J$ is a sequence $(J_1, t_1), (J_2, t_2), \cdots, (J_k, t_k)$ such that $\{J_1, J_2, \cdots, J_k\}$ is a partition of $J$ such that, for all $i = 1, 2, \cdots, k$, $|J_i| \leq B$; $\forall j \in J_i : r_j \leq t_i$; $t_i + \pi_i \leq t_{i+1}$ where $\pi_i = \max\{p_j, j \in J_i\}$. The makespan of the schedule is $t_k + \pi_k$. Obviously, the minimum makespan, $C^*$, over all schedules must be at least $r_{max}$ and $P_{sum}/B$: $C^* \geq \max\{P_{sum}/B, r_{max}\}$.

Before explaining our algorithms, we first describe the FBLPT (Full Batch Largest Processing Time) algorithm of Bartholdi [2] which computes the optimal solution for the $1|B|C_{max}$ problem.

**Algorithm** FBLPT

Step 1. Sort the jobs according to their processing times and re-order the indices so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

Step 2. Form batches by placing jobs $iB + 1$ through $(i + 1)B$ together in the same batch for $i = 0, 1, \ldots, \lfloor n/B \rfloor$.

Step 3. Schedule the batches in any arbitrary order.

The schedule contains at most $\lfloor n/B \rfloor + 1$ batches and all batches are full except possibly the last one. From now on, a schedule is said to *follow the FBLPT scheduling* if it groups the jobs according to the FBLPT algorithm and schedules them in non-increasing order of processing time.

## 3 A Pseudo-polynomial Time Algorithm for a Special Case

In this section, we restrict our discussion to cases where the release times and processing times are integers. Let the set of jobs $J$ have $m$ distinct job release times, $0 = r'_1 < r'_2 < \ldots < r'_m$. Thus, there are jobs with the same release times when $n > m$. For technical reasons, we also define $r'_{m+1} = \infty$. We index the jobs in non-increasing order of processing time, i.e., $p_1 \geq p_2 \geq \cdots \geq p_n$.

Given a schedule, $S$, for the set of jobs $J$, we can partition $J$ into $m$ disjoint subsets, $J_1(S), \ldots, J_m(S)$, such that a job is in $J_i(S)$ if and only if it is scheduled at or after time $r'_i$ but strictly before $r'_{i+1}$ according to $S$. A key observation is that we can locally rearrange the schedule of each subset $J_i(S)$, without increasing its makespan, so that it follows the FBLPT scheduling.

**Lemma 1.** *For any schedule $S$ for $J$ with makespan $C_{max}$, there exists a schedule $S'$ for $J$ with makespan $C'_{max}$ such that $C'_{max} \leq C_{max}$; and that $J_i(S) = J_i(S')$ and the schedule for $J_i(S')$ according to $S'$ follows the FBLPT scheduling for any $i$.*

We first describe the algorithm for computing the minimum makespan. The adaption to finding the schedule itself is straightforward. For every $1 \leq i \leq m$, if $J_i(S)$ is non-empty, then there is a start time $t \geq r'_i$ at which the first batch of $J_i(S)$ is started. Inequality happens when the last batch of $J_{i-1}(S)$ finished after time $r'_i$ (even though it started before time $r'_i$). The maximum delay of the first batch of $J_i(S)$ is at most $P_{max}$. We define the *delay time* of $J_i(S)$, denoted by $b_i(S)$, as the start time of the first batch of $J_i(S)$ minus $r'_i$. Thus $0 \leq b_i(S) \leq P_{max}$. We also define the *completion time* of $J_i(S)$, denoted by $c_i(S)$, as the time needed to complete all the jobs in $J_i(S)$. Thus $0 \leq c_i(S) \leq P_{sum}$. We define the *size* of $J_i(S)$, denoted by $n_i(S)$, as the number of jobs modulo $B$ in the last batch in $J_i(S)$. ($n_i(S)$ ranges from 0 to $B-1$.) Intuitively, $B - n_i(S) \bmod B$ is the number of jobs, with sufficiently small processing time, that can be added to the last batch without increasing the completion time of $J_i(S)$. If $J_i(S)$ is empty, we simply define $b_i(S)$, $c_i(S)$ and $n_i(S)$ as zero. Finally, we define the makespan of a schedule $S$ for jobs $\{1, \ldots, j\}$ as $r'_m + b_m(S) + c_m(S)$. Hence, the makespan is always at least $r'_m$. For every possible $\boldsymbol{B} = (b_2, \ldots, b_m)$, $\boldsymbol{C} = (c_1, \ldots, c_{m-1})$

and $\boldsymbol{N} = (n_1, \ldots, n_{m-1})$, we define $f(j, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$ as the minimum makespan of a schedule $S$ for jobs $\{1, \ldots, j\}$ such that

1. for $2 \le i \le m$, $J_i(S)$ has delay time $b_i$; and
2. for $1 \le i \le m-1$, $J_i(S)$ has completion time $c_i$ and size $n_i$;

(We can always assume $b_1(S) = 0$ and $n_m(S)$ can be computed by $j - n_1(S) - \cdots - n_{m-1}(S) \bmod B$.) Our algorithm uses a dynamic programming approach to compute $f(j, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$ for each possible $(j, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$. First, let us consider the value of $f(1, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$. Suppose job 1 has release time $r'_i$ for some $i$. Then it can only be scheduled at or after time $r'_i$. Furthermore, if it is scheduled in the time interval $[r'_k, r'_{k+1})$ for some $i \le k \le m$, then $J_k(S)$ contains only job 1 and all the other $J_i(S)$'s are empty. Also $J_k(S)$ can be completed at time $r'_k + b_k(S) + p_1$. Hence for $i \le k \le m$ and for all $b_k$, if we let

$$\boldsymbol{B}_k = (\overbrace{0, \ldots, 0, b_k}^{k-1}, \overbrace{0, \ldots, 0}^{m-k})$$
$$\boldsymbol{C}_k = (\overbrace{0, \ldots, 0, b_k + p_1}^{k}, \overbrace{0, \ldots, 0}^{m-k-1})$$
$$\boldsymbol{N}_k = (\overbrace{0, 0, \ldots, 1,}^{k} \overbrace{0, \ldots, 0}^{m-k-1}),$$

and take $\boldsymbol{B}_1 = \boldsymbol{B}$, $\boldsymbol{C}_m = \boldsymbol{C}$ and $\boldsymbol{N}_m = \boldsymbol{N}$, then we have

$$f(1, \boldsymbol{B}_k, \boldsymbol{C}_k, \boldsymbol{N}_k) = \max\{r'_k + b_k + p_1, r'_m\}.$$

For the other possible values of $(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$, we set $f(1, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N}) = \infty$.

Now suppose the value of $f(j', \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$ has been computed for every $j' < j$ and every possible $(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$. Consider $f(j, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$. Again, let job $j$ have release time $r_j = r'_i$ for some $i$. Then it can only be scheduled at or after time $r'_i$. Suppose it is inserted into $J_k(S)$ for some $k$ where $i \le k \le m$ by some schedule $S$. By Lemma 1, we can assume that $J_k(S)$ follows the FBLPT scheduling. Since job $j$ has the smallest processing time among all jobs in $J_k(S)$, it is inserted into the last batch of $J_k(S)$.

Imagine that the schedule for jobs $\{1, \ldots, j-1\}$ has been determined by $S$ and now job $j$ is added to the existing schedule. There are $n_k$ jobs in the last batch of $J_k(S)$ after adding job $j$. If $n_k > 1$, then the last batch of $J_k(S)$ is non-empty before inserting job $j$. Hence adding job $j$ does not increase the completion time of $J_k(S)$. (Remember that the processing time, $p_j$, of job $j$ is no more than the processing time of the last batch of $J_k(S)$ by the order we consider the jobs.) If $n_k = 1$, then the last batch of $J_k$ is either full or does not exist at all before inserting job $j$. Hence inserting job $j$ increases the completion time of $J_k$ by $p_j$. If $k = m$, then the makespan will also be increased.

We are now ready to describe the recursive relation for the dynamic programming. Recall that $\boldsymbol{C} = (c_1, \ldots, c_{m-1})$ and $\boldsymbol{N} = (n_1, \ldots, n_{m-1})$. Then for $i \leq k \leq m$, define $\boldsymbol{C}_k = (c_1, \ldots, c_{k-1}, c_k - p_j, c_{k+1}, \ldots, c_{m-1})$, $\boldsymbol{N}_k = (n_1, \ldots, n_{k-1}, n_k - 1 \bmod B, n_{k+1}, \ldots, n_{m-1})$. For $i \leq k < m$, define

$$f_k = \begin{cases} f(j-1, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N}_k) & \text{if } n_k > 1 \\ f(j-1, \boldsymbol{B}, \boldsymbol{C}_k, \boldsymbol{N}_k) & \text{if } n_k = 1 \text{ and } c_k \geq p_j \text{ and } k < m \\ \infty & \text{if } n_k = 1 \text{ and } c_k < p_j \end{cases}$$

and

$$f_m = \begin{cases} f(j-1, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N}) & \text{if } n_m > 1 \\ f(j-1, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N}) + p_j & \text{if } n_m = 1 \end{cases}$$

Then we have

$$f(j, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N}) = \min\{\, f_k \mid i \leq k \leq m \,\}$$

The minimum makespan of the original problem can be computed by taking the minimum $f(n, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$ over all possible $(\boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$. As described before, $0 \leq b_i \leq P_{max}$, $0 \leq c_i \leq P_{sum}$ and $0 \leq n_i \leq B - 1$. Therefore, there are $O(n(P_{max}P_{sum}B)^{m-1})$ sets of possible input values. For each set, it takes $O(m)$ time to compute the value of $f(j, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{N})$. Hence the algorithms takes time $O(mn(P_{max}P_{sum}B)^{m-1})$. If the number of distinct job release times, $m$, is constant, then the algorithm runs in pseudo-polynomial time. Hence the $1|r_j, B|C_{max}$ problem is not strong NP-hard in this special case.

## 4 Approximating Optimal Makespan in Polynomial Time

The previous algorithm solves the $1|r_j, B|C_{max}$ problem optimally when the input values are integers. We now extend this algorithm to a fully polynomial time approximation scheme for the same $1|r_j, B|C_{max}$ problem but without the integral value restriction. The algorithm AMM (Approximate-Minimum-Makespan) accepts as input a set of jobs $J$ with release times and processing times, and an accuracy parameter $\epsilon$; and outputs a schedule with makespan at most $(1 + \epsilon)$ times the minimum makespan.

**Algorithm** AMM$(J, \epsilon)$

Step 1. Let $M_0 = \max\{r_{max}, P_{max}\}$ and $M = \frac{M_0}{2n}\epsilon$.

Step 2. Round down all numbers which appeared in the input to the nearest multiples of $M$ to obtain another input instance $\tilde{J} = \{1, \ldots, n\}$ such that job $j$ has release time $\tilde{r}_j = \lfloor r_j/M \rfloor$ and processing time $\tilde{p}_j = \lfloor p_j/M \rfloor$.

Step 3. Compute an optimal schedule $S$ for the rounded down input $(\tilde{J}, B)$ using the algorithm in previous section.

Step 4. Return $S$ as the schedule for the original input.

Note that when applying schedule $S$ on both $J$ and $\tilde{J}$, the start time of each job in $J$ may be later than that of the corresponding job in $\tilde{J}$. This is because each number in the instance $\tilde{J}$ can be smaller than the corresponding number in the instance $J$.

**Theorem 1.** *The* AMM *algorithm is a fully polynomial approximation scheme for the* $1|r_j, B|C_{max}$ *problem when there are a constant number of distinct job release times.*

Next we extend the above algorithm to a polynomial time approximation scheme for the $1|r_j, B|C_{max}$ problem in which the number of distinct release times, $m$, is non-constant. The idea is to approximate the (at most) $n$ distinct release times using a constant number of distinct release times.

**Algorithm** $\text{AMM}_2(J, \epsilon)$

Step 1. Let $K = (\epsilon/2)r_{max}$.
Step 2. Round down each $r_j$ to the nearest multiple of $K$, that is, set $\tilde{r_j} = K\lfloor r_j/K \rfloor$. Note that there will be at most $2/\epsilon$ distinct release times in the rounded down problem.
Step 3. Use the $(1 + \epsilon/2)$-approximation scheme for the rounded problem to obtain a schedule.
Step 4. Schedule the jobs for the original problem by the ordering obtained in step 3.

**Theorem 2.** *The* $\text{AMM}_2$ *algorithm is a polynomial time approximation scheme for the general* $1|r_j, B|C_{max}$ *problem.*

## 5 The On-Line Scheduling Problem

In this section, we study the on-line version of the problem. In particular, we consider the situation in which we have absolutely no information about the jobs. We do not know how many jobs there are and for each job $i$, we do not know its processing time $p_i$ until it arrives, i.e., at time $r_i$ which also is unknown until the job arrives. We also do not allow pre-empting a scheduled batch. (Note that allowing pre-emption could help a lot.) Due to the lack of information concerning the future, no on-line algorithm performs very well in the worst case. We found that the worst case competitive ratio of any on-line algorithm is at least $(\sqrt{5}+1)/2$, the golden ratio. For convenience, we will let $\delta = (\sqrt{5}-1)/2$ throughout this section.

**Theorem 3.** *Any on-line algorithm has a worst-case competitive ratio of at least* $1 + \delta = (\sqrt{5} + 1)/2 \approx 1.618$.

**Proof:** (Sketch) We will construct a worst case input with one or two jobs according to the actions taken by the algorithm. Job 1 with processing time $p_1 = 1$ will arrive at time zero. Then the on-line algorithm will schedule it at some time $s_1$ (which could be zero). If $s_1 \geq \delta$, then the adversary will not give any more job. If $s_1 < \delta$, then the adversary will give a second job with processing time $p_2 = 1$, which arrives at time $s_1 + \epsilon$ for some small value $\epsilon > 0$. In both cases, the competitive ratio is at least $1 + \frac{1-\epsilon}{1+\epsilon}\delta$. □

**Theorem 4.** *When the capacity $B$ of the machine is infinite, there exists an on-line algorithm with competitive ratio* $1 + \delta$.

**Proof:** The idea of the algorithm is as follows. At any current moment $t$, the algorithm maintains a *commence* time $S(t) = \max_{j \in U(t)}\{(1+\delta)r_j + \delta p_j\}$ where $U(t)$ is the set of jobs available and remained unscheduled at time $t$. If no new jobs arrived before $S(t)$, then it will schedule all jobs of $U(t)$ to run at time $S(t)$. Otherwise, if a job $i$ arrives at time $t'$ such that $t < t' \leq S(t)$, then it computes the possibly new commence time $S(t') = \max_{j \in U(t')}\{(1+\delta)r_j + \delta p_j\}$ where $U(t') = U(t) \cup \{i\}$. Then it waits until time $S(t')$ to schedule all jobs in $U(t')$ or until another new job arrived upon which it will change its plan again.

To see that the algorithm achieves the stated competitve ratio, consider the last batch in the schedule obtained by the algorithm and trace backward in time to the first idle time (or to the first batch of the schedule if there is no idle time). Let the sequence of batches be $(B_1, B_2, \ldots, B_k)$ where $B_1$ is the earliest batch and $B_k$ the last batch in this sequence. Let $s_i$ be the start time of batch $B_i$. Obviously, $s_i < s_{i+1}$ for $1 \leq i < k$. Let job $i$ be the one that maximizes $(1+\delta)r_j + \delta p_j$ over all jobs $j$ in $B_i$. (Recall that $r_i$ and $p_i$ are the release time and processing time of job $i$ respectively.) By construction of our algorithm, the batch $B_1$ started exactly at time $(1+\delta)r_1 + \delta p_1$ and a subsequent batch $B_i$, $i > 1$, may need to wait until the previous batch $B_{i-1}$ finished. Thus, we have

$$s_1 = (1+\delta)r_1 + \delta p_1,$$
$$\text{and} \quad s_i \geq (1+\delta)r_i + \delta p_i \quad \text{for } 2 \leq i \leq k.$$

Also, job $i+1$ must arrive after batch $B_i$ has been started. Therefore,

$$r_{i+1} > s_i \quad \text{for } 2 \leq i \leq k.$$

Combining these we obtain the following lower bound on the $s_i$'s:

$$\begin{aligned} s_k &\geq (1+\delta)r_k + \delta p_k \\ &> (1+\delta)s_{k-1} + \delta p_k \\ &\geq (1+\delta)^2 r_{k-1} + (1+\delta)\delta p_{k-1} + \delta p_k \\ &> (1+\delta)^k r_1 + (1+\delta)^{k-1}\delta p_1 + \cdots + (1+\delta)\delta p_{k-1} + \delta p_k \end{aligned}$$

Since the machine is busy from time $s_1$ onwards until all the batches $B_1, \ldots, B_k$ are finished, the makespan of our algorithm is

$$\begin{aligned} C_{max} &= s_1 + p_1 + p_2 + \cdots + p_k \\ &= (1+\delta)(r_1 + p_1) + p_2 + \cdots + p_k \end{aligned}$$

On the other hand, the minimum makespan $C^*$ has to satisfy

$$\begin{aligned} C^* &\geq r_k + p_k \\ &> s_{k-1} + p_k \\ &> (1+\delta)^{k-1} r_1 + \delta p_{k-1}(1+\delta)\delta p_{k-2} + \cdots + (1+\delta)^{k-2}\delta p_1 + p_k \end{aligned}$$

Therefore we have

$$\frac{C}{C^*} \leq \frac{(1+\delta)r_1 + (1+\delta)p_1 + p_2 + p_3 + \cdots + p_k}{(1+\delta)^{k-1} r_1 + (1+\delta)^{k-2}\delta p_1 + \cdots + (1+\delta)\delta p_{k-2} + \delta p_{k-1} + p_k}$$

If $k = 1$, then

$$\begin{aligned} \frac{C}{C^*} &\leq \frac{(1+\delta)r_1 + (1+\delta)p_1}{r_1 + p_1} \\ &\leq 1 + \delta \end{aligned}$$

If $k \geq 3$, then observe that $(1+\delta)r_1/(1+\delta)^{k-1} r_1 \leq 1$, $(1+\delta)p_1/(1+\delta)^{k-2}\delta p_1 \leq 1+\delta$, $\ldots, p_k/p_k \leq 1+\delta$ making use of the fact that $\delta(1+\delta) = 1$.

For $k = 2$, we need an additional observation. The optimal schedule has to schedule the 2 batches $B_1$ and $B_2$ together at time $r_2$, if it is to behave differently from our algorithm. Therefore, $C^* \geq r_2 + \max\{p_1, p_2\}$. If $p_1 \leq p_2$, then

$$\frac{C_{max}}{C^*} \leq \frac{(1+\delta)p_1 + p_2}{\delta p_1 + p_2} \leq 1 + \delta.$$

If $p_1 > p_2$, then

$$\frac{C_{max}}{C^*} \leq \frac{(1+\delta)p_1 + p_2}{\delta p_1 + p_1} \leq \frac{(2+\delta)p_1}{(1+\delta)p_1} \leq 1 + \delta.$$

Hence

$$\frac{C}{C^*} < 1 + \delta.$$

in any case. □

## 6 Remarks and Discussion

Batch processing is an interesting problem with practical applications which has started to attract studies in its algorithmic aspects recently. A lot of questions

are still open and more research efforts are needed. The on-line version of the problem is particularly interesting. While we have the exact bound for the case $B = \infty$, the gap has yet to be bridged when $B$ is bounded. One can easily obtain a 2-competitive ratio algorithm in this case but the best lower bound is $1 + \delta$. Our conjecture is that the exact bound is $1 + \delta$, the golden ratio.

## Acknowledgment

The authors would like to express their thanks to Prof. Jie-ye Han for his inspiring comments and discussion in the early stage of this work.

## References

1. J.H. Ahmadi, R.H. Ahmadi, S. Dasu, and C.S. Tang. Batching and scheduling jobs on batch and discrete processors. *Operations Research*, 40:750–763, 1992.
2. J.J. Bartholdi. unpublished manuscript, 1988.
3. 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*, 1:31–54, 1998.
4. V. Chandru, C.Y. Lee, and R. Uzsoy. Minimizing total completion time on a batch processing machine with job families. *Operations Research Letters*, 13:61–65, 1993.
5. V. Chandru, C.Y. Lee, and R. Uzsoy. Minimizing total completion time on batch processing machines. *International Journal of Production Research*, 31:2097–2121, 1993.
6. G. Dobson and R.S. Nambinadom. The batch loading scheduling problem. Technical report, Simon Graduate School of Business Administration, University of Rochester, 1992.
7. C.R. Glassey and W.W. Weng. Dynamic batching heuristics for simultaneous processing. *IEEE Transactions on Semiconductor Manufacturing*, pages 77–82, 1991.
8. R.L. Graham, Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling. *Annals of Discrete Mathematics*, 5:387–326, 1979.
9. Y. Ikura and M. Gimple. Scheduling algorithm for a single batch processing machine. *Operations Research Letters*, 5:61–65, 1986.
10. C.Y. Lee and R. Uzsoy. Minimizing makespan on a single batch processing machine with dynamic job arrivals. Technical report, Department of Industrial and System Engineering, University of Florida, January 1996.
11. C.Y. Lee, R. Uzsoy, and L.A. Martin Vega. Efficient algorithms for scheduling semiconductor burn-in operations. *Operations Research*, 40:764–775, 1992.
12. C.L. Li and C.Y. Lee. Scheduling with agreeable release times and due dates on a batch processing machine. *European Journal of Operational Research*, 96:564–569, 1997.
13. R. Uzsoy. Scheduling batch processing machines with incompatible job families. *International Journal of Production Research*, pages 2605–2708, 1995.

# LexBFS-Ordering in Asteroidal Triple-Free Graphs*

Jou-Ming Chang[1,3], Chin-Wen Ho[1], and Ming-Tat Ko[2]

[1] Institute of Computer Science and Information Engineering, National Central University, Chung-Li, Taiwan, ROC
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC
[3] Department of Information Management, National Taipei College of Business, Taipei, Taiwan, ROC

**Abstract.** In this paper, we study the metric property of LexBFS-ordering on AT-free graphs. Based on a 2-sweep LexBFS algorithm, we show that every AT-free graph admits a vertex ordering, called the strong 2-cocomparability ordering, that for any three vertices $u \prec v \prec w$ in the ordering, if $d(u, w) \leq 2$ then $d(u, v) = 1$ or $d(v, w) \leq 2$. As an application of this ordering, we provide a simple linear time recognition algorithm for bipartite permutation graphs, which form a subclass of AT-free graphs.

## 1 Introduction

In past years, specific orderings of vertices characterizing certain graph classes are studied by many researchers. Usually, these ordering can be described from a metric point of view and the metric associated with a connected graph is, of course, the *distance function* $d$, giving the length of a shortest path between two vertices. One of the first results is due to Rose [19] for recognizing chordal graphs. A graph is *chordal* if every cycle of length at least four has a chord. A proof in [19] showed that every chordal graph $G$ admits a *perfect elimination ordering*, i.e., an ordering $v_1, v_2, \ldots, v_n$ of the vertices of $G$ such that for every $i < j < k$, if $d(v_i, v_j) = d(v_i, v_k) = 1$, it implies that $d(v_j, v_k) = 1$. Here the vertex $v_i$, $i = 1, \ldots, n$, is called a *simplicial vertex* of the subgraph of $G$ induced by the set $\{v_i, \ldots, v_n\}$. Note that a vertex is simplicial in a graph if and only if it is not a midpoint of every induced path on three vertices. The perfect elimination ordering of a chordal graph can be computed in linear time from the Lexicographic Breadth-First Search (LexBFS) algorithm [20] or the Maximum Cardinality Search (MCS) algorithm [23].

Another two well-known classes of graphs are comparability graphs and co-comparability graphs. A graph $G$ is a *comparability graph* if its vertex set has a *transitive ordering*; i.e., an ordering $v_i, v_2, \ldots, v_n$ of the vertices of $G$ such that for every $i < j < k$, if $d(v_i, v_j) = d(v_j, v_k) = 1$, then it implies that $d(v_i, v_k) = 1$. There is an $O(n^2)$ time algorithm [21] to test if a graph is a comparability

* This work was supported by the National Science Council, Republic of China under grant NSC89-2213-E-008-007.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 163–172, 1999.
© Springer-Verlag Berlin Heidelberg 1999

graph. In the case of a positive answer, the algorithm also produces a transitive ordering. A *cocomparability graph* is the complement of a comparability graph, or equivalently, its vertex set has a *cocomparability ordering*, i.e., an ordering $v_i, v_2, \ldots, v_n$ of the vertices such that for every $i < j < k$, if $d(v_i, v_k) = 1$, it implies that $d(v_i, v_j) = 1$ or $d(v_j, v_k) = 1$. Most efficient algorithms on comparability and cocomparability graphs are developed from these vertex orderings.

Besides, some generalizations of these well-known orderings have been investigated. Jamison and Olariu [12] generalized the concept of perfect elimination ordering in the following way: a vertex is *semi-simplicial* in a graph if it is not a midpoint of every induced path on four vertices. An ordering $v_1, \ldots, v_n$ of the vertices in a graph $G$ is a *semi-perfect elimination ordering* if and only if $v_i$ is semi-simplicial in the subgraph of $G$ induced by the set $\{v_i, \ldots, v_n\}$. They also characterized the classes of graphs for which every ordering produced by LexBFS or MCS is a semi-perfect elimination ordering. Hoàng and Reed [11] defined that a graph $G$ is $P_4$-*comparability* if it admits a vertex ordering which is transitive when restricted to any $P_4$. Thus, this class of graphs generalizes comparability graphs in a natural way. In [4], we generalized the concept of cocomparability ordering as follows. Let $t \geq 1$ be an integer. A $t$-*cocomparability ordering* (abbr. $t$-CCPO) of a graph $G$ is an ordering $v_1, v_2, \ldots, v_n$ of the vertices such that for every $i < j < k$, if $d(v_i, v_k) \leq t$, it implies that $d(v_i, v_j) \leq t$ or $d(v_j, v_k) \leq t$. A graph is called a $t$-*cocomparability graph* if it admits a $t$-CCPO. Thus, $G$ is a $t$-cocomparability graph if and only if every powers $G^s$ for $s \geq t$ is a cocomparability graph (where $G^s$ is the graph with the same vertex set as $G$ such that two vertices are adjacent if and only if their distance in $G$ is at most $s$). Indeed, a proof in [4] showed that a vertex ordering of $G$ is a $t$-CCPO if and only if it is a cocomparability ordering of $G^s$ for $s \geq t$, and determining a graph to be a $t$-cocomparability graph for the smallest integer $t$ can be solved in $\mathrm{O}(M(n) \log n)$ time, where $M(n)$ denotes the time complexity of multiplying two $n \times n$ matrices for integers. In particular, the 2-cocomparability graphs can be recognized in $\mathrm{O}(M(n))$ time.

An *asteroidal triple* (or AT for short) of a graph is an independent set of three vertices such that every pair of vertices are joined by a path avoiding the closed neighborhood of the third. Graphs without asteroidal triple are called *AT-free graphs*. Lekkerkerker and Boland [17] first introduced the concept of asteroidal triples to characterize the interval graphs. A graph is an *interval graph* if and only if it is chordal and AT-free. Golumbic et al. [10] showed that cocomparability graphs (and thus permutation and trapezoid graphs) are also AT-free. The best known recognition algorithm for AT-free graphs requires $\mathrm{O}(n^3)$ time [7]. The polynomial time algorithms for solving stability and various domination-type problems on AT-free graphs can be found in [3,6,8,15]. Besides, some algorithmic problems such as treewidth [13], minimum fill-in [13] and vertex ranking [14] on AT-free graphs are known to be NP-complete.

Recently, Corneil, Olariu and Stewart [7] obtained a collection of interesting structural properties for AT-free graphs. However, up to now nice characterizations of AT-free graphs such as a geometric intersection model and an elimination

scheme are not known. It would be interesting to see whether the AT-free graphs also possess some vertex ordering which is useful for algorithmic purposes. Based on a decomposition property, called the *involutive sequence*, proposed by Corneil et al., in [4] we showed that every AT-free graph possesses a 2-CCPO, and the class of AT-free graphs is properly contained in the class of 2-cocomparability graphs. Consequently, every proper powers $G^k$ $(k \geq 2)$ of an AT-free graph $G$ is a cocomparability graph. This result implies that the $k$-domination and $k$-stability problems for $k \geq 2$ on AT-free graphs can be solved by a more efficient way.

In this paper, we continue this work by investigating the metric property of LexBFS-ordering on AT-free graphs. We prove that every AT-free graph has a vertex ordering, called the *strong* 2*-CCPO*, which is stronger than the 2-CCPO and is also a generalization of the cocomparability ordering. In particular, we show that this ordering can be generated by a 2-sweep LexBFS algorithm. The concept of strong 2-CCPO and the 2-sweep LexBFS will be introduced in the next section. Moreover, based on a modified 2-sweep LexBFS algorithm, we show that the strong 2-CCPO can be used to design a recognition algorithm for bipartite permutation graphs in $O(n+m)$ time. This approach is easier than the algorithm of [22].

## 2 AT-Free Graphs Are Strong 2-Cocomparability Graphs

All graphs considered in this paper are undirected, simple (i.e., without loops and multiple edges) and connected. Let $G = (V, E)$ be a graph with vertex set $V$ of size $n$ and edge set $E$ of size $m$. The *distance* of two vertices $u, v \in V$, denoted by $d_G(u, v)$, is the number of edges of a shortest path from $u$ to $v$ in $G$. When no ambiguity arises, the subscript $G$ can be omitted. A path joining two vertices $u$ and $v$ is termed a *$u$-$v$ path*. The union of two paths $P$ and $P'$ with a common endpoint is denoted by $P \oplus P'$. A vertex $u$ *misses* a path $P$ if there are no vertices of $P$ adjacent to $u$; otherwise, we say that $u$ *intercepts* $P$. The *open neighborhood* $N(u)$ of a vertex $u \in V$ is the set $\{v \in V : (u, v) \in E\}$; and the *closed neighborhood* $N[u]$ is $N(u) \cup \{u\}$. Notations and terminologies not given here may be found in any standard textbook on graphs and algorithms.

We first introduce the notion about the strong $t$-cocomparability ordering. Let $t \geq 1$ be an integer. A *strong $t$-cocomparability ordering* (abbr. strong $t$-CCPO) of a graph $G$ is an ordering $v_1, v_2, \ldots, v_n$ of $V$ such that for every $i < j < k$, if $d(v_i, v_k) \leq t$, it implies that $d(v_i, v_j) = 1$ or $d(v_j, v_k) \leq t$. A graph is called a *strong $t$-cocomparability graph* if it admits a strong $t$-CCPO. Clearly, every cycle $C_{t+3}$ $(t \geq 1)$ admits a strong $t$-CCPO, and every strong $t$-cocomparability graph is a $t$-cocomparability graph, but the converse is not true. For instance, the even cycle $C_{2t+2}$ $(t \geq 2)$ is a $t$-cocomparability graph and it has no strong $t$-CCPO. Also, it is easy to see that all the classes of strong $t$-cocomparability graphs, $t \geq 1$, constitute a hierarchy by sets inclusion.

The LexBFS was designed to provide a linear-time recognition algorithm for chordal graphs [20]. According to LexBFS, the vertices of a graph $G$ are numbered from $n$ to 1 in decreasing order. The label $L(u)$ of an unnumbered

vertex $u$ is the list of its numbered neighbors in the current search. As the next vertex to be numbered, select the vertex $v$ with the lexicographically largest label, breaking ties arbitrarily. The algorithm runs in $O(n+m)$ time. Below we reproduce the details of LexBFS that begins from a distinguished vertex $v$ of $G$.

**Procedure** LexBFS$(G, v)$
*Input:* a connected graph $G = (V, E)$ and a vertex $v \in V$.
*Output:* a numbering $\sigma$ of the vertices of $G$.
**begin**
  $L(v) \leftarrow n$;
  **for** each vertex $u \in V \backslash \{v\}$ **do** $L(u) \leftarrow \emptyset$;
  **for** $i \leftarrow n$ **downto** 1 **do begin**
    choose an unnumbered vertex $u$ with the lexicographically largest label;
    $\sigma(u) \leftarrow i$; {assign the number $i$ to vertex $u$}
    **for** each unnumbered vertex $w \in N(u)$ **do**
      append $i$ to $L(w)$;
  **end**
**end.**

Let $(V, \prec)$ be the vertex ordering corresponding to the numbering $\sigma$ produced by LexBFS$(G, v)$. For any two vertices $a, b \in V$, we write $a \prec b$ (or $b \succ a$) whenever $\sigma(a) < \sigma(b)$ and we shall say that $b$ is *larger* than $a$ or $a$ is *smaller* than $b$. In addition, if $a$ is no larger than $b$, we simply write $a \preceq b$ (or $b \succeq a$). For convenience, the subgraph of $G$ induced by the vertex set $\{w \in V : w \succeq u\}$ is denoted by $G[u]$. Note that, if $G$ is a connected graph, then $G[u]$ is also connected for every $u \in V$.

A *dominating pair* in a graph is a pair of vertices, such that for any path connecting these two vertices, all remaining vertices of the graph are in the neighborhood of this path. Recently, an algorithm called the 2-*sweep LexBFS* was proposed by Corneil et al. [8] for finding dominating pairs of a given connected AT-free graph $G = (V, E)$. Their algorithm works as follows. First, start a LexBFS from an arbitrary vertex $v \in V$. Let $z$ be the vertex with the smallest number in this search. Again, start a new LexBFS from $z$. Let $(V, \prec) = (y \prec \cdots \prec z)$ be the ordering of vertices produced by LexBFS$(G, z)$. It is shown in [8] that $y$ and $z$ constitute a dominating pair of $G$. The correctness of their algorithm is based on the following property.

**Lemma 1.** (Corneil et al. [8]) *Let $G = (V, E)$ be an AT-free graph and $(V, \prec)$ be the vertex ordering of $G$ produced by a 2-sweep LexBFS. Then, for every two vertices $u \prec w$, $w$ intercepts all $u$-$z$ paths in $G$, where $z$ is the largest vertex of $(V, \prec)$.*

Through the rest of this paper, we assume that $G = (V, E)$ is a connected AT-free graph and $(V, \prec) = (y \prec \cdots \prec z)$ is a vertex ordering of $G$ produced by a 2-sweep LexBFS. For each vertex $u \in V$, the largest vertex in $N[u]$ is denoted by $ln(u)$. Note that, since $G[u]$ is connected, $ln(u) \succ u$ for all $u \neq z$, and $ln(z) = z$. We now show that $(V, \prec)$ is a strong 2-CCPO of $G$.

**Lemma 2.** *Let $a \prec b \prec c$ be any three vertices of $G$ that $b$ misses a path from $a$ to $c$. Then $ln(b) = ln(c)$.*

*Proof.* Since $b \prec c$, clearly $ln(b) \preceq ln(c)$. We now prove $ln(b) = ln(c)$ by contradiction. Suppose the contrary that $ln(b) \prec ln(c)$. Let $Q$ be an $a$-$c$ path missed by $b$ and $P$ be a $ln(c)$-$z$ path in $G[ln(c)]$. Since $ln(b) \prec ln(c)$, $b$ misses $P$. Thus, $b$ misses the $a$-$z$ path $Q \oplus (c, ln(c)) \oplus P$. However, it contradicts to Lemma 1 that $b$ intercepts all $a$-$z$ paths in $G$. □

**Lemma 3.** *The ordering $(V, \prec)$ is a strong 2-CCPO of $G$.*

*Proof.* Let $a \prec b \prec c$ be any three vertices of $G$ that $d(a, c) \leq 2$. If $b$ intercepts a shortest path from $a$ to $c$, it is easy to see that $(a, b) \in E$ or $d(b, c) \leq 2$. On the other hand, if $b$ misses any shortest path from $a$ to $c$, by Lemma 2, $ln(b) = ln(c)$. Thus $(b, ln(c), c)$ is a path in $G$ and $d(b, c) \leq 2$. □

Fig. 1 shows a strong 2-cocomparability graph with a strong 2-CCPO $a \prec b \prec c \prec d \prec e \prec f$ and it contains $\{a, b, f\}$ as an AT. Indeed, every vertex ordering produced by a 2-sweep LexBFS in the graph is a strong 2-CCPO. Thus we have the following result.

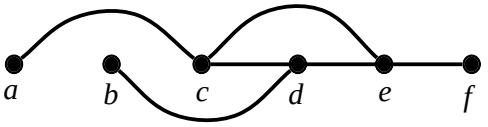



**Fig. 1.** A strong 2-cocomparability graph which is not AT-free.

**Theorem 1.** *AT-free graphs are properly contained in the class of strong 2-cocomparability graphs. Furthermore, a strong 2-CCPO of an AT-free graph can be produced by a 2-sweep LexBFS algorithm in $O(n + m)$ time.*

Since a strong 2-CCPO is a 2-CCPO and $G$ is a 2-cocomparability graph if and only if every power $G^k$ $(k \geq 2)$ is a cocomparability graph, Lemma 3 also implies that all proper powers of AT-free graphs are cocomparability graphs.

## 3 Recognition of Bipartite AT-Free Graphs

A graph is bipartite if its vertex set can be partitioned into $S$ and $T$ such that each edge has one end in $S$ and the other end in $T$. Let $G = (S, T, E)$ denote a bipartite graph. An ordering of $S$ has the *adjacency property* if for each vertex $t \in T$, $N(t)$ consists of consecutive vertices in the ordering of $S$. An ordering of $S$ has the *enclosure property* if for every pair of vertices $t, t' \in T$ such that $N(t) \subset N(t')$, $N(t') \backslash N(t)$ contains consecutive vertices in the ordering of $S$. A

*strong ordering* $(S \cup T, \prec)$ is a combination of an ordering of $S$ and an ordering of $T$ such that for any two edges $(s,t'),(s',t) \in E$, where $s,s' \in S$ with $s \prec s'$ and $t,t' \in T$ with $t \prec t'$, it implies that $(s,t) \in E$ and $(s',t') \in E$.

A graph is a *permutation graph* if there exist two permutations of its vertices such that two vertices, say $u$ and $v$, are adjacent if and only if $u$ precedes $v$ in one permutation and $v$ precedes $u$ in the other. Indeed, the orderings of vertices corresponding to these two permutations are both transitive ordering and cocomparability ordering. Thus, a graph $G$ is a permutation graph if and only if both $G$ and its complement are comparability graphs [18]. A *bipartite permutation graph* is both a bipartite graph and a permutation graph. This class of graphs was studied by Spinrad et al. [22] and can be characterized as follows.

**Theorem 2.** (Spinrad et al. [22]) *The following statements are equivalent for a bipartite graph $G = (S,T,E)$.*

(1) *$G$ is a bipartite permutation graph.*
(2) *There is a strong ordering of $S$ and $T$.*
(3) *There exists an ordering of $S$ (or $T$) which has the adjacency and enclosure properties.*

Note that, it can be seen from [22] that if an ordering of $S$ and $T$ is a strong ordering, then both the ordering of $S$ and the ordering of $T$ have the adjacency and enclosure properties, provided that all isolated vertices of $G$ appear at the beginning of the orderings of $S$ and $T$. However, if orderings of $S$ and $T$ satisfy the adjacency and enclosure properties, it does not imply that the combination of these two orderings forms a strong ordering of $G$. Based on the characterizations described in Theorem 2, Spinrad et al. developed an $O(n+m)$ time algorithm for recognizing bipartite permutation graphs.

A graph is *bipartite AT-free* if it is a bipartite graph and does not contain an AT. It is clear that a bipartite permutation graph is bipartite AT-free. In [9], Gallai characterized the comparability graphs and the cocomparability graphs (and thus the permutation graphs) in term of a complete list of forbidden induced subgraphs. Since every bipartite forbidden structure from Gallai's list always contains an AT, we conclude that a bipartite AT-free graph must be a bipartite permutation graph. In fact, the following observation is mentioned in a recent book of Brandstädt et al.

**Proposition 1.** (Brandstädt et al. [1]) *$G$ is a bipartite permutation graph if and only if $G$ is bipartite and $G$ contains no asteroidal triple.*

Due to the fact that bipartite permutation graphs are exactly bipartite AT-free graphs, we shall borrow from the notion of AT-free to provide a new recognition algorithm for this class of graphs. Since there is a simple linear time algorithm to determine whether a graph is bipartite, we can restrict our attention to bipartite graphs. Recall that the recognition algorithm provided in [22] maintains a data structure which is used to represent all possible orderings of $S$ satisfying the adjacency and enclosure properties. Indeed, their algorithm is similar to the algorithm of Booth and Lueker for testing consecutive arrangement

[2], where enclosure adds another type of constraint. If the input graph is a permutation graph, their algorithm generates an ordering which has the adjacency and enclosure properties. For otherwise, this algorithm may still produce a candidate ordering. Thus, they also provided a way to verify whether this ordering has adjacency and enclosure properties.

Our recognition algorithm has the same time complexity as the algorithm of [22]. These two algorithms are different in approach for generating the vertex ordering. We first use a modified 2-sweep LexBFS as a procedure to construct a vertex ordering. Note that, if a breadth-first search is applied to a bipartite graph $G = (S, T, E)$, then we can immediately determine the orderings of $S$ and $T$, respectively. We then show that $G$ is bipartite AT-free if and only if both the orderings of $S$ and $T$ generated from the above procedure have the adjacency and enclosure properties. So we can simply examine the adjacency and enclosure properties in this specific ordering. Before introducing the modified 2-sweep LexBFS algorithm, we establish some basic properties. Assume that $(S \cup T, \prec) = (y \prec \cdots \prec z)$ is an ordering of the vertices produced by a 2-sweep LexBFS on a bipartite AT-free graph $G$. Then we have the following lemmas.

**Lemma 4.** *For any $s \prec s' \prec t$, $s, s' \in S$ and $t \in T$, if $(s,t) \in E$ then $(s',t) \in E$.*

*Proof.* By Lemma 3, if $s \prec s' \prec t$ and $(s,t) \in E$, then $d(s,s') = 1$ or $d(s',t) \leq 2$. Since $(s,s') \notin E$ and the distance $d(s',t)$ is odd, the result follows. □

By the symmetry of $S$ and $T$, we also have the fact: for any three vertices $t \prec t' \prec s$, $s \in S$ and $t, t' \in T$, if $(s,t) \in E$ then $(s,t') \in E$.

Define $\Gamma(u,v) = \{w \in N(u) : v \prec w\}$ for $u, v \in S \cup T$, and let $\gamma(u,v) = |\Gamma(u,v)|$. In the following two lemmas, we assume $s, s' \in S$ and $t, t' \in T$ with $s \prec s'$ and $t \prec t'$.

**Lemma 5.** *If $(s,t'), (s',t) \in E$, then $(s',t') \in E$.*

*Proof.* Clearly, either $s \prec s' \prec t'$ or $t \prec t' \prec s'$ must be true. By Lemma 4, $(s,t') \in E$ implies $(s',t') \in E$ for the former case, and $(s',t) \in E$ implies $(s',t') \in E$ for the later case. □

**Lemma 6.** *Suppose $s \prec t$, $(s,t'), (s',t) \in E$ and $(s,t) \notin E$. Then, $\Gamma(t,s) \subseteq \Gamma(t',s)$ and $\Gamma(t,t') = \Gamma(t',t')$.*

*Proof.* By definition, every vertex in $\Gamma(t,s)$ is larger than $s$. Since $t \prec t'$ and $(s,t') \in E$, by Lemma 5 every vertex $w \in \Gamma(t,s)$ must be adjacent to $t'$. Thus, $\Gamma(t,s) \subseteq \Gamma(t',s)$. Note that it implies $\Gamma(t,u) \subseteq \Gamma(t',u)$ for all $s \preceq u$. In particular, $\Gamma(t,t') \subseteq \Gamma(t',t')$.

It remains to prove that $\Gamma(t,t') \supseteq \Gamma(t',t')$. Suppose the contrary that there exists a vertex $s''$ in $\Gamma(t',t')$ but not in $\Gamma(t,t')$. Consider the three vertices $s \prec t \prec s''$. Since $d(s,s'') \leq 2$ and $(s,t), (s'',t) \notin E$, $t$ misses every shortest path from $s$ to $s''$. By Lemma 2, $ln(t) = ln(s'')$. It is a contradiction because $s''$ and $t$ have no common neighbor. □

A symmetric statement is: if $t \prec s$, $(s,t'),(s',t) \in E$ and $(s,t) \notin E$, then $\Gamma(s,t) \subseteq \Gamma(s',t)$ and $\Gamma(s,s') = \Gamma(s',s')$.

To obtain a strong ordering of a bipartite AT-free graph, we modify the second sweep of a 2-sweep LexBFS as follows. Let $G = (V,E)$ be a graph. In the second sweep of LexBFS procedure, we first initialize a variable $f(v)$ for each $v \in V$ to be the degree $deg(v)$ of $v$. In each step of the LexBFS, we select a vertex to number. The selected vertex is a vertex with the lexicographically largest label. If there are more than one vertices with the same largest label, a vertex among those vertices with the smallest value $f$ is selected. After a vertex $v$ has been numbered, we update $f(w)$ to be $f(w)-1$ for each vertex $w \in N(v)$. In other word, when $u$ is about to be numbered, for all $v \in V$ with $v \prec u$, $f(v)$ denotes the degree of $v$ in the subgraph of $G$ induced by the set $\{w \in V : w \preceq u\}$. Notice that the modified 2-sweep LexBFS is a special 2-sweep LexBFS. Thus, Lemmas 4, 5, and 6 are still true for the ordering produced by the modified algorithm in a bipartite AT-free graph.

**Lemma 7.** *Any ordering of the vertices of a bipartite AT-free graph produced by a modified 2-sweep LexBFS algorithm is a strong ordering.*

*Proof.* Let $G = (S,T,E)$ be a bipartite AT-free graph and $(S \cup T, \prec)$ be the vertex ordering of $G$ that is produced by a modified 2-sweep LexBFS algorithm. Suppose the contrary that $(S \cup T, \prec)$ is not a strong ordering of $G$. That is, there exist vertices $s,s' \in S$ and $t,t' \in T$ with $s \prec s'$ and $t \prec t'$ such that $(s,t'),(s',t) \in E$ and $(s,t) \notin E$. Without loss of generality assume $s \prec t$. By Lemmas 6, we have $\Gamma(t,s) \subseteq \Gamma(t',s)$ and $\Gamma(t,t') = \Gamma(t',t')$. Thus, $\gamma(t,s) \leq \gamma(t',s)$ and $\gamma(t,t') = \gamma(t',t')$. Since $\Gamma(t,t') = \Gamma(t',t')$, when $t'$ is about to be numbered, the labels $L(t)$ and $L(t')$ are the same. At this time, $f(t') \leq f(t)$ by the fact that $t \prec t'$ and the selecting rule of the modified 2-sweep LexBFS. Thus,

$$deg(t') = f(t') + \gamma(t',t') \leq f(t) + \gamma(t,t') = deg(t).$$

On the other hand, we consider the values $f(t)$ and $f(t')$ when $s$ is about to be numbered. Since $deg(t) \geq deg(t')$ and $\gamma(t,s) \leq \gamma(t',s)$, it implies $f(t) \geq f(t')$ when $s$ is about to be numbered. Since $(s,t') \in E$ and $(s,t) \notin E$, there is some vertex $s^* \in S$ with $s^* \prec s$ such that it is adjacent to $t$ in $G$. Since $s^* \prec s \prec t$, by Lemma 4 $(s^*,t) \in E$ implies $(s,t) \in E$, which is a contradiction. □

**Theorem 3.** *A bipartite graph $G = (S,T,E)$ is AT-free if and only if the modified LexBFS algorithm generates an ordering of $G$ such that both the orderings of $S$ and $T$ have the adjacency and enclosure properties.*

*Proof.* The "only if" part directly follows from Lemma 7 and the fact, a strong ordering implies that both $S$ and $T$ have the adjacency and enclosure properties. Conversely, if $G$ is not a bipartite AT-free graph, by Theorem 2, none of the orderings of $S$ and $T$ satisfy the adjacency and enclosure properties. □

In order to use the modified 2-sweep LexBFS algorithm to recognize bipartite AT-free graphs, we need an efficient method to examine whether or not a specific vertex ordering produced by the algorithm has the adjacency and enclosure

properties. In fact, Spinrad et al. [22] have proposed an $O(n+m)$ time algorithm for this work. To show that the total complexity of our implementation is $O(n+m)$ time, we now consider the time it takes in the second sweep of LexBFS. Recall that the LexBFS algorithm presented in [20] does not actually calculate the labels, but rather keep the unnumbered vertices in lexicographic order. All unnumbered vertices are placed in a number of separate sets $U_l$, each of which contains those vertices of the same label $l$ in the current search. $U_l$ is represented by a doubly linked list. Keep the sets in a queue ordered lexicographically by label from smallest to largest. Initially, the queue contains only one set $U_\emptyset = V \backslash \{z\}$. When a vertex $u$ is selected to number, for each set $U$ containing a vertex $w \in N(u)$, create a new set $U'$ which is inserted in the very front of $U$ in the queue. Then move all such vertices $w$ from $U$ to the new set $U'$. This method maintains the lexicographic ordering without actually calculating the labels. For yielding the modified LexBFS, in the above implementation of LexBFS, we initially sort the adjacency list for each vertex of $G$ and the vertices of $U_\emptyset$ according to the degrees of vertices in increasing order. Then, in each stage all vertices of each new set $U'$ are kept in the sequence same as $U$ and the vertex with the smallest value $f$ is always located in the front of its list. Since computing the degrees of vertices and sorting the adjacency list of each vertex of $G$, and $U_\emptyset$ (using a radix sort) takes $O(n+m)$ time, the modified algorithm can be implemented in $O(n+m)$ time.

## 4 Concluding Remarks

In this paper, we investigate the metric property of LexBFS-ordering on AT-free graphs. We show that for an AT-free graph $G$, the ordering produced by a 2-sweeps LexBFS algorithm is a strong 2-CCPO, which generalizes the concept of cocomparability ordering. This result also implies that every proper power of an AT-free graph is a cocomparability graph. In particular, if $G$ is a bipartite permutation graph, a slight modification of the algorithm can construct a strong ordering for such a graph. This suggests a simple linear time recognition algorithm for bipartite permutation graphs. As a final comment, we note that the 2-sweep LexBFS is a valuable tool for solving algorithmic problems in AT-free graphs, such as finding dominating pairs [8], constructing tree spanners [16], and examining diameter [5]. In Section 2, we have given an example to show that a graph containing AT may produce a strong 2-CCPO by a 2-sweep LexBFS algorithm. Can one characterize the graphs for which every ordering of the vertices produced by a 2-sweep LexBFS is a strong 2-CCPO? Also, we ask whether or not there are efficient recognition algorithms for the class of strong $t$-cocomparability graphs where $t \geq 2$?

## References

1. A. Brandstädt, V. B. Le and J. P. Spinrad, Graph Classes: A Survey, (series SIAM Monographs on Discrete Mathematics and Applications, 1999).

2. K. S. Booth and G. S. Lueker, Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms, J. Comput. System Sci., **13** (1976) 335–379.
3. H. Broersma, T. Kloks, D. Kratsch and H. Muller, Independent sets in asteroidal triple-free graphs, SIAM J. Discrete Math., **12** (1999) 276–287.
4. J. M. Chang, C. W. Ho and M. T. Ko, Distances in asteroidal triple-free graphs and their powers, in: Proceedings of CTS Workshop on Combinatorics and Algorithms, Taipei, Taiwan, (1998) 1–14.
5. D. G. Corneil, F. F. Dragan, M. Habib and C. Paul, Diameter determination on restricted graph families, in: Proceedings of WG'98, LNCS Vol. 1517, Springer-Verlag, (1998) 192–202.
6. D. G. Corneil, S. Olariu and L. Stewart, A linear time algorithm to compute a dominating path in an AT-free graph, Inform. Process. Lett., **54** (1995) 253–257.
7. D. G. Corneil, S. Olariu and L. Stewart, Asteroidal triple-free graphs, SIAM J. Discrete Math., **10** (1997), 399–430.
8. D. G. Corneil, S. Olariu and L. Stewart, Linear time algorithms for dominating pairs in asteroidal triple-free graphs, in: Proceedings of ICALP'95, LNCS Vol. 944, Springer-Verlag, (1995) 292–302.
9. T. Gallai, Transitiv orientierbare Graphen, Acta Math. Acad. Sci. Hung., **18** (1967) 25–66
10. M. C. Golumbic, C. L. Monma and W. T. Trotter, Jr. Tolerance graphs, Discrete Appl. Math., **9** (1984), 157–170.
11. C. T. Hoàng and B. A. Reed, $P_4$-comparability graphs, Discrete Math., **74** (1989) 173–200.
12. B. Jamison and S. Olariu, On the semi-perfect elimination, Adv. Appl. Math., **9** (1988) 364–376.
13. T. Kloks, D. Kratsch and J. Spinrad, On treewidth and minimum fill-in of asteroidal triple-free graphs, Theor. Comput. Sci., **175** (1997) 309–335.
14. T. Kloks. H. Müller and C. K. Wong, Vertex ranking of asteroidal triple-free graphs, Inform. Process. Lett., **68** (1998) 201–206.
15. D. Kratsch, Domination and total domination on asteroidal triple-free graphs, Forschungsergebnisse Math/Inf/96/25, FSU Jena, Germany, 1996.
16. D. Kratsch, H. O. Le, H. Müller and E. Prisner, Additive tree spanners, manuscript, 1998.
17. C. G. Lekkerkerker and J. C. Boland, Representation of a finite graph by a set of intervals on the real line, Fund. Math., **51** (1962) 45–64.
18. A. Pnueli, A. Lempel and S. Even, Transitive orientation of graphs and identification of permutation graphs, Canad. J. Math., **23** (1971) 160–175.
19. D, J. Rose, Triangulated graphs and the elimination process, J. Math. Anal. Appl., **32** (1970) 597–609.
20. D. J. Rose, R. E. Tarjan and G. S. Lueker, Algorithmic aspects of vertex elimination on graphs, SIAM J. Comput., **5** (1976) 266–283.
21. J. Spinrad, Transitive orientation in $O(n^2)$ time, in: Proceedings of 15th Ann. ACM Symp. on Theory of Computing, (1983) 457–466.
22. J. Spinrad, A. Brandstädt and L. Stewart, Bipartite permutation graphs, Discrete Appl. Math., **18** (1987) 279–292.
23. R. E. Tarjan and M. Yanakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, SIAM J. Comput., **13** (1984) 566–579.

# Parallel Algorithms for Shortest Paths and Related Problems on Trapezoid Graphs*

F. R. Hsu**, Yaw-Ling Lin, and Yin-Te Tsai***

Providence University, Shalu, Taichung Hsien,
Taiwan 433, Republic of China
e-mails: {frhsu,yllin,yttsai}@pu.edu.tw.

**Abstract.** In this paper, we consider parallel algorithms for shortest paths and related problems on trapezoid graphs under the CREW PRAM model. Given a trapezoid graph with its corresponding trapezoid diagram, we present parallel algorithms solving the following problems: For the *single-source shortest path problem*, the algorithm runs in $O(\log n)$ time using $O(n)$ processors and space. For the *all-pair shortest path query problem*, after spending $O(\log n)$ preprocessing time using $O(n \log n)$ space and $O(n)$ processors, the algorithm can answer the query in $O(\log \delta)$ time using one processor. Here $\delta$ denotes the distance between two queried vertices. For the *minimum cardinality Steiner set problem*, the algorithm runs in $O(\log n)$ time using $O(n)$ processors and space.
We also extend our results to the generalized trapezoid graphs. The single-source shortest path problem and the minimum cardinality Steiner set problem on *d-trapezoid graphs* and *circular d-trapezoid graphs* can both be solved in $O(\log n \log d)$ time using $O(nd)$ space and $O(d^2 n/\log d)$ processors. The all-pair shortest path query problem on $d$-trapezoid graphs and circular $d$-trapezoid graphs can be answered in $O(d \log \delta)$ time using one processor after spending $O(\log n \log d)$ preprocessing time using $O(nd \log n)$ space and $O(d^2 n/\log d)$ processors.

## 1 Introduction

The intersection graph of a collection of trapezoids with corner points lying on two parallel lines is called the *trapezoid graph* [2]. Note that trapezoid graphs are perfect and properly contain both interval graphs and permutation graphs. Trapezoid graphs are perfect since they are cocomparability graphs.

The *single-source shortest path (SSSP) problem* is the problem of finding the shortest paths between a given vertex and all other vertices. The *all-pair shortest path problem* is the problem of finding the shortest paths between all pairs of vertices. In stead of finding all pairs of shortest paths, the *all-pair shortest path*

* Support in part by the National Science Council, Taiwan, R.O.C, grant NSC-89-2213-E-126-007.

** Corresponding author, Department of Accounting.

*** The corresponding address for the second and the third authors is Department of Computer Science and Information Management.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 173–182, 1999.
© Springer-Verlag Berlin Heidelberg 1999

*query (APSPQ) problem* is described as follows: First, apply a faster preprocessing algorithm and construct a data structure. Using the data structure, a query on the length of a shortest path between any two vertices can be answered very fast.

In [6], Ibarra *et al.* proposed an $O(\log n)$ time parallel algorithm using $O(n/\log n)$ processors on the EREW PRAM to solve the SSSP problem for permutation graphs. Recently, Chao *et al.* [5], using $O(n/\log n)$ EREW PRAM processors, presented an $O(\log n)$ preprocessing parallel algorithm to build $O(n)$ space data structure for the APSPQ problem on permutation graphs such that each query can be answered in constant time. In [3], using $O(n/\log n)$ CREW PRAM processors, Chen *et al.* proposed an $O(\log n)$ preprocessing parallel algorithm to build $O(n)$ space data structure for the APSPQ problem on interval and circular-arc graphs such that each query can be answered in constant time. For trapezoid graphs, Liang [8] designed a linear time sequential algorithm for the SSSP problem. According to our knowledge, there was no literature proposing any parallel algorithm for the SSSP problem on trapezoid graphs. Although there are efficient parallel algorithms for the APSPQ problem on permutation and interval graphs, there was no efficient sequential algorithm for this problem on trapezoid graphs.

Given a graph $G = (V, E)$ and a set $U \subset V$, a Steiner set for $U$ in $G$ is a set $S \subseteq V \setminus U$ of vertices, such that $S \cup U$ induced a connected subgraph. The *minimum cardinality Steiner set (MCSS) problem* is the problem of finding a Steiner set of the smallest cardinality. Liang [8] proposed a linear time sequential algorithm for the MCSS problem on trapezoid graphs.

In this paper, we consider parallel algorithms for shortest paths and related problems on trapezoid graphs under the CREW PRAM model. Given a trapezoid graph with $n$ vertices, we present parallel algorithms solving the following problems: For the SSSP problem, the algorithm runs in $O(\log n)$ time using $O(n)$ processors and space. For the APSPQ problem, after spending $O(\log n)$ preprocessing time using $O(n \log n)$ space and $O(n)$ processors, the algorithm can answer the query in $O(\log \delta)$ time using one processor. Here $\delta$ denotes the distance between two queried vertices. For the MCSS problem, the algorithm runs in $O(\log n)$ time using $O(n)$ processors and space.

We also extend our results to the generalized trapezoid graphs. The SSSP problem and the MCSS problem on $d$-trapezoid graphs and circular $d$-trapezoid graphs can both be solved in $O(\log n \log d)$ time using $O(nd)$ space and $O(d^2 n/\log d)$ processors. The APSPQ problem on $d$-trapezoid graphs and circular $d$-trapezoid graphs can be answered in $O(d \log \delta)$ time using one processor after spending $O(\log n \log d)$ preprocessing time using $O(nd \log n)$ space and $O(d^2 n/\log d)$ processors.

All of our algorithms use only simple techniques such as the parallel prefix and suffix computations [1]. Therefore, they are easy to implement. The rest of this paper is organized as follows. Section 2 establishes basic notations and some interesting properties of trapezoid graphs. Section 3 gives a parallel algorithm for the SSSP problem on trapezoid graphs. Section 4 shows how to preprocess a

given trapezoid graph in parallel such that any future shortest path query can be answered efficiently. Section 5 presents a parallel algorithm for the MCSS problem on trapezoid graphs. In Section 6, we discuss about generalized trapezoid graphs. Finally, we conclude our results in Section 7. Due to page limitation, our parallel algorithms for the generalized trapezoid graphs are omitted in the extended abstract.

## 2 Basic Notations and Properties

There are two parallel lines, *top channel* and *bottom channel* respectively. Each channel is labeled with consecutive integer values $1, 2, 3, \ldots$, from left to right. A trapezoid $t_i$ is defined by four corner points $[a_i, b_i, c_i, d_i]$ such that $a_i$ and $b_i$ are on the top channel and $c_i$ and $d_i$ are on the bottom channel respectively where $a_i < b_i$ and $c_i < d_i$. The above geometric representation is called the *trapezoid diagram* $T$. Without loss of generality, we assume that all corner points $a_i, b_i, i = 1, \ldots, n$, on the top channel (similarly, $c_i, d_i$ on the bottom channel) are distinct with coordinates of consecutive integer values $1, 2, \ldots, 2n$. Trapezoids are labeled in increasing order of their corner points $b_i$'s. A graph $G = (V, E)$ is a *trapezoid graph* if it can be represented by a trapezoid diagram $T$ such that each trapezoid corresponds to a vertex in $V$ and $(i, j) \in E$ if and only if $t_i$ and $t_j$ intersects in the trapezoid diagram. Figure 1 shows a trapezoid graph with its corresponding trapezoid diagram. In this paper, we assume that

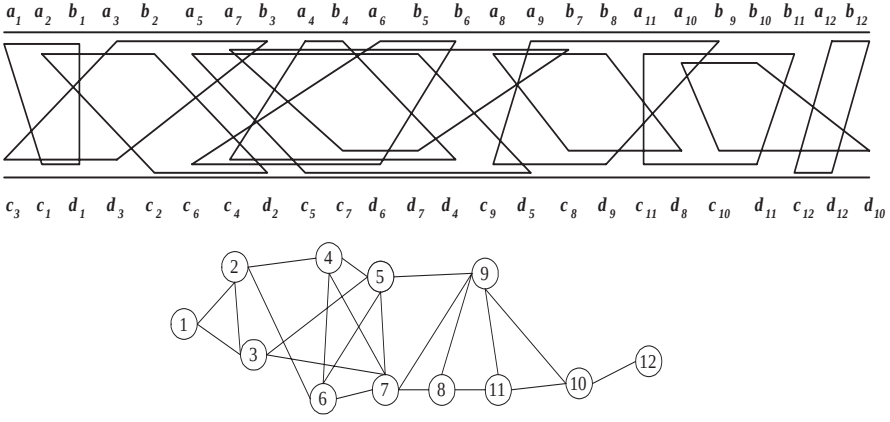



**Fig. 1.** A trapezoid graph and its trapezoid diagram.

the input trapezoid graph is connected. It is easy to see that our algorithms can be modified to handle the cases when the input graph is not connected. We also assume that its trapezoid diagram is given.

Suppose $p$ is a corner point of trapezoid $t_i$. For ease of reference, let $A(p)$, $B(p)$, $C(p)$ and $D(p)$ denote $a_i$, $b_i$, $c_i$ and $d_i$ respectively. If there is no confusion,

for ease of reference, for trapezoid $u$, we also use $A(u)$, $B(u)$, $C(u)$ and $D(u)$ to denote its four corner points. For trapezoids $u$ and $v$, let $\mathrm{dis}(u, v)$ denote the distance between $u$ and $v$ in the corresponding trapezoid graph. Consider trapezoid $t_i$. Among trapezoids which intersect $t_i$, consider their corners. We introduce notations to denote trapezoids with the rightmost and the leftmost corner on the top channel and the bottom channel respectively. Formally, for every trapezoid $t_i$, we define the *left farthest-reaching trapezoid-pair*, denoted by $L(t_i) = (L_T(t_i), L_B(t_i))$, and the *right farthest-reaching trapezoid-pair*, denoted by $R(t_i) = (R_T(t_i), R_B(t_i))$, as follows:

$$\begin{aligned}
L_T(t_i) = t_j \text{ iff } a_j &= \min\{a_l | \mathrm{dis}(t_i, t_l) \leq 1\} \\
L_B(t_i) = t_j \text{ iff } c_j &= \min\{c_l | \mathrm{dis}(t_i, t_l) \leq 1\} \\
R_T(t_i) = t_j \text{ iff } b_j &= \max\{b_l | \mathrm{dis}(t_i, t_l) \leq 1\} \\
R_B(t_i) = t_j \text{ iff } d_j &= \max\{d_l | \mathrm{dis}(t_i, t_l) \leq 1\}.
\end{aligned}$$

For example, in Figure 1, $L_T(t_4) = t_2$, $L_B(t_4) = t_2$, $R_T(t_4) = t_7$, $R_B(t_4) = t_5$.

Let $m_k$ and $w_k$ denote the $k$-th point on the top channel and bottom channel of a trapezoid diagram respectively. According to the following lemma, these farthest-reaching trapezoid-pairs can be computed efficiently.

**Lemma 1.** *In a trapezoid diagram with $n$ vertices, the following statements hold for all $i$, $1 \leq i \leq n$:*

$$\begin{aligned}
L_T(t_i) = t_j \text{ iff } a_j = \min(&\{A(m_k) | k = a_i, a_i + 1, \ldots, 2n\} \\
&\cup \{A(w_k) | k = c_i, c_i + 1, \ldots, 2n\}) \\
L_B(t_i) = t_j \text{ iff } c_j = \min(&\{C(m_k) | k = a_i, a_i + 1, \ldots, 2n\} \\
&\cup \{C(w_k) | k = c_i, c_i + 1, \ldots, 2n\}) \\
R_T(t_i) = t_j \text{ iff } b_j = \max(&\{B(m_k) | k = 1, 2, \ldots, b_i\} \cup \{B(w_k) | k = 1, 2, \ldots, d_i\}) \\
R_B(t_i) = t_j \text{ iff } d_j = \max(&\{D(m_k) | k = 1, 2, \ldots, b_i\} \cup \{D(w_k) | k = 1, 2, \ldots, d_i\})
\end{aligned}$$

□

Now, we generalize the concept of $L(t_i)$ and $R(t_i)$. Consider trapezoids which $t_i$ can reach within $k$ steps. Let $R_T^k(t_i)$ and $R_B^k(t_i)$ denote the trapezoid with the rightmost corner on the top channel and the bottom channel respectively. Formally, we define $L^k(t_i) = (L_T^k(t_i), L_B^k(t_i))$ and $R^k(t_i) = (R_T^k(t_i), R_B^k(t_i))$ as follows:

$$\begin{aligned}
L_T^k(t_i) = t_j \text{ iff } a_j &= \min\{a_l | \mathrm{dis}(t_i, t_l) \leq k\} \\
L_B^k(t_i) = t_j \text{ iff } c_j &= \min\{c_l | \mathrm{dis}(t_i, t_l) \leq k\} \\
R_T^k(t_i) = t_j \text{ iff } b_j &= \max\{b_l | \mathrm{dis}(t_i, t_l) \leq k\} \\
R_B^k(t_i) = t_j \text{ iff } d_j &= \max\{d_l | \mathrm{dis}(t_i, t_l) \leq k\}.
\end{aligned}$$

For example, consider Figure 1. The set of trapezoids which $t_1$ can reach within two steps is $\{t_i | i = 1, 2, \ldots, 7\}$. Therefore, $R^2(t_1) = (t_7, t_5)$. By definition, $R^0(u) = (u, u)$, $L^1(u) = L(u)$ and $R^1(u) = R(u)$.

By definition, $B(R_T^k(u)) \geq B(R_B^k(u))$ and $D(R_B^k(u)) \geq D(R_T^k(u))$, it follows $R_T^k(u)$ intersects $R_B^k(u)$. We have the following lemma.

**Lemma 2.** *For any trapezoid $u$, $R_T^k(u)$ intersects $R_B^k(u)$ and $L_T^k(u)$ intersects $L_B^k(u)$.* □

Now, we describe how to compute $\mathrm{dis}(u, v)$, for trapezoid $u$ and $v$. Without loss of generality, assume that $D(u) < D(v)$. If $u$ intersects $v$, obviously, $\mathrm{dis}(u, v) = 1$. Otherwise, if either $R_T^1(u)$ or $R_B^1(u)$ intersects $v$, $\mathrm{dis}(u, v) = 2$. If neither $R_T^1(u)$ nor $R_B^1(u)$ intersects $v$, we try $R_T^2(u)$ and $R_B^2(u)$. By this way, we will find some $k$ such that neither $R_T^{k-2}(u)$ nor $R_B^{k-2}(u)$ intersects $v$, yet $R_T^{k-1}(u)$ or $R_B^{k-1}(u)$ intersects $v$. It follows $\mathrm{dis}(u, v) = k$.

Immediately, we have the following lemma.

**Lemma 3.** *For any two trapezoids $u$ and $v$, suppose $D(u) < D(v)$ and $\mathrm{dis}(u, v) > 1$. Then $\mathrm{dis}(u, v) = k$, iff*

1. *neither $R_T^{k-2}(u)$ nor $R_B^{k-2}(u)$ intersects $v$ and,*
2. *either $R_T^{k-1}(u)$ or $R_B^{k-1}(u)$ intersects $v$.* □

Suppose that we already know $R^k(u) = (t_{j_1}, t_{j_2})$. How can we find $R^{k+1}(u)$? We can find it from $R(t_{j_1})$ and $R(t_{j_2})$. It follows the following lemma.

**Lemma 4.** *For any trapezoid $u$,*

$$B(R_T^{k+1}(u)) = \max\{B(R_T(R_T^k(u))), B(R_T(R_B^k(u)))\}$$
$$D(R_B^{k+1}(u)) = \max\{D(R_B(R_T^k(u))), D(R_B(R_B^k(u)))\} \quad \square$$

By induction and Lemma 4, we have the following theorem.

**Theorem 1.** *For any trapezoid $u$,*

$$B(R_T^{k+j}(u)) = \max\{B(R_T^j(R_T^k(u))), B(R_T^j(R_B^k(u)))\}$$
$$D(R_B^{k+j}(u)) = \max\{D(R_B^j(R_T^k(u))), D(R_B^j(R_B^k(u)))\}. \quad \square$$

In other words, $R^{k+j}(u)$ can be computed from $R^j(R_T^k(u))$ and $R^j(R_B^k(u))$ within constant time. Instead of walking to right only one step once, by this theorem, we can walk $j$ steps once. We call such action one *long-jump*. It will be used in the later section.

The following lemma is also helpful for constructing a shortest path between two trapezoids.

**Lemma 5.** *For any two trapezoids $u$ and $v$, if $D(u) < D(v)$ and $\mathrm{dis}(u, v) = k > 1$, then there exists a path $(p_0, p_1, \ldots, p_k)$ such that $p_0 = u$, $p_k = v$, $p_i \neq p_{i-1}$, $p_i = R_T(p_{i-1})$ or $R_B(p_{i-1})$ for $i = 1, 2, \ldots, k-1$.* □

When we compute $\mathrm{dis}(u, v)$, $D(u) < D(v)$, we start walking from $u$ to $v$. Suppose after $l$ steps, we still can not reach $v$. It means neither $R_T^q(u)$ nor $R_B^q(u)$ intersects $v$, for $q = 1, \ldots, l$. Now, we consider $\mathrm{dis}(R_T^l(u), v)$ and $\mathrm{dis}(R_B^l(u), v)$. We have the following lemma to show that $\mathrm{dis}(u, v) = l + \min\{\mathrm{dis}(R_T^l(u), v), \mathrm{dis}(R_B^l(u), v)\}$.

**Lemma 6.** *For any two trapezoids $u$ and $v$, $D(u) < D(v)$, if $\mathrm{dis}(u, v) > l + 1$, then $\mathrm{dis}(u, v) = l + \min\{\mathrm{dis}(R_T^l(u), v), \mathrm{dis}(R_B^l(u), v)\}$.* □

## 3 Single-Source Shortest Path (SSSP) Problem

In this section, given a trapezoid diagram $T$ and a starting vertex $s$, we will show how to solve the SSSP problem on trapezoid graphs in $O(\log n)$ time using $O(n)$ processors on a CREW PRAM.

Union of all these shortest paths forms a shortest path tree rooted at $s$. That is a breadth first search (BFS) tree starting from $s$. In this paper, our algorithm constructs a BFS tree to represent the solution of the SSSP problem. For ease of discussion, we assume that $s = t_n$. It is easy to modify our algorithms to deal with the case that $s \neq t_n$.

Let $G$ denote the corresponding trapezoid graph of $T$. First, we construct a digraph $G'$ from $G$ such that $G'$ is a supergraph of a $BFS$ tree of $G$ rooted at $s$. We construct $G' = (V', E')$ as follows: let $V' = V$ and for $i = 1, \ldots, n-1$, add edges from $t_i$ to $R_T(t_i)$ and $R_B(t_i)$.

By definition of $G'$ and Lemma 5, immediately we have the following lemma.

**Lemma 7.** *$G' = (V', E')$ has the following properties.*

1. *$G'$ is an directed acyclic graph with out-degree at most 2.*
2. *For any vertex $u$ in $G'$, the shortest path from $u$ to $s$ on $G'$ is also a shortest path from $u$ to $s$ on $G$.* □

By the above lemma, we have the following theorem immediately.

**Theorem 2.** *$G'$ is a supergraph of a $BFS$ tree of $G$ rooted at $s$.* □

Now, we describe how to find a BFS tree. Since $s$ has the rightmost corner point on the top channel, it is not difficult to see that: for any $u$, $u \neq s$, $\mathrm{dis}(u, s) = 1$ if and only if $R_T(u) = s$. We now consider the cases that $\mathrm{dis}(u, s) > 1$. At $j$-th iteration, our algorithm performs one long-jump and walk $2^i$ steps. Our algorithm uses the pointer jumping technique and spends $O(\log n)$ parallel steps to calculate the distance of each trapezoid to $s$. We assign each trapezoid a dedicated processor. Suppose that after $(j-1)$-th iteration, $\mathrm{dis}(u, s)$ is not decided yet. At the $j$-th iteration, the corresponding processor will check if $\mathrm{dis}(R_T^{2^{j-1}}(u), s)$ or $\mathrm{dis}(R_B^{2^{j-1}}(u), s)$ is found. If one of them is found, $\mathrm{dis}(u, s)$ will be decided by the equation in Theorem 1. Otherwise, $R^{2^j}(u)$ will be computed. That is, in each parallel step, each processor either has decided the distance of the trapezoid in question or the processor *doubles* the distance of the farthest-reaching tuple for the trapezoid whose distance then will be decided for later step.

In Figure 2, we show our parallel algorithm for the SSSP problem.

In the above algorithm, at the beginning, for every trapezoid $u$, $u \neq s$, we compute $R(u)$. At the $i$-th iteration, either we decide $\mathrm{dis}(u, s)$ or we compute $R^{2^i}(u)$. Note that at this moment $R^{2^{i-1}}(R_T^{2^{i-1}}(u))$ and $R^{2^{i-1}}(R_B^{2^{i-1}}(u))$ are already computed at the previous iteration. In other words, if we can not decide $\mathrm{dis}(u, s)$, we *double* the distance to probe $s$. By this way, to compute $\mathrm{dis}(u, s)$, it needs to compute $R^{2^i}(u)$, for $i = 1, \ldots, \lfloor \log(\mathrm{dis}(u, s)) \rfloor$. We call these trapezoid-pairs *powered-distance trapezoid-pairs* of $u$.

ALGORITHM **SSSP**
*Input:* A trapezoid diagram with $n$ trapezoids.
*Output:* The shortest path tree rooted at $s = t_n$.
*Step 0.* F[$i$] $= \infty$, for $i = 1, \ldots, n-1$.
*** Finally, F[i] will store dis($t_i, s$).
*Step 1.* Compute $R(t_i)$, for $i = 1, \ldots, n$.
For each trapezoid $t_i$, $t_i \neq s$, if $R_T(t_i) = s$, then F[$i$] $= 1$.
*Step 2.* For $i = 1$ to $\lceil \log n \rceil$, for each trapezoid $t_j \neq s$ and F[$j$] $= \infty$
if F[$R_T^{2^{i-1}}(t_j)$] $\neq \infty$ or F[$R_B^{2^{i-1}}(t_j)$] $\neq \infty$
F[$j$] $= 2^{i-1} + \min\{\text{F}[R_T^{2^{i-1}}(t_j)], \text{F}[R_B^{2^{i-1}}(t_j)]\}$
else
compute $R_T^{2^i}(t_j)$ and $R_B^{2^i}(t_j)$ from
$R^{2^{i-1}}(R_T^{2^{i-1}}(t_j))$ and $R^{2^{i-1}}(R_B^{2i-1}(t_j))$
end_if
*Step 3.* For each trapezoid $t_i$, compare $F[R_T(t_i)]$ and $F[R_B(t_i)]$.
Choose $R_T(t_i)$ or $R_B(t_i)$ as its parent
in the BFS tree such that its distance to $s$ is shorter.
END OF **SSSP**

**Fig. 2.** A parallel algorithm for the single-source shortest path problem on trapezoid graphs.

After Step 2 is performed, all distances to $s$ have been computed. By Theorem 2, for every $u$, among $R_T(u)$ and $R_B(u)$, we can choose the one which is nearer to $s$ as the parent of $u$ in the $BFS$ tree. Therefore, Algorithm SSSP computes the single-source shortest path tree correctly.

Consider the complexity of the above algorithm now.

*Step 1.* By Lemma 1, all right farthest-reaching trapezoid-pairs can be computed by utilizing the parallel prefix or suffix computations [1] in $O(\log n)$ time using $O(n/\log n)$ processors.

*Step 2.* It is not difficult to see that this step takes $O(\log n)$ time using $O(n)$ processors. Note that at the $i$-th iteration, for trapezoid $u$, $R^{2^{i-2}}(u)$ will not be used anymore. It follows we can use $O(n)$ space to store information we need.

*Step 3.* It is easy to perform this step in constant time using $O(n)$ processors.

Therefore, we have the following theorem.

**Theorem 3.** *The single-source shortest path problem on trapezoid graphs can be solved in $O(\log n)$ time using $O(n)$ space and processors under the CREW PRAM model.* □

## 4 All-Pair Shortest Path Query (APSPQ) Problem

In this section, we will present our parallel algorithm for the APSPQ problem on trapezoid graphs.

Consider the preprocessing stage now. We modify Algorithm SSSP to get the preprocessing algorithm. First, we construct the BFS tree rooted at $t_n$. Instead of only keeping constant number of powered-distance trapezoid-pairs for each trapezoid, we keep all powered-distance trapezoid-pairs. It takes $O(\log n)$ space to store these information for one trapezoid. So, it costs $O(n \log n)$ space for all trapezoids.

For any trapezoid $u$ in $T$, let $\text{lev}(u)$ denote the level of $u$ in the shortest path tree rooted at $t_n$. Note that $\text{lev}(u)$ is equal to the distance between $u$ and $t_n$. For the query stage, suppose the queried trapezoids are $u$ and $v$. Without loss of generality, we assume that $\text{lev}(u) \geq \text{lev}(v)$.

The following lemma and theorem show important properties for us to compute $\text{dis}(u, v)$.

**Lemma 8.** *For trapezoid $u$ and $v$, $u \neq v$, if $\text{lev}(u) = \text{lev}(v)$, then $\text{dis}(u, v) \leq 2$.* □

**Theorem 4.** *For trapezoid $u$ and $v$, if $\text{lev}(u) - \text{lev}(v) = k \geq 0$, then $k \leq \text{dis}(u, v) \leq k + 2$.* □

By Theorem 4, there are only three candidates of $\text{dis}(u, v)$: $k$, $k+1$ and $k+2$. At the query phase, we start from $k$. Consider $R^{k-1}(u)$. If one of its trapezoid intersects $u$, by Lemma 3, $\text{dis}(v, u) = k$. Otherwise, we try $k+1$ and then $k+2$.

We compute $R^{k-1}(u)$ using powered-distance trapezoid-pairs. Consider the binary representation of $k-1$. It needs only $\lceil \log_2(k-1) \rceil$ bits to represent $k-1$. Therefore, following the powered-distance trapezoid-pairs, we walk from $u$ toward $v$ and within $\lceil \log_2(k-1) \rceil$ times *long-jumps*, we can get $R^{k-1}(u)$. For example, suppose $k - 1 = 656$. Since $656 = 2^9 + 2^7 + 2^4$, we can compute $R^{656}(u)$ from powered-distance trapezoid-pairs $R^{2^9}(u)$, $R^{2^7}(u)$ and $R^{2^4}(u)$. By Theorem 1, it takes only constant time to perform one long-jump. Since all powered-distance trapezoid-pairs already computed and stored during the preprocessing stage, we have the following theorem. Let $\delta$ denote the distance between two queried vertices in the rest of this paper.

**Theorem 5.** *The preprocessing stage for the all-pair shortest path query problem on trapezoid graphs can be solved in $O(\log n)$ time and $O(n \log n)$ space using $O(n)$ processors under the CREW PRAM model. Further, a query can be answered in $O(\log \delta)$ time using one processor.* □

## 5 Minimum Cardinality Steiner Set (MCSS) Problem

The MCSS problem aims to find the minimum number of vertices to connect a set of target vertices. Liang [8] presented a linear time sequential algorithm for the MCSS problem on trapezoid graphs. His algorithm consists of the following three phase. Readers who are interested in the details should refer to the original paper.

*Phase 1.* Find the connected components among target trapezoids. Treat each component as a new target trapezoid. Let $T' = \{t_1', t_2', \ldots, t_i'\}$ be the set of new target vertices with $B(t_1') < B(t_2') \ldots < B(t_k')$.

*Phase 2.* For each non-target trapezoid, find the largest indexed new trapezoid intersects it. Merge this new target vertex into it.

*Phase 3.* The new trapezoid graph $\hat{G}$ consists of new target trapezoids and new non-target trapezoids. Find a shortest path from $t_1'$ to $t_i'$ on $\hat{G}$.

Now, we describe how to perform the three phases in parallel.

*Phase 1.* Consider the graph induced by target trapezoids. We observe that any two target trapezoids $u$ and $v$ are in the same connected component if and only if $R^n(u) = R^n(v)$. We can easily modify Algorithm SSSP to compute these trapezoid-pairs in $O(\log n)$ time using $O(n)$ processors.

*Phase 2.* Recall Lemma 1. Similar to computation of farthest-reaching trapezoid -pair, for each non-target trapezoid, we can find the largest indexed new trapezoid intersecting it using only parallel prefix and suffix computations [1]. This phase can be done in $O(\log n)$ using $O(n/\log n)$ processors.

*Phase 3.* Using Algorithm SSSP, we find the shortest path tree rooted at $t_i'$ on $\hat{G}$. It is easy to report the shortest path from $t_1'$ to $t_i'$ on this tree. It follows this phase can be done in $O(\log n)$ using $(n)$ processors.

Therefore, we have the following theorem.

**Theorem 6.** *The minimum cardinality Steiner set problem on trapezoid graphs can be solved in $O(\log n)$ time using $O(n)$ space and processors under the CREW PRAM model.* □

## 6 Shortest Paths on Generalized Trapezoid Graphs

Along with the direction that generalizes interval graphs and permutation graphs to trapezoid graphs, researchers are now trying to generalize the class of trapezoid graphs.

Flotow [4] introduces the class of $d$-trapezoid graphs that are the intersection graphs of $d$-trapezoids, where a $d$-trapezoid is defined by $d$ intervals on $d$ parallel lines. Note that the 1-trapezoid graph is exactly the class of interval graphs and the 2-trapezoid graph is exactly the class of trapezoid graphs. Kratsch [7] defines *circular trapezoid graphs* as the intersection graphs of circular trapezoid. Here a *circular trapezoid* is a generalized trapezoid between *two* concentric circles; two parallel lines of the circular trapezoid are circular arcs of each of the two circles, and the two other lines of the circular trapezoid are spiral segments. They also extends circular trapezoid graphs into $d > 2$ concentric circles; the generalized class of graphs is so called *circular $d$-trapezoid graphs*. In our full paper, we also show that the previous discussions about shortest paths and related problems can be extended to some generalized versions of trapezoid graphs including $d$-trapezoid graphs and circular $d$-trapezoid graphs. Due to page limitation, these algorithms are omitted in the extended abstract.

## 7 Conclusion

In this paper, we consider parallel algorithms for shortest paths and related problems on trapezoid graphs under the CREW PRAM model. Given a trapezoid graph with $n$ vertices, we present parallel algorithms solving the following problems: For the single-source shortest path problem, the algorithm runs in $O(\log n)$ time using $O(n)$ processors and space. For the all-pair shortest path query problem, after spending $O(\log n)$ preprocessing time using $O(n \log n)$ space and $O(n)$ processors, the algorithm can answer the query in $O(\log \delta)$ time using one processor. Here $\delta$ denotes the distance between two queried vertices. For the minimum cardinality Steiner set problem, the algorithm runs in $O(\log n)$ time using $O(n)$ processors and space.

We also extend our results to $d$-trapezoid graphs and circular $d$-trapezoid graphs. The class of (circular) trapezoid graphs is exactly the class of (circular) 2-trapezoid graphs. We have the following results. The single-source shortest path problem and the minimum cardinality Steiner set problem on $d$-trapezoid graphs and circular $d$-trapezoid graphs can both be solved in $O(\log n \log d)$ time using $O(nd)$ space and $O(d^2 n/\log d)$ processors. The all-pair shortest path query problem on $d$-trapezoid graphs and circular $d$-trapezoid graphs can be answered in $O(d \log \delta)$ time using one processor after spending $O(\log n \log d)$ preprocessing time using $O(nd \log n)$ space and $O(d^2 n/\log d)$ processors. It would be interesting to design parallel algorithms using fewer processors for these problems.

## References

1. S.G. Akl, "Parallel computation: models and methods", Prentice Hall, Upper Saddle River, New Jersey, 1997.
2. I. Dagan, M.C. Golumbic and R.Y. Pinter, "Trapezoid graphs and their coloring", *Discr. Applied Math.*,21:35-46,1988.
3. D.Z. Chen, D. T. Lee, R. Sridhar and C. N. Sekharan, "Solving the all-pair shortest path query problem on interval and circular-arc graphs", *Networks*, pp. 249-257, 1998.
4. Flotow, "On Powers of $m$-Trapezoid Graphs", *Discr. Applied Math.*, 63:187–192, 1995.
5. H.S. Chao, F.R. Hsu and R.C.T. Lee, "On the Shortest Length Queries for Permutation Graphs", *Proc. Of the 1998 International Computer Symposium, Workshop on Algorithms*, NCKU, Taiwan, pp. 132-138, 1998.
6. O.H. Ibarra and Q. Zheng, "An optimal shortest path parallel algorithm for permutation graphs", *J. of Parallel and Distributed Computing*, 24:94-99, 1995.
7. Dieter Kratsch, Ton Kloks and Haiko Müller", "Measuring the vulnerability for classes of intersection graphs", *Discr. Applied Math.*, 77:259-270, 1997.
8. Y. D. Liang, "Steiner set and connected domination in trapezoid graphs", *Information Processing Letters*, 56(2):101-108, 1995.

# Approximation Algorithms for Some Clustering and Classification Problems

Eva Tardos

Department of Computer Science, Cornell University,
4105A Upson Hall, Ithaca, NY, USA.
http://www.cs.cornell.edu/home/eva/eva.html

**Abstract.** Clustering and classification problems arise in a wide range of application settings from clustering documents, placing centers in networks, to image processing, biometric analysis, language modeling and the categorization of hypertext documents.

The applications mentioned above give rise to a number of related algorithms problems, each of which are NP-complete. Approximation algorithms provide a framework to develop algorithms for such problems that have provable performance guarantees. In this talk we shall survey some of the general techniques, and recent developments in approximation algorithms for these problems.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, p. 183–183, 1999.
© Springer-Verlag Berlin Heidelberg 1999

# How Many People Can Hide in a Terrain?

Stephan Eidenbenz

Institute for Theoretical Computer Science,
ETH Zürich, Switzerland
eidenben@inf.ethz.ch

**Abstract.** How many people can hide in a given terrain, without any two of them seeing each other? We are interested in finding the precise number and an optimal placement of people to be hidden, given a terrain with $n$ vertices. In this paper, we show that this is not at all easy: The problem of placing a maximum number of hiding people is almost as hard to approximate as the MAXIMUM CLIQUE problem, i.e., it cannot be approximated by any polynomial-time algorithm with an approximation ratio of $n^\epsilon$ for some $\epsilon > 0$, unless $P = NP$. This is already true for a simple polygon with holes (instead of a terrain). If we do not allow holes in the polygon, we show that there is a constant $\epsilon > 0$ such that the problem cannot be approximated with an approximation ratio of $1 + \epsilon$.

## 1 Introduction and Problem Definition

While many of the traditional art gallery problems such as VERTEX GUARD and POINT GUARD deal with the problem of guarding a given polygon with a minimum number of guards, the problem of hiding a maximum number of objects from each other in a given polygon is intellectually appealing as well. When we let the problem instance be a terrain rather than a polygon, we obtain the following background, which is the practical motivation for the theoretical study of our problem: A real estate agency owns a large, uninhabited piece of land in a beautiful area. The agency plans to sell the land in individual pieces to people who would like to have a cabin in the wilderness, which to them means that they do not see any signs of human civilization from their cabins. Specifically, they do not want to see any other cabins. The real estate agency, in order to maximize profit, wants to sell as many pieces of land as possible.

In an abstract version of the problem we are given a terrain which represents the uninhabited piece of land that the real estate agency owns. A *terrain* $T$ is a two-dimensional surface in three-dimensional space, represented as a finite set of vertices in the plane, together with a triangulation of their planar convex hull, and a height value associated with each vertex. By a linear interpolation inbetween the vertices, this representation defines a bivariate continuous function. The corresponding surface in space is also called a 2.5-dimensional terrain. A terrain divides three-dimensional space into two subspaces, i.e. a space above and a space below the terrain, in the obvious way. In the literature, a terrain is also called a *triangulated irregular network* (TIN), see [8]. The problem now consists

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 184–194, 1999.
© Springer-Verlag Berlin Heidelberg 1999

of finding a maximum number of lots (of comparatively small size) in the terrain, upon which three-dimensional bounding boxes can be positioned that represent the cabins such that no two points of two different bounding boxes see each other. Two points *see* each other, if the straight line segment connecting the two points does not intersect the space below the terrain. Since the bounding boxes that represent the cabins are small compared to the overall size and elevation changes in the terrain (assume that we have a mountainous terrain), we may consider these bounding boxes to be zero-dimensional, i.e. to be points on the terrain. This problem has other potential applications in animated computer-games, where a player needs to find and collect or destroy as many objects as possible. Not seeing the next object while collecting an object makes the game more interesting. We are now ready to formally define the first problem that we study:

**Definition 1.** *The problem* Maximum Hidden Set on Terrain *asks for a set $S$ of maximum cardinality of points on a given terrain $T$, such that no two points in $S$ see each other.*

In a variant of the problem, we introduce the additional restriction that these points on the terrain must be vertices of the terrain.

**Definition 2.** *The problem* Maximum Hidden Vertex Set on Terrain *asks for a set $S$ of maximum cardinality of vertices of a given terrain $T$, such that no two vertices in $S$ see each other.*

In a more abstract variant of the same problem, we are given a simple polygon with or without holes instead of a terrain. A *simple polygon with holes* in the plane is given by its ordered sequence of vertices on the outer boundary, together with an ordered sequence of vertices for each hole. A *simple polygon without holes* in the plane is simply given by its ordered sequence of vertices on the outer boundary. Again, we can impose the additional restriction that the points to be hidden from each other must be vertices of the polygon. This yields the following four problems.

**Definition 3.** *The problem* Maximum Hidden Set on Polygon with(out) Holes *asks for a set $S$ of maximum cardinality of points in the interior or on the boundary of a given polygon $P$, such that no two points in $S$ see each other.*

**Definition 4.** *The problem* Maximum Hidden Vertex Set on Polygon with(out) Holes *asks for a set $S$ of maximum cardinality of vertices of a given polygon $P$, such that no two vertices in $S$ see each other.*

Two points in the polygon see each other, if the straight line segment connecting the two points does not intersect the exterior (and the holes) of the polygon. In this paper, we propose a reduction from Maximum Clique to Maximum Hidden Set on Polygon with Holes. The same reduction with minor modifications will also work for Maximum Hidden Set on Terrain, Maximum Hidden Vertex Set on Polygon with Holes, and Maximum Hidden

Vertex Set on Terrain. Maximum Clique cannot be approximated by a polynomial-time algorithm with a ratio of $n^{1-\epsilon}$ unless $coR = NP$ and with a ratio of $n^{\frac{1}{2}-\epsilon}$ unless $NP = P$ for any $\epsilon > 0$, where $n$ is the number of vertices in the graph [7]. We will show that our reduction is gap-preserving (a technique proposed in [1]), and thus show inapproximability results for all four problems. Maximum Clique consists of finding a maximum complete subgraph of a given graph $G$, as usual.

We also propose a reduction from Maximum 5-Occurrence-2-Satisfiability to Maximum Hidden Set on Polygon without Holes, which will also work for Maximum Hidden Vertex Set on Polygon without Holes. Maximum 5-Occurrence-2-Satisfiability is $APX$-hard, which is equivalent to saying that there exists a constant $\epsilon > 0$ such that no polynomial algorithm can achieve an approximation ratio of $1 + \epsilon$ for Maximum 5-Occurrence-2-Satisfiability. See [3] for an introduction to the class $APX$ and for the relationship between the two classes $APX$ and $MaxSNP$, see [11] for the $MaxSNP$-hardness proof for Maximum 5-Occurrence-2-Satisfiability. Please note that $MaxSNP$-hardness implies $APX$-hardness [3]. We show that our reduction is gap-preserving and thus establish the $APX$-hardness of Maximum Hidden (Vertex) Set on Polygon without Holes. Maximum 5-Occurrence-2-Satisfiability consists of finding a truth assignment for the variables of a given boolean formula. The formula consists of disjunctive clauses with at most two literals and each variable appears in at most 5 literals. The truth assigment must satisfy a maximum number of clauses.

There are various problems that deal with terrains. Quite often, these problems have applications in the field of telecommunications, namely in setting up communications networks. There are some upper and lower bound results on the number of guards needed for several kinds of guards to collectively cover all of a given terrain [2]. Very few results on the computational complexity of terrain problems are known. The shortest watchtower (from where a terrain can be seen in its entirety) can be computed in time $O(n \log n)$ [15]. The problem of finding a minimum number of vertices of a terrain such that guards at these vertices see all of the terrain is $NP$-hard and cannot be approximated with an approximation ratio that is better than logarithmic in the number of vertices of the terrain. Similar results hold for the variation, where guards may only be placed at a certain given height above the terrain [5]. When we deal with polygons rather than terrains, we speak of art gallery or visibility problems. Many results (upper and lower bounds, as well as computational complexity results) are known for visibility problems. See [10,13,14] for an overview, as well as more recent work on the inapproximability of Vertex/Edge/Point Guard on polygons with [4] and without holes [6].

The problems Maximum Hidden Set on a Polygon without Holes and Maximum Hidden Vertex Set on a Polygon without Holes are known to be $NP$-hard [12]. This immediately implies the $NP$-hardness of the corresponding problems for polygons with holes. A quite simple reduction from these polygon problems to the terrain problems (as given in Sect. 3) even implies

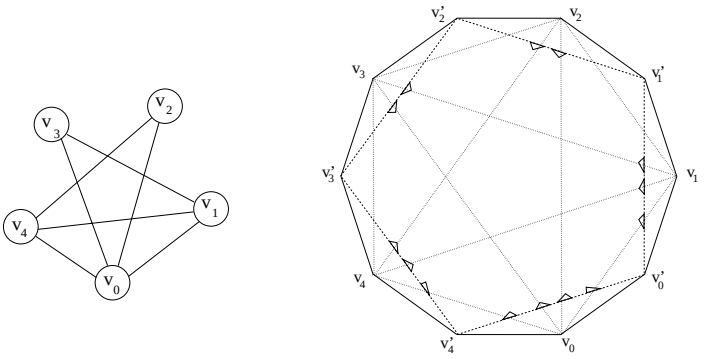



**Fig. 1.** Example graph and the polygon constructed from it

the $NP$-hardness for the two terrain problems as well. In this paper, we give the first inapproximability results for these problems. Our results suggest that these problems differ significantly in their approximation properties.

This paper is organized as follows. In Sect. 2, we propose a reduction from Maximum Clique to Maximum Hidden Set on Polygon with Holes. We show that our reduction is gap-preserving and obtain our inapproximability results for Maximum Hidden (Vertex) Set on a Polygon with Holes. We show that our proofs also work for Maximum Hidden (Vertex) Set on Terrain with minor modifications in Sect. 3. In Sect. 4, we show the $APX$-hardness of Maximum Hidden (Vertex) Set on Polygon without Holes. Finally, we draw some conclusions in Sect. 5.

## 2 Inapproximability Results for the Problems for Polygons with Holes

Suppose we are given an instance $I$ of Maximum Clique, i.e. an undirected graph $G = (V, E)$, where $V = v_0, \ldots, v_{n-1}$. Let $m := |E|$. We construct an instance $I'$ of Maximum Hidden Set on Polygon with Holes as follows. $I'$ consists of a polygon with holes. The polygon is basically a regular $2n$-gon with holes, but we replace every other vertex by a comb-like structure. Each hole is a small triangle designed to block the view of two combs from each other, whenever the two vertices, to which the combs correspond, are connected by an edge in the graph. Figure 1 shows an example of a graph and the corresponding polygon with holes. (Note that only the solid lines are lines of the polygon and also note that the combs are not shown in Fig. 1.)

Let the regular $2n$-gon consist of vertices $v_0, v'_0, \ldots, v_{n-1}, v'_{n-1}$ in counterclockwise order, to indicate that we map each vertex $v_i \in V$ in the graph to a vertex $v_i$ in the polygon. We need some notation, first. Let $e_{i,j}$ denote the intersection point of the line segment from $v'_{i-1}$ to $v'_i$ with the line segment from $v_i$ to $v_j$, as indicated in Fig. 2. (Note that we make liberal use of the notation

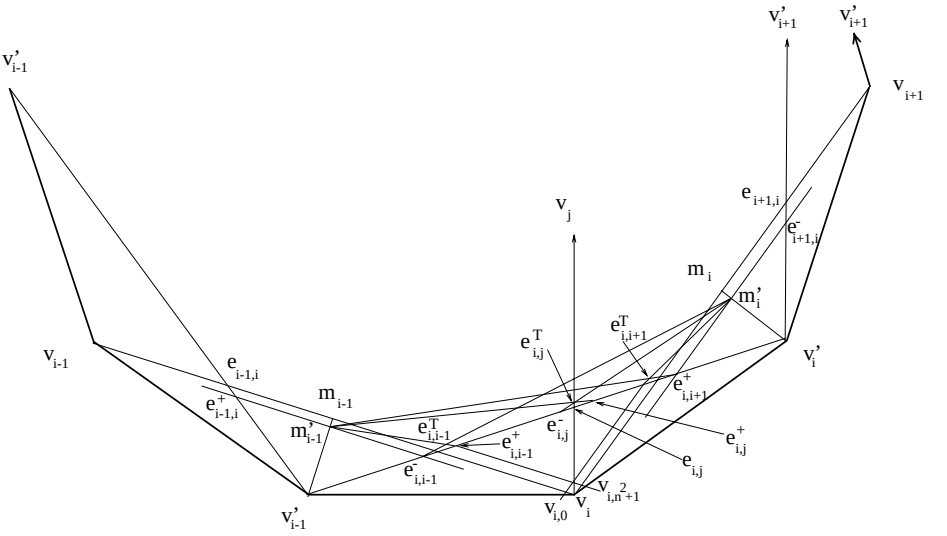



**Fig. 2.** Points $e^-_{i,j}$, $e^+_{i,j}$, and $e^T_{i,j}$

index for the vertices, i.e. $v_{i+1}$ is strictly speaking $v_{i+1 \bmod n}$, accordingly for $v_{i-1}$.) Let $d$ denote the minimum of the distances of $e_{i,j}$ from $e_{i,j+1}$, where the minimum is taken over all $i, j = 1, \ldots, n$. Let $e^-_{i,j}$ ($e^+_{i,j}$) denote the point at distance $\frac{d}{3}$ from $e_{i,j}$ on the line from $v'_{i-1}$ to $v'_i$ that is closer to $v'_{i-1}$ ($v'_i$). Let $m_i$ be the midpoint of the line segment from vertex $v_i$ to $v_{i+1}$ and let $m'_i$ be the intersection point of the line from $v'_i$ to $m_i$ and from $e^+_{i,i+1}$ to $e^-_{i+1,i}$ (see Fig. 2). Finally, let $e^T_{i,j}$ denote the intersection point of the line from $e^-_{i,j}$ to $m'_i$ and the line from $e^+_{i,j}$ to $m'_{i-1}$. The detailed construction of these points is shown in Fig. 2. We let the triangle formed by the three vertices $e^+_{i,j}$, $e^-_{i,j}$, and $e^T_{i,j}$ be a hole in the polygon iff there exists an edge in $G$ from $v_i$ to $v_j$. Recall Fig. 1, which gives an example.

We now refine the polygon obtained so far by cutting off a small portion at each vertex $v_i$. For each $i \in \{0, \ldots, n\}$, we introduce two new vertices $v_{i,0}$ and $v_{i,n^2+1}$ as indicated in Fig. 2. Vertex $v_{i,0}$ is defined as the intersection point of the line that is parallel to the line from $v_{i-1}$ to $v_i$ and goes through point $e^+_{i,i-1}$ and of the line from $v_i$ to $v'_i$. Symmetrically, vertex $v_{i,n^2+1}$ is defined as the intersection point of the line that is parallel to the line from $v_{i+1}$ to $v_i$ and that goes through point $e^-_{i,i+1}$ and of the line from $v_i$ to $v'_{i-1}$. We fix $n^2-1$ additional vertices $v_{i,1}, \ldots, v_{i,n^2}$ on the line segment from $v_{i,0}$ to $v_{i,n^2+1}$ for each $i$ as shown in Fig. 3. For a fixed $i$, the two vertices $v_{i,l}$ and $v_{i,l+1}$ have equal distance for all $l \in \{0, \ldots, n^2\}$. Finally, we fix $n^2$ additional vertices $w_{i,l}$ for $l \in \{0, \ldots, n^2\}$ for each $i$. Vertex $w_{i,l}$ is defined as the intersection point of the line from vertex $v'_{i-1}$ through $v_{i,l}$ with the line from vertex vertex $v'_i$ through $v_{i,l+1}$. The polygon between two vertices $v'_{i-1}$ and $v'_i$ is now given by the following ordered sequence

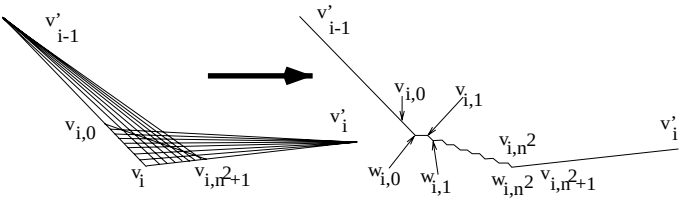



**Fig. 3.** Construction of the comb of $v_i$

of vertices: $v'_{i-1}, v_{i,0}, w_{i,0}, v_{i,1}, w_{i,1}, \ldots, v_{i,n^2}, w_{i,n^2}, v_{i,n^2+1}, v'_i$ as indicated in Fig. 3. We call the set of all triangles $v_{i,l}, w_{i,l}, v_{i,l+1}$ for a fixed $i$ and all $l \in \{0, \ldots, n^2\}$ the *comb of* $v_i$. We have the following property of the construction.

**Lemma 1.** *In any feasible solution $S'$ of the* Maximum Hidden Set on Polygon with Holes *instance $I'$, at most $2n$ points in $S$ can be placed outside the combs.*

*Proof.* In each of the $n$ trapezoids $\{v'_{i-1}, v'_i, v_{i,n^2+1}, v_{i,0}\}$ (see Figs. 1 and 2), there can be at most one point, which gives $n$ points in total. Moreover, by our construction any point $p$ in the trapezoid $\{v'_{i-1}, v'_i, m'_i, m'_{i-1}\}$ (not in the holes) can see every point $p'$ in the $n$-gon $\{v'_0, \ldots, v'_n\}$ except for points $p'$ in any of the holes and (possibly) except for points $p'$ in the triangles $\{v'_{i-1}, m'_{i-1}, e^+_{i-1,i}\}$ and $\{v'_i, m'_i, e^-_{i+1,i}\}$ (see Fig. 2). Therefore, all points in $S'$ that lie in the $n$-gon $\{v'_0, \ldots, v'_n\}$ must lie in only one of the $n$ polygons $\{e^+_{i-1,i}, m'_{i-1}, m'_i, e^-_{i+1,i}, v'_i, v'_{i-1}\}$. Obviously, at most $n$ points can be hidden in any one of these polygons. □

We have the following observation, which follows directly from the construction:

**Observation 1** *Any point in the comb of $v_i$ completely sees the comb of vertex $v_j$, if $(v_i, v_j)$ is not an edge in the graph. If $(v_i, v_j)$ is an edge in the graph, then no point in the comb of $v_i$ sees any point in the comb of $v_j$*

Given a feasible solution $S'$ of the Maximum Hidden Set on Polygon with Holes instance $I'$, we obtain a feasible solution $S$ of the Maximum Clique instance $I$ as follows: A vertex $v_i \in V$ is in the solution $S$, iff at least one point from $S'$ lies in the comb of $v_i$. To see that $S$ is a feasible solution, assume by contradiction that it is not a feasible solution. Then, there exists a pair of vertices $v_i, v_j \in S$ with no edge between them. But then, there is by construction no hole in the polygon to block the view between the comb of $v_i$ and the comb of $v_i$.

We need to show that the construction of $I'$ can be done in polynomial time and that a feasible solution can be transformed in polynomial time. There are $2n^2 + 1$ vertices in each of the $n$ combs. We have additional $n$ vertices $v'_i$. There are 2 holes for each edge in the graph and each hole consists of 3 vertices. Therefore, the polygon $P$ consists of $6m + 2n^3 + 2n$ vertices. It is known in computational geometry that the coordinates of intersection points of lines with

rational coefficients can be expressed with polynomial length. All of the points in our construction are of this type. Therefore, the construction is polynomial. The transformation of a feasible solution can obviously be done in polynomial time.

We obtain our inapproximability result by using the technique of gap-preserving reductions (as introduced in [1]), which consists of transforming a promise problem into another promise problem.

**Lemma 2.** *Let $OPT$ denote the size of an optimum solution of the* Maximum Clique *instance $I$, let $OPT'$ denote the size of an optimum solution of the* Maximum Hidden Set on Polygon with Holes *instance $I'$, and let $k \leq n$. The following holds: $OPT \geq k \Longrightarrow OPT' \geq n^2 k$*

*Proof.* If $OPT \geq k$, then there exists a clique in $I$ of size $k$. We obtain a solution for $I'$ of size $n^2 k$ by simply letting the $n^2$ vertices $w_{i,l}$ for $l \in \{0, \ldots, n^2\}$ be in the solution if and only if vertex $v_i \in V$ is in the clique. The solution thus obtained for $I'$ is feasible (see Observation 1). □

**Lemma 3.** *Let $OPT$ denote the size of an optimum solution of the* Maximum Clique *instance $I$, let $OPT'$ denote the size of an optimum solution of the* Maximum Hidden Set on Polygon with Holes *instance $I'$, let $k \leq n$, and let $\epsilon > 0$. The following holds: $OPT < \frac{k}{n^{1/2-\epsilon}} \Longrightarrow OPT' < \frac{n^2 k}{n^{1/2-\epsilon}} + 2n$*

*Proof.* We prove the contraposition: $OPT' \geq \frac{n^2 k}{n^{1/2-\epsilon}} + 2n \Longrightarrow OPT \geq \frac{k}{n^{1/2-\epsilon}}$. Suppose we have a solution of $I'$ with $\frac{n^2 k}{n^{1/2-\epsilon}} + 2n$ points. At most $2n$ of the points in the solution can be outside the combs, because of Lemma 1. Therefore, at least $\frac{n^2 k}{n^{1/2-\epsilon}}$ points must be in the combs. From the construction of the combs, it is clear that at most $n^2$ points can hide in each comb. Therefore, the number of combs that contain at least one point from the solution is at least $\frac{\frac{n^2 k}{n^{1/2-\epsilon}}}{n^2} = \frac{k}{n^{1/2-\epsilon}}$. The transformation of a solution as described above yields a solution of $I$ with at least $\frac{k}{n^{1/2-\epsilon}}$ vertices. □

Lemmas 2 and 3 and the fact that $|I'| \leq 10n^2$ allow us to prove our first main result, using standard concepts of gap-preserving reductions (see [1]). The proof easily carries over to the vertex restricted version of the problem.

**Theorem 1.** Maximum Hidden Set on Polygon with Holes *and* Maximum Hidden Vertex Set on Polygon with Holes *cannot be approximated by any polynomial time algorithm with an approximation ratio of $\frac{|I'|^{1/6-\gamma}}{4}$, where $|I'|$ is the number of vertices in the polygon, and where $\gamma > 0$, unless $NP = P$.*

## 3 Inapproximability Results for the Terrain Problems

**Theorem 2.** *The problems* Maximum Hidden Set on Terrain *and* Maximum Hidden Vertex Set on Terrain *cannot be approximated by any polynomial time algorithm with an approximation ratio of $\frac{|I''|^{1/6-\gamma}}{4}$, where $|I''|$ is the number of vertices in the terrain, and where $\gamma > 0$, unless $NP = P$.*

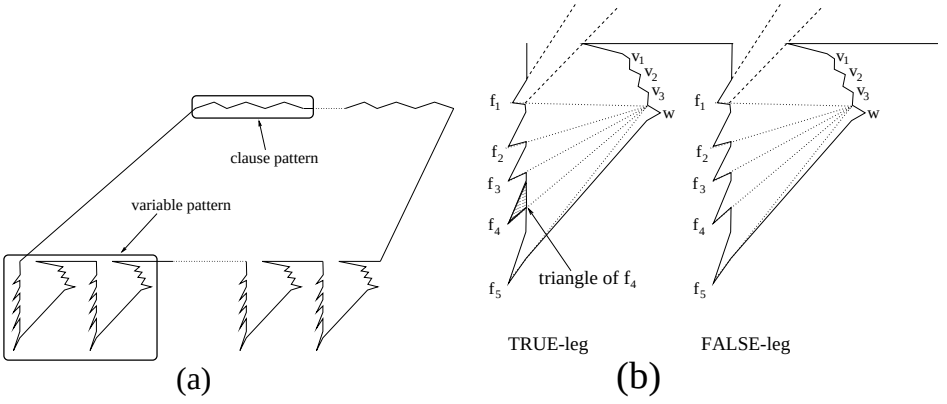



**Fig. 4.** (a) Schematic construction, (b) Variable pattern

*Proof.* The proof very closely follows the lines of the proof for the inapproximability of Maximum Hidden (Vertex) Set on Polygon with Holes. We use the same construction, but given the polygon with holes of instance $I'$ we create a terrain (i.e. instance $I''$) by simply letting all the area outside the polygon (including the holes) have height $h$ and by letting the area in the interior have height 0. We add four vertices to the terrain by introducing a rectangular bounding box around the regular $2n$-gon. This yields a terrain with vertical walls, which can be easily modified to have steep but not vertical walls, as required by the definition of a terrain. Finally, we triangulate the terrain. The terrain thus obtained looks like a canyon of a type that can be found in the south-west of the United States. All proofs work very similar. □

## 4 Inapproximability Results for the Problems for Polygons Without Holes

We reduce Maximum 5-Occurrence-2-Satisfiability to Maximum Hidden Set on Polygon without Holes to prove the $APX$-hardness of Maximum Hidden Set on Polygon without Holes. The same reduction will also work for Maximum Hidden Vertex Set on Polygon without Holes with minor modifications. Suppose we are given an instance $I$ of Maximum 5-Occurrence-2-Satisfiability, which consists of $n$ variables $x_0, \ldots, x_{n-1}$ and $m$ clauses $c_0, \ldots, c_{m-1}$. We construct a polygon without holes, i.e. an instance $I'$ of Maximum Hidden Set on Polygon without Holes, which consists of clause patterns and variable patterns, as shown schematically in Fig. 4 (a). The construction uses concepts similar to those used in [9]. The details of the construction are similar to a construction in [6], and will therefore be omitted. It is, however, necessary to introduce the variable pattern. We construct a variable pattern for each variable $x_i$ as indicated in Fig. 4 (b). The cone-like feature drawn with dashed lines simply helps in the construction and is not part of the polygon

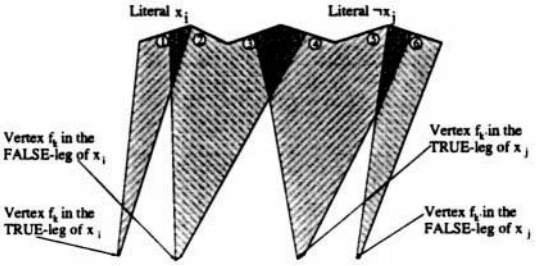



**Fig. 5.** Clause Pattern with cones

boundary. It represent the link to the clause patterns, as indicated in Fig. 5. Each variable pattern consists of a TRUE- and a FALSE-leg. The reduction has the following properties:

**Lemma 4.** *If there exists a truth assignment $S$ to the variables of $I$ that satisfies at least $(1-\epsilon)m$ clauses, then there exists a solution $S'$ of $I'$ with $|S'| \geq 10n + 2m + (1-\epsilon)m$.*

*Proof.* If variable $x_i$ is TRUE in $S$, then we let the vertices $f_1, \ldots, f_5$ and $w$ of the TRUE-leg of $x_i$, as well as the vertices $v_1, v_2, v_3$ and $w$ of the FALSE-leg of $x_i$ be in the solution $S'$. Vice-versa if $x_i$ is FALSE in $S$. This gives us $10n$ points in $S'$. The remaining points for $S'$ are in the clause patterns. Figure 5 shows the clause pattern for a clause $x_i, \neg x_j$[1], together with the cones that link the clause pattern to the corresponding variable patterns. Remember that these cones are not part of the polygon boundary. To understand Fig. 5, assume $x_i$ is assigned the value FALSE and $x_j$ is assigned the value TRUE, i.e., the clause $x_i, \neg x_j$ is not satisfied. Then there is a point in the solution that sits at vertex $f_k$ (for some $k$) in the FALSE-leg of $x_i$ and a point that sits at vertex $f'_k$ (for some $k'$) in the TRUE-leg of $x_j$. In this case, we can have only *two* additional points in the solution $S'$ at points ①, ⑥. In the remaining three cases, where the variables $x_i$ and $x_j$ are assigned truth values such that the clause is satisfied, we can have *three* additional points in $S'$ at ① – ⑥. Therefore, we have 2 points from all unsatisfied clauses and 3 points from all satisfied clauses, i.e. $2\epsilon m + 3(1-\epsilon)m$ points that are hidden in the clause patterns. Thus, $|S'| \geq 10n + 2m + (1-\epsilon)m$, as claimed.[2] □

**Lemma 5.** *If there exists a solution $S'$ of $I'$ with $|S'| \geq 10n + 3m - (\epsilon+\gamma)m$, then there exists a variable assignment $S$ of $I$ that satisfies at least $(1-\epsilon-\gamma)m$ clauses.*

*Proof.* For any solution $S'$, we can assume that in each leg of each variable pattern, all points in $S'$ are either in the triangles of vertices $f_1, \ldots, f_5$ and $w$,

[1] The proofs work accordingly for other types of clauses, such as $x_i, x_j$.

[2] Note that a point at some vertex $f_k$ actually sees a slightly larger cone than indicated in Fig. 5. This problem can be dealt with by making the triangle of $f_k$ very small. Corresponding methods are used to solve similar problems in [6] and [4].

or in the triangles of vertices $v_1, v_2, v_3$, and $w$ (see Fig. 4 (b) for the definition of these triangles), since any point in any triangle of $f_1, \ldots, f_5$ sees the triangles of $v_1, v_2, v_3$ completely and any single point in the leg outside these triangles would see almost all (at least 3) of these triangles, and we could obtain better solutions easily. We transform the solution $S'$ (with $|S'| \geq 10n + 3m - (\epsilon + \gamma)m$) in such a way that it remains feasible and that its size (i.e. the number of hidden points) does not decrease. This is done with an enumeration of all possible cases, i.e. we show how to transform the solution if there is a point in 3, 4, or 5 of the triangles of the points $f_1, \ldots, f_5$ in the TRUE-leg and the FALSE-leg of a variable pattern. The transformation is such that at the end, we have for each variable pattern the six points at $f_1, \ldots, f_5$, and $w$ from one leg in the solution and the 4 points $v_1, v_2, v_3$, and $w$ from the other leg. Thus, we can easily obtain a truth assignment for the variables by letting variable $x_i$ be TRUE iff the six points at $f_1, \ldots, f_5$, and $w$ from the TRUE-leg are in the solution. The transformed solution $S'$ consists of at least $10n + 3m - (\epsilon + \gamma)m$ points, $10n$ of which lie in the variable patterns. At most 3 points can lie in each clause pattern. If 3 points lie in a clause pattern, then this clause is satisfied. Therefore, if 2 points lie in each clause pattern, there are still at least $(1 - \epsilon - \gamma)m$ additional points in $S'$. These must lie in clause patterns as well. Therefore, at least $(1 - \epsilon - \gamma)m$ clauses are satisfied. □

Lemmas 4 and 5 show how to transform two promise problems into one another. By using standard concepts of gap-preserving reductions and by introducing some minor modification for the vertex-restricted problem, we obtain:

**Theorem 3.** Maximum Hidden (Vertex) Set on Polygon without Holes *is APX-hard, i.e. there exists a constant $\delta > 0$ for each of the two problems such that no polynomial time approximation algorithm for the problem can achieve an approximation ratio of $1 + \delta$, unless $P = NP$.*

## 5 Conclusion

We have shown that the problems Maximum Hidden (Vertex) Set on Polygon with Holes and Maximum Hidden (Vertex) Set on Terrain are almost as hard to approximate as Maximum Clique. We could prove for all these problems an inapproximability ratio of $O(|I'|^{1/3-\gamma})$, but under the assumption that $coR \neq NP$, using the stronger inapproximability result for Maximum Clique from [7]. Furthermore, we have shown that Maximum Hidden (Vertex) Set on Polygon without Holes is $APX$-hard. Note that an approximation algorithm for all considered problems that simply returns a single vertex achieves an approximation ratio of $n$. Note that our proofs can easily be modified to work as well for polygons or terrains, where no three vertices are allowed to be collinear. We have classified the problems Maximum Hidden (Vertex) Set on Polygon with Holes and Maximum Hidden (Vertex) Set on Terrain to belong to the class of problems inapproximable with an approximation ratio of $n^\epsilon$ for some $\epsilon > 0$, as defined in [1]. The $APX$-hardness results for

the problems for polygons without holes, however, do not precisely characterize the approximability characteristics of these problem. The gap between the best (known) achievable approximation ratio (which is $n$) and the best inapproximability ratio is still very large for these problems and should be closed in future research. As for other future work, we plan to consider several variations of the problems presented. For example, we plan to try to hide non-zero-dimensional objects.

## References


1. S. Arora and C. Lund; *Hardness of Approximations*; in: Approximation Algorithms for NP-Hard Problems (ed. Dorit Hochbaum), PWS Publishing Company, pp. 399-446, 1996.
2. P. Bose, T. Shermer, G. Toussaint, and B. Zhu; *Guarding Polyhedral Terrains*; Computational Geometry 7, pp. 173-185, Elsevier Science B. V., 1997.
3. P. Crescenzi, V. Kann; *A Compendium of NP Optimization Problems*; in the book by G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, *Complexity and Approximation. Combinatorial Optimization Problems and their Approximability Properties*, Springer-Verlag, Berlin, 1999; also available in an online-version at: http://www.nada.kth.se/theory/compendium/compendium.html.
4. S. Eidenbenz, C. Stamm, and P. Widmayer; *Inapproximability of some Art Gallery Problems*; Proc. 10th Canadian Conf. Computational Geometry, pp. 64-65, 1998.
5. S. Eidenbenz, C. Stamm, and P. Widmayer; *Positioning Guards at Fixed Height above a Terrain – An Optimum Inapproximability Result*; Lecture Notes in Computer Science, Vol. 1461 (ESA'98), p. 187-198, 1998.
6. S. Eidenbenz; *Inapproximability Results for Guarding Polygons without Holes*; Lecture Notes in Computer Science, Vol. 1533 (ISAAC'98), p. 427-436, 1998.
7. J. Hastad; *Clique is Hard to Approximate within* $n^{1-\epsilon}$; Proc. of the Symposium on Foundations of Computer Science, 1996.
8. M. van Kreveld; *Digital Elevation Models and TIN Algorithms*; in Algorithmic Foundations of Geographic Information Systems (ed. van Kreveld et al.), LNCS tutorial vol. 1340, pp. 37-78, Springer, 1997.
9. D. T. Lee and A. K. Lin; *Computational Complexity of Art Gallery Problems*; IEEE Trans. Info. Th, pp. 276-282, IT-32, 1986.
10. J. O'Rourke; *Art Gallery Theorems and Algorithms*; Oxford University Press, New York, 1987.
11. C.H. Papadimitriou; *Computational Complexity*; Addison-Wesley Publishing Company, 1994.
12. T. Shermer; *Hiding People in Polygons*; Computing 42, pp. 109 - 131, 1989.
13. T. Shermer; *Recent results in Art Galleries*; Proc. of the IEEE, 1992;
14. J. Urrutia; *Art Gallery and Illumination Problems*; in Handbook on Computational Geometry, edited by J.-R. Sack and J. Urrutia, 1998.
15. B. Zhu; *Computing the Shortest Watchtower of a Polyhedral Terrain in* $O(n \log n)$ *Time*; in Computational Geometry 8, pp. 181-193, Elsevier Science B.V., 1997.

# Carrying Umbrellas: An Online Relocation Problem on Graphs*

Jae-Ha Lee, Chong-Dae Park, and Kyung-Yong Chwa

Dept. of Computer Science, KAIST, Korea
{jhlee,cdpark,kychwa}@jupiter.kaist.ac.kr

**Abstract.** We introduce an online relocation problem on a graph, in which a player who walks around the nodes makes decisions on whether to relocate mobile resources, while not knowing the future requests. We call it *Carrying Umbrellas*. This paper gives a necessary and sufficient condition under which a competitive algorithm exists and describes an optimal algorithm and anaylzes its competitve ratio. We also extend this problem to the case of digraphs.

## 1 Introduction

"To carry an umbrella or not?" This is an everyday dilemma. To illustrate some of our concepts, we describe a detailed scenario. Picture a person who walks around $N$ places. At each place, he is told where to go next and then must go there. As a usual person, he dislikes being caught in the rain without umbrellas. Since today's weather forecast is correct, he knows if it will be rainy today. However, he does not know the future destinations and weathers, which might be controlled by a malicious adversary. We say a person is *safe* if he never get wet. There is a trivial safe strategy: $'$Always carry an umbrella$'$. However, carrying an umbrella in sunny days is annoying and stupid. As an alternative, he placed several umbrellas in advance and thinks about an *efficient* strategy; he hopes, through some cleverness, to minimize the number of days he carries umbrellas. By adopting the competitive analysis [8], we say a person's strategy is *competitive* if he carries umbrellas in a small number of days (for details, see Subsection 1.1). The following questions are immediate: 1. What is the minimum number of umbrellas with which the person can be safe and competitive? (For example, is placing one umbrella per place sufficient?) 2. What is a safe and competitive strategy? We formally define the problem next.

### 1.1 A Game

Let $G = (V, E)$ be a simple undirected graph with $N$ vertices and $M$ edges. (An extension to digraphs is straightforward and considered only in Section 5.) An integer $u(v)$ is associated with each vertex $v \in V$, indicating the number of

* Supported in part by KOSEF grant 98-0102-07-01-3.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 195–204, 1999.
© Springer-Verlag Berlin Heidelberg 1999

umbrellas placed on the vertex $v$. We use $u(G)$ to represent the total number of umbrellas in $G$. Consider an on-line game between a player and the adversary, assuming that $u(G)$ is fixed in advance.

**Game** Carrying-Umbrellas($G, u(G)$)
1. **Initialization**
   `Player` determines the initial configuration of $u(G)$ umbrellas;
   `Adversary` chooses the initial position $s \in V$ of `Player`;
2. **for** $i = 1$ **to** $L$ **do**
   (a) `Adversary` specifies $(v, w)$, where $w$ is a boolean value and $v \in V$ is adjacent to the current vertex of `Player`;
   (b) `Player` goes to $v$; If $w = 1$, he must carry at least one umbrella;

As an initialization, it is assumed that the player first determines the initial configuration of $u(G)$ umbrellas and later the adversary chooses the initial position $s \in V$ of the player, although our results described in the sequel also hold even if the initialization is done in reverse order. A *play* consists of $L$ *phases*, where $L$ is determined by the adversary and unknown to the player. In each phase, the adversary gives a request $(v, w)$, where $v$ is adjacent to the current vertex of the player and $w$ is a boolean value indicating the weather; $w = 1$ indicates that it is *rainy*, and $w = 0$ *sunny*. For this request $(v, w)$, the player must go to the specified vertex $v$ and decide whether to carry umbrellas or not. If $w = 1$, he must carry at least one umbrella; Only in sunny days, he may or may not carry umbrellas. Notice that the player can carry an arbitrary number of umbrellas, if available.

The player *loses* if he cannot find any umbrella at the current vertex in a rainy phase. A strategy of the player is said to be *safe* if it is guaranteed that whenever $w = 1$ the player carries at least one umbrella. Moreover, a strategy of the player should be *efficient*.

As a measure of efficiency, we adopt the *competitive ratio* [8], which has been widely used in analyzing the performance of online algorithms. Let $\sigma = \sigma_1 \cdots \sigma_L$ be a *request sequence* of the adversary, where $\sigma_i = (v_i, w_i)$ is the request in $i$-th phase. The *cost* of a strategy $\mathcal{A}$ for $\sigma$, written $C_{\mathcal{A}}(\sigma)$, is defined as the number of phases in which the player carries umbrellas by the strategy $\mathcal{A}$. Then, the *competitive ratio* of the strategy $\mathcal{A}$ is $\frac{C_{\mathcal{A}}(\sigma)}{C_{Opt}(\sigma)}$ where *Opt* is the optimal off-line strategy. (Since *Opt* knows the entire $\sigma$ in advance, it pays the minimum cost. However, *Opt* cannot be implemented by any player and is used only for comparison.) A strategy whose competitive ratio is bounded by $c$ is termed *$c$-competitive*; it is simply said to be *competitive*, if it is $c$-competitive for some bounded $c$ irrespective of the length of $\sigma$. The player *wins*, if he has a safe and competitive strategy; he *loses*, otherwise.

Not surprisingly, whether the player has a winning strategy depends on the number of umbrellas. In this paper, we are interested in it.

**Definition 1** *For a graph $G$, we define $u^*(G)$ to be the minimum number of umbrellas with which the player has a winning strategy in $G$.*

### 1.2 Summary of Our Results

Let $G = (V, E)$ be a simple graph (having no parallel edges) with $N$ vertices and $M$ edges. We show that $u^*(G) = M+1$. That is, no strategy of the player is safe and competitive if $u(G) < M+1$ (Section 3) and there exists a competitive strategy of the player if $u(G) \geq M+1$ (Section 4). The competitive ratio of our strategy is $b(G)$, the number of vertices in the largest biconnected component in $G$. Moreover, the upper bound is attained by a *'weak'* player that carries at most one umbrella in a phase, and the competitive ratio $b(G)$ is optimal for some graphs when $u(G) = M+1$. Finally, we extend this problem to the case of digraphs (Section 5). Interestingly, $u^*(G)$ of a digraph $G$ seems to be closely related with a structure of $G$. We present a general upper and lower bound on $u^*(G)$, which is naturally reduced to $M+1$ in undirected graphs.

### 1.3 Related Works

Every online problem can be viewed as a game between an online algorithm and the adversary[7,3,6,5]. In this prospect, Ben-David, et al. [3] introduced request-answer game as a general model of online problems. Clearly, the problem of the present paper is an example of the request-answer game. Many research works, though not directly related with our problem, have studied online problems on a graph. Such examples include the $k$-server problem [1], graph coloring [2,4].

## 2 Lower Bound

This section describes the lower bound on $u^*(G)$. For a simple explanation, we first consider a *'weak'* player that can carry at most one umbrella in a phase, and showed that $u^*(G) \geq M+1$ for the weak player. Later the same bound is extended to the general player that can carry multiple umbrellas.

**Theorem 1.** *Let $G$ be a simple graph (with no parallel edges) with $N$ vertices and $M$ edges. Under the constraint that the player can carry at most one umbrella in a phase, $u^*(G) \geq M+1$.*

*Proof.* Assume that $G$ is connected (otherwise, the number of umbrellas in some connected component of $G$ is no larger than the number of edges in it, and our proof can be applied to it). We show that if $u(G) \leq M$ then no strategy of the player is safe and competitive in $G$. Equivalently, it suffices to show that any safe strategy of the player is not $c$-competitive, for any fixed $c$ ($< \infty$). For convenience, imagine we are the adversary that would like to defeat the player.

Let $\sigma = \sigma_1\sigma_2 \cdots \sigma_L$ denote a request sequence that we generate. Recall that the starting vertex $s$ of the player is determined by the adversary; We choose an arbitrary vertex as $s$.

Suppose we were somehow able to make the player lie in $s$ so that $u^*(s)$ becomes $d+1$, where $d$ is the degree of $s$ in $G$. This would tell us that $G-\{s\}$ is *'deficient'* in umbrellas, that is, $u(G-\{s\})$ is strictly less than the number of

edges in $G-\{s\}$. In the next step, we make the player go into $G-\{s\}$. Even if the player has carried an umbrella, $u(G-\{s\})$ is no larger than the number of edges in $G-\{s\}$, and the player is in it. Now we use recursion: We make the player not safe and competitive in $G-\{s\}$. (In fact, some cautions are needed to make the entire competitive ratio unbounded.) So our subgoal is to make $u(s)$ increase to $d+1$.

Let $v_1, \cdots, v_d$ be the adjacent vertices of $s$ and let $C_1, \cdots, C_d$ denote the maximal connected components in $G-\{s\}$ such that $C_i$ includes $v_i$. Of course, $C_i$ happen to be equal to $C_j$ even when $i \neq j$. Let $u(C_k)$ denote the number of umbrellas placed in $C_k$. Observe that if $G-\{s\}$ is deficient in umbrellas, then some component $C_k$ also is.

We begin with explaining how to increase $u(s)$. Suppose that at the start of $(2i-1)$-th phase, the player is located at $s$ and $u(s)=j$. In subsequent two phases, the adversary makes the player go to $v_j$ and return to $s$. That is, $\sigma_{2i-1}=(v_j, \mathit{sunny})$ and $\sigma_{2i}=(s, w_{2i})$. The $2i$-th weather $w_{2i}$ is determined according to the player's decision in $(2i-1)$-th phase; If the player carried an umbrella in $(2i-1)$-th phase, $w_{2i}$ is set to *sunny*; Otherwise *rainy*. The strategy of the adversary is summarized below. The first and second **if** parts are the generation of $\sigma_{2i-1}$ and $\sigma_{2i}$, and the last **if** part is to check whether deficient $C_k$ exists. If it exists, the recursive procedure `Adversary`$(C_k, v_k)$ is called. Remember that if $u(s)$ becomes $d+1$, deficient $C_k$ exists and the recursive procedure starts.

```
Adversary(G, s)
    i = 1;
    while TRUE do
        if u(s) = j then
            σ_{2i-1} = (v_j, sunny);
        if (the player carried an umbrella for σ_{2i-1}) then
            σ_{2i} = (s, sunny);
          else
            σ_{2i} = (s, rainy);
        if (there is C_k such that u(C_k) < |E(C_k)|) then
            σ_{2i+1} = (v_k, sunny);
            Adversary(C_k, v_k);
    end-while
```

To see why this request sequence makes $u(s)$ increase, we define a weighted sum $\Phi$ of the umbrellas placed in vertex $s$ and its neighbors, where $j$ umbrellas in $s$ weighs $0.5, 1.5, \cdots, j$–0.5, respectively and each umbrella in vertex $v_i$ weighs $i$. Specifically,

$$\Phi = \sum_{i=1}^{d} i \cdot u(v_i) + \sum_{j=1}^{u(s)} (j - 0.5)$$

Let $\Phi_k$ denote $\Phi$ at the end of $k$-th phase. We are interested in the change between $\Phi_{2k-2}$ and $\Phi_{2k}$. Suppose that $u(s)$ is $j$ at the end of $(2k-2)$-th phase. Depending on the decision of the player, there are four cases to consider.

Case A. If the player carried an umbrella throughout $(2k–1)$-th and $2k$-th phase, we have $\Phi_{2k-2}=\Phi_{2k}$ because the number of umbrellas is unchanged. However, the weather must be *sunny* from `Adversary`$(G,s)$. Therefore, the player did an *useless carrying*.

Case B. If the player carried an umbrella from $s$ to $v_j$ only, $\Phi_{2k}=\Phi_{2k-2}+0.5$. This is because the umbrella moved had weight $j-0.5$ at $s$ and $j$ at $v_j$.

Case C. If the player carried an umbrella from $v_j$ to $s$ only, $\Phi_{2k}=\Phi_{2k-2}+0.5$. This is because the weight of the umbrella moved changes from $j$ to $j+0.5$.

Case D. The remaining case is that the player never carried an umbrella. This is impossible by `Adversary`$(G,s)$, because if the player didn't carry an umbrella in $(2k-1)$-th phase $w_{2k}$ is set to *rainy*.

In summary, over two phases $2k$–1 and $2k$, $\Phi$ increases or is unchanged; If $\Phi$ is unchanged, the player did *useless carrying* in consecutive sunny days. Note that successive *useless carryings* make the player's strategy not competitive, because *Opt* would not carry an umbrella over two *sunny* days. Hence, in order to be competitive, the player must sometimes increase $\Phi$.

How many times $\Phi$ can increase before $u(s)$ becomes $d+1$? Recall that once $u(s)$ becomes $d+1$, some deficient $C_k$ exists and we start the recursive procedures. While $u(s)\leq d$, each umbrella can have $2d$ different weights. Moreover, since $u(G)$ is less than or equal to the number of edges in $G$, $\Phi$ can increase at most $2d\times\frac{N^2}{2}<N^3$ phases before $u(s)$ becomes $d+1$. Thus, $\Phi$ can increase at most $N^3$ phases, before the recursive procedure is called.

In `Adversary`$(C_k,v_k)$, we recursively define a request sequence and weight sum $\Phi'$. Thus, in $C_k$, $\Phi'$ can increase in at most $(N-1)^3$ phases, before the player moves into some its deficient subgraph. In all recursive procedures, its weight sum can increase in at most $N^3+(N-1)^3+\cdots+2<N^4$ phases. In the limiting case that $N=2$, the player always does useless carrying. This means that the player carries an umbrella except at most $N^4$ phases, and that for the same input, *Opt* can carry an umbrella in at most $N^4$ phases.

Finally, we choose the length $L$ of input sequence $\sigma$ sufficiently large. Specifically, we let $L\geq(c+1)\cdot(N^4)$. Then, the cost of the player's strategy $\mathcal{A}$ is

$$C_{\mathcal{A}}(\sigma)\geq L-N^4\geq c\cdot(N^4)$$

And, the cost of *Opt* is

$$C_{Opt}(\sigma)<N^4$$

Therefore, the competitive ratio is, for an arbitrary $c(<\infty)$,

$$\frac{C_{\mathcal{A}}(\sigma)}{C_{Opt}(\sigma)}>c$$

completing the proof. □

With small modification, we can show the following theorem but details are omitted in this abstract.

**Theorem 2.** *If $G$ is a graph with $M$ edges (but no parallel ones), $u^*(G)\geq M+1$.*

## 3 Upper Bound

In this section, we show $u^*(G)$ is no larger than $M+1$. It suffices to show that if $u(G)=M+1$ then the player has a safe and competitive strategy. Throughout this section, we assume that $u(G)=M+1$. To show this theorem, we only consider the weak player that carries at most one umbrella in a phase. Imagine that we are the player that should be safe and competitive against the adversary.

The heart of our strategy is how to decide whether to carry an umbrella in sunny days. To help this decision, we maintain two things: labels of umbrellas and a rooted subtree of $G$. First, let us explain the labels, recalling that $u(G)=M+1$. $M$ umbrellas are labeled as $\gamma(e)$ for each edge $e \in E$, and one remaining umbrella is labeled as *Current*. Under these constraints, labels are updated during the game. In addition to the labels, we also maintain a dynamic rooted subtree $\mathcal{S}$ of $G$, called *skeleton*.

- $\mathcal{S}$ is a subtree of $G$ whose root is the current position of the player.
- The root of $\mathcal{S}$ has the umbrella labeled as *Current*.
- For an edge $e \in \mathcal{S}$, the child-vertex of $e$ has the umbrella labeled as $\gamma(e)$.
- For an edge $e \notin \mathcal{S}$, the umbrella labeled as $\gamma(e)$ lies in one endpoint of $e$.

Note that all umbrellas labeled as $\gamma(e)$ is placed on one endpoint of $e$. In an example of Figure 1a, current skeleton $\mathcal{S}$ is enclosed by a dotted line and every edge in $\mathcal{S}$ is directed towards the child, indicating the location of its umbrella.

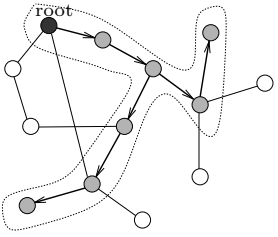



**Fig. 1.** Example of a skeleton (enclosed in a dotted line).

$\mathcal{S}$ is initialized as follows: The player first chooses an arbitrary spanning tree $T$ and places an umbrella in each vertex of $T$, with their labels undetermined. Then, the number of umbrellas used in $T$ is $N$. For an edge $e$ not in $T$, we place an umbrella on *any* endpoint of it; This umbrella is labeled as $\gamma(e)$. Then, the number of umbrellas labeled equals the number of edges not in $T$ that is $M-N+1$ and so the total number of umbrellas used is $M+1$. After the adversary chooses the start vertex $s$, the umbrellas in $T$ are labeled. The umbrella in $s$ is labeled as *Current*, and the umbrella in $v$ ($\neq s$) is labeled as $\gamma(e)$, where $e$ connects $v$ with its parent in $T$.

Now, we describe the strategy of the player. Suppose that currently, the player is at $v_{i-1}$ and the current request is $(v_i, w_i)$. If the weather is *rainy*, the

player has no choice; it has to move to $v_i$ with carrying the umbrella *Current*. In this case, the player resets $\mathcal{S}$ as a single vertex $\{v_i\}$. It is easily seen that the new skeleton satisfies the *invariants*, because only the umbrella *Current* is moved. The more difficult case is when the weather is *sunny*.

Suppose that $w_i = 0$, i.e., *sunny*. Depending on whether $v_i$ belongs to $\mathcal{S}$ or not, there are two cases. If $v_i$ belongs to $\mathcal{S}$ (Figure 2a), then the player moves to $v_i$ without an umbrella. Though no umbrellas are moved, labels must be changed. Let $e_1, e_2, \cdots, e_r$ be the edges encountered when we traverse from $v_{i-1}$ to $v_i$ in $\mathcal{S}$. The umbrella *Current* is relabeled as $\gamma(e_1)$ and the umbrella $\gamma(e_k)$ is relabeled as $\gamma(e_{k+1})$ for $1 \leq k \leq r-1$ and finally, the umbrella $\gamma(e_r)$ is relabeled as the new *Current*. Observe that new skeleton $\mathcal{S}$ still satisfies the *invariants*.

If $v_i$ does not belong to $\mathcal{S}$ (Figure 2b), the edge $e = (v_{i-1}, v_i)$ does not lie in $\mathcal{S}$. Remember that the umbrella $\gamma(e)$ was at $v_{i-1}$ or $v_i$ from the *invariants*. If the umbrella $\gamma(e)$ was placed at $v_{i-1}$, the player moves to $v_i$ carrying the umbrella *Current*, and adds the vertex $v_i$ and the edge $(v_{i-1}, v_i)$ to $\mathcal{S}$. If the umbrella $\gamma(e)$ was placed at $v_i$, the player moves to $v_i$ without carrying an umbrella and adds the vertex $v_i$ and the edge $(v_{i-1}, v_i)$ to $\mathcal{S}$ and additionally, swaps the labels of *Current* and $\gamma(e)$. In both cases, it is easy to see that $\mathcal{S}$ still satisfies the *invariants* (see Figure 2b).

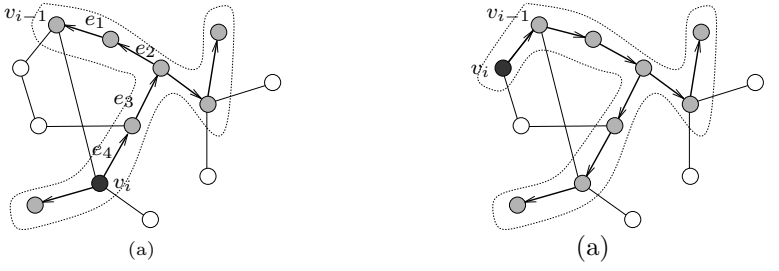



**Fig. 2.** After Fig. 1, $(v_i, \textit{sunny})$ is given. (a) $v_i$ is in $\mathcal{S}$. (b) $v_i$ is not in $\mathcal{S}$.

**Theorem 3.** *Let $G$ be a graph with $N$ nodes and $M$ edges. If $u(G) \geq M+1$, the player's strategy is safe and $b(G)$-competitive, where $b(G)$ is the number of vertices in the largest biconnected component in $G$.*

*Proof.* First, the player's strategy is safe from the *invariant* that the player always has the umbrella *Current*.

Next, we show that the strategy of the player is $N$-competitive. Let us divide the request sequence into a number of *stages*, each of which contains exactly one '*rainy*' and starts with '*rainy*'. Remember that the player resets $\mathcal{S}$ to a single vertex at the start of every stage. The cost of $\texttt{Player}(G)$ in a stage is at most $N$, because the cost 1 is paid only when $\mathcal{S}$ is set to a single vertex or becomes larger. The cost of *Opt* in a stage is at least one, because each stage starts with '*rainy*'. Therefore, the competitive ratio is at most $N$. The competitive ratio can

be refined to $b(G)$ by observing that for one rainy phase from $u$ to $v$, we only have to reset $S$ only in the biconnected component containing $u$ and $v$. □

Thus we have that $u^*(G)=M+1$. Unfortunately, the above competitive ratio is best possible for some graphs when $u(G)=M+1$.

**Theorem 4.** *For an $N$-node ring $G$ with $u(G)=M+1$, the competitive ratio of any strategy is at least $N$.*
Proof. *Omitted.*

## 4 Directed Graphs

In this section, we extend the result of the previous section to the case of digraphs. Our motivation is as follows: Let $G = (V(G), A(G))$ be a digraph. (For convenience, it is assumed that $G$ is strongly connected, that is, there is a path from any vertex to any other vertex.) In the special case that $G$ is a symmetric digraph (obtained by replacing each edge of an undirected graph by two directed arcs with reverse directions), $u^*(G) = \frac{|A(G)|}{2} + 1$ by theorems in Section 3 and 4. Does it hold for every digraph? It is easily seen that the answer is no. A counterexample is given in Figure 3. For a graph $G$ made by adding directed arcs alternatively to the undirected $n$-star, we have that $u^*(G) = u^*(n\text{-star})$, irrelevant with the number of added arcs.

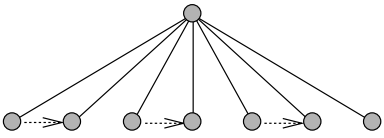


**Fig. 3.** For this graph $G$, $u^*(G) = u^*(8\text{-star}) = 8$. Thus $u^*(G)$ of a directed graph $G$ seems not to be directly determined by $|A|$, in contrast with undirected graphs.

This section shows upper and lower bound on $u^*(G)$, which are not tight in general but reduce to $|E| + 1$ for undirected graphs.

Let $G$ be a digraph. See Figure 4. $G_c$ is made by contracting a path consisting of vertices with indegree 1 and outdegree 1. More exactly, it $\langle v_1, v_2 \cdots v_k \rangle$ be a *one-way* path such that the indegree and outdegree of $v_i$ $(2 \le i \le k-1)$ is 1, we replace it by an arc $\langle v_1, v_k \rangle$. This operation is called a *contraction* and $G_c$ is obtained by repeatedly applying to $G$ contractions as long as possible. Let $\bar{G}_c$ be the undirected graph (with no parallel edges) such that $V(\bar{G}_c) = V(G_c)$ and $\bar{G}_c$ contains an edge $(u, w)$ if and only if $\bar{G}_c$ contains an arc $\langle u, w \rangle$ or $\langle w, u \rangle$. Observe that $\bar{G}_c$ is uniquely defined from $G_c$. We call $\bar{G}_c$ the *underlying graph* of $G_c$.

**Theorem 5.** *Let $G$ be a digraph. Then $u^*(G) \le |E(\bar{G}_c)| + |V(G)| - |V(G_c)| + 1$.*

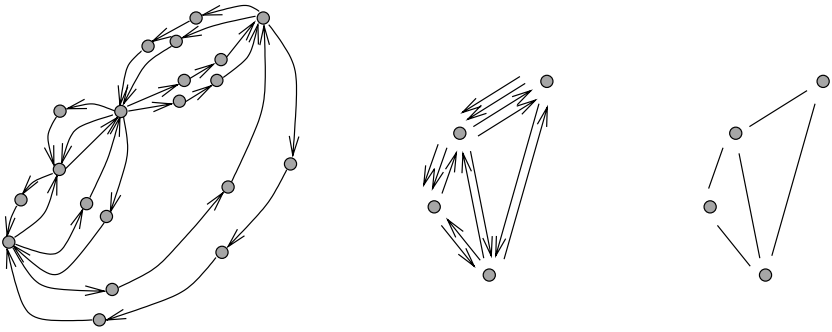


**Fig. 4.** (a) A graph $G$, (b) its contracted one $G_c$, and (c) its underlying graph $\bar{G}_c$.

*Proof.* Suppose that $u^*(G) = |E(\bar{G}_c)| + |V(G)| - |V(G_c)| + 1$. We present a competitive strategy. Among the vertices in $G$, we call the ones in $G_c$ *c-vertices*, the ones not in $G_c$ *p-vertices*. Note that the number of $p$-vertices is $|V(G)| - |V(G_c)|$. Roughly speaking, the player's strategy decides whether to carry umbrellas or not, depending on $|E(\bar{G}_c)| + 1$ umbrellas placed on $c$-vertices

Let us explain the initialization of $u(G)$ umbrellas. $|E(\bar{G}_c)| + 1$ umbrellas are placed on $c$-vertices by applying the initialization of the undirected graphs to $\bar{G}_c$. Each of $|V(G)| - |V(G_c)|$ umbrellas is placed at each $p$-vertices. Then each $p$-vertex has one and only one umbrella.

Consider a phase, say $j$, in which it is rainy for the first time. Let $\langle v_1, v_2 \cdots v_k \rangle$ be a one-way path in $G$ where $v_1$ and $v_k$ are $c$-vertices and every $v_i$ $(i \neq 1, k)$ is a $p$-vertex. Without losing generality, suppose that the player moves from $v_i$ to $v_{i+1}$ at phase $j$. Then the player carries an umbrella afterwards until it reaches $v_k$. If $i = 1$, then the player carries an umrbella from $v_1$ to $v_k$, which corresponds to one rainy phase occured in $\bar{G}_c$. Otherwise, that is, if $2 \leq i \leq k-1$, the player carries an umrbella from $v_i$ to $v_k$. By assuming that $v_1$ borrows an umbrella from $v_i$ and this umbrella will be returned to $v_i$ when the player goes to $v_i$ again, we can imagine as if the player carries an umrbella from $v_1$ to $v_k$. Thus any case corresponds to one rainy phase occured in $\bar{G}_c$.

Afterwards, as in Theorem 3, we construct the skeleton $S$ in $\bar{G}_c$ and decides whether to carry an umbrella or not according to $S$ in $\bar{G}_c$. Suppose that the player lies on a $c$-vertex $v_1$ and the adversary requests to move to $v_2$ where $\langle v_1, v_2, \cdots v_k \rangle$ is a one-way path. If it is rainy, we also follows the routine explained above and skeleton becomes only one vertex $v_k$. Otherwise, consider the strategy of the player on $\bar{G}_c$ and compute if the player carries an umbrella for the request $(v_1, v_k)$ on $\bar{G}_c$. If the the palyer carries an umbrella on $\bar{G}_c$, he also does in $G$ and always carries an umbrella while traveling to $v_k$. Otherwise, the player does not carry an umbrella, but if $v_1$ borrowed an umbrella from $v_i$, $v_1$ returns it.

After one rainy phase, the player can construct a full skeleton by carring an umbrella whenever visiting a $c$-vertex not in $S$. Thus the player's strategy is

$|V(G)|$-competitive. In fact, it is $b(G)$-competitive where $b(G)$ is the size of the largest biconnected component in $G$. □

Lower bound can also be obtained for digraphs (but proofs are omitted): For a digraph $G$, let $\bar{G}'_c$ be the undirected graph (with no parallel edges) such that $V(\bar{G}'_c) = V(G_c)$ and $\bar{G}_c$ contains an edge $(u, w)$ if and only if $\bar{G}_c$ contains both arcs $\langle u, w\rangle$ *and* $\langle w, u\rangle$. Then $u^*(G) \geq |E(\bar{G}'_c)| + |V(G)| - |V(G_c)| + 1$. Thus the gap between upper and lower bound on $u^*(G)$ is $|E(\bar{G}_c)| - |E(\bar{G}'_c)|$. Observe that Theorem 5 is optimal for the graph given in Figure 4.

## 5 Concluding Remarks

This paper is the first attempt on an online problem *Carrying Umbrellas*, and many questions remain open. First, finding $u^*(G)$ for a digraph $G$ seems to be interesting in a graph-theoretical viewpoint. Similarly, the characterization of the class of graphs such that $u^*(G) = |V(G)|$ is an interesting open problem (among undirected graphs, such examples are only trees). Second, we think that the problem *Carrying Umbrellas* can be extended to be of practical use by adopting different relocation constraints and considering general resources rather than umbrellas. For now, however, it is mainly of theoretical interest. Finally, reducing the competitive ratio with more umbrellas is an open problem.

**Acknowledgements** grateful to Oh-Heum Kwon for motivating this work.

## References

1. N. Alon, R. M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the -server problem. *SIAM J. of Comput.*, 24(1):78–100, 1995.
2. R. Beigel and W.I Gasarch. The mapmaker's dilemma. *Discrete Applied Mathematics*, 34(1):37–48, 1991.
3. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Widgerson. On the power of randomization in online algorithms. In *Proc. of 22nd STOC*, pages 379–386, 1990.
4. Hans L. Bodlaender. On the complexity of some coloring games. *Int. J. of Foundation of Computer Science*, 2:133–147, 1991.
5. A. Borodin and R. El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, 1998.
6. M. Chrobak and L. Larmore. The searver problem and online game. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 7, pages 11–64. AMS-ACM, 1992.
7. P. Raghavan and M. Snir. Memory versus randomization in online algorithms. In *Proc. of 16th ICALP*, pages 687–703, 1987.
8. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of ACM*, 2(28):202–208, 1985.

# Survivable Networks with Bounded Delay: The Edge Failure Case*

## (Extended Abstract)

Serafino Cicerone[1], Gabriele Di Stefano[1], and Dagmar Handke[2]

[1] Dipartemento di Ingegneria Elettrica, Università degli Studi di L'Aquila, I-67040 Monteluco di Roio - L'Aquila, Italy.
{cicerone,gabriele}@infolab.ing.univaq.it
[2] Universität Konstanz, Fakultät für Mathematik und Informatik, 78457 Konstanz, Germany.
Dagmar.Handke@uni-konstanz.de

**Abstract.** We introduce new classes of graphs to investigate networks that guarantee constant delays even in the case of multiple edge failures. This means the following: as long as two vertices remain connected if some edges have failed, then the distance between these vertices in the faulty graph is at most a constant factor times the original distance. In this extended abstract, we consider the case where the number of edge failures is bounded by a constant $\ell$. These graphs are called ( ,$\ell$)–*self-spanners*. We prove that the problem of maximizing $\ell$ for a given graph when $> 4$ is fixed is $\mathcal{NP}$-complete, whereas the dual problem of minimizing when $\ell$ is fixed is solvable in polynomial time. We show how the *Cartesian product* affects the self-spanner properties of the composed graph. As a consequence, several popular network topologies (like *grids*, *tori*, *hypercubes*, *butterflies*, and *cube-connected cycles*) are investigated with respect to their self-spanner properties.

## 1 Introduction

Fault-tolerance represents one of the major concerns in network design, and a large amount of research has been devoted to creating fault-tolerant parallel architectures. The techniques used in this research can be divided into two types. The first type consists of techniques that do not add redundancy to the desired architecture. Instead, these techniques attempt to mask the effects of faults by using the healthy part of the architecture to simulate the entire machine (e.g., see [1,6,12]). The second type consists of techniques that add redundancy to the desired architecture. These techniques attempt to isolate the faults while maintaining the complete desired architecture (e.g., see [3,15]). The goal of these techniques is to maintain the full performance of the desired architecture while minimizing the cost of the redundant components. Other works present

* Research partially supported by Italian MURST project 'Teoria dei Grafi e Applicazioni', and by the *Deutsche Forschungsgemeinschaft* under grant Wa 654/10-2

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 205–214, 1999.
© Springer-Verlag Berlin Heidelberg 1999

algorithms for routing messages around faults, or show how to perform certain computations in networks containing faults.

In this paper we follow a different approach. Starting from the observation that networks are usually modeled by graphs, we introduce and characterize new classes of graphs that guarantee constant delay factors even when a limited number of edges have failed. These graphs are called $(k, \ell)$–*self-spanners*, since a strict relationships to the concept of $k$–*spanners* [13].

A network modeled as a $(k, \ell)$–self-spanner graph can be characterized as follows: If at most $\ell$ edges have failed, as long as two vertices remain connected, the distance between these vertices in the faulty graph is at most $k$ times the distance in the non-faulty graph. By fixing the values $k$ and $\ell$ (called *stretch factor* and *fault-tolerance*, respectively), we obtain a specific new graph class. We are interested in characterizational as well as structural aspects of this class.

Motivations to this work are from [4] and [8]. In [4], Cicerone and Di Stefano introduce a class of graphs which guarantees constant delays even in the case of an unlimited number of *vertex failures* (but their results do not carry over to the dual case of *edge failures*). In [8], Handke considers the problem of finding sparse subgraphs in a given graph such that the distance within this subgraph is at most a (fixed) constant times the original distance, even when an edge or a vertex in the subgraph fails.

This extended abstract contains the following results: After a short introduction into the basic notation, we investigate networks modeled by $(k, \ell)$–*self-spanners*. The first main results concern the problems of deciding whether a given a graph is a $(k, \ell)$–self-spanner: The problem is $\mathcal{NP}$-complete for the general case where $k$ and $\ell$ are part of the input and remains $\mathcal{NP}$-complete if $k > 4$ is fixed. However, if $k \leq 3$ is fixed, or if $\ell \geq 0$ is fixed, then there are polynomial time algorithms. Thus, only the case where $k = 4$ is fixed remains open.

In the subsequent subsection, we examine how a graph that arises by the *Cartesian product* inherits the self-spanner properties of the underlying graphs. We also show strong results especially for small stretch factors and fault-tolerance values.

The last part shows how the new graph classes of $(k, \ell)$–self-spanners fit into the context of some popular network topologies. We show for example that mesh-like topologies such as *grids*, *tori*, and *hypercubes* exhibit strong self-spanner properties in particular for small fault-tolerance values. Bounded-degree approximations of the hypercube such as *butterflies* and *cube-connected cycles*, however, result in big stretch factors even in the case of small fault-tolerance values.

We have also considered $k$–*self-spanners*, i.e., networks that guarantee constant delays even in the case of an *unlimited* number of edge failures, but, due to space limitation, in this extended abstract we only summarize results concerning $(k, \ell)$–self-spanners. Full details and omitted proofs are included in [5].

## 2 Basic Notation

In this work, we use standard notation for graphs. Let $G = (V, E)$ be a simple, unweighted, and undirected graph with $n = |V|$ vertices and $m = |E|$ edges.

$G - e$, where $e \in E(G)$, is the graph obtained from $G$ by deleting edge $e$. $d_G(u, v)$ is the *distance* between two vertices $u$ and $v$ in $G$, i.e., the length of a shortest path.

An edge is a *chord* of a cycle $C$ if it connects two non-adjacent vertices of $C$. A cycle $C$ is *induced* if it does not contain a chord. $P_n$ is the *path graph* and $C_n$ is the *induced cycle graph* (also called ring), respectively, with $n$ vertices. The graph $K_n$ is the complete graph on $n$ vertices, and $K_3$ is also called *triangle*.

For a connected graph, an *articulation vertex* is a vertex the deletion of which disconnects the graph. A graph is called *biconnected* (or 2*–vertex–connected*) if it has no articulation vertex. It is called $\ell$*–vertex–connected* if there is no subset of vertices $S$ of size $\ell - 1$ such that the subgraph of $G$ induced by vertices in $V \backslash S$ is disconnected. A graph is $\ell$*–edge–connected* if no deletion of a $\ell - 1$ edges disconnects it. An edge $e$ of $G$ is called *bridge* if $G - e$ is disconnected.

For any fixed rational $k \geq 1$, a $k$*–spanner* of an unweighted graph $G$ is a spanning subgraph $S$ in $G$ such that the distance between every pair of vertices in $S$ is at most $k$ times their distance in $G$. The parameter $k$ is called *stretch factor*. It is clear that an unweighted graphs $S$ is a $k$–spanner of $G$ if and only if $S$ is a $\lfloor k \rfloor$–spanner of $G$. Thus, it suffices to consider integer stretch factors.

*Remark 1.* [13] A subgraph $S = (V, E')$ of a graph $G = (V, E)$ is a $k$–spanner if and only if $d_S(u, v) \leq k$, for every edge $e = \{u, v\} \in E \backslash E'$.

## 3 Bounded Delay and Limited Fault-Tolerance

We are interested in graphs that exhibit the following property: If at most $\ell$ edges in a graph $G$ fail, then for all pairs of vertices that remain connected a distance constraint is fulfilled. Thus, the number of faulty edges is limited to a fixed number. See the full paper [5] for results on the case where the number of edge failures is unlimited. Observe that we do not care for cases where two vertices are separated by the failure of edges since then the definition of distance does not apply; see also Remark 2 below. This leads to the following definition:

**Definition 1 ($(k, \ell)$–self-spanner).**

1. *For any fixed real $k \geq 1$ and fixed integer $\ell \geq 0$, a graph $G = (V, E)$ is called a* $(k, \ell)$–self-spanner *if for every subgraph $G' = (V, E')$ of $G$ with $|E'| \geq m - \ell$,*
$$d_{G'}(u, v) \leq k \cdot d_G(u, v)$$
*for every pair $\{u, v\}$ of connected vertices in $G'$.*
*Denote the class of all $(k, \ell)$–self-spanners by $SS(k, \ell)$. The parameter $k$ is called* stretch factor*, the parameter $\ell$ is called* fault-tolerance value *of the class $SS(k, \ell)$.*
2. *For a graph $G$, $minS_\ell(G)$ denotes the smallest $k$ such that $G \in SS(k, \ell)$, whereas $maxT_k(G)$ denotes the largest $\ell$ such that $G \in SS(k, \ell)$.*

As an example, consider the 'opaque cube' $G$ as shown in Figure 1. $G$ belongs to $SS(3,1)$. Since $minS_1(G) = 3$ and $maxT_3(G) = 1$, then $G$ does not belong to $SS(3,2)$. Observe that the definition works equally well for connected and disconnected graphs; but it is obvious that we can restrict our analysis to connected graphs (otherwise we can deal with each connected component separately). Thus, in the following we only consider *connected* graphs.

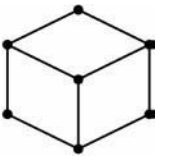


**Fig. 1** Opaque cube.

By similar arguments as in Remark 1, it suffices to consider only *faulty edges* of each subgraph if we want to check whether a given graph belongs to $SS(k,\ell)$: For any subgraph $G' = (V, E')$ of $G = (V, E)$ with $|E'| \geq |E| - \ell$ and $E' \subseteq E$, we have to check if

(*) $d_{G'}(u, v) \leq k$ for every $e = \{u, v\} \in E \backslash E'$.

As a consequence, in the following, we only deal with *integer* stretch factors $k$. We can furthermore simplify the procedure to check whether a graph belongs to a class $SS(k,\ell)$: we do not have to consider all (possibly disconnected) subgraphs but only connected subgraphs. We get the following:

**Lemma 1.** *For any fixed integers $k \geq 1$ and $\ell \geq 0$, $G \in SS(k,\ell)$ if and only if every* connected *and* spanning *subgraph $G' = (V, E')$ with $|E'| \geq m - \ell$ and $E' \subseteq E$, is a $k$–spanner of $G$.*

*Proof.* It suffices to show '$\Leftarrow$': Suppose every connected spanning subgraph $G' = (V, E')$ with $|E'| \geq m - \ell$ and $E' \subseteq E$ is a $k$–spanner of $G$ and, by contradiction, assume that $G$ is *not* a $(k,\ell)$–self-spanner. By definition, there is a subgraph $G'' = (V, E'')$ with $|E''| \geq m - \ell, E'' \subseteq E$ (not necessarily connected) such that there is a pair of vertices $u$ and $v$ (within one connected component of $G''$) and $d_{G''}(u, v) > k \cdot d_G(u, v)$. Since $G$ is connected, there is also a connected subgraph $\widetilde{G} = (V, \widetilde{E})$ with $E'' \subset \widetilde{E} \subseteq E$ (and thus $|\widetilde{E}| \geq m - \ell$) constructed as follows: Let $\mathcal{C}$ be the set of connected components of $G''$. Obtain $\widetilde{G}$ from $G''$ by adding $|\mathcal{C}| - 1$ bridge edges such that $\widetilde{G}$ is minimally connected. Then $d_{\widetilde{G}}(u, v) > k \cdot d_G(u, v)$ and thus $\widetilde{G}$ is not a $k$–spanner of $G$, a contradiction. □

This lemma motivates the name of the class by its strict relationship with the concept of $k$–spanners: a graph of this class 'spans itself'. Note that we cannot directly incorporate vertex failures. Consider for example again the 'opaque cube' as shown in Figure 1. As stated above, this graph is in $SS(3,1)$, but the graph $G'$ obtained from removing the internal vertex is not (in fact, it has a stretch factor $minS_1(G) = 5$, and thus is in $SS(5,1)$). Hence, in the case of $SS(k,\ell)$, we purely model edge failures. In the sequel, we use Lemma 1 as a characterization of the class $SS(k,\ell)$.

*Remark 2.* Note that the definition of $(k,\ell)$–self-spanners does *not* imply that $G$ is $(\ell+1)$–edge–connected. As stated above, we do not care for pairs of vertices (or edges) that are separated by the edge failures. If we want to take this into account (e.g., to achieve 'true' fault-tolerance, such that we can always guarantee for a 'short' connection between any pair of vertices even in the case of $\ell$ edge

failures) we can restrict our attention to graphs belonging to the intersection of the classes of $(\ell+1)$–edge–connected graphs and $(k,\ell)$–self-spanners.

### 3.1 Characterization of $(k,\ell)$–Self-Spanners

We are interested in finding (strict) efficient characterizations for the class $SS(k,\ell)$. For this aim, we start by stating some (more or less) straightforward results on $(k,\ell)$–self-spanners and define the problems to be considered formally. As our main results, we establish an almost complete set of complexity results for these problems. Let us first consider some trivial cases.

**Lemma 2.**

1. No delay, i.e., $k=1$: *For all* $\ell>0$, $SS(1,\ell)$ *is the set of all trees. If we omit the connectivity constraint then* $SS(1,\ell)$ *is the set of all forests.*
2. No edge failure, i.e., $\ell=0$: *For all* $k\geq 1$, $SS(k,0)=SS(1,0)$ *is the set of all (connected) graphs.*
3. Weak delay constraints, i.e., large stretch factors: *Any connected graph* $G$ *belongs to* $SS(k,\ell)$ *for all* $k\geq|V|-1$, *and for any* $\ell\geq 0$.
4. Strong fault-tolerance constraints, i.e., large fault-tolerance values: *Let* $G$ *be any connected graph. If* $G$ *belongs to* $SS(k,|E|-|V|+1)$ *(for some* $k\geq 1$*), then* $G$ *also belongs to* $SS(k,\ell)$ *for all* $\ell\geq|E|-|V|+1$.

*Proof.* Parts 1 and 2 are straightforward. To see Part 3, observe that if we allow for a stretch factor of $k\geq|V|-1$ then we do not really impose a distance constraint: any vertex of $G$ may be used for a detour. It remains to proof Part 4. If $G$ is a $(k,\ell)$–self-spanner for $\ell=|E|-|V|+1$ then, in particular, every spanning tree of $G$ is a $k$–spanner of $G$. If more than $|E|-|V|+1$ edges fail then the resulting subgraph is necessarily disconnected and thus (by Lemma 1) no further constraints are imposed. □

Thus in the following, given a graph $G$ we will only consider stretch factors of $2\leq k\leq|V|-2$ and fault-tolerance values of $1\leq\ell\leq|E|-|V|+1$. The cases for small and large stretch factors and small fault-tolerance values can be considered trivial, whereas the case of large fault-tolerance values coincides with the case of $k$–self-spanners (see [5]). By this lemma, it is clear that for every connected graph $G$ there are some parameters $k$ and $\ell$ such that $G$ belongs to $SS(k,\ell)$. Analogously, if we fix one of the parameters we can always find a feasible value for the other parameter. It is easy to see that $(k,\ell)$–self-spanners have inductive properties with respect to the parameters as stated below.

**Lemma 3.**

1. *If* $k\leq k'$ *then* $SS(k,\ell)\subseteq SS(k',\ell)$.
2. *If* $\ell\leq\ell'$ *then* $SS(k,\ell')\subseteq SS(k,\ell)$.

*Remark 3.* The class $SS(k,\ell)$ is *not* closed under subgraphs (cf. Figure 1). Also it is not closed under supergraphs in the following sense: If a graph $G$ is in $SS(k,\ell)$ for some fixed $k$ and $\ell$ then there may be a supergraph of $G$ on the same vertex set (i.e., a graph with additional edges) that does *not* belong to $SS(k,\ell)$. The same still holds if we consider only $(\ell+1)$–edge–connected graphs.

As a consequence of the previous remark, the self-spanner properties of a graph cannot be inferred directly from the self-spanner properties of sub- or supergraphs. For examples of standard graphs that exhibit some particular self-spanner properties, it is easy to see that $P_n \in SS(1,\ell)$ for any $\ell \geq 1$ because $P_n$ is a tree. Furthermore $C_n \in SS(n-1,\ell)$ but $C_n \notin SS(n-2,\ell)$ for any $\ell \geq 1$, since $minS_\ell(C_n) = n-1$ for any $\ell \geq 1$ (i.e., the failure of one edge results in a path of length $n-1$).

Starting from the above observations, we are interested in finding non-trivial parameters such that a graph is a $(k,\ell)$–self-spanner. This includes the problem of deciding for given parameters $k$ and $\ell$ whether a given graph belongs to $SS(k,\ell)$ as well as the more general recognition problems where we fix one of the parameters and try to optimize the other. This brings up the following optimization, resp. characterization problems:

*Problem 1.* Minimum $\ell$–Stretch-Factor
*Given:* A graph $G$ and an integer $k \geq 1$.
*Problem:* Does $G$ belong to $SS(k,\ell)$, i.e., $minS_\ell(G) \leq k$?

*Problem 2.* Maximum $k$–Fault-Tolerance
*Given:* A graph $G$ and an integer $\ell \geq 0$.
*Problem:* Does $G$ belong to $SS(k,\ell)$, i.e., $maxT_k(G) \geq \ell$?

*Problem 3.* General Self-Spanner
*Given:* A graph $G$ and two integers $k \geq 2, \ell \geq 1$.
*Problem:* Does $G$ belong to $SS(k,\ell)$?

Thus, in Problem 1 we consider $\ell$ as a fixed parameter for the problem whereas in Problem 2 $k$ is a fixed parameter. We now turn to analyzing the complexity of the problems mentioned above. Let us first consider the special case where we allow for single edge failures only, i.e., $\ell = 1$.

**Lemma 4.** *$G \in SS(k,1)$ if and only if every edge of $G$ is either a bridge or belongs to an induced cycle of length at most $k+1$.*

*Proof.* ($\Leftarrow$) Let $e$ be an arbitrary edge of $G$ and consider $G' = G - e$. We have to show property (*) of the remark after Definition 1. If $e$ is a bridge in $G$ then $G'$ is disconnected and there is nothing to show. If $e$ is not a bridge then $G'$ remains connected and by assumption $G'$ is a $k$–spanner of $G$.
($\Rightarrow$) Let us assume $G \in SS(k,1)$, and there is an edge $e = \{u,v\}$ that is not a bridge and that does not belong to an induced cycle of length at most $k+1$. Consider $G' = G - e$. Then $d_{G'}(u,v) > k$, a contradiction. □

Considering multiple edge failures, it is clear that bridges again do not contribute to the stretch factor. But unfortunately we cannot extend the characterization in a straightforward way. If we restrict ourselves to $(\ell+1)$–edge–connected graphs we get the following lemma:

**Lemma 5.** *Let $G = (V, E)$ be $(\ell+1)$–edge–connected. Then $G \in SS(k, \ell)$ if and only if for every edge $e = \{u, v\}$ of $G$ there are at least $\ell$ edge disjoint paths (not involving $e$) of length at most $k$ connecting $u$ and $v$.*

*Proof.* ($\Leftarrow$) Let $G' = (V, E')$ be a subgraph with $E' \subseteq E$ and $|E'| \geq |E| - \ell$, and $e = \{u, v\} \in E \backslash E'$. There are $\ell$ edge disjoint paths (not involving $e$) of length at most $k$ connecting $u$ and $v$. Thus, even if $\ell - 1$ edge failures happen to appear in one of these paths each, at least one covering path for $e$ in $G'$ remains.
($\Rightarrow$) By contradiction, let us assume $G \in SS(k, \ell)$, and there is an edge $e = \{u, v\}$ such that there are at most $j < \ell$ edge disjoint paths (not involving $e$) of length at most $k$ connecting $u$ and $v$. Since $j$ is maximal, there are $j$ edges within the edge disjoint paths such that the following holds: the subgraph $G'$ constructed by deleting $e$ and these selected edges remains connected (since $G$ is $(\ell + 1)$–edge–connected) but $d_{G'}(u, v) > k$, a contradiction to $G \in SS(k, \ell)$. □

Observe that we cannot relax on the edge–connectivity constraint in this lemma. Consider for example the diamond consisting of a $C_4$ and one chord: this graph belongs to $SS(3, 2)$ but it does not fulfill the constraints of Lemma 5.

Now, if we fix the fault-tolerance value $\ell$, we can determine the smallest stretch factor of a given graph in polynomial time (e.g., even by exhaustive search):

**Theorem 1.** Minimum $\ell$–Stretch-Factor *is in $\mathcal{P}$ for all $\ell \geq 0$.*

As a consequence of this theorem we also have:

**Corollary 1.** *The problem of deciding whether a graph is a $(k, \ell)$–self-spanner for fixed $k \geq 1$ and $\ell \geq 0$ is in $\mathcal{P}$.*

Now, we consider the dual problem where we fix the stretch factor and we want to find the largest fault-tolerance value of a given graph. As stated in Lemma 5, to solve this, it is crucial to find edge disjoint paths between any two vertices such that the paths have bounded length. Unfortunately, this problem is hard (cf. [7], (ND41)):

*Problem 4.* Maximum Length-Bounded Disjoint Paths
*Given:* A graph $G$, two vertices $s$ and $t$, positive integers $K, L \leq n$.
*Problem:* Does $G$ contain $L$ or more mutually edge disjoint paths from $s$ to $t$, which all have length at most $K$?

As shown in [10], the problem is $\mathcal{NP}$-complete for all fixed $K \geq 5$; it is polynomially solvable for $K \leq 3$, and it is open for $K = 4$. The proofs given there work for $(\ell + 1)$–edge–connected graphs as well. Observe that the problem of finding the maximum number of edge disjoint paths from $s$ to $t$, under *no length constraint*, is solvable in polynomial time by standard network flow techniques. But this does not help here, since we need to guarantee the length constraints. Together with the observation that we can decide in polynomial time whether a given graph is $(\ell + 1)$–edge–connected, we get the following results, and only Maximum 4–Fault-Tolerance remains to be settled.

**Theorem 2.**

1. MAXIMUM $k$–FAULT-TOLERANCE *is* $\mathcal{NP}$*-complete for all* $k \geq 5$.
2. MAXIMUM $k$–FAULT-TOLERANCE *is in* $\mathcal{P}$ *for* $k = 1, 2, 3$.
3. GENERAL SELF-SPANNER *is* $\mathcal{NP}$*-complete.*

### 3.2 Graph Operations for Constructing $(k, \ell)$–Self-Spanners

Here we are interested in structural aspects of the class of $(k, \ell)$–self-spanners. In particular, we want to find (easy) operations that allow for efficient construction of self-spanner networks or easy recognition of special cases. A common operation is the *Cartesian* (or *cross*) *product* [11], used to define several well-known network topologies.

The next lemma shows that graphs that arise from the Cartesian product of two graphs have strong self-spanner properties. In particular, it indicates that a stretch factor of 3 plays an important role.

**Theorem 3.** *Let* $G_1, G_2$ *be two connected graphs,* $G = G_1 \times G_2$*, and* $i = 1, 2$.

1. *If* $G_i \in SS(k_i, \ell_i)$ *and* $G_i$ *is* $(\ell_i + 1)$*–edge–connected then* $G \in SS(\max\{k_1, k_2\}, \min\{\ell_1, \ell_2\})$.
2. *Let* $\delta$ *be the minimum vertex degree of both* $G_1$ *and* $G_2$*. Then* $G \in SS(3, \delta)$ *(in particular,* $G \in SS(3, 1)$*).*
3. $G \in SS(2, \ell)$ *if and only if every edge in* $G_i$ *belongs to at least* $\ell$ *disjoint triangles within* $G_i$.
4. *If* $G_1$ *or* $G_2$ *contains a bridge then* $maxT_2(G) = 0$*, i.e., there is no* $\ell > 0$ *such that* $G \in SS(2, \ell)$*. In particular, if* $G_1$ *or* $G_2$ *contains a bridge and* $G \in SS(k, \ell)$ *for some* $\ell > 0$ *then* $k \geq 3$.

Observe that, for Part 1 of the previous theorem, it is really necessary to claim the respective edge connectivity. Otherwise we cannot guarantee that the graph considered in the proof remains connected. Also, for Part 3 of that lemma, it does not suffice to claim that $G_1 \in SS(2, \ell)$ (and $G_2 \in SS(2, \ell)$, resp.): we again need that both graphs are $(\ell + 1)$–edge–connected. For smaller stretch factors, i.e., $k = 1$, we already know that $G_1 \times G_2$ has a stretch factor smaller than 2 if and only if it is a tree.

*Remark 4.* Part 2 of Theorem 3 is tight in the following sense: If $G_i \notin SS(2, 1)$ and $G_i$ has minimum vertex degree $\delta$ for $i \in \{1, 2\}$, then $minS_\delta(G_1 \times G_2) = 3$ and $maxT_3(G_1 \times G_2) = \delta$. Thus $G_1 \times G_2 \in SS(3, \delta)$, but $G_1 \times G_2 \notin SS(2, \delta)$ and $G_1 \times G_2 \notin SS(3, \delta + 1)$.

### 3.3 Self-Spanner Properties of Some Popular Network Topologies

As we have seen in the previous subsection, we can construct graphs that exhibit certain self-spanner properties by using the Cartesian product. We now follow the opposite approach and examine some network topologies that are used widely, with respect to their self-spanner properties. In particular, we consider mesh-like

networks like *grid*,*torus*, and *hypercube*. As examples for bounded-degree approximations of the hypercube, we investigate *cube connected cycles* and *butterfly* networks (see [11] and the references therein).

*Grids, tori, and hypercubes.* The topologies that are easiest in structure are the mesh-like networks, which can be constructed by application of the Cartesian product: The hypercube $H_d$ is recursively defined from $P_2$: $H_1 = P_2$, and $H_d = P_2 \times H_{d-1}$ for each $d \geq 2$; the grid $G_{n,m}$ is the Cartesian product $P_n \times P_m$ for $n, m \geq 2$; the torus $T_{n,m}$ is the Cartesian product $C_n \times C_m$ for $n, m \geq 3$.

The following lemma indicates the self-spanner properties of these topologies. Observe that the fault-tolerance value of the torus is higher than that of the grid due to the additional wrap-around connections, which make the topology symmetric. But it is clear from Remark 3, that the addition of edges does not result in higher fault-tolerance values in general.

**Theorem 4.**

1. *$G_{n,m}$ belongs to $SS(3,1)$, but not to $SS(2,1)$.*
   *If $n > 2$ or $m > 2$ then $G_{n,m}$ does not belong to $SS(3,2)$.*
   *If $n, m > 2$ then $G_{n,m}$ belongs to $SS(5,2)$, but not to $SS(4,2)$ or $SS(5,3)$.*
2. *$T_{n,m}$ belongs to $SS(3,2)$, but not to $SS(2,2)$.*
   *If $n > 3$ or $m > 3$ then $T_{n,m}$ does not belong to $SS(3,3)$.*
   *$T_{n,m}$ belongs to $SS(\min\{5, \max\{n,m\} - 1\}, 3)$.*
   *If $n, m \geq 5$ then $T_{n,m}$ belongs to $SS(5,4)$, but not to $SS(4,4)$.*
   *If $n, m > 5$ then $T_{n,m}$ does not belong to $SS(5,5)$.*
3. *$H_d$ belongs to $SS(3, d-1)$, but not to $SS(3,d)$ or to $SS(2,1)$.*

*Hypercube derived networks.* We now consider two different types of bounded-degree approximations of the hypercube.

The *cube-connected cycles graph* of dimension $d$, denoted $CCC_d$, is derived from $H_d$ by replacing each vertex of $H_d$ by a *fundamental cycle* of length $d$. Each vertex of such a cycle is labeled by a tuple $(i, x), 0 \leq i \leq d-1$, and $i$ is called the *level* of the vertex. Apart from the *cycle edges* of the fundamental cycles, each vertex $(i, x)$ is connected to vertex $(i, x(i))$, where $x(i)$ denotes the vertex of $H_d$ that is labeled by the same string as vertex $x$ but with bit $i$ flipped. These edges are called *hypercube edges*.

The *butterfly graph* (with wrap-around) of dimension $d$, denoted $B_d$, is derived similarly from $H_d$: $B_d$ consists of the same vertices $(i, x), 0 \leq i \leq d-1$, as $CCC_d$ and the same *fundamental cycles* of length $d$. But now each vertex $(i, x)$ is connected by two *hypercube edges* to vertices $(i+1, x(i))$ and $(i-1, x(i-1))$.

$CCC_d$ can be obtained from $B_d$ by using a single edge $\{(i, x), (i, x(i))\}$ instead of the pair of edges $\{(i, x), (i+1, x(i))\}$ and $\{(i, x), (i-1, x(i-1))\}$. Thus, $CCC_d$ can be viewed as a spanning subgraph of $B_d$. In [2], it is shown that different hypercube-derived topologies can be embedded within other such topologies with small slowdown. Results on the existence of cycles and the construction of $k$–spanners can be found in [14] and [9], respectively. But all these results do not imply on the self-spanner properties of the topologies studied here. We get the following results concerning the self-spanner properties of the topologies above:

**Theorem 5.**

1. *$B_d$ belongs to $SS(3,1)$ and to $SS(d+1,2)$, but not to $SS(2,1)$, $SS(d,2)$, or $SS(d+1,3)$.*
2. *$CCC_d$ belongs to $SS(7,1)$ and to $SS(\max\{7,d-1\},2)$, but not to $SS(6,1)$.*

The previous lemma shows that bounded-degree approximations of the hypercube like $CCC_d$ and $B_d$ perform poorly with respect to their self-spanner properties: In the case of single edge failures the stretch factor is still a constant (though much larger than for the hypercube), but for double edge failures the stretch factor grows linearly with the dimension $d$. Thus, the guarantees for delays in case of failures are really weak for these kinds of topologies.

## References


1. F. S. Annexstein. Fault tolerance in hypercube-derivative networks. *Computer Architecture News*, 19(1):25–34, 1991.
2. F. S. Annexstein, M. Baumslag, and A. L. Rosenberg. Group action graphs and parallel architectures. *SIAM J. Comput.*, 19(3):544–569, 1990.
3. J. Bruck, R. Cypher, and C.-T. Ho. Fault-tolerant meshes with small degree. *SIAM J. Comput.*, 26(6):1764–1784, 1997.
4. S. Cicerone and G. Di Stefano. Graphs with bounded induced distance. In *Proceedings 24th International Workshop on Graph-Theoretic Concepts in Computer Science, WG'98*, pages 177–191. Springer-Verlag, Lecture Notes in Computer Science, vol. 1517, 1998.
5. S. Cicerone, G. Di Stefano, and D. Handke. Survivable Networks with Bounded Delays. Technical Report, Konstanzer Schriften in Mathematik und Informatik n. 81, University of Konstanz, Germany, February 1999.
6. R. Cole, B. M. Maggs, and R. K. Sitaraman. Reconfiguring arrays with faults part I: Worst-case faults. *SIAM J. Comput.*, 26(6):1581–1611, 1997.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$–Completeness.* W H Freeman & Co Ltd, 1979.
8. D. Handke. Independent tree spanners: Fault-tolerant spanning trees with constant distance guarantees (extended abstract). In *Proceedings 24th International Workshop on Graph-Theoretic Concepts in Computer Science, WG'98*, pages 203–214. Springer-Verlag, Lecture Notes in Computer Science, vol. 1517, 1998.
9. M.-C. Heydemann, J. G. Peters, and D. Sotteau. Spanners of hypercube-derived networks. *SIAM J. on Discr. Math.*, 9(1):37–54, 1996.
10. A. Itai, Y. Perl, and Y. Shiloach. The complexity of finding maximum disjoint paths with length constraints. *Networks*, 12:277–286, 1982.
11. F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes.* Morgan Kaufman Publishers, 1992.
12. F. T. Leighton, B. M. Maggs, and R. K. Sitaraman. On the fault tolerance of some popular bounded-degree networks. *SIAM J. Comput.*, 27(6):1303–1333, 1998.
13. D. Peleg and A. A. Schäffer. Graph spanners. *J. of Graph Theory*, 13:99–116, 1989.
14. A. L. Rosenberg. Cycles in networks. Technical Report UM-CS-1991-020, Dept. of Computer and Information Science, Univ. of Massachusetts, 1991.
15. T.-Y. Sung, M.-Y. Lin, and T.-Y. Ho. Multiple-edge-fault tolerance with respect to hypercubes. *IEEE Trans. Parallel and Distributed Systems*, 8(2):187–192, 1997.

# Energy-Efficient Initialization Protocols for Ad-hoc Radio Networks*

J.L. Bordim[1], J. Cui[1], T. Hayashi[1], K. Nakano[1], and S. Olariu[2]

[1] Department of Electrical and Computer Engineering,
Nagoya Institute of Technology, Showa-ku, Nagoya 466-8555, Japan,
nakano@elcom.nitech.ac.jp
[2] Department of Computer Science, Old Dominion University,
Norfolk, VA 23529, USA,
olariu@cs.odu.edu

**Abstract.** The main contribution of this work is to propose energy-efficient randomized initialization protocols for ad-hoc radio networks (ARN, for short). First, we show that if the number $n$ of stations is known beforehand, the single-channel ARN can be initialized by a protocol that terminates, with high probability, in $O(n)$ time slots with no station being awake for more than $O(\log n)$ time slots. We then go on to address the case where the number $n$ of stations in the ARN is not known beforehand. We begin by discussing, an elegant protocol that provides a tight approximation of $n$. Interestingly, this protocol terminates, with high probability, in $O((\log n)^2)$ time slots and no station has to be awake for more than $O(\log n)$ time slots. We use this protocol to design an energy-efficient initialization protocol that terminates, with high probability, in $O(n)$ time slots with no station being awake for more than $O(\log n)$ time slots. Finally, we design an energy-efficient initialization protocol for the -channel ARN that terminates, with high probability, in $O(\frac{n}{k} + \log n)$ time slots, with no station being awake for more than $O(\log n)$ time slots.

## 1 Introduction

An ad-hoc radio network (ARN, for short) is a distributed system with no central arbiter, consisting of $n$ radio transceivers, henceforth referred to as *stations*. We assume that the stations are identical and cannot be distinguished by serial or manufacturing number. We refer the reader to Figure 1 depicting a 7-station ARN.

As customary, time is assumed to be slotted and all the stations have a local clock that keeps synchronous time, perhaps by interfacing with a GPS system. The stations are assumed to have the computing power of a usual laptop

* Work supported in part by ONR grant N00014-91-1-0526 and by Grant-in-Aid for Scientific Research (C) (10680351) from Ministry of Education, Science, Sports, and Culture of Japan.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 215–224, 1999.
© Springer-Verlag Berlin Heidelberg 1999

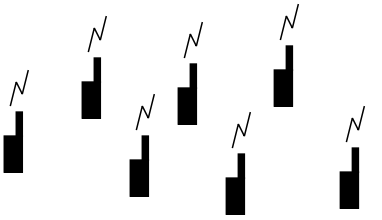


**Fig. 1.** *Illustrating a 7-station ARN.*

computer; in particular, they all run the same protocol and can generate random bits that provide local data on which the stations may perform computations.

The stations communicate using $k$ radio frequencies (i.e. channels). We assume that in any time slot, a station can tune to one radio channel and/or transmit on at most one, possibly the same, channel. A transmission involves a data packet whose length is such that the transmission can be completed within one time slot.

We employ the commonly-accepted assumption that when two or more stations are transmitting on a channel in the same time slot, the corresponding packets *collide* and are lost. We further assume that the system has *collision detection* capability. Accordingly, at the end of a time slot the status of a radio channel is:
*NULL:* if no station transmitted on the channel in the current time slot,
*SINGLE:* if exactly one station transmitted on the channel in the current time slot, and
*COLLISION:* if two or more stations transmitted on the channel in the current time slot.

We assume that the stations in the ARN run on batteries and saving battery power is exceedingly important as recharging batteries may not be possible while in operation. It is well known that a station expends power while its transceiver is active, that is, while transmitting or receiving a packet. It is perhaps surprising at first that a station expends power even if it receives a packet that is not destined for it. Consequently, we are interested in developing protocols that allow stations to *power off* their transceiver to the largest extent possible. Accordingly, we judge the goodness of a protocol by the following two yardsticks:

- the overall number of time slots required by the protocol to terminate
- for each individual station the total number of time slots when it has to transmit/receive a packet.

The *initialization* problem is to assign to each of the $n$ stations in the ARN an integer ID number in the range $[1..n]$ such that no two stations are assigned the same ID. The initialization problem is fundamental in both network design and in multiprocessor systems [3,6].

Recent advances in wireless communications and mobile computing have exacerbated the need for efficient protocols for ARNs. Indeed, a large number of such protocols have been reported in the literature [1,2,5,7,8]. However, many of these protocols function under the assumption that the $n$ stations in the ARN have been initialized in advance. The highly non-trivial task of assigning the stations distinct ID numbers, i.e. initializing the stations, is often ignored in the literature. It is, therefore, of importance to design efficient initialization protocols for ARNs both in the case where the system has a collision detection capability and for the case where this capability is not present.

Recently, Hayashi et al. [3] have reported an initialization protocol for an $n$-station, $k$-channel ARN that terminates with high probability in $O(\frac{n}{k})$ time slots, provided that $k \leq \frac{n}{3 \log n}$. However, the protocol of [3] was not designed to be energy-efficient and the stations have to be awake for the entire duration of the protocol.

Our first main contribution is to propose energy-efficient randomized initialization protocols for ARNs. First, we show that if the number $n$ of stations is known beforehand, the single-channel ARN can be initialized by a protocol terminating, with high probability, in $O(n)$ time slots, with no station being awake for more than $O(\log n)$ time slots.

Next, we consider the case where the number $n$ of stations is not known beforehand. The key insight that leads to an energy-efficient solution in this scenario is an elegant strategy for finding a good approximation for $n$. Specifically, we propose an energy-efficient approximation protocol for $n$ that terminates, with high probability, in $O((\log n)^2)$ time slots with no station being awake for more than $O(\log n)$ time slots. Using our approximation protocol, we show that even if $n$ is not known the single-channel ARN can be initialized by a protocol terminating, with high probability, in $O(n)$ time slots, with no station being awake for more than $O(\log n)$ time slots.

Finally, we extend this protocol to the $k$-channel ARN. Specifically, we propose an energy-efficient initialization protocol for an $n$-station, $k$-channel ARN that terminates, with high probability in $O(\frac{n}{k} + \log n)$ time slots with no station being awake for more than $O(\log n)$ time slots.

## 2 Basics

The main goal of this section is to review a number of fundamental results in basic probability theory and to discuss simple prefix sums protocols that will be needed in the remainder of the paper.

### 2.1 A Refresher of Basic Probability Theory

Throughout, $\Pr[A]$ will denote the probability of event $A$. For a random variable $X$, $E[X]$ denotes the expected value of $X$. Let $X$ be a random variable denoting the number of successes in $n$ independent Bernoulli trials with parameters $p$ and

$1-p$. It is well known that $X$ has a *binomial distribution* and that for every $r$, $(0 \leq r \leq n)$,

$$\Pr[X = r] \;=\; \binom{n}{r} p^r (1-p)^{n-r}. \tag{1}$$

Further, the expected value of $X$ is given by

$$E[X] = \sum_{r=0}^{n} r \cdot \Pr[X = r] = np. \tag{2}$$

To analyze the tail of the binomial distribution, we shall make use of the following estimate, commonly referred to as *Chernoff bound* [4]:

$$\Pr[X > (1+\delta)E[X]] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{E[X]} \quad (0 \leq \delta). \tag{3}$$

We also use the following estimates easily derived from (3):

$$\Pr[X \leq (1-\epsilon)E[X]] \leq e^{-\frac{\epsilon^2}{2}E[X]} \quad (0 \leq \epsilon \leq 1). \tag{4}$$

$$\Pr[X \geq (1+\epsilon)E[X]] \leq e^{-\frac{\epsilon^2}{3}E[X]} \quad (0 \leq \epsilon \leq 1). \tag{5}$$

### 2.2 Prefix Sums Protocols

Suppose that the ARN has $n$ stations each of which has a unique ID in the range $[1..m]$ $(m \geq n)$. Let $S_i$ denote a station with ID $i$ $(1 \leq i \leq m)$. Note that for some $i$, $S_i$ may not exist. Suppose that each station $S_i$ has a real number $x_i$. The *prefix sum* of $x_i$ is the sum of the real numbers with indices no more than $i$, that is,

$$\sum\{x_j \mid 1 \leq j \leq i \text{ and } S_j \text{ exists}\}.$$

The *prefix sums problem* asks to compute all prefix sums. A naive protocol can solve the prefix sums problem in $m-1$ time slots in the one-channel ARN as follows: In each time slot $i$ $(1 \leq i \leq m-1)$, station $S_i$ transmits $x_i$ on the channel, and every station $S_j$ $(i < j \leq n)$ monitors the channel. By summing the real numbers received, each station learns the prefix sum. However, this protocol is not energy efficient, because the last station $S_m$ monitors the channel in all of the $m-1$ time slots.

When the protocol terminates the following three conditions are satisfied:

(ps1) Every active station $S_i$, $(1 \leq i \leq n)$, stores its prefix sum.
(ps2) The last station, that is, station $S_k$ such that no station $S_i$ with $i > k$ exists, has been identified
(ps3) The protocol takes $2m-2$ time slots and no station is awake for more than $2 \log m$ time slots.

If $m = 1$, then $S_1$ knows $x_1$ and the above conditions are verified. Now, assume that $m \geq 2$. Partition the $n$ stations into two groups $\mathcal{P}_1 = \{S_i \mid 1 \leq i \leq \frac{m}{2}\}$ and $\mathcal{P}_2 = \{S_i \mid \frac{m}{2} + 1 \leq i \leq m\}$. Recursively compute the prefix sums in $\mathcal{P}_1$ and $\mathcal{P}_2$. By the induction hypothesis, conditions (ps1)–(ps3) above are satisfied and, therefore, each of the two subproblems can be solved in $m - 2$ time slots, with no station being awake for more than $2 \log m - 2$ time slots. Let $S_j$ and $S_k$ be the last active stations stations in $\mathcal{P}_1$ and in $\mathcal{P}_2$, respectively. In the next time slot, station $S_j$ transmits the sum $\sum\{x_i \mid 1 \leq i \leq j \text{ and } S_i \text{ exists}\}$ on the channel. Every station in $\mathcal{P}_2$ monitors the channel and updates the value of its prefix sum. In one additional time slot station $S_k$ reveals its identity. The reader should have no difficulty to confirm that the protocol satisfies (ps1)–(ps3) above are satisfied. Thus, we have,

**Lemma 1.** *The prefix sums problem on the single-channel ARN of $n$ stations with unique ID $[1..m]$ $(m \geq n)$ can be solved in $2m-2$ time slots with no station being awake for more than $2 \log m$ time slots.*

Next we extend the single-channel prefix sums protocol to a $k$-channel ARN. We begin by partitioning the stations into $k$ equal-sized subsets $X_1, X_2, \ldots, X_k$ such that $X_i = \{S_j \mid (i-1) \cdot \frac{m}{k} + 1 \leq j \leq i \cdot \frac{m}{k} \text{ and } S_j \text{ exists}\}$. By assigning one channel to each subsequence, the prefix sums within each $X_i$ can be computed using the single-channel prefix sums protocol discussed above. This needs $2\frac{m}{k} - 2$ time slots with no station being awake for more than $2 \log \frac{m}{k}$ time slots. At this point, we have the local sum $\text{sum}(X_i)$ of each $X_i$ and we need to compute the prefix sums of $\text{sum}(X_1), \text{sum}(X_2), \ldots, \text{sum}(X_k)$. This can be done by modifying slightly the single-channel prefix-sums protocol. Recall that in the single-channel protocol, the prefix sums of $\mathcal{P}_1$ is computed recursively, and then that for $\mathcal{P}_2$ is computed recursively. Since $k$ channels are available, the prefix sums of $\mathcal{P}_1$ and $\mathcal{P}_2$ are computed simultaneously using $\frac{k}{2}$ channels each. After that, the overall solution can be obtained in two more time slots. Using this idea, the prefix sums can be computed in $2\frac{m}{k} + 2 \log k - 2$ with no station being awake for more than $2 \log \frac{m}{k} + 2 \log k = 2 \log m$ time slots. To summarize, we have proved the following result.

**Lemma 2.** *The prefix sums problem on the $k$-channel ARN with $n$ stations each having a unique ID $[1..m]$ $(m \geq n)$ can be solved in $2\frac{m}{k} + 2 \log k - 2$ time slots with no station being awake for more than $2 \log m$ time slots.*

## 3 An Energy-Efficient Initialization for Known $n$

Suppose that each station knows the number $n$ of stations. The purpose of this section is to show that with probability exceeding $1 - O(n^{-2.67})$ the $n$-station ARN can be initialized in $O(n)$ time slots with no station being awake for more than $O(\log n)$ time slots.

The protocol uses the initializing protocol discussed in [3] which was not designed under energy efficiency in mind.

**Lemma 3.** *Even if $n$ is not known beforehand, an $n$-station, single-channel ARN can be initialized in at most $16n$ time slots with probability at least $1 - 2^{-1.62n}$.*

In paper [3], the constant factor of the time slots is not evaluated explicitly.

The details of our energy efficient initialization protocol are spelled out as follows.

---

Protocol `Initialization`

Step 1 Each station selects uniformly at random an integer $i$ in the range $[1..\frac{n}{\log n}]$. Let $h(i)$ denote the set of stations that have selected integer $i$;

Step 2 Initialize each set $h(i)$ individually in $64 \log n$ time slots;

Step 3 Let $N_i$ denote the number of stations in $h(i)$. By computing the prefix sums of $N_1, N_2, \ldots, N_{\frac{n}{\log n}}$ every station determines, in the obvious way, its global ID within the ARN.

---

Clearly, Step 1 needs no transmission.

Step 2 can be performed in $64n$ time slots using the protocol for Lemma 3 as follows: the stations in group $h(i)$, $(1 \leq i \leq \frac{n}{\log n})$, are awake for $64 \log n$ time slots from time slot $(i-1) \cdot 64 \log n + 1$ to time slot $i \cdot 64 \log n$. Outside of this number of slots all the stations in group $h(i)$ are asleep and consume no power.

As we are going to show, with high probability, every group $h(i)$ contains at most $4 \log n$ stations. To see that this is the case, observe that the expected number of stations in $h(i)$ is $E[|h(i)|] = n \times \frac{\log n}{n} = \log n$. Now, using the Chernoff bound in (3), we can write

$$\begin{aligned}
\Pr[|h(i)| > 4 \log n] &= \Pr[|h(i)| > (1+3)E[|h(i)|]] \\
&< (\frac{e^3}{4^4})^{E[|h(i)|]} \quad \text{(by (3) with } \delta = 3) \\
&< n^{-3.67} \quad (\text{since } \log \tfrac{e^3}{4^4} = -3.67...)
\end{aligned}$$

It follows that the probability that group $h(i)$ contains more than $4 \log n$ stations is bounded above by $n^{-3.67}$. Hence, with probability exceeding $1 - n \cdot n^{-3.67} = 1 - n^{-2.67}$, none of the groups $h(1), h(2), \ldots, h(\frac{n}{\log n})$ contains more than $4 \log n$ stations. By Lemma 3, with probability exceeding $1 - 2^{-1.62 \times 4 \log n} = 1 - n^{-6.48}$ the stations in group $h(i)$ can be initialized in $16 \times 4 \log n = 64 \log n$ time slots. Thus, with probability exceeding $1 - n^{-5.48} - n^{-2.67} > 1 - O(n^{-2.67})$ all the groups $h(i)$ will be initialized individually in $64 \log n \times \frac{n}{\log n} = 64n$ time slots, with no station being awake for more than $64 \log n$ time slots.

Let $P_i$, $(1 \leq i \leq \frac{n}{\log n})$, denote the last station in group $h(i)$. At the end of Step 2, each station $P_i$ knows the number $N_i$ of stations in $h(i)$. Step 3 can use the stations $P_i$ to compute the prefix sums of $N_1, N_2, \ldots, N_{\frac{n}{\log n}}$. The prefix sums protocol discussed in Section 2 will terminate in $2\frac{n}{\log n} - 2 < 2n$ time slots with no station being awake for more than $2 \log \frac{n}{\log n} < 2 \log n$ time slots. To summarize, we have proved the following result.

**Theorem 4.** *If the number $n$ of stations is known beforehand, with probability exceeding $1 - O(n^{-2.67})$, an $n$-station, single-channel ARN can be initialized in $O(n)$ time slots, with no station being awake for more than $O(\log n)$ time slots.*

## 4 Finding a Good Approximation of $n$

At the heart of our energy-efficient initialization protocol of an ARN where the number $n$ of stations is not known beforehand lies a simple and elegant approximation protocol for $n$. Specifically, this protocol returns an integer $I$ satisfying, with probability at least $1 - O(n^{-1.83})$, the double inequality:

$$\log n - \log\log n - 4 < I < \log n - \log\log n + 1. \tag{6}$$

Once a good approximation of $n$ is available, we can use the protocol discussed in Section 3 to initialize the entire ARN.

The details of our approximation protocol is spelled out as follows:

---

Protocol `Approximation`
Each station generates uniformly at random a real number $x$ in $(0, 1]$
Let $g(i)$, $(i \geq 1)$, denote the group of stations for which $\frac{1}{2^i} < x \leq \frac{1}{2^{i-1}}$.
**for** $i \leftarrow 1$ **to** $\infty$ **do**
  protocol `Initialization` is run on group $g(i)$ for $128i$ time slots;
  **if** the initialization is complete and if $g(i)$ contains at most $8i$ stations
  **then** the first station in $g(i)$ transmits an "exit" signal.
**end for**

---

Let us first evaluate the number of time slots it takes protocol `Approximation` to terminate. Let $I$ be the value of $i$ when the **for**-loop is exited. One iteration of the for-loop takes $128i + 1$ time slots. Thus, the total number of time slots is at most

$$\sum_{i=1}^{I} 128i + 1 < 64(I + 1)^2.$$

Next, we evaluate for each station the maximum number of time slots during which it has to be awake. Clearly, each station belongs to exactly one of the groups $g(1), g(2), \ldots$, and therefore, every station is awake for $128i \leq 128I$ time slots. Further, all the stations must monitor the channel to check for the "exit" signal. Of course, this takes an additional $I$ time slots. Thus, no station needs to be awake for more than $129I$ time slots.

Our next task is to show that, with high probability, $I$ satisfies (6). For this purpose we rely on the following technical results.

**Lemma 5.** *If $i$ satisfies $1 \leq i \leq \log n - \log\log n - 4$ then $\Pr[|g(i)| > 8i] > 1 - n^{-2.88}$.*

*Proof.* Clearly, $i \leq \log n - \log\log n - 4$ implies that

$$2^i \leq \frac{n}{16\log n} \text{ and similarly } 16i < 16\log n \leq \frac{n}{2^i}.$$

Since $g(i)$ consists of stations generating a real number $x$ satisfying $\frac{1}{2^i} < x \leq \frac{1}{2^{i-1}}$ the expected number of stations is $E[|g(i)|] = \frac{n}{2^i}$. Using the Chernoff bound in (4) with $\epsilon = \frac{1}{2}$, we can evaluate the probability $\Pr[|g(i)| < 8i]$ that group $g(i)$ contains strictly less than $8i$ stations as follows:

$$\begin{aligned}
\Pr[|g(i)| < 8i] &< \Pr[|g(i)| < (1-\frac{1}{2})\frac{n}{2^i}] \quad (\text{since } 16i < \tfrac{n}{2^i}) \\
&< e^{-(\frac{1}{2})^3 \frac{n}{2^i}} \quad (\text{by (4) with } \epsilon = \tfrac{1}{2}) \\
&< e^{-2\log n} \quad (\text{since } 16\log n \leq \tfrac{n}{2^i}) \\
&< n^{-2.88} \quad (\text{since } \log(e^{-2}) = -2.88\ldots)
\end{aligned}$$

as claimed.

**Lemma 6.** *If* $i = \lfloor \log n - \log\log n \rfloor + 1$ *then* $\Pr[|g(i)| < 8i] > 1 - n^{-1.83}$.

*Proof.* If $i = \lfloor \log n - \log\log n \rfloor + 1$ then clearly,

$$\log n - \log\log n < i \leq \log n - \log\log n + 1.$$

This also implies that

$$\frac{n}{\log n} < 2^i \leq \frac{2n}{\log n} \text{ and that } \frac{i}{2} \leq \frac{\log n}{2} \leq \frac{n}{2^i} < \log n < i + \log\log n < 2i.$$

Using these relations, we can evaluate the probability $\Pr[|g(i)| > 8i]$ that group $g(i)$ contains strictly more than $8i$ stations as follows:

$$\begin{aligned}
\Pr[|g(i)| > 8i] &< \Pr[|g(i)| > 4\frac{n}{2^i}] \quad (\text{since } \tfrac{n}{2^i} < 2i) \\
&< (\frac{e^3}{4^4})^{\frac{n}{2^i}} \quad (\text{from (3) with } \delta = 3) \\
&\leq (\frac{e^3}{4^4})^{\frac{\log n}{2}} \quad (\text{from } \tfrac{\log n}{2} \leq \tfrac{n}{2^i}) \\
&< n^{-1.83}. \quad (\text{from } \log(\tfrac{e^3}{4^4})^{\frac{1}{2}} = -1.83\ldots)
\end{aligned}$$

This completes the proof.

Lemma 5 and 6 combined, yield the following result.

**Lemma 7.** *The value $I$ of $i$ when the* **for***-loop is exited satisfies, with probability at least $1 - O(n^{-1.83})$, condition (6).*

Thus, we have proved the following important result.

**Theorem 8.** *Protocol* `Approximation` *returns an integer $I$ satisfying, with probability at least $1 - O(n^{-1.83})$, condition (6). Moreover, the protocol terminates in $64(\log n)^2$ time slots with no station is awake for more than $129\log n$ time slots.*

## 5 An Energy-Efficient Initialization for Unknown $n$

The main goal of this section is to present an energy-efficient initialization protocol in the case where the number of stations in the ARN is not known.

Recall that protocol `Initialization`, partitions the $n$ stations into $\frac{n}{\log n}$ groups and initializes each group independently. Since $n$ is unknown, this partitioning cannot be done. Instead, using protocol `Approximation`, we find an integer $I$ satisfying condition (6) with probability at least $1 - O(n^{-1.83})$. After that, execute `Initialization` with $2^{I+4}$ groups $h(1), h(2), \ldots, h(2^{I+4})$. Observe that condition (6) is satisfied. Then, we can write

$$\frac{\log n}{32} < \frac{n}{2^{I+4}} < \log n. \tag{7}$$

In this case by Theorem 4, with probability $1 - O(n^{-3.67})$, $64 \log n$ time slots are sufficient to initialize a particular $h(i)$. However, since $n$ is unknown, we cannot arrange $64 \log n$ time slots to each group. Instead, we assign $128(I + 4)$ $(> 64 \log n)$ time slots. As a result, with probability at least $1 - O(2^{I+4} \cdot n^{-3.67}) > 1 - O(n^{-2.67})$, all of the groups can be initialized locally in $2^{I+4} \cdot 128(I+4) = O(n)$ time slots with no station being awake for more than $128(I + 4) = O(\log n)$ time slots. After that, the prefix sums are computed in $2 \cdot 2^{I+4} - 2 = O(\frac{n}{\log n})$ time slots with no station being awake for more than $2 \log(2^{I+4}) = O(\log n)$ time slots. Since condition (6) is satisfied with probability at least $1 - O(n^{-1.83})$, we have

**Theorem 9.** *Even if the number $n$ of stations is not known beforehand, with probability exceeding $1 - O(n^{-1.83})$, an $n$-station, single-channel ARN can be initialized in $O(n)$ time slots, with no station being awake for more than $O(\log n)$ time slots.*

## 6 An Energy-Efficient Initialization for the $k$-Channel ARN and Unknown $n$

The main purpose of this section is to present an energy-efficient initialization protocol for the $n$-station $k$-channel ARN, when $n$ is not known beforehand. Let $\mathrm{C}(1), \mathrm{C}(2), \ldots, \mathrm{C}(k)$ denote the $k$ channels available in the ARN.

The idea is to parallelize the protocol in Section 5 to take advantage of the $k$ channels. Recall that this initialization protocol first executes protocol `Approximation` and then execute protocol `Initialization`.

We first extend protocol `Approximation` for the case where $k$ channels are available. Having determined groups $g(1), g(2), \ldots, g(I)$, we allocate groups to channels. For every $i$, $(1 \leq i \leq k)$, we allocate group $g(i)$ to channel $\mathrm{C}(i)$ and attempt to initialize group $g(i)$ in $128i$ time slots. If none of these attempts is successful, we allocate the next set of $k$ groups $g(k + 1), g(k + 2), \ldots, g(2k)$ to the $k$ channels in a similar fashion. This is then continued, as described, until eventually one of the groups is successfully initialized. We now estimate the

number of time slots required to get the desired value of $I$. Let $c$ be the integer satisfying $ck+1 \leq I \leq (c+1)k$. In other words, in the $c$-th iteration, $I$ is found. If $c \geq 1$, then the total number of time slots is

$$128k + 128(2k) + 128(3k) + \cdots + 128(ck) = O(c^2 k) = O\left(\frac{I^2}{k}\right) = O\left(\frac{(\log n)^2}{k}\right).$$

If $c = 0$, then the total number of time slots is $128(I + 4) = O(\log n)$. Therefore, the $k$ channel version of `Approximation` terminates, with high probability, in $O\left(\frac{(\log n)^2}{k} + \log n\right)$ time slots, with no station being awake for more than $O(\log n)$ time slots.

Similarly, we can extend protocol `Initialization` for the $k$-channel case as follows: Recall that we need to initialize each of the groups $h(1), h(2), \ldots, h(2^{I+4})$. This task can be extended as follows: Since we have $k$ channels, $\frac{2^{I+4}}{k}$ groups can be assigned to each channel and be initialized efficiently. Since each group $h(j)$ can be initialized in $128(I+4)$ time slots, all of the $2^{I+4}$ groups can be initialized in $128(I + 4) \cdot \frac{2^I}{k} = O(\frac{n}{k})$ time slots, with no station being awake no more than $O(\log n)$ time slots. After that we use the $k$-channel version of the prefix sums protocol discussed in Section 2. This takes $2\frac{2^{I+4}}{k} + 2\log k - 2 < \frac{64n}{k \log n} + 2\log k$ time slots, with no station being awake for more than $2\log \frac{2^I}{k} + 2\log k$ time slots. Therefore, we have the following result:

**Theorem 10.** *Even if the number $n$ of stations is unknown, with probability exceeding $1-O(n^{-1.83})$, an $n$-station, $k$-channel ARN can be initialized in $O(\frac{n}{k} + \log n)$ time slots, with no station being awake for more than $O(\log n)$ time slots.*

## References


1. N. Alon, A. Bar-Noy, N. Linial, and D. Peleg, Single-round simulation on radio networks, *Journal of Algorithms*, 13, (1992), 188–210.
2. R. Dechter and L. Kleinrock, Broadcast communication and distributed algorithms, *IEEE Transactions on Computers*, C-35, (1986), 210–219.
3. T. Hayashi, K. Nakano, and S. Olariu, Randomized initialization protocols for Packet Radio Networks, *Proc. 13th International Parallel Processing Symposium*, (1999), 544–548.
4. R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.
5. S. Murthy and J. J. Garcia-Luna-Aceves, A routing protocol for packet radio networks, *Proc. First Annual ACM Conference on Mobile Computing and Networking*, Berkeley, California, 1995, 86–95.
6. K. Nakano, Optimal initializing algorithms for a reconfigurable mesh, *Journal of Parallel and Distributed Computing*, 24, (1995), 218-223.
7. K. Nakano, Optimal sorting algorithms on bus-connected processor arrays, *IEICE Transactions Fundamentals*, E-76A, 11, (1994), 2008–2015.
8. K. Nakano, S. Olariu, and J. L. Schwing, Broadcast-efficient protocols for ad-hoc radio networks with few channels, *IEEE Transactions on parallel and Distributed Systems*, to appear.

# Constructing the Suffix Tree of a Tree with a Large Alphabet

Tetsuo Shibuya

IBM Tokyo Research Laboratory,
1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan.
tshibuya@trl.ibm.co.jp

**Abstract.** The problem of constructing the suffix tree of a common suffix tree (CS-tree) is a generalization of the problem of constructing the suffix tree of a string. It has many applications, such as in minimizing the size of sequential transducers and in tree pattern matching. The best-known algorithm for this problem is Breslauer's $O(n \log |\Sigma|)$ time algorithm where $n$ is the size of the CS-tree and $|\Sigma|$ is the alphabet size, which requires $O(n \log n)$ time if $|\Sigma|$ is large. We improve this bound by giving an $O(n \log \log n)$ algorithm for integer alphabets. For trees called shallow -ary trees, we give an optimal linear time algorithm. We also describe a new data structure, the Bsuffix tree, which enables efficient query for patterns of completely balanced -ary trees from a -ary tree or forest. We also propose an optimal $O(n)$ algorithm for constructing the Bsuffix tree for integer alphabets.

## 1 Introduction

The suffix tree of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S\$$ ($\$ \notin \Sigma$). This is a very fundamental and useful structure in combinatorial pattern matching. Weiner [13] introduced this structure and showed that it can be computed in $O(n|\Sigma|)$ time, where $|\Sigma|$ is the alphabet size. Since then, much work has been done on simplifying algorithms and improving bounds [3,11,12], with algorithms achieving an $O(n \log |\Sigma|)$ computing time (see also [8] for details). Recently, Farach [5] proposed a new algorithm that achieved a linear time (independent from the alphabet size) for integer alphabets.

A common suffix tree, or a CS-tree for short, is a data structure that represents a set of strings. This is also an important problem that appears in tasks such as minimizing sequential transducers of deterministic finite automata [2] and tree pattern matching [10]. Kosaraju [10] mentioned that the generalized suffix tree of all the suffixes of a set of strings represented by a CS-tree can be constructed in $O(n \log n)$ time where $n$ is the size of the CS-tree. Breslauer [2] improved this bound by giving an $O(n \log |\Sigma|)$ algorithm. Note that both of the algorithms were based on Weiner's suffix tree construction algorithm [13]. But this algorithm becomes $O(n \log n)$ when $\Sigma$ is large. In this paper, we improve their bound by giving an $O(n \log \log n)$ algorithm for integer alphabets. Shallow trees are trees such that their depths must be at most $c \log n$, where $n$ is the size

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 225–236, 1999.
© Springer-Verlag Berlin Heidelberg 1999

of the tree and $c$ is some constant. We give an optimal $O(n)$ algorithm for trees called shallow $k$-ary trees, for constant $k$.

We also deal with a new data structure called a Bsuffix tree, which is a generalization of the suffix tree of a string. Using the suffix tree of a CS-tree, we can find a given path in a tree very efficiently. The Bsuffix tree is a data structure that enables us to query any given completely balanced $k$-ary tree pattern from a $k$-ary tree or forest very efficiently. Note that the concept of a Bsuffix tree is very similar to that of an Lsuffix tree [1,7], which enables us to query any square submatrix of a square matrix efficiently. We will show that this data structure can be built in $O(n)$ time for integer alphabets. Bsuffix trees have many useful features in common with ordinary suffix trees. For example, using this data structure, we can find a pattern (a completely balanced $k$-ary tree) in a text $k$-ary tree in $O(m \log m)$ time, where $m$ is the size of the pattern. Moreover, we can enumerate common completely balanced $k$-ary subtrees in a linear time. Considering that general tree pattern matching requires a $O(n \log^3 n)$ time [4], these results mean that a Bsuffix tree is a very useful data structure.

## 2 Preliminaries

### 2.1 The Suffix Tree

The suffix tree of a string $S \in \Sigma^n$ is the compacted trie of all the suffixes of $S\$$ ($\$ \notin \Sigma$). The tree has $n+1$ leaves and each internal node has more than one child. Each edge is labeled with a non-empty substring of $S\$$ and no two edges out of a node can have labels which start with the same character. Each node is labeled with the concatenated string of edge labels on the path from the root to the node, and each leaf has a label that is a different suffix of $S\$$. Because each edge label is represented by the first and the last indices of the corresponding substring in $S\$$, the data structure can be stored in $O(n)$ space. In this paper, we deal with only the suffix trees in which the edges going out from a node are sorted according to their labels. Notice that this property is very convenient for querying substrings.

For this powerful and useful data structure, we have the following theorems:

**Theorem 1 (Farach [5]).** *The suffix tree of a string $S \in \{1, \ldots, n\}^n$ can be constructed in $O(n)$ time.*

Note that alphabet $\{1, \ldots, n\}$ is called an integer alphabet. In this paper, we will deal with only integer alphabets. Farach's suffix tree construction algorithm and our algorithms to be presented use the following theorem:

**Theorem 2 (Harel and Tarjan [9]).** *For any tree with $n$ nodes, we can find the lowest common ancestor of any two nodes in a constant time after $O(n)$ preprocessing if the following values can be obtained in a constant time: bitwise AND, OR, and XOR of two binary numbers, and the positions of the leftmost and rightmost 1-bit in a binary number.*

This theorem indicates that the longest common prefix (LCP) of any two suffixes can be obtained from the suffix tree in a constant time after linear-time preprocessing.

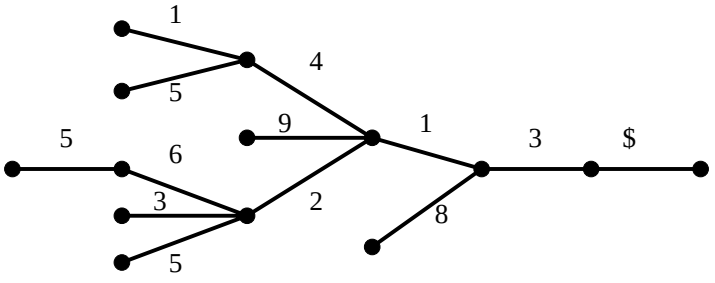



**Fig. 1.** CS-tree of the strings 1413$, 5413$, 913$, 56213$, 3213$, 5213$, and 83$

### 2.2 The Suffix Tree of a Tree

A set of strings $\{S_1, \ldots, S_k\}$, such that no string is a suffix of another, can be represented by a common suffix tree (CS-tree for short), which is defined as follows:

**Definition 1 (CS-tree).** *In the CS-tree of a set of strings $\{S_1, \ldots, S_k\}$, each edge is labeled with a single character, and each node is labeled with the concatenated string of edge labels on the path from the node to the root. In the tree, no two edges out of a node can have the same label. Furthermore, the tree has k leaves, each of which has a different label that is one of the strings, $S_i$.*

Figure 1 shows an example of a CS-tree. The number of nodes in the CS-tree is equal to the number of different suffixes of strings. Thus, the size of a CS-tree is not larger than the sum of the lengths of the strings represented by the CS-tree. Note that the CS-tree can be constructed easily from strings in a time linear to the sum of the lengths of the strings.

The generalized suffix tree of a set of strings $\{S_1, \ldots, S_k\}$ is the compacted trie of all the suffixes of all the strings in the set. As mentioned in [10], the suffix tree of a CS-tree is the same as the generalized suffix tree of the strings represented by the CS-tree. Furthermore, the size of the generalized suffix tree is linear to that of the CS-tree, because the number of leaves of the suffix tree is equal to the number of edges in the CS-tree. Note that the edge labels of the suffix tree of a CS-tree corresponds to a path in the CS-tree, and they can be represented by the pointers to the first edge (nearest to the leaves) and the path length.

Let $n_i$ be the length of $S_i$, and let $N = \sum_i n_i$. Let $n$ be the number of nodes in the CS-tree of the strings. The generalized suffix tree can be obtained in $O(N)$ time in the case of integer alphabets (*i.e.*, $S_i \in \{1, \ldots, n\}^{n_i}$) as follows. First, we construct the suffix tree of a concatenated string of $S_1\$S_2\$\cdots\$S_k$ using Farach's suffix tree construction algorithm. Then, we obtain the generalized suffix tree by cutting away the unwanted edges and nodes. But $N$ is sometimes much larger than the size $n$ of the CS-tree: for example, there exists a tree for which $N$ is $O(n^2)$. This means that the $O(N)$-time suffix tree construction algorithm given above is not at all a linear time algorithm. The best-known $O(n \log |\Sigma|)$

algorithm [2] for this problem is based on Weiner's suffix tree construction algorithm [13]. We will improve it by giving a new algorithm based on Farach's linear-time suffix tree construction algorithm.

## 3 New Algorithm for Constructing the Suffix Tree of a CS-Tree

### 3.1 Algorithm Outline

Our approach to constructing the suffix tree of a CS-tree is based on Farach's suffix tree construction algorithm [5]. Farach's algorithm has three steps. First, it constructs a tree called an odd tree recursively. Next, it constructs another tree called an even tree by using the odd tree. Finally it constructs the suffix tree by merging these two trees. Note that the odd tree is a trie of suffixes $S[2i-1]\dots S[n]\$$, and the even tree is a trie of suffixes $S[2i]\dots S[n]\$$. This algorithm achieves an $O(n)$ computation time for integer alphabets.

We later also define the odd and even trees for the suffix tree of a CS-tree, and our algorithm also has three following similar steps. First we build the odd tree or the even tree recursively, then we construct the even or odd tree by using the odd or even tree, respectively, and finally we merge them to construct the suffix tree.

In the algorithm, we use the following theorems:

**Theorem 3.** *In any tree with $n$ nodes, for any node $v$ in the tree and any integer $d > 0$ that is smaller than the depth of $v$, we can find the ancestor of $v$ whose depth is $d$ in $O(\log\log n)$ time after $O(n)$ preprocessing, if the following values can be obtained in a constant time: OR, shift of any $i$ bits, the number of 1-bits in the binary number, and any $i$th-leftmost 1-bit in the binary number.*

**Theorem 4.** *In any shallow $k$-ary binary tree with $n$ nodes where $k$ is constant, for any node $v$ in the tree and any integer $d > 0$ that is smaller than the depth of $v$, we can find the ancestor of $v$ whose depth is $d$ in a constant time after $O(n)$ preprocessing, if any $i$-bit shift of a binary number can be performed in a constant time.*

Proofs of these theorems are given in the appendix. See section 1 for the definition of a 'shallow tree'. Let $T_{lookup}(n)$ be the time needed to compute a node's ancestor of depth $d$ after $O(n)$ preprocessing.

Let us now define several notations. Let $\{S_1, \dots, S_k\}$ be the strings represented by a given CS-tree. Let $n_i$ be the length of $S_i$ and let $S_i = S_i[n_k]\dots S_i[1]$. Note that the indices are arranged in reverse order. Above theorems 3 and 4 indicates that, for any $i$ and $j$, we can access $S_i[j]$ in $T_{lookup}(n)$ time after $O(n)$ preprocessing. Let $S_i(m)$ be $S_i$'s suffix of length $m$, *i.e.*, $S_i[m]\dots S_i[1]$. Let $lcp(S, S')$ and $lcs(S, S')$ be the lengths of the longest common prefix and suffix of strings $S$ and $S'$, respectively. Let $parent_U(v)$ be the parent node of $v$ in the CS-tree $U$ if $v$ is not the root node $t$; otherwise, let it be $t$: *i.e.*, $parent_U(v_{i,j}) = v_{i,\max(0,j-1)}$. Let $label(e)$ be the label given to edge $e$ in the CS-tree. Let $T_U$ be the suffix tree of the CS-tree $U$.

### 3.2 Building a Half of the Suffix Tree Recursively

All nodes in the CS-tree $U = (V, E)$ have either odd or even label length. Let $V_{odd}$ and $V_{even}$ be the nodes with odd label lengths and those with even label lengths, respectively. If $|V_{odd}| \geq |V_{even}|$, let $V_{small} = V_{even}$ and $V_{large} = V_{odd}$; otherwise, let $V_{small} = V_{odd}$ and $V_{large} = V_{even}$. We can obtain $|V_{odd}|$ and $|V_{even}|$ in $O(n)$ time by the ordinary depth-first search on the CS-tree. Therefore, we can determine in a linear time which node set is $V_{small}$. In this subsection, we will recursively construct the compacted trie $T_{small}$ of all the labels of nodes in $V_{small}$. Note that the technique for constructing $T_{small}$ is very similar to that for constructing the odd tree in Farach's algorithm.

Consider a new CS-tree $U' = (V_{small}, E_{small})$, where $E_{small} = \{(v, parent_{U'} = parent_U(parent_U(v)))|v \in V_{small}, v \neq t\}$ and the edge labels are determined as follows. Radix sort the label pairs $pair(v) = (label((v, parent_U(v))), label((parent_U(v), parent_U(parent_U(v)))))$ for all $v \in V_{small} \backslash t$ and remove duplicates, where $label(e)$ denotes the label of an edge $e$ in the original CS-tree $U$ (let $label(t, t) = \phi \notin \{\Sigma, \}$). Let $rank(v)$ be the rank of $pair(v)$ in the sorted list, which belongs to an integer alphabet $[1, n/2]$ because the size of the new tree $U'$ is not larger than half of that of the original CS-tree $U$. Let $orig_pair(i)$ be a label pair $pair(v)$ such that $rank(v) = i$. Let the label of an edge $(v, parent_{U'}(v)) \in E_{small}$ be $rank(v)$. Notice that all of these procedures can be performed in a linear time.

We then construct the suffix tree $T_{U'}$ of $U'$ by using our algorithm recursively. After that, we construct $T_{small}$ from $T_{U'}$ as follows. We can consider a tree $T'$ whose edge labels of $T_{U'}$ are modified to the original labels in $U$: for example, if the label of an edge in $T_{U'}$ is $ijk$, the label of the corresponding edge in $T'$ is $orig_pair(i), orig_pair(j), orig_pair(k)$. Notice that this modification can be performed by making only a minor modification of the edge label representation and that it takes only linear time.

We can construct $T_{small}$ from $T'$ very easily. $T'$ contains all the labels of nodes in $V_{small}$, but is not the compacted trie: the first characters of labels of outgoing edges from the same node may be the same. But the second character is different, and the edges are sorted lexicographically. Thus we can change $T'$ to $T_{small}$ by making only a minor adjustment: we merge such edges and make a node, and if all the first characters of all the labels of edges are the same, we delete the original node.

In this way, we can construct $T_{small}$ in a $T(n/2) + O(n)$ time, where $T(n)$ is the time our algorithm takes to build the suffix tree of a CS-tree of size $n$.

### 3.3 Building the Other Half of the Tree

In this section, we show how to construct the compacted trie $T_{large}$ of all the labels of nodes in $V_{large}$ from $T_{small}$ in a linear time. The technique is a slightly modified form of the second step of Farach's algorithm, which constructs the even tree from the odd tree.

If we are given an lexicographic traverse of the leaves of the compacted trie (which is called lex-ordering in [5]), and the length of the longest common prefix of adjacent leaves, we can reconstruct the trie [5,6]. Note that it can be done in

linear time in the case of constructing the suffix tree of a string. We will obtain these two parts of $T_{large}$ from $T_{small}$, and construct $T_{large}$ in the same way. But this method can obtain only the label length from the leaf or root for each node of the compacted trie. Recall that each label is represented by the first node and the label length in our case. Thus we must obtain that node from its specified depth and its some descendant leaf, which requires $T_{lookup}(n)$ time. Hence the total time required by this procedure is $O(nT_{lookup}(n))$.

Any leaf in $T_{large}$, except for those with labels of only one character, has a label consisting of a single character followed by the label of some corresponding leaf in $T_{small}$. We can obtain the lex-ordering of the labels of leaves in $T_{small}$ by an in-order traverse of $T_{small}$ which takes only a linear time. Thus we can obtain the lex-ordering of the labels of leaves ($S_i(m)$) in $T_{large}$ by using the radix sorting technique, because we have $S_i[m]$ and the lexicographically sorted list of $S_i(m-1)$.

The longest common prefix length of adjacent leaves of $T_{large}$ can also be obtained easily by using $T_{small}$. Let $S_i(m)$ and $S_j(n)$ be the labels of two adjacent leaves in $T_{large}$. If $S_i[m] \neq S_j[n]$, the longest common prefix length is 0. Otherwise, it is $1+lcp(S_i(m-1), S_j(n-1))$ which can be obtained in a constant time from $T_{small}$ after linear-time preprocessing on $T_{small}$ (see Theorem 2). In this way, we can construct $T_{large}$ from $T_{small}$ in $O(nT_{lookup}(n))$. According to Theorems 3 and 4, it is $O(n \log\log n)$ for general CS-trees, and $O(n)$ for shallow $k$-ary CS-trees ($k$: constant).

### 3.4 Merging the Trees

Now we have two compacted tries $T_{odd}$ and $T_{even}$. In this subsection, we merge $T_{odd}$ and $T_{even}$ to construct the target suffix tree $T_U$. We call the compacted trie of odd/even-length suffixes of strings the generalized odd/even tree of the strings. The odd/even tree of a CS-tree is also the generalized odd/even tree of the strings represented by the CS-tree. Farach's algorithm merges the odd and even trees in a time linear to the sum of the sizes of odd and even trees. It can be directly applied also to our problem of merging generalized odd and even trees. Note that the merging can be done a linear time in Farach's algorithm, but requires $O(nT_{lookup}(n))$ time in our case. The outline of the algorithm is as follows.

First, we merge the even and odd trees as following by considering that one of two edge labels is a prefix of the other label if the first characters of labels of two edges are the same. Let edges $e_1 = (v, v_1)$ and $e_2 = (v, v_2)$ be the edges which starts from the same node $v$ and the same first character. Let $l_1$ and $l_2$ be the label lengths of $e_1$ and $e_2$, respectively. Without loss of generality, we let $l_1 \geq l_2$. Then we construct a internal node $v_1'$ between $v$ and $v_1$ if $l_1 > l_2$, otherwise let $v_1'$ be $v_1$. In case that $l_1 > l_2$, let the label of edge $(v, v_1')$ be the first $l_2$ characters of the label of original edge $(v, v_1)$ and let the label of edge $(v_1', v_1)$ be the last $l_1 - l_2$ characters of the label of original edge $(v, v_1)$. Then we merge two edge $(v, v_1')$ and $e_2$. Note that this merging requires $T_{lookup}(n)$ time because we must find the node which corresponds to the first character of new edge $(v_1', v_1)$. We merge recursively all over the two trees by the normal coupled

depth first search. Thus the total computing time required for the merging is $O(nT_{lookup}(n))$.

Next, we unmerge the edges with different labels because we have merged edges too far. In this unmerging stage, we first compute the longest common prefix length of any merged pair of node labels in the suffix tree. Farach [5] showed that all the required longest common prefix lengths for all the merged pairs can be obtained in $O(n)$ time by using a data structure called d-links; for details of d-links and the algorithm, see [5]. Using the common prefix length of merged nodes, we can easily determine how far to unmerge edges. For each unmerged edge, we must find the node of the CS-tree that corresponds to the first character of its label, which requires $T_{lookup}(n)$ time. Thus the total computing time for unmerging is also $O(nT_{lookup}(n))$.

Hence the step of our algorithm for merging trees takes a total of $O(nT_{lookup}(n))$ time. Thus we obtain an equation $T(n) = T(n/2)+O(nT_{lookup}(n))$, where $T(n)$ is the time needed to construct the suffix tree of a CS-tree of size $n$. Therefore, our algorithm achieves a $T(n) = O(n \log\log n)$ computing time for general CS-trees, and an optimal $T(n) = O(n)$ computing time for shallow $k$-ary CS-trees.

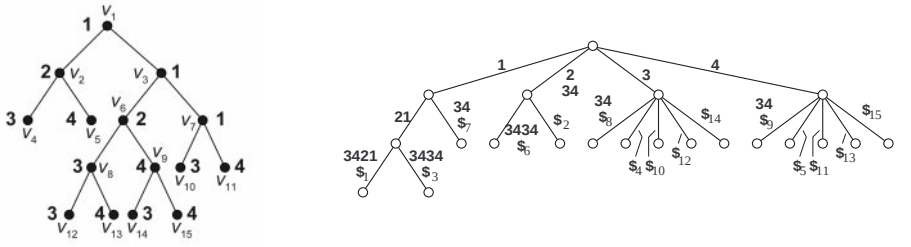
## 4 The BSuffix Tree

In this section, we propose a new data structure, the Bsuffix tree, which enables efficient queries of completely balanced binary trees from any binary forest (including a single tree). It can also be used for querying completely balanced $k$-ary subtrees from any $k$-ary forest ($k$ need not be constant in this case), but we will deal with binary trees at first. The Bsuffix tree is a data structure for matching of nodes, but it can be also used for matching of edges (see subsection 4.3).

### 4.1 Definition of the BSuffix Tree

Consider a completely balanced binary tree $P$ of height $h$. Let $p_1, p_2, \ldots, p_{2^h-1}$ be the nodes of $P$ in breadth-first order, and let $c_i \in \{1, \ldots, n\}$ be the alphabet given for node $p_i$. Note that $p_{\lfloor i/2 \rfloor}$ is the parent of $p_i$ in this order. We call $c_1 c_2 \cdots c_{2^h-1}$ the label of $P$. We call substring $c_{2^i} \cdots c_{2^{i+1}-1}$ of this label a Bcharacter. Furthermore, we call a string of Bcharacters a Bstring. For Bstring $b_1 b_2 \ldots b_n$, we call $b_1 b_2 \ldots b_m (m < n)$ a Bprefix of the Bstring. Note that $c_1 c_2 \cdots c_{2^h-1}$ is a Bstring of length $h$. For two Bcharacters $b_1$ and $b_2$, we let $b_1 > b_2$ if $b_1$ is lexicographically larger than $b_2$ in the normal string representation. Note that Bcharacter $b = c_{2^i} \cdots 2_{2^{i+1}-1}$ can be represented by node $p_{2^i} \in P$ and integer $i$.

Consider a binary forest $U$ of size $n$ whose nodes are labeled with a character of an integer alphabet $\{1, \ldots, n\}$. Let $v_1, v_2, \ldots, v_n$ be the concatenated list of the breadth-first-ordered node lists of all the binary trees in forest $U$, and let $a_i \in \{1, \ldots, n\}$ be the label of node $v_i$. Let $L_i$ be the the label of the largest completely balanced binary subtree of $U$ whose root is node $v_i$. We call $L_i$ followed by $\$_i \notin \{1, \ldots, n\}$ ($\$_i \neq \$_j$) the label of node $v_i$. If the roots of two




(1) Binary tree $U$ (2) Bsuffix tree for $U$

**Fig. 2.** An example of the Bsuffix tree.

completely balanced binary subtrees $P_1$ and $P_2$ of $U$ are the same node and $P_1$ includes $P_2$, the label of $P_2$ is a Bprefix of the label of $P_1$. The Bsuffix tree of $U$ is the compacted trie $T$ of the labels of all the nodes in $U$ in the Bstring sense, *i.e.*, the outgoing edges from some node in the suffix tree have a label of different Bcharacter. Figure 2 shows an example of a Bsuffix tree. By using $T$, we can very easily query any completely balanced binary subtree of $U$.

Edge labels of $T$ can be represented by the first node in $T$ and the depths of the first and the last nodes in the pattern. Therefore $T$ can be stored in $O(n)$ space. Note that we can access any member of the edge label of $T$ in a constant time if we have both the breadth-first list and the depth-first list of the nodes of each tree in forest $U$.

### 4.2 Construction of the BSuffix Tree

In this subsection, we describe the $O(n)$ algorithm for constructing the Bsuffix tree $T$ of $U$.

If forest $U$ consists of only nodes with less than two children, it is obvious that we can construct the Bsuffix tree of $U$ in $O(n)$ time. Otherwise, we first construct a new binary forest $U'$ as follows: For every node $v_i$ with two children $v_j, v_{j+1}$, construct a node of $U'$ (let it be $w_i$). If $v_j$ and/or $v_{j+1}$ have two children, let $w_i$ be the parent of $w_j$ and/or $w_{j+1}$ in forest $U'$. Radix sort the label pairs $(a_j, a_{j+1})$ and remove duplicates. Let the label $a'_i$ of $w_i$ be the rank of the label pair $(a_i, a_{i+1})$ in the sorted list. Notice that the number of nodes in $U'$ is not larger than $n/2$. We construct the Bsuffix tree $T'$ of $U'$ by using our algorithm recursively. Figure 3 shows an example of this recursive construction of new binary forests (trees in this case). Next, we construct $T$ from $T'$.

If we are given the lexicographically sorted list of all the node labels of $U$ and the length (*i.e.*, number of Bcharacters) of the longest common Bprefix of adjacent Bstring labels in this list, we can construct Bsuffix tree $T$ in a linear time. We obtain these two pieces of information from $T'$.

Notice that the in-order traverse of leaves of $T'$ is also a lexicographically sorted list of all the first-character-deleted labels of nodes that have two children

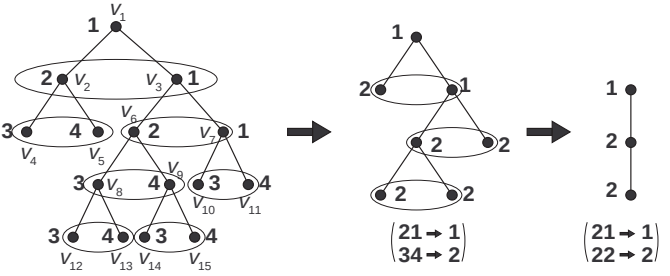



**Fig. 3.** Recursive construction of new binary trees in computing Bsuffix tree

in $T$. Thus we can obtain the lexicographically sorted list of all the node labels of $U$ by radix sorting the concatenated list of the in-order traverse of leaves of $T'$ and the labels of nodes with no or only one child.

The longest common Bprefix length $l$ of adjacent labels can also be obtained from $T'$. If the first characters of two adjacent labels are different, $l = 0$. Otherwise, if one of the adjacent labels consists of only one character, the depth is $l = 1$. Otherwise, we compute the depth as follows. Let $v_i$ and $v_j$ be the adjacent nodes. Notice that we can obtain the longest common Bprefix length $l'$ of labels of $w_i$ and $w_j$ in $U'$ in a constant time (see Theorem 2). Then it is clear that $l = l' + 1$.

In this way we can construct $T$ from $T'$ in a linear time. We obtain $T(n) = T(n/2) + O(n)$, where $T(n)$ denotes the time taken to compute the Bsuffix tree of a binary tree of size $n$. Therefore we conclude that our algorithm runs in $O(n)$ time.

### 4.3 Discussions on the Bsuffix Tree

Bsuffix trees are very similar to normal suffix trees. It enables $O(m \log m)$ query for a completely balanced binary tree pattern of size $m$. It can also be used for finding (largest) common completely balanced binary subtrees of two binary trees in linear time. We can also enumerate frequent patters of completely balanced binary trees in linear time by using this data structure.

The data structure and our algorithm assume that the labels are given to nodes, but they can very easily be modified to deal with edge-matching problems as follows: Let the label of any node except for the root be the label of the incoming edge from its parent. Then $T'$ in the above algorithm can be used as the compacted trie for edge matching.

Bsuffix trees can also be used for querying completely balanced $k$-ary trees from any $k$-ary forest $U$. First, if a node has less than $k$ children, remove the edges between it and its children. Otherwise, we reconstruct each node that has $k$ children as a completely balanced binary tree of depth $\lceil \log_2 k \rceil$ and move each child to its leaf. For each inside node and leaf to which no node was mapped, give as its label a new character that is not in use. Notice that the size of the reconstructed forest is at most twice as that of the original one. Then construct

the Bsuffix tree for this reconstructed binary tree. It can obviously used for querying completely balanced $k$-ary trees.

## 5 Concluding Remarks

We have described an $O(n \log \log n)$ algorithm for constructing the suffix tree of a common suffix tree (CS-tree). For trees called shallow $k$-ary trees, we also described an $O(n)$ algorithm. In addition, we proposed a new data structure called a Bsuffix tree, that enables efficient query for completely balanced subtrees.

The existence of a linear time algorithm for constructing the suffix tree of any trees for large alphabets remains as an open question, as does the existence of more useful suffix trees that allow querying more general and flexible patterns than paths or completely balanced trees.

## References

1. A. Apostolico and Z. Galil, eds., "Pattern Matching Algorithms," *Oxford University Press, New York,* 1997.
2. D. Breslauer, "The Suffix Tree of a Tree and Minimizing Sequential Transducers," *J. Theoretical Computer Science, Vol. 191,* 1998, pp. 131-144.
3. M. T. Chen and J. Seiferas, "Efficient and Elegant Subword Tree Construction," A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words, Chapter 12,* NATO ASI Series F: Computer and System Sciences, 1985, pp. 97-107.
4. R. Cole, R. Hariharan and P. Indyk, "Tree Pattern Matching and Subset Matching in Deterministic $O(n \log^3 n)$-time," *Proc. 4th Symposium on Discrete Mathematics (SODA '99),* 1999, pp. 245-254.
5. M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," *Proc. 38th IEEE Symp. Foundations of Computer Science (FOCS '97),* 1997, pp. 137-143.
6. M. Farach and S. Muthukrishnan, "Optimal Logarithmic Time Randomized Suffix Tree Construction," *Proc. 23rd International Colloquium on Automata Languages and Programming (ICALP '96),* 1996, pp. 550-561.
7. R. Giancarlo, "The Suffix Tree of a Square Matrix, with Applications," *Proc. 4th Symposium on Discrete Mathematics (SODA '93),* 1993, pp. 402-411.
8. D. Gusfield, "Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology," *Cambridge University Press,* 1997.
9. D. Harel and R. R. Tarjan, "Fast Algorithms for Finding Nearest Common Ancestors," *SIAM J. Computing, Vol. 13,* 1984, pp. 338-355.
10. S. R. Kosaraju, "Efficient Tree Pattern Matching," *Proc. 30th IEEE Symp. Foundations of Computer Science (FOCS '89),* 1989, pp. 178-183.
11. E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM, Vol. 23,* 1976, pp. 262-272.
12. E. Ukkonen, "On-Line Construction of Suffix-Trees," *Algorithmica, Vol. 14,* 1995, pp. 249-60.
13. P. Weiner, "Linear Pattern Matching Algorithms," *Proc. 14th Symposium on Switching and Automata Theory,* 1973, pp. 1-11.

## Appendix: Proofs of Theorems 3 and 4

### Proof of Theorem 3

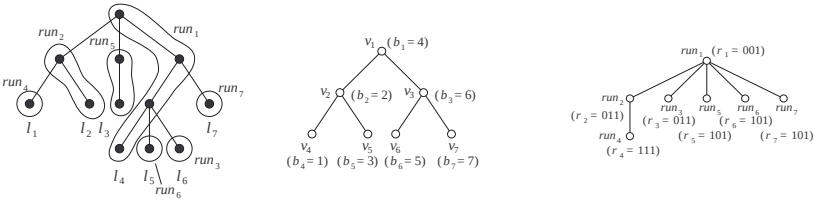
We achieve an $O(\log \log n)$ computing time by means of the following algorithm.

Let $m$ be the number of leaves in the target tree $T$. We call a path from some node to some leaf a 'run'. We first divide all nodes in the tree into $m$ runs. as follows:

1. Index the leaves by in-order traversing of $T$, and let them be $l_1, l_2, \ldots, l_m$.
2. Consider a completely balanced binary tree $B$ of height $\lceil \log_2(m+1) \rceil$. Let $v_1, v_2, \ldots, v_{m'}$ be the nodes of $B$ in the breadth-first order, where $m \leq m' = 2^{\lceil \log_2(m+1) \rceil} - 1 < 2m$. Map the leaves $l_1, l_2, \ldots, l_m$ to the in-order traverse of the nodes in $B$. Let $l_{b_i}$ be the leaf of $T$ to be mapped to $v_i$ in $B$.
3. For $i = 1, 2, \ldots, m'$, construct runs $run_1, run_2, \ldots, run_{m'}$ (note that $m' - m$ of these are empty runs) as follows:
   - If $l_{b_i}$ does not exist, let $run_i = \phi$ (an empty run) and continue.
   - Find the maximum run $run_i$ that does not contain any node of runs $\{run_j | j < i\}$ and ends at leaf $l_{b_i}$. Note that unless $run_i$ starts from the root of $T$, the parent node of the first node (nearest to the root) of $run_i$ must belongs to some run $run_{p_i}$. We call $run_{p_i}$ the parent run of $run_i$. Note that we can construct a tree $R$ of runs by using this parent-child relationship.
   - Let $r_1 = 1$. If $i > 1$, compute the following $r_i$: Let $r$ be a binary number with only one 1-bit that is at the same position as the leftmost 1-bit of $i$, *i.e.*, let $r = 2^{\lfloor \log_2 i \rfloor}$. Then let $r_i = r_{p_i} \vee r$, where $\vee$ denotes bitwise OR. Note that the depth of $v_i$ in $B$ is $1 + \lfloor \log_2 r_i \rfloor = 1 + \log_2 r = 1 + \lfloor \log_2 i \rfloor$.

Figure 4 shows an example of $T$, $B$, and $R$. Note that $r_i$ is displayed in a binary number in Figure 4 (3).

Note that the parent of node $v_i$ in $B$ is $v_{\lfloor i/2 \rfloor}$. Thus for any node of depth $d$ in $B$ and some integer $d'$ such that $0 < d' < d$, we can access the node's ancestor of depth $d$ in a constant time by simply right-shifting its index by $d - d'$ bits. For any run and some integer $d$, it is clear that the node of depth $d$ in the run can be accessed in a constant time if each run manages its nodes. Thus, once we




(1) $T$ (2) $B$ (3) $R$

**Fig. 4.** Example of $T$, $B$, and $R$.

find the run that contains the target ancestor, we can find it in a constant time. We now discuss how to find the run.

Consider node $w$ in $T$ and let $run_i$ be the run that contains $w$. It is obvious that any ancestor of $w$ in $T$ is contained by one of the ancestor runs of $run_i$ in $R$. Furthermore, if $run_j$ is an ancestor of $run_i$ in $R$, then $v_j$ is also an ancestor of $v_i$ in $B$. Let $run_{a_1}, run_{a_2}, \ldots, run_{a_k}$ $(a_1 = 1 < a_2 < \cdots < a_k = i)$ be the ancestor runs of $run_i$ (including itself). Note that the binary number $r_i$ contains $k$ 1-bits.

For any $j$ such that $0 < j \leq k$, we can access $run_{a_j}$ in a constant time by using the value of $r_i$ as follows: Let $r'$ be a binary number that has only one 1-bit whose position is the same as the $j$th-rightmost 1-bit of binary number $r_i$. Let $r''$ be a binary number that has only one 1-bit whose position is the same as the leftmost 1-bit of binary number $r_i$. Then $a_j = \lfloor i \cdot (r'/r'') \rfloor$ (right-shift by $\log_2 r'' - \log_2 r'$ bits). Using this constant-time access to the ancestor runs, we can search the target run in $O(\log k)$ time by checking the depths of the first nodes of ancestor runs. Hence we conclude that the time for finding the target node is $O(\log \log n)$, because $k \leq 1 + \log_2 m' < 2 + \log_2 m$. Note that it is obvious that all of the data structures used above can be constructed in a total of $O(n)$ time.

**Proof of Theorem 4**

This case is far simpler than that of Theorem 3. For a completely balanced binary tree, we can find the ancestor of depth $d$ in a constant time by indexing nodes in breadth-first order and shifting the bits of indices. The case of shallow binary trees is also obvious: We can consider a minimum complete balanced binary tree that contains a shallow binary tree of size $n$ as its subgraph. Its size is $O(n)$, and it can be built in $O(n)$ time. We can find the target ancestor in the new tree in a constant time.

In a general shallow $k$-ary tree where $k$ is some constant, every node can be mapped to an $O(n)$ binary shallow tree in such way that the depth of a mapped node is a constant times as the depth of the original node in the original tree. In this way, we conclude that such an ancestor can be found in a constant time after linear-time preprocessing.

# An $O(1)$ Time Algorithm for Generating Multiset Permutations

Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
tad@cosc.canterbury.ac.nz

**Abstract.** We design an algorithm that generates multiset permutations in $O(1)$ time from permutation to permutations, using only data structures of arrays. The previous $O(1)$ time algorithm used pointers, causing $O(n)$ time to access an element in a permutation, where $n$ is the size of permutations. The central idea in our algorithm is tree traversal. We associate permutations with the leaves of a tree. By traversing this tree, going up and down and making changes when necessary, we spend $O(1)$ time from permutation to permutation. Permutations are generated in a one-dimensional array.

## 1 Introduction

Algorithms for generating combinatorial objects, such as (multiset) permutations, (multiset) combinations, well-formed parenthesis strings are a well studied area and many results are documented in Nijenhuis and Wilf [6], and Reingold, Nievergelt, and Deo [8], etc.

Let $n$ be the size of the objects to be generated. The most primitive algorithms are recursive ones for generating those objects in lexicographic order, causing $O(n)$ changes from object to object, and thus $O(n)$ time. To overcome this drawback, many algorithms were invented, which generate objects with a constant number of changes, $O(1)$ changes, from object to object. This idea of generating combinatorial objects with $O(1)$ changes is named "combinatorial Gray codes", and a good survey is given in [11]. In many cases, these changes are made by swappings of two elements, that is, two changes. It is still easy to design recursive algorithms for combinatorial generation with $O(1)$ changes, since we can control the paths of the tree of recursive calls and thus we can rather easily identify changing places. Note that combinatorial objects correspond to the leaves of the tree, meaning that it takes $O(n)$ time from object to object as the height of the tree is $n$. Further to overcome this shortcoming, several attempts were made to design iterative algorithms, which are called loopless algorithms in some ltera iterature, removing recursion, so that $O(1)$ time is achieved from object to object. At this stage, we need some care in defining the $O(1)$ time from object to object. In Korsh and Lipschutz [3], $O(1)$ time was achieved to generate multiset permutations, whose algorithm is a refinement of that by Hu and Tien

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 237–246, 1999.
© Springer-Verlag Berlin Heidelberg 1999

[1]. In this algorithm, multiset permutations are given one after another in a linked list. The operations on the list are manipulated by pointers, involving shift operations in $O(1)$ time. For example, the list $(1, 1, 1, 2, 2, 2)$ with $n = 6$ can be converted to $(2, 2, 2, 1, 1, 1)$ in $O(1)$ time by changing pointers. We assume that the above conversion takes $O(n)$ time in this paper, and we claim that multiset permutations can be generated in $O(1)$ time using arrays, not pointers.

This kind of strict requirement for $O(1)$ time was demonstrated in the recent development in parenthesis strings generation. An $O(1)$ change algorithm was developed in Ruskey and Proskurowski [10] and an $O(1)$ time algorithm with pointer structures was achieved in Roelants van Baronaigien [9], and they challenged the readers, asking whether there could be $O(1)$ algorithms with arrays, whereby stricter $O(1)$ time could be achieved. This problem was recently solved by three independent works of Mikawa and Takaoka [5], Vajnowski [13], and Walsh [14]. Note that we can access any element of a combinatorial object in $O(1)$ time in array implementation, whereas we need $O(n)$ time in linked list implementation, as we must traverse the pointer structure. The algorithm by Ko and Ruskey [2] generates multiset permutations with swappings of two elements, but not with $O(1)$ time from permutation to permutation.

The main idea of $O(1)$ time for multiset permutation generation in this paper is tree traversal. The generation tree for a set of permutations, arranged in some order, on the given multiset is a tree whose paths to the leaves correspond to the permutations. Basically we traverse the tree in movements of (up, cross, down). The move "up" is to go up the tree from a node to one of its ancestors. The move "cross" is to move from a node to its adjacent sibling, causing a swapping with the element at that level and the one at a level closer to the leaf. The move "down" is to go down from a node to one of its descendants, which we call the landing point. The landing point has no sibling and the path to the leaf has no branching, causing a straight line. It is important that we avoid traversing this straight line node by node. The core part of the algorithm is centered on how to compute the positions to which we go up and down, and where we should perform swappings. Although the use of tree structure for combinatorial generation was originated in Lucas [4] and Zerling [15], and well known, the technique of tree traversal in this paper is new.

Since the final algorithm is rather complicated, we go through a stepwise refinement process, going from simple structures to details. In Section 2, we define the generation tree and design a recursive algorithm that traverses this tree to generate multiset permutations. We give a formal proof of the recursive algorithm. In Section 3, we design an iterative algorithm based on the recursive algorithm. We first describe an informal framework for an iterative algorithm, and translate the recursive algorithm into an iterative one guided by the framework. The resulting iterative algorithm generates multiset permutations in $O(1)$ time in a one-dimensional array. As additional data structures, we use a few more arrays, causing $O(kn)$ space requirement, where $k$ is the number of distinct elements in the multiset. In Section 4, we give concluding remarks.

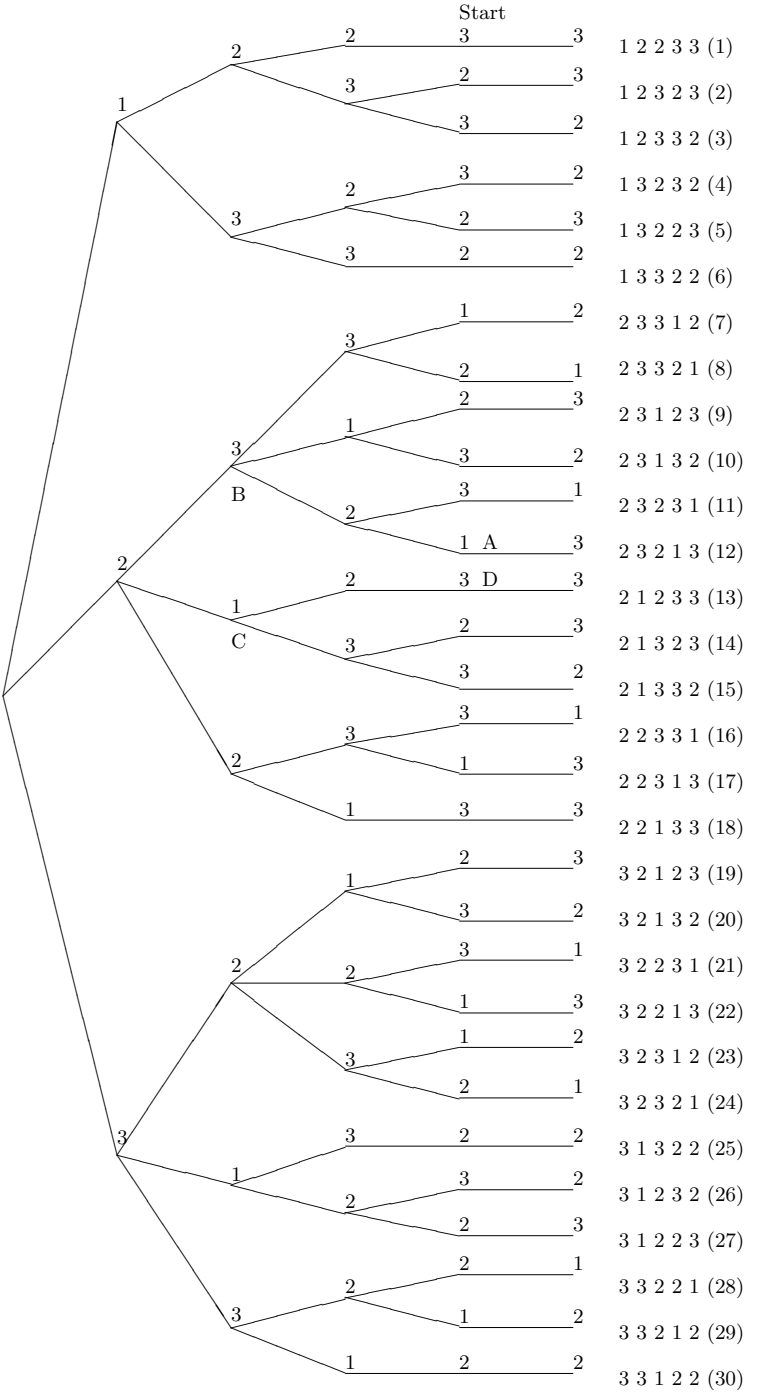



**Fig. 1.** Generation tree for permutations on [1,2,2,3,3]

## 2 Permutation Tree and Recursive Algorithm

We denote a multiset by [...] and ordinary set by {...}. Those notations identify operations such as set union and set subtraction when the same symbols are used on sets and multisets. We convert a multi-set $S$ to the set $set(S)$ by removing repetition of each element. If $S = [1, 1, 2]$, for example, $set(S) = \{1, 2\}$. Let a multiset $S = [1, ..., 1, 2, ..., 2, ..., k, ..., k]$ be defined by $(m_1, m_2, ..., m_k)$, where $m_i$ is the multiplicity of $i$. Let $P$ be a set of all multiset permutations on $S$ arranged in some order. Since $S$ is the base multiset for $P$, we use the notation $base(P) = S$. We use word "permutation" for "multiset permutation" for simplicity. Let $N = n!/(m_1!...m_k!)$. Then we have $|P| = N$. Let $x \in P$ be a permutation given by $x = a_1a_2...a_n$. We construct the permutation tree of $P$, $T(P)$, in such a way that each $x \in p$ is associated with a path from the root to a leaf. Since the path from the root to a leaf is unique in a tree, $x$ will also correspond to the leaf at the end of the path. If $x'$ is the next permutation of $x$ in $P$, we correspond $x'$ to the next leaf of that for $x$. Let $x'$ be given by $x' = a_1...a_ia'_{i+1}...a'_n$. That is, $x'$ shares some prefix (possibly empty) with $x$. Then the paths to the two adjacent leaves $x$ and $x'$ share the path corresponding to $a_1...a_i$.

*Example 1.* Let $S$ be given by $(m_1, m_2, m_3) = (1, 2, 2)$. We give $P$ and $T(P)$ in the previous page.

In this example, we assume we give permutations in $P$ in this order. The number shown by $(i)$ to the right side of each permutation is to indicate the $i$th permutation. This list of permutations also gives the shape of the tree $T(P)$. The root at level 0 has three branches leading to sibling nodes at level 1 with labels 1, 2, and 3. Then the node at level 1 with label 1 has two branches leading to sibling nodes with labels 2 and 3, etc. We have $5!/(1!2!2!) = 30$ members in $P$. We draw the tree horizontally, rather than vertically, for notational convenience.

We use a list $nodes[i]$ of elements from the set $set(S)$ of a multiset $S$ to keep track of siblings at level $i$, We define two types of operation with notation $\Leftarrow$. Operation $t \Leftarrow nodes[i]$ means that the first element of $nodes[i]$ is moved to a single variable $t$. Operation $nodes[i] \Leftarrow t$ means that $t$ is appended to the end of $nodes[i]$. The history of variable $t$ keeps track of all elements in $nodes[i]$. For a list $L$, $set(L)$ is the set made of elements taken from $L$. $Next(L)$ is the second element of $L$. We identify nodes of the tree by array elements of $a$ whenever clear from context. A recursive algorithm is given below.

ALGORITHM 1 *Recursive algorithm*

```
1. procedure generate(i);
2. var t, s;
3. begin
4.    nodes[i] := (a[i]);
5.    if i ≤ n then
6.    repeat
7.       generate(i + 1);
```

*8.* *Let s be the leftmost position of next(nodes[i]) in a such that* $i < s$
*9.* **if** *a[i] is not a last child* **then** *swap*$(a[i], a[s])$*;*
*10.* **if** *a*$[i-1]$ *is a first child* **then**
*11.* **if** $a[i] \neq a[i-1]$ **then** $nodes[i-1] \Leftarrow a[i]$*;*
*12.* $t \Leftarrow nodes[i]$
*13.* **until** $nodes[i] = \emptyset$
*14.* **end***;*
*15.* **begin** { *main program* }
*16.* *Let* $a = [1, ..., 1, 2, ..., 2, ..., k, ..., k]$*;*
*17.* *generate*$(1)$
*18.* **end***.*

Let $tail(a)$ be the consecutive portion of the tail part of $a$ such that all elements in $tail(a)$ are equal to $a[n]$. Let $Q$ be a set of all permutations generated from $S - [a_1, ..., a_i]$. Then the notation $a_1...a_iQ$ means the set of permutations made by concatenating $a_1...a_i$ with all members of $Q$. We use notations $a_i$ and $a[i]$ interchangeably to denote the $i$-th element of array $a$. We state the following obvious lemmas.

**Lemma 1.** *Let $P$ be the set of permutations on the multiset $S$ of size $n$ and $first(P) = \{x_1 | x_1x_2...x_n \in P\}$. Then $first(P) = set(S)$.*

**Lemma 2.** *Let $S$ be a multiset and $P$ be the set of permutations on $S$. Then $P = b_1Q_1 \cup ... \cup b_lQ_l$, where $set(S) = \{b_1, ..., b_l\}$ and $Q_j$ is the set of permutations on the multiset $S - [b_j]$.*

**Theorem 1.** *Algorithm 1 generates all permutations on $S$ by swaooing.*

Proof. We show by backward induction that $generate(i)$ generates the set $P$ of all permutations on $[a[i], ..., a[n]]$. The case of $i = n$ is obvious. Suppose the theorem is true for $i + 1$. Then observe that the first call of $generate(i + 1)$ in $generate(i)$ will generate all permutations on $[a[i + 1], ..., a[n]]$ by induction, which we denote by $Q$. ¿From Lemma 1, it holds that $first(Q) = set(a[i + 1], ..., a[n])$. ¿From lines 10-11 of the program we have $set(nodes[i]) = \{a[i]\} \cup first(Q) = set[a[i], ..., a[n]]$ immediately after the first call of $generate(i + 1)$. Let $set[a[i], ..., a[n]] = \{b_1, ..., b_l\}$ for some $l$ such that $b_1 = a[i]$ at the beginning of $generate(i)$. Then we are generating $a[1]...a[i - 1]b_jQ_j$ for $j = 1, ..., l$, where $base(Q_j) = base(Q_{j-1}) - [b_j] \cup [b_{j-1}]$ for $j > 1$, and $Q_1 = Q$. Since we swap $a[i]$ and $a[s]$ at the end of each call of $generate(i+1)$, $l$ different multisets are given in $(a[i + 1], ..., , a[n])$ as $base(Q_j)$ before calls of $generate(i + 1)$. From Lemma 2, we conclude that the set $P$ is generated by calling $generate(i+1)$ with all $b_j$ given in $t$.

*Example 2.* Let $i = 1$, and suppose we start from $a = [1, 2, 2, 3, 3]$. Then we have $base(Q) = [2, 2, 3, 3]$, and $first(Q) = \{2, 3\}$. Since these elements are appended to $nodes[i] = (1)$, we have $nodes(1) = (1, 2, 3)$, which forms the $first(P)$, where $P$ is the entire set of permutations.

Note that the choice of position $s$ at line 9 can be arbitrary as long as we choose a position $s$ such that $next[nodes[i]] = a[s]$ and $i < s$. Two consecutive permutations before and after *swap* are different only at $i$ and $s$ such that $i < s$. In this context, we say $i$ is the difference point and $s$ is the solution point. In Example 1, the permutations on [1, 2, 2, 3, 3] are generated by this algorithm.

## 3 $O(1)$ Implementation

Algorithm 1 takes $O(n)$ time from permutation to permutation due to its recursive structure. In this section we avoid this $O(n)$ overhead time for traversing the tree. By using some data structures, we jump from node to node in the permutation tree.

When we first call $generate(1)$, it will go down to level $n$ and come back to level $i = n - tail(a) + 1$ without doing any substantial work, since all nodes on this path are last children. At this level the algorithm append $a[i] = k$ to $nodes[i-1]$ and comes to level $i = n - tail(a)$, that is, $i$ is decreased by 1. Then it swaps $a[i]$ and $a[i+1]$, add new $a[i]$ to $next[i-1]$, and go down to level $n$. When the algorithm traverses the tree downwards and upwards, there are many steps that can be avoided. Specifically we can start from level $i = n - tail(a) + 1$. After we perform swapping, we can come down straight to level $i = n - tail(a) + 1$ with the new $tail(a)$. We keep two arrays $up$ and $down$ to navigate our traversal in the tree; $up[i]$ tells where to go up from level $i$ and $down[i]$ tells where to go down from level $i$. Level $up[i]$ is the level where we hit a non-last child when we traverse the tree from level $i$. The formal definition of $down[i]$ is given later.

When we perform $swap(a[i], a[s])$, we need the information of $s$ at hand without computing the leftmost position of $next(nodes[i])$ to the right of $i$. Obtaining this information for level $up[i]$ is carried out when we come to a last child at level $i$ by updating $s[up[i]]$ by $i$ if $a[i] = next(nodes[up[i]])$ for the first time. For this purpose, the variable $s$ is given by array $s$ to keep the information of $s$ for each level.

*Example 3.* In Figure 1, we can start from the point *Start*. Suppose we reached the point $A$ after several steps. We have $up[4] = 2$, which we inherited from $up[3]$. Since $next(nodes[2]) = 1$, we set $s[up[4]] = 4$. We make transition $A \to B \to C \to D$. When we cross from $B$ to $C$, we swap $a[2]$ and $a[4]$ and come to the landing point $D$.

We translate Algorithm 1 into the following informal iterative algorithm for traversing the tree, resulting subsequently in Algorithm 3.

ALGORITHM 2 *Informal iterative tree traversal*

*initialize a to be the first permutation on S;*
*initialize up[i] and down[i] to i for* $i = 0, ..., n$*;*
*initialize nodes[i] to (a[i]) for* $i = 1, ..., n$*;*

$i := n - |tail(a)| + 1$;
**repeat**
  **if** $nodes[i-1]$ *has not been updated by* $a[i-1]$*'s children* **then** *update it;*
  *output(a);*
  **if** $a[i]$ *is not a last child* **then** $swap(a[i], a[s[i]])$; {*action cross*}
  *update* $nodes[i-1]$;
  **if** $a[i]$ *is a last child* **then begin**
    $up[i] := up[i-1]$; $up[i-1] := i-1$;
    *update* $s[up[i]]$;
    *update* $down[up[i]]$;
    **if** $i = n - |tail(a)| + 1$ {$a[i], ..., a[n]$ *form a straight line*}
      **then** $i := up[i]$; {*going up*}
      **else** $i := down[i]$; {*going down*}
  **end**
  **else** {$a[i]$ *is not a last child*}
    $i := down[i]$ {*going down*}
**until** $i = 0$ {*root level*}.

As we cross from a node to the next, swapping two array elements, $tail(a)$ grows or shrinks. For the computation of $tail(a)$, which, in turn, gives the information of *down*, we use array *run*. Array *run* is to keep track of the length of consecutive array elements that are equal to $a[i]$ when we traverse the path of last children starting at $a[up[i]]$. Array *run* is computed by increasing $run[up[i]]$ by 1 when we hit $a[up[i]] = a[i]$ on the path, and reset to 0 otherwise. These values of *run* are used to compute $tail(a)$ after we perform the swap operation, whereby we can compute the values of *down*. Specifically we can set $down[up[i]] := i - run[up[i]]$ if $a[up[i]] = a[n]$ and $down[i] = i + 1$. Note that $down[i] = i + 1$ means $a[i+1] = ... = a[n]$, since the landing point is the left end of $tail(a)$. In other cases, $down[up[i]]$ is set to $i + 1$ or $i$ depending on the situation at level $i$, as described in the comments of Algorithm 3.

The Boolean value of $mark[i] = true$ is to show that the value of $down[i]$ has been set and prevent further modification.

If we hit a non-last child we always go down guided by $down[i]$. If we hit a last child, we may go down or go up, if $i < down[i]$ or $i = down[i]$ respectively.

When we go up to the ancestor, the path to the node on which we stand consists of last children. We call this path the current path. When we go down from a node to a descendant, the path from the node to the descendant consists of first children. We call this path the opposite path. Most of the work in the algorithm is to prepare the necessary environment for the opposite path when we are traversing the current path. We jump over the opposite path from the left end to the landing point, whereas we traverse the current path node by node. Whenever we come to a node, the necessary information for the next action must be ready.

*Example 4.* In Fig.2, $run(up[i]) = 2$ for two $b$'s between $d$ and $c$ on the current path. We swap $b$ at level $up[i]$ and $c$ at level $i$ and go down to level $down[up[i]]$,

that is, the leftmost position of $tail(a)$ on the opposite path, which consists of five $b$'s.

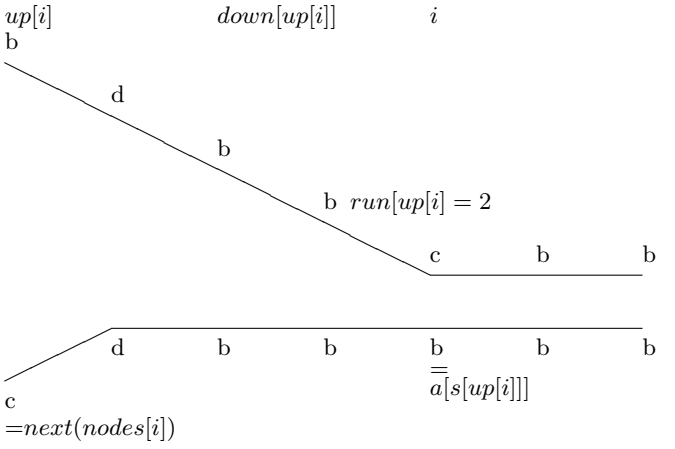



**Fig. 2.** Illustration of run

We leave the details of implementation including the data structure $nodes[i]$ in a full Pascal program at "http://www.cosc.canterbury. ac.nz/∼tad/perm.p".

ALGORITHM 3 *Iterative algorithm for multiset permutations* {*with comments*}.

```
1. a := [1, ..., 1, 2, ..., 2, ..., k, ..., k];
2. for i := 1 to n do begin nodes[i] := (a[i]); s[i] := 0;
                            mark[i] := false; up[i] := i end;
3. i := n − m[k] + 1;
4. up[0] := 0;
5. repeat
6.   if a[i−1] is a first child and nodes[i−1] has not been updated by its children
7.     then if a[i] ≠ a[i − 1] then nodes[i − 1] ⇐ a[i];
8.     if |nodes[i]| > 1 then begin {current node a[i] is not a last child}
9.       swap(a[i], a[s[i]]); {crossing}
10.      mark[i] := false; {this shows down[i] for level i needs to be updated
                            for later use}
11.      nodes[s[i]] := (a[s[i]]); {prepare nodes for the solution point}
12.      remove first of nodes[i];
13.      if a[i − 1] is a first child then
14.        if a[i] ≠ a[i − 1] then nodes[i − 1] ⇐ a[i]; {update nodes[i − 1]}
15.      s[i] := 0 {solution point for level i is to be set}
16.  end;
```

```
17.   run[i] := 0; {initialize run[i]}
18.   if a[i] is a last child then begin
19.     up[i] := up[i − 1]; {up propagates} up[i − 1] := i − 1; {up[i − 1] is reset}
20.     if i < n then
21.       if a[up[i]] = a[i] and up[i] < i then
22.         run[up[i]] := run[up[i]] + 1 {extend run for level up[i]}
23.       else run[up[i]] := 0;{reset run}
24.     if a[i] = next(nodes[up[i]]) then begin {do the following for up[i]}
25.       if s[up[i]] = 0 begin
26.         if i = down[i] − 1 and a[up[i]] = a[n] then begin
27.           down[up[i]] := i − run[up[i]]; {compute down for up[i]}
28.           mark[up[i]] := true; {down for up[i] has been finalized}
29.         end else down[up[i]] := i + 1; {down[up[i]] at least i + 1}
30.         s[up[i]] := i {solution point for up[i] is i}
31.       end
32.       else begin {s[up[i]] ≠ 0}
33.         nodes[i] := (a[i]); {prepare nodes for the opposite path}
34.         if mark[up[i]] = false then down[up[i]] := i {update down}
35.       end
36.     else begin {a[i] ≠ next(nodes[up[i]])}
37.       nodes[i] := (a[i]); {similar to 33}
38.       if mark[up[i]] = false then down[up[i]] := i {similar to 34}
39.     end;
40.     if i < down[i] then begin {going down}
41.       mark[i] := false; {set mark to false for the opposite path}
42.       i := down[i];
43.       nodes[i] := (a[i]); {initialize nodes}
44.     end else
45.     begin {going up}
46.       i1 := i;
47.       i := up[i];
48.       up[i1] := i1; {reset up for the old i}
49.     end
50.   end else
51.   begin {a[i] is not a last child, going down}
52.     i := down[i];
53.     nodes[i] := (a[i]); {similar to 43}
54.   end;
55. until i = 0.
```

## 4 Concluding Remarks

We developed an $O(1)$ time algorithm for generating multiset permutations. The main idea is tree traversal and identification of swapping positions. This technique is general enough to solve other combinatorial generation problems. In

fact, this technique stemmed from that used in generation of parenthesis strings in [5]. The author succeeded in designing $O(1)$ time generation algorithms for other combinatorial objects, such as in-place combinations, reported in [12].

The key point is the computation of *up*, *down*, and $s$, the solution point, in which *up* is very much standard in almost all kinds of combinatorial objects. If we always go down to leaves, we need not worry about *down*. This happens with more regular structures, such as binary reflected Gray codes, ordinary permutations, and parenthesis strings, where we can concentrate on the computation of $s$. Multiset combinations and permutations have more irregular structures, that is, straight lines at some places, which require the computation of *down*, in addition to that of $s$. There are still many kinds of combinatorial objects, for which only O(1) change algorithms are known. The present technique will bring about O(1) time algorithms for those objects.

The space requirement for the algorithm is $O(kn)$. It is open whether this can be optimized to $O(n)$.

## References


1. Hu, T.C. and B.N. Tien, Generating permutations with nondistinct items, Amer. Math. Monthly, 83 (1976) 193-196
2. Ko, C.W. and F. Ruskey, Generating permutations of a bag by interchanges, Info. Proc. Lett., 41 (1992) 263-269
3. Korsh, J. and S. Lipshutz, Generating multiset permutations in constant time, Jour. Algorithms, 25 (1997) 321-335
4. Lucas, J., The rotation graph of binary trees is Hamiltonian, Jour. Algorithms, 8 (1987) 503-535
5. Mikawa, K. and T. Takaoka, Generation of parenthesis strings by transpositions, Proc. the Computing: The Australasian Theory Symposium (CATS '97) (1997) 51-58
6. Nijenhuis, A. and H.S. Wilf, Combinatorial Mathematics, Academic Press (1975)
7. Proskurowski, A. and F. Ruskey, Generating binary trees by transpositions, Jour. Algorithms, 11 (1990) 68-84
8. Reingold, E.M., J. Nievergelt, and N. Deo, Combinatorial Algorithms, Prentice-Hall (1977)
9. Roelants van Baronaigien, D., A loopless algorithm for generating binary tree sequences, Info. Proc. Lett., 39 (1991) 189-194.
10. Ruskey, F. and D. Roelants van Baronaigien, Fast recursive algorithms for generating combinatorial objects, Congr. Numer., 41 (1984) 53-62
11. Savage, C, A survey of combinatorial Gray codes, SIAM Review, 39 (1997) 605-629
12. Takaoka, T., $O(1)$ Time Algorithms for combinatorial generation by tree traversal, Computer Journal (to appear)(1999)
13. Vajnovski, V., On the loopless generation of binary tree sequences, Info. Proc. Lett., 68 (1998) 113-117
14. Walsh, T.R., Generation of well-formed parenthesis strings in constant worst-case time, Jour. Algorithms, 29 (1998) 165-173
15. Zerling, D., Generating binary trees by rotations, JACM, 32 (1985) 694-701

# Upper Bounds for MaxSat: Further Improved*

Nikhil Bansal[1] and Venkatesh Raman[2]

[1] Carnegie Mellon University, Pittsburgh, USA
nikhil@cs.cmu.edu
[2] The Institute of Mathematical Sciences, Chennai, India,
vraman@imsc.ernet.in

**Abstract.** Given a boolean CNF formula $F$ of length $|F|$ (sum of the number of variables in each clause) with $m$ clauses on $n$ variables, we prove the following results.

- The MAXSAT problem, which asks for an assignment satisfying the maximum number of clauses of $F$, can be solved in $O(1.341294^m|F|)$ time.
- The parameterized version of the problem, that is determining whether there exists an assignment satisfying at least clauses of the formula (for some integer ), can be solved in $O( \ ^2 1.380278^k + |F|)$ time.
- MAXSAT can be solved in $O(1.105729^{|F|}|F|)$ time.

These bounds improve the recent bounds of respectively $O(1.3972^m|F|)$, $O( \ ^2 1.3995^k + |F|)$ and $O(1.1279^{|F|}|F|)$ due to Niedermeier and Rossmanith [11] for these problems. Our last bound comes quite close to the $O(1.07578^{|F|}|F|)$ bound of Hirsch[6] for the Satisfiability problem (not MAXSAT).

## 1 Introduction

There are several approaches to deal with an NP-complete problem. A popular approach is to settle for a polynomial time approximate solution with provable guarantee on the quality of the solution. Another well studied approach is to explore some structure in the problem to design efficient algorithms (for example designing efficient algorithms for NP-complete graph problems in special classes of graphs). Another recent approach for the parameterized versions of the NP-complete problems is to look for fixed parameter tractable algorithms [5]. Despite all these approaches, often many NP-hard problems have to be solved exactly, in practice. Due to this reason, considerable attention has been paid to designing efficient (better than the naive $2^n$) algorithms for several NP-hard problems[1,10,9,13,12,7,6,3,15]. Satisfiability, Vertex Cover, Independent Set are some of the problems for which such efficient exact algorithms are known. In this paper we develop efficient exact algorithms for another fundamental NP-hard problem MAXSAT (Maximum Satisfiability). Despite considerable advances in

* The work was done while the first author was at IIT Mumbai and visited IMSc Chennai as a summer student.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 247–258, 1999.
© Springer-Verlag Berlin Heidelberg 1999

the study of exact algorithms for the Satisfiability problem, desigining efficient exact algorithms for MAXSAT has started receiving the attention of researchers only recently.

For the rest of the paper we will be dealing with boolean CNF formulae on $n$ variables and $m$ clauses. For a formula $F$, $|F|$ will denote the length of the formula which is the sum of the number of variables in each clause.

Raman, Ravikumar and Srinivasa Rao [14] showed that MAXSAT problem remained NP-hard even if every variable appears at most three times. They gave an $O(\sqrt{3}^n|F|)$ algorithm for this version of MAXSAT problem. For the general MAXSAT, Mahajan and Raman [8] gave an $O(|F|\phi^m)$ algorithm, where $\phi = 1.6181...$ is the golden ratio. This was improved to $O(|F|1.3995^m)$ by Niedermeier and Rossmanith [11]. In this paper, we improve this to $O(|F|1.341294^m)$. For the natural parameterized question, whether at least $k$ clauses of the formula be satisfiable, Niedermeier and Rossmanith [11] improved the algorithm of Mahajan and Raman [8] to obtain a bound of $O(k^2 1.3995^k + |F|)$. In this paper we improve this to $O(k^2 1.380278^k + |F|)$ time.

In the development of efficient exact algorithms for SAT, bounds where the exponent is the length of the formula have been obtained (see, for example, [6,7]). We also design algorithms for MAXSAT along similar lines to obtain a bound of $O(1.105729^{|F|}|F|)$ for MAXSAT improving upon the recent bound of $O(1.1279^{|F|}|F|)$ due to Niedermeier and Rossmanith [11]. This bound implies that if every variable occurs at most 6 times, then we have a $((2-\epsilon)^n|F|)$ algorithm for some positive $\epsilon < 1$. Our bound also comes quite close to the $O(1.07578^{|F|}|F|)$ bound of Hirsch[6] for the Satisfiability problem (not MAXSAT).

En route to our main algorithms proving the above bounds, we develop an $O(1.324719^n|F|)$ algorithm for the case when every variable in the formula appears at most three times. (Due to page limitations, this algorithm appears only in the technical report version[2] of the paper.) This bound significantly improves the bound of $O(\sqrt{3}^n|F|)$ due to Raman, Ravikumar and Srinivasa Rao [14] mentioned above. This improved algorithm for the special case is also used later in our main algorithms.

We remark that all our algorithms employ some simplification rules as well as the standard Davis-Putnam type branching rules, and involve an extensive case analysis as is typically the case with most of the exact algorithms refered earlier [1,10,9,7,3,11]. Further we emphasize that throughout the development of algorithms for these problems, even small improvements in the second or even the third digit after the decimal in the exponent can mean significant progress.

In the next section, we give some notations and definitions used in our algorithms. Section 3 gives some simplification or reduction rules. Section 4 gives some branching rules which will be later employed in the main algorithms. In Section 5, two new algorithms are proposed for the general MAXSAT problem. The exponent in the time bound of the first one is the number of clauses and that of the second one is the length of the formula.

## 2 Notation

Let $l$ be a literal in a formula $F$. We will call it an $(i,j)[p_1,p_2,\ldots,p_i][n_1,n_2,\ldots,n_j]$ literal if it occurs $i$ times positively and $j$ times negatively and the clauses containing $l$ are of length $p_1 \leq p_2 \leq \ldots \leq p_i$ and those containing $\bar{l}$ are of length $n_1 \leq n_2 \leq \ldots \leq n_j$. If the length of the clauses is not important we will simply call it an $(i,j)$ literal. If we say that some variable or literal occurs $i^+$ times, we will mean that it occurs at least $i$ times. Similarly $i^-$ denotes at most $i$ occurrences. For a variable $x$, we say $\tilde{x}$ occurs in a clause $C$ if $x \in C$ or $\bar{x} \in C$. We call $x$ a $k$-variable, if $x$ is some $(i,j)$ literal such that $i+j=k$.

If a literal $x$ occurs as a unit clause $\{x\}$, $k$ times in the formula, we will denote the fact by $n_u(x)=k$. For a literal $x$, $l(x)$ will denote the sum of the lengths of the clauses containing $x$.

If $F$ is a formula, $F[p_1,\ldots,p_k][n_1,\ldots,n_l]$ will denote the formula obtained by putting the literals $p_1=\cdots=p_k=true$ and $n_1=\cdots=n_l=false$. If $C$ is a clause then $F[][C]$ will denote the formula obtained by putting all the literals in $C$ as false.

Our algorithms for MAXSAT first go through some simplification and reduction rules where the given formula is simplified. On the reduced formula, we then apply some branching rules. If a branching rule for the formula $F$ branches as $F[X_1][Y_1]$, $F[X_2][Y_2]\ldots,F[X_i][Y_i]$ it means that each of the formulae $F[X_j][Y_j], j=1$ to $i$ is solved (recursively or otherwise) and the assignment that satisfies the maximum number (or the required number, in the parameterized problem) of clauses among them is returned. The branching rule ensures that the returned assignment actually satisfies the maximum (or the required) number of clauses of $F$. If the question to be answered is a decision question (like "can $k$ clauses be satisfied?"), then the answer returned is 'yes' if any of the branches returns 'yes' and 'no' otherwise. If the "size" of $F$ is $q$ and the sizes of $F[X_j][Y_j] \leq q-r_j, \forall j$, where "size" is the number of variables, the number of clauses or the number of clauses to be satisfied, as the case may be, then such a branching rule is said to have a *branching vector* $(r_1,r_2,\ldots,r_i)$.

If two clauses $C_1$ and $C_2$ contain some pair of complementary variables, we will denote the fact by $Co(C_1,C_2)$. A subset of clauses is said to be *closed* if no variable in this subset occurs outside this subset in the rest of the formula. For a clause $C$, $C-\{a_1,a_2,\ldots,a_k\}$ will denote the clause obtained by deleting all instances of the literals $a_1,a_2,\ldots,a_k$ from the clause $C$. We will denote the number of clauses in a formula $F$, by $m(F)$. The maximum number of clauses that are satisfied by an optimal assignment will be denoted by $opt(F)$ .

## 3 Simplification Rules

We list a set of rules to reduce a given CNF formula so that solving the MAXSAT problem for the original formula is equivalent to solving it for the reduced one. First we assume that the formula has no empty clause (which is vacuously false), and that every variable appears both positively and negatively - i.e. there are

no $(i,0)$ or $(0,i)$ variables as they can be eliminated easily by assigning them an appropriate value.

1. **Elimination of** $(1,1)$ **literals:** If $a$ is a $(1,1)$ literal, and $a \in C_1$ and $\bar{a} \in C_2$, then if $Co(C_1 - \{a\}, C_2 - \{\bar{a}\})$ just remove the clauses. Otherwise, replace them by a single clause $\{C_1 \cup C_2 - \{a, \bar{a}\}\}$.
2. **Replacement of almost common clauses:** If $\exists$ clauses $C_1$ and $C_2$ and a literal $a$, such that $C_1 - \{a\} = C_2 - \{\bar{a}\}$, then replace $C_1$ and $C_2$ by the clause $\{C_1 \cup C_2 - \{a, \bar{a}\}\}$.
   In particular this implies that we cannot have $\{a\}$ and $\{\bar{a}\}$ together as unit clauses in the same formula.

In both these rules, if $F'$ is obtained from $F$ by application of the rule, clearly $opt(F) - opt(F') = m(F) - m(F')$ and this justifies the rules.

Note that each simplification rule can be applied for a formula $F$ in $O(|F|)$ time. Further after all simplification rules are applied, every (remaining) variable appears both positively and negatively and each variable appears at least three times in the formula. Also if a variable appears as a unit clause, all its occurrences in the unit clauses are either all pure or all negated.

## 4 Branching Rules

In this section, we give some rules to abandon certain branches of the davis-putnam type branching algorithm. The reason for abandoning them (which we may not specifically mention in some rules due to page limitation) is that in one of the branches followed, the number of clauses satisfied is at least as many as the maximum number of clauses satisfied in any of the branches abandoned.

1. If $x$ is a $(k,l)$ literal, and $n_u(\bar{x}) \geq k$, then branch as $F[][x]$.
2. If $x$ is a $(k,l)$ literal, and $n_u(\bar{x}) = k-1$, let $C_1, \ldots, C_k$ denote the clauses containing $x$. If $\exists\ i,j$ such that $Co(C_i, C_j)$, then branch as $F[][x]$ (as at least one of $C_i$ and $C_j$ is always satisfiable). Otherwise, branch as $F[][x]$ and $F[x][C_1 \cup \ldots \cup C_k - \{x\}]$.

3. Branching rules for $(1,k)$ literals.
   a) If $x$ is a $(1,k)[\ldots][1,\ldots]$ literal, then branch as $F[][x]$. This is a special case of rule 1.
   b) If $x$ is a $(1,k)[2^+][\ldots]$ literal, then if $C$ is the clause containing $x$, branch as $F[][x]$ and $F[x][C - \{x\}]$. This is a special case of rule 2*b*.
   c) If $x$ is a $(1,k)[\ldots][2,\ldots]$ literal, then if $C$ is the clause of size 2 containing $\bar{x}$, branch as $F[][x]$ and $F[x, C - \{\bar{x}\}][]$. This gives a $(k,2)$ branch.
4. Let $x$ be a $(k,l)$ literal and let $C_1, \ldots, C_k$ be the clauses containing $x$. If there is a variable $\tilde{y}$ such that there are only one or two occurrences of $\tilde{y}$ in clauses other than $C_1, \ldots, C_k$, then branch as $F[x][]$ and $F[][x]$. This gives a $(k+1,l)$ branching vector. For, setting $x = true$ and applying the reduction rule will eliminate $\tilde{y}$, thus satisfying at least $k+1$ clauses.
5. If $x$ is a $(2,2)$ literal, and $C_1$ and $C_2$ are the clauses containing $\bar{x}$. Branch as $F[x][], F[][x, C_1 - \{\bar{x}\}]$ and $F[][x, C_2 - \{\bar{x}\}]$.

Since, if there is an optimum assignment with $x = false$ and $C_1 - \{\bar{x}\} = true$ and $C_2 - \{\bar{x}\} = true$, then we might as well set $x = true$.

6. Let $x$ be a $(1,3)[1][\ldots]$ literal and $C_1, C_2$ and $C_3$ are the clauses containing $\bar{x}$. Then, if there exist some two clauses, say $C_2$ and $C_3$ which do not contain any pair of complementary variables, it is sufficient to branch as $F[x][], F[][x, C_1 - \{\bar{x}\}]$ and $F[][x, C_2 - \{\bar{x}\}, C_3 - \{\bar{x}\}]$. Otherwise, branch as $F[x][]$.
Since if $Co(C_1, C_2)$, $Co(C_1, C_3)$ and $Co(C_2, C_3)$, then it can be seen that for any assignment at least two of the clauses in $C_1, C_2$ and $C_3$ are satisfied. Otherwise, if there is an optimal solution with $x = false$ such that at least two of $C_1 - \{\bar{x}\}, C_2 - \{\bar{x}\}$ and $C_3 - \{\bar{x}\}$ are true, then setting $x = true$ does not decrease the number of clauses satisfied.
7. If $x$ is a $(1,2)$ literal, and $C_1$ and $C_2$ are the clauses containing $\bar{x}$. Branch as $F[x][], F[][x, C_1 - \{\bar{x}\}, C_2 - \{\bar{x}\}]$.

## 5 Algorithms for General MAXSAT

In this section we present two algorithms for the general MAXSAT. One has the number of clauses as the exponent in the running time and the other has the length of the formula as the exponent in the running time. As in the previous section, we will first employ the reduction rules as far as possible and work with the reduced formula. It is also assumed that after applying every branching rule, any applicable reduction rules are applied.

### 5.1 A Bound With Respect to the Number of Clauses

We note that if $\{x\}$ is a unit clause then whether $x$ is set *true* or *false*, the clause will be eliminated. This means that if $x$ is a $(k, l)$ literal, such that $n_u(x) = i$ (recall that $n_u(x)$ is the number of times $x$ occurs as a unit clause in the formula) and $n_u(\bar{x}) = j$, then branching as $F[x][]$ and $F[][x]$ gives us a $(k + j, l + i)$ branch for the non-parameterized case (number of clauses eliminated), and a $(k, l)$ branch for the parameterized case (number of clauses satisfied). In the following if we say that a branch is $(r, s)$ for the parameterized case, we mean that in one branch $r$ clauses are satisfied and in the other $s$ clauses. Similarly an $(r, s)$ branch for the non-parameterized case is a branch where $r$ clauses are eliminated in one and $s$ in the other. Note also that an $(r, s)$ branch for the parameterized case is also an $(r, s)$ branch for the non-parameterized case.

**Algorithm:**

1. If there is some $(1, 5+)$ or a $(2+, 3+)$ literal $x$, branch as $F[x][]$ and $F[][x]$. This gives *good* branches for the parameterized question (and hence for the general MAXSAT as well).
2. If there is some $(1, 3+)[2+][\ldots]$ literal $x$, branch as $F[][x]$ and $F[x][\{C - \{x\}\}]$, where $C$ is the clause containing $x$. This is a $(3, 2)$ branch for the parameterized case. This is essentially branching rule 3b.

3. If there is some $(1,4)[1][\ldots]$ literal $x$, branch as $F[x][]$ and $F[][x]$. Clearly, a $(1,5)$ branch for the non-parameterized case and a $(1,4)$ branch for the parameterized case.
4. If there is some $(1,3)[1][2,\ldots]$ literal $x$, let $C$ denote the length 2 clause containing $\bar{x}$ (choose any one if many). Branch as $F[][x]$ and $F[x,\{C-\{\bar{x}\}\}][]$. Branching rule 3*c*. Clearly a $(3,2)$ branch for the parameterized case.
5. If there is some $(2,2)[1,\ldots][\ldots]$ literal $x$. According to branching rule 2, we get that, if $C_1$ and $C_2$ are the clauses containing $\bar{x}$, branch as $F[x][]$ if $Co(C_1,C_2)$. Otherwise branch as $F[x][]$ and $F[][x,C_1\cup C_2-\{\bar{x}\}]$. Clearly a $(2,3)$ branch for the parameterized case, since $|C_1\cup C_2-\{\bar{x}\}|\geq 1$.
   After the above rules can no longer be applied, the formula just contains $(1,3)[1][3^+,3^+,3^+]$ literals and their complements, $(2,2)[2^+,2^+][2^+,2^+]$ literals and *3-variables*.
6. If $\tilde{y}$ is a *3-variable* which occurs in a clause containing a (2,2) literal $x$, then branch as $F[x][]$ and $F[][x]$. Thus, branching rule 4 gives a $(2,3)$ branch for the parameterized case.
7. If $\tilde{y}$ is a *3-variable* which occurs in a clause containing a (3,1)[...][1] literal $x$, then if $\tilde{y}$ occurs in all the 3 clauses containing $x$, branch as $F[][x]$. This is because $y$ can be assigned so that at least 2 of the 3 clauses containing $x$ are satisfied with $x=false$. Otherwise, branch as $F[x][]$ and $F[][x]$. This gives a $(1,5)$ branch for non-parameterized case and a $(1,4)$ branch for the parameterized case.
   Now any clause containing a *4-variable* contains only *4-variables*.
   So the subset of clauses containing *3-variable* is closed. This can be eliminated efficiently using the algorithm alluded to in the Introduction[2]. Thus the formula now contains only *4-variables*.
8. If $x$ is a $(2,2)$ literal and some $\tilde{y}$ occurs in the two clauses containing $\bar{x}$ or in the two clauses containing $x$ , then branch as $F[x][]$ and $F[][x]$. A $(2,3)$ branch for the parameterized case.
9. If $x$ is a $(1,3)$ literal and any $\tilde{y}$ occurs in two or more clauses containing $\bar{x}$ then branch as $F[][x]$ and $F[x][]$. Thus, branching rule 4 gives a $(1,4)$ branch and a $(1,5)$ branch for the parameterized case and the non-parameterized case respectively.

At this state, note that if $x$ is a $(1,3)$ literal, and $C_1,C_2$ and $C_3$ are the clauses containing $\bar{x}$, then except for $\bar{x}$ they all have disjoint variables. Also, let $a_1$ be a $(2,2)$ literal, and $C_1$ and $C_2$ be the clauses containing $\bar{a_1}$. Let $a_2$ be any other literal, then after steps 8 and 9 of the algorithm at most one of $C_1$ and $C_2$ can contain $\bar{a_2}$.
This observation gives us the following lemma.

**Lemma 1.** *Let $a_1,a_2,\ldots,a_l$ be $(2,2)$ literals and let $C_1,\ldots,C_k$ be precisely the clauses that contain some $\bar{a_i}$, $1\leq i\leq l$. After step 9 of the algorithm $k\geq T(l)$*
*where $T(l)=\min\max\{\lceil\frac{2l}{j}\rceil,j+1\}$, where the minimum is taken over all $1\leq j\leq l$. In particular $k\geq\lceil\sqrt{(2l+\frac{1}{4})}+\frac{1}{2}\rceil$.*

*Proof.* Without loss of generality, let $C_1$ be the clause such that $\{C_1 \cap \{a_1, a_2, \ldots, a_l\}\}$ has maximum cardinality $j$. Then the number of clauses needed is at least $\lceil \frac{2l}{j} \rceil$. Also by the condition imposed by step 9 of the algorithm, each of the other occurrences of the $j$ variables must occur in different clauses, thus at least $j$ additional clauses are needed. Thus number of clauses $k$ will be at least $T(l) = \min_j \max\{\lceil \frac{2l}{j} \rceil, j+1\}$. It can be verified that $T(l) \geq \lceil \sqrt{(2l + \frac{1}{4})} + \frac{1}{2} \rceil$.

10. Let $x$ be a $(1,3)[1][\ldots]$ literal and let $C_1, C_2$ and $C_3$ be the clauses containing $\bar{x}$. If $C_1$ contains a $(2,2)$ literal $a$ or has size 4 or more then branch as $F[x][], F[][x, C_1 - \{\bar{x}\}], F[][x, C_2 - \{\bar{x}\}, C_3 - \{\bar{x}\}]$. This is essentially branching rule 5*b*.
    a) If $y$ is a $(1,3)$ literal such that $\tilde{y} \in C_i$, then it is $\bar{y}$ that belongs to $C_i$. Since all $(1,3)$ literals are $(1,3)[1][\ldots]$ literals.
    b) If there are $p$ $(3,1)$ literals $u_1, \ldots, u_p$ and $q$ $(2,2)$ literals $v_1, v_2, \ldots, v_q$ in $C_1 - \{\bar{x}\}$, then setting $C_1 - \{\bar{x}\} = false$ eliminates at least $p + T(q)$ clauses, other than $C_1, C_2$ and $C_3$. This is because, as $\bar{u_i}'s$ are present as unit clauses, setting $u_i's = false$ eliminates at least $p$ clauses and since the $\tilde{v_i}'s$ will occur in clauses other than $C_2$ and $C_3$ (after step 9 of algorithm). setting $v_i's = false$ will eliminate another $T(q)$ clauses.

    $F[][x, \{C_1 - \{\bar{x}\}\}]$ satisfies at least 3 clauses other than $C_1, C_2$ and $C_3$, since $p + T(q) \geq 3$ for $\{C_1 - \{\bar{x}\}\}$. Thus 6 clauses are satisfied and 7 are eliminated by the assignment.

    Similarly, since $\{C_2 - \{\bar{x}\}\}$ and $\{C_3 - \{\bar{x}\}\}]$ have no common variable, the assignment $F[][x, \{C_2 - \{\bar{x}\}\}, \{C_3 - \{\bar{x}\}\}]$ satisfies 4 clauses other than $C_1, C_2$ and $C_3$. Thus 7 clauses are satisfied and 8 are eliminated by the assignment. Thus we have a $(1, 7, 8)$ branch for the non-parameterized case and a $(1, 6, 7)$ branch for the parameterized case.

    Now the clauses containing $(1,3)[1][3,3,3]$ literals and their complements form a closed subformula $F_1$. The remaining clauses form a closed subformula $F_2$ of $(2,2)$ literals. So, $F = F_1 \wedge F_2$. Rule 11 describes how to eliminate $F_1$ efficiently.
11. If $a$ is a $(1,3)$ literal, then the clauses with $\tilde{a}$ will be $\{a\}, \{\bar{a}, b, c\}, \{\bar{a}, \ldots\}, \{\bar{a}, \ldots\}$. We branch as $F[][a], F[a, b][]$ and $F[a, c][b]$.

    We know that $b$ and $c$ will be $(3,1)[3,3,3][1]$ literals, if $a$ is true and both $b$ and $c$ are false, then $a$ can be set to $false$. For the parameterized case, $F[][a]$ and $F[a, b][]$ satisfies 3 and 4 clauses respectively, $F[a, c][b]$ satisfies 5 clauses (the unit clauses $\{a\}, \{\bar{b}\}$ and the 3 clauses containing $c$). Thus we have a $(3, 4, 5)$ branch for the parameterized case. Similarly we have a $(4, 5, 6)$ branch for the non-parameterized case.
12. If $x$ is a $(2,2)[2^+, 2^+][3^+, 3^+]$ literal, let $C_1$ and $C_2$ be the clauses containing $\bar{x}$. Apply branching rule 5*a* and branch as $F[x][], F[][x, \{C_1 - \{\bar{x}\}\}]$ and $F[][x, \{C_2 - \{\bar{x}\}\}]$.

    The second and the third branch will eliminate at least $2 + T(2) = 5$ clauses each. $F[x][]$ eliminates 2 clauses and also leads to $3 - variables$ in the formula; now either these variables would occur as a closed subformula or in a clause

with some $(2,2)$ literal. Thus the next step would at least be a $(2,3)$ branch or better, since steps 6 of this algorithm and branching rules for $(n,3)$ MAXSAT are all *good* for the parameterized case. Thus we have a $(4,5,5,5)$ branch for the parameterized case.

13. Any $4-variable$ $a$ will be of the type $(2,2)[2,2^+][2,2^+]$. Let $C_1 = \{a,b\}$ be the clause of size 2 containing $a$. We will show how to branch with $a = false$. The case with $a = true$ is similar.
    a) If $\tilde{b}$ is present in some clause containing $\bar{a}$. Then just branch as $F[b][a]$. By rule 8 of this algorithm, it must be that $\bar{b}$ occurs in the clause with $\bar{a}$; thus setting $a = false$ the clauses containing $\tilde{b}$ will reduce to $\{b\}, \{b,\ldots\}$ and $\{\bar{b},\ldots\}$; thus setting $b = true$ is sufficient, and this is essentially branching rule $3a$.
    b) Otherwise, If $C_1'$ and $C_2'$ are the clauses containing $\bar{b}$, then branch as $F[b][a]$ and $F[][a,b,C_1' \cup C_2' - \bar{b}]$. Clearly $F[b][a]$ eliminates 4 clauses, and the second branch eliminates the clauses containing $\bar{b}$, the unit clause $\{b\}$ and another $3(= T(2))$ clauses corresponding to $C_1' \cup C_2' - \bar{b}$. So at least 6 clauses are eliminated in the non-parameterized case and at least 5 clauses in the parameterized case.
    Thus in the worst case the branch is a $(4,6,4,6)$ branch for the non-parameterized case and a $(4,5,4,5)$ branch for the parameterized case.

For the non-parameterized case, the branch $(4,5,5,5)$ is the most restrictive giving an $O(1.341294^m|F|)$ algorithm for MAXSAT by solving the appropriate recurrence relation.

For the parameterized case, the branch $(1,4)$ is the most restrictive giving an $O(1.380278^k k^2 + |F|)$ algorithm by reducing the problem to the kernel and applying the above branching algorithm.

Thus we have proved the following.

**Theorem 1.** *For a formula $F$ in CNF on $n$ variables and $m$ clauses, an assignment satisfying the maximum number of clauses can be found in time $O(1.341294^m|F|)$. Also an assignment satisfying at least $k$ clauses of $F$, if exists, can be found in time $O(k^2 1.380278^k + |F|)$.*

### 5.2 A Bound With Respect to the Length of the Formula

In this section, we call a branching vector *good* if either it is a $(4,11), (5,10), (6,8)$ or a $(7,7)$ vector.

We observe that if $x$ is a $(p,q)$ literal, setting $x = true$ reduces the length of the formula by at least $l(x) + q$ (recall that $l(x)$ is the sum of the lengths of the clauses containing $x$.) Also since every variable $\tilde{x}$ is a $3^+ - variable$ after applying the reduction rules, setting a value to $x$ reduces the length of the formula by at least 3.

Using these facts and observing that both $x$ and $\bar{x}$ cannot be present as unit clauses together, simply branching as $F[x][]$ and $F[][x]$ gives the recurrence $T(|F|) \leq T(|F|-3) + T(|F|-4) + |F|$ for the running time. This gives a very simple $O(1.220745^{|F|}|F|)$ algorithm.

In what follows, we argue through a number of cases to obtain a more efficient algorithm.

**Algorithm**

1. If $\tilde{x}$ is a $7^+ - variable$, then branching as $F[x][]$ and $F[][x]$ gives a good branch.
2. If $x$ is a $(3,3)$ literal, then either $l(x)$ or $l(\bar{x}) \geq 6$, since only one of $x$ and $\bar{x}$ can appear in a unit clause. Thus, branching as $F[x][]$ and $F[][x]$ gives us a good branch.
   Now the formula contains only $(1,k)$ literals, for $k \leq 5$ or $(2,k)$ literals, for $k \leq 4$.
3. If $x$ is a $(1,k)[\ldots][1,\ldots]$ literal, then just branch as $F[][x]$. This is from branching rule $3a$.
4. The following rules give *good* branches for $k \geq 3$.
   a) If $x$ is a $(1,k)[2^+][\ldots]$ literal, then just branch as $F[][x]$ and $F[x][C-\{x\}]$ where $C$ is the $2^+$-clause containing $x$. This is branching rule $3b$ and this gives a $(2k+1, k+4)$ branch.
   b) If $x$ is a $(1,k)[1][2,\ldots]$ literal, then just branch as $F[][x]$ and $F[x, C-\{\bar{x}\}][]$. $C$ is the clause containing $\bar{x}$ of size 2. This is essentially branching rule $3c$. Clearly this gives us a $(2k+1, k+4)$ branch.
   c) If $x$ is a $(1,k)[1][3^+,\ldots]$ literal, and $l(\bar{x}) \geq 10$, then just branch as $F[][x]$ and $F[x][]$. This gives a $(k+1, l(\bar{x})+1)$ branch.

   So now if $x$ is a $(1,k)$ literal, it has to be a $(1,3)[1][3,3,3]$ literal or a $(1,2)$ literal.
5. If $x$ is a $(2,k)[\ldots][1,1,\ldots]$ literal then, we branch as $F[][x]$. This is simply the branching rule 1.
6. The following rules for eliminating $(2,k)$ literals, give *good* branches for $k \geq 3$.
   a) If $x$ is a $(2,k)[\ldots][1,2^+,\ldots]$ literal then, we branch as $F[x][]$ and $F[][x]$. This gives a $(4+k, 2k+1)$ branch, since $x$ cannot appear in a unit clause.
   b) If $x$ is a $(2,k)[\ldots][2^+,\ldots]$ and $l(x) \geq 3$, we branch as $F[x][]$ and $F[][x]$. This gives a $(3+k, 2k+2)$ branch.
7. If $x$ is a $(2,k)[1,1][2^+,\ldots]$ literal, branching as $F[x][]$ and $F[][x]$ gives a $(2+k, 2k+2)$ branch, which is *good* for $k \geq 4$.
8. If $x$ is a $(2,3)[1,1][2^+,\ldots]$ literal, let $C_1, C_2$ and $C_3$ denote the clauses containing $\bar{x}$,
   a) If $Co(C_i, C_j)$ for some $1 \leq i, j \leq 3$, branch as $F[x][]$.
   b) If $C_1, C_2$ and $C_3$ contain only one variable other than $\bar{x}$, then $C_1 = C_2 = C_3 = \{\bar{x}, c\}$, thus we branch as $F[x, c][]$ and $F[][x]$, which is at least an $(8,8)$ branch. This is because since $c$ will be almost a $6-variable$, setting $x = true$ makes $c$ a $(3^+, 3^-)[1,1,1,\ldots][\ldots]$ literal, and so $c = true$ in an optimal solution.
   c) If $C_1, C_2$ and $C_3$ contain 2 or more variables other than $\bar{x}$, then branching as $F[x][]$ and $F[][x, C_1 \cup C_2 \cup C_3 - \{\bar{x}\}]$ gives a $(5,11)$ branch.

   Thus the formula contains only $3-variables$,$(2,2)$ literals and $(1,3)[1][3,3,3]$ literals.

a) The rules below eliminate $(1,3)[1][3,3,3]$ literals. If $x$ is a $(1,k)[1][3,3,3]$, and some clause containing $\bar{x}$ contains a $3-variable\ \tilde{y}$, then if all the 3 occurrences of $\tilde{y}$ are in clauses containing $\bar{x}$ then, just branch as $F[x][]$. Otherwise branching as $F[x][]$ and $F[][x]$ gives a $(4,11)$ branch.

b) If $x$ is a $(1,3)[1][3,3,3]$, and some clause containing $\bar{x}$ contains two or more occurrences of a $4-variable\ \tilde{y}$, then branching as $F[x][]$ and $F[][x]$ gives a $(4,11)$ branch.

c) Now, if $x$ is a $(1,3)[1][3,3,3]$ literal, then the clauses containing $\bar{x}$ contain single occurrences of $4-variables$ other than $\bar{x}$. We show that branching according to the rule $5b$, gives a $(4,16,22)$ branch. Clearly $F[x]$ reduces the length by 4, $F[][x]$ reduces the length of the formula by 10, now in F[][x], $|C_1| = 2$ and $|C_2 \cup C_3| = 4$ and since the variables in $C_1 \cup C_2 \cup C_3$ are $3-variables$, we reduce the length of the formula by at least 3 for each assignment of these.

The formula contains only $(2,2)$ literals and $3-variables$.

9. If $x$ is a $(2,2)[3^+,3^+][2^+,2^+]$ literal or a $(2,2)[2,3^+][2,3^+]$ literal, branching as $F[x][]$ and $F[][x]$, gives a *good* branch.
Thus the formula just contains $(2,2)[2^-,2^-][2^-,2^-]$ literals.
10. If $x$ is a $(2,2)[1,2][2,2]$ literal. Let $C$ denote the $2-clause$ containing $x$ and $D_1, D_2$ denote the clauses containing $\bar{x}$.
    a) If $C \cup D_1 \cup D_2 - \{x, \bar{x}\}$ contains only one variable, then the clauses containing $\tilde{x}$, must be $\{x\}, \{x,c\}, \{\bar{x}, \bar{c}\}$ and $\{\bar{x}, \bar{c}\}$. We branch as $F[x][c]$, it can be seen that this is a valid branch as $c$ is a $3-variable$ or a $(2,2)$ literal.
    b) If $C \cup D_1 \cup D_2 - \{x, \bar{x}\}$ contains three variables. Branching as $F[x][]$ and $F[C-x][x, D_1 \cup D_2 - \{\bar{x}\}]$, we get a $(5,13)$ branch. This is because $F[x][]$ reduces the length by 5, and assignment to $x$ reduces the length by at least 4 and assignment to each of the other three variables reduces the length by at least 3.
    c) If $C \cup D_1 \cup D_2 - \{x, \bar{x}\}$ contains two variables then the clauses containing $\tilde{x}$ can be
        i. $\{x\}, \{x,b\}, \{\bar{x}, \bar{b}\}$ and $\{\bar{x}, c\}$. Branching as $F[x][]$ and $F[C-x][x, D_1 \cup D_2 - \{\bar{x}\}]$, we get a $(7,10)$ branch. For if $b$ is a $3-variable$, then it will be eliminated by setting $x = true$. Otherwise if $b$ is a $(2,2)$ literal, then the clauses containing $\tilde{b}$ will be $\{\bar{b}\}, \{b, \ldots\}$ and $\{\bar{b}, \ldots\}$, which can be directly eliminated by setting $b = false$.
        ii. $\{x\}, \{x,b\}, \{\bar{x}, c\}$ and $\{\bar{x}, c\}$. Branching as $F[x][]$ and $F[C-x][x, D_1 \cup D_2 - \{\bar{x}\}]$, we get a $(7,10)$ branch, since setting $x = true$, then $D_1, D_2$ will become unit clauses $\{c\}, \{c\}$, for which we will only have one way branch $c = true$.
11. If $x$ is a $(2,2)[2,2][2,2]$ literal, then
    a) if some clause containing $x$ contains a $3-variable\ \tilde{y}$, then if number of occurrences of $\tilde{y}$ in the clauses containing $x$ is 1, then setting $x = true$ reduces the length of the formula by at least 8. So $F[x][]$ and $F[][x]$ gives a $(6,8)$ branch. Otherwise, if the number of occurrences of $\tilde{y}$ in

the clauses containing $x$ is 2 then setting $x = false$ gives 2 unit clauses containing $\tilde{y}$, which can be eliminated directly. So $F[x][]$ and $F[][x]$ gives a $(6, 8)$ branch.

b) Otherwise branch as $F[x][]$ and $F[][x]$ and apply rule 11 of this algorithm for each instance. Setting $x = true$ or $x = false$ reduces the length of the formula by 6, but this leads to a $(2, 2)[1, \ldots][2, 2]$ literal. So, we either have a $(7, 10)$ branch in the next step or a $(5, 13)$ branch. Thus we a $(13, 16, 13, 16)$ branch or a $(11, 19, 11, 19)$ branch in the worst case.

12. The formula now contains only $3 - variables$, we now apply the algorithm for $(n, 3)$ MAXSAT alluded to in the Introduction[2]. Since the length of the formula $l$, will be equal to $3n$ (since $2^- - variables$ do not exist in the formula), we can eliminate the formula in $O(1.324719^{\frac{l}{3}}|F|)$ time.
The branch $(4, 11)$ is the most restrictive thus giving an $O(1.105729^{|F|}|F|)$ algorithm by solving the appropriate recurrence relation.

Thus we have proved

**Theorem 2.** *For a formula $F$ in conjunctive normal form, an assignment satisfying the maximum number of clauses can be found in time $O(1.105729^{|F|}|F|)$.*

## 6 Conclusions

It would be interesting to see how our algorithms perform in practice. Also it would be useful to identify some general branching techniques which would reduce the number of cases in the algorithms.

## References


1. R. Balasubramanian, M. R. Fellows and V. Raman, 'An Improved fixed parameter algorithm for vertex cover' *Information Processing Letters* , **65** (3):163-168, 1998.
2. N. Bansal and V. Raman, 'Upper Bounds for MaxSat: Further Improved', Technical Report of the Institute of Mathematical Sciences, IMSc preprint 99/08/30.
3. R. Beigel and D. Eppstein, '3-coloring in time $O(1.3446^n)$: A no-MIS Algorithm', In Proc of IEEE Foundations of Computer Science (1995) 444-452.
4. E. Dantsin, M. R. Gavrilovich E. A. Hirsch and B. Konev, 'Approximation algorithms for MAX SAT: a better performance ratio at the cost of a longer running time', PDMI preprint 14/1998, Stekolov Institute of Mathematics at St. Petersburg, 1998.
5. R. G. Downey and M. R. Fellows. Parameterized Complexity. Springer-Verlag, November 1998.
6. E.A. Hirsch, 'Two new upper bounds for SAT', Proc. of 9th Symposium on Discrete Algorithms, (1998) 521-530.
7. O. Kullmann, 'Worst-case analysis, 3-SAT decision and lower bounds: approaches for improved SAT algorithms', DIMACS Proc. SAT Workshop 1996, AMS, 1996.
8. M. Mahajan and V. Raman, 'Parameterizing above guaranteed values: MaxSat and MaxCut'. Technical Report TR97-033, ECCC Trier, 1997. To appear in Journal of Algorithms.

9. B. Monien, E. Speckenmeyer, 'Solving Satisfiability in less than $2^n$ steps', *Discrete Applied Mathematics* , **10** (1985) 287-295.
10. R. Niedermeier and P. Rossmanith, 'Upper bounds for Vertex Cover: Further Improved'. in Symposum on Thoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science, Springer Verlag. March (1999).
11. R. Niedermeier and P. Rossmanith, 'New Upper Bounds for MAXSAT', International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, Springer Verlag (1999).
12. R. Paturi, P. Pudlak, M. Saks, and F. Zane. 'An improved exponential-time algorithm for -SAT', In Proc. of 39th IEEE Foundations of Computer Science (1998).
13. P. Pudlak, 'Satisfiability-algorithms and logic', In Proc. of 23rd Conference on Mathematical Foundations of Computer Science, **1450** in Lecture Notes in Computer Science Springer Verlag, August (1998) 129-141.
14. V. Raman, B. Ravikumar and S. Srinivasa Rao, 'A Simplified NP-Complete MAXSAT problem' *Information Processing Letters*, **65**, (1998) 1-6.
15. J. M. Robson, 'Algorithms for the Maximum Independent Sets'. *Journal of Algorithms* **7**, (1986) 425-440.
16. I. Schiermeyer, 'Solving 3-Satisfiability in less than $1.579^n$ steps', Lecture Notes in Computer Science, Springer Verlag **702**, (1993) 379-394.

# A Linear Time Algorithm for Recognizing Regular Boolean Functions

Kazuhisa Makino

Department of Systems and Human Science, Graduate School of Engineering Science, Osaka University, Toyonaka, Osaka, 560, Japan.
(makino@sys.es.osaka-u.ac.jp)

**Abstract.** A positive (or monotone) Boolean function is regular if its variables are naturally ordered, left to right, by decreasing strength, so that shifting the non-zero component of any true vector to the left always yields another true vector. In this paper, we propose a simple linear time algorithm to recognize whether a positive function is regular.

## 1 Introduction

A *Boolean function*, or a *function* in short, is a mapping $f : \{0,1\}^n \mapsto \{0,1\}$, where $v \in \{0,1\}^n$ is called a *Boolean vector* (a *vector* in short). Let $V = \{1, 2, \ldots, n\}$. For a pair of vectors $v, w \in \{0,1\}^n$, we write $v \leq w$ if $v_j \leq w_j$ holds for all $j \in V$, and $v < w$ if $v \leq w$ and $v \neq w$, where we define $0 < 1$. If $f(v) = 1$ (resp., 0), then $v$ is called a *true* (resp., *false*) vector of $f$. The set of all true vectors (resp., false vectors) of $f$ is denoted by $T(f)$ (resp., $F(f)$). We use notations $ON(v) = \{j \,|\, v_j = 1,\ j \in V\}$ and $OFF(v) = \{j \,|\, v_j = 0,\ j \in V\}$. A function $f$ is *positive* if $v \leq w$ always implies $f(v) \leq f(w)$. A positive function is also said to be *monotone*. A true vector $v$ of $f$ is *minimal* if there is no other true vector $w$ such that $w < v$. Let $\min T(f)$ denote the set of all minimal true vectors of $f$.

If $f$ is positive, it is known that $f$ has a unique minimal disjunctive normal form (DNF), consisting of all prime implicants. There is a one-to-one correspondence between prime implicants and minimal true vectors. For example, a positive function $f = x_1x_2 \vee x_2x_3 \vee x_3x_1$, has prime implicants $x_1x_2$, $x_2x_3$, and $x_3x_1$ which correspond to minimal true vectors (110), (011), and (101), respectively. Thus, the input length to describe a positive function $f$ is $O(n|\min T(f)|)$ if it is represented in this manner.

A positive function $f$ is said to be *regular* if, for every $v \in \{0,1\}^n$ and every pair $(i, j)$ with $i < j$, $v_i = 0$ and $v_j = 1$, the following condition holds:

$$f(v) \leq f(v + e^{(i)} - e^{(j)}), \tag{1}$$

where $e^{(k)}$ denotes the unit vector which has a 1 in its $k$-th position and 0 in all other positions. A positive function $f$ is called 2-*monotonic* if there exists a linear ordering on $V$, for which $f$ is regular. Let $\mathcal{C}_R$ and $\mathcal{C}_{2M}$, respectively, denote the classes of regular and 2-monotonic functions.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 259–268, 1999.
© Springer-Verlag Berlin Heidelberg 1999

The 2-monotonicity and related concepts have been studied in various contexts in fields such as threshold logic [3,6,12,13], game theory, hypergraph theory [5] and learning theory [4,9,10]. The 2-monotonicity was originally introduced in conjunction with threshold functions (e.g., [12]), where a positive function $f$ is a *threshold* function if there exist $n+1$ nonnegative real numbers $w_1, w_2, \ldots, w_n$ and $t$ such that:

$$f(x) = \begin{cases} 1, & \text{if } \sum w_i x_i \geq t \\ 0, & \text{if } \sum w_i x_i < t. \end{cases}$$

As this $f$ satisfies (1) by permuting variables so that $w_i > w_j$ implies $i < j$, a threshold function is always 2-monotonic, although the converse is not true [12]. Let $\mathcal{C}_{TH}$ denote the class of all threshold functions.

In this paper, we consider the recognition problem for regular and 2-monotonic functions. The recognition problem is defined for a class of positive functions $\mathcal{C}$ as follows:

Problem RECOG($\mathcal{C}$)
Input: $\min T(f)$, where $f$ is a positive function of $n$ variables.
Question: Does $f$ belong to $\mathcal{C}$ ?

The recognition problem is also called the *representation* problem in computational learning theory, and has been studied to prove the hardness of learnability [1]. It is known [1] that, given a polynomially size-bounded and polynomially reasonable class of functions $\mathcal{C}$, if $\mathcal{C}$ is polynomially exactly learnable with membership, equivalence, exhaustiveness, disjointness and superset queries, then RECOG($\mathcal{C}$) is in coNP. This fact is used to show that, if RECOG($\mathcal{C}$) is NP-hard under $\leq_m^p$ reductions (also called many-one reduction), then $\mathcal{C}$ is not polynomially exactly learnable with the above queries unless NP=coNP.

Problem RECOG($\mathcal{C}_{2M}$) has been used as a server to solve RECOG($\mathcal{C}_{TH}$) [13,6,3]. Problem RECOG($\mathcal{C}_{TH}$) is a classical problem in threshold logic [12], and is also called the *threshold synthesis* problem. The polynomial solvability of this classical problem was open for a number of years, until the publication by U. N. Peled and B. Simeone [13]. Their algorithm consists of the following three steps. The first step solves RECOG($\mathcal{C}_{2M}$). If $f$ is not 2-monotonic, then output "No" and halt. Since $\mathcal{C}_{TH} \subset \mathcal{C}_{2M}$, we can conclude that $f$ is not threshold in this case. The second step computes the set of the maximal false vectors $\max F(f)$. If $f$ is 2-monotonic, we can compute $\max F(f)$ in $O(n^2|\min T(f)|)$ time [6,3]. Finally, we check if there is a hyperplane $\sum_i w_i x_i = t$ that separates $\min T(f)$ from $\max F(f)$, which can be solved in polynomial time [8,7].

Besides them, problem RECOG($\mathcal{C}_{2M}$) is important in practical applications, since a number of intractable problems become tractable, if we restrict our attention to 2-monotonic functions. Such examples include set covering problem [3,13], reliability computation [2], dualization problem [3,6,13] and exact learning problem [4,10].

Let us consider the definition of regularity. It is easy to see that it is equivalent to the following: for every $v \in \min T(f)$ and every pair $(i,j)$ with $i < j$, $v_i = 0$

and $v_j = 1$,

$$f(v + e^{(i)} - e^{(j)}) = 1. \tag{2}$$

This implies that problem RECOG($\mathcal{C}_R$) can be solved in $O(n^3|\min T(f)|^2)$ time. J. S. Provan and M. O. Ball [14] have improved it to $O(n^2|\min T(f)|)$ time. Since $|\min T(f)| \gg n$ can be expected in most cases, this is a significant improvement upon the straightforward approach. To achieve the improvement, they make use of two facts. The first is given in the following proposition:

**Proposition 1.** [12] *Let $f$ be a positive function. Then $f$ is regular if and only if, for all pairs of $v \in \min T(f)$ and $i \in V$ with $v_i = 0$ and $v_{i+1} = 1$,*

$$f(v + e^{(i)} - e^{(i+1)}) = 1 \tag{3}$$

*holds.*

This proposition leads to an $O(n^2|\min T(f)|^2)$-time algorithm.

The next one is an $O(n)$-time *membership oracle* (i.e., an algorithm to check if $f(v) = 1$ or not for a given vector $v$) for a regular function $f$. It makes use of a binary tree as a data structure for $\min T(f)$. For instance, Figure 1 shows such a binary tree for a positive function

$$f = x_1x_2 \vee x_1x_3x_4 \vee x_1x_3x_5 \vee x_2x_3x_4 \vee x_2x_3x_5x_6, \tag{4}$$

i.e., $\min T(f) = \{v^{(1)} = (110000), v^{(2)} = (101100), v^{(3)} = (101010), v^{(4)} = (011100), v^{(5)} = (011011)\}$. The importance of their membership oracle is that, even if $f$ is not regular, the answer is correct if it outputs "Yes" (but may not be correct if it outputs "No"). Therefore, for each $v \in \min T(f)$ and $i$ with $v_i = 0$ and $v_{i+1} = 1$, we can check condition (3) in $O(n)$ time. By the above discussion, if the oracle outputs "Yes", then it is correct, i.e., $f(v + e^{(i)} - e^{(i+1)}) = 1$. On the other hand, if the oracle outputs "No", then it may be wrong, but we can conclude that $f$ is not regular (If the answer is correct, $f(v + e^{(i)} - e^{(i+1)}) = 0$, implying that $f$ is not regular; otherwise, $f$ is not regular, since the oracle gives the wrong answer). Hence $O(n^2|\min T(f)|)$ time is achieved.

As for the 2-monotonicity, R. O. Winder shows the following nice proposition.

**Proposition 2.** [15] *Let $f$ be a 2-monotonic positive function of $n$ variables. Define the $n$-dimensional vectors $\alpha^{(j)}$ $(j \in V)$ by*

$$\alpha_k^{(j)} = |\{v \in \min T(f) \,:\, v_j = 1, |ON(v)| = k\}|, \qquad k \in V.$$

*Let $\alpha^{(j_1)} \geq_{LEX} \alpha^{(j_2)} \geq_{LEX} \cdots \geq_{LEX} \alpha^{(j_n)}$, where $\geq_{LEX}$ denotes the lexicographic order between $n$-dimensional vectors, and let $\pi$ be a permutation on $V$ such that $\pi(j_i) = i$ for all $i$. Then $f$ becomes regular by permuting $V$ by $\pi$.*

This proposition says that, in order to check the 2-monotonicity of a positive function $f$, we first permute $V$ by the above $\pi$, and then check the regularity of

the resulting function. Since we can permute $V$ by $\pi$ in $O(n^2+n|\min T(f)|)$ time, the result by J. S. Provan and M. O. Ball also implies an $O(n^2|\min T(f)|)$-time algorithm for checking the 2-monotonicity.

In this paper, we present an $O(n|\min T(f)|)$ time-algorithm for checking the regularity of a positive function $f$. Our algorithm makes use of a *fully condensed binary tree* (*FCB*) as a data structure for $\min T(f)$ (see the definition in Section 2). The algorithm checks condition (3) by finding a *"left-most"* vector $w \in \min T(f)$ such that $w \leq v + e^{(i)} - e^{(i+1)}$. It explores a fully condensed binary tree in the breadth-first fashion. Since the number of nodes in *FCB* is $O(|\min T(f)|)$, our algorithm only requires $O(n|\min T(f)|)$ time.

By combining this with the result by R. O. Winder, we also provide an $O(n(n+|\min T(f)|))$-time algorithm for checking the 2-monotonicity of a positive function $f$.

The rest of this paper is organized as follows. Section 2 introduces three types of binary trees called *ordinary, partially condensed, fully condensed binary trees* as data structures for $\min T(f)$. In Section 3, we give a simple algorithm for RECOG($\mathcal{C}_R$). The algorithm uses a partially condensed tree, and its running time is $O(n^2|\min T(f)|)$, which is equal to the best known result by J. S. Provan and M. O. Ball [14]. Section 4 improves the algorithm described in Section 3 to a linear time one by making use of a fully condensed tree.

For space reasons, proofs of some results are omitted (see [11]).

## 2 Data Structures for $\min T(f)$

In this section, we present three data structures to represent $\min T(f)$. The first one is an *ordinary binary tree* $B(\min T(f))$ of height $n$, in which the left edge (resp., right edge) from a node at depth $j-1$ represents the case $x_j = 1$ (resp., $x_j = 0$). A leaf node $t$ of $B(S)$ at depth $n$ stores the vector $v \in S\,(\subseteq \{0,1\}^n)$, the components of which correspond to the edges of the path from the root to $t$. In order to have a compact representation, the edges with no descendants are removed from $B(S)$. For example, Figure 1 shows the binary tree for $\min T(f) = \{v^{(1)} = (110000), v^{(2)} = (101100), v^{(3)} = (101010), v^{(4)} = (011100), v^{(5)} = (011011)\}$.

The next one is called *partially condensed binary tree* $PCB(\min T(f))$. It is a binary tree obtained from $B(\min T(f))$ by recursively removing all leaves $t$ such that its parent has no left-child. Figure 2 gives an example.

We finally define a *fully condensed binary tree* $FCB(\min T(f))$. It is a binary tree obtained from $PCB(\min T(f))$ by recursively shrinking all edges $(s_1, s_2)$ such that $s_1$ has no right-child. Differently from $B$ and $PCB$, every node $s$ of *FCB* has a label $L(s)$, where $L(s)$ is either $\emptyset$ or the interval of variables $[L(s)_1, L(s)_2] = [x_i, x_j]$. Let $s$ be a node in *FCB* obtained from a path $t_1 \to t_2 \to \ldots \to t_l$ in *PCB* by shrinking all edges $(t_i, t_{i+1})$ for $i = 1, 2, \ldots, l-1$, where $(t_i, t_{i+1})$, $i = 1, 2, \ldots, l-1$, has a label $x_{p+i} = 1$. If $s$ is either a right-child or the root, then $L(s) = [x_{p+1}, x_{p+l-1}]$ for $l \neq 1$, and $L(s) = \emptyset$ for $l = 1$ (i.e., $s$ is constructed from a node $t_1$ in *PCB*). On the other hand, if $s$ is a left-child,

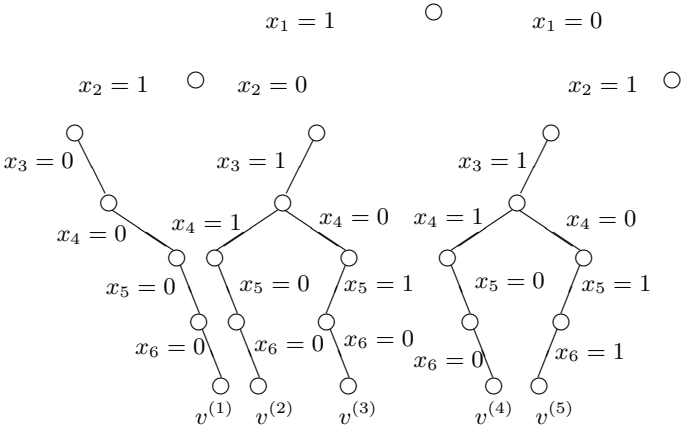



**Fig. 1.** A binary tree $B(\min T(f))$, where $f$ is given by (4).

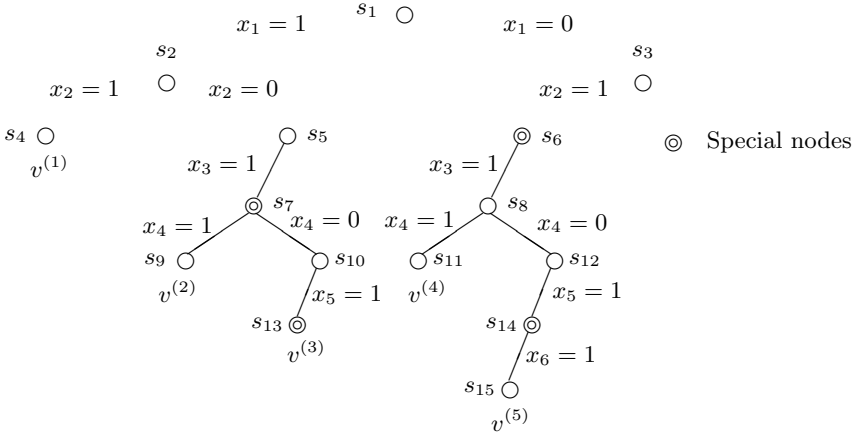



**Fig. 2.** A partially condensed binary tree $PCB(\min T(f))$, where $f$ is given by (4).

then $L(s) = [x_p, x_{p+l-1}]$, since the edge $(parent(t_1), t_1)$ in $PCB$ has a label $x_p = 1$. Note that $L(s) \neq \emptyset$ holds for all nodes $s$ that are left-children of their parents. A leaf node $t$ of $FCB(S)$ stores the vector $v \in S\,(\subseteq \{0,1\}^n)$, where $v$ is represented by the sequence of the labels appearing on the path from the root to $t$; e.g., $v^{(4)}$ is represented by $\emptyset$, $[x_2, x_3]$ and $[x_4, x_4]$, implying $v^{(4)} = (011100)$ ($v_2^{(4)} = v_3^{(4)} = v_4^{(4)} = 1$ and $v_i^{(4)} = 0$ for every other $i$). Figure 3 shows the corresponding example to Figures 1 and 2.

For a node $s$ in a binary tree ($B$, $PCB$ or $FCB$), let *parent*($s$), *grandparent*($s$), *left-child*($s$) and *right-child*($s$) denote parent, grandparent, left-child and right-child of $s$, respectively. For example, node $s_5$ in Figure 2 has parent $s_2$, grandparent $s_1$, left-child $s_7$, and no right-child. Hence $parent(s_5) = s_2$, $grandparent(s_5) = s_1$, $left\text{-}child(s_5) = s_7$ and $right\text{-}child(s_5) = \emptyset$.

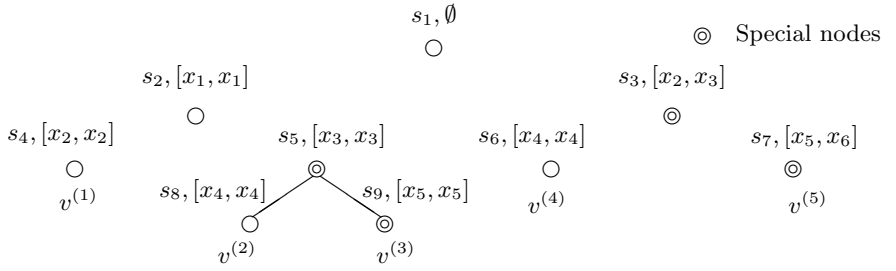



**Fig. 3.** A fully condensed binary tree $FCB(\min T(f))$, where $f$ is given by (4).

## 3 An $O(n^2|\min T(f)|)$ Time Algorithm for RECOG($\mathcal{C}_R$)

In this section, in order to easily understand a linear time algorithm for problem RECOG($\mathcal{C}_R$), we first present an $O(n^2|\min T(f)|)$ time algorithm for it. The algorithm makes use of a partially condensed tree as a data structure for $\min T(f)$.

Let $s$ be a node in $PCB$ at depth $d$, where the root is at depth 0. Let $\sigma(s)$ be a binary vector of length $d$, whose $i$-th component is the value of $x_i$ on the path from the root to $s$. For example, we have $\sigma(s_5) = (10)$ and $\sigma(s_{11}) = (0111)$ for nodes $s_5$ and $s_8$ in Figure 2. Node $s$ is said to be *special* if the last two components of $\sigma(s)$ is 0 and 1, respectively. That is, $s$ is a left-child and its parent is a right-child. For a special node $s$, a node $t$ is called *support* if either $t = \textit{right-child}(\textit{left-child}(\textit{grandparent}(s)))$ or $t$ is a leaf and $t = \textit{left-child}(\textit{grandparent}(s))$. Let $\textit{support}(s)$ denote the support of a special node $s$. For example, nodes $s_6, s_7, s_{13}$ and $s_{14}$ in Figure 2 are special, and their supports are $\textit{support}(s_6) = s_5$, $\textit{support}(s_7) = s_4$, $\textit{support}(s_{13}) = s_9$ and $\textit{support}(s_{14}) = s_{11}$. For a vector $v \in \{0,1\}^n$ and a nonnegative integer $k$, let $ON(v)_k$ denote the first $k$ elements in $ON(v)$ in the order of variables; e.g., for a vector $v = (110101)$, $ON(v)_2 = \{1,2\}$ and $ON(v)_3 = \{1,2,4\}$.

The next lemma is crucial for our algorithms.

**Lemma 3.** *A positive function $f$ is regular if and only if, for each pair of $v \in \min T(f)$ and $i \in V$ with $v_i = 0$ and $v_{i+1} = 1$, there exists a vector $w \in \min T(f)$ such that*

$$i \ \in \ ON(w) = ON(v + e^{(i)} - e^{(i+1)})_{|ON(w)|} \tag{5}$$

The following algorithm tries to find a vector $w \in \min T(f)$ satisfying (5) for each pair $(v, i)$ such that $v \in \min T(f)$ and $i \in V$ with $v_i = 0$ and $v_{i+1} = 1$. Note that, in $PCB(\min T(f))$, if a leaf representing a vector $v \in \min T(f)$ is a descendant of a special node $s$, then $v_i = 0$ and $v_{i+1} = 1$ for some $i$, and these are the labels of the two edges immediately upstream from $s$. Thus the reader can imagine the path representing the imaginary vector $v + e^{(i)} - e^{(i+1)}$ in $PCB(\min T(f))$.

In the algorithm, the set of nodes $S(s)$ is associated with each node $s$ in $PCB$. $S$ is initialized as $S(s) = \emptyset$ for all nodes $s$. We scan the nodes of $PCB(\min T(f))$

breadth-first. The node $s$ (say, at depth $d$) whose $S(s)$ becomes non-empty is a special node ($support(s)$ is added to it). Intuitively, if a leaf representing $v \in \min T(f)$ is a descendant of $s$, then $\sigma(support(s))$ can be the first $d$ components of the vector $w$ in (5). Once $S(parent(s))$ becomes non-empty, $S(s)$ is updated, so that the following property is maintained; if $t \in S(s)$, then $\sigma(t)$ is a prefix of $w$ in (5). Finally, if $f$ is regular, for every leaf $t$ representing $v \in \min T(f)$, $S(t)$ will contain the set of all $w$'s in (5). $S(t)$ will contain as many $w$'s as there are special nodes on the path from the root to $t$.

**Algorithm** CHECK-PCB
Input: $\min T(f)$, where $f$ is a positive function of $n$ variables.
Output: If $f$ is regular, "Yes"; otherwise "No".

Step 0. construct a partially condensed binary tree $PCB(\min T(f))$;
Step 1. **if** there is a node $s$ in $PCB(\min T(f))$ such that $left\text{-}child(s) = \emptyset$ and $right\text{-}child(s) \neq \emptyset$
**then** output "No" and halt
**end**;
Step 2. **for** each node $s$ **do**
$S(s) = \emptyset$
**end**{for};
$d := 2$; /* Initialize depth $d$, where the root is at depth 0. */
Step 3. **for** each node $s$ at depth $d$ **do begin**
**if** $s = left\text{-}child(parent(s))$ **then**
**call** Procedure LEFT;
**if** LEFT= 0 **then** output "No" and halt **end**
**else call** Procedure RIGHT; /* i.e., $s = right\text{-}child(parent(s))$. */
**if** RIGHT= 0 **then** output "No" and halt **end**
**end**;
**end**{for};
$d := d + 1$;
**if** there is no node at depth $d$ **then goto** Step 4;
**else goto** Step 3;
**end**;
Step 4. **for** each leaf $t$ **do begin**
**if** $S(t)$ contains an internal node **then** output "No" and halt **end**
**end**{for};
output "Yes" and halt.
Procedure LEFT
**for** each node $t$ in $S(parent(s))$ **do begin**
**if** $t$ is a leaf **then** $S(s) := S(s) \cup \{t\}$;
**else** $S(s) := S(s) \cup \{left\text{-}child(t)\}$; /* Note that $left\text{-}child(t) \neq \emptyset$. */
**end**
**end**{for};
**if** $s$ is special **then**
**if** $support(s) = \emptyset$ **then** LEFT:= 0 and **return**; /* Regularity test fails. */
**else** $S(s) := S(s) \cup \{support(s)\}$;
**end**
**end**;
LEFT:= 1 and **return**.
Procedure RIGHT
**for** each node $t$ in $S(parent(s))$ **do begin**
**if** $t$ is a leaf **then** $S(s) := S(s) \cup \{t\}$;
**else if** $right\text{-}child(t) = \emptyset$ **then** RIGHT:= 0 and **return**; /* Regularity test fails. */
**else** $S(s) := S(s) \cup \{right\text{-}child(t)\}$;
**end**
**end**;
**end**{for};
RIGHT:= 1 and **return**.

Although we omit the details (see [11]), Algorithm CHECK-PCB gives an $O(n^2|\min T(f)|)$ time algorithm for RECOG($\mathcal{C}_R$).

**Theorem 4.** *Algorithm* CHECK-PCB *correctly checks if a given positive function $f$ is regular in $O(n^2|\min T(f)|)$ time.*

## 4 A Linear Time Algorithm for Problem RECOG($\mathcal{C}_R$)

In this section, we describe a linear time algorithm for problem RECOG($\mathcal{C}_R$). The algorithm makes use of the fully condensed tree as a data structure for $\min T(f)$. Let us start with redefining a special node and its support in *FCB*. A node $s$ is said to be *special* if either

$$s = \textit{left-child}(\textit{parent}(s)),\ \textit{parent}(s) = \textit{right-child}(\textit{grandparent}(s)),$$
$$\text{and } L(\textit{parent}(s)) = \emptyset, \quad \text{or} \tag{6}$$
$$s = \textit{right-child}(\textit{parent}(s)) \text{ and } L(s) \neq \emptyset. \tag{7}$$

For a special node $s$, we now define its support $\textit{support}(s)$. In order to clarify the definition, we consider the following four cases, separately.

Case $(i)$. Let $s$ be a special node as defined by (6) such that $L(s)_1 = L(s)_2$, where $L(s)_i$ denotes the $i$-th component if $L(s)$. Then a node $t$ is called the *support* of $s$ if either

$(i.a)$ $t = \textit{right-child}(\textit{left-child}(\textit{grandparent}(s)))$, $L(\textit{parent}(t))_1 = L(\textit{parent}(t))_2$, and $L(t) = \emptyset$, or
$(i.b)$ $t$ is a leaf, $t = \textit{left-child}(\textit{grandparent}(s)))$, and $L(t)_1 = L(t)_2$.

Case $(ii)$. Let $s$ be a special node as defined by (6) such that $L(s)_1 < L(s)_2$, where we write $x_i < x_j$ if $i < j$. Then a node $t$ is called the *support* of $s$ if either

$(ii.a)$ $t$ is a left-descendant of $\textit{right-child}(\textit{left-child}(\textit{grandparent}(s)))$, $L(\textit{left-child}(\textit{grandparent}(s)))_1 = L(\textit{left-child}(\textit{grandparent}(s)))_2$, and $L(t)_2 = L(s)_2$, or
$(ii.b)$ $t$ is a leaf, $t$ is a left-descendant of $\textit{right-child}(\textit{left-child}(\textit{grandparent}(s)))$, $L(\textit{left-child}(\textit{grandparent}(s)))_1 = L(\textit{left-child}(\textit{grandparent}(s)))_2$, and $L(t)_2 < L(s)_2$, or
$(ii.c)$ $t$ is a leaf, $t = \textit{left-child}(\textit{grandparent}(s)))$, and $L(t)_1 = L(t)_2$.

Case $(iii)$. Let $s$ be a special node as defined by (7) such that $L(s)_1 = L(s)_2$. Then a node $t$ is called the *support* of $s$ if either

$(iii.a)$ $t = \textit{right-child}(\textit{left-child}(\textit{parent}(s)))$, $L(\textit{parent}(t))_1 = L(\textit{parent}(t))_2$, and $L(t) = \emptyset$, or
$(iii.b)$ $t$ is a leaf, $t = \textit{left-child}(\textit{parent}(s)))$, and $L(t)_1 = L(t)_2$.

Case $(iv)$. Otherwise (i.e., $s$ is a special node as defined by (7) such that $L(s)_1 < L(s)_2$), a node $t$ is called the *support* of $s$ if either

$(iv.a)$ $t$ is a left-descendant of $\textit{right-child}(\textit{left-child}(\textit{parent}(s)))$, $L(\textit{left-child}(\textit{parent}(s)))_1 = L(\textit{left-child}(\textit{parent}(s)))_2$, and $L(t)_2 = L(s)_2$, or
$(iv.b)$ $t$ is a leaf, $t$ is a left-descendant of $\textit{right-child}(\textit{left-child}(\textit{parent}(s)))$, $L(\textit{left-child}(\textit{parent}(s)))_1 = L(\textit{left-child}(\textit{parent}(s)))_2$, and $L(t)_2 < L(s)_2$, or
$(iv.c)$ $t$ is a leaf, $t = \textit{left-child}(\textit{parent}(s)))$, and $L(t)_1 = L(t)_2$.

Note that the definitions of supports for cases (iii) and (iv) are, respectively, obtained from those for cases (i) and (ii) by replacing all occurrences of $\textit{grandparent}(s)$ by $\textit{parent}(s)$. Let $\textit{support}(s)$ denote the support of a special node $s$. For example, nodes $s_3, s_5, s_7$ and $s_9$ in Figure 3 are all special, and their supports are $\textit{support}(s_3) = s_5$, $\textit{support}(s_5) = s_4$, $\textit{support}(s_7) = s_6$ and

$support(s_9) = s_8$. We can see that the definitions of a special node and its support in $FCB$ correspond to those in $PCB$.

Now we describe our algorithm.

**Algorithm** CHECK-FCB
Input: $\min T(f)$, where $f$ is a positive function of $n$ variables.
Output: If $f$ is regular, "Yes"; otherwise "No".

Step 0. construct a fully condensed binary tree $FCB(\min T(f))$;
Step 1. **if** $FCB(\min T(f))$ is not complete binary tree **then** output "No" and halt
/* i.e., there is an internal node $s$ having exactly one child */
**else goto** Step 2;
**end**;
Step 2. **for** each node $s$ **do**
$S(s) = \emptyset$
**end**{for};
$d := 1$; /* Initialize depth $d$. */
Step 3. **for** each node $s$ at depth $d$ **do begin**
**if** $s = left\text{-}child(parent(s))$ **then**
**call** Procedure LEFT;
**if** LEFT= 0 **then** output "No" and halt **end**
**else call** Procedure RIGHT; /* i.e., $s = right\text{-}child(parent(s))$. */
**if** RIGHT= 0 **then** output "No" and halt **end**
**end**;
**end**{for};
$d := d + 1$;
**if** there is no node at depth $d$ **then goto** Step 4;
**else goto** Step 3;
**end**;
Step 4. **for** each leaf $t$ **do begin**
**if** $S(t)$ contains an internal node **then** output "No" and halt **end**
**end**{for};
output "Yes" and halt.
Procedure LEFT
**for** each node $t$ in $S(parent(s))$ **do begin**
**if** $t$ is a leaf **then** $S(s) := S(s) \cup \{t\}$;
**else** find a left-descendant $t^*$ of $left\text{-}child(t)$ such that either $L(t^*)_2 = L(s)_2$ or
$t^*$ is a leaf and $L(t^*)_2 < L(s)_2$.
/* Since $s$ and $t^*$ are left-children, $L(s), L(t^*) \neq \emptyset$. */
**if** there is no such $t^*$ **then** LEFT:= 0 and **return**; /* Regularity test fails.*/
**else** $S(s) := S(s) \cup \{t^*\}$;
**end**
**end**
**end**{for};
**if** $s$ is special **then**
**if** $support(s) = \emptyset$ **then** LEFT:= 0 and **return**; /* Regularity test fails. */
**else** $S(s) := S(s) \cup \{support(s)\}$;
**end**
**end**;
LEFT:= 1 and **return**.
Procedure RIGHT
**for** each node $t$ in $S(parent(s))$ **do begin**
**if** $t$ is a leaf **then** $S(s) := S(s) \cup \{t\}$;
**else** find a left-descendant $t^*$ of $right\text{-}child(t)$ such that either $L(t^*)_2 = L(s)_2$ or
$t^*$ is a leaf and $L(t^*)_2 < L(s)_2$.
/* $L(r) = \emptyset$ is regarded as $L(r) = [\emptyset, \emptyset]$ for a node $r$.
Assume that $\emptyset \neq k$, $\emptyset \not> k$ and $\emptyset \not< k$ for all integer $k$. */
**if** there is no such $t^*$ **then** RIGHT:= 0 and **return**; /* Regularity test fails.*/
**else** $S(s) := S(s) \cup \{t^*\}$;
**end**
**end**
**end**{for};
**if** $s$ is special **then**
**if** $support(s) = \emptyset$ **then** RIGHT:= 0 and **return**; /* Regularity test fails. */
**else** $S(s) := S(s) \cup \{support(s)\}$;
**end**
**end**;
LEFT:= 1 and **return**.

**Theorem 5.** *Algorithm* CHECK-FCB *correctly checks if a given positive function $f$ is regular in $O(n|\min T(f)|)$ time.*

**Corollary 6.** *Problem* RECOG($\mathcal{C}_{2M}$) *can be solved in $O(n(n + |\min T(f)|))$ time.*

## References


1. H. Aizenstein, T. Hegedűs, L. Hellerstein and L. Pitt, Complexity theoretic hardness results for query learning, *Computational Complexity*, 7 (1998) 19-53.
2. M. O. Ball and J. S. Provan, Disjoint products and efficient computation of reliability, *Operations Research*, 36 (1988) 703-715.
3. P. Bertolazzi and A. Sassano, An $O(mn)$ time algorithm for regular set-covering problems, *Theoretical Computer Science*, 54 (1987) 237-247.
4. E. Boros, P. L. Hammer, T. Ibaraki and K. Kawakami, Polynomial time recognition of 2-monotonic positive Boolean functions given by an oracle, *SIAM J. Computing*, 26 (1997) 93-109.
5. V. Chvátal and P. L. Hammer, Aggregation of inequalities in integer programming, *Annals of Discrete Mathematics*, 1 (1977) 145-162.
6. Y. Crama, Dualization of regular Boolean functions, *Discrete Applied Mathematics*, 16 (1987) 79-85.
7. N. Karmarkar, A new polynomial-time algorithm for linear programming, *Combinatorica*, 4 (1984) 373-396.
8. L. G. Khachiyan, A polynomial algorithm in linear programming, *Soviet Mathematics Doklady*, 20 (1979) 191-194.
9. K. Makino and T. Ibaraki, The maximum latency and identification of positive Boolean functions, *SIAM Journal on Computing*, 26 (1997) 1363-1383.
10. K. Makino and T. Ibaraki, A fast and simple algorithm for identifying 2-monotonic positive Boolean functions, *Journal of Algorithms*, 26 (1998) 291-305.
11. K. Makino, A linear time algorithm for recognizing regular Boolean functions, RUTCOR Research Report RRR 32-98, Rutgers University 1998.
12. S. Muroga, "*Threshold Logic and Its Applications*," Wiley-Interscience, 1971.
13. U. N. Peled and B. Simeone, Polynomial-time algorithm for regular set-covering and threshold synthesis, *Discrete Applied Mathematics*, 12 (1985) 57-69.
14. J. S. Provan and M. O. Ball, Efficient recognition of matroids and 2-monotonic systems, in "*Applications of Discrete Mathematics*," (R. Ringeisen and F. Roberts, Eds.), pp. 122-134, SIAM, Philadelphia, 1988.
15. R. O. Winder, "*Threshold Logic*," Doctoral dissertation, Mathematics Department, Princeton University, 1962.

# Station Layouts in the Presence of Location Constraints
## (Extended Abstract)

Prosenjit Bose[2], Christos Kaklamanis[1], Lefteris M. Kirousis[1], Evangelos Kranakis[23], Danny Krizanc[23], and David Peleg[4]

[1] University of Patras, Department of Computer Engineering and Informatics, Rio 26500, Patras, Greece,
{kakl,kirousis}@ceid.upatras.gr.

[2] Carleton University, School of Computer Science, Ottawa, ON, K1S 5B6, Canada,
{jit,kranakis,krizanc}@scs.carleton.ca.

Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

[3] Research supported in part by MITACS project CANCCOM (Complex Adaptive Networks for Computing and Communication).

[4] Weitzman Institute, Department of Applied Mathematics and Computer Science, POB 26, 76100, Israel,
peleg@wisdom.weizman.ac.il.

**Abstract.** In wireless communication, the signal of a typical broadcast station is transmited from a broadcast center $p$ and reaches objects at a distance, say, $R$ from it. In addition there is a radius $r$, $r < R$, such that the signal originating from the center of the station is so strong that human habitation within distance $r$ from the center $p$ should be avoided. Thus every station determines a region which is an "annulus of permissible habitation". We consider the following *station layout (SL)* problem: Cover a given (say, rectangular) planar region which includes a collection of orthogonal buildings with a minimum number of stations so that every point in the region is within the reach of a station, while at the same time no building is within the dangerous range of a station. We give algorithms for computing such station layouts in both the one- and two-dimensional cases.

## 1 Introduction

In wireless communication we are interested in providing access to communication to a region (e.g. a city, a campus, etc) within which several sites (e.g. buildings) are located. Closeness to stations may be undesirable in certain instances, e.g. hospital or laboratory facilities, people with heart pace-makers, etc. Thus, although we are interested in providing communication access everywhere, part of the buildings may need to be away from strong electronic emissions of stations.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 269–278, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Cellular phones are radio receivers which operate in the ultra-high frequency (UHF) band. They receive radio transmissions from a central base station (or cell) at frequencies between 869 and 894 MHz and retransmit their radio signal back to the base station at frequencies between 824 and 850 MHz. Stations emit signals whose strength is inversely proportional to the square of the distance from the station. It follows that the signal's strength degrades as we move away from the center of the station. This determines a threshold ($1W$ is the currently accepted value) beyond which the signal is sufficiently safe but still strong enough to reach its desirable destination. A comprehensive study and survey of the biological effects of exposure to radio frequency resulting from the use of mobile and other personal communication services can be found in [9].

In this paper we consider broadcast station layouts in wireless communication in which we take into account health hazards resulting from the closeness of human habitation to the transmission station. Given such constraints we are interested in minimizing the number of broadcast stations used. The buildings are located within a region $\mathcal{R}$, which for the sake of simplicity we assume to be rectangular. In the most general case the buildings may be represented by simple polygons with or without holes.

### 1.1 Formulation of the Problem and Notation

The parameters involved in transmissions for a typical station in the plane are the transmission center $p$ of the station, and positive real numbers $r < R$ such that $R$ is the reachability range of the station, i.e. the signal transmitted from the center $p$ can reach any destination at distance $R$ from the center, and $r$ is the dangerous range of the station, i.e. the strength of the transmitted signal exceeds permissible health constraints within distance $r$ from the center. Let $d(\cdot, \cdot)$ be the given distance function. The disc $D(p; r) = \{x : d(x, p) < r\}$ is the locus of points that are "too close" to the broadcast center $p$. Existing health constraints make it advisable that human habitation is not allowed within the disc $D(p; r)$. At the same time the signal reception does not cause a health hazard beyond distance $r$ from the broadcast center of the station; moreover the signal can reach any location at distance at most $R$ from the center. This determines an annulus $A(p; r, R) = D(p; R) \setminus D(p, r)$. Thus $A(p; r, R)$ is the annulus formed by two squares centered at $p$ and diameter $2r, 2R$, respectively. Throughout this paper we assume that $d$ is the $L_1$ or Manhattan metric.

The numbers $r, R$ represent the parameters suggested by the manufacturer. In addition, we want to produce a layout of transmitting stations in such a way that all points of the region $\mathcal{R}$ are within range $R$ of a transmitting station while at the same time no site is within distance $r$ from any transmitting station. More specifically, we have the following definition.

**Definition 1.** *A collection of $m$ points $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ is called an $(r, R)$-*cover for $(\mathcal{R}, \mathcal{P})$ *if the collection $\{D(a_i; R) : i = 1, 2, \ldots, m\}$ of discs covers the rectangular region $\mathcal{R}$, but none of the discs $D(a_i, r), i = 1, \ldots, m$ have a point interior to any building in $\mathcal{P}$. If $r = 0$ then an $(0, R)$-cover is also called an $R$-cover.*

*Problem 1 (General problem).*
**Input:** A rectangular region $\mathcal{R}$ and a collection $\mathcal{P}$ of simple polygons (or buildings) inside the region and two real numbers $0 \leq r < R$.
**Output:** An $(r, R)$-cover $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ for $(\mathcal{R}, \mathcal{P})$ or a report that no such cover exists.

The important parameter to be optimized is the number $m$ of transmitting stations. In general we are interested in an algorithm that will report an optimal or even near-optimal number of stations. A cover is said to be *optimal* iff it uses a minimum number of stations. If $\mathcal{P}$ is the collection of buildings then we use the notation $A(\mathcal{P}; r, R)$ to abbreviate the square annulus version of the problem. We stipulate that every point in the region $\mathcal{R}$ must be within the reach of a station. At the same time although a point lying in a building cannot be within the dangerous zone of a station, this is not a priori prohibited if the point does not lie inside a building. In addition, it is permissible that a point (in a bulding) may lie within the range of more than one station. An important observation that will be used in the sequel is that Problem 1 is computationally equivalent to the following problem:

*Problem 2 (Reduced general problem).*
**Input:** A rectangular region $\mathcal{R}$ and a collection $\mathcal{P}$ of simple polygons (or buildings) inside the region and a real number $0 < R$.
**Output:** An $R$-cover $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ for $(\mathcal{R}, \mathcal{P})$ or a report that no such cover exists.

Clearly, Problem 1 is more general than Problem 2. To prove the reverse reduction we surround each building with a strip of width $r$ and merge the resulting orthogonal polygons into the new buildings. Details of the proof of this are left to the reader.

### 1.2 Results of the Paper

In the sequel we consider first the one-dimensional case of the problem. In the two-dimensional case, first we consider an algorithm for testing the existence of a solution. Subsequently, we show how to reduce the problem to a discrete problem in which the centers of the stations are to be located at predetermined points within the region $\mathcal{R}$. This is used later on to provide (1) a linear time, logarithmic approximation algorithm by reduction to SET-COVER, (2) a polynomial time, constant approximation algorithm, and (3) for "thin" buildings, a linear time constant approximation algorithm.

### 1.3 On the Number of Stations

We observe that the size of an $(r, R)$-cover for $(\mathcal{R}, \mathcal{P})$, i.e., the number of points needed to cover a rectangular region $\mathcal{R}$, is not only proportional to $\text{Area}(\mathcal{R})/R^2$ but also to the number of vertices of the given polygons $\mathcal{P}$ bounded by the rectangular region.

**Theorem 1.** *There is a ractangular region $\mathcal{R}$ of area $O(R^2)$ with a single polygon in $\mathcal{P}$, such that any $(r, R)$-cover for it must be of size $\Omega(n)$ where $n$ is the number of vertices in the given polygon in $\mathcal{P}$.*

As a consequence, the complexities of the given algorithms are best expressed as a function of the input size of the problem. This is defined to be $N = \frac{\text{Area}(\mathcal{R})}{R^2} + n$, where $n$ is the total number of vertices of the polygonal buildings and $\text{Area}(\mathcal{R})$ is the area of the given region $\mathcal{R}$.

## 2 Algorithm on the Line

This section considers the one-dimensional analogue of the station layout problem, problem 1-SL. In this case, the transmitting station is modeled by the one-dimensional analogue of the annulus, i.e., the set $I(p; r, R)$ of points $x$ on a line such that $r \leq |x - p| < R$. The region is a line segment $I_0$, and the set of buildings is $\mathcal{I} = \{I_1, \ldots, I_n\}$, where each building $I_j$, $1 \leq j \leq n$ is an interval $I_j = [p_j, q_j]$ in the line segment $I_0$. In this version, an $(r, R)$-cover for the instance at hand is a collection of $m$ points $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ none of which is at a distance less than $r$ from any interval in $\mathcal{I}$, such that the collection of intervals $I(a_i; 0, R), i = 1, \ldots, m$ covers the segment $I_0$.

*Problem 3 (One-dimensional problem).*
**Input:** A line segment $I_0$ and a collection of (possibly ovelapping) intervals $\mathcal{I} = \{I_1, \ldots, I_n\}$ inside the segment, and two real numbers $0 \leq r < R$.
**Output:** an $(r, R)$-cover $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ for $(I_0, \mathcal{I})$ or a report that no such cover exists.

**Theorem 2.** *There exists an $O(N \log N)$ time algorithm for computing a minimum size $R$-cover for an instance $\mathcal{I}$ of the 1-SL problem.*

## 3 Algorithms on the Plane

In this section we study the case of orthogonal buildings and stations which are square annuli.

### 3.1 Testing for a Solution

Recall our previous reduction of Problem 1 to Problem 2. Hence without loss of generality, we may assume $r = 0$.

**Theorem 3.** *A solution exists if and only if there is no point $p$ interior to a polygon (i.e., building) such that $D(p; R)$ lies entirely inside the interior of the polygon. In particular, there exists an $O(\max\{N, n^2\})$ time algorithm to determine whether or not a solution exists.*

### 3.2 Reduction to a Discrete Problem Without Buildings

Now we reduce Problem 2 to a discrete problem without any buildings (see Problem 4). First we define a collection $L$ of points inside the rectangular region as the union of two sets $L_0$ and $L_1$, to be defined below.

1. To obtain the points in $L_0$ we partition the rectangular region $\mathcal{R}$ into parallel and horizontal strips at distance $R$ apart and let $L_0$ be the collection of points of intersection of these lines which lie outside any building in $\mathcal{P}$.
2. The points in $L_1$ lie on the perimeter of buildings in $\mathcal{P}$. These points are of two types: (a) all vertices of these polygons, (b) for any polygon in $\mathcal{P}$, and starting from an arbitrary vertex of the polygon, walk along the perimeter and place points on the perimeter at distance $R$ apart.

We refer to squares whose centers are points in $L$ as *$L$-squares*. A *discrete* $(r, R)$-cover for the region $\mathcal{R}$ is a cover by $L$-squares. The basic lemma is the following.

**Lemma 1.** *An $(r, R)$-cover to the square version of problem 1 exists if and only if a discrete $(r, R)$-cover exists. Moreover, the size of an optimal discrete cover is at most four times that of an optimal cover and the $(r, R)$-cover can be constructed in time $O(N)$.*

The previous lemma reduces Problem 1 to the following one.

*Problem 4 (General discrete problem).*
**Input:** A rectangular region $\mathcal{R}$, a set $L$ of points inside the region, and a positive number $R$.
**Output:** A subset $\mathcal{S}$ of minimal size of the set $L$ such that the set of squares of radius $R$ centered at points of $\mathcal{S}$ cover the entire rectangular region.

Conversely, it is easy to see that Problem 4 can be reduced to Problem 1. To see this consider an instance of Problem 4. For each point $p \in L$ place a square $D(p; R) \setminus \{p\}$. Append these squares as part of the input set of polygons. It is clear that in the resulting instance of Problem 1 stations can only be placed at points $p \in L$.

### 3.3 Logarithmic Approximation Algorithm

In the sequel we give an $O(\log N)$-approximation algorithm for Problem 4 by reducing it to the well-known problem SET-COVER, where $N$ is the size of the input. Consider an input as in Problem 4. For each point $p \in L$ consider the square with radius $R$ centered at $p$. The collection of these squares forms a planar subdivision of the rectangular region $\mathcal{R}$. Consider the bipartite graph $(A, L)$ such that $A$ is the set of planar rectangular subdomains thus formed. Moreover, for $a \in A$ and $p \in L$, $\{a, p\}$ is an edge if and only if the subdomain $a$ lies entirely inside the square of radius $R$ centered at $p$. Now observe that any solution of SET-COVER for the graph $(A, L)$ corresponds to a solution of Problem 4 and vice versa. In view of the fact that there are $O(\log N)$ approximation algorithms for SET-COVER (e.g. the greedy algorithm [4]) we obtain the following theorem.

**Theorem 4.** *There is a linear time, logarithmic approximation algorthm for Problem 1.*

### 3.4 Constant Approximation Algorithm

In this subsection we provide a polynomial time constant approximation algorithm for solving Problem 4. From now on and for the rest of the paper that the radius of the stations is $R = 1$. Our solution is via a reduction to the following problem.

*Problem 5 (Discrete rectangle problem).*
**Input:** A rectangle $R$ with both height and width of length $\leq 1$, and a collection $\mathcal{Z} = \{p_1, p_2, \ldots, p_n\}$ of $n$ points not necessarily all inside the rectangle.
**Output:** The minimum number of unit squares with centers lying at the given points whose union covers the rectangle $R$.

In particular, we will prove the following theorem.

**Theorem 5.** *There is a polynomial time, constant approximation algorithm for Problem 5.*

Before proving this theorem we indicate how it can be used to find a solution to the General discrete problem, i.e., Problem 4. We can prove the following theorem.

**Theorem 6.** *There is a polynomial time constant approximation algorithm for Problem 4, where $N$ is the size of the input. The constant is at most four.*

#### Outline of the Proof of Theorem 5

We divide up the description of the proof into a classification of stations depending on how the stations cover the rectangle. The resulting algorithm is recursive and is based on dynamic programming. The idea is as follows. We consider the "stations" centered at the given points. For a given rectangle $R$ we consider all possible coverings of this rectangle by stations. We classify the square stations according to how they cover $R$, e.g. a square station may either cover $R$ completely, or the left-, right-, down-, up-side of $R$, or the left-down-, left-up-side, etc. It follows that the number of stations in an optimal solution is determined from solutions to other subrectangles. By scanning the solutions we can select the optimal solution to the rectangle $R$.

#### Classification of the Min Size Cover

Let $R := R[x, x', y, y']$ be the axis parallel rectangle with lower left corner $(x, y)$ and upper right corner $(x', y')$, height $V_R = y' - y$ and width $H_R = x' - x$. Let the given points be $\mathcal{Z} = \{p_1, p_2, \ldots, p_n\}$ and suppose that $R_i$ denote the unit square centered at $p_i$. We use the notation $R_i = R[x_i^L, x_i^R, y_i^D, y_i^U]$ for the station with lower left corner $(x_i^L, y_i^D)$ and upper right corner $(x_i^R, y_i^U)$. We want to find

the minimum size subset $P \subseteq \mathcal{Z}$ such that the collection $\mathcal{R}(P) = \{R_i : p_i \in P\}$ of squares covers the rectangle $R$.

We now define the $R$-Classification of stations. Given a rectangle $R := R[x, x', y, y']$ we classify the squares in $\mathcal{R}(\{p_1, p_2, \ldots, p_n\})$ as follows using the notation $C, L, R, U, D$ for Contains, Left, Right, Up, and Down, respectively. We also use the notation $LD$ for the set $L \cap D$, i.e.,

$$\begin{aligned} C &= \{R_i : R_i \text{ contains } R\} \\ L &= \{R_i : y_i^U \geq y', y_i^D \leq y, x < x_i^R < x', x_i^L \leq x\} \\ LD &= \{R_i : y < y_i^U < y', y_i^D < y, x_i^L < x, x < x_i^R < x'\} \end{aligned}$$

The other classes $R, U, D$ and $LU, RD, RU$ are defined similarly. The sets $L$ and $LD$ are depicted in Figure 1. Note that these sets are disjoint and their union is equal to $\mathcal{R}(\mathcal{Z})$.

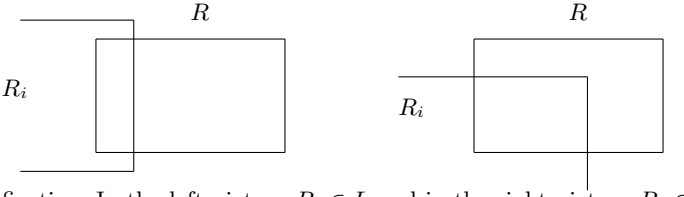



**Fig. 1.** $R$-Classification: In the left picture $R_i \in L$ and in the right picture $R_i \in LD$.

Given $\mathcal{R}$ and $R$ let $C^*(\mathcal{R}, R)$ denote the minimum size cover of $R$ by stations from $\mathcal{R}$. We consider the following cases. Each case assumes that the previous case does not hold. With this in mind it is clear that the classification is complete, in the sense that $C^*(\mathcal{R}, R)$ must belong to one of the cases below.

**Case 1.** $C \neq \emptyset$. In this case $|C^*(\mathcal{R}, R)| = 1$.

**Case 2L.** $L \neq \emptyset$. In this case $C^*(\mathcal{R}, R)$ contains exactly one $R_i \in L$ (namely the one farthest to the right which dominates all the other rectangles in $L$) (See Figure 2.)

**Cases 2R, 2U, 2D.** Similar.

Next we consider the classes $LD, LU, RD, RU$. We study only Case 3LD. The other three cases are similar.

**Case 3LD.** $C^*(\mathcal{R}, R)$ contains at least two rectangles from LD.

Let the squares of $C^*(\mathcal{R}, R) \cap LD$ be $R_{i_1}, R_{i_2}, \ldots, R_{i_k}$ ordered by ascending $x$-coordinate. Without loss of generality we may assume that they are also ordered by descending $y$-coordinate. Indeed, otherwise one of them, say $R_{i_j}$ is dominated by the following square $R_{i_{j+1}}$ in terms of its contribution to covering $R$, and hence it can be discarded. Let $R_{i_1}$ and $R_{i_2}$ be the two rectangles of LD in $C^*(\mathcal{R}, R)$, and let $\rho$ be the upper right intersection point between $R_{i_1}$ and $R_{i_2}$, $\rho = (\hat{x}, \hat{y}) = (x_{i_1}^R, y_{i_2}^U)$ (see Figure 3).

We note that the same observation as for Case 3LD, holds also for Cases 3LU, 3RD, 3RU. The last case left is the following.

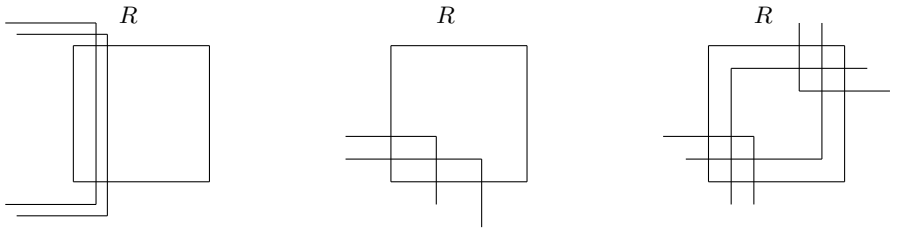



**Fig. 2.** Leftmost figure depicts Case 2L; middle figure depicts Case 3LD, and rightmost figure depicts Case 4.

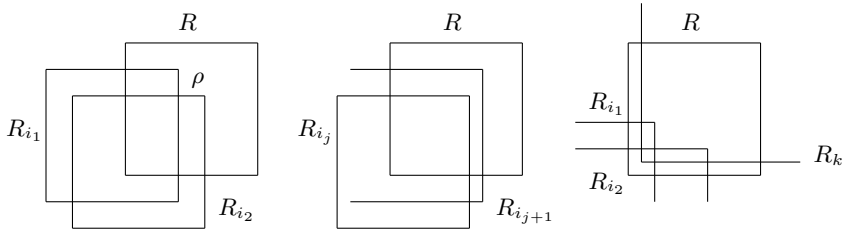



**Fig. 3.** Clasification of $C^*(\mathcal{R}, R)$.

**Case 4.** $C^*(\mathcal{R}, R)$ contains exactly one rectangle from each of $LU, LD, RU, RD$.

### Dynamic Programming Algorithm

We are now in a position to use the above $R$-Classification of squares in order to provide a dynamic programming algorithm computing the minimal number of squares in a covering. An optimal solution is constructed by recursion. The purpose of the previous classification is to establish the fact that all possible cases for the structure of $C^*$ were examined by the algorithm, and no possibility was omitted. Define the sets

$$X = \{x_0^L \leq x_i^L, x_i^R < x_0^R : 1 \leq i \leq n\} \cup \{x_0^L, x_0^R\} ,$$
$$Y = \{y_0^D \leq y_i^D, y_i^U \leq x_0^U : 1 \leq i \leq n\} \cup \{y_0^D, x_0^U\},$$

where $x_0^L, x_0^R, y_0^D, y_0^U$ are the coordinates of the original rectangle. For any $x, x' \in X$ and $y, y' \in Y$, let $T(x, x', y, y')$ be the size of the minimum cover of the rectangle $R[x, x', y, y']$ by squares in $\mathcal{R}(\mathcal{Z})$. The procedure is the following.

**Procedure:**

Calculate $T(x, x', y, y')$ for every $x, x' \in X$ and $y, y' \in Y$ by first order, i.e., calculating $T(x, x', y, y')$ only after finishing all rectangles $T(a, a', b, b')$ with both $|a - a'| \leq |x - x'|$ and $|b - b'| \leq |y - y'|$. In order to calculate $T(x, x', y, y')$ for $R = R[x, x', y, y']$ and $\mathcal{R}$, check systematically through all possibilities for $C^*(\mathcal{R}, R)$.

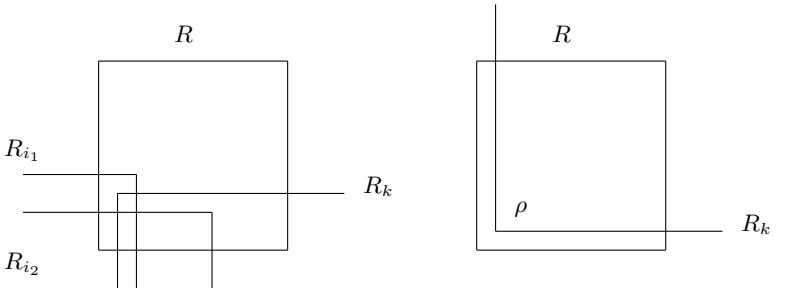



**Fig. 4.** $R$-Classification of $C^*(\mathcal{R}, R)$.

**Case 1.** If we are in Case 1, then there should be some $R_i$ that contains $R$. This is checkable in time $O(n)$.
**Case 2L.** In this case suppose $L \neq \emptyset$. Go through each $R_i \in L$. For each of these, consult the table concerning the value $t_i = T(x_i^R, x', y, y')$, which is the minimum coverage for $R[x_i^R, x', y, y']$. If such an $R_i$ exists then return $t_i + 1$. Of course it suffices to take the "most dominant" $R_i \in L$, i.e., the one with greatest $x_i^R$.
Cases 2R, 2U, 2D are similar, while Case 4 is easy.
**Case 3LD.** From the observation we know that in this case we have $R_k$ as in the rightmost picture depicted in Figure 3. Cycle through all choices of $R_{i_1}, R_{i_2} \in LD$ and $R_k \in RU$. If not in "right shape" ignore. Else

$$t' \leftarrow T(R')$$
$$t'' \leftarrow T(R'')$$
$$Reply(i_1, i_2, k) \leftarrow t' + t'' + 3$$

Choose the best of $O(n^3)$ replies $Reply(i_1, i_2, k)$.

Cases 3LU, 3RD, 3RU are similar, while Case 4 is easy. Combining all these cases we obtain the general procedure for computing $T(x, x', y, y')$ by selecting the best of all replies. This completes the outline of the proof of Theorem 5. ■

### 3.5 Conditional, Constant Approximation Algorithm

In this section we provide a linear time, constant approximation algorithm when the buildings satisfy certain width contraints.

**Theorem 7.** *If there is no point $p$ interior to a polygon such that $D(p; R/2)$ lies entirely inside the interior of the polygon then there is a linear time approximation algorithm for covering the region $\mathcal{R}$ whose number of squares is at most four times the optimal.*

We can improve on the constant four above as follows. The horizontal $h$ (respectively, vertical $v$) width of an orthogonal polygon is the maximum length horizontal (respectively, vertical) line segment that lies inside the polygon.

**Theorem 8.** *If either $h \leq 2R$ or $v \leq 2R$ then there is a linear time algorithm for finding a solution to Problem 2 such that the number of stations is at most two times the optimal.*

### 3.6 Conclusion

It is an open problem whether or not finding an optimal solution to our problem can be dome in polynomial time. Another interesting open problem arises when we consider an upper bound on the number of stations permitted to cover a given point in the region. (As Theorem 1 indicates, such a coverage may not always exist.)

We note that the results of the paper are stated only for the Manhattan or $L_1$ metric. Similar algorithms and results are possible for the more realistic "hexagonal" metric. The only modification necessary is that the resulting constants in approximation algorithms are now derived using stations with hexagonal transmission range. Details will appear in the final version of the paper,

## References

1. L. I. Aupperle, H. E. Conn, J. M. Kell, and J. O'Rourke, "Covering Orthogonal Polygons with Squares", in proceedings of 26th Annual Allerton Conference on Coom. Contr, and Comp., Urbana, 28-30, Sep. 1988.
2. R. Bar-Yehuda, and E. Ben-Chanoch, "An $O(N \log^* N)$ Time Algorithm for Covering Simple Polygons with Squares", in proceedings of the 2nd Canadian Conference on Computational Geometry, held in Ottawa, pp. 186-190, 1990.
3. B. N. Clark, C. J. Colbourn, and D. S. Johnson, "Unit Disk Graphs", Discrete Mathematics 86(1990) 165-177.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "An Introduction to Algorithms", MIT Press, 1990.
5. N. A. DePano, Y. Ko, and J. O'Rourke, "Finding Largest Equilateral Triangles and Squares", Proceedings of the Allerton Conference, pages 869 - 878, 1987.
6. F. Gavril, "Algorithms for Minimum Coloring, Minimum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph", SIAM J. Comput., Vol. 1, No. 2, June 1972, pp. 180-187.
7. D. S. Hochbaum and W. Maass, "Approximation Schemes for Covering and Packing Problems in Image Processing and VLSI", J. ACM 32:130-138, 1985.
8. H. B. Hunt, M. V. Marathe, V. Radhakrishnan, S. S. Ravi, D. J. Rosenkrantz, R. E. Stearns, "NC Approximation Schemes for NP- and PSPACE-Hard Problems for Geometric Graphs", in Proceedings of 2nd ESA, pp. 468-477.
9. J. C, Lin, "Biological Aspects of Mobile Communication Data", Wireless Networks, pages 439-453, Vol. 3 (1997) No, 6.
10. D. Moitra, "Finding a Minimal Cover for Binary Images: An Optimal Parallel Algorithm", Algorithmica (1991) 6: 624-657.
11. K. Pahlavan and A. Levesque, "Wireless Information Networks," Wiley-Interscience, New York, 1995.

# Reverse Center Location Problem*

Jianzhong Zhang[1], Xiaoguang Yang[2], and Mao-cheng Cai[3,**]

[1] Department of Mathematics, City University of Hong Kong, Hong Kong
MAZHANG@cityu.edu.hk
[2] Institute of Systems Science, Academia Sinica, Beijing, China
xgyang@iss04.iss.ac.cn
[3] Institute of Systems Science, Academia Sinica, Beijing, China
caimc@bamboo.iss.ac.cn

**Abstract.** In this paper we consider a reverse center location problem in which we wish to spend as less cost as possible to ensure that the distances from a given vertex to all other vertices in a network are within given upper bounds. We first show that this problem is NP–hard. We then formulate the problem as a mixed integer programming problem and propose a heuristic method to solve this problem approximately on a spanning tree. A strongly polynomial method is proposed to solve the reverse center location problem on this spanning tree.

**Keywords**: networks and graphs, NP-hard, satisfiability problem, relaxation, maximum cost circulation.

**AMS subject classification.** 68Q25, 90C27.

## 1 Introduction

The center location problem, which is to find the "best" position of a facility in a network to minimize the distance from the facility to the farthest vertices of the network, is a very practical OR problem which has attracted much attention. The problem is well–solved, see, for example, [5]. A slight generalization of the center location problem is to find a vertex whose distances to other vertices do not exceed given upper bounds. This can be done by computing the distance matrix using the Floyd–Warshall algorithm, and comparing the entries of the matrix with the given upper bounds.

In this paper, we consider the reverse problem of the generalized center location problem, that is, to modify the weights of a network optimally and with bound constraints, such that the distances between a given vertex and other vertices are bounded by given limits. Note that here a vertex is first selected and then we need to adjust the lengths to make the vertex become the solution of a generalized center location problem. That is why we call it a reverse problem.

* The authors gratefully acknowledge the partial support of the Hong Kong Universities Grant Council (CERG CITYU-9040189).

** Research partially supported by the National Natural Science Foundation of China.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 279–294, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Obviously, to solve this problem, we only need to decrease, but never increase, some weights.

We can describe the reverse center location problem (RCL) formally as follows. Let $G = (V, E, w, l, b, c)$ be a weighted undirected (directed) connected graph, where $V$ is a vertex set, $E$ is an edge (arc) set, $w : E \to R_+$ is a weight or distance vector. Let $s$ be a given vertex in $V$, and the vector $l : V\backslash\{s\} \to R_+$ gives the upper bound for distances from other vertices to $s$. Let $b : E \to R_+$ be a bound vector for the adjustment of weights which satisfies $b(e) \leq w(e)$, $\forall e \in E$. Let $c : E \to R_+$ be the cost vector incurred by decreasing per unit of $w$. The reverse center location problem is to decrease $w$ to $w^*$ such that

(a) $d_{w^*}(s, v) \leq l(v)$ holds under $w^*$ for all $v \in V\backslash\{s\}$.
(b) $w(e) - b(e) \leq w^*(e)$ for each $e \in E$.
(c) The total cost $\sum_{e\in E} c(e)(w(e) - w^*(e))$ is minimum.

Note that in this paper we use $d_w(u, v)$ to represent the distance from vertex $u$ to vertex $v$ under weight $w$, which is equal to the length of the shortest path between $u$ and $v$ under weight $w$.

It is straightforward to see that a given instance is feasible if and only if $d_{w-b}(s, v) \leq l(v)$ holds for all $v \in V\backslash\{s\}$.

A closely related work was reported by Burton, Pulleyblank & Toint [1]. They consider such a problem: for $m$ given pairs of vertices $(s_i, t_i)$ and upper bounds $u_i$, modify $w$ to $w^*$ such that $d_{w^*}(s_i, t_i) \leq u_i$ under $w^*$ and $\sum_{e\in E}(w(e) - w^*(e))^2$ is minimum. They proved that the problem is NP–hard, and gave an algorithm to find a local optimum solution.

Let $s_i \equiv s$, $\{t_i \mid i = 1, 2, \cdots, m\} = V\backslash\{s\}$, then our reverse center location problem can be regarded as a special case of theirs if only the distance requirement is concerned. But to make the reverse problem more reasonable, we use a weighted linear objective function to replace their $l$-2 measure so that the cost coefficients on various edges can be different. Also, we impose restrictions on the magnitude of the adjustments. What we pursue in this paper is an approximate global optimum solution. Consequently, the method which we developed in this paper is independent of theirs. The paper is organized as follows. In Section 2, we try to show that the reverse center location problem is NP–hard. Then the problem is formulated as a mixed integer programming problem in Section 3 . In Section 4 we relax the integer requirement and then use a heuristic method to determine a spanning tree such that the problem can be treated approximately on the tree. In Section 5 we solve the dual problem of the approximate RCL problem as a maximum cost circulation problem, and then in Section 6 we recover the primal approximate optimal solution from the dual optimal solution. Finally we summarize our method and give conclusions in Section 7.

## 2 Complexity Analysis

We now show that the reverse center location problem is NP-hard. The idea is first motivated from [1], but our proof is more elegant.

The SATISFIABILITY problem [15] can be stated as follows: Given $m$ clauses $\{C_1, C_2, \cdots, C_m\}$ involving $n$ Boolean variables $\{x_1, x_2, \cdots, x_n\}$, does there exist a set of values of the variables (called a *truth assignment*) such that all clauses are *true*?

If such a set of values exists, we say the SATISFIABILITY problem is *satisfiable*. The SATISFIABILITY problem is the earliest natural NP–complete problem proven by Cook [6]. The technique which we use to show the NP-hardness of the reverse center location problem is a polynomial time reduction of the SATISFIABILITY problem into an instance of decision problem of the RCL problem.

**Theorem 1** *Even if $c(e) = const$, $l(v) = const$ and $b(e) = w(e)$, the reverse center location problem is NP–hard.*

**Remark**. This is the simplest case of the RCL model, and by this theorem, for general $c, l, b$, the reverse center location problem is of course NP-hard.

*Proof.* Given an instance of the RCL problem $G = (V, E, w, l, b, c)$ and a number $L$, the decision problem of the RCL problem is whether there is a solution $w^*$ satisfying the conditions (a), (b) and

$$(d) \qquad \sum_{e \in E} c(e)(w(e) - w^*(e)) \leq L \ .$$

We call this $w^*$ a feasible solution of the RCL problem within the budget limit $L$.

Let us now construct a reduction from the SATISFIABILITY problem to an instance of decision problem of the RCL problem.

Consider a SATISFIABILITY problem with $n$ variables $\{x_1, x_2, \cdots, x_n\}$ and $m$ clauses $\{C_1, C_2, \cdots, C_m\}$. Without loss of generality, we assume that, for each variable $x_i$ and each clause $C_j$, $x_i$ does not appear in $C_j$ more than once, and $x_i$ and $\bar{x}_i$ do not appear in $C_j$ at the same time.

Construct a graph $G = (V, E)$ as follows:

$$\begin{aligned} V &= \bigcup_{i=1}^{n} \{t_i, u_i, v_i\} \cup \{t_{n+1}, t_{n+2}, t_{n+3}\} \cup \{q_j \mid j = 1, 2, \cdots, m\} \ , \\ E &= \bigcup_{i=1}^{n} \{(t_i, u_i), (t_i, v_i), (u_i, t_{i+1}), (v_i, t_{i+1})\} \cup \{(t_{n+1}, t_{n+2}), (t_{n+1}, t_{n+3})\} \\ &\quad \cup \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (\{(u_i, q_j) \mid x_i \in C_j\} \cup \{(v_i, q_j) \mid \bar{x}_i \in C_j\}) \ . \end{aligned}$$

In the graph, Boolean variable $x_i$ corresponds to vertex $u_i$, and its negation $\bar{x}_i$ corresponds to vertex $v_i$. Clause $C_j$ corresponds to vertex $q_j$. Further, there is an edge $(u_i, q_j)$ if and only if $x_i \in C_j$, and $(v_i, q_j)$ exists if and only if $\bar{x}_i \in C_j$. We call $u_i$ and $v_i$ *literal vertices*, $q_j$ *clause vertices*, the edges $(u_i, q_j)$ or $(v_i, q_j)$ *clause edges*, and other edges *literal edges*.

Let $s = t_1$, $l(v) = 1$ for all $v \in V \backslash \{s\}$, $c(e) \equiv 1$, and define a weight function on $G$ as

$$w(e) = \begin{cases} 2n+1 & \text{if } e = (u_i, q_j) \text{ or } (v_i, q_j)\ , \\ 1 & \text{otherwise}\ . \end{cases}$$

Now we claim that the SATISFIABILITY problem is satisfiable if and only if in the above network the RCL problem has a feasible solution whose modification cost is at most $L = 2n(m+1)$.

First we assume that the SATISFIABILITY problem is satisfiable, then the literals with value *true* correspond to a path from $t_1$ to $t_{n+1}$. For example, if $\bar{x}_1 = x_2 = true$, then the part between $t_1$ and $t_3$ of the path should be $(t_1, v_1), (v_1, t_2), (t_2, u_2), (u_2, t_3)$. Let us change the weights of the edges on this path to zero. For each clause vertex $q_j$, choose one clause edge connecting it to a literal vertex corresponding to the *true* literal in $C_j$, and change the weight of this edge to one. It is easy to verify that this is a feasible solution of the reverse center location problem, and the modification cost is exactly $2n(m+1)$.

Conversely, suppose that the reverse center location problem has a feasible solution with a modification cost at most $2n(m+1)$. First, for each clause vertex $q_j$, obviously the weight of one edge between a literal vertex and the $q_j$ must be decreased to one or less, for otherwise any path from $t_1$ to $q_j$ will be longer than 1. The cost for this reduction is at least $2n$. It is easy to understand that for each $q_j$ we can have only one such edge, because even if only one of them has two such edges, then all $2n(m+1)$ budget has to be spent on the clause edges and thus no path from $t_1$ to $t_{n+2}$ can have a length one or less. Denote by $Q$ the set of such $m$ edges for the $m$ clause vertices. The total cost of these changes on $Q$ is at least $2nm$.

For any path from $t_1$ to $t_{n+1}$ passing a clause vertex $q_j$, the path must have two edges between the literal vertices and $q_j$. Therefore, at least one of these two edges does not belong to $Q$. Furthermore, the length of this path is at least $4n+4$, and in order to shorten the length of this path to not greater than one, the additional cost is at least $2n+3$. Since $2nm + (2n+3) > 2n(m+1)$, we can not reduce the length of this path to one within the budget limit. So, a path from $t_1$ to $t_{n+1}$, which can be modified to no longer than one, must consist of literal edges only.

Now let us consider how the feasible solution can make the distance between $t_1$ and $t_{n+2}$, and the distance between $t_1$ to $t_{n+3}$ be not longer than one. Notice that all paths between $t_1$ and $t_{n+2}$ as well as between $t_1$ and $t_{n+3}$ must pass through $t_{n+1}$. So, we consider how to reduce the length between $t_1$ and $t_{n+1}$ first. Notice also that the remaining budget is at most $2n$ and the original length for any acyclic literal path between $t_1$ and $t_{n+1}$ is $2n$. Suppose we reduce the length of a path between $t_1$ and $t_{n+1}$ to $\ell$, and $0 < \ell \leq 1$, then the budget left is at most $\ell$. Then each of the two edges $(t_{n+1}, t_{n+2})$ and $(t_{n+1}, t_{n+3})$ must be shortened by at least $\ell$. That requires a cost of at least $2\ell$ which is impossible. So, $\ell$ must be 0, i.e., the only possible way is to change a path from $t_1$ to $t_{n+1}$, consisting of literal edges only, to be of zero length. Such a path consists of one and only one $\{E_i^+, E_i^-\}$ for each $i = 1, 2, \cdots n$, where $E_i^+ = \{(t_i, u_i), (u_i, t_{i+1})\}$

and $E_i^- = \{(t_i, v_i), (v_i, t_{i+1})\}$. This path corresponds to a truth assignment of the SATISFIABILITY problem. That is, if $E_i^+$ is used, we assign $x_i$ the value *true*; otherwise $x_i = false$.

We now show that such a truth assignment can guarantee that each clause $C_j$ is *true*. In fact, for any clause vertex $q_j$, if all the literal vertices connecting to it are not on the above mentioned path from $t_1$ to $t_{n+1}$, then the length of any path between $t_1$ and $q_j$ is at least 2, a contradiction. Hence we know that at least one literal vertex connecting to the clause vertex $q_j$ must be on the path. The literal vertex corresponds to a *true* literal of clause $C_j$, and hence clause $C_j$ is *true*. Therefore, we conclude that the SATISFIABILITY problem is satisfiable.

As the number of vertices is $m + 3n + 3$, and the number of edges does not exceed $mn + 4n + 2$, the transformation from the SATISFIABILITY problem to the decision problem of the reverse center location problem is a polynomial reduction. Thus the reverse center location problem is NP–hard. The proof is completed. □

The proof is in fact also true for directed graphs. But if we only consider directed graphs, the proof can be slightly simpler.

## 3 A Mixed Integer Program Formulation

Our next aim is to formulate the RCL problem as a mixed–integer linear program (MIP).

Let $T$ be a shortest path tree rooted at $s$ under some weight $w^*$, and denote by $\pi(v)$ the length of the path from $s$ to $v$ on $T$, then it is straightforward to observe that $\pi(v) = d_{w^*}(s, v)$. Therefore under $w^*$, $d_{w^*}(s, v) \leq l(v)$ holds for all $v \in V \backslash \{s\}$ if and only if the shortest path tree $T$ with $s$ as the root meets the condition $\pi(v) \leq l(v)$ for all $v \in V \backslash \{s\}$.

¿From Bellman's equation [12], we know that, $T$ is a shortest path tree rooted at $s$ under $w^*$ if and only if

$$\begin{array}{ll} \pi(s) = 0 \ , & \\ \pi(v) - \pi(u) = w^*(e) & \forall e = (u, v) \in T \ , \\ \pi(v) - \pi(u) \leq w^*(e) & \text{otherwise} \ , \end{array}$$

where the tree is oriented from the root to all leave vertices.

For any spanning tree $T$ which is rooted at $s$, we can use zero-one variables to distinct the edges on $T$ or not, for example,

$$x(e) = \begin{cases} 0 & \forall e \in T \ , \\ 1 & \text{otherwise} \ . \end{cases}$$

¿From the above analysis, we have

**Theorem 2** *Let $(\pi, x)$ be a feasible solution of the following system of inequalities and equations,*

$$\begin{cases} \pi(v) - \pi(u) \le w^*(e) & \forall e = (u,v) \in E\ , \\ \pi(v) - \pi(u) + Mx(e) \ge w^*(e) & \forall e = (u,v) \in E\ , \\ \pi(s) = 0\ , \\ \sum\limits_{e \in E} x(e) = |E| + 1 - |V|\ , \\ \sum\limits_{e \in C} x(e) \ge 1 & \forall \text{ cycle } C\ , \\ x(e) \in \{0,1\} & \forall e \in E\ . \end{cases} \tag{1}$$

*Then $T = \{e \in E | x(e) = 0\}$ corresponds to a shortest path tree rooted at $s$ under $w^*$, where $M$ is a sufficiently large number.*

Obviously, $M = \sum\limits_{e \in E} w(e)$ is big enough, where $w$ is the original weight vector. By Theorem 2, the reverse center location problem can be formulated as the following MIP problem,

$$\begin{aligned} \min\ & \sum_{e \in E} c(e)\theta(e) \\ s.t.\ & \\ & \pi(v) - \pi(u) + \theta(e) \le w(e) && \forall e = (u,v) \in E\ , \\ & \pi(v) - \pi(u) + \theta(e) + Mx(e) \ge w(e) && \forall e = (u,v) \in E\ , \\ & \pi(s) = 0\ , \\ & \pi(v) \le l(v) && \forall v \in V \backslash \{s\}\ , \\ & \sum_{e \in E} x(e) = |E| - |V| + 1\ , \\ & \sum_{e \in C} x(e) \ge 1 && \forall \text{ cycle } C\ , \\ & 0 \le \theta(e) \le b(e) && \forall e \in E\ , \\ & x(e) \in \{0,1\} && \forall e \in E\ . \end{aligned} \tag{2}$$

## 4 A Heuristic Algorithm

Due to the above MIP formulation, we can design a heuristic which is based on a LP relaxation. First let us consider the linear programming relaxation of problem (2). That is, we replace the last constraint, i.e., the integer requirement: $x(e) \in \{0,1\}$ by bounded continuous variables: $0 \le x(e) \le 1$. However, since problem (2) includes one constraint for each cycle in $G$, the number of these constraints may be of an exponential order of $|V|$. To enumerate all the cycles is an exhausting job. So, in order to solve the LP relaxation quickly, we adopt a cutting plane method proposed by J. Leung [13].

For a collection $\mathcal{C}$ of cycles in $G$, let $RLP(\mathcal{C})$ be the linear program:

$$
\begin{aligned}
&\min \sum_{e\in E} c(e)\theta(e) \\
&s.t. \\
&\quad \pi(v)-\pi(u)+\theta(e)\le w(e) && \forall e=(u,v)\in E\,, \\
&\quad \pi(v)-\pi(u)+\theta(e)+Mx(e)\ge w(e) && \forall e=(u,v)\in E\,, \\
&\quad \pi(s)=0\,, \\
&\quad \pi(v)\le l(v) && \forall v\in V\backslash\{s\}\,, \\
&\quad \sum_{e\in E} x(e)=|E|-|V|+1\,, \\
&\quad \sum_{e\in C} x(e)\ge 1 && \forall C\in\mathcal{C}\,, \\
&\quad 0\le\theta(e)\le b(e) && \forall e\in E\,, \\
&\quad 0\le x(e)\le 1 && \forall e\in E\,.
\end{aligned}
\tag{3}
$$

The following cutting plane method [13] can be used to solve the LP relaxation problem (3).

**Algorithm C**:
Step 1. $\mathcal{C}=\emptyset$.
Step 2. Solve $RLP(\mathcal{C})$, denote by $x^*$ the optimal solution.
Step 3. Determine if there is a cycle $C$ such that

$$\sum_{e\in C} x^*(e)<1\,.$$

Let $\mathcal{C}'$ be the collection of such cycles.
Step 4. If $\mathcal{C}'=\emptyset$, stop, the LP relaxation is solved. Otherwise, let $\mathcal{C}\leftarrow\mathcal{C}\cup\mathcal{C}'$, and return to Step 2.

In Step 3, we can use the algorithm below, given by Grötschel, Jünger and Reinelt [9], to find $\mathcal{C}'$.

**Algorithm GJR**:
Step 1. $\mathcal{C}'=\emptyset$.
Step 2. For each edge $e=(u,v)$, using $x^*$ as a weight vector to compute the distance $d_{x^*}(u,v)$ in the subgraph $G\setminus\{e\}$.
Step 3. For each pair $u$ and $v$, if $e=(v,u)\in E$ and $x^*(e)+d_{x^*}(u,v)<1$, then let $C$ be the cycle consisting of the edge $e$ and the edges which realize the distance $d_{x^*}(u,v)$ between $u$ and $v$. If $C\notin\mathcal{C}'$, add $C$ to $\mathcal{C}'$.

If the optimal solution $x^*$ of the LP relaxation of problem (2) is integer-valued, this solution is indeed the optimal solution of the reverse center location problem and we are done. Otherwise, we need to go further to find a heuristic solution for the RCL problem.

¿From Theorem 2, we know that, the optimal solution of problem (2) corresponds to a shortest path tree, (we may call it the optimal shortest path tree), and we need to change only the weights on the tree, meanwhile keep other weights unchanged. Also, on this tree, the total value of $x^*$ equals zero which is of course the smallest total $x^*$ value in all $s-$rooted spanning trees. So, if we

set the optimal solution $x^*$ of the LP relaxation as the weight vector of $E$, it is reasonable to assume that the minimum spanning tree $T$ (rooted at $s$) is a 'good' approximation of the optimal shortest path tree. Then we can restrict our consideration on $T$ only, solve problem (2) with $T$ replacing $E$ (we call it the approximate RCL problem), and take its solution as a heuristic solution of problem (2).

## 5 The Dual Problem and Its Solution

When we substitute $E$ with $T$, problem (2) is reduced to the following linear program,

$$\begin{aligned} &\min \sum_{e\in E} c(e)\theta(e) \\ &s.t. \\ &\quad \pi(v)-\pi(u)+\theta(e)=w(e) \quad \forall e=(u,v)\in T\ , \\ &\quad \pi(s)=0\ , \\ &\quad \pi(v)\le l(v) \quad \forall v\in V\backslash\{s\}\ , \\ &\quad 0\le\theta(e)\le b(e) \quad \forall e\in T\ . \end{aligned} \tag{4}$$

Although problem (4) can be solved by any LP method, we can present a strongly polynomial combinatorial algorithm with complexity $O(|V|^2)$ to solve it.

We can regard $T$ as a directed tree with the orientation from the root $s$ to all leave vertices. For each $v\in V$, let $P(v)$ denote the path from $s$ to $v$ on the tree $T$, $P(u,v)$ the subpath of $P(v)$ starting at $u$ for any vertex $u$ on $P(v)$ and $r(v):=\sum_{e\in P(v)} w(e)-l(v)$. A vertex $v$ is called a violating vertex if $r(v)>0$, i.e., the distance from $s$ to $v$ under the original weight vector $w$ is greater than the upper bound $l(v)$. Denote by $V^*\subset V$ the set of violating vertices. Then the problem (4) can be re-written as

$$\begin{aligned} &\min \sum_{e\in E} c(e)\theta(e) \\ &s.t. \\ &\quad \sum_{e\in P(v)} \theta(e)\ge r(v) \quad \forall v\in V^*\ , \\ &\quad 0\le\theta(e)\le b(e) \quad \forall e\in T\ , \end{aligned} \tag{5}$$

which is the approximate reverse center location problem we are going to solve.

The dual of problem (5) can be written as

$$\begin{aligned} &\max \sum_{v\in V^*} r(v)y(v)-\sum_{e\in E} b(e)z(e) \\ &s.t. \\ &\quad \textstyle\sum\{y(v)\mid P(v)\ni e,\, v\in V^*\}-z(e)\le c(e) \quad \forall e\in T\ , \\ &\quad y(v)\ge 0 \quad \forall v\in V^*\ , \\ &\quad z(e)\ge 0 \quad \forall e\in T\ . \end{aligned} \tag{6}$$

Without loss of generality, we may assume that all leave vertices of $T$ are violating vertices, for otherwise we can remove such vertices and the incident arcs from $T$ and let the values of $\theta$ on these arcs be zero.

Now construct an auxiliary network $N = (V^+, A^+, c^+, k^+)$ of $T$, in which $V^+ = V \cup \{t\}$ is the vertex set of $N$, $A^+ = \{e^1 = (u, v), e^2 = (u, v) \mid e = (u, v) \in T\} \cup \{(v, t) \mid v \in V^*\} \cup \{(t, s)\}$ is the arc set of $N$. $c^+$ and $k^+$ are capacity vector and cost vector of $N$ respectively, which are defined as follows.

$$c^+(e) = \begin{cases} c(u, v) & \forall e = e^1 = (u, v) \in A^+ , \\ +\infty & \text{otherwise} . \end{cases}$$

$$k^+(e) = \begin{cases} 0 & \forall e = e^1 \in A^+ \text{ or } e = (t, s) , \\ -b(u, v) & \forall e = e^2 = (u, v) \in A^+ , \\ r(v) & \forall e = (v, t) \in A^+ . \end{cases}$$

Note that each $e \in T$ corresponds to two arcs $e^1$ and $e^2$ in $N$, and we introduce an arc from $t$ to $s$ with an infinite capacity and zero cost. We consider the maximum cost circulation problem on $N$ and can show that

**Theorem 3** *The problem* (6) *is equivalent to finding a maximum cost circulation on* $N$.

Generally speaking, to find a maximum cost flow on a network can be done by some strongly polynomial combinatorial algorithms [14,16], but due to the special structure of $N$, we can find a maximum cost circulation on $N$ much faster by the following simple algorithm. Note that in stead of obtaining the flow $f$, the algorithm gives directly the optimal solution $(y, z)$ of problem (6).

**Algorithm F**.

Step 0. Let $y(v) = 0$ for all $v \in V^*$, and $z(e) = 0$ for all $e \in T$.

Step 1. Find a maximum cost path $P$ from $s$ to $t$ on $N$.

Step 2. Let $k^+(P)$ be the cost of $P$. If $k^+(P) \leq 0$, stop, the current $(y, z)$ is the optimal solution of problem (6).

Step 3. If $k^+(P) > 0$, let $e^+ = \arg\min\{c^+(e) \mid e \in P\}$. If $c^+(e^+) = +\infty$, stop, problem (6) is unbounded and thus problem (4) is infeasible. Otherwise go to Step 4.

Step 4. Let $v \in V^*$ be the precedent vertex of $t$ on the path $P$. Set $y(v) \leftarrow y(v) + c^+(e^+)$.

Step 5. For each $e \in T$ with the corresponding $e^1$ or $e^2$ on $P$, if $e^2 \in P$, then set $z(e) \leftarrow z(e) + c^+(e^+)$; if $e^1 \in P$, then set $c^+(e^1) \leftarrow c^+(e^1) - c^+(e^+)$. If $c^+(e^1) = 0$, delete $e^1$ from $N$. Return to Step 1.

Since $N \setminus \{(t, s)\}$ is acyclic, we can find the maximum cost $s - t$ path in $O(|A^+|)(= O(|V|))$ operations. Moreover, at each iteration, at least one arc with finite capacity is deleted (unless $c^+(e^+) = +\infty$ which terminates the algorithm), hence there are at most $|V| - 1$ iterations since the number of arcs with finite capacity is $|V| - 1$. Therefore, we can conclude that Algorithm F runs in $O(|V|^2)$ operations.

## 6 Recovering the Primal Optimal Solution

Now we consider how to recover an optimal solution of the primal problem (5) from an optimal solution of the dual problem (6). Let $(y^*, z^*)$ be the optimal solution of the dual problem (6). Then the primal problem (5) also has an optimal solution, denoted by $\hat{\theta}$. In fact as we already mentioned, the components of $\hat{\theta}$ which correspond to the edges not on $T$ are zero. So, we only need to determine $\hat{\theta}$ on $T$. By the complementary slackness theorem of linear programming, $\theta^* \in R^T$ is an optimal solution of (5) if and only if

$$\sum_{e \in P(v)} \theta^*(e) = r(v) \qquad \forall v \in V^* \quad \text{and} \quad y^*(v) > 0 \,, \tag{7}$$

$$\sum_{e \in P(v)} \theta^*(e) \geq r(v) \qquad \forall v \in V^* \quad \text{and} \quad y^*(v) = 0 \,, \tag{8}$$

$$0 \leq \theta^*(e) \leq b(e) \qquad \forall e \in T \,, \tag{9}$$

$$\theta^*(e) = 0 \qquad \forall e \in T \quad \text{and} \quad \sum_{P(v) \ni e} y^*(v) - z^*(e) < c(e) \,, \tag{10}$$

$$\theta^*(e) = b(e) \qquad \forall e \in T \quad \text{and} \quad z^*(e) > 0 \,. \tag{11}$$

Now we try to solve the above inequality system. Due to its special structure, we are able to present an algorithm which can solve the problem in $O(|V| \log(|V|))$ operations.

For each edge $e$ satisfying one of the strict inequalities in (10) and (11), we can have a pre-process as follows.

If $e = (u, v) \in T$ satisfies the strict inequality in (10), then we can set $\theta^*(e) = 0$, and contract the arc to vertex $u$ and set $r(u) \leftarrow \max\{r(u), r(v)\}$. It is easy to see that the feasible solution of the resulting system corresponds to a feasible solution of the original system, and vice versa. Similarly, if $e = (u, v) \in T$ satisfies the strict inequality in (11), we can set $\theta^*(e) = b(e)$ and then contract the arc to vertex $u$, set $r(u) \leftarrow \max\{r(u), r(v) - b(e)\}$, and set $r(q) \leftarrow r(q) - b(e)$ for any successive vertex $q$ of $v$.

After this pre-process, we can assume that there is no arcs on $T$ satisfying one of the two strict inequalities in (10) and (11).

Let $V^s = \{v \in V^* \mid y^*(v) > 0\}$. We are now ready to present the algorithm to solve the inequality system.

**Algorithm R**.

Step 0. Sort the vertices in $V^s$ into $v_1, v_2, \ldots, v_k$ (renumbering if necessary) by the nondecreasing order of $r(v)$ with the additional condition that if $r(v_i) = r(v_j)$ and $v_i$ is on $P(v_j)$, then $i < j$.
Set $T'_1 = \{s\}$, $p(s) = 0$ and $j = 1$.

Step 1. If $j = k + 1$, put $\theta^*(e) = b(e)$ for all arcs $e \in T \backslash T'_{k+1}$, stop. The optimal solution has been obtained.

Step 2. Scan vertex $v_j$. That is, let $s'$ be the last vertex of $P(v_j)$ belonging to

$T'_j$. Along the path $P(s', v_j)$, one arc by one arc, say $e = (u, v)$ being the current arc, let

$$p(v) \leftarrow \min\{p(u) + b(e), r(v_j)\} \tag{12}$$

and $\theta^*(e) = p(v) - p(u)$.

Step 3. Let $T'_{j+1} \leftarrow T'_j \cup P(s', v_j)$ and $j \leftarrow j + 1$. Return to Step 1.

**Theorem 4** *The vector $\theta^*$ generated by Algorithm R meets conditions* (7)-(11)*, and thus is an optimal solution to problem* (5)*.*

Let us analyze the complexity of Algorithm R now. It is easily seen that the ordering of the vertices in $V^s$ can be done in $O(|V^s| \log(|V^s|)$ operations, and each vertex is handled only once. Notice that the ordering of the vertices in $V^s$ dominates the computational complexity, and hence the total complexity is at most $O(|V| \log(|V|))$.

## 7 Conclusions

We now summarize our complete heuristic algorithm for solving the reverse center location problem.

**Algorithm H**.
Input: $G = (V, E, w)$, $s, l, b$ .
Output: $w^*$ .
Step 1. Employ Algorithm C to find the optimal solution $x^*$ of the linear programming relaxation of problem (2).
Step 2. Use $x^*$ as weights on $G$ to find a minimum spanning tree $T$ on $G$.
Step 3. Use Algorithm F to obtain a dual optimal solution $(y^*, z^*)$.
Step 4. Use Algorithm R to recover the optimal solution $\theta^*$ of problem (5). Then $w^* = w - \theta^*$ is an approximate solution of problem (2).

Let us now return to the general case of inverse optimization. There are several types of inverse problems to be put forward. The first type is that a feasible solution is given, and we need to adjust the weights as less as possible such that the given one becomes an optimal solution. So far this type of inverse problems receives most attention, see [2,3,11], [17]-[27]. This type of inverse problems is often polynomially solvable if the original problems is so (a general result can be seen in [19]). In fact strongly polynomial methods are often available if inverse *network* problems are concerned. However, there are still exceptional cases in which although the original problems are polynomially solvable, its first type inverse problem is NP-hard. See for example [4].

The second type of inverse problems does not require the given feasible solution to become an optimal solution, or no particular feasible solution is specified,

and we only require that after the adjustment of weights the new optimal solution ensures that its corresponding optimal value meets some given bound requests. The problems discussed in this paper and in [1] are of this type. The results of these two papers indicate that this type of inverse problems are often NP-hard (to distinct the two types of problems verbally, we call the first type of problems *inverse problems* and the second type *reverse problems*). So, to solve this type of problems is more challenging and remains to be one of the major issues in the study of inverse optimization problems. Hopefully, recent progress in developing approximation algorithms for NP-hard problems, see for example [10], shall help solve this type of inverse problems.

In fact there is also even more difficult type of inverse problems, see for example [7]. Indeed, inverse optimization problem is still a widely open and dynamic field.

## References

1. Burton, D., Pulleyblank, W.R., Toint, Ph.L.: The inverse shortest paths with upper bounds on shortest path costs. Report 93/03, (1993), Department of Mathematics, Facultes, Univeristaires ND de la Paix, B-5000 Nemur, Belgium
2. Burton, D., Toint, Ph.L.: On an instance of the inverse shortest paths problem. Mathematical Programming **53** (1992) 45-61
3. Cai, M., Li, Y.: Inverse matroid intersection problem. ZOR Mathematical Methods of Operations Research **45** (1997) 235-243
4. Cai, M., Yang, X., Zhang, J.: The complexity analysis of the inverse center location problem. Journal of Global Optimization (to appear)
5. Christofides, N.: Graph Theory: An Algorithmic Approach. Academic Press Inc. (London) Ltd, (1975)
6. Cook, S.A.: The complexity of theorem proving procedures. Proc. 3rd ACM Symp. on the Theory of Computing, ACM(1971), 151–158
7. Fekete, S., Kromberg, S., Hochstattler, W., Moll, C.: The Complexity of an Inverse Shortest Paths Problem. Working paper (1993), University of Cologne.
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide of the Theory of NP-Completeness. Freeman, San Francisco, (1979)
9. Grötschel, M., Jünger, M., Reinelt, G.: On the acyclic subgraph polytope. Mathematical Programming **33** (1985) 28–42
10. Hochbaum, D.(eds.): Approximation Algorithms for NP-hard Problems. PWS, Boston, 1997
11. Hu, Z., Liu, Z.: A strongly polynomial algorithm for the inverse shortest arborescence problem. Discrete Applied Mathematics **82** (1998) 135-154
12. Lawler, E.L.: Combinatorial Optimization: Networks and Matroids. Holt, Rinehart and Winston, (1976)
13. Leung, L.: A graph-theoretic heuristic for designing loop-layout manufacturing systems. European Journal of Operational Research, **57** (1992) 243–252
14. Orlin, J.B.: A faster strongly polynomial minimum cost flow algorithm. Proc. 20th ACM Symp. on the Theory of Comp., (1988), 377–387
15. Papadimitriou, C.H., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall Inc. Englewood Cliffs, New Jersey, (1982)

16. Tardos, É.: A strongly polynomial minimum cost circulation algorithm. Combinatorica **5** (1985) 247–255
17. Yang, C., Zhang, J.: Inverse maximum flow and minimum cut problems. Optimization **40** (1997) 147-170
18. Yang, C., Zhang, J.: Inverse maximum capacity problem. Operations Research Spektrum (to appear)
19. Yang, C., Zhang, J.: Two general methods for inverse optimization problems. Applied Mathematics Letters (to appear)
20. Zhang, J., Cai, M.: Inverse problem of minimum cuts. ZOR Mathematical Methods of Operations Research **48** (1998) 51-58
21. Zhang, J., Liu, Z.: Calculating some inverse linear programming problem. Journal of Computational and Applied Mathematics **72** (1996) 261-273
22. Zhang, J., Liu, Z., Ma, Z.: On inverse problem of minimum spanning tree with partition constraints. ZOR Mathematical Methods of Operations Research **44** (1996) 347-358
23. Zhang, J., Liu, Z., Ma, Z.: Inverse Fractional Matching Problem. The Journal of Australia Mathematics Society, Ser. B: Applied Mathematics (to appear)
24. Zhang, J., Ma, Z.: A network flow method for solving some inverse combinatorial optimization problems. Optimization **37** (1996) 59–72
25. Zhang, J., Ma, Z.: Solution structure of some inverse optimization problems. Journal of Combinatorial Optimization (to appear)
26. Zhang, J., Ma, Z., Yang, C.: A column generation method for inverse shortest path problem. ZOR Mathematical Methods of Operations Research **41** (1995) 347-358
27. Zhang, J., Xu, S., Ma, Z.: An algorithm for inverse minimum spanning tree problem. Optimization Methods and Software **8** (1997) 69-84

## Appendix A: Proof of Theorem 3

First, suppose $f$ is a maximum cost circulation on $N$, let us define $y(v) = f(v,t)$ for any $v \in V^*$, and $z(e) = f(e^2)$ for any $e \in T$, then we can verify that $(y,z)$ is a feasible solution of (6). Also, the maximum cost is

$$\sum_{e \in A^+} k^+(e) f(e) = \sum_{v \in V^*} r(v) y(v) - \sum_{e \in E} b(e) z(e) \ ,$$

which is just the objective value of problem (6) at the feasible solution $(y,z)$.

Conversely, if $(y,z)$ is an optimal solution of (6), we have

$$z(e) = \max\{0, \ \sum\{y(v) \mid P(v) \ni e, v \in V^*\} - c(e)\} \ .$$

Define

$$\begin{array}{ll} f(v,t) = y(v) & \forall v \in V^* \ , \\ f(e^1) = \min\{c(e), \sum\{y(v) \mid P(v) \ni e, v \in V^*\}\} & \forall e \in T \ , \\ f(e^2) = z(e) & \forall e \in T, \\ f(t,s) = \sum\limits_{v \in V^*} f(v,t) \ . & \end{array}$$

As for any $e \in T$, $f(e^1) \le c(e)$, the capacity requirement is met. Also, no matter $c(e) \le \sum y(v)$ or not, we always have that for each $e \in T$, the flow

$$f(e^1) + f(e^2) = \sum\{y(v) \mid P(v) \ni e, v \in V^*\} \ .$$

from which it is easy to see that $f$ is a circulation on $N$, and the total cost of the flow $f$ equals the maximum objective value of (6).

Therefore, the two problems have the same optimum value, and the optimal solution of problem (6) can be obtained easily once we solved the maximum cost circulation problem. The theorem is proved. □

## Appendix B: Remarks for Algorithm R

**Remark 1.** In the algorithm we obtain $\theta^*$ by treating the tree $T$ piece by piece. After scanning a $v_j$ in $V^s$, the arcs on $P(s', v_j)$ have been assigned their $\theta^*$ values. $p(v)$ represents the total decrement of the weights on $P(s,v)$, and $T'_j$ stands for the subtree which has already been processed after scanning the first $j-1$ vertices $v_1, \ldots, v_{j-1}$. It is possible that after scanning all vertices in $V^s$, $T'_k$ is still a proper subset of $T$. Under such case, we assign $\theta^*(e) = b(e)$ for all arcs $e \in T \setminus T'_k$.

**Remark 2.** ¿From the algorithm, it is easy to see that

$$\sum_{e \in P(v)} \theta^*(e) = p(v) \quad \forall v \in V \ .$$

**Remark 3.** An arc $e \in T'_k$ is called **saturated** if $\theta^*(e) = b(e)$ and **unsaturated** otherwise. Suppose $e = (u, v) \in T'_j$, then the value $\theta^*(e)$ is assigned when one of $\{v_1, \ldots, v_{j-1}\}$ is scanned. Say, the arc is processed when we scan $v_i$, $(i \leq j-1)$. Then $P(v_i)$ passes through $e$. If this arc is unsaturated, then by the formula (12), we have $p(v) = r(v_i)$ which means for any vertex $x$ on the path $P(v, v_i)$, $p(x) = r(v_i)$, and thus for any arc along $P(v, v_i)$, the value of $\theta^*$ must be zero.

## Appendix C: Proof of Theorem 4

As we assume a pre-process has been taken before the Algorithm R is employed, we only need to consider (7) − (9).

If $k = 0$, i.e., $V^s = \emptyset$, then (7) does not occur, and by Step 1, $\theta^*(e) = b(e)$ for all arcs $e$ of $T$, which means (8) and (9) hold. So, in the proof below, we assume $k > 0$.

We now show that for all arcs and vertices which are processed when we scan $v_j$, $j = 1, \ldots, k$, they must satisfy (7)–(9).

We consider any $j \leq k$. Recall that $s'$ is the last vertex of $P(v_j)$ belonging to $T'_j$.

We begin with proving (9). For each $e \in P(s', v_j)$, by Step 2, of course $\theta^*(e) \leq b(e)$. To prove $\theta^*(e) \geq 0$ we only need to show that $p(s') \leq r(v_j)$. Assuming $p(s')$ is defined when we scan $v_i$, then $i < j$, implying $p(s') \leq r(v_i) \leq r(v_j)$. Thus (9) holds.

Next let us show (7). By formula (12), $p(v_j) \leq r(v_j)$. Assuming $p(v_j) < r(v_j)$, then there exists at least one unsaturated arc on $P(s')$. Let $e = (u, v)$ be the last such arc. Then by Remark 3 there exists $v_i \in V^s$, $i < j$, such that $P(v_i)$ passes through $e$ and $p(v) = r(v_i)$. Let $\hat{\theta}$ be the optimal solution of problem (5) which we mentioned at the beginning of this section, and thus for any $v_j \in V^s$, $\hat{\theta}$ satisfies the inequality in (7). So, we have

$$\begin{aligned} r(v_j) &= \sum_{e \in P(v_j)} \hat{\theta}(e) \\ &= \sum_{e \in P(v)} \hat{\theta}(e) + \sum_{e \in P(v, v_j)} \hat{\theta}(e) \\ &\leq \sum_{e \in P(v_i)} \hat{\theta}(e) + \sum_{e \in P(v, v_j)} b(e) \\ &= r(v_i) + \sum_{e \in P(v, v_j)} b(e) \\ &= p(v) + \sum_{e \in P(v, v_j)} b(e) \\ &= p(v_j) \ , \end{aligned}$$

a contradiction. Hence $p(v_j) = r(v_j)$, i.e., (7) is true for every $v_j \in V^s$.

We then turn to prove (8). For each vertex $v' \in V^*$ on $P(s', v_j) \setminus \{s'\}$ ($s'$ is not processed in the $j$-th scan), if all arcs of $P(v')$ are saturated, then due to the feasibility of the inverse problem, clearly (8) holds. Otherwise let $e' = (u, v)$ be the last unsaturated arc on $P(v')$. There are only two possibilities. First, if $e'$ is on $P(s', v_j)$, then

$$p(v') = p(v_j) = r(v_j) = \sum_{e \in P(v_j)} \hat{\theta}(e) \geq \sum_{e \in P(v')} \hat{\theta}(e) \geq r(v') \ .$$

That means $v'$ satisfies the inequality in (8). Second, if $e'$ is on $P(s')$, then again by Remark 3, there exists $v_i \in V^s$, $i < j$, such that $P(v_i)$ passes through $e'$ and $p(v) = r(v_i)$. We have

$$\begin{aligned} r(v') &\leq \sum_{e \in P(v')} \hat{\theta}(e) \\ &= \sum_{e \in P(v)} \hat{\theta}(e) + \sum_{e \in P(v, v')} \hat{\theta}(e) \\ &\leq \sum_{e \in P(v_i)} \hat{\theta}(e) + \sum_{e \in P(v, v')} b(e) \\ &= r(v_i) + \sum_{e \in P(v, v')} b(e) \\ &= p(v) + \sum_{e \in P(v, v')} b(e) \\ &= p(v') \ . \end{aligned}$$

So, (8) always holds for any vertex $v' \in V^*$ which has been processed when we scan $v_j$.

Finally we consider the arcs and vertices which are not contained in $T'_{k+1}$, i.e., the case $j = k + 1$. By Step 1, all arcs $e \in T \backslash T'_k$ have $\theta^*(e) = b(e)$ and thus (9) holds.

For all vertices $q \in V^* \bigcap \{T \backslash T'_k\}$, there are two possible cases. First, there is a vertex $v \in V^s$ on $P(q)$. Let $v^*$ be the last such vertex. Then

$$\begin{aligned} r(q) &\leq \sum_{e \in P(q)} \hat{\theta}(e) \\ &\leq \sum_{e \in P(v^*)} \hat{\theta}(e) + \sum_{e \in P(v^*, q)} b(e) \\ &= r(v^*) + \sum_{e \in P(v^*, q)} b(e) \\ &= \sum_{e \in P(q)} \theta^*(e) \ , \end{aligned}$$

i.e., (8) is true for this vertex. Second, there is no such vertex. In this case (8) holds trivially as all arcs on $P(q)$ are saturated.

The proof of the theorem is completed. □

# Performance Comparison of Linear Sieve and Cubic Sieve Algorithms for Discrete Logarithms over Prime Fields

Abhijit Das and C.E. Veni Madhavan

Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560 012, India
abhij,cevm@csa.iisc.ernet.in

**Abstract.** It is of interest in cryptographic applications to obtain practical performance improvements for the discrete logarithm problem over prime fields $\mathbb{F}_p$ with $p$ of size $\leq 500$ bits. The linear sieve and the cubic sieve methods described in Coppersmith, Odlyzko and Schroeppel's paper [3] are two practical algorithms for computing discrete logarithms over prime fields. The cubic sieve algorithm is asymptotically faster than the linear sieve algorithm.
We discuss an efficient implementation of the cubic sieve algorithm incorporating two heuristic principles. We demonstrate through empirical performance measures that for a special class of primes the cubic sieve method runs about two to three times faster than the linear sieve method *even* in cases of *small* prime fields of size about 150 bits.

## 1 Introduction

Computation of discrete logarithms over a finite field $\mathbb{F}_q$ is a difficult problem. No algorithms are known that solve the problem in time polynomially bounded by the size of the field (i.e. $\log q$). The index calculus algorithm [3,7,9,10,11] is currently the best known algorithm for this purpose and has a sub-exponential expected running time given by

$$L\langle q, \gamma, c\rangle = \exp\left((c + o(1))(\log q)^{\gamma}(\log\log q)^{1-\gamma}\right)$$

for some constant $c$ and for some real number $0 < \gamma < 1$. For practical applications, one typically uses prime fields or fields of characteristic 2. In this paper, we focus on prime fields only.

Let $\mathbb{F}_p$ be a prime field of cardinality $p$. For an element $a \in \mathbb{F}_p$, we denote by $\overline{a}$ the representative of $a$ in the set $\{0, 1, \ldots, p-1\}$. Let $g$ be a primitive element of $\mathbb{F}_p$ (i.e. a generator of the cyclic multiplicative group $\mathbb{F}_p^*$). Given an element $a \in \mathbb{F}_p^*$, there exists a unique integer $0 \leq x \leq p-2$ such that $a = g^x$ in $\mathbb{F}_p$. This integer $x$ is called the *discrete logarithm* or *index* of $a$ in $\mathbb{F}_p$ with respect to $g$ and is denoted by $\mathrm{ind}_g(a)$. The determination of $x$ from the knowledge of $p$, $g$ and $a$ is referred to as the *discrete logarithm problem*. In general, one need not assume $g$ to be a primitive element and is supposed to compute $x$ from $a$ and

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 295–306, 1999.
© Springer-Verlag Berlin Heidelberg 1999

$g$, if such an $x$ exists (i.e. if $a$ belongs to the cyclic subgroup of $\mathbb{F}_p^*$ generated by $g$). In this paper, we always assume for simplicity that $g$ is a primitive element of $\mathbb{F}_p$.

In what follows we denote by $L(p, c)$ any quantity that satisfies

$$L(p, c) = L\langle p, 1/2, c\rangle = \exp\left((c + o(1))\sqrt{\ln p \ln\ln p}\right),$$

where $c$ is a positive constant and $\ln x$ is the natural logarithm of $x$.[1] When $p$ is understood from the context, we write $L[c]$ for $L(p, c)$. In particular, $L[1]$ is denoted simply by $L$.

The naï ve index calculus algorithm [10, Section 6.6.2] for the computation of discrete logarithms over prime fields and the adaptations of this algorithm take time $L[c]$ for $c$ between 1.5 and 2 and are not useful in practice for prime fields $\mathbb{F}_p$ with $p > 2^{100}$. Coppersmith, Odlyzko and Schroeppel [3] proposed three variants of the index calculus method that run in time $L[1]$ and are practical for $p \leq 2^{250}$. A subsequent paper [7] by LaMacchia and Odlyzko reports implementation of two of these three variants, namely the linear sieve method and the Gaussian integer method. They were able to compute discrete logarithms in $\mathbb{F}_p$ with $p$ of about 200 bits.

The paper [3] also describes a cubic sieve algorithm due to Reyneri for the computation of discrete logarithms over prime fields. The cubic sieve algorithm has a heuristic running time of $L[\sqrt{2\alpha}]$ for some $\frac{1}{3} \leq \alpha < \frac{1}{2}$ and is, therefore, asymptotically faster than the linear sieve algorithm (and the other $L[1]$ algorithms described in [3]). However, the authors of [3] conjectured that the theoretical asymptotics do not appear to take over for $p$ in the range of practical interest (a few hundred bits). A second problem associated with the cubic sieve algorithm is that it requires a solution of a certain Diophantine equation. It's not known how to find a solution of this Diophantine equation in the general case. For certain special primes $p$ a solution arises naturally, for example, when $p$ is *close* to a whole cube.

Recently, a new variant of the index calculus method based on general number field sieves (NFS) has been proposed [6] and has a conjectured heuristic run time of

$$L\langle p, 1/3, c\rangle = \exp\left((c + o(1))(\log p)^{\frac{1}{3}}(\log\log p)^{\frac{2}{3}}\right).$$

Weber et. al. [12,14,15] have implemented and proved the practicality of this method. Currently the NFS-based methods are known to be the fastest algorithms for solving the discrete logarithm problem over prime fields.

In this paper, we report efficient implementations of the linear sieve and the cubic sieve algorithms. To the best of our knowledge, ours is the first large-scale implementation of the cubic sieve algorithm. In our implementation, we employ ideas similar to those used in the quadratic sieve algorithm for integer factorization [1,5,13]. Our experiments seem to reveal that the equation collecting phase of the cubic sieve algorithm, whenever applicable, runs faster than that in the linear sieve algorithm.

[1] We denote $\log x = \log_{10} x$, $\ln x = \log_e x$ and $\lg x = \log_2 x$.

In the next two sections, we briefly describe the linear sieve and the cubic sieve algorithms. Performance of our implementation and comparison of the two algorithms for a randomly chosen prime field are presented in Sections 4 and 5. Our emphasis is not to set a record on the computation of discrete logarithms, but to point out that our heuristic principles really work in practical situations. We, therefore, experimented with a *small* prime (of length around 150 bits). Even for this field we get a performance gain between two and three. For larger prime fields, the performance improvement of the cubic sieve method over the linear sieve method is expected to get accentuated. We conclude the paper in Section 6.

## 2 The Linear Sieve Algorithm

The first stage for the computation of discrete logarithms over a prime field $\mathbb{F}_p$ using the currently known subexponential methods involves calculation of discrete logarithms of elements of a given subset of $\mathbb{F}_p$, called the *factor base*. To this end, a set of linear congruences are solved modulo $p-1$. Each such congruence is obtained by checking the factorization of certain integers computed deterministically or randomly. For the linear sieve algorithm, the congruences are generated in the following way.

Let $H = \lfloor \sqrt{p} \rfloor + 1$ and $J = H^2 - p$. Then $J \leq 2\sqrt{p}$. For *small* integers $c_1, c_2$, the right side of the following congruence (henceforth denoted as $T(c_1, c_2)$)

$$(H + c_1)(H + c_2) \equiv J + (c_1 + c_2)H + c_1 c_2 \pmod{p} \tag{1}$$

is of the order of $\sqrt{p}$. If the integer $T(c_1, c_2)$ is smooth with respect to the first $t$ primes $q_1, q_2, \ldots, q_t$, that is, if we have a factorization like $J + (c_1 + c_2)H + c_1 c_2 = \prod_{i=1}^{t} q_i^{\alpha_i}$, then we have a relation

$$\operatorname{ind}_g(H + c_1) + \operatorname{ind}_g(H + c_2) = \sum_{i=1}^{t} \alpha_i \operatorname{ind}_g(q_i). \tag{2}$$

For the linear sieve algorithm, the factor base comprises of primes less than $L[1/2]$ (so that by the prime number theorem $t \approx L[1/2]/\ln(L[1/2])$) and integers $H + c$ for $-M \leq c \leq M$. The bound $M$ on $c$ is chosen such that $2M \approx L[1/2 + \epsilon]$ for some small positive real $\epsilon$. Once we check the factorization of $T(c_1, c_2)$ for all values of $c_1$ and $c_2$ in the indicated range, we are expected to get $L[1/2 + 3\epsilon]$ relations like (2) involving the unknown indices of the factor base elements. If we further assume that the primitive element $g$ is a small prime which itself is in the factor base, then we get a relation $\operatorname{ind}_g(g) = 1$. The resulting system with asymptotically more equations than unknowns is expected to be of full rank and is solved to compute the discrete logarithms of elements in the factor base.

In order to check for the smoothness of the integers $T(c_1, c_2) = J + (c_1 + c_2)H + c_1 c_2$ for $c_1, c_2$ in the range $-M, \ldots, M$, sieving techniques are used. First one fixes a $c_1$ and initializes to zero an array $\mathfrak{A}$ indexed $-M, \ldots, M$. One then

computes for each prime power $q^h$ ($q$ is a small prime in the factor base and $h$ is a small positive exponent), a solution for $c_2$ of the congruence $(H+c_1)c_2+(J+c_1H) \equiv 0 \pmod{q^h}$. If the gcd $(H+c_1, q) = 1$, i.e. if $H+c_1$ is not a multiple of $q$, then the solution is given by $d \equiv -(J+c_1H)(H+c_1)^{-1} \pmod{q^h}$. The inverse in the last equation can be calculated by running the extended gcd algorithm on $H+c_1$ and $q^h$. Then for each value of $c_2$ ($-M \leq c_2 \leq M$) that is congruent to $d \pmod{q^h}$, $\lg q$ is added[2] to the corresponding array locations $\mathfrak{A}_{c_2}$. On the other hand, if $q^{h_1} || (H+c_1)$ with $h_1 > 0$, we compute $h_2 \geq 0$ such that $q^{h_2} || (J+c_1H)$. If $h_1 > h_2$, then for each value of $c_2$, the expression $T(c_1, c_2)$ is divisible by $q^{h_2}$ and by no higher powers of $q$. So we add the quantity $h_2 \ln q$ to $\mathfrak{A}_{c_2}$ for all $-M \leq c_2 \leq M$. Finally, if $h_1 \leq h_2$, then we add $h_1 \ln q$ to $\mathfrak{A}_{c_2}$ for all $-M \leq c_2 \leq M$ and for $h > h_1$ solve the congruence as $d \equiv -\left(\frac{J+c_1H}{q^{h_1}}\right)\left(\frac{H+c_1}{q^{h_1}}\right)^{-1} \pmod{q^{h-h_1}}$.

Once the above procedure is carried out for each small prime $q$ in the factor base and for each small exponent $h$,[3] we check for which values of $c_2$, the entry of $\mathfrak{A}$ at index $c_2$ is *sufficiently close* to the value $\lg(T(c_1, c_2))$. These are precisely the values of $c_2$ such that for the given $c_1$, the integer $T(c_1, c_2)$ factorizes smoothly over the small primes in the factor base.

In an actual implementation, one might choose to vary $c_1$ in the sequence $-M, -M+1, -M+2, \ldots$ and, for each $c_1$, consider only the values of $c_2$ in the range $c_1 \leq c_2 \leq M$. The criterion for 'sufficient closeness' of the array element $\mathfrak{A}_{c_2}$ and $\lg(T(c_1, c_2))$ goes like this. If $T(c_1, c_2)$ factorizes smoothly over the small primes in the factor base, then it should differ from $\mathfrak{A}_{c_2}$ by a small positive or negative value. On the other hand, if the former is not smooth, it would have a factor at least as small as $q_{t+1}$, and hence the difference between $\lg(T(c_1, c_2))$ and $\mathfrak{A}_{c_2}$ would not be too less than $\lg q_{t+1}$. In other words, this means that the values of the difference $\lg(T(c_1, c_2)) - \mathfrak{A}_{c_2}$ for smooth values of $T(c_1, c_2)$ are well-separated from those for non-smooth values and one might choose for the criterion a check whether the absolute value of the above difference is less than 1.

This completes the description of the equation collecting phase of the first stage of the linear sieve algorithm. This is followed by the solution of the linear system modulo $p-1$. The second stage of the algorithm involves computation of discrete logarithms of arbitrary elements of $\mathbb{F}_p^*$ using the database of logarithms of factor base elements. We do not deal with these steps in this paper, but refer the reader to [3,7,8] for details.

[2] More precisely, some approximate value of $\lg q$, say, for example, the integer $\lfloor 1000 \lg q \rfloor$.

[3] The exponent $h$ can be chosen in the sequence $1, 2, 3, \ldots$ until one finds an $h$ for which none of the integers between $-M$ and $M$ is congruent to $d$.

## 3 The Cubic Sieve Algorithm

Let us assume that we know a solution of the Diophantine equation

$$\begin{aligned} X^3 &\equiv Y^2 Z \pmod{p} \\ X^3 &\neq Y^2 Z \end{aligned} \tag{3}$$

with $X, Y, Z$ of the order of $p^\alpha$ for some $\frac{1}{3} \leq \alpha < \frac{1}{2}$. Then we have the congruence

$$\begin{aligned} (X + AY)(X + BY)(X + CY) \equiv \\ Y^2 \Big[ Z + (AB + AC + BC)X + (ABC)Y \Big] \pmod{p} \end{aligned} \tag{4}$$

for all triples $(A, B, C)$ with $A + B + C = 0$. If the bracketed expression on the right side of the above congruence, henceforth denoted as $R(A, B, C)$, is smooth with respect to the first $t$ primes $q_1, q_2, \ldots, q_t$, that is, if we have a factorization $R(A, B, C) = \prod_{i=1}^{t} q_i^{\beta_i}$, then we have a relation like

$$\begin{aligned} \operatorname{ind}_g(X + AY) + \operatorname{ind}_g(X + BY) + \operatorname{ind}_g(X + CY) \equiv \\ 2 \operatorname{ind}_g(Y) + \textstyle\sum_{i=1}^{t} \beta_i \operatorname{ind}_g(q_i) \pmod{p-1} \end{aligned} \tag{5}$$

If $A$, $B$, $C$ are *small* integers, then $R(A, B, C)$ is of the order of $p^\alpha$, since each of $X$, $Y$ and $Z$ is of the same order. This means that we are now checking integers smaller than $O(p^{\frac{1}{2}})$ for smoothness over first $t$ primes. As a result, we are expected to get relations like (5) more *easily* than relations like (2) as in the linear sieve method.

This observation leads to the formulation of the cubic sieve algorithm as follows. The factor base comprises of primes less than $L[\sqrt{\alpha/2}]$ (so that $t \approx L[\sqrt{\alpha/2}]/\ln\Big(L[\sqrt{\alpha/2}]\Big)$), the integer $Y$ (or $Y^2$) and the integers $X + AY$ for $0 \leq |A| \leq M$, where $M$ is of the order of $L[\sqrt{\alpha/2}]$. The integer $R(A, B, C)$ is, therefore, of the order of $p^\alpha L[\sqrt{3\alpha/2}]$ and hence the probability that it is smooth over the first $t$ primes selected as above, is about $L[-\sqrt{\alpha/2}]$. As we check the smoothness for $L[\sqrt{2\alpha}]$ triples $(A, B, C)$ (with $A + B + C = 0$), we expect to obtain $L[\sqrt{\alpha/2}]$ relations like (5).

In order to check for the smoothness of $R(A, B, C) = Z + (AB + AC + BC)X + (ABC)Y$ over the first $t$ primes, sieving techniques are employed. We maintain an array $\mathfrak{A}$ indexed $-M \ldots + M$ as in the linear sieve algorithm. At the beginning of each sieving step, we fix $C$, initialize the array $\mathfrak{A}$ to zero and let $B$ vary. The relation $A + B + C = 0$ allows us to eliminate $A$ from $R(A, B, C)$ as $R(A, B, C) = -B(B + C)(X + CY) + (Z - C^2 X)$. For a fixed $C$, we try to solve the congruence

$$-B(B + C)(X + CY) + (Z - C^2 X) \equiv 0 \pmod{q^h} \tag{6}$$

where $q$ is a small prime in the factor base and $h$ is a small positive exponent. This is a quadratic congruence in $B$. If $X + CY$ is invertible modulo $q^h$ (i.e. modulo $q$), then the solution for $B$ is given by

$$B \equiv -\frac{C}{2} + \sqrt{(X + CY)^{-1}(Z - C^2X) + \frac{C^2}{4}} \pmod{q^h} \tag{7}$$

where the square root is modulo $q^h$. If the expression inside the radical is a quadratic residue modulo $q^h$, then for each solution $d$ of $B$ in (7), $\lg q$ is added to those indices of $\mathfrak{A}$ which are congruent to $d$ modulo $q^h$. On the other hand, if the expression under the radical is a quadratic non-residue modulo $q^h$, we have no solutions for $B$ in (6). Finally, if $X + CY$ is non-invertible modulo $q$, we compute $h_1 > 0$ and $h_2 \geq 0$ such that $q^{h_1} || (X + CY)$ and $q^{h_2} || (Z - C^2X)$. If $h_1 > h_2$, then $R(A, B, C)$ is divisible by $q^{h_2}$ and by no higher powers of $q$ for each value of $B$ (and for the fixed $C$). We add $h_2 \lg q$ to $\mathfrak{A}_i$ for each $-M \leq i \leq M$. On the other hand, if $h_1 \leq h_2$, we add $h_1 \lg q$ to $\mathfrak{A}_i$ for each $-M \leq i \leq M$ and try to solve the congruence $-B(B + C)\left(\frac{X+CY}{q^{h_1}}\right) + \left(\frac{Z-C^2X}{q^{h_1}}\right) \equiv 0 \pmod{q^{h-h_1}}$ for $h > h_1$. Since $\frac{X+CY}{q^{h_1}}$ is invertible modulo $q^{h-h_1}$, this congruence can be solved similar to (7).

Once the above procedure is carried out for each small prime $q$ in the factor base and for each small exponent $h$, we check for which values of $B$, the entry of $\mathfrak{A}$ at index $B$ is *sufficiently close* to the value $\lg(R(A, B, C))$. These are precisely the values of $B$ for which $R(A, B, C)$ is smooth over the first $t$ primes for the given $C$. The criterion of 'sufficient closeness' of $\mathfrak{A}_B$ and $\lg(R(A, B, C))$ is the same as described in connection with the linear sieve algorithm.

In order to avoid duplication of effort, we should examine the smoothness of $R(A, B, C)$ for $-M \leq A \leq B \leq C \leq M$. With this condition, it can be easily shown that $C$ varies from 0 to $M$ and for a fixed $C$, $B$ varies from $-C/2$ to $\min(C, M - C)$. Though we do not use the value of $A$ directly in the sieving procedure described above, it's useful[4] to note that for a fixed $C$, $A$ varies from $\max(-2C, -M)$ to $-C/2$. In particular, $A$ is always negative.

After sufficient number of relations are available, the resulting system is solved modulo $p - 1$ and the discrete logarithms of the factor base elements are stored for computation of individual discrete logarithms. We refer the reader to [3,7,8] for details on the solution of sparse linear systems and on the computation of individual discrete logarithms with the cubic sieve method.

Attractive as it looks, the cubic sieve method has several drawbacks which impair its usability in practical situations.

1. It is currently not known how to solve the congruence (3) for a general $p$. And even when it is solvable, how large can $\alpha$ be? For practical purposes $\alpha$ should be as close to $\frac{1}{3}$ as possible. No non-trivial results are known to the authors, that can classify primes $p$ according as the smallest possible values of $\alpha$ they are associated with.

[4] for a reason that will be clear in Section 5

2. Because of the quadratic and cubic expressions in $A$, $B$ and $C$ as coefficients of $X$ and $Y$ in $R(A, B, C)$, the integers $R(A, B, C)$ tend to be as large as $p^{\frac{1}{2}}$ even when $\alpha$ is equal to $1/3$. If we compare this scenario with that for $T(c_1, c_2)$ (See Equation (1)), we see that the coefficient of $H$ is a linear function of $c_1$ and $c_2$ and as such, the integers $T(c_1, c_2)$ are larger than $p^{\frac{1}{2}}$ by a small multiplicative factor. This shows that though the integers $R(A, B, C)$ are asymptotically smaller than the integers $T(c_1, c_2)$, the formers are, in practice, around $10^4$–$10^6$ times smaller than latter ones, even when $\alpha$ assumes the most favorable value (namely, $1/3$). In other words, when one wants to use the cubic sieve algorithm, one should use values of $t$ (i.e. the number of small primes in the factor base) much larger than the values prescribed by the asymptotic formula for $t$.
3. The second stage of the cubic sieve algorithm, i.e. the stage that involves computation of individual logarithms, is asymptotically as slow as the equation collection stage. For the linear sieve algorithm, on the other hand, individual logarithms can be computed much faster than the equation collecting phase.

In this paper, we address this second issue related to the cubic sieve algorithm. We report an efficient implementation of the cubic sieve algorithm for the case $\alpha = 1/3$, that runs faster than the linear sieve method for the same prime. Our experimentation tends to reveal that the cubic sieve algorithm, when applicable, outperforms the linear sieve method, even when the cardinality of the ground field is around 150 bits long.

## 4 An Efficient Implementation of the Linear Sieve Method

Before we delve into the details of the comparison of the linear and cubic sieve methods, we describe an efficient implementation of the linear sieve algorithm. The tricks that help us speed up the equation collecting phase of the linear sieve method are very similar to those employed in the quadratic sieve algorithm for integer factorization (See [1,5,13] for details).

We first recall that at the beginning of each sieving step, we find a solution for $c_2$ modulo $q^h$ in the congruence $T(c_1, c_2) \equiv 0 \pmod{q^h}$ for every small prime $q$ in the factor base and for a set of small exponents $h$. The costliest operation that need be carried out for each such solution is the computation of a modular inverse (namely, that of $H + c_1$ modulo $q^h$). As described in [7] and as is evident from our experiments too, calculations of these inverses take more than half of the CPU time needed for the entire equation collecting stage. Any trick that reduces the number of computations of the inverses, speeds up the algorithm.

One way to achieve this is to solve the congruence every time only for $h = 1$ and ignore all higher powers of $q$. That is, for every $q$ (and $c_1$), we check which of the integers $T(c_1, c_2)$ are divisible by $q$ and then add $\lg q$ to the corresponding indices of the array $\mathfrak{A}$. If some $T(c_1, c_2)$ is divisible by a higher power of $q$, this

strategy fails to add $\lg q$ the required number of times. As a result, this $T(c_1, c_2)$, even if smooth, may fail to pass the 'closeness criterion' described in Section 2. This is, however, not a serious problem, because we may increase the cut-off from a value smaller than $\lg q_t$ to a value $\zeta \lg q_t$ for some $\zeta \geq 1$. This means that some non-smooth $T(c_1, c_2)$ will pass through the selection criterion in addition to some smooth ones that could not, otherwise, be detected. This is reasonable, because the non-smooth ones can be later filtered out from the smooth ones and one might use even trial divisions to do so. For primes $p$ of less than 200 bits, values of $\zeta \leq 2.5$ work quite well in practice [1,13].

The reason why this strategy performs well in practice is as follows. If $q$ is small, for example $q = 2$, we should add *only* 1 to $\mathfrak{A}_{c_2}$ for every power of 2 dividing $T(c_1, c_2)$. On the other hand, if $q$ is much larger, say $q = 1299709$ (the $10^5$th prime), then $\lg q \approx 20.31$ is *large*. But $T(c_1, c_2)$ would not be, in general, divisible by a *high* power of this $q$. The approximate calculation of logarithm of the smooth part of $T(c_1, c_2)$, therefore, leads to a situation where the probability that a smooth $T(c_1, c_2)$ is actually detected as smooth is quite high. A few relations would be still missed out even with the modified 'closeness criterion', but that is more than compensated by the speed-up gained by the method.

The above strategy helps us in a way other than by reducing the number of modular inverses. We note that for practical values of $p$, the small primes in the factor base are usually single-precision ones. As a result, the computation of $d$ (See Section 2) can be carried out using single-precision operations only.

Throughout the rest of this section we compare the performance of the modified strategy with that of the original strategy for a value of $p$ of length around 150 bits. This prime is chosen as a random one satisfying the conditions (i) $(p-1)/2$ is also a prime, and (ii) $p$ is close to a whole cube. This second condition is necessary, because for these primes, the cubic sieve algorithm is also applicable, so that we can compare the performance of the two sieve algorithms for these primes. Our experiments are based on the Galois Field Library routines developed by the authors [4] and are carried out on a 200 MHz Pentium machine running Linux version 2.0.34 and having 64 Mb RAM. The GNU C Compiler version 2.7 is used.

**Table 1.** Performance of the linear sieve algorithm

$p = 132024547465630918351398872937358324284287168 3$
$t = 7000,\ M = 30000$

| Algorithm | $\zeta$ | No. of Relations ($\bar{\rho}$) | No. of Variables ($\bar{\nu}$) | $\bar{\rho}/\bar{\nu}$ | CPU Time (seconds) |
|---|---|---|---|---|---|
| Exact | 0.1 | 108637 | 67001 | 1.6214 | 225590 |
| Approximate | 1.0 | 108215 | 67001 | 1.6151 | 101712 |
| | 1.5 | 108624 | 67001 | 1.6212 | 101818 |
| | 2.0 | 108636 | 67001 | 1.6214 | 102253 |
| | 2.5 | 108637 | 67001 | 1.6214 | 102250 |

In Table 1 we compare the performance of the 'exact' version of the algorithm (where all relations are made available by choosing values of $h \geq 1$) with that of the 'approximate' version of the algorithm (in which powers $h > 1$ are neglected). The CPU times listed in the table do not include the time for filtering out the 'spurious' relations obtained in the approximate version. It is evident from the table that the performance gain obtained using the heuristic variant is more than 2. It's also clear that values of $\zeta$ between 1.5 and 2 suffice for fields of this size.

## 5 An Efficient Implementation of the Cubic Sieve Method

For the cubic sieve method, we employ strategies similar to those described in the last section. That is, we solve the congruence $R(A, B, C) \equiv 0 \pmod{q}$ for each small prime $q$ in the factor base and ignore higher powers of $q$ that might divide $R(A, B, C)$. As before, we set the cut-off at $\zeta \lg q_t$ for some $\zeta \geq 1$. We are not going to elaborate the details of this strategy and the expected benefits once again in this section. We concentrate on an additional heuristic modification of the equation collecting phase instead.

We recall from Section 3 that we check the smoothness of $R(A, B, C)$ for $-M \leq A \leq B \leq C \leq M$. With this condition, $C$ varies from 0 to $M$. We note that for each value of $C$, we have to execute the entire sieving operation once. For each such sieving operation (that is, for a fixed $C$), the sieving interval for $B$ is (i.e. the admissible values of $B$ are) $-C/2 \leq B \leq \min(C, M - C)$. Correspondingly $A = -(B+C)$ can vary from $\max(-2C, -M)$ to $-C/2$. It's easy to see that in this case total number of triples $(A, B, C)$ for which the smoothness of $R(A, B, C)$ is examined is $\tau = \sum_{C=0}^{M} \Big(1 + \lfloor C/2 \rfloor + \min(C, M - C)\Big) \approx M^2/2$. The number of unknowns, that is, the size of the factor base, on the other hand, is $\nu \approx 2M + t$.

If we remove the restriction $A \geq -M$ and allow $A$ to be as negative as $-\lambda M$ for some $1 < \lambda \leq 2$, then we are benefitted in the following way. As before, we allow $C$ to vary from 0 to $M$ keeping the number of sieving operations fixed. Since $A$ can now assume values smaller than $-M$, the sieving interval increases to $-C/2 \leq B \leq \min(C, \lambda M - C)$. As a result, the total number of triples $(A, B, C)$ becomes $\tau_\lambda = \sum_{C=0}^{M} \Big(1 + \lfloor C/2 \rfloor + \min(C, \lambda M - C)\Big) \approx \frac{M^2}{4}(4\lambda - \lambda^2 - 1)$, whereas the size of the factor base increases to $\nu_\lambda \approx (\lambda + 1)M + t$. (Note that with this notation the value $\lambda = 1$ corresponds to the original algorithm and $\tau = \tau_1$ and $\nu = \nu_1$.) The ratio $\tau_\lambda/\nu_\lambda$ is approximately proportional to the number of smooth integers $R(A, B, C)$ generated by the algorithm divided by the number of unknowns. Therefore, $\lambda$ should be set at a value for which this ratio is maximum. If one treats $t$ and $M$ as constants, then the maximum is attained at $\lambda^* = -U + \sqrt{U^2 + 4U + 1}$, where $U = \dfrac{M + t}{M} = 1 + \dfrac{t}{M}$. As we

increase $U$ from 1 to $\infty$ (or, equivalently the ratio $t/M$ from 0 to $\infty$), the value of $\lambda^*$ increases monotonically from $\sqrt{6}-1 \approx 1.4495$ to 2. In the following table (Table 2), we summarize the variation of $\tau_\lambda/\nu_\lambda$ for some values of $U$. These values of $U$ correspond from left to right to $t \ll M$, $t \approx M/2$, $t \approx M$ and $t \approx 2M$ respectively. The corresponding values of $\lambda^*$ are respectively 1.4495, 1.5414, 1.6056 and 1.6904. It's clear from the table, that for practical ranges of values of $U$, the choice $\lambda = 1.5$ gives performance quite close to the optimal.

**Table 2.** Variation of $\tau_\lambda/\nu_\lambda$ with $\lambda$

| | $\tau_\lambda/\nu_\lambda$ (approx) | | | |
|---|---|---|---|---|
| $\lambda$ | $U=1$ | $U=1.5$ | $U=2$ | $U=3$ |
| 1 | $0.2500\,M$ | $0.2000\,M$ | $0.1667\,M$ | $0.1250\,M$ |
| 1.5 | $0.2750\,M$ | $0.2292\,M$ | $0.1964\,M$ | $0.1527\,M$ |
| 2 | $0.2500\,M$ | $0.2143\,M$ | $0.1875\,M$ | $0.1500\,M$ |
| $\lambda^*$ | $0.2753\,M$ | $0.2293\,M$ | $0.1972\,M$ | $0.1548\,M$ |

We note that this scheme keeps $M$ and the range of variation of $C$ constant and hence does not increase the number of sieving steps and, in particular, the number of modular inverses and square roots. It is, therefore, advisable to apply the trick (with, say, $\lambda = 1.5$) instead of increasing $M$. With that one is expected to get a speed-up of about 10 to 20% and obtain a larger database.

In what follows, we report about the performance of the cubic sieve algorithm for various values of the parameters $\zeta$ and $\lambda$. We also compare the performance of the cubic sieve algorithm with that of the linear sieve algorithm. We work in the prime field $\mathbb{F}_p$ with

$$p = 1320245474656309183513988729373583242842871683$$

as in the last section. For this prime, we have

$$X = \lfloor \sqrt[3]{p} \rfloor + 1 = 1097029305312372, Y = 1, Z = 31165$$

as a solution of (3).

To start with, we fix $\lambda = 1.5$ and examine the variation of the performance of the equation collecting stage with $\zeta$. We did not implement the 'exact' version of this algorithm in which one tries to solve (6) for exponents $h > 1$ of $q$. Table 3 lists the experimental details for the 'approximate' algorithm. As in Table 1, the CPU times do not include the time for filtering out the spurious relations available by the more generous closeness criterion for the approximate algorithm. For the cubic sieve method, the values of $\zeta$ around 1.5 works quite well for our prime $p$.

In Table 4, we fix $\zeta$ at 1.5 and tabulate the variation of the performance of the cubic sieve algorithm for some values of $\lambda$. It's clear from the table that among the cases observed, the largest value of the ratio $\bar{\rho}/\bar{\nu}$ is obtained at $\lambda = 1.5$.

**Table 3.** Performance of the cubic sieve algorithm for various values of $\zeta$

$p = 1320245474656309183513988729373583242842871683$
$t = 10000,\ M = 10000,\ \lambda = 1.5$

| $\zeta$ | No. of Relations ($\bar{\rho}$) | No. of Variables ($\bar{\nu}$) | $\bar{\rho}/\bar{\nu}$ | CPU Time (seconds) |
|---|---|---|---|---|
| 1.0 | 54805 | 35001 | 1.5658 | 43508 |
| 1.5 | 54865 | 35001 | 1.5675 | 43336 |
| 2.0 | 54868 | 35001 | 1.5676 | 43492 |

(The theoretical maximum is attained at $\lambda \approx 1.6$) We also note that changing the value of $\lambda$ incurs variation in the running time by at most 1%. Thus our heuristic allows us to build a larger database at approximately no extra cost.

**Table 4.** Performance of the cubic sieve algorithm for various values of $\lambda$

$p = 1320245474656309183513988729373583242842871683$
$t = 10000,\ M = 10000,\ \zeta = 1.5$

| $\lambda$ | No. of Relations ($\bar{\rho}$) | No. of Variables ($\bar{\nu}$) | $\bar{\rho}/\bar{\nu}$ | CPU Time (seconds) |
|---|---|---|---|---|
| 1.0 | 43434 | 30001 | 1.4478 | 43047 |
| 1.5 | 54865 | 35001 | 1.5675 | 43336 |
| 1.6 | 56147 | 36001 | 1.5596 | 43347 |
| 2.0 | 58234 | 40001 | 1.4558 | 43499 |

### 5.1 Performance Comparison With Linear Sieve

The speed-up obtained by the cubic sieve method over the linear sieve method is about 2.5 for the field of size around 150 bits. For larger fields, this speed-up is expected to be more. It is, therefore, evident that the cubic sieve algorithm, at least for the case $\alpha = 1/3$, runs faster than the linear sieve counterpart for the practical range of sizes of prime fields.

## 6 Conclusion

In this paper, we have described various practical aspects for efficient implementation of the linear and the cubic sieve algorithms for the computation of discrete logarithms over finite fields. We have also compared the performances of these two algorithms and established the superiority of the latter method over the former for the cases when $p$ is close to a whole cube. It, however, remains unsettled whether the cubic sieve algorithm performs equally well for a general prime $p$. More importantly, the applicability of the cubic sieve algorithm banks

on the availability of a 'favorable' solution of a certain Diophantine equation. Finding an algorithm for computing the solution of this Diophantine equation or even for certifying if a solution exists, continues to remain an open problem and stands in the way of the general acceptance of the cubic sieve algorithm. Last but not the least, we need performance comparison of the cubic sieve method with the number field sieve method.

## References


1. Bressoud, D.M.: Factorization and Primality Testing, UTM, Springer-Verlag, 1989.
2. Cohen, H.: A course in computational algebraic number theory, GTM **138**, Springer-Verlag, 1993.
3. Coppersmith, D., Odlyzko, A.M., Schroeppel, R.: Discrete logarithms in $GF(p)$, Algorithmica **1** (1986), 1–15.
4. Das, A., Veni Madhavan, C.E.: Galois field library: Reference manual, Technical report No. IISc-CSA-98-05, Department of Computer Science and Automation, Indian Institute of Science, Feb 1998.
5. Gerver, J.: Factoring large numbers with a quadratic sieve, Math. Comp. **41** (1983), 287–294.
6. Gordon, D.M.: Discrete logarithms in $GF(p)$ using the number field sieve, SIAM Journal of Discrete Mathematics **6** (1993), 124–138.
7. LaMacchia, B.A., Odlyzko, A.M.: Computation of discrete logarithms in prime fields, Designs, Codes, and Cryptography **1** (1991), 46–62.
8. LaMacchia, B.A., Odlyzko, A.M.: Solving large sparse linear systems over finite fields, Advances in Cryptology – CRYPTO'90, A. J. Menezes and S. A. Vanstone (eds.), LNCS **537** (1991), Springer-Verlag, 109–133.
9. McCurley, K.S.: The discrete logarithm problem, Cryptology and Computational Number Theory, Proc. Symp. in Appl. Math. **42** (1990), 49–74.
10. Menezes, A.J., ed.: 'Applications of finite fields', Kluwer Academic Publishers, 1993.
11. Odlyzko, A.M.: Discrete logarithms and their cryptographic significance, Advances in Cryptology: Proceedings of Eurocrypt'84, LNCS **209** (1985), Springer-Verlag, 224–314.
12. Schirokauer, O., Weber, D., Denny, T.: Discrete logarithms: the effectiveness of the index calculus method, Proc. ANTS II, LNCS **1122** (1996), Springer-Verlag, 337–361.
13. Silverman, R.D.: The multiple polynomial quadratic sieve, Math. Comp. **48** (1987), 329–339.
14. Weber, D.: Computing discrete logarithms with the general number field sieve, Proc. ANTS II, LNCS **1122** (1996), Springer-Verlag, 99–114.
15. Weber, D., Denny, T.: The solution of McCurley's discrete log challenge, Crypto'98, LNCS **1462** (1998), Springer-Verlag, 458–471.

# External Memory Algorithms for Outerplanar Graphs

Anil Maheshwari* and Norbert Zeh

School of Computer Science, Carleton University, Ottawa, Canada
{maheshwa,nzeh}@scs.carleton.ca

**Abstract.** We present external memory algorithms for outerplanarity testing, embedding outerplanar graphs, breadth-first search (BFS) and depth-first search (DFS) in outerplanar graphs, and finding a $\frac{2}{3}$-separator of size 2 for a given outerplanar graph. Our algorithms take $O(sort(N))$ I/Os and can easily be improved to take $O(perm(N))$ I/Os, as all these problems have linear time solutions in internal memory. For BFS, DFS, and outerplanar embedding we show matching lower bounds.

## 1 Introduction

*Motivation.* Outerplanar graphs are a well-studied class of graphs (e.g., see [3,4,6]). This class is restricted enough to admit more efficient algorithms than those for general graphs and general enough to have practical applications, e.g. [3].

Our motivation to study outerplanar graphs in external memory is threefold. Firstly, efficient algorithms for triangulating and separating planar graphs were presented in [5,12]. The major drawback of their separator algorithm is that it requires an embedding and a BFS-tree of the given graph as part of the input. Embedding planar graphs and BFS in general graphs are hard problems in external memory.[1] Our goal is to show that these problems are considerably easier for outerplanar graphs. Secondly, outerplanar graphs can be seen as combinatorial representations of triangulated simple polygons and their subgraphs. Thirdly, every outerplanar graph is a planar graph. Thus, any lower bound that we can show for outerplanar graphs also holds for planar graphs.

*Model of computation.* When the data set to be handled becomes too large to fit into the main memory of the computer, the transfer of data between fast internal memory and slow external memory (disks) becomes a significant bottleneck. Existing internal memory algorithms usually access their data in a random fashion, thereby causing significantly more I/O operations than necessary. Our goal in this paper is to minimize the number of I/O operations performed. Several computational models for estimating the I/O-efficiency of algorithms have been

* Research supported by NSERC and NCE GEOIDE.

[1] No external memory algorithm for embedding planar graphs is known. For BFS in general graphs, the best known algorithm takes $O(|V| + |E|/|V|sort(|V|))$ I/Os [9].

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 307–316, 1999.
© Springer-Verlag Berlin Heidelberg 1999

developed. We adopt the *parallel disk model* PDM [11] as our model of computation for this paper due to its simplicity, and the fact that we consider only a single processor. In the PDM, an *external memory*, consisting of $D$ disks, is attached to a machine with memory size $M$ data items. Each of the disks is divided into blocks of $B$ consecutive data items. Up to $D$ blocks, at most one per disk, can be transferred between internal and external memory in a single I/O operation. The complexity of an algorithm is the number of I/O operations it performs.

*Previous work.* For planar graphs, a linear-time algorithm for finding a $\frac{2}{3}$-separator of size $O\left(\sqrt{N}\right)$ was presented in [7]. It is well known that every outerplanar graph has a $\frac{2}{3}$-separator of size 2 and that such a separator can be computed in linear time. Outerplanarity testing [8] and embedding outerplanar graphs take linear time. There are simple linear time algorithms for BFS and DFS in general graphs (see [2]). Refer to [4,6] for a good exposition of outerplanar graphs.

In the PDM, sorting, permuting, and scanning an array of size $N$ take $sort(N) = \Theta\left(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$, $perm(N) = \Theta\left(\min\left\{N, sort(N)\right\}\right)$, and $scan(N) = \Theta\left(\frac{N}{DB}\right)$ I/Os [10,11]. For a comprehensive survey of external memory algorithms, refer to [10]. The best known BFS-algorithm for general graphs takes $O\left(\frac{|E|}{|V|}sort(|V|) + |V|\right)$ I/Os [9]. In [1], $O(sort(N))$ algorithms for computing an open ear decomposition and the connected and biconnected components of a given graph $G = (V, E)$ with $|E| = O(|V|)$ were presented. They also develop a technique, called *time-forward processing*, that can be used to evaluate a directed acyclic graph of size $N$, viewed as a (logical) circuit, in $O(sort(N))$ I/Os. They apply this technique to develop an $O(sort(N))$ algorithm for list-ranking. An external memory separator algorithm for planar graphs has been presented in [5,12]; it takes $O(sort(N))$ I/Os provided that a BFS-tree and an embedding of the graph is given. We do not know of any other results for computing separators in external memory efficiently. Also, no efficient external memory algorithms for embedding planar or outerplanar graphs in the plane are known.[2]

*Our results.* In this paper, we show the following theorem.

**Theorem 1.** *It takes $O(perm(N))$ I/Os to decide whether a given graph $G$ with $N$ vertices is outerplanar. If $G$ is outerplanar, breadth-first search, depth-first search, and computing an outerplanar embedding of $G$ take $\Theta(perm(N))$ I/Os. Computing a $\frac{2}{3}$-separator of size 2 for $G$ takes $O(perm(N))$ I/Os.*

*Preliminaries.* A *graph* $G = (V, E)$ is a pair of sets, $V$ and $E$. $V$ is the *vertex set* of $G$. $E$ is the *edge set* of $G$ and consists of unordered pairs $\{v, w\}$, where

[2] One can use the PRAM simulation technique of [1] together with known PRAM results. Unfortunately, the PRAM simulation introduces $O(sort(N))$ I/Os for every PRAM step, and so the resulting I/O complexity is not attractive for our purposes.

$v, w \in V$. In this paper, $N = |V|$. A *path* in $G$ is a sequence, $P = \langle v_0, \ldots, v_k \rangle$, of vertices such that $\{v_{i-1}, v_i\} \in E$, for $1 \leq i \leq k$. A graph is *connected* if there is a path between any pair of vertices in $G$. A graph is *biconnected* if for every vertex $v \in V$, $G - v$ is connected. A *tree* with $N$ vertices is a connected graph with $N-1$ edges. A *subgraph* of $G$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. The *connected components* of $G$ are the maximal connected subgraphs of $G$. The *biconnected components* (bicomps) of $G$ are the maximal biconnected subgraphs of $G$. A graph is *planar* if it can be drawn in the plane so that no two edges intersect except at their endpoints. This defines an order of the edges incident to every vertex $v$ of $G$ counterclockwise around $v$. We call $G$ *embedded* if we are given these orders for all vertices of $G$. $\mathbb{R}^2 \setminus (V \cup E)$ is a set of connected regions. Call these regions the *faces* of $G$. Denote the set of faces of $G$ by $F$. A graph is *outerplanar* if it can be drawn in the plane so that there is a face that has all vertices of $G$ on its boundary. For every outerplanar graph, $|E| \leq 2|V| - 3$. The *dual* of an embedded planar graph $G = (V, E)$ with face set $F$ is a graph $G^* = (V^*, E^*)$, where $V^* = F$ and $E^*$ contains an edge between two vertices in $V^*$ if the two corresponding faces in $G$ share an edge.

An *ear decomposition* of a biconnected graph $G$ is a decomposition of $G$ into edge-disjoint paths $P_0, \ldots, P_k$, $P_i = \langle v_i, \ldots, w_i \rangle$, such that $G = \bigcup_{j=0}^{k} P_j$, $P_0 = \langle v_0, w_0 \rangle$, and for every $P_i$, $i \geq 1$, $P_i \cap G_{i-1} = \{v_i, w_i\}$, where $G_{i-1} = \bigcup_{j=0}^{i-1} P_j$. The paths $P_j$ are called *ears*. An *open ear decomposition* is an ear decomposition such that for every ear $P_i$, $v_i \neq w_i$.

Let $w : V \to \mathbb{R}^+$ be an assignment of weights to the vertices of $G$ such that $\sum_{v \in V} w(v) \leq 1$. The weight of a subgraph of $G$ is the sum of the vertex weights in the subgraph. A $\frac{2}{3}$-*separator* of $G$ is a set $S$ such that none of the connected components of $G - S$ has weight exceeding $\frac{2}{3}$.

We will use the following characterization of outerplanar graphs [4].

**Theorem 2.** *A graph $G$ is outerplanar if and only if it does not contain a subgraph that is an edge expansion of $K_{2,3}$ or $K_4$.*

In our algorithms we represent a graph $G$ as the two sets $V$ and $E$. The embedding of a graph is represented as labels $n_v(e)$ and $n_w(e)$ for every edge $e = \{v, w\}$, where $n_v(e)$ and $n_w(e)$ are the positions of $e$ in the counterclockwise orders of the edges around $v$ and $w$, respectively.

## 2 Embedding and Outerplanarity Testing

We show how to compute a combinatorial embedding (i.e., edge labels $n_v(e)$ and $n_w(e)$) of a given outerplanar graph $G$. Our algorithm for outerplanarity testing is based on the embedding algorithm. We restrict ourselves to biconnected outerplanar graphs. If the given graph is not biconnected, we compute the biconnected components in $O(sort(N))$ I/Os [1]. Then we compute embeddings of the biconnected components and join them at the cutpoints of the graph.

Our algorithm for embedding biconnected outerplanar graphs $G$ consists of two steps. In Step 1 we compute the cycle $C$ in $G$ that represents $G$'s outer

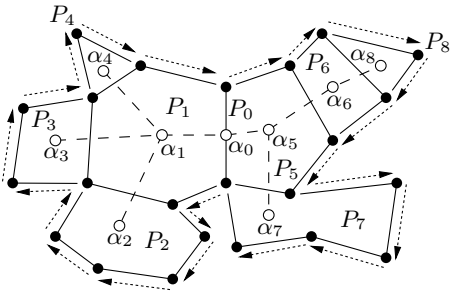



**Fig. 1**

boundary in the embedding. In Step 2 we embed the remaining edges, which are diagonals of $C$. Step 1 relies on Observation 1. Let $\mathcal{E}_G = \langle P_0, \ldots, P_k \rangle$ be the given open ear decomposition. We call an ear $P_i$, $i \geq 1$, *trivial* if it consists of a single edge. Otherwise we call it *non-trivial*. Ear $P_0$ is non-trivial by definition. Let $G_i = \bigcup_{j=0}^{i} P_j$. Given an embedding of $G_i$, we call two vertices, $v$ and $w$, of $G_i$ *consecutive* if there is an edge $\{v, w\}$ in $G_i$ that is on the outer boundary of $G_i$.

**Observation 1.** *Given a decomposition of a biconnected outerplanar graph $G$ into open ears $P_0, \ldots, P_k$ and an embedding of $G$. Then either $P_i$ is trivial or the endpoints of $P_i$ are consecutive in $G_{i-1}$, for $1 \leq i \leq k$.*

Observation 1 implies that, except if a non-trivial ear $P_i$ is attached to the endpoints of $P_0$, there is exactly one non-trivial ear $P_j$, $j < i$, that contains both endpoints of $P_i$. We represent this relationship between non-trivial ears in the *ear tree* $T_E$ of $G$ (see Fig. 1). This tree contains a node $\alpha_k$ for every non-trivial ear $P_k$. Node $\alpha_j$ is the parent of node $\alpha_i$ ($\alpha_j = p(\alpha_i)$) if $P_j$ contains both endpoints of $P_i$. If $P_i$'s endpoints are also $P_0$'s endpoints, $\alpha_i$ is the child of $\alpha_0$. The vertices in $P_i$ must appear in the same order along the outer boundary of $G$ as they appear in $P_i$. Using these observations, we construct the order of the vertices along $C$ using a depth-first traversal of $T_E$ as follows:

Start the traversal of $T_E$ at node $\alpha_0$. At a node $\alpha_i$, we traverse the ear $P_i$ and append $P_i$'s vertices to $C$. When we reach an endpoint of an ear $P_j$ with $\alpha_i = p(\alpha_j)$, we recursively traverse the subtree rooted at $\alpha_j$. When we are done with the traversal of that subtree, we continue traversing $P_i$.

To construct the final embedding, we use the following observation: Let $e_1, \ldots, e_d$ be the edges incident to a vertex $v$, sorted clockwise around $v$. Let $u_i$ be the other endpoint of edge $e_i$, for $1 \leq i \leq d$. Then the vertices $u_1, \ldots, u_d$ appear in clockwise order along the outer boundary of $G$ (see Fig. 1).

**Lemma 1.** *An outerplanar embedding of a given outerplanar graph $G$ with $N$ vertices can be computed in $O(sort(N))$ I/Os.*

*Proof sketch. Open ear decomposition:* The open ear-decomposition of $G$ can be computed in $O(sort(N))$ I/Os [1]. We scan the list, $\mathcal{E}_G$, of ears to construct the list, $\mathcal{E}'_G$, of non-trivial ears.
*Ear tree construction:* Let $v$ be a vertex that is interior to ear $P_i$. (Note that every vertex, except for the two endpoints of $P_0$, is interior to exactly one ear.) Then we define $v$'s ear number as $\epsilon(v) = i$. For the two endpoints, $v_0$ and $w_0$, of $P_0$ we define $\epsilon(v_0) = \epsilon(w_0) = 0$. Let $\alpha_i = p(\alpha_j)$ in $T_E$. It can be shown that $i = \max\{\epsilon(v_j), \epsilon(w_j)\}$. We scan the list of non-trivial ears to compute all ear numbers. The ear tree can then be constructed in $O(sort(N))$ I/Os.
*Construction of $C$:* Consider the ears corresponding to the nodes stored in a subtree $T_E(\alpha_j)$ of $T_E$ rooted at a node $\alpha_j$. The internal vertices of these ears are exactly the vertices that appear between the two endpoints, $v_j$ and $w_j$, of $P_j$ on the outer boundary of $G$, i.e., in $C$. The number of these vertices can be computed as follows: We assign the number of internal vertices of ear $P_j$ as a weight $w(\alpha_j)$ to every node $\alpha_j$ and use time-forward processing to compute the subtree weights $w(T_E(\alpha_j))$ of all subtrees $T_E(\alpha_j)$.

Now we sort the ears $P_i$ in $\mathcal{E}'_G$ by their indices $i$. Scanning this sorted list of ears corresponds to processing $T_E$ from the root to the leaves. We use time-forward processing to send small pieces of additional information down the tree. We start at node $\alpha_0$ and assign $n(v_0) = 0$ and $n(w_0) = w(T_E(\alpha_1)) + 1$, where $n(v)$ is $v$'s position in a clockwise traversal of $C$. For every subsequent ear, $P_j$, we maintain the invariant that when we start the scan of $P_j$, we have already computed $n(v_j)$ and $n(w_j)$. Then we scan along $P_j$ and number the vertices of $P_j$ in their order of appearance. Let $x$ be the current vertex in this scan and $y$ be the previous vertex. If there is no ear attached to $x$ and $y$, $n(x) = n(y) + 1$. Otherwise, let ear $P_i$ be attached to $x$ and $y$ (i.e., $\{x, y\} = \{v_i, w_i\}$). Then $n(x) = n(y) + w(T_E(\alpha_i)) + 1$. We send $n(v_i)$ and $n(w_i)$ to $\alpha_i$ so that this information is available when we process $P_i$. Note that the current ear $P_j$ might be stored in reverse order (i.e., $n(w_j) < n(v_j)$). This can easily be handled using a stack to reverse $P_j$ before scanning.
*Computing the final embedding:* First, we relabel the vertices of $G$ in their order around $C$. Then we replace every edge $e = \{v, w\}$ by two directed edges $(v, w)$ and $(w, v)$. We sort the list of these edges lexicographically and scan it to compute the desired labels $n_v(e)$ and $n_w(e)$. □

Note that ear $P_0$ requires some special treatment because it is the only ear that can have two non-trivial ears attached on two different sides. However, the details are fairly straightforward and therefore omitted. The above algorithm can be augmented to do outerplanarity testing in $O(sort(N))$ I/Os. Due to space constraints, we refer the reader to the full version of the paper.

## 3 Breadth-First and Depth-First Search

We can restrict ourselves to BFS and DFS in biconnected outerplanar graphs. If the graph is not biconnected, we compute its connected and biconnected components in $O(sort(N))$ I/Os [1]. Then we represent every connected component

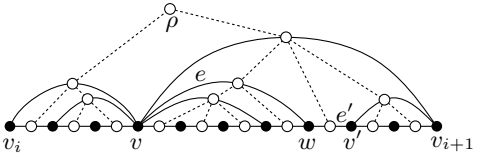



**Fig. 2**

by a rooted tree describing the relationship between its bicomps and cutpoints. We process each such tree from the root to the leaves, applying Lemmas 3 and 2 to the bicomps of the graph, in order to compute a BFS resp. DFS-tree for every connected component.

Given a biconnected outerplanar graph $G$, we first embed it. The list $C$ computed by our embedding algorithm contains the vertices of $G$ sorted counterclockwise along the outer boundary of $G$. Given a source vertex $r$, a path along the outer boundary of $G$, starting at $r$, is a DFS-tree of $G$. Thus, we can compute a DFS-tree of $G$ by scanning $C$.

**Lemma 2.** *Given a biconnected outerplanar graph $G$ with $N$ vertices, depth-first search in $G$ takes $O(sort(N))$ I/Os.*

The construction of a BFS-tree is based on the following observation: Let the vertices of $G$ be numbered counterclockwise around the outer boundary of $G$ and such that the source, $r$, has number 1. Let $v_1 < \cdots < v_k$ be the neighbours of $r$. The removal of these vertices partitions $G$ into subgraphs $G_i$, $1 \leq i < k$, induced by vertices $v_i + 1, \ldots, v_{i+1} - 1$. Indeed, if there was an edge $\{v, w\}$ between two such graphs $G_i$ and $G_j$, $i < j$, $G$ would contain a $K_4$ consisting of the paths between $r$, $v$, $w$, $v_i$, $v_j$, and $v_{j+1}$. We consider graphs $\tilde{G}_i$ that are induced by vertices $v_i, \ldots, v_{i+1}$. Let $e$ and $e'$ be two edges in such a graph $\tilde{G}_i$ such that the left endpoint of $e$ is to the left of the left endpoint of $e'$. Then either $e$ is completely to the left of $e'$ or $e$ spans $e'$.

The vertices $v_i$ and $v_{i+1}$ are at distance 1 from the root $r$, and the shortest path from any vertex in $\tilde{G}_i$ to $r$ must contain $v_i$ or $v_{i+1}$. Thus, we can do BFS in $\tilde{G}_i$ by finding the shortest path from every vertex in $\tilde{G}_i$ to either $v_i$ or $v_{i+1}$, whichever is shorter. We build an *edge tree* $T_e$ for $\tilde{G}_i$ (see Fig. 2). $T_e$ contains a node $v(e)$ for every edge $e$ in $\tilde{G}_i$. Node $v(e)$ is the parent of another node $v(e')$ if edge $e$ spans edge $e'$ and there is no edge $e''$ that spans $e'$ and is spanned by $e$. $T_e$ has an additional root node $\rho$, which is the parent of all nodes $v(e)$, where edge $e$ is not spanned by any other edge. The *level* of an edge $e$ in $\tilde{G}_i$ is the distance of node $v(e)$ from the root $\rho$ of $T_e$.

We call an edge $e = \{v, w\}$ incident to vertex $w$ a *left* (resp. *right*) edge if $v < w$ (resp. $v > w$). The minimal left (resp. right) edge for $w$ has the minimal level in $T_e$ among all left (resp. right) edges incident to $w$. The shortest path from $w$ to $v_i$ ($v_{i+1}$) must contain either the minimal left edge, $e = \{v, w\}$, or the minimal right edge, $e' = \{v', w\}$, incident to $w$ (see Fig. 2). Indeed, the shortest path must contain $v$ or $v'$, and $e$ (resp. $e'$) is the shortest path from

$w$ to $v$ (resp. $v'$). Thus, we can process $T_e$ level by level from the root towards the leaves maintaining the following invariant: When we visit a node $v(e)$ in $T_e$, where $e = \{v, w\}$, we know the distances $d(r, v)$ and $d(r, w)$ from $v$ and $w$ to the source vertex $r$. Assume that $v(e)$'s children are sorted from left to right. We scan the list of the corresponding edges from left to right to compute the distances of their endpoints to $v$. In a right-to-left scan we compute the distances to $w$. Then it takes another scan to determine for every edge endpoint $x$, the distance $d(r, x) = \min\{d(r, v) + d(v, x), d(r, w) + d(w, x)\}$ and to set the parent pointers in the BFS-tree properly. This computation ensures the invariant for $v(e)$'s children.

**Lemma 3.** *Given a biconnected outerplanar graph $G$ with $N$ vertices, breadth-first search in $G$ takes $O(sort(N))$ I/Os.*

*Proof sketch. Constructing $\tilde{G}_1, \ldots, \tilde{G}_{k-1}$:* We use our embedding algorithm to embed $G$ and compute the order of the vertices of $G$ along the outer boundary of $G$. Given this ordering, we use sorting and scanning to split $G$ into subgraphs $\tilde{G}_i$.
*Constructing $T_e$:* We sort the list, $V_i$, of vertices of $\tilde{G}_i$ from left to right and store with every vertex $w$, the list, $E(w)$, of edges incident to $w$, sorted clockwise around $w$. Then we scan $V_i$ and apply a simple stack algorithm to construct $T_e$: We initialize the stack by pushing $\rho$ on the stack. When visiting vertex $w$, we first pop the nodes $v(e)$ for all left edges $e = \{v, w\}$ in $E(w)$ from the stack and make the next node on the stack the parent of the currently popped node. Then we push the nodes $v(e)$ for all right edges in $E(w)$ on the stack, in their order of appearance.
*Breadth-first search in $\tilde{G}_i$:* We sort the edges of $\tilde{G}_i$ by increasing level and from left to right and use time-forward processing to send the computed distances downward in $T_e$. Scanning the list of children of the currently visited node $v(e)$ takes $O(scan(N))$ I/Os for all nodes of $T_e$ because the edges of every level are sorted from left to right, and the right-to-left scans can be implemented using a stack. □

## 4 Separating Outerplanar Graphs

We assume that $G$ is connected and no vertex in $G$ has weight greater than $\frac{1}{3}$. If there is a vertex $v$ with weight $w(v) > \frac{1}{3}$, $S = \{v\}$ is trivially a $\frac{2}{3}$-separator of $G$. If $G$ is disconnected, we compute $G$'s connected components in $O(sort(N))$ I/Os [1]. If there is no component of weight greater than $\frac{2}{3}$, $S = \emptyset$ is a $\frac{2}{3}$-separator of $G$. Otherwise, we compute a separator of the connected component of weight greater than $\frac{2}{3}$.

Our strategy for finding a size-2 separator of $G$ is as follows: First we embed $G$ and make $G$ biconnected by adding appropriate edges to the outer face of $G$. Then we triangulate the interior faces of the resulting graph and compute the dual tree $T^*$ corresponding to the interior faces of the triangulation $G_\triangle$ of $G$. Every edge in $T^*$ corresponds to a diagonal of the triangulation, whose two

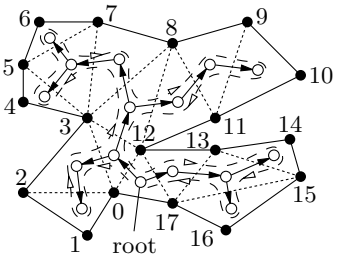



**Fig. 3**

endpoints are a size-2 separator of $G_\triangle$ and thus of $G$. We assign appropriate weights to the vertices of $T^*$ that allow us to find a tree edge that corresponds to a $\frac{2}{3}$-separator of $G_\triangle$.

We use the following observations: (1) $G$ is biconnected if and only if its outer face is simple. (A face is simple, if every vertex appears at most once in a counterclockwise traversal of its boundary.) (2) If $G$ is a cycle, we can choose one of the two faces of $G$ as the outer face. Otherwise, the outer face of $G$ is the only face that has all vertices of $G$ on its boundary.

The triangulation algorithm for planar graphs in [5] first makes all faces of the given graph simple and then triangulates the resulting simple faces. Using the two observations just made, this triangulation algorithm can easily be modified to triangulate all faces of $G$ except the outer face, which is only made simple.

**Lemma 4.** *A size-2 $\frac{2}{3}$-separator of a given outerplanar graph $G$ with $N$ vertices can be computed in $O(sort(N))$ I/Os.*

*Proof sketch.* We have to show how to construct the dual tree $T^*$ corresponding to the interior faces of $G_\triangle$ and how to find an edge of $T^*$ corresponding to a $\frac{2}{3}$-separator of $G_\triangle$.

*Constructing the dual tree:* We first number the vertices 0 through $N-1$ clockwise around $G_\triangle$. With every vertex $v$ we store its adjacency list $A(v)$ sorted counterclockwise around $v$, where $(v-1) \bmod N$ is the first vertex in $A(v)$. Denote the concatenation of $A(0), \ldots, A(N-1)$ by $A$. We construct $T^*$ recursively (see Fig. 3). We start at edge $\{a, b\} = \{0, N-1\}$ and consider triangle $\triangle_1 = (a, b, x)$.

We make the vertex $v(\triangle_1)$ corresponding to $\triangle_1$ the root of $T^*$. Let $\triangle_2 = (a, x, y)$ and $\triangle_3 = (x, b, z)$ be the two triangles adjacent to $\triangle_1$. Then the corresponding vertices $v(\triangle_2)$ and $v(\triangle_3)$ are the children of $v(\triangle_1)$. We recursively construct the subtrees of $T^*$ rooted at $v(\triangle_2)$ and $v(\triangle_3)$, calling the tree construction procedure with parameters $\{a_1, b_1\} = \{a, x\}$ and $\{a_2, b_2\} = \{x, b\}$, respectively. Using this strategy we basically perform a depth-first traversal of $T^*$. The corresponding Euler tour (as represented by the dashed line in Fig. 3) crosses the edges of $G_\triangle$ in their order of appearance in $A$. Thus, $T^*$ can be constructed in a single scan over $A$ and using $O(N)$ stack operations.

*Finding the separator:* At every recursive call we assign weights $w(v(\triangle_1)) = w(x)$ and $w_p(v(\triangle_1)) = w(a) + w(b)$ to the newly created vertex $v(\triangle_1)$. Let $T^*(v)$ be the subtree of $T^*$ rooted at a vertex $v$. During the construction of $T^*$ we can compute for every vertex $v$, the weight $w(T^*(v)) = \sum_{u \in T^*(v)} w(u)$ of the subtree $T^*(v)$. The removal of the edge connecting $v(\triangle_1)$ to its parent in $T^*$ corresponds to removing vertices $a$ and $b$ from $G_\triangle$. This partitions $G_\triangle$ into two subgraphs of weights $w(T^*(v(\triangle_1)))$ and $w(G_\triangle) - w(T^*(v(\triangle_1))) - w_p(v(\triangle_1))$. Thus, once the weights $w(T^*(v))$ and $w_p(v)$ have been computed for every vertex $v$ of $T^*$, it takes a single scan over the vertex list of $T^*$ to compute a size-2 $\frac{2}{3}$-separator of $G_\triangle$ and thus of $G$. □

## 5 Lower Bounds

In this section, we prove matching lower bounds for all results in this paper, except for computing separators. We show these lower bounds by reducing list-ranking, which has an $\Omega(perm(N))$ lower bound [1], to DFS, BFS, and embedding of biconnected outerplanar graphs. The list-ranking problem is defined as follows: *Given a singly linked list $L$ and a pointer to the head of $L$, compute for every node of $L$ its distance to the tail.*
Note that list-ranking reduces trivially to DFS and BFS in general outerplanar graphs, as we can consider the list itself as an outerplanar graph and choose the tail of the list as the source of the search.

**Lemma 5.** *List-ranking can be reduced to computing a combinatorial embedding of a biconnected outerplanar graph in $O(scan(N))$ I/Os.*

*Proof sketch.* Given the list $L$, we can compute the tail $t$ of $L$ ($t$ is the node with no successor) and node $t'$ with $succ(t') = t$ in two scans over $L$. We consider $L$ as a graph with an edge between every vertex and its successor. In another scan, we add edges $\{v, t\}$ to $L$, for $v \notin \{t, t'\}$. This gives us a graph $G_1$ as in Fig. 4(a). It can be shown that the outerplanar embedding of $G_1$ is unique except for flipping the whole graph. Thus, the rank of $v$ in $L$ is the position of edge $\{v, t\}$ in clockwise or counterclockwise order around $t$. □

**Lemma 6.** *List-ranking can be reduced to breadth-first search and depth-first search, respectively, in biconnected outerplanar graphs in $O(scan(N))$ I/Os.*

*Proof sketch.* We only show the reduction for BFS. The reduction to DFS is even simpler. As in the proof of the previous lemma, we identify the tail $t$ of $L$ in a single scan. Then we add an edge $\{h, t\}$ to $L$. This produces a cycle $G_2$. We perform two breadth-first searches (see Fig. 4(b)), one with source $t$ (labels outside) and one with source $h$ (labels inside). It is easy to see that the distance $d(v)$ of every node to the tail of the list can now be computed as $d(v) = N - 1 - d(h, v)$ if $d(h, v) < d(t, v)$, and $d(v) = d(t, v)$ otherwise. □

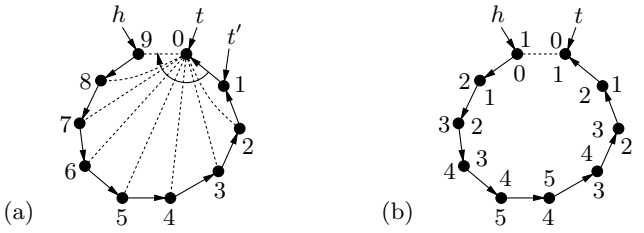



**Fig. 4**

*Proof sketch (Theorem 1).* The theorem follows from the lemmas in this paper, if we can reduce the upper bounds from $O(sort(N))$ to $O(perm(N))$ I/Os. Let $\mathcal{P}$ be the problem at hand, $\mathcal{A}$ be an $O(N)$ time internal memory algorithm that solves $\mathcal{P}$, and $\mathcal{A}'$ be an $O(sort(N))$ external memory algorithm that solves $\mathcal{P}$. (For the problems studied in this paper, linear time solutions in internal memory are known, and we have provided the $O(sort(N))$ algorithms.) We run algorithms $\mathcal{A}$ and $\mathcal{A}'$ in parallel, switching from one algorithm to the other at every I/O operation. The computation stops as soon as one of the algorithms terminates. At this point algorithms $\mathcal{A}$ and $\mathcal{A}'$ have performed at most $\min\{O(N), O(sort(N))\} = O(perm(N))$ I/Os. □

## References

1. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, J. S. Vitter. External-memory graph algorithms. *Proc. 6th SODA*, Jan. 1995.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.
3. G. N. Frederickson. Searching among intervals in compact routing tables. *Algorithmica*, 15:448–466, 1996.
4. F. Harary. *Graph Theory.* Addison-Wesley, 1969.
5. D. Hutchinson, A. Maheshwari, N. Zeh. An external memory data structure for shortest path queries. *Proc. COCOON'99*, LNCS 1627, pp. 51–50, July 1999.
6. J. van Leeuwen. *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity.* MIT Press, 1990.
7. R. J. Lipton, R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. on Applied Mathematics*, 36(2):177–189, 1979.
8. S. L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Inf. Proc. Letters*, 9(5):229–232, Dec. 1979.
9. K. Munagala, A. Ranade. I/O-complexity of graph algorithms. *Proc. 10th SODA*, Jan. 1999.
10. J. S. Vitter. External memory algorithms. *Proc. 17th ACM Symp. on Principles of Database Systems*, June 1998.
11. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.
12. N. Zeh. *An External-Memory Data Structure for Shortest Path Queries.* Diplomarbeit, Fak. f. Math. und Inf. Friedrich-Schiller-Univ. Jena, Nov. 1998.

# A New Approximation Algorithm for the Capacitated Vehicle Routing Problem on a Tree*

Tetsuo Asano[1], Naoki Katoh[2], and Kazuhiro Kawashima[1]

[1] School of Information Science, JAIST,
Asahidai, Tatsunokucho, 923-1292, Japan,
{t-asano, kawasima}@jaist.ac.jp
[2] Department of Architecture and Architectural Systems, Kyoto University,
Sakyo-ku, Kyoto, 606-8501 Japan,
naoki@archi.kyoto-u.ac.jp

**Abstract.** This paper presents a new approximation algorithm for a vehicle routing problem on a tree-shaped network with a single depot. Customers are located on vertices of the tree. Demands of customers are served by a fleet of identical vehicles with limited capacity. It is assumed that the demand of a customer is splittable, i.e., it can be served by more than one vehicle. The problem we are concerned with in this paper asks to find a set of tours of the vehicles with minimum total lengths. Each tour begins at the depot, visits a subset of the customers and returns to the depot. We propose a 1.35078-approximation algorithm for the problem (exactly, $(\sqrt{41}-1)/4$), which is an improvement over the existing 1.5-approximation.

## 1 Introduction

In this paper we consider a capacitated vehicle routing problem on a tree-shaped network with a single depot. Let $T = (V, E)$ be a tree, where $V$ is a set of $n$ vertices and $E$ is a set of edges, and $r \in V$ be a designated vertex called *depot*. Nonnegative weight $w(e)$ is associated with each edge $e \in E$, which represents the length of $e$. Customers are located at vertices of the tree, and a customer at $v \in V$ has a positive demand $D(v)$. Thus, when there is no customer at $v$, $D(v) = 0$ is assumed. Demands of customers are served by a set of identical vehicles with limited capacity. We assume throughout this paper that the capacity of every vehicle is equal to one, and that the demand of a customer is splittable, i.e., it can be served by more than one vehicle. Each vehicle starts at the depot, visits a subset of customers to (partially) serve their demands and returns to the depot without violating the capacity constraint. The problem we deal with in this paper asks to find a set of tours of vehicles with minimum total lengths to satisfy all the demands of customers. We call this problem TREE-CVRP.

* Research of this paper is partly supported by the Grant-in-Aid for Scientific Research on Priority Areas (B) by the Ministry of Education, Science, Sports and Culture of Japan.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 317–326, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Vehicle routing problems have long been studied by many researchers (see [3,5] for a survey), and are found in various applications such as scheduling of truck routes to deliver goods from a warehouse to retailers, material handling systems and computer communication networks. Recently, AGVs (automated guided vehicle) and material handling robots are often used in manufacturing systems, but also in offices and hospitals, in order to reduce the material handling efforts. The tree-shaped network can be typically found in buildings with simple structures of corridors and in simple production lines of factories.

Vehicle scheduling problems on tree-shaped networks have recently been studied by several authors [1,4,7,8,9]. Most of them dealt with a single-vehicle scheduling that seeks to find an optimal tour under certain constraints.

However, TREE-CVRP has not been studied in the literature until very recently. Last year Hamaguchi and Katoh [6] proved its NP-hardness and proposed a 1.5-approximation algorithm ([9] considered the variant of TREE-CVRP where demand of each customer is not splittable and gave 2-approximation algorithm.)

In this paper, we shall present an improved 1.35078-approximation algorithm for TREE-CVRP by exploiting the tree structure of the network. This is an improvement of the existing 1.5-approximation algorithm by Hamaguchi and Katoh [6]. A basic idea behind the improvement is the use of reforming operations preserving the lower bound on the cost, which simplifies the analysis.

## 2 Preliminaries

For vertices $u, v \in V$, let $path(u, v)$ be the unique path between $u$ and $v$. The length of $path(u, v)$ is denoted by $w(path(u, v))$. We often view $T$ as a directed tree rooted at $r$. For a vertex $v \in V - \{r\}$, let $parent(v)$ denote the parent of $v$. We assume throughout this paper that when we write an edge $e = (u, v)$, $u$ is a parent of $v$. For any $v \in V$, let $T_v$ denote the subtree rooted at $v$, and $w(T_v)$ and $D(T_v)$ denote the sum of weights of edges in $T_v$, and the sum of demands of customers in $T_v$, respectively. Since customers are located on vertices, customers are often identified with vertices.

For an edge $e = (u, v)$, let

$$LB(e) = 2w(e) \cdot \lceil D(T_v) \rceil. \tag{1}$$

$LB(e)$ represents a lower bound of the cost required for traversing edge $e$ in an optimal solution because, due to the unit capacity of a vehicle, the number of vehicles required for any solution to serve the demands in $T_v$ is at least $\lceil D(T_v) \rceil$ and each such vehicle passes $e$ at least twice (one is in a forward direction and the other is in a backward direction). Thus, we have the following lemma.

**Lemma 1.** *$\sum_{e \in E} LB(e)$ gives a lower bound of the optimal cost of TREE-CVRP.*

## 3 Reforming Operations

Our approximation algorithm repeats the following two steps until all the demands are served. The first step is a reforming step in which we reshape a given

tree following seven different operations all of which are "safe" in the sense that they do not increase the lower bound on the cost given in Lemma 1. The second is to choose an appropriate subtree and choose among a few possible strategies depending on the cases the best one to serve the demands in the subtree.

The first reforming operation $R_1$ is applicable when some nodes have demands greater than or equal to 1. Suppose that a node $v$ has a demand $D(v) \geq 1$. Then, we allocate $k = \lfloor D(v) \rfloor$ vehicles to $v$ to serve $k$ units of its demand. This operation results in demand at $v$ less than one. Note that this operation is apparently safe. Thus, it is assumed that each demand is less than one.

The second operation $R_2$ is to remove positive demand from each internal node. If there is any internal node $v$ with positive demand, we create a new node connected with $v$ by an edge of weight zero and descend the weight of $v$ to the new node. It is easy to see that this operation is safe. Therefore, we can assume that positive demand is placed only at leaves.

The third operation $R_3$ is applied to a pair of nodes $(u, v)$ such that a leaf $v$ is a unique child of $u$. If $D(u) + D(v) \leq 1$, we contract the edge $(u, v)$, i.e., delete $(u, v)$ and the node $v$, after replacing the demand $D(u)$ at $u$ by $D(u) + D(v)$ and then increasing the cost of the edge to $u$ by $w(u, v)$. On the other hand, if $D(u) + D(v) > 1$, then we send one vehicle to serve the full demand at $v$ and serve a partial demand at $u$ to fulfill the capacity, and then reduce the demand at $u$ accordingly. The edge to $v$ is removed together with $v$.

The fourth operation $R_4$ is to merge a subtree whose demand is less than or equal to 1 into a single edge. Namely, for an internal node $v$ with $D(T_v) \leq 1$, $T_v$ is replaced by a single edge $(v, v')$ with edge weight equal to $w(T_v)$ and $D(v') = D(T_v)$. Since $D(T_v) \leq 1$ holds, this operation is also safe.

To define more essential reform operations for approximation algorithm, we need some more assumptions and definitions.

A node $v$ of a tree $T$ is called a **p-node** if

(i) $v$ is an internal node, and
(ii) all of the children of $v$ are leaves, and
(iii) the sum of the demands at those children is between 1 and 2.

A node $u$ is called a **q-node** if

(i) the sum of the demands in the subtree $T_u$ rooted at $u$, denoted by $D(T_u)$, is at least 2, and
(ii) no child of $u$ has the property (i).

The fifth operation $R_5$ is to merge leaves of nodes. For a node $u$, let $\{v_1, v_2, \ldots, v_k\}$ be a subset of its children that are leaves. By $w_i$ we denote the weight of the edge $(u, v_i)$. We examine every pair of leaves. For the pair $(v_i, v_j)$ we check whether the sum of their weights exceeds 1. If $D(v_i) + D(v_j) \leq 1$, then we merge them. Exactly speaking, we remove the leaf $v_j$ together with its associated edge $(u, v_j)$ after replacing the demand of $v_i$ with $D(v_i) + D(v_j)$ and the weight $w_i$ with $w_i + w_j$. We repeat this process while there is any mergeable pair of leaves. Figure 1 illustrates this merging process. This operation is useful particularly for p- and q-nodes to reduce the number of possible cases to be considered.

An important property of the resulting tree after performing $R_5$ to p-nodes is that any p-node has at most three children (leaves) since otherwise the sum of the demands of those children exceeds 2, a contradiction to the definition of a p-node. Thus, we can assume that any p-node has at most three children.

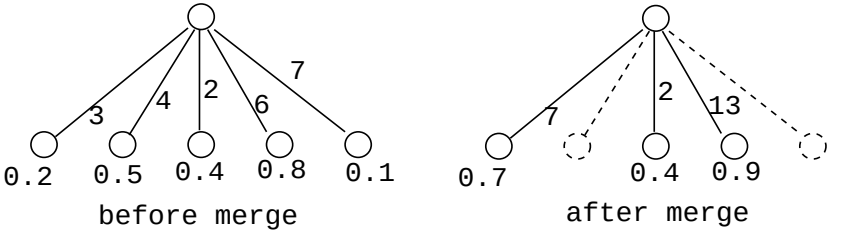



**Fig. 1.** The merging operation.

Now we have a limited number of situations for q-nodes listed as follows:
**Case (a):** A q-node $u$ has more than one p-node in its descendants. In this case, children of the q-node are either p-nodes or leaves (see Figure 2(a)).
**Case (b):** A q-node $u$ has exactly one p-node $v$ in its descendants. In this case we may have arbitrary number of internal nodes on the unique path from $u$ to $v$ each of which has only one edge connected to a leaf (see Figure 2(b)).
**Case (c):** A q-node $u$ has no p-node $v$ in its descendants. In this case, all of the children of the q-node are leaves (see Figure 2(c)). This is just like a p-node except that the sum of weights exceeds 2.

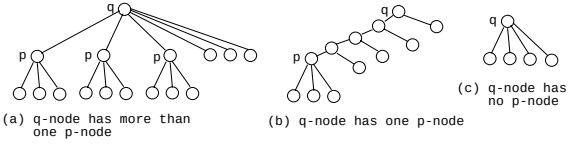



**Fig. 2.** The three cases for a q-node.

There are only two types of p-nodes since they have two or three children (leaves). The sixth reforming operation $R_6$ removes p-nodes having only two children. This is done by connecting those children directly by the parent of the p-node. Formally, suppose a p-node $v$ is connected to its parent $u$ by an edge of weight $a$ and to two children $v_1$ and $v_2$ by edges of weights $w_1$ and $w_2$, respectively. Then, $v_i$ is connected directly to $u$ by an edge of weight $w_i + a$. Note that this operation is also safe.

Yet another reforming operation $R_7$ is required in Cases (a) and (b) above. In these cases, for a p-node $v$, there may have some branches to leaves on the way from $u$ to $v$. Then, those leaves are placed as the children of the p-node with edges of weights equal to those for the branches. In addition, internal nodes on the path from $u$ to $v$ are erased so that the path is replaced by a single edge with

the weight equal to that for the path. This operation also preserves the lower bound. Notice that when $u$ has children which are leaves directly connected to $u$, such a leaf $x$ can be a candidate for this operation if $D(T_v) + D(x) < 2$ holds.

After all, we can assume that if a q-node has any p-node as its child then the p-node must have three children.

## 4 Approximation Algorithm

The approximation algorithm to be presented in this paper is based on the reformations preserving the lower bound and combining two or more different strategies. The main difference from the previous approximation algorithm given by Hamaguchi and Katoh [6] is the definition of a minimal subtree to which algorithmic strategies are applied. They introduced the notion of *D-minimality* and *D-feasibility*. That is, a vertex $v \in V$ is called *D-feasible* if $D(T_v) \geq 1$ and is called *D-minimal* if it is *D-feasible* but none of its children is. Their algorithm first finds a *D-minimal* vertex, and determines a routing of one or two vehicles that (partially) serve demands of vertices in $T_v$ by applying one of the two strategies depending on their merits.

Our algorithm pays attention to subtrees of demands exceeding 2 instead of 1. Usually this causes explosion of the possible cases, but the point here is that the reforming operations described above extremely simplify the possible cases. This is the main contribution of this paper.

Now, let us describe our algorithm. It first applies seven reforming operations $R_1$ through $R_7$ to an input tree so that any of these operations cannot be applied any more. The algorithm consists of a number of rounds. In each round, it focuses on a particular $q$-node $u$ and prepare a few strategies each of which allocates two, three, or four vehicles to (partially) serve the demands in the subtree $T_u$. Among strategies prepared, we choose the best one and apply it to $T_u$. In the same manner as in [6], for each strategy, we compute the cost of tours required by the strategy. We also compute the decrease of the lower bound. Namely, let $P$ denote the problem before applying the strategy. After applying the strategy, demands of nodes in the subtree $T_u$ are decreased, we obtain another problem instance $P'$ to which the algorithm will be further applied. The decrease of the lower bound is defined as $LB(P) - LB(P')$. The best strategy is the one giving the smallest ratio of the cost of tours to the decrease of the lower bound. As we shall show later, the smallest ratio is always at most 1.35078.

When there is not any $q$-node any more, it is the final round and is called *base case*. In this case, similarly to the other cases, it will be shown that this ratio is also at most 1.35078.

**Theorem 1.** *The approximation of our algorithm for TREE-CVRP is* 1.35078.

*Proof.* The proof technique is similar to the one by Hamaguchi and Katoh [6]. In fact, the theorem can be proved by induction on the number of rounds.

Assuming that the theorem holds for problem instances that require at most $k$ rounds, we consider the problem instance $P$ of TREE-CVRP for which our

algorithm requires $k+1$ rounds. Each time we find a q-node and apply an appropriate strategy based on the ratios defined above. Let $P'$ be the problem instance obtained from $P$ by decreasing demands served in this round. Let $LB(P')$ be the lower bound for the problem $P'$ and $LB_1$ be the decreased lower bound at this round. Let $cost(P)$, $cost_1$ and $cost(P')$ denote the total cost required for the original problem $P$ by our algorithm, the cost required by the first round and the cost for the remaining problem $P'$ to be required by our algorithm, respectively, (i.e., $cost(P) = cost_1 + cost(P')$). Then, we have

$$\frac{cost(P)}{LB(P)} \leq \frac{cost_1 + cost(P')}{LB_1 + LB(P')}. \tag{2}$$

Since $cost(P')/LB(P') \leq 1.35078$ holds from the induction hypothesis, it suffices to prove $cost_1/LB_1 \leq 1.35078$.

As we shall prove below (Lemmas 2 through 4), this inequality holds in every case (the base case will be proved in Lemma 5). Thus, we have the theorem. □

Recall that after each round, we obtain the new problem instance and we need to apply again reforming operations to the problem until any of seven reforming operations cannot be applied any more. Suppose there is at least one $q$-node. Strategies we prepare depend on the cases explained below.

**Case 1:** A q-node has more than one p-node as its children.

**Case 2:** A q-node has only one child of p-node.

**Case 3:** A q-node has no child of p-node.

In Case 1, we focus on arbitrary two p-nodes. Remaining p-nodes, if any, will be considered in later rounds of the algorithm. In Case 2, let $u$ and $v$ denote the q-node and the p-node respectively. From definition of a q-node, $u$ has at least one child other than the p-node. We choose arbitrary one child $v'$ other than the p-node. The algorithm then focuses on the subgraph consisting of edges $(u, v), (u, v')$ and the subtree $T_v$. We notice here that $D(T_v) + D(v') > 2$ since otherwise $v'$ can be shifted down to become a child of $v$ by reform operation $R_7$.

In Case 3, we can assume that the q-node has at least three leaves from definition of the q-node. We arrange those leaves in any order and find where the sum of the demands exceeds 2. Recall that the merging operation $R_5$ is already applied, and thus the sum of demands of the first four leaves certainly exceeds 2. We can conclude that there are only two possibilities, that is, either that of the first three leaves exceeds 2 (Subcase 3A) or that of the first four does (Subcase 3B).

Let us describe the algorithm for treating q-nodes depending on the above cases and then consider the base case. For each case, we shall explain how to schedule vehicles to serve demands of nodes that the algorithm focuses on, and prove that the ratio of the cost to the lower bound is at most 1.35078. In the proofs for all cases, we shall implicitly use the following simple facts:

**Fact 1:** $0 < x < y$ and $a > 0 \Longrightarrow \frac{y+a}{x+a} < \frac{y}{x}$.

**Fact 2:** For $p, q, r, s > 0$, $0 \leq a < b$ and $\frac{r}{p} > \frac{s}{q} \Longrightarrow \frac{ra+s}{pa+q} < \frac{rb+s}{pb+q}$.

**Case 1:** Let $u$ denote the q-node and let $x$ and $x'$ denote the two p-nodes the algorithm focuses on. The weight of the unique path from the tree root $r$ to $u$ is denoted by $a$, and those from $u$ to $x$ and $x'$ by $b$ and $b'$, respectively. Let $v_1, v_2$ and $v_3$ denote three leaves of the subtree $T_x$. We denote by $D_1, D_2$ and $D_3$ their demands, and by $w_1, w_2$, and $w_3$ weights of edges connecting leaves to their parent. Now, by assumption, $0 < D_i < 1, i = 1, 2, 3,\ 1 < D_1 + D_2 < 2$, and $1.5 < D_1 + D_2 + D_3 < 2$ ($D_1 + D_2 + D_3 > 1.5$ follows since otherwise the sum of some two demands among $D_1$, $D_2$ and $D_3$ is less than or equal to 1, and these two demands are mergeable, a contradiction). Here, without loss of generality we assume $w_1 \geq w_2 \geq w_3$. Similarly, let $v'_1, v'_2$ and $v'_3$ denote three leaves of the subtree $T_{x'}$, and define symbols $D'_1, D'_2, D'_3$ and $w'_1, w'_2, w'_3$ in the same manner as for $T_x$. Similarly, by assumption, $0 < D'_i < 1, i = 1, 2, 3,\ 1 < D'_1 + D'_2 < 2$, and $1.5 < D'_1 + D'_2 + D'_3 < 2$ hold, and $w'_1 \geq w'_2 \geq w'_3$ is assumed.

Here we prepare only one strategy that allocates two vehicles for each of subtrees $T_x$ and $T_{x'}$ to serve the demands of each of these subtrees; For $T_x$, the first vehicle serves the demand at $v_1$ and the partial demand at $v_3$ so that the sum of demands is equal to 1, and the second vehicle serves the demand at $v_2$ and the remaining demand at $v_3$. Similarly to $T_x$, we schedule two vehicles to serve demands of $T_{x'}$.

The cost required by these four vehicles is given by

$$8a + 4b + 4b' + 2w_1 + 2w_2 + 4w_3 + 2w'_1 + 2w'_2 + 4w'_3.$$

The decrease of the lower bound is given by

$$6a + 4b + 4b' + 2w_1 + 2w_2 + 2w_3 + 2w'_1 + 2w'_2 + 2w'_3.$$

The first term $6a$ comes from the fact that for every edge $e$ on the path from the root $r$ to $u$, the decrease of the lower bound $LB(e)$ is at least $6w(e)$ and at most $8w(e)$ from (2) because the decrease of $D(T_u)$ is between 3 and 4. Thus, the ratio of the cost of the tours to the decreased lower bound is given by

$$r_1 = \frac{8a + 4b + 4b' + 2w_1 + 2w_2 + 4w_3 + 2w'_1 + 2w'_2 + 4w'_3}{6a + 4b + 4b' + 2w_1 + 2w_2 + 2w_3 + 2w'_1 + 2w'_2 + 2w'_3}. \tag{3}$$

From $w_1 \geq w_2 \geq w_3$ and $w'_1 \geq w'_2 \geq w'_3$, we can show that $r_1$ is bounded by 4/3, using Facts 1 and 2:

$$r_1 \leq \frac{8a + 2w_1 + 2w_2 + 4w_3 + 2w'_1 + 2w'_2 + 4w'_3}{6a + 2w_1 + 2w_2 + 2w_3 + 2w'_1 + 2w'_2 + 2w'_3} \leq \frac{8a + 8w_3 + 8w'_3}{6a + 6w_3 + 6w'_3} = \frac{4}{3}. \tag{4}$$

**Lemma 2.** *In Case 1, the approximation ratio is at most* 4/3.

**Case 2:** Suppose that a q-node $u$ has only one p-node $v$ having three leaves $v_1, v_2$ and $v_3$ together with a single leaf $v_4$. As before, we denote the demand and edge weight associated with $v_i$ by $D_i$ and $w_i$, respectively, and $w_1 \geq w_2 \geq w_3$ is assumed. We denote the path length from the root to $u$ by $a$, and the weight of the edge between the $u$ and $v$ by $b$. The algorithm prepares different strategies depending on whether

$$D_1 + D_2 + D_4 \leq 2$$

holds or not.

**Subcase 2A:** $D_1 + D_2 + D_4 \leq 2$ holds. We prepare the following four strategies.

**Strategy 1:** This strategy allocates three vehicles. The first vehicle serves the full demand at $v_1$ and partial demand at $v_3$ to the full capacity. The second one serves the full demand at $v_2$ and the remaining demand at $v_3$ (which is less than 1). The third one serves the demand of $v_4$. Then, the ratio is given by

$$r_1 = \frac{6a + 4b + 2W + 2w_3}{4a + 4b + 2W}, \tag{5}$$

where $W = w_1 + w_2 + w_3 + w_4$.

**Strategy 2:** This strategy is the same as Strategy 1 except that the roles of $v_3$ and $v_4$ are exchanged. The ratio is given by

$$r_2 = \frac{6a + 4b + 2W + 2w_4}{4a + 4b + 2W}. \tag{6}$$

**Strategy 3:** This strategy allocates two vehicles, (1) to serve the full demand at $v_1$ and partial demand at $v_3$ to fill the capacity, and (2) to serve the full demand at $v_2$ and the remaining demand at $v_3$ and moreover partial demand at $v_4$. This is possible since $D_1 + D_2 + D_3 < 2$. The remaining demand at $v_4$ is left for the next round. Then, the ratio $r_3$ is defined by

$$r_3 = \frac{4a + 4b + 2W + 2w_3}{4a + 4b + 2W - 2w_4}. \tag{7}$$

**Strategy 4:** This strategy allocates two vehicles, (1) to serve the full demand at $v_1$ and partial demand at $v_2$ to fill the capacity, and (2) to serve the full demand at $v_4$ and the remaining demand at $v_2$ and moreover partial demand at $v_3$. This is possible since $D_1 + D_2 + D_4 < 2$. The remaining demand at $v_3$ is left for the next round. Then, the ratio $r_4$ is defined by

$$r_4 = \frac{4a + 4b + 2W + 2w_2}{4a + 2b + 2W - 2w_3}. \tag{8}$$

The smallest value among the above four values is evaluated by the following case analysis.

(i) $w_3 \geq w_4$. We choose the better one between Strategies 2 and 3. From $w_3 \geq w_4$, we have

$$r_2 \leq \frac{6a + 4b + 10w_4}{4a + 4b + 8w_4} \leq \frac{6a + 10w_4}{4a + 8w_4}, \text{ and } r_3 \leq \frac{4a + 10w_4}{4a + 6w_4}. \tag{9}$$

By letting $x = a/w_4$, we compute

$$\max_x \min\{\frac{6x + 10}{4x + 8}, \frac{4x + 10}{4x + 6}\}. \tag{10}$$

The maximum is attained when $\frac{6x+10}{4x+8} = \frac{4x+10}{4x+6}$ holds, i.e., $x = \frac{-1+\sqrt{41}}{4} \simeq 1.35078$. The maximum value of (10) is also $\frac{-1+\sqrt{41}}{4} \simeq 1.35078$.

(ii) $w_2 \geq w_4 \geq w_3$. We choose the better one between Strategies 1 and 3. The analysis is done in the same way as Case 1.

(iii) $w_4 \geq w_2$. We choose the better one between Strategies 1 and 4.

$$r_1 = \frac{6a + 4b + 2W + 2w_3}{4a + 4b + 2W} \leq \frac{6a + 4b + 2w_1 + 6w_2 + 2w_4}{4a + 4b + 2w_1 + 4w_2 + 2w_4} \leq \frac{6a + 4b + 10w_4}{4a + 4b + 8w_4}.$$

$$r_4 = \frac{4a+4b+2W+2w_2}{4a+2b+2W-2w_3} \leq \frac{4a+4b+2w_1+6w_2+2w_4}{4a+2b+2w_1+2w_2+2w_4} \leq \frac{4a+4b+10w_4}{4a+2b+6w_4}.$$

By letting $x = a/w_4, y = b/w_4$, we have

$$\min\{r_1, r_4\} \leq \min\{\frac{3x+2y+5}{2x+2y+4}, \frac{2x+2y+5}{2x+y+3}\}. \tag{11}$$

Letting $f(x,y)$ be the term on the right-hand side of (11), we compute $\max_{x,y\geq 0} f(x,y)$ using the known theorem for generalized fractional program [2]. For this, we consider the following parametric problem:

$$z(\lambda) = \max_{x,y} \min\{3x+2y+5-\lambda(2x+2y+4), 2x+2y+5-\lambda(2x+y+3)\}. \tag{12}$$

Letting

$$f_1(x,y) = 3x+2y+5-\lambda(2x+2y+4), \quad f_2(x,y) = 2x+2y+5-\lambda(2x+y+3),$$

the minimum of $z(\lambda)$ is attained when $f_1(x,y) = f_2(x,y)$, i.e., $x = \lambda y + \lambda$. Then substituting $x = \lambda(y+1)$ into $f_1(x,y)$ we have

$$f_1(x,y) = (-2\lambda^2 + \lambda + 2)y + 5 - \lambda - 2\lambda^2. \tag{13}$$

When $-2\lambda^2 + \lambda + 2 > 0$, i.e., $\lambda < (1+\sqrt{17})/4 \simeq 1.28$, $5 - \lambda - 2\lambda^2 > 0$ always holds for any $y \geq 0$. Thus, $f_1(x,y) = 0$ does not occur when $-2\lambda^2 + \lambda + 2 > 0$. When $-2\lambda^2 + \lambda + 2 < 0$, maximum of (13) is attained at $y = 0$, and the value of $f_1(x,y)$ at $y = 0$ is $5 - \lambda - 2\lambda^2$. Thus, when $5 - \lambda - 2\lambda^2 = 0$, i.e., $\lambda = 1.35078$, $z(\lambda) = \max_{x,y} \min\{f_1(x,y), f_2(x,y)\} = 0$ holds. Therefore, the approximation ratio is 1.35078 in this case. Summarizing the analysis made in (i), (ii) and (iii), we have the following lemma.

**Lemma 3.** *In Subcase 2A, the approximation ratio is at most* 1.35078.

The proof for the Subcase 2B ($D_1 + D_2 + D_4 > 2$) proceeds similarly.

**Lemma 4.** *In Subcase 2B, the approximation ratio is at most* 1.35078.

**Subcase 3A:** A q-node $u$ has three leaves. Let $\hat{v}_1, \hat{v}_2$, and $\hat{v}_3$ be those three leaves. We denote by $\hat{D}_1, \hat{D}_2$ and $\hat{D}_3$ their demands, and by $\hat{w}_1, \hat{w}_2$, and $\hat{w}_3$ the edge weights. The weight of the unique path from the root $r$ to $u$ is denoted by $a$. This case can be treated as a special case of Subcase 2B in which $v_4 = \hat{v}_3$, $b = 0$, $D_3 = 0$, and $w_3 = 0$ while $v_i = \hat{v}_i$, $D_i = \hat{D}_i$ and $w_i = \hat{w}_i$ for $i = 1, 2$.
**Subcase 3B:** A q-node $u$ has four leaves. This case can be viewed as a special case of Case 2 in which the length of the path from $u$ to $v$ is equal to 0. Thus, it will be treated in Case 2.

Finally, we shall turn our attention to the base case where there is no q-node in the input tree, that is, the total sum of the demands is less than 2. Applying the reforming operations to the tree results in a simple tree of the three forms depending on the number of leaves: (a) the root $r$ has only one leaf, (b) $r$ has two leaves, and (c) $r$ has one child $u$ which has three leaves.

In case (a), the remaining demand is less than one, and thus is served optimally by a single vehicle. In case (b), we allocate two vehicles, one for each leaf.

It is easy to see that this strategy is also optimal. Now let us consider case (c) in which a p-node $u$ has three leaves $v_1, v_2$, and $v_3$. Symbols $D_i$, $w_i$, and $a$ are used as before. We also assume that $w_1 \geq w_2 \geq w_3$. We allocate two vehicles; one to serve the demands $D_1$ and a fraction of $D_3$ so as to fill the full capacity of the vehicle, and another to serve $D_2$ and the remaining demand of $D_3$. Then, the ratio is given by

$$r_1 = \frac{4a + 2w_1 + 2w_2 + 4w_3}{4a + 2w_1 + 2w_2 + 2w_3} \leq \frac{4a + 8w_3}{4a + 6w_3} \leq 4/3 < 1.35078. \tag{14}$$

**Lemma 5.** *In the base case, the approximation ratio is at most* 1.35078.

## 5 Conclusions

We have presented a new approximation algorithm for finding optimal tours to serve demands located at nodes of a tree-shaped network. Our new algorithm establishes the approximation ratio 1.35078 (exactly, $(\sqrt{41} - 1)/4$). This ratio seems to be almost best possible since there is an instance of TREE-CVRP for which the cost of an optimal solution is asymptotically 4/3 times larger than the lower bound of the cost. To have better ratio we have to improve the lower bound, which is left for future research.

## References

1. I. Averbakh and O. Berman, Sales-delivery man problems on treelike networks, *Networks*, 25 (1995), 45-58.
2. J.P. Crouzeix, J.A. Ferland and S. Schaible, An algorithm for generalized fractional programs, *J. of Optimization Theory and Applications*, 47 (1985), 35-49.
3. M.L. Fischer. Vehicle Routing. in *Network Routing,* Handbooks in Operations Research and Management Science, **8**, Ball, M. O., T. L. Magnanti, C. L. Monma and G. L. Nemhauser (Eds.), Elsevier Science, Amsterdam, 1-33, 1995.
4. G. Frederickson, Notes on the complexity of a simple transportation problem, *SIAM J. Computing*, 22-1 (1993), 57-61.
5. B.L. Golden and A. A. Assad (Eds.). *Vehicle Routing: Methods and Studies,* Studies in Manag. Science and Systems 16, North-Holland Publ., Amsterdam, 1988.
6. S. Hamaguchi and N. Katoh. A Capacitate Vehicle Routing Problem on a Tree, Proc. of ISAAC'98, *LNCS 1533, Springer-Verlag* 397-406, 1998.
7. Y. Karuno, H. Nagamochi, T. Ibaraki, Vehicle Scheduling on a Tree with Release and Handling Times, Proc. of ISAAC'93, *LNCS 762, Springer-Verlag* 486-495, 1993.
8. Y. Karuno, H. Nagamochi, T. Ibaraki, Vehicle Scheduling on a Tree to Minimize Maximum Lateness, *Journal of the Operations Research Society of Japan*, Vol. 39, No.3 (1996) 345-355.
9. M. Labbé, G. Laporte and H. Mercure. Capacitated Vehicle Routing Problems on Trees, *Operations Research*, Vol. 39 No. 4 (1991) 616-622.

# Approximation Algorithms for Channel Assignment with Constraints*

Jeannette Janssen[1] and Lata Narayanan[2]

[1] Department of Mathematics and Statistics,
Dalhousie University, Halifax,
Nova Scotia, Canada, B3H 3J5,
email: janssen@mscs.dal.ca, FAX: (902) 494-5130.
[2] Department of Computer Science, Concordia University,
Montreal, Quebec, Canada, H3G 1M8.
email: lata@cs.concordia.ca, FAX (514) 848-2830.

**Abstract.** Cellular networks are generally modeled as node-weighted graphs, where the nodes represent cells and the edges represent the possibility of radio interference. An algorithm for the *channel assignment* problem must assign as many channels as the weight indicates to every node, such that any two channels assigned to the same node satisfy the *co-site* constraint, and any two channels assigned to adjacent nodes satisfy the *inter-site* constraint.
We describe several approximation algorithms for channel assignment with arbitrary co-site and inter-site constraints and reuse distance 2 for odd cycles and so-called *hexagon* graphs that are often used to model cellular networks. The algorithms given for odd cycles are optimal for some values of constraints, and have performance ratio at most $1+1/(n-1)$ for all other cases, where $n$ is the length of the cycle. Our main result is an algorithm of performance ratio at most $4/3+\epsilon$ for hexagon graphs with arbitrary co-site and inter-site constraints.

## 1 Introduction

The demand for wireless telephony and other services has been growing dramatically over the last decade. As a result of this, radio spectrum resources are scarce, and their efficient use becomes of critical importance. The cellular concept was proposed as an early solution to the problem of spectrum congestion. By dividing the service area into small coverage areas called *cells* served by low power transmitters, it became possible to *reuse* the same frequencies in different cells, provided they are far enough apart. With growing demand, it is necessary to perform this reuse as efficiently as possible, while ensuring that radio interference is at acceptable levels.

Cellular networks are generally modeled as node-weighted graphs, where the nodes represent the cells, and the edges represent the possibility of radio frequency interference. The weight on a node represents the number of calls originating in the cell represented by the node. The base station in a cell must

* Research supported in part by NSERC, Canada.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 327–336, 1999.
© Springer-Verlag Berlin Heidelberg 1999

assign frequency channels to each call originating in the cell. However, this assignment of channels must satisfy certain interference constraints. In particular, channels that are too close together may interfere with each other when they are assigned to calls that originate in the same or adjacent cells. These interference constraints can be represented by a set of integers $c_0 \geq c_1 \geq c_2, \ldots$, where $c_i$ is the minimum separation required between two channels assigned to calls in cells that are distance $i$ apart in the network. The parameter $c_0$ which is the minimum gap between two channels assigned to the same cell is called the *co-site* constraint and the other constraints are called *inter-site* constraints. The minimum graph distance between nodes that can be assigned the same channel is called the *reuse distance*. In order to optimize the use of the spectrum, the objective of the channel assignment algorithm is to minimize the *span* of the assignment, that is, the difference between the largest numbered channel used and the smallest channel used.

When $c_0 = c_1 = 1$, and $c_i = 0$ for all $i > 1$, the problem reduces to the multicolouring problem, which has been widely studied. The problem is NP-hard for many classes of graphs, including the so-called hexagon graphs, which have traditionally been used to represent cellular networks [MR97]. Hexagon graphs are subgraphs of the triangular lattice (see Figure 1). They are particularly relevant to channel assignment, since they represent a regular cellular layout where the cells are hexagonal, and interference only occurs between neighboring cells. Optimal solutions for this restricted channel assignment problem are possible for some classes of graphs including complete graphs, bipartite graphs, odd cycles, and outerplanar graphs [NS97]. When the chromatic number of the underlying graph is $k$, an approximation algorithm with a performance ratio of $k/2$ has been shown [JK95]. For hexagon graphs, approximation algorithms with performance ratio of $4/3$ are known [NS97,MR97]. For the case $c_0 = c_1 = c_2 = 1$, and $c_i = 0$ for all $i > 2$, an approximation algorithm with performance ratio $7/3$ is given in [FS98].

In this paper, we study the case of reuse distance 2, but arbitrary co-site and inter-site constraints. More precisely, we study the case $c_0 \geq c_1$ and $c_i = 0$ for all $i > 1$. Thus channels assigned to the same cell must differ by at least $c_0$ and those assigned to adjacent cells must differ by at least $c_1$. For the case $c_1 = 1$, Schabanel *et al.* [SUZ97] give a $4/3$ approximation algorithm for hexagon graphs. For arbitrary $c_0$ and $c_1$, a tight bound for cliques was given by Gamst [Gam86], and an optimal algorithm for bipartite graphs was given by Gerke [Ger99].

We give the first known algorithms with provable performance guarantees for channel assignment with arbitrary constraints in odd cycles and hexagon graphs. The performance of our algorithms is evaluated using known lower bounds based on the maximum weight on a node, the total weight and the maximal number of nodes that can receive the same channel, or the weights on a clique and their distribution. We first show six simple algorithms for bipartite graphs, odd cycles, and 3-colourable graphs. Using these as building blocks, we derive an optimal algorithm for odd cycles when $c_0 \geq \frac{2n}{n-1} c_1$, where $n$ is the length of the cycle. For the case where $c_0 < \frac{2n}{n-1} c_1$ we give near-optimal algorithms with performance ratio at most $1 + \frac{1}{n-1}$.

For hexagon graphs, we give approximation algorithms with performance ratio at most 4/3, when $c_1 \leq c_0 \leq 2c_1$, and $9c_1/4 \leq c_0 < 3c_1$. For the intermediate case $2c_1 < c_0 < 9c_1/4$, the performance ratio of the algorithm is less than $4/3 + 1/100$. There is a straightforward optimal algorithm for hexagon graphs when $c_0 \geq 3c_1$. Thus for arbitrary co-site and inter-site constraints, our algorithms nearly match the performance of the best known algorithm for the case when co-site and inter-site constraints are both exactly equal to 1.

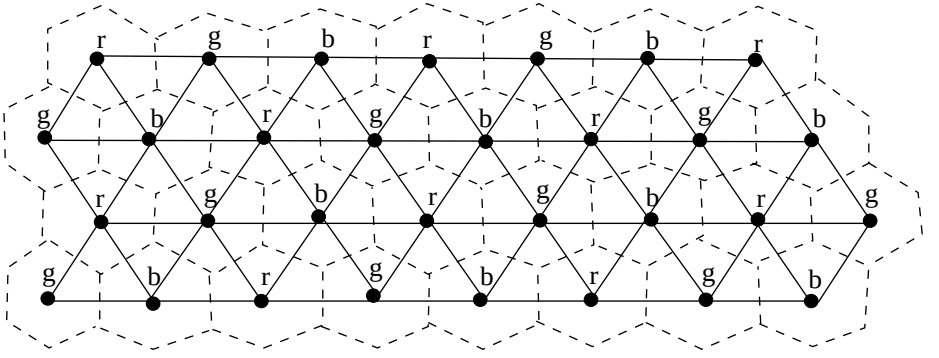



**Fig. 1.** A hexagon graph and a 3-colouring of the nodes of the graph. The hexagonal area around each node represents the calling area it serves.

The rest of the paper is organized as follows. We define the problem formally in Section 2. We give simple (but not necessarily optimal) algorithms for channel assignment in bipartite graphs, odd cycles, and hexagon graphs in Section 3. Near-optimal algorithms for odd cycles and approximation algorithms for hexagon graphs are then given in in Section 4.

## 2 Preliminaries

For the basic definitions of graph theory we refer to [BM76]. A **stable set** in a graph is a set of nodes of which no pair is adjacent. A **clique** in a graph is a set of nodes of which every pair is adjacent.

A **constrained graph** $G = (V, E, c_0, c_1)$ is a graph $G = (V, E)$ and positive integer parameters $c_0$ and $c_1$ representing the reuse differences prescribed between pairs of channels assigned to the same node and adjacent nodes, respectively. A **constrained, weighted graph** is a pair $(G, w)$ where $G$ is a constrained graph and $w$ is a positive integral weight vector indexed by the nodes of $G$. The component of $w$ corresponding to node $u$ is denoted by $w(u)$ and called the **weight** of node $u$. The weight of node $u$ represents the number of calls to be serviced at node $u$. We use $w_{max}$ to denote $max\{w(v) \mid v \in V\}$ and $w_{min}$ to denote the corresponding minimum weight of any node in the graph.

A **channel assignment** for a constrained, weighted graph $(G, w)$ where $G = (V, E, c_0, c_1)$ is an assignment $f$ of sets of non-negative integers (which will represent the channels) to the nodes of $G$ which satisfies the conditions:

$$|f(u)| = w(u) \qquad (u \in V),$$
$$i \in f(u) \text{ and } j \in f(v) \Rightarrow |i - j| \geq c_1 \quad ((u, v) \in E,\ u \neq v),$$
$$i, j \in f(u) \text{ and } i \neq j \Rightarrow |i - j| \geq c_0 \quad (u \in V).$$

The **span** $S(f)$ of a channel assignment $f$ of a constrained weighted graph is the difference between the lowest and the highest channel assigned by $f$, in other words, $S(f) = \max f(V) - \min f(V)$, where $f(V) = \bigcup_{u \in V} f(u)$. The span $S(G, w)$ of a constrained, weighted graph $G$ and a positive integer vector $w$ indexed by the nodes of $G$ is the minimum span of any channel assignment for $(G, w)$. We use $\chi(G, w)$ to denote the minimal *number* of channels needed for an assignment of the weighted, unconstrained graph $G$. Note that $\chi((V, E, c_0, c_1), w) = S((V, E, 1, 1), w) + 1$, where the additive term is due to the fact that $k$ consecutive channels have a span of $k - 1$. The **spectrum** used by a channel assignment algorithm is the interval of channels between the highest and lowest channels assigned.

A channel assignment $f$ is said to be **optimal** for a weighted constrained graph $G$ if $S(f) = S(G, w) + \Theta(1)$. Here we consider the span to be a function of the weights and the size of the graph, so the $\Theta(1)$ term can include terms that depend on the constraints $c_0$ and $c_1$. An approximation algorithm for channel assignment has performance ratio $k$ when the span of the assignment produced by the algorithm on $(G, w)$ is at most $kS(G, w) + \Theta(1)$.

The following lower bounds will be used to evaluate our algorithms and calculate the performance ratio. The first bound derives from the fact that any two channels on the same node must be at least $c_0$ apart. The next two bounds are based on weights and their distribution on cliques in the graph, and are derived from a bound for cliques given by Gamst [Gam86]. The last two bounds use the fact that, because of the inter-site constraint, all nodes that receive channels from any particular channel interval of length $c_1$ must form a stable set.

**Theorem 1 (Known lower bounds).**
*Let $G = (V, E, c_0, c_1)$ be a constrained graph, and $w \in \mathbf{Z}_+^V$ a weight vector for $G$. Then*

$$S(G, w) \geq c_0 w_{max} - c_0 \tag{1}$$
$$S(G, w) \geq max\{c_0 w(u) + (2c_1 - c_0)w(v) | (u, v) \in E\} - c_0 \quad \textit{when } c_0 \leq 2c_1 \tag{2}$$
$$S(G, w) \geq max\{c_0 w(u) + (2c_1 - c_0)(w(v) + w(t)) | \{u, v, t\} \textit{ a clique}\} - c_0 \quad \textit{when } c_0 \leq 2c_1 \tag{3}$$
$$S(G, w) \geq c_1 max\{w(u) + w(v) + w(t) | \{u, v, t\} \textit{ a clique}\} - c_1 \tag{4}$$
$$S(G, w) \geq \frac{2c_1}{n - 1} \sum_{v \in V} w(v) - c_1 \quad \textit{when } G \textit{ is an odd cycle of length } n \tag{5}$$

## 3 Basic Algorithms for Channel Assignment

In this section, we provide six simple algorithms for channel assignment in specific situations. The first two, Algorithms A and B, are optimal algorithms for bipartite graphs for the cases $c_0 \geq 2c_1$ and $c_1 \leq c_0 < 2c_1$ respectively. Note that essentially the same algorithms are given and proved to be exactly optimal (without a constant additive term) in [Ger99]. We give them here for completeness, as we use these algorithms to prove further results in the next section. Also, our exposition is simpler as we ignore constant additive terms in our definition of optimality. Due to lack of space, we omit the proofs of correctness of these algorithms; they can be found in [JN99].

Algorithms C and D both perform channel assignment for odd cycles. Neither of these algorithms is optimal, but we use them in the next section in combination with other algorithms to obtain optimal and near-optimal bounds for odd cycles. It is easy to check that all the algorithms given in this section have linear-time implementations.

Algorithms E and F are for 3-colourable graphs. While Algorithm E's performance is always at least as good as that of Algorithm F, the latter has some room for channel borrowing. Thus when used in combination with other algorithms, it can have an advantage over Algorithm E. We will use these two algorithms combined with modification techniques and with the algorithms for bipartite graphs, to derive near-optimal algorithms for hexagon graphs.

### Algorithm A (for bipartite graphs when $c_0 \geq 2c_1$)

Let $G = (V, E, c_0, c_1)$ be a constrained bipartite graph of $n$ nodes, where $c_0 \geq 2c_1$ and $w$ an arbitrary weight vector. Let each node be coloured red or green according to the bipartition. Red nodes use as many colours as necessary from the set $0, c_0, 2c_0, \ldots, (w_{max}-1)c_0$. Green nodes use as many colours as necessary from the set $c_1, c_0 + c_1, \ldots, (w_{max} - 1)c_0 + c_1$. It is easy to see that the span of the assignment is no more than $c_0 w_{max} - c_1$.

### Algorithm B (for bipartite graphs when $c_1 \leq c_0 < 2c_1$)

Let $G = (V, E, c_0, c_1)$ be a constrained bipartite graph of $n$ nodes, where $c_1 \leq c_0 \leq 2c_1$, and $w$ an arbitrary weight vector. Let each node be coloured red or green according to the bipartition.

Given a node $v$, define $p(v) = max\{w(u) \mid (u, v) \in E\}$. The general idea is that red nodes always get channels starting from 0 and the green nodes get channels starting from $c_1$. If a node has demand greater than any of its neighbors then it initially gets some channels that are $2c_1$ apart (in order to allow interspersing the channels of its neighbors) and the remaining distance $c_0$ apart. A more precise description can be found in [JN99]. It is not difficult to see that the span of the assignment above is at most $max_{(u,v)\in E}\ \{c_0 w(u) + (2c_1 - c_0)w(v)\}$ (see [Ger99] for a complete explanation).

**Algorithm C (for odd cycles)**

Let $G = (V, E, c_0, c_1)$ be a constrained cycle of $n$ nodes, where $n > 3$ is odd, and $w$ be an arbitrary weight vector. Fix $c = max\{c_0, c^*\}$ where $c^* = 2nc_1/(n-1)$. For convenience, the nodes of the cycle are $\{1, \ldots n\}$, numbered in cyclic order, where node 1 is a node of maximum weight in the cycle.

The algorithm is based on an initial basic assignment of one channel per node. Additional channels are then given to each node by adding the appropriate number of multiples of $c$ to the basic assigned channel of the node. The basic assignment uses a spectrum $[0, c]$. Initially, it will proceed by assigning the channel obtained by adding $c_1$ (modulo $c$) to the previously assigned channel to the next node in the cycle. At a certain point, it will switch to an alternating assignment.

More precisely, let $m > 1$ be the smallest odd integer such that $c \geq \frac{2m}{m-1}c_1$. Since $c \geq c^*$, a value of $m \leq n$ satisfying this must exist, and $m$ is well-defined. Note that the definition of $m$ implies that $c < \frac{2(m-2)}{m-3}c_1$. Let $b$ be the basic assignment assigned as follows:

$$b(i) = \begin{cases} (i-1)c_1 \bmod c & \text{when } 1 \leq i \leq m, \\ 0 & \text{when } i > m \text{ and } i \text{ is even}, \\ (m-1)c_1 \bmod c & \text{when } i > m \text{ and } i \text{ is odd}. \end{cases}$$

To each node $i$, the algorithm assigns the channels $b(i) + jc$, where $j = 0, \ldots, w(i) - 1$.

*Span of the assignment:* The span of the channel assignment described here is at most $w_{max} max\{c_0, c^*\}$ where $c^* = 2nc_1/(n-1)$).

**Algorithm D (for odd cycles)**

Let $G = (V, E, c_0, c_1)$ be a constrained cycle of $n$ nodes, where $n > 3$ is odd, and $w$ be an arbitrary weight vector. We state the following fact about $\chi(G, w)$.

**Fact 2** *For $G$ an odd cycle of $n$ nodes,*

$$\chi(G, w) = max\{2\Sigma_{v \in V} w(v)/(n-1), max\{w(u) + w(v) | (u, v) \in E\}\}.$$

This algorithm is a straightforward adaptation of the optimal algorithm for multicolouring an odd cycle (without constraints) given in [NS97].

Fix $c = max\{c_0, 2c_1\}$, and $\omega \geq max\{\chi(G, w), 2w_{max}\}$. We use the spectrum $[0, \ldots, c\lceil \omega/2 \rceil]$, and describe a fixed sequence of channels to be used from this spectrum in that order by the multicolouring algorithm. The sequence is:

$$(0, c, 2c, \ldots, (\lceil \frac{\omega}{2} \rceil - 1)c, c_1, c + c_1, 2c + c_1, \ldots, (\lfloor \frac{\omega}{2} \rfloor - 1)c + c_1)$$

It is straightforward to verify that there are exactly $\omega$ channels in this sequence. We now proceed as for multicolouring, using the sequence given here to assign channels rather than a continuous part of the spectrum.

Precisely, let $k$ be the smallest integer such that $\Sigma_{i=1}^{2k+1} w(i) \leq k\omega$. Nodes 1 through $2k$ are assigned *contiguous* channels in a cyclic manner from the spectrum given. Specifically, for $1 \leq j \leq 2k$, node $j$ is assigned the $\ell$-th through $m$-th channel of the spectrum, where $\ell = 1 + \sum_{i=1}^{j-1} w(i)$ and $m = \sum_{i=1}^{j} w(i)$. This assignment is done cyclically, so if $\ell > m$, then the channels 'wrap around' from the $\ell$-th channel to the end of the spectrum, and back from the beginning of the spectrum to the $m$-th channel. The assignment for nodes $2k+1$ through $n$ is based on their parity; for $2k+1 \leq i \leq n$, node $i$ is assigned the first $w(i)$ channels of the spectrum if $i$ is even, or the last $w(i)$ channels if $i$ is odd.

*Span of the assignment:* The span of the assignment is $max\{c_0, 2c_1\}\lceil \omega/2 \rceil$ where $\omega$ is any integer which is at least $max\{2w_{max}, \chi(G, w)\}$.

**Algorithm E (for 3-colourable graphs)**

Let $G = (V, E, c_0, c_1)$ be a 3-colourable graph, and $w$ be an arbitrary weight vector. Fix $c = max\{3c_1, c_0\}$.

This algorithm uses a colouring of the nodes of the graph with colours red, blue and green. It assigns at most $w_{max}$ channels to each node. For $i = 0, \ldots, w_{max} - 1$, the channels $ic$ are reserved for red nodes, the channels $ic + c_1$ are reserved for green nodes, and the channels $ic + 2c_1$ are reserved for blue nodes. Each node $v$ is assigned $w(v)$ channels from its own reserved sets of channels.

*Span of the assignment:* This algorithm produces an assignment of span at most $cw_{max} - c_1 = max\{3c_1, c_0\}w_{max} - c_1$.

**Algorithm F (for 3-colourable graphs)**

Let $G = (V, E, c_0, c_1)$ be a 3-colourable graph, and $w$ be an arbitrary weight vector. Fix $c = max\{c_1, c_0/2\}$ and $T \geq 3w_{max}$.

We use a spectrum of $T$ channels, with consecutive channels separated by $c$, where channels reserved for different colours are interspersed. (This alternation of channels was first used in [SUZ97].) We assume for ease of explanation that $T$ is a multiple of 6. Precisely, the red channels consist of a first set $R_1 = [0, 2c, \ldots, (T/3 - 2)c]$ and a second set $R_2 = [(T/3 + 1)c + c_0, (T/3 + 3)c + c_0, \ldots, (2T/3 - 1)c + c_0]$. The blue channels consist of first set $B_1 = [(T/3)c + c_0, (T/3 + 2)c + c_0, \ldots, (2T/3 - 2)c + c_0]$ and second set $B_2 = [(2T/3 + 1)c + 2c_0, (2T/3 + 3)c + 2c_0, \ldots, (T - 1)c + 2c_0]$, and the green channels consist of first set $G_1 = [(2T/3)c + 2c_0, (2T/3 + 2)c + 2c_0, \ldots, (T - 2)c + 2c_0]$ and second set $G_2 = [c, 3c, \ldots, (T/3 - 1)c]$. Thus, we can think of the spectrum as being divided into three parts, each containing $T/3$ channels (with a gap of at least $c_0$ between the parts).

Each node $v$ is assigned $w(v)$ channels from those of its colour class, where the first set is exhausted before starting on the second set, and lowest numbered channels are always used first within each set.

*Span of the assignment:* The span equals $cT + 2c_0 = \max\{c_1, c_0/2\}T + 2c_0$, where $T$ is at least $3w_{max}$.

## 4 Approximation Algorithms for Odd Cycles and Hexagon Graphs

In this section, we will outline how variations and combinations of the algorithms described in Section 3 can be used to derive optimal and near-optimal algorithms for channel assignment in odd cycles and hexagon graphs.

Our result for odd cycles is given in the following theorem:

**Theorem 3.** *For any $G = (V, E, c_0, c_1)$ a constrained odd cycle of length $n$, and $w$ an arbitrary weight vector, there is a linear time algorithm for channel assignment in $(G, w)$ with performance ratio*

$$\begin{cases} 1 & \text{when } c_0 \geq c^* = 2nc_1/(n-1) \text{ (that is, an optimal algorithm)}, \\ 1 + 1/n & \text{when } 2c_1 \leq c_0 < c^* = 2nc_1/(n-1) \\ 1 + 1/(n-1) & \text{when } c_0 < 2c_1. \end{cases}$$

*Proof.* When $c_0 \geq c^*$, Algorithm C is optimal. When $2c_1 \leq c_0 < c^*$, we use a combination of Algorithms A, C, and D to obtain the result. For the remaining case, we use a combination of Algorithms B and C. Details and proofs can be found in [JN99].

Next, we describe approximation algorithms for channel assignment with constraints in hexagon graphs. The algorithms we describe use a standard 3-colouring of hexagon graphs, which gives a partition of the nodes into red, blue, and green nodes (see Figure 1). The first two theorems are based on Algorithm E and give results for the cases $c_0 \geq 3c_1$, where the algorithm is optimal, and for $c_1 \leq c_0 < 3c_1$. The last two theorems use a combination of the algorithms given in Section 3 with additional modifications, and deal with the cases where $2c_1 \leq c_0 \leq (9/4)c_1$ and $c_1 \leq c_0 \leq 2c_1$, respectively.

**Theorem 4.** *For any $c_0 \geq 3c_1$, $G = (V, E, c_0, c_1)$ a constrained hexagon graph, and $w$ an arbitrary weight vector, there is an optimal linear time approximation algorithm for channel assignment in $(G, w)$.*

*Proof.* Since $c_0 \geq 3c_1$, Algorithm E gives an assignment of span at most $c_0 w_{max}$, and it follows from lower bound (1) of Theorem 1 that this is an optimal assignment.

**Theorem 5.** *For any $c_1 \leq c_0 < 3c_1$, $G = (V, E, c_0, c_1)$ a constrained hexagon graph, and $w$ an arbitrary weight vector, there is a linear time approximation algorithm for channel assignment in $(G, w)$ that has performance ratio $3c_1/c_0$.*

*Proof.* Since $3c_1 \geq c_0$, Algorithm E gives an assignment of span at most $3c_1 w_{max}$. By lower bound (1), $S(G, w) \geq c_0 w_{max} - c_0$, so this span is at most $(3c_1/c_0)S(G, w) + \Theta(1)$, as claimed.

Note that when $c_0 \geq (9/4)c_1$, the performance ratio of the above algorithm is at most 4/3. In the following two theorems, we give algorithms that improve on the performance ratio given in Theorem 5 for values of $c_0 < (9/4)c_1$.

The two remaining algorithms in this section use the same type of strategy. First, each node is assigned enough channels from those assigned to its colour class to guarantee that there are no triangles left in the graph. Next, each node *borrows* any available channels of a designated borrowing colour. The resulting graph is then shown to be a bipartite graph, for which an optimal channel assignment is found. (This general approach was first used for multicolouring of hexagon graphs in [NS97] and [MR97].) The algorithms differ in the initial separation of channels into different colour classes. The algorithm of Theorem 7 uses an additional technique of *squeezing* channels when possible. Both borrowing and squeezing make use of already assigned parts of the spectrum, and thus do not add to the total span of the assignment. In the following, let $D$ represent the maximum of the sum of weights on any clique in the graph.

**Theorem 6.** *For any* $2c_1 < c_0 \leq (9/4)c_1$, $G = (V, E, c_0, c_1)$ *a constrained hexagon graph, and* $w$ *an arbitrary weight vector, there is a linear time approximation algorithm for channel assignment in* $(G, w)$ *that has performance ratio* $1 + 3\frac{c_0 - 2c_1}{c_0} + \frac{9c_1 - 4c_0}{3c_1}$.

*Proof.* The algorithm proceeds in four phases. The first two phases assign a total of $D$ channels, partly according to Algorithm E (Phase 1) and partly according to Algorithm F (Phase 2). The next phase is a borrowing phase in which nodes borrow any available channels of the borrowing colour, *i.e.* any channels that are unused by all of its neighbors of the borrowing colour. The resulting graph is shown to be a bipartite graph, for which an assignment is found in Phase 4 using Algorithm $A$. The details of the phases and the proof of correctness and span are omitted here due to lack of space, and can be found in [JN99].

The above theorem yields an algorithm with performance ratio that is always less than $4/3+1/100$. In particular, the maximum value of the performance ratio is reached when $c_0/c_1 = 3/\sqrt{2}$. When $c_0 = 2c_1$ or $c_0 = 9c_1/4$, the performance ratio is exactly $4/3$.

**Theorem 7.** *For any* $c_0 \leq 2c_1$, $G = (V, E, c_0, c_1)$ *a constrained hexagon graph, and* $w$ *an arbitrary weight vector, there is a linear time approximation algorithm for channel assignment in* $(G, w)$ *that has performance ratio* $4/3$.

*Proof.* The algorithm consists of four phases. First Algorithm F is used with a spectrum of $D$ channels. In Phase 2, nodes try to borrow available channels of the borrowing colour. In the next phase, some of the channels assigned are "squeezed" closer together where possible to accommodate more channels. The remaining graph is then bipartite, and Algorithm B is used. The details of the phases and the proof of correctness and span are omitted here due to lack of space, and can be found in [JN99].

Table 1 below summarizes the results of this section. Note that the given values are upper bounds; as $c_0$ approaches $3c_1$, the performance ratio approaches 1, and similarly, the performance ratio approaches $4/3$ at both ends of the range $2c_1 < c_0 < 9c_1/4$.

| | $c_1 \leq c_0 \leq 2c_1$ | $2c_1 < c_0 < 9c_1/4$ | $9c_1/4 \leq c_0 < 3c_1$ | $c_0 \geq 3c_1$ |
|---|---|---|---|---|
| Perf. Ratio | $4/3$ | $4/3 + 1/100$ | $4/3$ | $1$ |

**Table 1.** Upper bounds on performance ratio on hexagon graphs for different values of $c_0$ and $c_1$.

## 5 Conclusions

We described new algorithms for channel assignment with arbitrary co-site and inter-site constraints on odd cycles and hexagon graphs. For odd cycles, our algorithms are optimal or near-optimal. We conjecture that in the cases where we do not achieve optimality, the existing lower bounds for odd cycles are inadequate. For hexagon graphs, for the case $c_0 < 3c_1$, we give approximation algorithms with performance ratios of at most $4/3 + \epsilon$ (when $c_0 \geq 3c_1$, there is a straightforward optimal algorithm). We point out that this matches the performance ratio of the best known algorithm for multicolouring on hexagon graphs, for most values in this range, and is only greater by a very small factor for all values (see Table 1).

Our algorithms are centralized and static algorithms. However, in practice, channel allocation is a distributed and online task; future work will involve the investigation of efficient online and distributed algorithms for this problem.

## References

BM76. J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications*. Macmillan Press Ltd, 1976.

FS98. Tomás Feder and Sunil M. Shende. Online channel allocation in fdma networks with reuse constraints. *Inform. Process. Lett.*, 67(6):295–302, 1998.

Gam86. A. Gamst. Some lower bounds for a class of frequency assignment problems. *IEEE Trans. Veh. Technol.*, 35(1):8–14, 1986.

Ger99. S. N. T. Gerke. Colouring weighted bipartite graphs with a co-site constraint. unpublished, 1999.

JK95. J. Janssen and K. Kilakos. Adaptive multicolouring. To appear in *Combinatorica*, 1999.

JN99. J. Janssen and L. Narayanan. Approximation algorithms for channel assignment with constraints. Available from `http://www.cs.concordia.ca/~faculty/lata`. Submitted for publication.

MR97. C. McDiarmid and B. Reed. Channel assignment and weighted colouring. Submitted for publication, 1997.

NS97. L. Narayanan and S. Shende. Static frequency assignment in cellular networks. In *Proceedings of SIROCCO 97*, pages 215–227. Carleton Scientific Press, 1997. To appear in *Algorithmica*.

SUZ97. N. Schabanel, S. Ubeda, and Zerovnik. A note on upper bounds for the span of frequency planning in cellular networks. *Submitted for publication*, 1997.

# Algorithms for Finding Noncrossing Steiner Forests in Plane Graphs

Yoshiyuki Kusakari[1], Daisuke Masubuchi, and Takao Nishizeki[1]

Graduate School of Information Sciences, Tohoku University
Sendai 980-8579, Japan
{kusakari, nishi}@nishizeki.ecei.tohoku.ac.jp

**Abstract.** Let $G = (V, E)$ be a plane graph with nonnegative edge lengths, and let $\mathcal{N}$ be a family of vertex sets $N_1, N_2, \cdots, N_k \subseteq V$, called *nets*. Then a noncrossing Steiner forest for $\mathcal{N}$ in $G$ is a set $\mathcal{T}$ of trees $T_1, T_2, \cdots, T_k$ in $G$ such that each tree $T_i \in \mathcal{T}$ connects all vertices in $N_i$, any two trees in $\mathcal{T}$ do not cross each other, and the sum of edge lengths of all trees is minimum. In this paper we give an algorithm to find a noncrossing Steiner forest in a plane graph $G$ for the case where all vertices in nets lie on two of the face boundaries of $G$. The algorithm takes time $O(n \log n)$ if $G$ has $n$ vertices.

## 1 Introduction

The Steiner tree problem is to find a minimum tree connecting a "net," *i.e.* a set of "pins," and often appears in the routing of VLSI layouts. One may often wish to find Steiner trees for many nets instead of a single net. Any two of the trees must not cross each other in the single layer VLSI layout due to a physical condition. Algorithms to solve such problems are desired for automatic routing of VLSI layouts [5,7,8].

One of such problems is finding, in a given plane graph with $k$ terminal-pairs, a set of $k$ noncrossing paths such that any two of them do not cross each other and the sum of their weights is minimum. This problem is a natural model for the single layer VLSI routing of "two-terminals nets," each of which contains exactly two pins. One may regard edges of a graph as routing regions, paths as routes, and each terminal pair of a path as a pair of pins connected by a route. One may wish to find a set of noncrossing paths the total weight of which is minimum. If all terminals are on at most two of the face boundaries of a plane graph, then such a set of $k$ noncrossing paths can be found in time $O(n \log n)$ for any $k \geq 1$, where $n$ is the number of vertices in a plane graph [9].

Another well-known problem is finding a Steiner tree on a plane graph [1,2,3,11]. Although finding a Steiner tree on a plane graph is very useful for VLSI routing, the problem is $NP$-complete [4] and it seems that there exists no efficient algorithm to solve the problem. However, if all terminals lie on a single face boundary of a plane graph, a Steiner tree can be found in time $O(l^3 n + l^2 n \log n)$ where $l$ is the number of terminals [1,3].

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 337–346, 1999.
© Springer-Verlag Berlin Heidelberg 1999

In this paper, we deal with a problem extended from the two problems above and defined as follows. For a plane graph $G = (V, E)$ with nonnegative edge lengths and a family $\mathcal{N}$ of $k$ vertex sets $N_1, N_2, \cdots, N_k$, called *nets*, find a set $\mathcal{T}$ of $k$ trees $T_1, T_2, \cdots, T_k$, called a *noncrossing Steiner forest*, such that

- for each $N_i \in \mathcal{N}$, tree $T_i \in \mathcal{T}$ connects all terminals in $N_i$;
- any two trees in $\mathcal{T}$ do not cross each other (a formal definition of crossing trees will be given in Section 2); and
- the *total weight* $\sum_{T_i \in \mathcal{T}} \sum_{e \in T_i} w(e)$ of the forest $\mathcal{T}$ is minimum, where $w$ is a weight function from $E$ to nonnegative real numbers.

An example of a noncrossing Steiner forest is depicted in Fig. 1, where $k = 7$, all terminals in net $N_i$, $1 \leq i \leq k$, are labeled by $i$, and $w(e) = 1$ for every edge $e$.

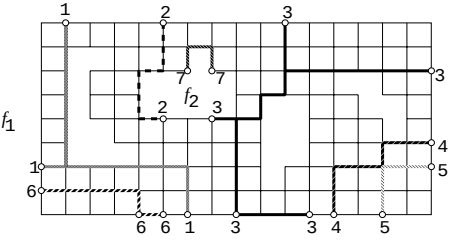



**Fig. 1.** A noncrossing Steiner forest.

If every net contains exactly two terminals, then a noncrossing Steiner forest is merely a set of noncrossing paths such that the sum of their lengths is minimum. On the other hand, if $k = 1$, that is, there is exactly one net, then a noncrossing Steiner forest is merely an ordinary Steiner tree. Since the Steiner tree problem is $NP$-complete for plane graphs [4], the noncrossing Steiner forest problem is $NP$-complete in general for plane graphs. However, some restrictions for the location of terminals may contribute to the construction of an efficient algorithm. In this paper we show that it is indeed the case. That is, we give an efficient algorithm for finding a noncrossing Steiner forest in a plane graph if all terminals are on at most two of the face boundaries as in Fig. 1. Our algorithm takes $O(n \log n)$ time if there are $n$ vertices in a graph.

This paper is organized as follows. In Section 2, we give some preliminary definitions. In Section 3, we give an algorithm for finding a noncrossing Steiner forest in a plane graph for the case where all terminals are on a single face boundary. In Section 4, we give an algorithm for finding a noncrossing Steiner forest for the case where all terminals are on at most two of the face boundaries. Finally, we conclude in Section 5.

## 2 Preliminaries

In this section we define terms and formally describe a noncrossing Steiner forest problem.

We denote by $G = (V, E)$ a graph consisting of a vertex set $V$ and an edge set $E$. We denote by $V(G)$ and $E(G)$ the vertex set and the edge set of $G$, respectively. Let $n$ be the number of vertices in $G$. Assume that $G$ is an undirected plane connected graph and that every edge $e \in E$ has a nonnegative weight $w(e)$. Furthermore we assume that $G$ is embedded in the plane $\mathbb{R}^2$. The image of $G$ in $\mathbb{R}^2$ is denoted by $Img(G) \subseteq \mathbb{R}^2$. A *face* of $G$ is a connected component of $\mathbb{R}^2 - Img(G)$. The *boundary* $B(f)$ of a face $f$ is the maximal subgraph of $G$ whose image is included in the closure of the face $f$. The unbounded face of $G$ is called the *outer face*, and the boundary of the outer face is called the *outer boundary*. For two graphs $G = (V, E)$ and $H = (W, F)$, we define $G \cup H = (V \cup W, E \cup F)$.

A set of vertices in $G$ which we wish to connect by a tree is called a *net*. All vertices in a net are called *terminals*. Let $\mathcal{N} = \{N_1, N_2, \cdots, N_k\}$ be a set of $k$ nets. For the sake of simplicity, we assume that $N_i \cap N_j = \phi$ for any two different nets $N_i, N_j \in \mathcal{N}$. Let $l_i = |N_i|$ and we write $N_i = \{u_{i1}, u_{i2}, \cdots, u_{il_i}\}$ for each $i$, $1 \leq i \leq k$. We assume that all $l_i$, $1 \leq i \leq k$, are fixed constants, and hence there exists a fixed integer $L$ such that $2 \leq l_i \leq L$ for all $i$, $1 \leq i \leq k$.

We assume that all terminals are on the boundaries of two faces, say $f_1$ and $f_2$. One may assume that $f_1$ is the outer face and $f_2$ is an *inner face*. For the sake of simplicity, we assume that $G$ is 2-connected and that $B(f_1)$ and $B(f_2)$ have no common vertices or edges.

For each net $N_i \in \mathcal{N}$, a tree in $G$ connecting all terminals in $N_i$ is called a *tree $T_i$ for net $N_i$ (in $G$)*. A *weight* $w(T_i)$ of $T_i$ is $\sum_{e \in E(T_i)} w(e)$. A *Steiner tree for $N_i$ (in $G$)* is a tree $T_i$ for $N_i$ in $G$ with the minimum weight $w(T_i)$.

The so-called noncrossing paths may share common vertices or edges but do not cross each other in the plane. For example, paths $P_1$ and $P_2$ depicted in Fig. 2(a) cross each other on the plane, while paths $P_1$ and $P_2$ in Fig. 2(b) do not cross each other.

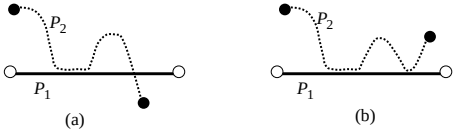



**Fig. 2.** (a) Crossing paths and (b) noncrossing paths.

We then define "crossing trees." Let $G^+$ be a plane graph obtained from $G$ as follows: add a new vertex $v_O$ in the outer face $f_1$, and join $v_O$ and every terminal on $B(f_1)$; similarly, add a new vertex $v_I$ in the inner face $f_2$, and join $v_I$ and every terminal on $B(f_2)$. Add to tree $T_i$ the following edges, those joining $v_O$ and terminals of $N_i$ on $B(f_1)$ and those joining $v_I$ and terminals of $N_i$ on

$B(f_2)$, and let $T_i^+$ be the resulting subgraph of $G^+$. We say that trees $T_i$ and $T_j$ do not *cross* each other if any pair of a path in $T_i^+$ and a path in $T_j^+$ do not cross each other. In Fig. 3, trees $T_1$ and $T_4$ do not cross each other. On the other hand, trees $T_2$ and $T_3$ cross each other since the path connecting $v_O$ and $u_{22}$ through $u_{21}$ in $T_2^+$ and the path connecting $u_{31}$ and $u_{33}$ through $u_{21}$ in $T_3^+$ cross each other. The definition above is appropriate for the VLSI single-layer routing problem.

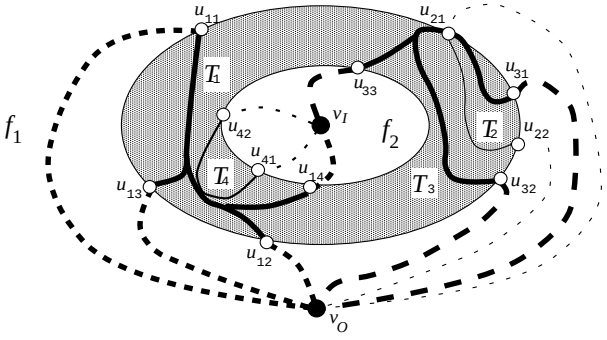



**Fig. 3.** A crossing forest.

A *forest* $\mathcal{T}$ *for* $\mathcal{N}$ *in* $G$ is defined to be a set $\{T_1, T_2, \cdots, T_k\}$ such that, for each $i$, $1 \leq i \leq k$, $T_i$ is a tree for net $N_i$ in $G$. We say that a forest $\mathcal{T}$ is *noncrossing* if every pair of trees in $\mathcal{T}$ do not cross each other. The forest in Fig. 1 is noncrossing, but the forest in Fig. 3 is crossing. The *weight* $w(\mathcal{T})$ of a forest $\mathcal{T}$ is $\sum_{i=1}^{k} w(T_i)$. A *noncrossing Steiner forest for* $\mathcal{N}$ *in* $G$ is a noncrossing forest for $\mathcal{N}$ in $G$ with the minimum total weight $w(\mathcal{T})$. Our goal is to find a noncrossing Steiner forest for $\mathcal{N}$ in a plane graph for the following two restricted cases: a case where all terminals lie on a single face boundary, and the other case where all terminals lie on two of the face boundaries.

## 3 One Face Problem

In this section, we present an efficient algorithm to solve the problem for the case where all terminals lie on a single face boundary $B(f_1)$ of a plane graph $G$. We call such a restricted problem *the one face problem.*

One can observe that the following lemma holds.

**Lemma 1** *There exists a noncrossing forest for* $\mathcal{N}$ *if and only if there are no four distinct terminals* $u_{ii'}, u_{jj'}, u_{ii''}$ *and* $u_{jj''}$ *appearing clockwise on* $B(f_1)$ *in this order, where* $u_{ii'}, u_{ii''} \in N_i \in \mathcal{N}$ *and* $u_{jj'}, u_{jj''} \in N_j \in \mathcal{N}$. □

Using Lemma 1, one can easily determine in linear time whether there is a noncrossing forest for $\mathcal{N}$. Thus, from now on, we may assume that there is a noncrossing forest for $\mathcal{N}$.

Our algorithm first finds $k$ cycles $C_1, C_2, \cdots, C_k$ in $G$, and then finds $k$ Steiner trees, one for each net $N_i \in \mathcal{N}$, in the inside of $C_i$. A cycle $C_i$, $1 \leq i \leq k$, for $N_i$ is defined as follows. Let $S_O = v_1, v_2, \cdots, v_q$ be the clockwise sequence of all vertices on $B(f_1)$ starting from $u_{11}$, where $v_1 = u_{11}$. One may assume that, for each nets $N_i \in \mathcal{N}$, the terminals $u_{i1}, u_{i2}, \cdots, u_{il_i}$ in $N_i$ appear in $S_O$ in this order, and that the first terminals $u_{11}, u_{21}, \cdots, u_{k1}$ in all nets appear in $S_O$ in this order, as illustrated in Fig. 4 for the case $k = 4$. For each $i$, $1 \leq i \leq k$, there

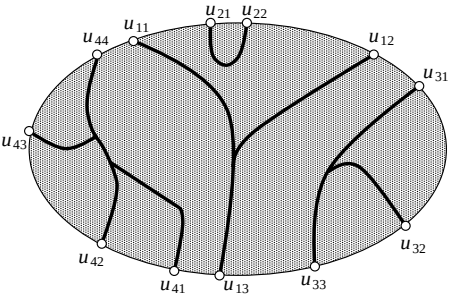



**Fig. 4.** Illustration of a noncrossing forest.

exists a set $\mathcal{P}_i$ of $l_i$ paths $P_1, P_2, \cdots, P_{l_i}$ such that each $P_{i'}$, $1 \leq i' \leq l_i$, is a shortest path connecting $u_{ii'}$ and $u_{i(i'+1)}$ in $G$, and any two paths in $\mathcal{P}_i$ do not cross each other, where $u_{i(l_i+1)} = u_{i1}$. The cycle $C_i$ obtained by concatenating all the $l_i$ paths in $\mathcal{P}_i$ is called *a cycle $C_i$ for $N_i$*. We denote by $G(C_i)$ the subgraphs of $G$ inside $Img(C_i)$. We say that cycles $C_i$ and $C_j$ do not *cross* each other if any pair of a path in $\mathcal{P}_i$ and a path in $\mathcal{P}_j$ do not cross each other. A set of *noncrossing cycles for $\mathcal{N}$* is defined to be a set $\mathcal{C} = \{C_1, C_2, \cdots, C_k\}$ such that each $C_i$ is a cycle for $N_i$ and any two cycles in $\mathcal{C}$ do not cross each other. Fig. 5(a) illustrates a set of noncrossing cycles. Then the following lemma holds.

**Lemma 2** *There exists a set $\mathcal{C}$ of noncrossing cycles for $\mathcal{N}$ if there exists a noncrossing forest $\mathcal{T}$ for $\mathcal{N}$.* □

Finding a set $\mathcal{C} = \{C_1, C_2, \cdots, C_k\}$ of noncrossing cycles would require time $\Omega(n^2)$, because vertices in $G$ may be shared by many distinct cycles and hence the sum $\sum_{i=1}^{k} |V(C_i)|$ is not always bounded by $O(n)$. Remember that $k$ is not always a fixed number but $k = O(n)$. Therefore, we find a subgraph $G_{\mathcal{C}} = \bigcup_{C_i \in \mathcal{C}} C_i$ of $G$ instead of a set $\mathcal{C}$ of noncrossing cycles. Clearly, $G_{\mathcal{C}}$ has at most $n$ vertices.

Similarly, we find a subgraph $G_{\mathcal{T}} = \bigcup_{T_i \in \mathcal{T}} T_i$ of $G$ instead of a noncrossing Steiner forest $\mathcal{T}$ for $\mathcal{N}$. From $G_{\mathcal{T}}$ one can find $T_i$ in time $O(|V(T_i)|)$ for each $i$, $1 \leq i \leq k$.

We are now ready to present an algorithm **Forest1** to solve the one face problem.

**Algorithm Forest1**
**step 1.** Find $G_{\mathcal{C}}$, where $\mathcal{C}$ is a set of noncrossing cycles for $\mathcal{N}$;
**step 2.** For each cycle $C_i \in \mathcal{C}$, find a Steiner tree $T_i$ for $N_i$ in the plane subgraph $G(C_i)$;
**step 3.** Construct a graph $G_{\mathcal{T}} = \bigcup_{i=1}^{k} T_i$.

We first consider how to execute step 1. For each $i$, $1 \leq i \leq k$, let $\mathcal{N}'_i$ be a set of $l_i$ two-terminals nets defined as follows: $\mathcal{N}'_i = \{\{u_{i1}, u_{i2}\}, \{u_{i2}, u_{i3}\}, \cdots, \{u_{il_i} u_{i1}\}\}$. Let $\mathcal{N}' = \bigcup_{i=1}^{k} \mathcal{N}'_i$. Then $\mathcal{N}'$ consists of $\sum_{i=1}^{k} l_i$ two-terminals nets. In Fig. 5(b), each two-terminals net in $\mathcal{N}'$ are drawn by a pair of a white point and a black point. Since all terminals in $\mathcal{N}'$ are on a single face boundary $B(f_1)$, using the

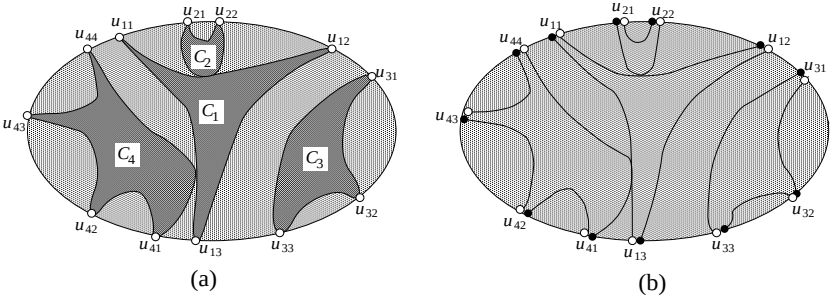



**Fig. 5.** Illustration for step 1.

algorithm in [9], one can find the minimum noncrossing paths for $\mathcal{N}'$ in time $O(n \log n)$. Furthermore, it is shown in [9] that each of the paths is a shortest path connecting the pair of terminals in a net. Therefore, for each $i$, $1 \leq i \leq k$, one can obtain a cycle $C_i$ by concatenating the $l_i$ paths for the nets in $\mathcal{N}'_i$. Thus, the minimum noncrossing paths for $\mathcal{N}'$ immediately yield the subgraph $G_{\mathcal{C}} = \bigcup_{i=1}^{k} C_i$.

We next consider how to execute steps 2 and 3. Let $m_i$ be the number of the edges in $G(C_i)$. Using the algorithm in [1,3], for each net $N_i \in \mathcal{N}$, one can find a Steiner tree $T_i$ for $N_i$ in time $O(m_i \log m_i)$ since $|N_i| = O(1)$ and all terminals in $N_i$ lies on the outer boundary of $G(C_i)$. Therefore, one can find a noncrossing Steiner forest $\mathcal{T}$ for $\mathcal{N}$ in time $O(\sum_{i=1}^{k} m_i \log m_i)$. Thus one can find $G_{\mathcal{T}}$ in time $O(n \log n)$ although the detail is omitted in this extended abstract.

The correctness of our algorithm is based on the following lemma, the proof of which is omitted in this extended abstract.

**Lemma 3** *For any set $\mathcal{C} = \{C_1, C_2, \cdots, C_k\}$ of noncrossing cycles, a Steiner tree $T_i$ for $N_i$ in $G(C_i)$ is a Steiner tree $T_i$ for $N_i$ in $G$ for each $i$, $1 \leq i \leq k$.* □

Thus we have the following Theorem 1.

**Theorem 1** *If all terminals lie on a single face boundary in a plane graph, then one can find a noncrossing Steiner forest $\mathcal{T}$ in time $O(n \log n)$.* □

## 4 Two Faces Problem

In this section, we give an algorithm to find a noncrossing Steiner forest for $\mathcal{N}$ for the case where all terminals lie on two of the face boundaries, $B(f_1)$ and $B(f_2)$. We call such a restricted problem *the two faces problem*.

We divide $\mathcal{N}$ into the following three subsets $\mathcal{N}_{out}$, $\mathcal{N}_{in}$ and $\mathcal{N}_{io}$:

**1.** for each net $N_i \in \mathcal{N}_{out}$, all terminals of $N_i$ lie on the outer face boundary $B(f_1)$;
**2.** for each net $N_i \in \mathcal{N}_{in}$, all terminals of $N_i$ lie on the inner face boundary $B(f_2)$; and
**3.** for each net $N_i \in \mathcal{N}_{io}$, $N_i$ contains a terminal on $B(f_1)$ and a terminal on $B(f_2)$.

Obviously, these three sets are disjoint and $\mathcal{N} = \mathcal{N}_{out} \cup \mathcal{N}_{in} \cup \mathcal{N}_{io}$. One may assume $\mathcal{N}_{io} \neq \phi$: otherwise, one can easily find a noncrossing Steiner forest $\mathcal{T}$ for $\mathcal{N}$; find a noncrossing Steiner forest $\mathcal{T}_{out}$ for $\mathcal{N}_{out}$ by executing **Forest1** for $\mathcal{N}_{out}$, all terminals of which are on $B(f_1)$; then find a noncrossing Steiner forest $\mathcal{T}_{in}$ for $\mathcal{N}_{in}$ by executing **Forest1** for $\mathcal{N}_{in}$, all terminals of which are on $B(f_2)$; and finally let $\mathcal{T} = \mathcal{T}_{out} \cup \mathcal{T}_{in}$. For the sake of simplicity, we assume $|\mathcal{N}_{io}| \geq 2$. Let $\mathcal{N}_{io} = \{N_1, N_2, \cdots, N_m\}$, where $m = |\mathcal{N}_{io}|$. For each net $N_i \in \mathcal{N}_{io}$, let $N_i^O = N_i \cap V(B(f_1))$, and $N_i^I = N_i \cap V(B(f_2))$. One may assume that $N_i^O = \{u_{i1}, u_{i2}, \cdots, u_{il_i'}\}$ and $N_i^I = \{u_{i(l_i'+1)}, u_{i(l_i'+2)}, \cdots, u_{i(l_i'+l_i'')}\}$, where $l_i = l_i' + l_i''$. One may assume that terminals $u_{11}$, $u_{12}$, $\cdots$, $u_{1l_1'}$, $u_{21}$, $u_{22}$, $\cdots$, $u_{2l_2'}$, $\cdots$, $u_{m1}$, $u_{m2}$, $\cdots$, $u_{ml_m'}$ appear clockwise on $B(f_1)$ in this order, and that $u_{1(l_1'+1)}$, $u_{1(l_1'+2)}$, $\cdots$, $u_{1l_1}$, $u_{2(l_2'+1)}$, $u_{2(l_2'+2)}$, $\cdots$, $u_{2l_2}$, $\cdots$, $u_{m(l_m'+1)}$, $u_{m(l_m'+2)}$, $\cdots$, $u_{ml_m}$ appear clockwise on $B(f_2)$ in this order, as illustrated in Fig. 6. Otherwise, there is no noncrossing forest. Let $s = u_{11}$, and let $t = u_{1(l_1'+1)}$.

The main idea of our algorithm is to reduce the two faces problem for $G$ to the one face problem for three graphs. For a path $P$ between $s$ and $t$, a *slit graph $SG(P)$ of $G$ for $P$* is generated from $G$ by slitting apart path $P$ into two paths $P_l$ and $P_r$, duplicating the vertices and edges of $P$ as follows (See Fig. 7). Each vertex $v$ in $P$ is replaced by new vertices $v_l$ and $v_r$. Each edge $(v, v')$ in $P$ is replaced by two parallel edges $(v_l, v_l')$ in $P_l$ and $(v_r, v_r')$ in $P_r$. Any edge $(v, w)$ that is not in $P$ but is incident to a vertex $v$ in $P$ is replaced by $(v_l, w)$ if $(v, w)$ is to the left of a path $P$ going from $s$ to $t$, and by $(v_r, w)$ if $(v, w)$ is to the right of the path. The operation above is called *slitting $G$ along $P$*. A graph and its slit graph are illustrated in Fig. 7. The net $N_1 \in \mathcal{N}_{io}$

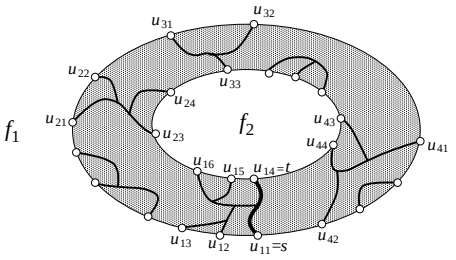



**Fig. 6.** A noncrossing forest.

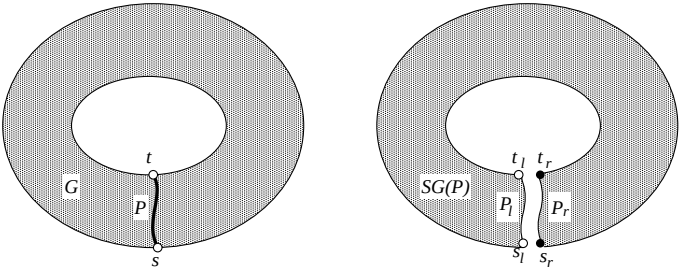



**Fig. 7.** Illustration for a graph $G$ and its slit graph $SG(P)$.

contains $s = u_{11}$ and $t = u_{1(l'_1+1)}$. We replace net $N_1 \in \mathcal{N}$ for $G$ by a new net $N'_1 = (N_1 - \{s, t\}) \cup \{s_l, t_l\}$ for $SG(P)$.

Let $P^*$ be any shortest path connecting $s$ and $t$ in $G$. We denote by $P^+$ a path in $G$ corresponding to a shortest path connecting $s_l$ and $t_r$ in $SG(P^*)$. (In Fig. 8(a) the path is drawn by a thick line.) Similarly, we denote by $P^-$ a path in $G$ corresponding to a shortest path connecting $s_r$ and $t_l$ in $SG(P^*)$. (In Fig. 8(a) the path is drawn by a dotted line.)

We are now ready to present an algorithm **Forest2** to solve the two faces problem.

**Algorithm Forest2**
**step 1.** Find a shortest path $P^*$ between $s$ and $t$ in $G$;
**step 2.** Construct a slit graph $SG(P^*)$ of $G$, and find $P^+$ and $P^-$; {Fig. 8(a)}
**step 3.** Construct slit graphs $SG(P^+)$ and $SG(P^-)$ of $G$; {Figs. 8(b) and (c)}
**step 4.** Using **Forest1**, find noncrossing Steiner forests $\mathcal{T}^*$ in $SG(P^*)$, $\mathcal{T}^+$ in $SG(P^+)$, and $\mathcal{T}^-$ in $SG(P^-)$;
step 5. Output one of these three forests having the minimum total weight.

One can find $P^*$ either in time $O(n \log n)$ by an ordinary Dijkstra algorithm or in time $O(n)$ by a sophisticated shortest path algorithm in [6,10]. Thus one can execute steps 1-3 in time $O(n \log n)$.

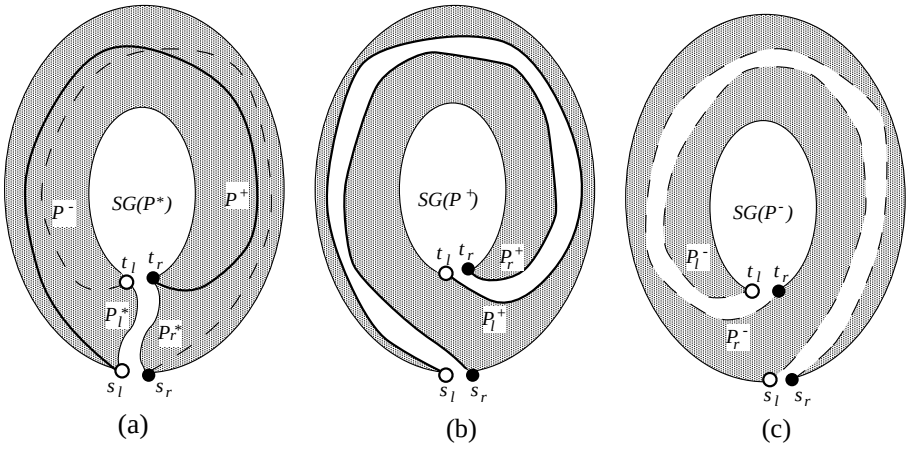



**Fig. 8.** Illustration for $SG(P^*), SG(P^+), SG(P^-)$.

Since all terminals in $SG(P^*)$ lie on the outer boundary of $SG(P^*)$, one can find $G_{\mathcal{T}^*}$ in $SG(P^*)$ in time $O(n \log n)$ using **Forest1**. Similarly, one can find $G_{\mathcal{T}^+}$ and $G_{\mathcal{T}^-}$ in time $O(n \log n)$. Thus, one can execute step 4 in time $O(n \log n)$.

Obviously, one can execute step 5 in a constant time.

The correctness of our algorithm is based on the following lemma, the proof of which is omitted in this extended abstract.

**Lemma 4** *There exists a noncrossing Steiner forest $\mathcal{T}$ such that either $P^*$,$P^+$ or $P^-$ crosses none of the trees in $\mathcal{T}$.* □

Thus we have the following Theorem 2.

**Theorem 2** *If all terminals lies on at most two face boundaries in a plane graph, then one can find a noncrossing Steiner forest in time $O(n \log n)$.* □

## 5 Conclusion

In this paper, we present an efficient algorithm for finding a noncrossing Steiner forest in a plane graph for the case all terminals are located on at most two of the face boundaries. Our algorithm takes $O(n \log n)$ time and $O(n)$ space if there exists a constant upper bound $L$ for the number of terminals in each net. It takes time $O(L^3 n + L^2 n \log n)$ if the maximum number $L$ of terminals in all nets is not always a fixed constant.

Slightly modifying our algorithm, one can find an "optimal" noncrossing forest. Let $\mathcal{T} = \{T_1, T_2, \cdots, T_k\}$ be a noncrossing forest for $\mathcal{N}$. Denote the weight

$w(T_i)$ of $T_i$ simply by $w_i$. Let $f(w_1, w_2, \cdots, w_k)$ be an arbitrary (objective) function which is nondecreasing with respect to each variable $w_i$. We call a noncrossing forest $\mathcal{T} = \{T_1, T_2, \cdots, T_k\}$ minimizing $f(w_1, w_2, \cdots, w_k)$ an *optimal noncrossing forest* (with respect to the objective function $f$).

**EXAMPLE 1.** The noncrossing Steiner forest is an optimal noncrossing forest minimizing the objective function $f = \sum_{i=1}^{k} w_i$. Clearly, $f$ is nondecreasing with respect to each $w_i$.

**EXAMPLE 2.** If the wires for all nets have the same width, then a noncrossing Steiner forest correspond to a routing minimizing the area required by wires. On the other hand, if the wires have various widths, say width $\alpha_i$ for net $N_i$, then the optimal noncrossing forest minimizing $f = \sum_i \alpha_i w_i$ corresponds to a routing minimizing the area. This function $f$ is also nondecreasing with respect to $w_i$ if all $\alpha_i$ are not negative.

One of future works is to obtain an algorithm to solve the noncrossing Steiner forest problem in a general case where terminals lie on three or more face boundaries.

## References

1. M. Bern: "Faster exact algorithm for Steiner trees in planar networks," Networks, 20, pp. 109-120 (1990).
2. S. E. Dreyfus and R. A. Wagner: "The Steiner problem in graphs," Networks, 1, pp. 195-208 (1972).
3. R. E. Erickson, C. L. Monma, and A. F. Veinott: "Send-and-split method for minimum-concave-cost network flows," Math. Oper. Res., 12, pp. 634-664 (1987).
4. M. R. Garey and D. S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman (1979).
5. M. R. Kramer and J. V. Leeuwen: "Wire-routing is NP-complete," RUU-CS-82-4, Department of Computer Science, University of Utrecht, Utrecht, the Netherlands (1982).
6. P. Klein, S. Rao, M. Rauch, and S. Subramanian: "Faster shortest-path algorithms for planar graphs," Proc. of 26th Annual Symp. on Theory of Computing, pp. 27-37 (1994).
7. T. Lengauer: Combinatorial Algorithms for Integrated Circuit Layout, John Wiley & Sons, Chichester, England (1990).
8. J. F. Lynch: "The equivalence of theorem proving and the interconnection problem," ACM SIGDA Newsletter, 5, 3, pp. 31-36 (1975).
9. J. Takahashi, H. Suzuki and T. Nishizeki: "Shortest noncrossing paths in plane graphs," Algorithmica, 16, pp. 339-357 (1996).
10. M. Thorup: "Undirected single source shortest paths in linear time," Proc. of 38th Annual Symp. on Foundations of Computer Science, pp. 12-21 (1997).
11. P. Winter: "Steiner problem in networks: A survey," Networks, 17, pp. 129-167 (1987).

# A Linear Algorithm for Finding Total Colorings of Partial $k$-Trees

Shuji Isobe*, Xiao Zhou**, and Takao Nishizeki***

Graduate School of Information Sciences, Tohoku University
Aoba-yama 05, Sendai, 980-8579, Japan.

**Abstract.** A total coloring of a graph $G$ is a coloring of all elements of $G$, i.e. vertices and edges, in such a way that no two adjacent or incident elements receive the same color. The total coloring problem is to find a total coloring of a given graph with the minimum number of colors. Many combinatorial problems can be efficiently solved for partial -trees, i.e., graphs with bounded tree-width. However, no efficient algorithm has been known for the total coloring problem on partial -trees although a polynomial-time algorithm of very high order has been known. In this paper, we give a linear-time algorithm for the total coloring problem on partial -trees with bounded .

## 1 Introduction

A total coloring is a mixture of ordinary vertex-coloring and edge-coloring. That is, a *total coloring* of a graph $G$ is an assignment of colors to its vertices and edges so that no two adjacent vertices have the same color, no two adjacent edges have the same color, and no edge has the same color as one of its ends. The minimum number of colors required for a total coloring of a graph $G$ is called the *total chromatic number* of $G$, and denoted by $\chi_t(G)$. Figure 1 illustrates a total coloring of a graph $G$ using $\chi_t(G) = 4$ colors. This paper deals with the *total coloring problem* which asks to find a total coloring of a given graph $G$ using the minimum number $\chi_t(G)$ of colors. Since the problem is NP-complete for general graphs [Sán89], it is very unlikely that there exists an efficient algorithm for solving the problem for general graphs. On the other hand, many combinatorial problems including the vertex-coloring problem and the edge-coloring problem can be solved for partial $k$-trees with bounded $k$ very efficiently, mostly in linear time [ACPS93,AL91,BPT92,Cou90,CM93,ZSN96]. However, no efficient algorithm has been known for the total coloring problem on partial $k$-trees. Although the total coloring problem can be solved in polynomial time for partial $k$-trees by a dynamic programming algorithm, the time complexity $O(n^{2^{4(k+1)}+1})$ is very high [IZN99].

* E-Mail: iso@nishizeki.ecei.tohoku.ac.jp
** E-Mail: zhou@ecei.tohoku.ac.jp
*** E-Mail: nishi@ecei.tohoku.ac.jp

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 347–356, 1999.
© Springer-Verlag Berlin Heidelberg 1999

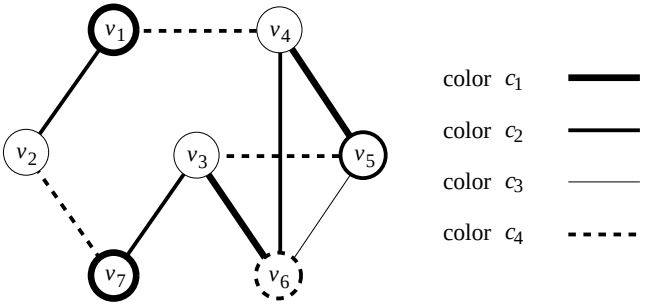



**Fig. 1.** A total coloring of a graph with four colors.

In this paper, we give a linear-time algorithm to solve the total coloring problem for partial $k$-trees with bounded $k$. The outline of the algorithm is as follows. For a given partial $k$-tree $G = (V, E)$, we first find an appropriate subset $F \subseteq E$ inducing a forest of $G$, then find a "generalized coloring" of $G$ for $F$ and an ordinary edge-coloring of the subgraph $H = G[\bar{F}]$ of $G$ induced by $\bar{F} = E - F$, and finally superimpose the edge-coloring on the generalized coloring to obtain a total coloring of $G$. The generalized coloring is an extended version of a total coloring and an ordinary vertex-coloring, and is newly introduced in this paper. Since $F$ induces a forest of $G$, a generalized coloring of $G$ for $F$ can be found in linear time. Since $H$ is a partial $k$-tree, an edge-coloring of $H$ can be found in linear time. Hence the total running time of our algorithm is linear. Thus our algorithm is completely different from an ordinary dynamic programming approach.

The paper is organized as follows. In Section 2, we give some basic definitions. In Section 3, we give a linear algorithm for finding a total coloring of a partial $k$-tree, and verify the correctness of the algorithm.

## 2 Terminologies and Definitions

In this section we give some basic terminologies and definitions.

For two sets $A$ and $B$, we denote by $A - B$ the set of elements $a$ such that $a \in A$ and $a \notin B$.

We denote by $G = (V, E)$ a simple undirected graph with a vertex set $V$ and an edge set $E$. For a graph $G = (V, E)$ we often write $V = V(G)$ and $E = E(G)$. We denote by $n$ the cardinality of $V(G)$. We denote by $\chi'(G)$ the minimum number of colors required for an ordinary edge-coloring of $G$, and call $\chi'(G)$ the *chromatic index of* $G$.

For a set $F \subseteq E$ and a vertex $v \in V$, we write $d_F(v, G) = |\{(v, w) \in F : w \in V\}|$ and $\Delta_F(G) = \max\{d_F(v, G) : v \in V\}$. In particular, we call $d(v, G) = d_E(v, G)$ the *degree* of $v$, and $\Delta(G) = \Delta_E(G)$ the *maximum degree* of $G$.

Let $F$ be a subset of $E$, called a *colored edge set*, and let $C$ be a set of colors. A *generalized coloring of a graph $G$ for $F$* is a mapping $f : V \cup F \to C$ satisfying the following three conditions:

(1) the restriction of the mapping $f$ to $V$ is a vertex-coloring of $G$, that is, $f(v) \neq f(w)$ for any pair of adjacent vertices $v$ and $w$ in $G$;
(2) the restriction of the mapping $f$ to $F$ is an edge-coloring of the subgraph $G[F]$ of $G$ induced by $F$, that is, $f(e) \neq f(e')$ for any pair of edges $e, e' \in F$ sharing a common end; and
(3) $f(v) \neq f(e)$ for any pair of a vertex $v \in V$ and an edge $e \in F$ incident to $v$.

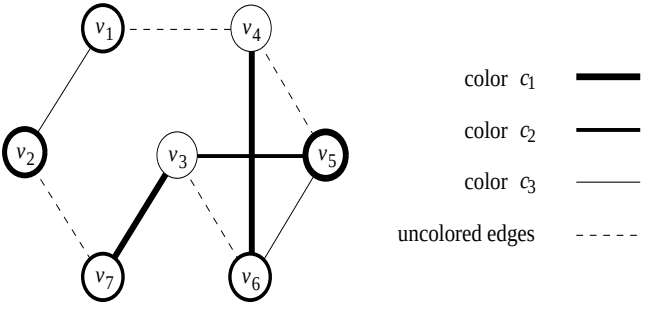



**Fig. 2.** A generalized coloring of a graph with three colors.

Note that the edges in $\bar{F} = E - F$ are not colored by the generalized coloring $f$. We call the edges in $F$ *colored edges* and the edges in $\bar{F}$ *uncolored edges*. A *total coloring of $G$* is a generalized coloring for a colored edge set $F = E$, while a vertex-coloring is a generalized coloring for a colored edge set $F = \emptyset$. Thus a generalized coloring is an extension of a total coloring and a vertex-coloring. It should be noted that a generalized coloring of $G$ for $F$ is a total coloring of $G[F]$ but a total coloring of $G[F]$ is not always a generalized coloring of $G$ for $F$. The minimum number of colors required for a generalized coloring of $G$ for $F$ is called the *generalized chromatic number of $G$*, and is denoted by $\chi_t(G, F)$. In particular, we denote $\chi_t(G, E)$ by $\chi_t(G)$, and call $\chi_t(G)$ the *total chromatic number of a graph $G$*. Clearly $\chi_t(G, F) \geq \Delta_F(G) + 1$ and $\chi_t(G) \geq \Delta(G) + 1$. Figure 2 depicts a generalized coloring of a graph $G$ using $\chi_t(G, F) = 3$ colors for the colored edge set $F = \{(v_1, v_2), (v_3, v_5), (v_3, v_7), (v_4, v_6), (v_5, v_6)\}$, where the uncolored edges $(v_1, v_4)$, $(v_2, v_7)$, $(v_3, v_6)$ and $(v_4, v_5)$ in $\bar{F}$ are drawn by dotted lines.

Suppose that $g$ is a generalized coloring of $G$ for $F$, $h$ is an ordinary edge-coloring of the subgraph $H = G[\bar{F}]$ of $G$ induced by $\bar{F}$, and $g$ and $h$ use disjoint sets of colors. Then, superimposing $h$ on $g$, one can obtain a total coloring $f$ of $G$. Unfortunately, the total coloring $f$ obtained in this way may use more than $\chi_t(G)$ colors even if $g$ uses $\chi_t(G, F)$ colors and $h$ uses $\chi'(H)$ colors, because $\chi_t(G) \leq \chi_t(G, F) + \chi'(H)$ but the equality does not always hold; for example, $\chi_t(G) = 4$, $\chi_t(G, F) = 3$ and $\chi'(H) = 2$, and hence $\chi_t(G) < \chi_t(G, F) + \chi'(H)$ for

the graph $G$ in Figure 2. However, in Section 3, we will show that, for a partial $k$-tree $G = (V, E)$ with the large maximum degree, there indeed exists $F \subseteq E$ such that $\chi_t(G) = \chi_t(G, F) + \chi'(H)$, and show that such a set $F$, a generalized coloring of $G$ for $F$ and an edge-coloring of $H$ can be found in linear time.

A graph $G = (V, E)$ is defined to be a *$k$-tree* if either it is a complete graph of $k$ vertices or it has a vertex $v \in V$ 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 [Bod90]. In the paper we assume that $k = O(1)$. The graph in Figure 1 is a partial 3-tree.

For a natural number $s$, an *$s$-numbering* of a graph $G = (V, E)$ is a bijection $\varphi : V \to \{1, 2, \cdots, n\}$ such that $|\{(v, x) \in E : \varphi(v) < \varphi(x)\}| \leq s$ for each vertex $v \in V$. A graph having an $s$-numbering is called an *$s$-degenerated graph*. Every partial $k$-tree $G$ is a $k$-degenerated graph, and its $k$-numbering can be found in linear time.

For an $s$-numbering $\varphi$ of $G$ and a vertex $v \in V$, we define

$$
\begin{aligned}
&E_\varphi^{\mathrm{fw}}(v, G) = \{(v, x) \in E : \varphi(v) < \varphi(x)\};\\
&E_\varphi^{\mathrm{bw}}(v, G) = \{(x, v) \in E : \varphi(x) < \varphi(v)\};\\
&d_\varphi^{\mathrm{fw}}(v, G) = |E_\varphi^{\mathrm{fw}}(v, G)|; \text{ and}\\
&d_\varphi^{\mathrm{bw}}(v, G) = |E_\varphi^{\mathrm{bw}}(v, G)|.
\end{aligned}
$$

The edges in $E_\varphi^{\mathrm{fw}}$ are called *forward edges*, and those in $E_\varphi^{\mathrm{bw}}$ *backward edges*. The definition of an $s$-numbering implies that $d_\varphi^{\mathrm{fw}}(v, G) \leq s$ for each vertex $v \in V$.

## 3 A Linear Algorithm

In this section we prove the following main theorem.

**Theorem 1.** *Let $G = (V, E)$ be a partial $k$-tree with bounded $k$. Then there exists an algorithm to find a total coloring of $G$ with the minimum number $\chi_t(G)$ of colors in linear time.*

We first have the following lemma [ZNN96,IZN99].

**Lemma 1.** *For any $s$-degenerated graph $G$, the following* (a) *and* (b) *hold*:
(a) *if $\Delta(G) \geq 2s$, then $\chi'(G) = \Delta(G)$*; and
(b) *$\chi_t(G) \leq \Delta(G) + s + 2$.*

Using a standard dynamic programming algorithm in [IZN99], one can solve the total coloring problem for a partial $k$-tree $G$ in time $O(n\chi_t^{2^{4(k+1)}})$ where $\chi_t = \chi_t(G)$; the size of a dynamic programming table updated by the algorithm is $O(\chi_t^{2^{4(k+1)}})$. Since $G$ is a partial $k$-tree, $G$ is $k$-degenerated. Furthermore $k = O(1)$. Therefore, if $\Delta(G) = O(1)$, then $\chi_t(G) = O(1)$ by Lemma 1(b) and hence the algorithm takes linear time to solve the total coloring problem. Thus it suffices to give an algorithm for the case $\Delta(G)$ is large, say $\Delta(G) \geq 8k^2$.

Our idea is to find a subset $F$ of $E$ such that $\chi_t(G) = \chi_t(G, F) + \chi'(H)$ as described in the following lemma.

**Lemma 2.** *Assume that $G = (V, E)$ is an $s$-degenerated graph and has an $s$-numbering $\varphi$. If $\Delta(G) \geq 8s^2$, then there exists a subset $F$ of $E$ satisfying the following conditions* (a)–(h):

(a) $\Delta(G) = \Delta_F(G) + \Delta_{\bar{F}}(G)$, *where* $\bar{F} = E - F$;
(b) $\Delta_F(G) \geq s + 1$;
(c) $\Delta_{\bar{F}}(G) \geq 2s$;
(d) *the set $F$ can be found in linear time*;
(e) *$\varphi$ is a 1-numbering of $G' = (V, F)$, and hence $G'$ is a forest* ;
(f) $\chi_t(G, F) = \Delta_F(G) + 1$, *and a generalized coloring of $G$ for $F$ using $\Delta_F(G) + 1$ colors can be found in linear time*;
(g) $\chi'(H) = \Delta_{\bar{F}}(G)$, *where* $H = (V, \bar{F})$; and
(h) $\chi_t(G) = \chi_t(G, F) + \chi'(H)$.

*Proof.* The proofs of (a)–(e) will be given later. We now prove only (f)–(h).

(f) Let $C$ be a set of $\Delta_F(G) + 1$ colors. For each $i = 1, 2, \cdots, n$, let $v_i$ be a vertex of $G$ such that $\varphi(v_i) = i$, let $N^{\mathrm{fw}}(v_i) = \{x \in V : (v_i, x) \in E,\ \varphi(v_i) < \varphi(x)\}$, and let $E_F^{\mathrm{fw}}(v_i) = \{(v_i, x) \in F : \varphi(v_i) < \varphi(x)\}$. Since $\varphi$ is an $s$-numbering of $G$, $d_\varphi^{\mathrm{fw}}(v_i, G) = |N^{\mathrm{fw}}(v_i)| \leq s$ for each $i = 1, 2, \cdots, n$. By (e) $\varphi$ is a 1-numbering of $G' = (V, F)$, and hence $d_\varphi^{\mathrm{fw}}(v_i, G') = |E_F^{\mathrm{fw}}(v_i)| \leq 1$ for each $i = 1, 2, \cdots, n$.

We construct a generalized coloring $g$ of $G$ for $F$ using colors in $C$ as follows. We first color $v_n$ by any color $c$ in $C$: let $g(v_n) := c$. Suppose that we have colored the vertices $v_n, v_{n-1}, \cdots, v_{i+1}$ and the edges in $E_F^{\mathrm{fw}}(v_{n-1}) \cup E_F^{\mathrm{fw}}(v_{n-2}) \cup \cdots \cup E_F^{\mathrm{fw}}(v_{i+1})$, and that we are now going to color $v_i$ and the edge in $E_F^{\mathrm{fw}}(v_i)$ if $E_F^{\mathrm{fw}}(v_i) \neq \emptyset$. There are two cases to consider.

**Case 1**: $E_F^{\mathrm{fw}}(v_i) \neq \emptyset$.

In this case $E_F^{\mathrm{fw}}(v_i)$ contains exactly one edge $e = (v_i, v_j)$, where $i < j \leq n$.

We first color $e$. Let $C' = \{g((v_j, v_l)) : (v_j, v_l) \in F,\ i + 1 \leq l \leq n\} \subseteq C$, then we must assign to $e$ a color not in $\{g(v_j)\} \cup C'$. Since $e = (v_j, v_i) \in F$, we have

$$|\{(v_j, v_i)\} \cup \{(v_j, v_l) \in F : i + 1 \leq l \leq n\}| \leq d(v_j, G')$$

and hence $|C'| \leq d(v_j, G') - 1$. Clearly $d(v_j, G') \leq \Delta_F(G) = |C| - 1$. Therefore we have $|C'| \leq |C| - 2$. Thus there exists a color $c' \in C$ not in $\{g(v_j)\} \cup C'$. We color $e$ by $c'$: let $g(e) := c'$.

We next color $v_i$. Let $C'' = \{g(x) : x \in N^{\mathrm{fw}}(v_i)\}$, then we must assign to $v_i$ a color not in $\{c'\} \cup C''$. Since $|C''| \leq |N^{\mathrm{fw}}(v_i)| \leq s$ and $\Delta_F(G) \geq s + 1$ by (b) above, we have $|\{c'\} \cup C''| \leq s + 1 \leq \Delta_F(G) = |C| - 1$. Thus there exists a color $c'' \in C$ not in $\{c'\} \cup C''$, and we can color $v_i$ by $c''$: let $g(v_i) := c''$.

**Case 2**: $E_F^{\mathrm{fw}}(v_i) = \emptyset$.

In this case we need to color only $v_i$. Similarly as above, there exists a color $c'' \in C$ not in $C''$ since $|C''| \leq s < \Delta_F(G) < |C|$. Therefore we can color $v_i$ by $c''$: let $g(v_i) := c''$.

Thus we have colored $v_i$ and the edge in $E_F^{\mathrm{fw}}(v_i)$ if $E_F^{\mathrm{fw}}(v_i) \neq \emptyset$. Repeating the operation above for $i = n-1, n-2, \cdots, 1$, we can construct a generalized coloring $g$ of $G$ for $F$ using colors in $C$. Hence $\chi_t(G,F) \leq |C| = \Delta_F(G) + 1$. Clearly $\chi_t(G,F) \geq \Delta_F(G) + 1$, and hence we have $\chi_t(G,F) = \Delta_F(G) + 1$. Clearly the construction of $g$ above takes linear time. Thus we have proved (f).

(g) Since $G$ is $s$-degenerated, the subgraph $H$ of $G$ is $s$-degenerated. By (c) we have $\Delta(H) = \Delta_{\bar{F}}(G) \geq 2s$. Therefore by Lemma 1(a) we have $\chi'(H) = \Delta(H) = \Delta_{\bar{F}}(G)$. Thus we have proved (g).

(h) We can obtain a total coloring of $G$ by superimposing an edge-coloring of $H$ on a generalized coloring of $G$ for $F$. Therefore we have $\chi_t(G) \leq \chi_t(G,F) + \chi'(H)$. Since $\chi_t(G) \geq \Delta(G) + 1$, by (a), (f) and (g) we have

$$\begin{aligned}\chi_t(G) &\geq \Delta(G) + 1 \\ &= \Delta_F(G) + \Delta_{\bar{F}}(G) + 1 \\ &= \chi_t(G,F) + \chi'(H).\end{aligned}$$

Thus we have $\chi_t(G) = \chi_t(G,F) + \chi'(H)$.

We now have the following theorem.

**Theorem 2.** *If $G$ is an $s$-degenerated graph and $\Delta(G) \geq 8s^2$, then $\chi_t(G) = \Delta(G) + 1$.*

*Proof.* By (a), (f), (g) and (h) in Lemma 2 we have

$$\begin{aligned}\chi_t(G) &= \chi_t(G,F) + \chi'(H) \\ &= \Delta_F(G) + 1 + \Delta_{\bar{F}}(G) \\ &= \Delta(G) + 1.\end{aligned}$$

We are now ready to present our algorithm to find a total coloring of a given partial $k$-tree $G = (V, E)$ with $\Delta(G) \geq 8k^2$.

[**Total-Coloring Algorithm**]

**Step 1**. Find a subset $F \subseteq E$ satisfying Conditions (a)–(h) in Lemma 2.

**Step 2**. Find a generalized coloring $g$ of $G$ for $F$ using $\chi_t(G,F) = \Delta_F(G) + 1$ colors.

**Step 3**. Find an ordinary edge-coloring $h$ of $H$ using $\chi'(H) = \Delta_{\bar{F}}(G)$ colors.

**Step 4**. Superimpose the edge-coloring $h$ on the generalized coloring $g$ to obtain a total coloring $f$ of $G$ using $\chi_t(G) = \Delta(G) + 1$ colors.

Since $G$ is a partial $k$-tree, $G$ is $k$-degenerated. Since $\Delta(G) \geq 8k^2$, by Lemma 2(d) one can find the subset $F \subseteq E$ in Step 1 in linear time. By Lemma 2(f) one can find the generalized coloring $g$ in Step 2 in linear time. Since $G$ is a partial $k$-tree, a subgraph $H$ of $G$ is also a partial $k$-tree. Therefore, in Step 3 one can find the edge-coloring $h$ of $H$ in linear time by an algorithm in [ZNN96] although $\chi'(H)$ is not always bounded. Thus the algorithm runs in linear time.

This completes the proof of Theorem 1.

In the remainder of this section we prove (a)–(e) of Lemma 2. We need the following two lemmas.

**Lemma 3.** *Let $G$=($V$,$E$) be an $s$-degenerated graph, and let $\varphi$ be an $s$-numbering of $G$. If $\Delta(G)$ is even, then there exists a subset $E'$ of $E$ satisfying the following three conditions* (a)–(c):

(a) $\Delta(G') = \Delta(G'') = \Delta(G)/2$;
(b) *$\varphi$ is an $\lceil s/2 \rceil$-numbering of $G'$*; and
(c) $|E'| \leq |E|/2$,

*where $G' = (V, E')$ and $G'' = (V, E - E')$. Furthermore, such a set $E'$ can be found in linear time.*

*Proof.* Omitted in this extended abstract due to the page limitation.

**Lemma 4.** *Let $G = (V, E)$ be an $s$-degenerated graph, and let $\alpha$ be a natural number. If $\Delta(G) \geq 2\alpha \geq 2s$, then there exists a subset $E'$ of $E$ such that $\Delta_{E'}(G) + \Delta_{\bar{E}'}(G) = \Delta(G)$ and $\Delta_{E'}(G) = \alpha$ where $\bar{E}' = E - E'$. Furthermore, such a set $E'$ can be found in linear time.*

*Proof.* Since $G$ is an $s$-degenerated graph and $\Delta(G) \geq 2\alpha \geq 2s$, there exists a partition $\{E_1, E_2, \cdots, E_l\}$ of $E$ satisfying the following three conditions (i)–(iii) [ZNN96, pp. 610]:

(i) $\sum_{i=1}^{l} \Delta_{E_i}(G) = \Delta(G)$;
(ii) $\Delta_{E_i}(G) = \alpha$ for each $i = 1, 2, \cdots, l-1$; and
(iii) $\alpha \leq \Delta_{E_l}(G) < 2\alpha$.

Let $E' = E_1$. Then $\Delta_{E'}(G) = \Delta_{E_1}(G) = \alpha$ and by (ii), clearly

$$\Delta_{\bar{E}'}(G) \geq \Delta(G) - \Delta_{E'}(G). \tag{1}$$

On the other hand, since $\bar{E}' = E_2 \cup E_3 \cup \cdots \cup E_l$, by (i) we have

$$\begin{aligned} \Delta_{\bar{E}'}(G) &\leq \sum_{i=2}^{l} \Delta_{E_i}(G) \\ &= \Delta(G) - \Delta_{E_1}(G) \\ &= \Delta(G) - \Delta_{E'}(G). \end{aligned} \tag{2}$$

Thus by Eqs. (1) and (2) we have $\Delta_{\bar{E}'}(G) = \Delta(G) - \Delta_{E'}(G)$, and hence $\Delta_{E'}(G) + \Delta_{\bar{E}'}(G) = \Delta(G)$. Since the partition $\{E_1, E_2, \cdots, E_l\}$ of $E$ can be found in linear time [ZNN96], the set $E' = E_1$ can be found in linear time.

We are now ready to give the remaining proof of Lemma 2.

**Remaining Proof of Lemma 2**: Since we have already proved (f)–(h) before, we now prove (a)–(e).

Let $p = \lfloor \log \Delta(G) \rfloor$. Then $2^p \leq \Delta(G) < 2^{p+1}$. Since $\Delta(G) \geq 8s^2$, we have $p = \lfloor \log \Delta(G) \rfloor \geq 3 + \lfloor 2 \log s \rfloor > 2 + 2 \log s$. Therefore we have $\Delta(G) \geq 2^p > 2^{2+2\log s} = 4s^2 > 2s$.

Let $q = \lceil \log s \rceil$. Then $2^{q-1} < s \leq 2^q$. We find $F$ by constructing a sequence of $q+1$ spanning subgraphs $G_0, G_1, \cdots, G_q$ of $G$ as follows.

1 **procedure FIND-F**
2 **begin**
3 by Lemma 4, find a subset $E_0$ of $E$ such that
(3-1) $\Delta_{E_0}(G) = 2^{p-1}$; and
(3-2) $\Delta_{E_0}(G) + \Delta_{\bar{E}_0}(G) = \Delta(G)$, where $\bar{E}_0 = E - E_0$;
{Choose $\alpha = 2^{p-1}$, then $\Delta(G) \geq 2^p = 2\alpha \geq 2s$, and hence there exists such a set $E_0$ by Lemma 4}
4 let $G_0 := (V, E_0)$ and let $s_0 := s$;
{$\Delta(G_0) = 2^{p-1}$ and $\varphi$ is an $s_0$-numbering of $G_0$}
5 **for** $i := 0$ to $q-1$ **do**
6 **begin**
7 by Lemma 3, find a subset $E'_i$ of $E_i$ satisfying
(7-1) $\Delta(G'_i) = \Delta(G''_i) = \Delta(G_i)/2$; {$\Delta(G_i)$ is even}
(7-2) $\varphi$ is an $s_{i+1}$-numbering of $G'_i$, where $s_{i+1} = \lceil s_i/2 \rceil$; and
(7-3) $|E'_i| \leq |E_i|/2$,
where $G'_i = (V, E'_i)$, $G''_i = (V, E''_i)$ and $E''_i = E_i - E'_i$;
8 let $E_{i+1} := E'_i$ and let $G_{i+1} := (V, E_{i+1})$;
9 **end**
10 let $F := E_q$;
11 **end**.

We first prove (a). Since $F = E_q$, $\Delta_F(G) = \Delta_{E_q}(G) = \Delta(G_q)$. Therefore we have

$$\Delta_{\bar{F}}(G) \geq \Delta(G) - \Delta_F(G) = \Delta(G) - \Delta(G_q). \tag{3}$$

By line 7 and line 8 in the procedure above, we have $G_{i+1} = G'_i$, $\Delta(G_{i+1}) = \Delta(G'_i)$ and $\Delta(G'_i) + \Delta(G''_i) = \Delta(G_i)$, and hence $\Delta(G''_i) = \Delta(G_i) - \Delta(G_{i+1})$ for each $i = 0, 1, \cdots, q-1$. Therefore we have

$$\begin{aligned} \sum_{i=0}^{q-1} \Delta(G''_i) &= \sum_{i=0}^{q-1} (\Delta(G_i) - \Delta(G_{i+1})) \\ &= \Delta(G_0) - \Delta(G_q). \end{aligned} \tag{4}$$

Since $\Delta(G_0) = \Delta_{E_0}(G)$, by (3-2) in the procedure above we have

$$\Delta(G_0) + \Delta_{\bar{E}_0}(G) = \Delta(G). \tag{5}$$

Furthermore, since $\bar{F} = E - F = \bar{E}_0 \cup E''_0 \cup E''_1 \cup \cdots \cup E''_{q-1}$ and $\Delta_{E''_i}(G) = \Delta(G''_i)$ for each $i = 0, 1, \cdots, q-1$, by Eqs. (4) and (5) we have

$$\begin{aligned}
\Delta_{\bar{F}}(G) &\leq \Delta_{\bar{E}_0}(G) + \sum_{i=0}^{q-1} \Delta_{E''_i}(G) \\
&= \Delta_{\bar{E}_0}(G) + \sum_{i=0}^{q-1} \Delta(G''_i) \\
&= \Delta_{\bar{E}_0}(G) + \Delta(G_0) - \Delta(G_q) \\
&= \Delta(G) - \Delta(G_q). \qquad (6)
\end{aligned}$$

Therefore by Eqs. (3) and (6) we have $\Delta_{\bar{F}}(G) = \Delta(G) - \Delta(G_q)$. Since $\Delta(G_q) = \Delta_F(G)$, we have $\Delta_{\bar{F}}(G) = \Delta(G) - \Delta_F(G)$, and hence $\Delta_F(G) + \Delta_{\bar{F}}(G) = \Delta(G)$. Thus we have proved (a).

We next prove (b). By (7-1) and line 8 in the procedure above we have $\Delta(G_{i+1}) = \Delta(G'_i) = \Delta(G_i)/2$ for each $i = 0, 1, \cdots, q-1$, and by (3-1) we have $\Delta(G_0) = 2^{p-1}$. Therefore we have

$$\Delta_F(G) = \Delta(G_q) = \frac{\Delta(G_0)}{2^q} = 2^{p-q-1}. \qquad (7)$$

Since $2^p > 4s^2$ and $2^{q-1} < s$, we have $\Delta_F(G) = 2^{p-q-1} > 4s^2/4s = s$. Thus we have $\Delta_F(G) \geq s + 1$, and hence (b) holds.

We next prove (c). By (a) and Eq. (7) we have $\Delta_{\bar{F}}(G) = \Delta(G) - \Delta_F(G) = \Delta(G) - 2^{p-q-1}$, and hence

$$\begin{aligned}
\Delta_{\bar{F}}(G) &= \Delta(G) - \frac{2^p}{2^{q+1}} \\
&\geq \Delta(G) - \frac{\Delta(G)}{2^{q+1}} \\
&= \Delta(G)(1 - \frac{1}{2^{q+1}}) \\
&\geq 8s^2(1 - \frac{1}{2s}) \\
&= 4s(2s - 1) \\
&\geq 2s
\end{aligned}$$

since $\Delta(G) \geq 8s^2$, $\Delta(G) \geq 2^p$ and $s \leq 2^q$. Thus we have proved (c).

We next prove (d). By Lemma 4, line 3 can be done in time $O(|E|)$. By Lemma 3, line 7 can be done in time $O(|E_i|)$. Therefore the **for** statement in line 5 can be done in time $O(\sum_{i=0}^{q-1} |E_i|)$ time. Since $|E_0| \leq |E|$ and $|E_{i+1}| = |E'_i| \leq |E_i|/2$ for each $i = 0, 1, \cdots, q-1$ by (7-3) in the procedure above, we have $\sum_{i=0}^{q-1} |E_i| \leq 2|E|$. Thus one can know that the **for** statement can be done in time $O(|E|)$. Thus $F$ can be found in linear time, and hence (d) holds.

We finally prove (e). Since $s = s_0 \leq 2^q$ and $s_i = \lceil s_{i-1}/2 \rceil \leq s_{i-1}/2 + 1/2$ for each $i = 1, 2, \cdots, q$, we have

$$\begin{aligned} s_q &\leq \frac{s_0}{2^q} + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^q} \\ &= \frac{s_0}{2^q} + 1 - \frac{1}{2^q} \\ &< 2, \end{aligned}$$

and hence $s_q = 1$. Therefore $\varphi$ is a 1-numbering of $G' = (V, F)$. Thus we have proved (e).

This completes the proof of Lemma 2.

## References


ACPS93. S. Arnborg, B. Courcelle, A. Proskurowski and D. Seese, An algebraic theory of graph reduction, J. Assoc. Comput. Mach. 40(5), pp. 1134–1164, 1993.

AL91. S. Arnborg and J. Lagergren, Easy problems for tree-decomposable graphs, J. Algorithms, 12(2), pp. 308–340, 1991.

Bod90. H.L. Bodlaender, Polynomial algorithms for graph isomorphism and chromatic index on partial -trees, Journal of Algorithms, 11(4), pp. 631–643, 1990.

BPT92. 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, pp. 555–581, 1992.

Cou90. B. Courcelle, The monadic second-order logic of grpahs I: Recognizable sets of finite graphs, Inform. Comput., 85, pp. 12–75, 1990.

CM93. B. Courcelle and M. Mosbath, Monadic second-order evaluations on tree-decomposable graphs, Theoret. Comput. Sci., 109, pp.49–82, 1993.

IZN99. S. Isobe, X. Zhou and T. Nishizeki, A polynomial-time algorithm for finding total colorings of partial -trees, Int. J. Found. Comput. Sci., 10(2), pp. 171–194, 1999.

Sán89. A. Sánchez-Arroyo. Determining the total colouring number is NP-hard, Discrete Math., 78, pp. 315–319, 1989.

ZNN96. X. Zhou, S. Nakano and T. Nishizeki, Edge-coloring partial -trees, J. Algorithms, 21, pp. 598–617, 1996.

ZSN96. X. Zhou, H. Suzuki and T. Nishizeki, A linear algorithm for edge-coloring series-parallel multigraphs, J. Algorithm, 20, pp. 174-201, 1996.

# Topology-Oriented Approach to Robust Geometric Computation

Kokichi Sugihara

Department of Mathematical Engineering and Information Physics
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan
sugihara@simplex.t.u-tokyo.ac.jp

**Abstract.** The topology-oriented approach is a principle for translating geometric algorithms into practically valid computer software. In this principle, the highest priority is placed on the topological consistency of the geometric objects; numerical values are used as lower-priority information. The resulting software is completely robust in the sense that no matter how large numerical errors arise, the algorithm never fail. The basic idea of this approach and various examples are surveyed.

## 1 Introduction

There is a great gap between theoretically correct geometric algorithms and practically valid computer programs. This is mainly because the correctness of the algorithms is based on the assumption that numerical computation is done in infinite precision, whereas in actual computers computation is done only in finite precision. Numerical errors often generate topological inconsistency, and thus make the computer programs to fail.

To overcome this difficulty, many approaches have been proposed. They can be classified according to how much they rely on numerical values.

The first group of approaches relies on numerical values absolutely. The topological structure of a geometric object is determined by the signs of the results of numerical computations. If we restrict the precision of the input data, these signs can be judged correctly in a sufficiently high but still finite precision. Using this principle, the topological structures are judged correctly as if the computation is done exactly. These approaches are called exact-arithmetic approaches [1,3,9,14,15,17,21,23,26]. In these approaches, we need not worry about misjudgement and hence theoretical algorithms can be implemented in a rather straightforward manner. However, the computation is expensive because multiple precision is required, and furthermore exceptional branches of processing for degenerate cases are necessary because all the degenerate cases are recognized exactly.

The second group of approaches relies on numerical values moderately. Every time numerical computation is done, the upper bound of the error is also evaluated. On the basis of this error bound, the result of computation is judged to be either reliable or unreliable, and the only reliable result is used [2,4,6,10,16]. However, these approaches make program codes unnecessarily complicated because

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 357–366, 1999.
© Springer-Verlag Berlin Heidelberg 1999

every numerical computation should be followed by two alternative branches of processing, one for the reliable case and the other for the unreliable case. Moreover, they decrease the portability of the software products, because the amount of errors depends on computation environment.

The third group of approaches do not rely on numerical values at all. In this group of approaches, we start with the assumption that every numerical computation contains errors and that the amount of the error is unknown. We place the highest priority on the consistency of topological properties, and use numerical results only when they are consistent with the topological properties, thus avoiding inconsistency [7,8,11,12,13,19,20,24,25]. These approaches are called topology-oriented approaches.

In this paper, we concentrate on the third group of approaches, and survey its basic idea and various applications.

## 2 Basic Idea

Let $A$ be an algorithm for solving geometric problem $P$. In general the algorithm $A$ contains many crotches at which the flow of the procedure diverges to two or more alternative branches. Each path from the starting point in this branching structure corresponds to a behavior of the algorithm $A$.

Once we fix an instance of the problem, all of such paths can be classified into three groups. The first group of paths leads to the correct solution of the problem instance; in many algorithms this group contains only a single path. The second group of paths leads to the solutions of other instances belonging to the problem $P$. The third group of paths corresponds to inconsistency. Note that even though the algorithm $A$ is correct, there are the second and the third groups of paths. The correctness of the algorithm is guaranteed only in the sense that these groups of paths will never be selected in precise arithmetic.

Let us consider a simple example. Let $\mathrm{P}_1$, $\mathrm{P}_2$, $\mathrm{P}_3$, $\mathrm{P}_4$ be a cyclic sequence of vertices forming a convex quadrangle in the three-dimensional space, and $H$ be a non-vertical plane that does not contain any of these vertices. Suppose that we want to cut the quadrangle by the plane $H$. How the quadrangle is cut depends on which side of $H$ each vertex lies. Hence, a natural algorithm to solve this problem will have two-branch crotches, each of which corresponds to whether the vertex $\mathrm{P}_i$ is above $H$ or below $H$, Hence, the flow of the procedure forms a binary-tree structure as shown in Fig. 1, where each non-leaf node corresponds to the judgement on whether $\mathrm{P}_i$ is above $H$, and the left branch and the right branch correspond to the affirmative case and the negative case, respectively.

As seen in Fig. 1, the procedure contains $2^4 = 16$ paths from the root to the leaves. Among them, the two paths represented by bold lines correspond to inconsistent cases; in these cases all the adjacent vertices are on mutually opposite sides of $H$, which never happens in Euclidean geometry because two distinct planes in the three-dimensional space can admit at most one line of intersection. Thus we get the next requirement.

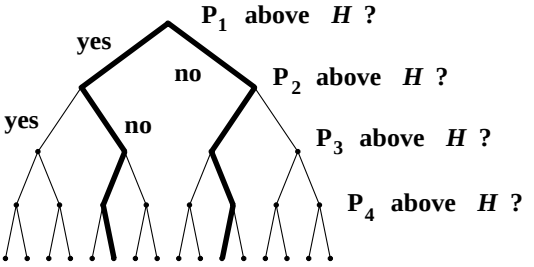



**Fig. 1.** Tree structure of the flow of the procedure.

**TR 2.1.** If mutually diagonal two vertices belong to the same side of $H$, at least one of the other vertices should also belong to the same side.

TR is an abbreviation of "Topological Requirement". We call the above requirement a "topological requirement" because it can be stated in purely topological (i.e., nonmetric) terms.

Geometric algorithms usually contain inconsistent paths, just as the bold paths in Fig. 1. Nevertheless, these algorithms are theoretically correct because in the theoretical world numerical judgement is always correct and hence will never choose such an inconsistent path. In computers, on the other hand, branches are chosen on the basis of finite-precision computation and hence inconsistent paths are sometimes chosen.

The left bold path in Fig. 1 says that $P_1$ is above $H$, $P_2$ is below $H$ and $P_3$ is above $H$. Then, the topological requirement TR 2.1 implies that $P_4$ should be above $H$. In this sense the fourth judgement is redundant. Such redundant judgements cause inconsistent branching.

The tree structure shown in Fig. 2 is obtained from Fig. 1 by removing such redundant judgements. In this structure all the paths belong to either the first group or the second group, and hence, no matter how poor precision is used in computation, the algorithm never come across inconsistency.

In the topology-oriented approach, we first collect topological requirements that should be satisfied by the geometric objects, next find redundant judgements, and finally remove them from the tree structure of the flow of processing, just as we rewrite the structure in Fig. 1 to that in Fig. 2. What we want to emphasize here is that the topological requirements can be stated in a purely topological terms, which means that these requirements can be checked by logical (non-numerical) computation only, and hence can be done always correctly. The resulting procedure contains only "possible" paths (i.e., paths belonging to the first and the second groups), and consequently is free from inconsistency.

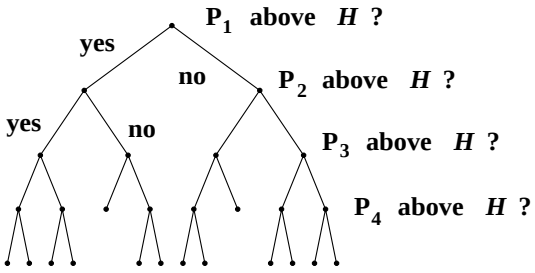



**Fig. 2.** Tree structure of the flow of the procedure formed by only nonredundant judgements.

## 3 Examples

In this section we list examples of the topology-oriented approach applied successfully. We place special attention on the topological requirement used in these applications.

### 3.1 Incremental Construction of Voronoi Diagrams

Let $S = \{\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n\}$ be a set of finite number of points in the plane. The region $R(S; \mathrm{P}_j)$ defined by $R(S; \mathrm{P}_i) = \{\mathrm{P} \in \mathbf{R}^2 \mid d(\mathrm{P}, \mathrm{P}_i) < d(\mathrm{P}, \mathrm{P}_j),\ j = 1, \ldots, i-1, i+1, \ldots, n\}$ is called the *Voronoi region* of $\mathrm{P}_i$, where $d(\mathrm{P}, \mathrm{Q})$ represents the Euclidean distance between the two points P and Q. The partition of the plane into Voronoi regions $R(S; \mathrm{P}_i)$, $i = 1, 2, \ldots, n$ , and their boundaries is called the *Voronoi diagram* for $S$.

In the incremental algorithm, we start with the Voronoi diagram for a few points, and modify it by adding the other points one by one. At each addition of a new point, a substructure of the Voronoi diagram is removed and a cyclic sequence of new edges are generated in order to represent the region for the new point.

We place three additional points whose convex hull contains all the given points, and start the incremental construction with these three points. Then, the following properties should be satisfied.

**TR 3.1.1.** At each additon of a new point, the substructure to be removed should be a tree in a graph theoretic sense.

**TR 3.1.2.** Two Voronoi regions should not share two or more common edges.

The software based on these topological requirements could construct the Voronoi diagram for one million points placed at random in single-precision floating-point arithmetic in $\mathrm{O}(n)$ time on the average [24,25]. Since this software

can generate even highly degenerate Voronoi diagrams, it can be used for the approximate computation of the Voronoi diagrams for general figures [18].

### 3.2 Divide-and-Conquer Construction of Delaunay Diagrams

From the Voronoi diagram, we can generate another diagram by connecting two points by line segments if their Voronoi regions are adjacent. This diagram is called the *Delaunay diagram*. The construction of the Voronoi diagram and the construction of the Delaunay diagram are almost equivalent, because we can easily generate one from the other.

The divide-and-conquer algorithm can construct the Delaunay diagram in $O(n \log n)$ worst-case optical time. In this algorithm, the input points are divided into left and right halves recursively, and the Delaunay diagram for the left points and that for the right points are merged into the Delaunay diagram for the whole points.

In the merge step, some edges are removed from the left and the right diagrams, and new edges (called *traverse edges*) connecting the left and the right are generated. We can place the following requirements among others.

**TR 3.2.1.** An edge should not be removed if the resulting diagram becomes disconnected.

**TR 3.2.2.** An edge should not be removed if the edge is adjacent to traverse edges at both of the terminal vertices.

**TR 3.2.3.** Parallel edges should not be generated.

**TR 3.2.4.** An edge should not be generated if it violates the planarity of the diagram.

These requirements were used in our computer program, which runs in $O(n \log n)$ time for general input, and runs in $O(n)$ average time for uniformly distributed points [13].

### 3.3 Construction of 3-d Voronoi Diagrams

The Voronoi diagram can be defined in any dimensions in the same way as in the two-dimensional space. In the three-dimensional space, the Voronoi region is a convex polyhedron and the Voronoi diagram is the partition of the space into these polyhedra and their boundaries. This partition admits the following requirements.

**TR 3.3.1.** A Voronoi region should be simply connected.

**TR 3.3.2.** Two Voronoi regions should share at most one common polygonal face.

These topological requirements were used in the computer program [8], in which the way of numerical computation was tuned up to improve the quality of

the output. The numerical difficulty we found in this experience was discussed in [22].

### 3.4 Intersection of Convex Polyhedra

The problem of intersecting two convex polyhedra can be reduced to a series of problems of cutting a convex polyhedron by a plane. To cut a convex polyhedron by a plane, we have to partition the vertices into two groups, one belonging to one side of the cutting plane, and the other belonging to the opposite side. This partition should satisfy the following requirement.

**TR 3.4.1.** The subgraph composed of the vertices belonging to each side of the cutting plane and the edges connecting them should form a connected graph.

This topological requirement was used to construct a robust software [19].

### 3.5 Three-Dimensional Convex Hull

For a finite set $S = \{\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_n\}$ in the three-dimensional space, the convex hull, denoted by $C(S)$ forms a convex polyhedron.

In the divide-and-conquer method, we apply the following procedure recursively to construct $C(S)$: first the set $S$ is divided into the left half $S_{\mathrm{L}}$ and the right half $S_{\mathrm{R}}$, next the convex hulls $C(S_{\mathrm{L}})$ and $C(S_{\mathrm{R}})$ are constructed, and finally $C(S_{\mathrm{L}})$ and $C(S_{\mathrm{R}})$ are merged into $C(S)$.

Suppose that we have already constructed $C(S_{\mathrm{L}})$ and $C(S_{\mathrm{R}})$. Then, in the merge process, we first find common tangent line $uv$ connecting vertex $u$ of $C(S_{\mathrm{L}})$ and vertex $v$ of $C(S_{\mathrm{R}})$. Next, we find a triangle having the vertices $u$, $v$ and another vertex $w$ of $C(S_{\mathrm{L}})$ or $C(S_{\mathrm{R}})$ such that the triangle $uvw$ is contained in a common tangent plane of $C(S_{\mathrm{L}})$ and $C(S_{\mathrm{R}})$. This triangle is called a *bridging triangle*. This bridging triangle has the other common tangent edge; this edge is $uw$ if $w$ is a vertex of $C(S_{\mathrm{R}})$, whereas it is $wv$ if $w$ is a vertex of $C(S_{\mathrm{L}})$. We replace $uv$ with this new common tangent edge, and repeat finding the next bridging triangle until we reach the initial common tangent edge. Finally, we remove the faces that are in the interior of the new surface, thus obtaining the convex hull $C(S_{\mathrm{L}} \cup S_{\mathrm{R}})$. The merge process can be executed in $\mathrm{O}(n)$ time, and hence the whole algorithm runs in $\mathrm{O}(n \log n)$ time.

A face or an edge of $C(S_{\mathrm{L}})$ or of $C(S_{\mathrm{R}})$ is said to be *weakly visible* if it is visible from at least one point of the other convex hull. A face that is not weakly visible is said to be *invisible*. A face of $O(S_{\mathrm{L}})$ or $C(S_{\mathrm{R}})$ is removed in the merge process if and only if it is weakly visible from the other convex hull. A weakly visible edge is said to be *clearly visible* if there is at least one point on the other convex hull from which the edge and the both side faces are visible simultaneously.

Let $X_{\mathrm{L}}$ be the set of all weakly visible open faces (meaning the faces excluding their boundaries) and all clearly visible edges of $C(S_{\mathrm{L}})$. $X_{\mathrm{L}}$ forms a connected

region, and the interior (i.e., the region obtained by removing the boundary from $X_{\mathrm{L}}$) is simply connected (i.e., has no holes). If we trace the boundary of $X_{\mathrm{L}}$ in such a way that $X_{\mathrm{L}}$ is always to the left, we get a directed cycle, which we call the *left silhouette cycle*. The *right silhouette cycle* is defined similarly. Here we have the following requirements.

**TR 3.5.1.** The silhouette cycle should not traverse the same edge twice in the same direction.

**TR 3.5.2.** The silhouette edge should not cross over itself (i.e., should not travel from one side to the other of other part of the silhouette cycle).

**TR 3.5.3.** The feft and right silhouette cycles should be chosen in such a way that the total number of remaining vertices is at least four.

Similar to the three-dimensional Voronoi diagram, the software based on these topological requirements also requires tuning the way of numerical computation if we want to get good quality output [11,12].

Gift-wrapping algorithm was also rewritten along the topology-oriented approach [20].

### 3.6 Voronoi Diagram for Line Segments

For a finite number of mutually disjoint line segments in the plane, the region nearer to a line segment than to any others is called the Voronoi region of the line segment. The plane is partioned into the Voronoi regions of the line segments and their boundaries; this partition is called the Voronoi diagram for the line segments. The boundary edges of this Voronoi diagram contains parabolic curves, and hence any algorithm for this diagram is much more sensitive to numerical errors than the other examples we have seen above.

In the incremental construction of this Voronoi diagram, we partition each line segment into the open line segment and the two terminal points, and construct the Voronoi diagram for all the terminal points first using one of the robust methods mentioned in the previous subsections. Next, we add the open line segments one by one and modify the diagram accordingly. In each addition of an open line segment, we remove some substructure from the old diagram and generate new edges to form the new region, where the following properties should be satisfied.

**TR 3.6.1.** The substructure to be removed should form a tree in a graph theoretic sense.

**TR 3.6.2.** The substructure to be removed should contain a path connecting the Voronoi regions of the two terminal points.

**TR 3.6.3.** The newly generated edges should form a cycle that exactly encloses the substructure to be removed.

These topological requirements were used in our topology-oriented software [7]. The experiments showed that this software runs in $O(n)$ time on the average for a large class of input data.

## 4 Concluding Remarks

The basic idea of the topology-oriented approach to robust geometric computation, and various implementation examples have been presented. The topology-oriented approach is a general principle for implementing geometric algorithms in a numerically robust manner. In this approach we first collect the topological requirements that should be satisfied by the geometric objects, and next rewrite the algorithms so that these requirements are always fulfilled.

The resulting software has many good properties. First, it is completely robust to numerical errors; no matter how large errors happen, the software never comes across inconsistency, and hence can pursue the task to the end, giving some output. Secondly, the output is at lease topologically consistent in the sense that the collected topological requirements are absolutely satisfied. Thirdly, the output converges to the true solution as the precision in computation increases. Fourthly, we need not worry about the degenerate situations at all in the implementation of the software, and hence the structure of the resulting software is very simple; it contains the procedure only for the general case. Recall that we start our implementation with the assumption that numerical errors are inevitable. In this environment we cannot recognize degeneracy, and hence need not prepare any exceptional processing for degenerate cases. This may sound paradoxical, but actually if we accept the existence of errors, the implementation becomes simple. Fifthly, topology-oriented implementation does not require any condition on the arithmetic precision, and consequently can use floating-point arithmetic, which is fast compared with multiple precision rational arithmetic commonly used in the exact-arithmetic approaches.

This work is partly supported by the Grant-in-Aid for Scientific Research of the Ministry of Education, Science, Sports and Culture of Japan.

## References

1. M. Benouamer, D. Michelucci and B. Peroche: Error-free boundary evaluation using lazy rational arithmetic—A detailed implementation. *Proceedings of the 2nd Symposium on Solid Modeling and Applications*, Montreal, 1993, pp. 115–126.
2. S. Fortune: Stable maintenance of point-set triangulations in two dimensions. *Proceedings of the 30th IEEE Annual Symposium on Foundations of Computer Science*, Research Triangle Park, California, 1989, pp. 494–499.
3. S. Fortune and C. von Wyk: Efficient exact arithmetic for computational geometry. *Proceedings of the 9th ACM Annual Symposium on Computational Geometry*, San Diego, 1993, pp. 163–172.
4. L. Guibas, D. Salesin and J. Stolfi: Epsilon geometry—Building robust algorithms from imprecise computations. *Proc. 5th ACM Annual Symposium on Computational Geometry* (Saarbrücken, May 1989), pp. 208–217.

5. C. M. Hoffmann: The problems of accuracy and robustness in geometric computation. *IEEE Computer*, vol. 22, no. 3 (March 1989), pp. 31–41.
6. C. M. Hoffmann: *Geometric and Solid Modeling.* Morgan Kaufmann Publisher, San Mateo, 1989.
7. T. Imai: A topology-oriented algorithm for the Voronoi diagram of polygon. *Proceedings of the 8th Canadian Conference on Computational Geometry*, 1996, pp. 107–112.
8. H. Inagaki, K. Sugihara and N. Sugie, N.: Numerically robust incremental algorithm for constructing three-dimensional Voronoi diagrams. *Proceedings of the 4th Canadian Conference Computational Geometry*, Newfoundland, August 1992, pp. 334–339.
9. M. Karasick, D. Lieber and L. R. Nackman: Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, vol. 10 (1991), pp. 71–91.
10. V. Milenkovic: Verifiable implementations of geometric algorithms using finite precision arithmetic. *Artificial Intelligence*, vol. 37 (1988), pp. 377–401.
11. T. Minakawa and K. Sugihara: Topology oriented vs. exact arithmetic—experience in implementing the three-dimensional convex hull algorithm. H. W. Leong, H. Imai and S. Jain (eds.): *Algorithms and Computation, 8th International Symposium, ISAAC'97* (Lecture Notes in Computer Science 1350), (December, 1997, Singapore), pp. 273–282.
12. T. Minakawa and K. Sugihara: Topology-oriented construction of three-dimensional convex hulls. *Optimization Methods and Software*, vol. 10 (1998), pp. 357–371.
13. Y. Oishi and K. Sugihara: Topology-oriented divide-and-conquer algorithm for Voronoi diagrams. *Computer Vision, Graphics, and Image Processing: Graphical Models and Image Processing*, vol. 57 (1995), pp. 303–314.
14. T. Ottmann, G. Thiemt and C. Ullrich: Numerical stability of geometric algorithms. *Proceedings of the 3rd ACM Annual Symposium on Computational Geometry*, Waterloo, 1987, pp. 119–125.
15. P. Schorn: Robust algorithms in a program library for geometric computation. Dissertation submitted to the Swiss Federal Institute of Technology (ETH) Zürich for the degree of Doctor of Technical Sciences, 1991.
16. M. Segal and C. H. Sequin: Consistent calculations for solid modeling. *Proceedings of the ACM Annual Symposium on Computational Geometry*, Baltimore, 1985, pp. 29–38.
17. K. Sugihara: A simple method for avoiding numerical errors and degeneracy in Voronoi diagram construction. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E75-A (1992), pp. 468–477.
18. K. Sugihara: Approximation of generalized Voronoi diagrams by ordinary Voronoi diagrams. *CVGIP: Graphical Models and Image Processing*, vol. 55 (1993), pp. 522–531.
19. K. Sugihara: A robust and consistent algorithm for intersecting convex polyhedra. *Computer Graphics Forum*, EUROGRAPHICS'94, Oslo, 1994, pp. C-45–C-54.
20. K. Sugihara: Robust gift wrapping for the three-dimensional convex hull. *J. Computer and System Sciences*, vol. 49 (1994), pp. 391–407.
21. K. Sugihara: Experimental study on acceleration of an exact-arithmetic geometric algorithm. *Proceedings of the 1997 International Conference on Shape Modeling and Applications*, Aizu-Wakamatsu, 1997, pp. 160–168.
22. K. Sugihara and H. Inagaki: Why is the 3d Delaunay triangulation difficult to construct? *Information Processing Letters*, vol. 54 (1995), pp. 275–280.

23. K. Sugihara and M. Iri: A solid modelling system free from topological inconsistency. *Journal of Information Processing*, vol. 12 (1989), pp. 380–393.
24. K. Sugihara and M. Iri: Construction of the Voronoi diagram for "one million" generators in single-precision arithmetic. *Proceedings of the IEEE*, vol. 80 (1992), pp. 1471–1484.
25. K. Sugihara and M. Iri: A robust topology-oriented incremental algorithm for Voronoi diagrams. *International Journal of Computational Geometry and Applications*, vol. 4 (1994), pp. 179–228.
26. C. K. Yap: The exact computation paradigm. D.-Z. Du and F. Hwang (eds.): *Computing in Euclidean Geometry, 2nd edition*. World Scientific, Singapore, 1995, pp. 452–492.

# Approximating Multicast Congestion*

Santosh Vempala[1] and Berthold Vöcking[2]

[1] Dept. of Mathematics, MIT
vempala@math.mit.edu
[2] International Computer Science Institute
voecking@icsi.berkeley.edu

**Abstract.** We present a randomized algorithm for approximating multicast congestion (a generalization of path congestion) to within $O(\log n)$ times the best possible. Our main tools are a linear programming relaxation and iterated randomized rounding.

## 1 The Problem

Let $G = (V, E)$ represent a communication network with $n$ vertices. A *multicast* request is specified as a subset of vertices, called terminals, that should be connected. In the multicast congestion problem, we are given several multicast requests, $S_1, \ldots, S_m \subseteq V$. A solution to the problem is a set of $m$ trees, where the $i^{th}$ tree spans the terminals of the $i^{th}$ multicast request. The *congestion* of an edge in a solution is the number of multicast trees that use the edge. The problem is to find a set of multicast trees that minimize the maximum congestion (over all the edges).

The multicast congestion problem generalizes two well-known optimization problems. The special case where each request consists of just two terminals is the standard routing problem of finding *integral paths with minimum congestion*. The latter problem (which itself is a generalization of the problem of finding edge disjoint paths) is in Karp's original list of NP-hard combinatorial problems [5], and hence our general problem is also NP-hard. On the other hand, integral path congestion can be approximated within a factor of $O(\log n)$ [6]. Our main result is an algorithm for approximating the multicast congestion to within a factor of $O(\log n)$, thus placing it in the same level of approximability as the special case.

The second problem that multicast congestion generalizes is the minimum Steiner tree problem which is the special case of a single multicast with a slightly different objective function. The goal is to minimize the sum rather than the maximum of the congestion over all edges. Finding a minimum Steiner tree is max-SNP hard, i.e. there is a constant $\epsilon > 0$ such that it is NP-hard to find a $(1 + \epsilon)$ approximation [1] (and thus our problem is also max-SNP hard). The precise constant to which the Steiner tree can be approximated is an open problem. The best known upper bound is 1.598 [3].

* Santosh Vempala was supported by a Miller fellowship at U.C.Berkeley and an NSF CAREER award. Berthold Vöcking was supported by a grant of the "Gemeinsames Hochschulsonderprogramm III von Bund und Ländern" through the DAAD.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 367–372, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Our algorithm relies on a linear programming relaxation of the problem. The relaxation is motivated by the following observation: in any integral solution to the problem, there is at least one edge across any cut that separates two terminals belonging to the same multicast request. This can be viewed as a multicommodity flow for each multicast, with the objective of minimizing the net congestion. The LP formulation is described in detail in section 2.

A solution to the LP can be viewed as a separate fractional solution for each multicast. Unfortunately, the fractional solution cannot be decomposed into a set of trees of fractional weights, which would allow to adapt the simple rounding algorithm used for the standard routing problem [6], i.e., for each multicast, choose one of the fractional trees at random with respect to the fractional weights of the tree. Instead we decompose the fractional solution for each multicast into a set of paths. These paths have the property that the total flow from each multicast terminal to other terminals in the same multicast is at least 1. We apply a variant of randomized rounding to these paths. A single iteration of randomized rounding, however, does not suffice, and we need to iterate randomized rounding on a subproblem till we find a solution. The rounding algorithm is presented in section 3.

To prove the approximation guarantee, we make two observations. The first is that in each phase (i.e. one iteration) the expected congestion is no more than twice the fractional congestion (i.e. the congestion of the LP). Second, the total number of phases is at most $\log k$, where $k$ is the maximum number of terminals in a single multicast, which is, of course, bounded by $n$. Since the fractional congestion in each phase is a lower bound on the integral congestion, the expected congestion of an edge over the entire algorithm is at most $O(\log k)$ times the optimal congestion. Further, the congestion is concentrated around this expectation and a straightforward application of Chernoff bounds leads to an $O(\log n)$ approximation (the details are in section 4).

## 2 The Relaxation

Let $x_{te}$ be a 0-1 variable indicating whether edge $e$ is chosen for the $t^{th}$ multicast. Consider the following integer program (IP).

$$\min z$$

$$\sum_{e \in \delta(S)} x_{te} \geq 1 \quad \forall t, \forall S \subset V \text{ s.t. } S \cap S_t \neq \phi, (V \setminus S) \cap S_t \neq \phi \tag{1}$$

$$\sum_{t} x_{te} \leq z \quad \forall e \in E$$

$$x_{te} \in \{0,1\} \quad \forall t, \forall e \in E \tag{2}$$

The constraints (1) ensure that any solution to the integer program connects the vertices in each multicast. Thus, this program corresponds exactly to our

problem. An LP relaxation is obtained by relaxing the constraints (2) to simply $0 \leq x_{te} \leq 1$. Let us refer to this relaxation as LP1. Although the program has exponentially many constraints, it can be solved in polynomial (in $n$) time by designing a separation oracle ([2]) that checks whether the flow across every relevant cut is at least 1.

In the case of the standard routing problem, i.e., two terminals in each multicast, a fractional solution of LP1 can be decomposed into several paths of fractional weight so that the sum of the weights of the fractional paths for each multicast is one, and the sum of the weights of the paths crossing an edge $e$ corresponds to the weight of that edge, i.e., $\sum_t x_{te}$. Therefore, choosing one of the fractional paths for each multicast at random (with respect to the weights) results in an integral solution in which the expected number of paths that cross an edge corresponds to the weight of that edge. Unfortunately, in the case of more than two terminals, the fractional solution cannot be be decomposed into a collection of fractional trees that add up to $\sum_t x_{te}$, for each edge $e$. For this reason, it is unclear how to obtain an integral solution of bounded cost from LP1.

Instead we consider a different relaxation LP2 which is at least as strong as the one above (i.e. the feasible region of the second relaxation is contained in the feasible region of the first relaxation). The relaxation is described in terms of a multicommodity flow between pairs of terminals of each multicast. In the relaxation, the variable $f_t(i,j)$ denotes the flow between the terminals $i$ and $j$ in the $t^{th}$ multicast, and $x_{te}(i,j)$ denotes the flow on edge $e$ of commodity $(i,j)$ in the $t^{th}$ multicast.

$$\min z$$

$$x_{te}(i,j) \text{ is a flow of value } f_t(i,j) \quad \forall t, \forall i < j \in S_t \tag{3}$$

$$\sum_{i \in S \cap S_t, j \in (V \setminus S) \cap S_t} f_t(i,j) \geq 1 \quad \forall t, \forall S \subset V \tag{4}$$

$$\sum_{t,i,j} x_{te}(i,j) \leq z \quad \forall e \in E$$

$$0 \leq x_{te}(i,j) \leq 1 \tag{5}$$

**Theorem 1.** $\min LP1 \leq \min LP2 \leq \min IP$

*Proof.* Any solution to the multicast congestion problem (IP) can be mapped to a solution of LP2 with same objective value as follows. For each multicast, connect all nodes in the integral tree by a cycle so that each edge in the tree is included in the cycle exactly twice. Each pair of terminals that are neighbors in the cycle (i.e., terminals that are connected by a subpath of the cycle that does not include any other terminal) exchange a flow of weight $\frac{1}{2}$. Figure 1 gives

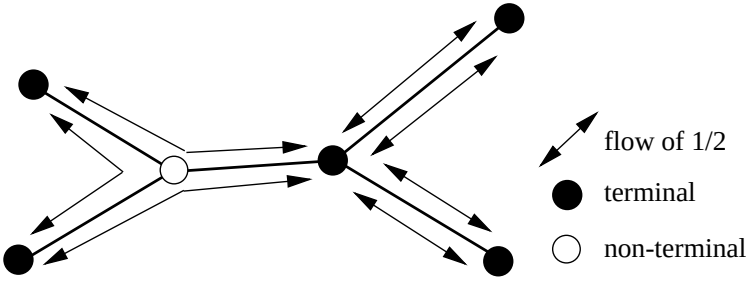



**Fig. 1.** Transformation of an IP solution to a solution of LP2.

an example of this transformation. This construction defines a multicommodity flow along the edges of the tree so that the total flow along each edge is 1 and the total flow across any cut in the graph that separates two or more terminals is at least 1. Hence, the constraints of LP2 can be satisfied while preserving the objective value.

Any solution of LP2 can be mapped to a solution of LP1 with the same objective value by simply adding up all the flows on an edge, separately for each multicast. □

**Theorem 2.** *The relaxation LP2 can be solved in polynomial time.*

*Proof.* We design a separation oracle for LP2. Given an assignment to the variables of LP2, for each multicast $S_t$, we can construct a complete graph $G_t = (S_t, E_t)$ where the weight of an edge $(i, j)$ is $f_t(i, j)$. Then constraints (4) simply require that every cut in this graph has weight at least 1. This can be easily checked by finding the minimum cut in the graph. The other constraint sets can be checked in polynomial time by examination. □

## 3 Iterated Randomized Rounding

A solution to LP2 can be separated into fractional solutions for each multicast. We will now describe how to round these fractional solutions into trees, one for each multicast. The rounding is completely independent for each multicast and is achieved by the following procedure applied separately to the fractional solution for each multicast (i.e. the values of $x_{te}(i, j)$, separately for each $t$).

1. Decompose the fractional solution into flow paths.
2. Choose one path randomly out of each multicast terminal with probability equal to the value of the flow on the path (randomized rounding).
3. If the multicast terminals are all connected, then stop and output the solution.
4. Otherwise, contract the vertices corresponding to connected components, and form a new multicast problem with the contracted vertices as the new multicast terminals.

5. The original fractional solution is still a solution for the new multicast problem. Repeat the above steps till all multicast terminals are connected.

**Lemma 1.** *There are at most* $\log k$ *iterations, where* $k$ *is the maximum number of terminals in a multicast.*

*Proof.* In each iteration, each multicast terminal is connected to at least one other terminal. Thus, the number of components drops by at least a factor of two in each iteration. □

## 4 The Approximation Guarantee

Let $X_{te}^{\ell}$ be the random variable indicating whether an active edge $e$ is selected for the $t^{th}$ multicast in the $\ell^{th}$ phase of the rounding. Define $x_{te} = \sum_{i,j} x_{te}(i,j)$, i.e., the total flow through edge $e$ of the $t^{th}$ multicast.

**Lemma 2.** $E[X_{te}^{i}] \leq 2 \cdot x_{te}$.

*Proof.* The flow through the edge $e$ for each commodity $(i,j)$ is divided among a set of paths. The random variable $X_{te}^{\ell}$ is 1 if one of these paths is selected and zero otherwise. The expected value of $X_{te}^{\ell}$ is the sum of the probabilities of these paths being selected. Since the probability of a path being selected by any fixed one of its two end points is exactly the flow on it, the sum of the probabilities of all paths through $e$ (for the $t^{th}$ multicast) is $2 \cdot x_{te}$. □

**Theorem 3.** *With high probability, the congestion of the solution found by the algorithm is less than* $O(\log k \cdot OPT + \log n)$.

*Proof.* Fix an edge $e$. Let $L(e)$ denote the number of multicasts including $e$. By definition, $L(e) = \sum_t \sum_\ell X_{te}^{\ell}$. ¿From Lemma 2, we can conclude that $E[L(e)] \leq 2 \cdot x_{te} \leq 2 \cdot \log k OPT$. We will apply a Chernoff bound to show that it is unlikely that $L(e)$ deviates significantly from its expectation.

For each multicast $t$, the random variables $X_{te}^{\ell}$ are independent from random variables for other multicasts. Random variables for the same multicast, however, may be dependent, but these variables are negatively correlated, because an edge $e$ that is chosen in round $\ell$ for a multicast will not be chosen in round $\ell' > \ell$ for the same multicast. As a consequence, we can apply a Chernoff bound (see e.g. [4]). Set $C \geq \max\{2\mathrm{e}E[L(e)], 2 \cdot (\alpha+2) \cdot \log n\}$ with $\alpha > 0$ denoting an arbitrary constant. Then

$$\mathrm{Prob}[L(e) \geq C] \;\leq\; 2^{-C/2} \;\leq\; n^{-\alpha-2} \;.$$

Summing this probability over all edges yields that the congestion does not exceed $C = O(\log k \cdot OPT + \log n)$, with probability $1 - n^{-\alpha}$. □

## 5 Open Questions

An obvious open question is whether the approximation factor can be improved. It is worth noting that path congestion is conjectured to be approximable to within a factor of 2. In fact, the worst known gap for the LP relaxation is a factor of 2. However, the best known approximation even for that special case is $O(\log n)$.

The path congestion problem can actually be approximated to within a constant times the optimum *plus* $O(\log n)$ (which is also a multiplicative $O(\log n)$ when the congestion is a constant, but better nevertheless). A less ambitious open problem is whether the tree congestion can also be approximated to within a constant plus $O(\log n)$.

## References

1. M. Bern, and P. Plassmann, *The Steiner problem with edge lengths 1 and 2*. Information Processing Letters 32:171-176, 1989.
2. M. Grötschel, L. Lovász and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*. Springer-Verlag, 1987.
3. S. Hougardy and H. J. Promel, *A 1.598 approximation algorithm for the steiner problem in graphs*. Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms, ACM-SIAM, 448-453, 1999.
4. T. Hagerup and C. Rüb, *A guided tour of Chernoff bounds*. Information Processing Letters 33(6):305-308, 1990.
5. R. M. Karp, *Reducibility among combinatorial problems*. Complexity of Computer Computations, R. E. Miller, J. W. Thatcher, Eds., Plenum Press, New York, 85-103, 1972.
6. P. Raghavan and C.D. Thompson *Randomized rounding: a technique for provably good algorithms and algorithmic proofs*. Combinatorica, 7(4):365-374, 1987.

# Approximating the Minimum $k$-way Cut in a Graph via Minimum 3-way Cuts*

Liang Zhao, Hiroshi Nagamochi, and Toshihide Ibaraki

Dept. of Applied Mathematics and Physics
Graduate School of Informatics
Kyoto University, Kyoto, Japan 606-8501
{zhao, naga, ibaraki}@kuamp.kyoto-u.ac.jp

**Abstract.** For an edge weighted undirected graph $G$ and an integer $\geq 2$, a -way cut is a set of edges whose removal leaves $G$ with at least components. We propose a simple approximation algorithm to the minimum -way cut problem. It computes a nearly optimal -way cut by using a set of minimum 3-way cuts. We show that the performance ratio of our algorithm is $2 - 3/$ for an odd and $2 - (3 - 4)/( ^2 - )$ for an even . The running time is $O( mn^3 \log(n^2/m))$ where $n$ and $m$ are the numbers of vertices and edges respectively.

## 1 Introduction

Given a simple, edge weighted undirected graph $G$ with $n$ vertices and $m$ edges, a $k$-way cut is a set of edges whose removal leaves $G$ with at least $k$ components. The minimum $k$-way cut problem is to find a $k$-way cut with the minimum weight (called a min-$k$-way cut). This problem is one of the extensions of the classical minimum $s,t$-cut problem and has practical significance in the area of VLSI design [8] and parallel computing systems [7,11]. It is known that this problem is NP-hard if $k$ is arbitrary [2]. For a fixed $k$ there are several polynomial time algorithms that solve it exactly. Dalhaus et al. [1] gave an $O(n^{O(k)})$ time algorithm for planar graphs. For arbitrary graphs, Goldschmidt et al. [2] gave an $O(n^{k^2/2-3k/2+4}F(m,n))$ time algorithm, where $F(m,n)$ stands for the running time of a maximum flow algorithm (e.g., $O(mn\log(n^2/m))$ due to [3]). Karger et al. [5] presented an $O(n^{2(k-1)}\log^3 n)$ time randomized Monte Carlo algorithm. Recently in the article [6] Kamidoi et al. proposed another deterministic algorithm and they claim that it can find an optimal solution in $O(n^{2(k-2)}F(m,n))$ time. Faster algorithms have been developed for special cases: [9] for $k = 3, 4$ and [10] for $k = 5, 6$.

Since the problem for arbitrary $k$ is NP-hard, it is interesting to design approximation algorithms that run in polynomial time. Saran et al. [12] gave two algorithms based on maximum flow computations. They showed that both of their

* This research was partially supported by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan. The first author was also supported by the IBM Asia Fellowship Program.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 373–382, 1999.
© Springer-Verlag Berlin Heidelberg 1999

algorithms can achieve a ratio of $2-2/k$ and have running times of $O(nF(m,n))$ and $O(kF(m,n))$ respectively. It is also natural to compute an approximation solution in the following greedy way. For a fixed $j$ $(2 \leq j < k)$, first divide the graph into $j$ components by removing a min-$j$-way cut, then repeat removing a minimum weight edge set whose removal increases the number of components by at least $j-1$ (the last iteration may increase a fewer number of components) until there are at least $k$ components. Then the set of all removed edges is a $k$-way cut. Intuitively one may guess that a larger $j$ will take longer time but can yield a better ratio. In fact, Kapoor [4] showed that for $j = 2$ the ratio $2 - 2/k$ can be achieved in $O(kn(m + n\log n))$ time. The same article [4] also claimed that for any $j \geq 3$, a ratio of $2 - j/k + (j-2)/k^2 + O(j/k^3)$ can be achieved in polynomial time for fixed $j$. However, the proof of the correctness for $j \geq 3$ is not complete since it contains a lemma, Lemma 4.3 [4], which is not valid in general (as will be discussed in Sect. 3.4). In this paper, we show that with a slight modification, the above algorithm with $j = 3$ can achieve a ratio of $2 - 3/k$ for odd $k \geq 3$ and $2 - (3k-4)/(k^2-k)$ for even $k \geq 2$ (note that $2 - 3/k < 2 - (3k-4)/(k^2-k) = 2 - 3/k + 1/k^2 + O(1/k^3)$). Our algorithm needs at most $k/2$ min-3-way cut computations. Since one computation takes $O(mn^3 \log(n^2/m))$ time due to [9], the running time of our algorithm is $O(kmn^3 \log(n^2/m))$.

## 2 Preliminaries and Algorithm

We use $(G, w)$ to denote a network where $G$ is a graph and $w$ is an edge weight function. We denote the number of the components in $G$ by $comp(G)$. For two vertices $v_1$ and $v_2$, we denote the edge between $v_1$ and $v_2$ by $(v_1, v_2)$. For two vertex sets $V_1$ and $V_2$ we use $E_G(V_1, V_2)$ to denote the edge set $\{(v_1, v_2) \in G \mid v_1 \in V_1, v_2 \in V_2\}$. Generally, we define $E_G(V_1, V_2, \ldots, V_p) = \bigcup_{1 \leq i < j \leq p} E_G(V_i, V_j)$ for vertex sets $V_1, V_2, \ldots, V_p$. For an edge set $E'$, we use $G - E'$ (resp., $G + E'$) to denote the graph derived from $G$ by removing (resp., adding) $E'$.

For a network $(G, w)$ with a real weight function $w$ (note that $G$ may not be connected), we want to find a minimum weight edge set $F$ whose removal increases the number of components by at least 2. This problem can be reduced to a minimum 3-way cut problem in the following way. (It can also be solved via a DP approach, see [4].)

PROCEDURE INC2COMP($G$,$w$)
Input: A network $(G, w)$ with a real weight function $w$.
Output: A minimum weight edge set $F$, subject to $comp(G-F) - comp(G) \geq 2$.

1. $C_1, C_2, \ldots, C_p \leftarrow$ the components in $G$;
2. choose arbitrarily vertices $v_1 \in C_1, v_2 \in C_2, \ldots, v_p \in C_p$;
3. $EXTG \leftarrow G + \{(v_1, v_j) \mid j = 2, \ldots, p\}$;
4. $w((v_1, v_j)) \leftarrow \infty$ for $j = 2, \ldots, p$;
5. $F \leftarrow$ a min-3-way cut in $(EXTG, w)$.

(Note that if $w > 0$ then $comp(G-F)-comp(G) = 2$ holds.) Now we can describe our approximation algorithm for the minimum $k$-way cut problem. We suppose that the given graph is connected and the edge weight function is positive. For an even $k$, we will first compute a min-2-way cut before computing min 3-way cuts (in $EXTG$). (This is different from the algorithm in [4] for $j = 3$, which computes the min-3-way cuts first.)

ALGORITHM 3-SPLIT($G_0$,$w$,$k$)
Input: A connected network $(G_0, w)$ with $w > 0$, and an integer $k$ $(2 \leq k < n)$.
Output: An edge set $H$, such that $comp(G_0 - H) \geq k$.

```
1.    G ← G_0;    H ← φ;
2.    if k is even then
3.        F ← a min-2-way cut in (G, w);    G ← G − F;    H ← F;
4.    end
5.    while comp(G) < k do
6.        F ← INC2COMP(G,w);
7.        G ← G − F;    H ← H ∪ F;
8.    end
```

**Theorem 1.** *For a connected network $(G_0, w)$ and an integer $k$, where $G_0$ has $n$ vertices and $m$ edges, $w > 0$ and $2 \leq k < n$, algorithm 3-SPLIT terminates in $O(kmn^3 \log(n^2/m))$ time. Let $C^*$ be a minimum $k$-way cut in $(G_0, w)$ and $H$ be the output of 3-SPLIT. Then*

$$w(H) \leq \begin{cases} (2 - 3/k)w(C^*) & \text{if } k \text{ is odd,} \\ (2 - \dfrac{3k-4}{k^2-k})w(C^*) & \text{if } k \text{ is even.} \end{cases}$$

□

## 3 Proof of Theorem 2.1

### 3.1 Mainstream of the Proof

By the assumption of $w > 0$, 3-SPLIT executes exactly $\lfloor k/2 \rfloor$ minimum 2 or 3-way cut computations. Since one computation takes $O(mn^3 \log(n^2/m))$ time [9], the stated running time follows. Next we consider the approximation ratio. Let $H = F_1 \cup \cdots \cup F_{\lfloor k/2 \rfloor}$ denote the $k$-way cut output of 3-SPLIT, where each $F_i$ denotes the $i$-th edge set found by 3-SPLIT. Let $G_i = G_{i-1} - F_i = G_0 - (F_1 \cup \cdots \cup F_i)$. Let $V$ be the vertex set of graph $G_0$. We say that $\mathcal{P} = \{V_1, V_2, \ldots, V_p\}$ is a *$p$-way partition* if all $V_i$ are nonempty, disjoint and $V_1 \cup V_2 \cup \cdots \cup V_p = V$ holds. Define $E_{G_0}(\mathcal{P}) = E_{G_0}(V_1, V_2, \ldots, V_p)$ which is a $p$-way cut in $G_0$. Given a $p$-way cut $C$ in $G_0$, let $V_1, V_2, \ldots, V_q$ be the vertex sets of components in $G_0 - C$. Define $\mathcal{V}(C) = \{V_1, V_2, \ldots, V_q\}$, which is a $q$-way partition. Note that $|\mathcal{V}(C)| = q = comp(G_0 - C) \geq p$.

**Definition 1.** *Given an odd $k$ and a $k$-way partition $\mathcal{P} = \{V_1, V_2, \ldots, V_k\}$ (as an ordered set), define $A_i(\mathcal{P}) = E_{G_0}(V_{2i-1}, V_{2i}, V - (V_{2i-1} \cup V_{2i}))$ for $i = 1, 2, \ldots, (k-1)/2$.*

**Definition 2.** *Given an even $k$ and a $k$-way partition $\mathcal{P} = \{V_1, V_2, \ldots, V_k\}$ (as an ordered set), define $A_1(\mathcal{P}) = E_{G_0}(V_1, V - V_1)$ and $A_i(\mathcal{P}) = E_{G_0}(V_{2i-2}, V_{2i-1}, V - (V_{2i-2} \cup V_{2i-1}))$ for $i = 2, \ldots, k/2$.*

Now we state the following two lemmas, based on which the proof of Theorem 1 is then given. We will prove the two lemmas in subsection 3.2 and 3.3.

**Lemma 1.** *Given an odd $k$ and a connected network $(G_0, w)$ where $w > 0$, for any $k$-way partition $\mathcal{P} = \{V_1, \ldots, V_k\}$,*

$$w(H) = \sum_{i=1}^{k_0} w(F_i) \leq \sum_{i=1}^{k_0} w(A_i(\mathcal{P})), \tag{1}$$

*where $k_0 = (k-1)/2$, and $A_i(\mathcal{P})$ are defined in Definition 1.* □

**Lemma 2.** *Given an even $k$ and a connected network $(G_0, w)$ where $w > 0$, for any $k$-way partition $\mathcal{P} = \{V_1, \ldots, V_k\}$,*

$$w(H) = \sum_{i=1}^{k_0} w(F_i) \leq \sum_{i=1}^{k_0} w(A_i(\mathcal{P})), \tag{2}$$

*where $k_0 = k/2$, and $A_i(\mathcal{P})$ are defined in Definition 2.* □

These lemmas tell that for any $k$-way partition $\mathcal{P}$, the output of 3-SPLIT has a weight no more than $\sum_{i=1}^{k_0} w(A_i(\mathcal{P}))$. We now estimate this sum in terms of the weight of the min-$k$-way cut $C^*$. Let $\mathcal{P}^* = \mathcal{V}(C^*) = \{V_1^*, V_2^*, \ldots, V_k^*\}$ (notice that $|\mathcal{P}^*| = k$ holds by $w > 0$ and the optimality of $C^*$). Notice that $\sum_{i=1}^{k_0} w(A_i(\mathcal{P}^*))$ varies depending on the numbering of $V_1^*, V_2^*, \ldots, V_k^*$.

**Lemma 3.** *For an odd $k$, there is a numbering of $V_1^*, V_2^*, \ldots, V_k^*$ which satisfies*

$$\sum_{i=1}^{k_0} w(A_i(\mathcal{P}^*)) \leq (2 - 3/k) w(C^*), \text{ where } k_0 = (k-1)/2. \tag{3}$$

*Proof.* Since $G_0$ is undirected, we have $\sum_{i=1}^{k} w(E_{G_0}(V_i^*, V - V_i^*)) = 2w(C^*)$. Thus we can suppose without loss of generality that $w(E_{G_0}(V_k^*, V - V_k^*)) \geq \frac{2}{k} w(C^*)$. Let $\beta = w(E_{G_0}(V_1^*, V_2^*, \ldots, V_{k-1}^*)) = w(C^* - E_{G_0}(V_k^*, V - V_k^*))$. Then we have

$$\beta \leq (1 - \frac{2}{k}) w(C^*).$$

Let $\mathcal{G}$ be the complete graph with $2k_0$ $(= k-1)$ vertices $u_1, u_2, \ldots, u_{k-1}$. For all $1 \leq i < j \leq k-1$, edge $(u_i, u_j)$ has weight $w(E_{G_0}(V_i^*, V_j^*))$. Note that

the total weight of the edges in $\mathcal{G}$ is $\beta$ by this definition. For a numbering $i_1, i_2, \ldots, i_k$ of $V_1^*, V_2^*, \ldots, V_k^*$ where $i_k = k$ (thus $\mathcal{P}^* = \{V_{i_1}^*, \ldots, V_{i_{k-1}}^*, V_k^*\}$), consider a perfect matching $\left\{(u_{i_1}, u_{i_2}), (u_{i_3}, u_{i_4}), \ldots, (u_{i_{k-2}}, u_{i_{k-1}})\right\}$ in $\mathcal{G}$. Let $\alpha = \sum_{i=1}^{k_0} w(A_i(\mathcal{P}^*))$ (recall that $A_j(\mathcal{P}^*) = E_{G_0}(V_{i_{2j-1}}^*, V_{i_{2j}}^*, V - (V_{i_{2j-1}}^* \cup V_{i_{2j}}^*))$). Then the weight of the matching $\mu = \sum_{j=1}^{k_0} w(E_{G_0}(V_{i_{2j-1}}^*, V_{i_{2j}}^*))$ satisfies $\alpha + \mu = w(C^*) + \beta$ (where edges in $E_{G_0}(V_k^*, V - V_k^*)$ are counted once, and the other edges are counted twice). Consider a numbering that makes $\mu$ be maximum in $\mathcal{G}$. We show that $\mu \geq \beta/(k-2)$. For two matching edges $(u_{i_{2j-1}}, u_{i_{2j}})$ and $(u_{i_{2l-1}}, u_{i_{2l}})$, there are four unmatched edges $(u_{i_{2j-1}}, u_{i_{2l-1}})$, $(u_{i_{2j-1}}, u_{i_{2l}})$, $(u_{i_{2j}}, u_{i_{2l-1}})$ and $(u_{i_{2j}}, u_{i_{2l}})$. Since the matching is maximum, $w(u_{i_{2j-1}}, u_{i_{2l-1}}) + w(u_{i_{2j}}, u_{i_{2l}}) \leq w(u_{i_{2j-1}}, u_{i_{2j}}) + w(u_{i_{2l-1}}, u_{i_{2l}})$ and $w(u_{i_{2j-1}}, u_{i_{2l}}) + w(u_{i_{2j}}, u_{i_{2l-1}}) \leq w(u_{i_{2j-1}}, u_{i_{2j}}) + w(u_{i_{2l-1}}, u_{i_{2l}})$ hold. Thus the four unmatched edges have weight at most $2(w(u_{i_{2j-1}}, u_{i_{2j}}) + w(u_{i_{2l-1}}, u_{i_{2l}}))$. Thus the unmatched edges have weight at most $2\sum_{1 \leq j < l \leq k_0}(w(u_{i_{2j-1}}, u_{i_{2j}}) + w(u_{i_{2l-1}}, u_{i_{2l}})) = 2(k_0 - 1)\mu$. Therefore we have

$$\mu \geq \frac{\sum_{1 \leq i < j \leq k-1} w(u_i, u_j)}{2(k_0 - 1) + 1} = \frac{\beta}{k-2}. \tag{4}$$

Thus the corresponding numbering satisfies $\sum_{i=1}^{k_0} w(A_i(\mathcal{P}^*)) = w(C^*) + \beta - \mu \leq w(C^*) + (1 - \frac{1}{k-2})\beta \leq (2 - \frac{3}{k})w(C^*)$. □

In a similar way we can show the next lemma (an alternative proof can be found in [4]).

**Lemma 4.** *For an even $k$, there is a numbering of $V_1^*, V_2^*, \ldots, V_k^*$ which satisfies*

$$\sum_{i=1}^{k_0} w(A_i(\mathcal{P}^*)) \leq (2 - \frac{3k-4}{k^2-k})w(C^*), \quad \textit{where } k_0 = k/2. \tag{5}$$

□

By these lemmas, Theorem 1 has been proved.

### 3.2 Proof of Lemma 1

We proceed to prove Lemma 1 by induction on $k$. It is trivial for $k = 3$ since $F_1$ is a min-3-way cut and $A_1$ is a 3-way cut in $G_0$. Suppose that Lemma 1 holds for $k = 2i - 1$. We consider the case of $k = 2i + 1$. First, if there exists a $j \in \{1, 2, \ldots, i\}$ satisfying $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) \geq 2$, then edge set $A_j(\mathcal{P})$ is a possible choice of $F_i$. By the optimality of $w(F_i)$, we have $w(F_i) \leq w(A_j(\mathcal{P}))$. By the induction hypothesis, we can assume that $w(F_1) + w(F_2) + \cdots + w(F_{i-1}) \leq w(A_1(\mathcal{P})) + \cdots + w(A_{j-1}(\mathcal{P})) + w(A_{j+1}(\mathcal{P})) + \ldots + w(A_i(\mathcal{P}))$ holds for a $(2i-1)$-way partition $\{V_1, \ldots, V_{2j-2}, V_{2j+1}, \ldots, V_{2i}, V_{2j-1} \cup V_{2j} \cup V_{2i+1}\}$, which implies (1). Therefore, in what follows, we consider the case in which $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) \leq 1$ holds for all $j = 1, 2, \ldots, i$.

**Proposition 1.** *$comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) = 0$ holds if and only if $A_j(\mathcal{P}) \subseteq F_1 \cup \cdots \cup F_{i-1}$.*

*Proof.* Recall that $G_{i-1} = G_0 - (F_1 \cup \cdots \cup F_{i-1})$. By the definition of $A_j(\mathcal{P})$ the proposition is clear. □

**Proposition 2.** *If $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) = 1$, then there is exactly one subset $W \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$ for which exactly one of the following (a) and (b) holds.*

*(a) $W \subseteq V_{2j-1} \cup V_{2j}$ and $W \cap V_{2j-1} \neq \emptyset \neq W \cap V_{2j}$.*

*(b) $W \cap V_{2j-1} = \emptyset$ and $W \cap V_{2j} \neq \emptyset \neq W - V_{2j}$ (or symmetrically, $W \cap V_{2j} = \emptyset$ and $W \cap V_{2j-1} \neq \emptyset \neq W - V_{2j-1}$).*

*Furthermore, for any $W' \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1}) - W$, $W' \cap V_{2j-1} \neq \emptyset$ (resp., $W' \cap V_{2j} \neq \emptyset$) implies $W' \subseteq V_{2j-1}$ (resp., $W' \subseteq V_{2j}$).*

*Proof.* By Proposition 1, there is an edge $(v_1, v_2) \in A_j(\mathcal{P}) - (F_1 \cup \cdots \cup F_{i-1})$. Let $W \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$ satisfy $v_1, v_2 \in W$. Edge $(v_1, v_2) \in A_j(\mathcal{P})$ means that at least one of $v_1, v_2$ is in $V_{2j-1} \cup V_{2j}$. Assume without loss of generality that $v_1 \in V_{2j}$ (thus $v_2 \notin V_{2j}$). Since $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) = 1 < 2$ is assumed, it is easy to see that if $v_2 \in V_{2j-1}$ then $(a)$ holds, while if $v_2 \notin V_{2j-1}$ then $(b)$ holds. Notice that such $W$ must be unique, otherwise $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) \geq 2$ would hold. Thus for any $W' \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1}) - W$, $W' \cap V_{2j-1} \neq \emptyset$ (resp., $W' \cap V_{2j} \neq \emptyset$) implies $W' \subseteq V_{2j-1}$ (resp., $W' \subseteq V_{2j}$). □

**Proposition 3.** *If $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) \leq 1$ holds for all $j = 1, 2, \ldots, i$, then there exist two indices $j_1, j_2 \in \{1, 2, \ldots, i\}$ such that*

$$comp(G_{i-1} - (A_{j_1}(\mathcal{P}) \cup A_{j_2}(\mathcal{P}))) - comp(G_{i-1}) \geq 2. \tag{6}$$

*More precisely, there are subsets $W_1, W_2 \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$, which satisfy one of the following cases (i), (ii), and (iii) (symmetric cases are omitted).*

*(i) $W_1$ (resp., $W_2$) satisfies (a) in Proposition 2 with $j = j_1$ (resp., $j = j_2$),*

*(ii) $W_1$ (resp., $W_2$) satisfies (a) (resp., (b)) in Proposition 2 with $j = j_1$ (resp., $j = j_2$),*

*(iii) $W_1$ (resp., $W_2$) satisfies (b) in Proposition 2 with $j = j_1$ (resp., $j = j_2$), and either $W_1 \neq W_2$ or $W_1 = W_2$, $W_1 - (V_{2j_1} \cup V_{2j_2}) \neq \emptyset$ holds.*

*Proof.* Let $J_0 = \{\ j \mid comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) = 0\}$ and $J_1 = \{1, 2, \ldots, i\} - J_0 = \{\ j \mid comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) = 1\}$. By Proposition 1, we see that $G_{i-1} - \bigcup_{j=1}^{i} A_j(\mathcal{P}) = G_{i-1} - \bigcup_{j \in J_1} A_j(\mathcal{P})$. By definition, $comp(G_{i-1} - \bigcup_{j=1}^{i} A_j(\mathcal{P})) \geq comp(G_0 - \bigcup_{j=1}^{i} A_j(\mathcal{P})) \geq 2i + 1$ holds. Thus

$$\begin{aligned} &comp(G_{i-1} - \bigcup_{j \in J_1} A_j(\mathcal{P})) - comp(G_{i-1}) \\ &= comp(G_{i-1} - \bigcup_{j=1}^{i} A_j(\mathcal{P})) - comp(G_{i-1}) \geq (2i+1) - (2i-1) = 2. \end{aligned} \tag{7}$$

(Notice that $comp(G_{i-1}) = 2i-1$ since $w > 0$.) Thus by (7) there exist at least two indices $j_1', j_2' \in J_1$ (that is, $|J_1| \geq 2$). By applying Proposition 2 to $j = j_1'$ and $j_2'$, we see that there exist subsets $W_1', W_2' \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$, which satisfy one of the following cases $(i')$, $(ii')$, and $(iii')$ (symmetric cases are omitted).

$(i')$ $W_1'$ (resp., $W_2'$) satisfies $(a)$ in Proposition 2 with $j = j_1'$ (resp., $j_2'$),

$(ii')$ $W_1'$ (resp., $W_2'$) satisfies $(a)$ (resp., $(b)$) in Proposition 2 with $j = j_1'$ (resp., $j_2'$),

$(iii')$ $W_1'$ (resp., $W_2'$) satisfies $(b)$ in Proposition 2 with $j = j_1'$ (resp., $j_2'$).

If $(i')$ or $(ii')$ occurs, then it is clear that $W_1' \neq W_2'$ and (6) holds for $j_1'$ and $j_2'$. Thus we can get Proposition 3. Similarly if $(iii')$ occurs with $W_1' \neq W_2'$, or $W_1' = W_2'$, $W_1 - (V_{2j_1} \cup V_{2j_2}) \neq \emptyset$, then Proposition 3 still holds.

The only case that (6) does not hold is $(iii')$ with $W_1' = W_2' \subseteq V_{2j_1'} \cup V_{2j_2'}$ (symmetric cases are omitted). In this case, by (7) we see that $J_1 \neq \{j_1', j_2'\}$. Let $j_3' \in J_1 - \{j_1', j_2'\}$. We show that $j_1'$ and $j_3'$ satisfy (6).

By applying Proposition 2 to $j_3'$, we see that there is a $W_3' \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$ which satisfies $(a)$ or $(b)$ in Proposition 2 with $j = j_3'$. Since $W_3' \neq W_1'$ (because $W_3' \cap (V_{2j_3'-1} \cup V_{2j_3'}) \neq \emptyset$ and $W_1' \subseteq V_{2j_1'} \cup V_{2j_2'}$ and $j_3' \neq j_1', j_2'$), we see that $j_1'$ and $j_3'$ satisfy (6). Hence Proposition 3 holds. □

We will finish proving Lemma 1 by showing that (1) holds in all the three cases in Proposition 3. First in case $(i)$, it is clear that the edge set $E_{G_0}(W_1 \cap V_{2j_1-1}, W_1 \cap V_{2j_1}) \cup E_{G_0}(W_2 \cap V_{2j_2-1}, W_2 \cap V_{2j_2})$ is a possible choice of $F_i$. Thus

$$\begin{aligned} w(F_i) &\leq w(E_{G_0}(W_1 \cap V_{2j_1-1}, W_1 \cap V_{2j_1}) \cup E_{G_0}(W_2 \cap V_{2j_2-1}, W_2 \cap V_{2j_2})) \\ &\leq w(E_{G_0}(V_{2j_1-1}, V_{2j_1}) \cup E_{G_0}(V_{2j_2-1}, V_{2j_2})). \end{aligned}$$

Consider a $(2i-1)$-way partition $\mathcal{P}' = \{V_{2j-1}, V_{2j}$ (for all $j \neq j_1, j_2)$, $V_{2j_1-1} \cup V_{2j_1}, V_{2j_2-1} \cup V_{2j_2}, V_{2i+1}\}$. By the induction hypothesis, we have

$$\begin{aligned} &\sum_{j=1}^{i-1} w(F_j) \leq \sum_{j=1}^{i-1} w(A_j(\mathcal{P}')) \\ &= \sum_{j \neq j_1, j_2} w(A_j(\mathcal{P})) \\ &\quad + w\Big((A_{j_1}(\mathcal{P}) \cup A_{j_2}(\mathcal{P})) - (E_{G_0}(V_{2j_1-1}, V_{2j_2}) \cup E_{G_0}(V_{2j_2-1}, V_{2j_2}))\Big) \\ &\leq \sum_{j=1}^{i} w(A_j(\mathcal{P})) - w(E_{G_0}(V_{2j_1-1}, V_{2j_1}) \cup E_{G_0}(V_{2j_2-1}, V_{2j_2})). \end{aligned}$$

Thus (1) (with $k_0 = i$) holds in this case.

Similarly, in case $(ii)$ we have

$$\begin{aligned} w(F_i) &\leq w(E_{G_0}(W_1 \cap V_{2j_1-1}, W_1 \cap V_{2j_1}) \cup E_{G_0}(W_2 \cap V_{2j_2}, W_2 - V_{2j_2})) \\ &\leq w\Big(E_{G_0}(V_{2j_1-1}, V_{2j_1}) \cup E_{G_0}(V_{2j_2}, V - (V_{2j_2-1} \cup V_{2j_2}))\Big). \end{aligned}$$

(Notice that $W_2 \cap V_{2j_2-1} = \emptyset$.) Apply the induction hypothesis on a $(2i-1)$-way partition $\mathcal{P}' = \{V_{2j-1}, V_{2j}$ (for all $j \neq j_1, j_2)$, $V_{2j_1-1} \cup V_{2j_1}, V_{2j_2-1}, V_{2j_2} \cup V_{2i+1}\}$. We have

$$\begin{aligned}
&\sum_{j=1}^{i-1} w(F_j) \leq \sum_{j=1}^{i-1} w(A_j(\mathcal{P}')) \\
&= \sum_{j \neq j_1, j_2} w(A_j(\mathcal{P})) + w\Big((A_{j_1}(\mathcal{P}) - E_{G_0}(V_{2j_1-1}, V_{2j_2})) \\
&\qquad\qquad \cup \big(A_{j_2}(\mathcal{P}) - E_{G_0}(V_{2j_2}, V - (V_{2j_2-1} \cup V_{2j_2}))\big)\Big) \\
&\leq \sum_{j=1}^{i} w(A_j(\mathcal{P})) - w\Big(E_{G_0}(V_{2j_1-1}, V_{2j_1}) \cup E_{G_0}(V_{2j_2}, V - (V_{2j_2-1} \cup V_{2j_2}))\Big).
\end{aligned}$$

Thus (1) (with $k_0 = i$) holds in this case, too.

Finally in the case $(iii)$, in both cases of $W_1 = W_2$ and $W_1 \neq W_2$, we have

$$\begin{aligned}
&w(F_i) \leq w(E_{G_0}(W_1 \cap V_{2j_1}, W_1 - V_{2j_1}) \cup E_{G_0}(W_2 \cap V_{2j_2}, W_2 - V_{2j_2})) \\
&\leq w\Big(E_{G_0}(V_{2j_1}, V - (V_{2j_1-1} \cup V_{2j_1})) \cup E_{G_0}(V_{2j_2}, V - (V_{2j_2-1} \cup V_{2j_2}))\Big) \\
&\leq w(A_{j_1}) + w(A_{j_2}) - w(E_{G_0}(V_{2j_1-1}, V - V_{2j_1-1}) \cup E_{G_0}(V_{2j_2-1}, V - V_{2j_2-1})).
\end{aligned}$$

(Notice that $W_1 \cap V_{2i-3} = W_2 \cap V_{2i-1} = \emptyset$.) Applying the induction hypothesis on a $(2i-1)$-way partition $\mathcal{P}' = \{V_{2j-1}, V_{2j}$ (for all $j \neq j_1, j_2)$, $V_{2j_1-1}, V_{2j_2-1}$, $V_{2j_1} \cup V_{2j_2} \cup V_{2i+1}\}$, we have

$$\begin{aligned}
&\sum_{j=1}^{i-1} w(F_j) \leq \sum_{j=1}^{i-1} w(A_j(\mathcal{P}')) \\
&= \sum_{j \neq j_1, j_2} w(A_j(\mathcal{P})) + w(E_{G_0}(V_{2j_1-1}, V - V_{2j_1-1}) \cup E_{G_0}(V_{2j_2-1}, V - V_{2j_2-1})).
\end{aligned}$$

Thus (1) (with $k_0 = i$) holds in this case too.
Hence Lemma 1 has been proved. □

### 3.3 Proof of Lemma 2

Analogously to the proof of Lemma 1, we also proceed by induction on $k$. It is trivial for $k = 2$. Supposing that Lemma 2 holds for $k = 2i-2$, we consider the case of $k = 2i$. First consider the case in which there exists a $j \in \{1, 2, \ldots, i\}$ satisfying $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) \geq 2$. If $j > 1$, then the proof is the same as for Lemma 1. If $j = 1$, we have $w(F_i) \leq w(A_1)$ by the optimality of $F_i$. Consider a $(2i-2)$-way partition $\{V_{2i-2} \cup V_{2i-1}, V_2, V_3, \ldots, V_{2i-3}, V_{2i} \cup V_1\}$. From the induction hypothesis we can easily get (2). Thus we only need to consider the case that $comp(G_{i-1} - A_j(\mathcal{P})) - comp(G_{i-1}) \leq 1$ holds for all

$j = 1, 2, \ldots, i$. In a similar way as in proving Lemma 1, we can show that there exist $j_1 \neq j_2 \in \{1, 2, \ldots, i\}$ satisfying

$$\begin{cases} comp(G_{i-1} - A_{j_1}(\mathcal{P})) - comp(G_{i-1}) & = 1, \\ comp(G_{i-1} - A_{j_2}(\mathcal{P})) - comp(G_{i-1}) & = 1, \\ comp(G_{i-1} - (A_{j_1}(\mathcal{P}) \cup A_{j_2}(\mathcal{P}))) - comp(G_{i-1}) & \geq 2. \end{cases} \tag{8}$$

Suppose without loss of generality that $j_2 \neq 1$. If $j_1 \neq 1$ then the proof of Lemma 1 again applies. If $j_1 = 1$, then there exists $W_1 \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$ such that $\emptyset \neq V_1 \cap W_1 \neq W_1$. Also, there exists $W_2 \in \mathcal{V}(F_1 \cup \cdots \cup F_{i-1})$ which satisfies one of the following cases $(i)$ – $(ii)$.

$(i)$ $W_2 \subseteq V_{2j_2-2} \cup V_{2j_2-1}$ and $W_2 \cap V_{2j_2-2} \neq \emptyset \neq W_2 \cap V_{2j_2-1}$.

$(ii)$ $W_2 \cap V_{2j_2-2} = \emptyset$ and $W_2 \cap V_{2j_2-1} \neq \emptyset \neq W_2 - V_{2j_2-1}$ (or symmetrically, $W_2 \cap V_{2j_2-1} = \emptyset$ and $W_2 \cap V_{2j_2-2} \neq \emptyset \neq W_2 - V_{2j_2-2}$).

In case $(i)$, we have $w(F_i) \leq w(A_1) + w(E_{G_0}(V_{2j_2-2}, V_{2j_2-1}))$. Then it is easy to show (2) by applying induction hypothesis to a $(2i-2)$-way partition $\{V_{2j_2-2} \cup V_{2j_2-1}, V_{2j-2}, V_{2j-1}$ (for $j \neq 1, j_2$), $V_1 \cup V_{2i}\}$.

In case $(ii)$, we have $w(F_i) \leq w(A_1 \cup E_{G_0}(V_{2j_2-1}, V - (V_{2j_2-2} \cup V_{2j_2-1})))$. Then (2) follows from a $(2i-2)$-way partition $\{V_{2j_2-2}, V_{2j-2}, V_{2j-1}$ (for $j \neq 1, j_2$), $V_1 \cup V_{2j_2-1} \cup V_{2i}\}$. Thus we have proved Lemma 2. □

### 3.4 Remarks

The correctness of the algorithm [4] (which is described in Sect. 1) relies on the inequality (2) (or an extended form for $j \geq 4$), as claimed in Lemma 4.3 [4]. We remark that the property of Lemma 2 is no longer valid if we first compute $F_1, \ldots, F_{k_0-1}$ as min-3-way cuts (e.g., by INC2COMP), and then compute $F_{k_0}$ as a min-2-way cut in $EXTG$. This is illustrated by the next example. Let $k = 4$. Consider a graph with vertices $\{a, b, c, d, e\}$ and edges $\{(a, b), (b, c), (c, d), (d, e), (e, c)\}$. Edge $(b, c)$ has weight 1.5 and others have weight 1. If we compute a min-3-way cut first, then we have $F_1 = \{(a, b), (b, c)\}$ and $F_2 = \{(c, d), (d, e)\}$. Consider a 4-way partition $\{\{d\}, \{e\}, \{a\}, \{b, c\}\}$ (thus $A_1 = \{(c, d), (d, e), (e, c)\}$ and $A_2 = \{(a, b)\}$), we have

$$w(F_1) + w(F_2) = 2.5 + 2 > 3 + 1 = w(A_1) + w(A_2).$$

(In fact, the algorithm [4] constructs a $k$-way cut in this way for $j = 3$ and an even $k$. Thus Lemma 4.3 [4] is not valid in this case.)

There is also an example to show that such an extension of Lemma 1 (as introduced in Lemma 4.3 [4]) is no longer valid. For $j = 4$ and $k = 7$, let $G$ be a graph with vertices $\{a, b, c, d, e, f, g, h\}$ and edges $\{(a, b), (b, c), (c, d), (d, b), (d, e), (e, f), (f, g), (g, e), (g, h)\}$, where $(d, e)$ has weight 3 and others have 2. In the similar way of 3-SPLIT (or the algorithm [4]), we first get a min-4-way cut $F_1 = \{(a, b), (d, e), (g, h)\}$, and then $F_2 = \{(b, c), (c, d), (d, b), (e, f), (e, g)\}$. One may expect that the next inequality holds for any 7-way partition $\{V_1, V_2, \ldots, V_7\}$.

$$w(F_1) + w(F_2) \leq w(A_1) + w(A_2),$$

where $A_1 = E_G(V_1, V_2, V_3, V_4 \cup V_5 \cup V_6 \cup V_7))$, and $A_2 = E_G(V_4, V_5, V_6, V_1 \cup V_2 \cup V_3 \cup V_7)$. However, consider a 7-way partition $\{\{a\}, \{b\}, \{c\}, \{f\}, \{g\}, \{h\}, \{d, e\}\}$ (thus $A_1 = \{(a, b), (b, c), (b, d), (c, d)\}$ and $A_2 = \{(e, f), (e, g), (f, g), (g, h)\}$). We see that

$$w(F_1) + w(F_2) = 7 + 10 > 8 + 8 = w(A_1) + w(A_2).$$

## 4 Conclusion

We have shown that, via repeated applications of minimum 3-way cuts we can obtain a $k$-way cut whose weight is no more than $2 - 3/k$ (resp., $2 - (3k - 4)/(k^2 - k)$) times of the optimal for odd $k$ (resp. even $k$). It is not difficult to show that the ratios are tight. Can this be improved if we compute a $k$-way cut via minimum $j$-way cuts with $j \geq 4$? It seems that a different approach is needed for general $j \geq 4$.

## References

1. E. Dalhaus, D. S. Johnson, C. H. Papadimitriou, P. Seymour and M. Yannakakis, *The complexity of the multiway cuts*, extended abstract 1983. *The complexity of the multiterminal cuts*, SIAM J. Comput. Vol. 23, No. 4, 1994, 864–894.
2. O. Goldschmidt and D. S. Hochbaum, *Polynomial algorithm for the -cut problem*, In Proc. 29th IEEE FOCS, 1988, 444–451.
3. A. V. Goldberg and R. E. Tarjan, *A new approach to the maximum flow problem*, J. of ACM, Vol. 35, No. 4, 1988, 921–940.
4. S. Kapoor, *On minimum 3-cuts and approximating -cuts using cut trees*, Lecture Notes in Comput. Sci., 1084, Springer-Verlag, Integer programming and combinatorial optimization, 1996, 132–146.
5. D. R. Karger and C. Stein, *A new approach to the minimum cut problems*, J. of ACM, Vol. 43, No. 4, 1996, 601–640.
6. Y. Kamidoi, S. Wakabayasi and N. Yoshida, *A new approach to the minimum -way partition problem for weighted graphs*, Technical Report of IEICE. COMP97-25, 1997, 25–32.
7. C. H. Lee, M. Kim and C. I. Park, *An efficient -way graph partitioning algorithm for task allocation in parallel computing systems*, In Proc. IEEE Int. Conf. on Computer-Aided Design, 1990, 748–751.
8. T.Lengaur, Combinatorial Algorithms for Integrated Circuit Layout, Wiley 1990.
9. H. Nagamochi and T. Ibaraki, *A fast algorithm for computing minimum 3-way and 4-way cuts*, Lecture Notes in Comput. Sci., 1610, Springer-Verlag, 7th Conf. on Integer Programming and Combinatorial Optimization, 1999, 377–390.
10. H. Nagamochi, S. Katayama and T. Ibaraki, *A faster algorithm for computing minimum 5-way and 6-way cuts in graphs*, Lecture Notes in Computer Science 1627, Springer-Verlag, 5th Annual Int. Computing and Combinatorics Conf., 1999, 164–173.
11. H. S. Stone, *Multiprocessor scheduling with the aid of network flow algorithms*, IEEE Trans. on Software Engg., SE-3, 1977, 85–93 .
12. H. Saran and V. V. Vazirani, *Finding -cuts within twice the optimal*, In Proc. 32nd IEEE FOCS, 1991, 743–751.

# Online Scheduling of Parallel Communications with Individual Deadlines*

Jae-Ha Lee and Kyung-Yong Chwa

Dept. of Computer Science, KAIST, Korea
{jhlee,kychwa}@jupiter.kaist.ac.kr

**Abstract.** We consider the online competitiveness for scheduling a set of communication jobs (best described in terms of a weighted graph where nodes denote the communication agents and edges denote communication jobs and three weights associated with each edge denote its length, release time, and deadline, respectively), where each node can only send or receive one message at a time. A job is *accepted* if it is scheduled without interruption in the time interval corresponding to its length between release time and deadline. We want to maximize the sum of the length of the accepted jobs. When an algorithm is not able to preempt (i.e., `abort`) jobs in service in order to make room for better jobs, previous lower bound shows that no algorithm can guarantee any constant competitive ratio. We examine a natural variant in which jobs can be aborted and each aborted job can be rescheduled from start (called *restart*). We present simple algorithms under the assumptions on job length: 2-competitive algorithm for unit jobs under the discrete model of time and $(6+4\cdot\sqrt{2}\approx 11.656)$-competitive algorithm for jobs of arbitrary length. These upper bounds are compensated by the lower bounds 1.5, $8-\epsilon$, respectively.

## 1 Introduction

In parallel and distributed environment the following scheduling problem arises: We have a set of communication jobs, best described in terms of a weighted graph. Each node denotes the communication agent, edge denotes communication job, and the weight of an edge denotes the time required for transmission. It is implied that each node can only send or receive one message at a time, but messages between different nodes can overlap; thus, a simultaneous transmission of messages between several node pairs is a matching of the graph. One special case of this scheduling problem is modeled as a bipartite graph. In this case, one node set denotes the senders, the other set the receivers, and a simultaneous transmission of messages is a bipartite matching. This model arises in many applications of switching systems including SS/TDMA network [6,4] and I/O platform (where one side of a bipartite graph is the set of disks and the other side is the set of I/O processors) for scheduling I/O requests [9,8,10].

* Supported in part by KOSEF grant 98-0102-07-01-3.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 383–392, 1999.
© Springer-Verlag Berlin Heidelberg 1999

One interesting variant here is the one in which *jobs have individual release times and deadlines*. The problem is online as we assume that an algorithm has no knowledge about the existence of any job until the release time. A job is *accepted* if it is scheduled without interruption in the time interval between its release time and its deadline. We want to maximize the sum of the length of the accepted jobs. Though jobs, to be accepted, must be scheduled without interruption, jobs can be preempted (i.e., `abort`) to make room for better jobs. Aborted jobs can be rescheduled from the start as if it has been never scheduled (i.e., `restart` in [12]). Note that aborts and restarts are meaningless in off-line scheduling [12]. (More precisely, in traditional off-line scheduling, preemptive jobs can be executed piece by piece and non-preemptive jobs are executed without interruption. In online scheduling, there is an in-between one: jobs can abort but we count only the executions without interruption of jobs.) However, in our problem, the introduction of abort and restart is quite natural, because without abort, no algorithm can guarantee any constant competitive ratio. We call this problem ONLINE REALTIME SCHEDULING, defined formally as follows:

> Given a graph $G = (V, E)$, each job $J_j$ that occurs along an edge $e_j$ is a triple of non-negative real numbers $\langle r_j, p_j, d_j \rangle$, where $r_j$ is the release time or arrival time of the job, $p_j$ is the length of the processing time and $d_j$ is the deadline. We refer to the *expiration time*, $x_j = d_j - p_j$, as the latest time at which a job can be started while still meeting its deadline. We require that two jobs adjacent to the same node cannot be scheduled simultaneously at a time. The gain of a schedule $\sigma$, which we denote as $||\sigma||$, is the sum of the length of the jobs accepted by $\sigma$. A problem instance is given in Figure 1, where each job $J_j$ is denoted by a triple $\langle r_j, p_j, d_j \rangle$ associated with an edge.

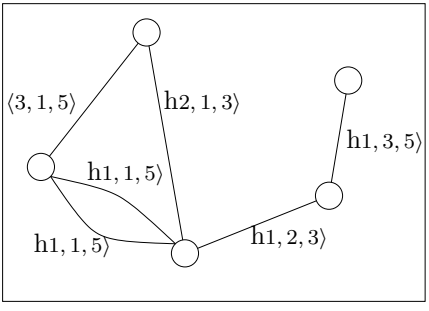



**Fig. 1.** An instance of the ONLINE REALTIME SCHEDULING problem.

One special case of ONLINE REALTIME SCHEDULING is obtained by adopting the discrete model of time. In this case, each job is represented as a triple of non-negative integers and jobs are scheduled at the discrete times. Such a problem naturally arises in the SS/TDMA switching and synchronous I/O

scheduling. Especially, packet switching in ATM networks involves the discrete time model and the messages of unit length, which is discussed in Section 2.

We will measure the performance of an online scheduling algorithm by comparing the gain of the algorithm to the gain of the optimal off-line algorithm *Opt* that knows the entire input jobs in advance. We will say that an online algorithm $A$ is $c$-competitive if $gain_{Opt}(\mathcal{I}) \leq c \cdot gain_A(\mathcal{I})$, for all input instance $\mathcal{I}$. It is easily seen that a natural off-line version of the ONLINE REALTIME SCHEDULING is NP-hard in its most restricted form (proof is omitted in this abstract). Thus any $c$-competitive algorithm is also a $c$-approximation algorithm for an NP-hard problem.

### 1.1 Our Results

We first present a simple greedy algorithm for unit jobs under the discrete model of time and show that it is 2-competitive. Next, we consider jobs of arbitrary length and present a $(6+4\cdot\sqrt{2} \approx 11.656)$-competitive algorithm. Our algorithm repeatedly computes the *maximum matching with small change* that comprises two opposing goals: maximum matching and minimization of aborts of scheduled jobs. This upper bound is compensated by the lower bound $8\text{–}\epsilon$, which is obtained by extending the lower bound 4 in the interval scheduling problem [13].

### 1.2 Related Work

There have been many researches on the problem of scheduling communication jobs *without* deadlines. In general, they adopted the discrete model of time and aimed to minimize the total time required for the completion of all communication jobs. Gopal and Wong [6] formulated this problem, inspired by SS/TDMA switches, and gave an heuristic algorithm to minimize the total switching time in SS/TDMA switches. Recently, Crescenzi, et al. [4] presented 2-approximation algorithms to find a preemptive schedule that minimizes the sum of switching times and the number of switchings and showed the lower bound of the approximation ratio $\frac{7}{6}$. Jain, et al. [9,8,10] studied many variants of this problem raised in I/O platform. However, in these works, neither deadline constraints nor online algorithms were considered.

Our problem is an extension of the interval scheduling problem, which was introduced by Woeginger [13]. In the interval scheduling problem, entire system includes only two communication nodes (thus, every communication jobs occur between the two nodes and at most one communication job can be scheduled at a time) and the expiration time $x_j$ of a job $J_j$ equals the release time $r_j$ (usually termed as a job with *no slack time* $x_j - r_j$). The second restriction implies that each job is either scheduled at its release time or never scheduled. Thus our problem is the one that extends the interval scheduling problem into two directions: arbitrary slack time and conflict constraint [7] in that jobs adjacent with the same node cannot be scheduled simultaneously. As observed in [5],

existence of slack acts as a double-edged sword for competitive analysis. It is because the added flexibility helps not only an online algorithm but also Opt.

Another researches on the interval scheduling problem assume that once a job begins running, the job cannot be aborted (see [11,5]). Recently, variants of online scheduling of realtime communications dealt with unit jobs, focusing on the congestion in linear network, trees, and meshes [2,1] and selection of requests among alternatives [3].

## 2 Unit Jobs Under the Discrete Time Model

In this section, we assume the discrete time model and consider the jobs of unit length. Naturally, our goal is to maximize the number of jobs that are scheduled between their release times and deadlines. The algorithm given here is very simple – it computes a maximum matching in each time slot – and achieves a competitive ratio of 2.

---

`Shelf-based-Max-Matching` (SMM)
  $A = \emptyset$;  { $A$ denotes the set of available jobs, not yet scheduled. }
  **for** $k = 1$ **to** $\infty$
    Insert into $A$ newly-arrived jobs;
    Compute a maximum matching $M$ for jobs in $A$;
    Remove the jobs in $M$ from $A$;
    Remove the jobs whose deadline is $k$ from $A$;

---

**Theorem 1.** *The algorithm SMM is 2-competitive.*

*Proof.* Let $D = \max_j d_j$. For $i, j$ such that $1 \le i \le j \le D$, let $SMM(i, j)$ denote the set of jobs scheduled in time interval $[i, j]$. Analogously, $OPT(i, j)$ denote the set of jobs scheduled by $OPT$. For a set of jobs, $|\cdot|$ denotes its cardinality. Since $SMM(i, i)$ is a maximum matching, its cardinality is best possible if we ignore the decisions in the previous steps. Thus, the following is immediate:

$$|SMM(i,i)| \ge |Opt(i,i) - SMM(1,i-1)|$$

Therefore, by summing both sides,

$$\begin{aligned} |SMM(1,k)| &= \textstyle\sum_{i=1}^{k} |SMM(i,i)| \\ &\ge \textstyle\sum_{i=1}^{k} |Opt(i,i) - SMM(1,i-1)| \\ &= \textstyle\sum_{i=1}^{k} |Opt(i,i)| - \sum_{i=1}^{k} |Opt(i,i) \cap SMM(1,i-1)| \\ &\ge \textstyle\sum_{i=1}^{k} |Opt(i,i)| - \sum_{i=1}^{k} |SMM(i,i)| \end{aligned}$$

The last inequality needs some explanation: For different $i$'s, $Opt(i,i) \cap SMM(1, i-1)$ are disjoint each other, and thus their union is a subset of $SMM(1, k)$, Then

the cardinality of this union, $\sum_{i=1}^{k} |Opt(i,i) \cap SMM(1,i-1)|$, is no larger than that of $SMM(1,k)$, which proves the last inequality. Therefore, we have

$$2 \cdot |SMM(1,D)| \geq |Opt(1,D)|$$

completing the proof. □

**Theorem 2.** *No online algorithm for unit jobs is better than 1.5-competitive.*

*Proof.* Suppose that a job $J_1 = \langle 1,1,2 \rangle$ arises along an edge $(v,w)$ and a job $J_2 = \langle 1,1,2 \rangle$ arises along an edge $(v,w')$ initially. Any online algorithm can schedule only one of $J_1$ and $J_2$ at time 1. Without loss of generality, assume that $J_1$ was scheduled at 1. Then, the adversary gives the next job $J_3 = \langle 2,1,2 \rangle$ along an edge $(v',w')$. At time 2, any online algorithm can schedule only one of $J_2$ and $J_3$. In total, at most two jobs are scheduled. However, for the same jobs, OPT can schedule all of them, which implies that no online algorithm can be better than 1.5-competitive. □

## 3 Jobs of Arbitrary Length

This section considers jobs of arbitrary length and adopts the continuous model of time. We present an 11.656-competitive algorithm and shows no algorithm can be better than $(8-\epsilon)$-competitive. We let $r(J_j), p(J_j), d(J_j)$ denote $r_j, p_j, d_j$.

First we discuss the upper bound. Main challenge in this case, compared with the unit jobs, is that some running job should be aborted to make room for longer jobs. For example, suppose a job $J_i$ with $p(J_i) = 2$ is scheduled at $t$ and a job $J_j$ with $p(J_j) = k$ and $d(J_j) = t + k + 1$ arrives at $t + 1$ and conflicts with $J_i$. In order to be better than $\frac{k}{2}$-competitive, any strategy would abort $J_i$ and schedule $J_j$.

The algorithm given here is similar with that of the previous section – computes a maximum weighted matching whenever a new job arrives or some running job completes its execution. Main modification is to abort running jobs that are in conflict with *'sufficiently long'* jobs. We formally define the *'long'* jobs: Fix a constant $\mathcal{C}$ ($\geq 1$). Let $M$ be a matching at a certain time. If a job $J_e$ is not in $M$, is in conflict with jobs[1] $J_i, J_j$ in $M$ and satisfies $p(J_e) > \mathcal{C} \cdot (p(J_i)+p(J_j))$, then we call $J_e$ an *evicter* of $J_i, J_j$ for $M$. (Of course, in case that $J_e$ is in conflict with only one job, say $J_i$, and $p(J_e) > \mathcal{C} \cdot p(J_i)$, $J_e$ is an *evicter* of $J_i$.) The matching we want to find is not simply a maximum matching but a special matching $M$ such that no evicter exists for $M$. To implement this matching, we first define new weight $w$ of jobs as follows: For each running job $J$ scheduled earlier, let $w(J) = \mathcal{C} \cdot p(J)$. And, for other jobs available, let $w(J) = p(J)$. Next, we compute a maximum weight matching with the jobs in $M$ and $A$ and weights $w$. This modified weight $w$ is used only in this routine.

[1] It is clear that an edge can conflict with *at most* two edges of a matching.

---

`General-Shelf-based-Max-Matching` (GSMM($\mathcal{C}$))
initialize $A=\emptyset$; $F=\emptyset$; $M=\emptyset$;
**for** (each job $J$ that is newly arrived or completed its execution) **do**
Insert into $A$ newly-arrived jobs;
{ Let M be the current matching }
Remove from $M$ completed jobs;
For each job $J$ in $M$, let $w(J) = \mathcal{C} \cdot p(J)$;
For each job $J$ in $A$, let $w(J) = p(J)$;
Compute a new maximum matching $M$ for jobs in $M$ and $A$, using $w$;
Remove the jobs in $M$ from $A$;

---

A job is said to be *accepted* if it is succefully completed; otherwise, *rejected*. Note that some rejected job can be scheduled and aborted several times. For a set $S$ of jobs, we let $p(S)$ denote the sum of the length of the jobs in $S$.

**Theorem 3.** *The algorithm* **GSMM**($\mathcal{C}$) *is* $(\frac{\mathcal{C}+3}{\mathcal{C}-1} + 2\mathcal{C} + 3)$*-competitive.*

*Proof.* Let $S$ be the set of jobs accepted by GSMM($\mathcal{C}$), and let $R$ be the set of jobs accepted by Opt but rejected by GSMM($\mathcal{C}$). We would like to bound $p(R)$ with respect to $p(S)$. Let $R_1 \subseteq R$ be the set of jobs that are (partially) scheduled at least once by GSMM($\mathcal{C}$) and $R_2 \subseteq R$ be the set of jobs that are never scheduled by GSMM($\mathcal{C}$).

We first bound $p(R_1)$. Suppose that GSMM($\mathcal{C}$) schedules a job $e \in R_1$ and later abort it at $t$. We fix an attention to the iteration of **for** loop in GSMM($\mathcal{C}$) at $t$. Then we claim as follows:

Claim Let $R_t$ is the set of jobs aborted at $t$ and $N(R_t)$ be the set of jobs that are scheduled at $t$ and in conflict with some jobs in $R_t$. Then, we have that
- $p(N(R')) \geq \mathcal{C} \cdot p(R')$, for every subset $R' \subseteq R_t$.
- it is possible to divide each job $e \in R_t$ into two fragments $e_1, e_2$ and each job $f \in N(R_t)$ into two fragments $f_1, f_2$ so that $(i)$ $p(e_1) + p(e_2) = p(e)$, and $(ii)$ $p(f_1) + p(f_2) = p(f)$, and $(iii)$ each fragment $e'$ of any job $e \in R_t$ corresponds one-to-one to a fragment $f'$ of some job $f \in N(R_t)$ and satisfy $p(f') \geq \mathcal{C} \cdot p(e')$. We call $f'$ the *evicter* of $e'$.

*Proof of Claim:* Omitted.

Each job may be scheduled and aborted several times and we consider all aborts of each job. We assume $R_1$ is a multiset whose element is (a scheduling of) jobs in $R$. By the claim we can divide each job in $R_1$ into two fragments and satisfy that each fragment of a aborted job corresponds one-to-one to its evicter, which is at least $\mathcal{C}$-times longer. It should be mentioned that specific scheduling time of each fragment is unimportant since we are only interested in the length bound.

Now we define an abort-diagram $D$, represented by a rooted tree whose node set consists of the jobs in $S$ (scheduled by GSMM($\mathcal{C}$)) and the fragments of the

jobs in $R_1$. The root of $D$ is a job $e$ in $S$. If $e$ consists of two fragments (by the procedure of Claim), we assume the root is their union. We construct $D$ by repeatedly applying the following rule. (During the construction, we further divide the fragments, if necessary.) For each fragment $f$ of a job in $R_1$, if some $x$-portion $e'$ of its evicter is in $D$, then we further divide $f$ into its $x$-portion and remainder and add the $x$-portion of $f$ to $D$ as a child of $e'$. (Note that $e'$ is at least $\mathcal{C}$-times longer than the $x$-portion of $f$.) It is easily seen that the diagram $D$ is a tree. See Figure 2.

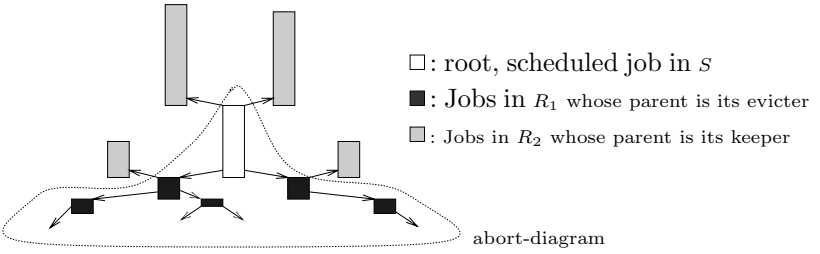



**Fig. 2.** An abort-diagram.

In the same fashion, we can construct all abort-diagrams for each job in $S$. Each job in $R_1$ is divided into a number of fragments, each of which is pushed into some diagrams.

For analyses, let us fix attention to a specific abort-diagram $D$. Suppose that $e$ is the root of $D$. The sum of the length of two children is at most $1/\mathcal{C}$ of that of their parent. Therefore, the sum of the length of all descendents is no larger than $\frac{1/\mathcal{C}}{1-1/\mathcal{C}}$-times $p(e)$. Summation of the above equation over all abort-diagram leads to that

$$p(R_1) \leq \frac{1}{\mathcal{C}-1} p(S)$$

Next, we consider $R_2$, the set of jobs in $R$ that are never scheduled. Recall that jobs in $R$ are scheduled by Opt. Consider a job $e \in R_2$ and suppose that Opt schedules $e$ in time interval $[t, t+t(e_k)]$. Since GSMM($\mathcal{C}$) never schedule $e$, it must be that GSMM($\mathcal{C}$) schedules other jobs, say $e'$ and $e''$, before $t$ that are in conflict with $e$ and keeps them against $e$ at $t$. In other words, $w(e') + w(e'') \geq w(e)$ at $t$.

By taking $\alpha = \frac{p(e')}{p(e')+p(e'')}$ and $\beta = 1-\alpha$, we call $e'$ the $\alpha$-keeper of $e$ and $e''$ the $\beta$-keeper of $e$. In this case, we conceptually divide the job $e$ into two fragments, its $\alpha$ portion and $\beta$ portion. Observe that if a fragment is the $x$-keeper of $e$, then its length is no less than $\frac{x \cdot t(e)}{\mathcal{C}}$. Indeed otherwise, $t(e') + t(e'') < \frac{t(e)}{\mathcal{C}}$. Then, from the definition of $w$, $w(e') + w(e'') < w(e)$, which is a contradiction.

In this way, we divide each job in $R_2$ into two fragments. Then each fragment in $R_2$ has its unique keeper. If a job $e$ is a keeper, then it must be scheduled (possibly aborted later). Thus, each keeper is in $S$ or $R_1$. First suppose that a keeper $e$ is in $S$. Let $K(e)$ be the set of fragments in $R_2$ whose keeper is $e$. Since

Opt can schedule at most two jobs simultaneously that conflict with $e$ and the length of each of them is at most $\mathcal{C}$-times $p(e)$, the sum of the length of these fragments in $K(e)$ is at most $(2\mathcal{C}+2)$-times $t(e)$.

Next, suppose that a keeper $e$ is in $R_1$. Let $K'(e)$ be the set of fragments in $R_2$ whose keeper is $e$. Though this case is similar with $K(e)$, since $e$ is aborted by some $e'$, the sum of the length of these fragments in $K'(e)$ is at most $(\mathcal{C}+2)$-times $t(e)$. (More precisely. let $e=(i,j)$ and $e'=(*,j)$. Jobs in $K(e)$ are of the form $(i,*)$ or $(*,j)$. Latter jobs can conflict with the jobs in $K'(e')$. Therefore, the sum of the length of the jobs in $R_2$ whose keeper is in $R_1$ is at best $(\mathcal{C}+2)$-times the length of their keeper.) Thus we have that

$$p(R_2) \leq (2\mathcal{C}+2)\cdot p(S) + (\mathcal{C}+2)\cdot p(R_1)$$

Therefore,

$$p(R_1)+p(R_2) = [(\mathcal{C}+3)\cdot p(R_1) + (2\mathcal{C}+2)\cdot p(S)$$

$$= [\frac{\mathcal{C}+3}{\mathcal{C}-1} + 2\mathcal{C}+2]\cdot p(S)$$

Formally, let $Opt(R)$ (resp, $Opt(S)$) denote the set of the jobs in $R$ (resp, $S$) that are scheduled by Opt.

$$p(Opt(R)) \leq [\frac{\mathcal{C}+3}{\mathcal{C}-1} + 2\mathcal{C}+2]\cdot p(S)$$

Moreover, since all jobs in $S$ are somehow scheduled by GSMM($\mathcal{C}$), we have that

$$p(Opt(S)) \leq p(S)$$

Therefore,

$$||Opt|| \leq [\frac{\mathcal{C}+3}{\mathcal{C}-1} + 2\mathcal{C}+3]\cdot ||GSMM(\mathcal{C})||$$

completing the proof. □

By optimizing $\mathcal{C}$, the competitive ratio of Theorem 3 becomes $6+4\cdot\sqrt{2}(\approx 11.656)$, when $\mathcal{C}=1+\sqrt{2}$.

**Theorem 4.** *The competitive ratio of the ONLINE REALTIME SCHEDULING problem is at most* $6+4\cdot\sqrt{2}(\approx 11.656)$.

We now turn to the lower bound.

**Theorem 5.** *In the ONLINE REALTIME SCHEDULING problem, the competitive ratio of any algorithm is at least* $8-\epsilon$.

*Proof.* Our proof is an extension of Woeginger's given in [13], which established a lower bound of $4-\epsilon$ for the interval scheduling problem. We assume that all jobs have no slack time, as in [13]. Recall that in the interval scheduling problem, entire system includes only two communication nodes and thus, every communication jobs occur between these two nodes and at most one communication job can be scheduled at one instant. The proof for the interval scheduling problem should be of this form: The adversary generates the intervals $J_1, J_2, \cdots, J_n$. If any online algorithm accepts intervals $J'_1, J'_2, \cdots, J'_{n'}$, then Opt can accept intervals $J''_1, J''_2, \cdots, J''_{n''}$ such that $\sum_{i=1}^{k'} p(J''_i) \geq (4-\epsilon)\sum_{i=1}^{k} p(J'_i)$, where $\epsilon$ is an arbitrary constant.

In the sequel, we modify the proof of [13] to obtain the lower bound of $8-\epsilon$. First, we consider a very simple bipartite graph with node sets $\{1,2\}$ in both sides, instead of two nodes in the interval scheduling. Next, we simulate each interval $J_i$ with three jobs, called *job set*.

*Definition of job sets:* For the interval $J_i$ with the release time $r(J_i)$ and deadline $d(J_i)$ and length $p(J_i)$ (recall that in the interval scheduling problem, $p(J_i) = d(J_i) - r(J_i)$), the adversary generates three jobs $K_i = \{T_{i,1}, T_{i,2}, T_{i,3}\}$ as a unit; let $\delta_i = \frac{\delta}{2^i}$, where $\delta$ is sufficiently small constant.

- The time interval $[r(J_{i,j}), d(T_{i,j})]$ of the job $T_{i,j} \in K_i$ fulfills
  $r(T_{i,j}) = r(J_i) + \frac{j-1}{3}\delta_i$ $(j = 2, 3)$, $\quad d(T_{i,1}) = d(T_{i,2}) = d(T_{i,3}) = d(J_i)$,
  $p(T_{i,j}) = d(T_{i,j}) - r(T_{i,j})$
- The node pairs between which the jobs in $K_i$ occur depend on the behavior of an online algorithm $H$. Suppose that currently scheduled job is located at $(1,1)$ and current time is $r(T_{i,1})$. Then, job $T_{i,1}$ is given at $(1,1)$ and $T_{i,2}$ is given to $(1,2)$ (at $r(T_{i,2})$). If the algorithm $H$ schedules the job $T_{i,2}$, then $T_{i,3}$ is given to $(2,2)$. Otherwise $T_{i,3}$ is given to $(2,1)$. Note that any online algorithm $H$ can execute at most one job at one instance, whereas the off-line algorithm can schedule two jobs.
- *Some properties from the above:* If a job in $K_i$ is of length $v$, then any other job in $K_i$ is at least $v - \frac{2}{3}\delta_i$; and sum of the length of two jobs in $K_i$ is at least $2v - \delta_i$.

The remaining proof is straightforward: If any online algorithm schedules jobs in job sets $K'_1, K'_2, \cdots, K'_{n'}$, then its gain is at most $\sum_{i=1}^{k} p(T'_{i,1})$. If we consider the corresponding interval scheduling problem, some online algorithm schedules the intervals $J'_1, J'_2, \cdots, J'_{n'}$ and Opt can schedule some intervals $J''_1, J''_2, \cdots, J''_{n'}$ that satisfies $\sum_{i=1}^{n''} p(J''_i) \geq (4-\epsilon_1)\sum_{i=1}^{k} p(J'_i)$. For each interval $J''_i$, Opt can schedule at least two jobs in each job set $K''_i$, whose sum is at least $2p(J''_i) - \delta_i$. Thus the gain of Opt is at least $(8-2\epsilon_1)\sum_{i=1}^{k} p(J'_i) - 2\delta$. By letting $\epsilon_1 = \frac{\epsilon}{2}$, we showed that no online algorithm can be better than $(8-\epsilon)$-competitive. □

**Acknowledgement** grateful to Oh-Heum Kwon for motivating this work.

## References

1. Micah Adler, Sanjeev Khanna, Rajmohan Rajaraman, and Adi Rosen. Time-constrained scheduling of weighted packets on trees and meshes. In *Proc. of 11th SPAA*, pages 1–12, 1999.
2. Micah Adler, Ramesh K. Sitaraman, Arnold L. Rosenberg, and Walter Unger. Scheduling time-constrained communication in linear networks. In *Proc. of 10th SPAA*, pages 269–278, 1998.
3. Petra Berenbrink, Marco Riedel, and Christian Scheideler. Simple competitive request scheduling strategies. In *Proc. of 11th SPAA*, pages 33–42, 1999.
4. P. Crescenzi, X. Deng, and Ch. Papadimitriou. On approximating a scheduling problem. In *Proc. of Approx98*, 1998.
5. M. Goldwasser. Patience is a virtue: the effect of slack on competitiveness for admission control. In *Proc. of 10th ACM-SIAM SODA*, pages 396–405, 1999.
6. I.S. Gopal and C.K. Wong. Minimizing the number of switchings in an ss/tdma system. *IEEE Trans. on Communications*, 33:497–501, 1985.
7. S. Irani and V. Leung. Scheduling with conflicts and applications to traffic signal control. In *Proc. of 7th ACM-SIAM SODA*, pages 85–94, 1996.
8. R. Jain. *Scheduling data transfers in parallel computer and communication systems.* PhD thesis, Department of Computer Science, University of Texas Austin, 1992.
9. R. Jain, K. Somalwar, J. Werth, and J.C. Browne. Scheduling parallel i/o operations in multiple bus system. *J. of parallel and distributed computing*, 16:352–362, 1992.
10. R. Jain, K. Somalwar, J. Werth, and J.C. Browne. Heuristics for scheduling i/o operations. *IEEE Trans. on Parallel and Distributed Systems*, 8(3), 1997.
11. R.J. Lipton and A. Tomkins. Online interval scheduling. In *Proc. of 5-th ACM SODA*, pages 302–305, 1994.
12. J. Sgall. Online scheduling. *In Online Algorithms: The State of the Art, eds. A. Fiat and G. J. Woeginger, Lecture Notes in Comput. Sci. 1442, Springer Verlag*, pages 196–231, 1998.
13. G.J. Woeginger. On-line scheduling of jobs with fixed start and end times. *Theoretical Computer Science*, 130:5–16, 1994.

# A Faster Algorithm for Finding Disjoint Paths in Grids*

Wun-Tat Chan, Francis Y.L. Chin, and Hing-Fung Ting

Department of Computer Science and Information Systems
The University of Hong Kong, Pokfulam Road, Hong Kong
{wtchan,chin,hfting}@csis.hku.hk

**Abstract.** Given a set of sources and a set of sinks in the two dimensional grid of size $n$, the *disjoint paths (DP) problem* is to connect every source to a distinct sink by a set of *edge-disjoint paths*. Let $v$ be the total number of sources and sinks. In [3], Chan and Chin showed that without loss of generality we can assume $v \leq n \leq 4v^2$. They also showed how to compress the grid optimally to a *dynamic network* (structure of the network may change depending on the paths found currently) of size $O(\sqrt{nv})$, and solve the problem in $O(\sqrt{n}v^{3/2})$ time using augmenting path method in maximum flow. In this paper, we improve the time complexity of solving the DP problem to $O(n^{3/4}v^{3/4})$. The factor of improvement is as large as $\sqrt{v}$ when $n$ is $\Theta(v)$, while it is at least $\sqrt[4]{v}$ for $n$ is $\Theta(v^2)$.

## 1 Introduction

Given a set of sources $S$ and a set of sinks $T$ in the two dimensional grid, the *disjoint paths (DP) problem* is to connect every source in $S$ to a distinct sink in $T$ (assuming $|S| \leq |T|$) by a set of *edge-disjoint paths*. Note that a source can be connected to any sink. Suppose $|S| + |T| = v$. It has been shown in [3] that the size of the grid is bounded by $n$ for $v \leq n \leq 4v^2$ after an $O(v)$ time preprocessing.

In practice, the DP problem is a generalization of the *breakout routing problem* [8] in printed circuit board and the *single-layer routing problem* for pin/ball grid array packages [14, 15]. When $T$ is the set of all boundary vertices of the grid, this problem is known as the *escape problem* [4] or the *reconfiguration problem* [12, 11, 2, 1, 13, 10] on VLSI/WSI processor arrays in the presence of faulty processors.

To find the set of edge-disjoint paths, we usually reduce the DP problem to the maximum flow problem in a unit capacity network. The network can be constructed by adding to the grid a super-source $s$ and a super-sink $t$ which connect to all the sources in $S$ and all the sinks in $T$ respectively. In [3], Chan and Chin observe the sparseness of the $v$ sources and sinks in the grid of size $n$, and then introduce a "compression" technique which compresses the network of

* The research is partially supported by a Hong Kong RGC grant 338/065/0022.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 393–402, 1999.
© Springer-Verlag Berlin Heidelberg 1999

size $\Theta(n)$ to a *dynamic network* of size $\Theta(\sqrt{nv})$. The compression is initiated by deleting a set of disjoint and empty rectangles (i.e., source and sink free). (See Figure 1.) Each $p \times q$ empty rectangle in the set is then replaced with a *reachability graph* of size $O(p+q)$ to restore the connectivity of the network. Basically, in the resulting network the algorithm finds a maximum flow which gives a solution to the DP problem. The augmenting paths method [6] is used as the framework of the algorithm. In each iteration, an augmenting path (an $s$-$t$ path in the residual network) is found and the affected reachability graphs are updated. (See Figure 2.) The algorithm finds the maximum flow in $O(v)$ iterations. The running time of the algorithm can be divided into two categories. The first category is the time to update the reachability graphs. In an iterations, all the reachability graphs may need to be updated for the worst case. Since the total number of vertices and edges in the reachability graphs is bounded by the network size, which is $O(\sqrt{nv})$, the total update time is $O(\sqrt{n}v^{3/2})$ in the $O(v)$ iterations. The second category is the time to find augmenting paths. Since the upper bound of time to find an augmenting path is the size of the network, it takes $O(\sqrt{n}v^{3/2})$ time to find the $O(v)$ augmenting paths. Therefore, the running time of the augmenting paths method for the DP problem is $O(\sqrt{n}v^{3/2})$† [3].

In this paper, we improve the time for both the update of reachability graphs and the search of augmenting paths. First, we observe that if the flow is augmented along the shortest $s$-$t$ paths, we can bound the total number of updates for the reachability graphs to $O(\sqrt{nv}\log v)$. Also, we need to limit the size of individual reachability graph. For this reason, we give a new procedure which isolates a set of disjoint and empty rectangles each with size at most $\sqrt{n/v} \times \sqrt{n/v}$. Thus each reachability graph has size $O(\sqrt{n/v})$, and the total update time for the reachability graphs is reduced from $O(\sqrt{n}v^{3/2})$ to $O(\sqrt{n/v}\sqrt{nv}\log v)$, i.e., $O(n \log v)$. (It is an improvement because $n \log v < \sqrt{n}v^{3/2}$ for $v \leq n \leq 4v^2$.) On the other hand, we extend the traditional layered network technique to our dynamic network for finding augmenting paths. In the traditional layered network approach, we repeatedly add a blocking flow in the layered network to the current flow until we find the maximum flow. Owing to the unit capacity network, the blocking flow in the layered network can be considered as a maximal set $P$ of disjoint and shortest $s$-$t$ paths in the residual network. The layered network approach is efficient in unit capacity network because the path set $P$ can be found and the flow can be augmented along all paths in $P$ in $O(m)$ time where $m$ is the number of edges in the network. Also, we can yield the maximum flow after $O(\sqrt{m})$ rounds, i.e., in $O(m\sqrt{m})$ time. However, since our network is dynamic, i.e., the number of vertices (in the reachability graphs) may change after augmenting a flow along any path in $P$, the same layered network cannot be used to find more than one augmenting path. Thus, we cannot directly apply the layered network technique and improve the $O(\sqrt{n}v^{3/2})$ running time

† The result given in [3] is expressed in $O(d\sqrt{mnN})$ where $d$ is the maximum flow and $\sqrt{mnN}$ is the size of the dynamic network. In this paper, for the ease of comparison, the same formula is rewritten as $\sqrt{n}v^{3/2}$ where $v$ is the maximum flow and $\sqrt{nv}$ is the size of the dynamic network.

immediately. Later in this paper, we will show how to build a different layered network which may contain some dynamic components. This layered network is built using a different measurement of distance between two vertices. In spite of the dynamic nature in the layered network, we can find the path set $P$ (under the new distance measure) in the same layered network with some modification in $O(\sqrt{nv})$ time. Moreover, we can repeat this process to get the maximum flow in $O(\sqrt[4]{nv})$ rounds. As a result, we improve the time complexity of solving the DP problem to $O(n^{3/4}v^{3/4})$. It is faster than the previous $O(\sqrt{n}v^{3/2})$ result by a factor of $\sqrt[4]{v^3/n}$. This factor is as large as $\sqrt{v}$ when $n$ is $\Theta(v)$, while it is at least $\sqrt[4]{v}$ for $n$ is $\Theta(v^2)$.

Using the techniques presented in this paper, we can also improve the time complexity of the algorithm in solving the DP problem with vertex-disjoint paths [3] to $O(n^{3/4}v^{3/4})$. Since our procedure in isolating the disjoint and empty rectangles is simple and efficient, our algorithm should perform well in practice (especially if the push-relabel algorithm [7] is implemented instead of the layered network approach). On the other hand, there are two reasons showing that it is difficult to further improve the $O(n^{3/4}v^{3/4})$ time complexity. For one reason, since our compression of network from $O(n)$ to $O(\sqrt{nv})$ is optimal [3], it is not possible to apply our extended layered network technique to the DP problem with a smaller network. For another reason, we know that the layered network algorithm which runs in $O(m\sqrt{m})$ time is still the fastest known algorithm for finding maximum flow in unit capacity and sparse network. (See [9] for more details.) Considering that our dynamic network is also a sparse network where $m$ is $O(\sqrt{nv})$, we believe that further improvement on the $O(m\sqrt{m})$ or $O(n^{3/4}v^{3/4})$ bound on the DP problem may require new insights in the grid structure or new techniques in finding maximum flow in sparse networks.

## 2 Preliminaries

### 2.1 Layered Network and Partial Flow

Let $G = (V, E)$ be a network with $s, t \in V$ and each edge has unit capacity. A flow in $G$ can be regarded as a set of edge-disjoint $s$-$t$ paths. Let $G_f$ be the residual network [4] induced by a flow $f$. Define $\delta_f(u)$ to be the shortest distance[‡] from $s$ to $u$ in $G_f$. A *layered network* is a subgraph of $G_f$ containing only the vertices reachable from $s$ and only those edges $(u, w)$ such that $\delta_f(w) = \delta_f(u) + 1$. A *blocking flow* $f$ is a flow in $G$ such that every $s$-$t$ path in the network contains an edge $(u, w)$ with $f(u, w) = 1$.

Let $C$ be a set of disjoint and empty rectangles in the grid. Note that no two rectangles in $C$ share the same vertex. An *uncovered edge* is a grid edge outside all rectangles (we denote "rectangle" for an empty rectangle) in $C$. Both end vertices of an uncovered edge are the *uncovered vertices*. A *partial flow* is a flow defined only on the uncovered edges but not on the edges inside any rectangle in $C$ (Figure 1). Since a rectangle does not contain any source or sink, the partial

[‡] Each edge is assumed unit distance.

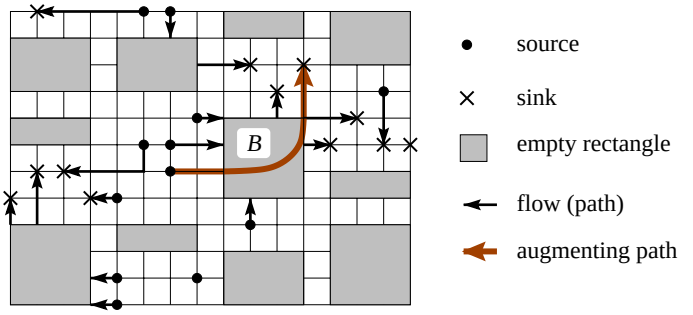



**Fig. 1.** Example of an isolation of disjoint empty rectangles, a partial flow and an augmenting path.

flow $\tilde{f}$ at every rectangle should conserved, i.e., the net flow into every rectangle is zero.

Given a rectangle, an h-cut $hc_i$ denotes the set of edges connecting between the $i$th row of vertices and the $(i+1)$th row of vertices in the rectangle. A v-cut $vc_i$ denotes the set of edges connecting between the $i$th column of vertices and the $(i+1)$th column of vertices. The *capacity* of $hc_i$ is $|hc_i|$, and it equals the width of the rectangle. Given a partial flow $\tilde{f}$, the *demand* of $hc_i$ is defined to be the absolute value of the net flow into the rectangle on or above the $i$th row. For example in Figure 2(a), the net flow across $hc_1$ is 1. The capacity and demand of a v-cut are defined similarly. An h-cut or v-cut is *saturated* if its demand equals its capacity and *oversaturated* if its demand is larger than its capacity.

In order for a partial flow $\tilde{f}$ to be developed into a "complete" flow by having the flow values (edge-disjoint paths) defined inside all rectangles, each rectangle should satisfy the following necessary and sufficient conditions.

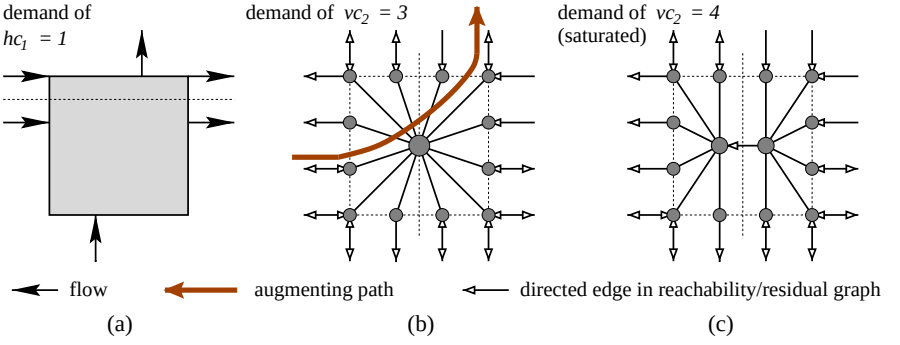



**Fig. 2.** A step of augmentation and update in the $4 \times 4$ empty rectangle $B$ in Figure 1. (a) There are 3 units of flow entering the rectangle and 3 units of flow leaving the rectangle. (b) An augmenting path passes through the corresponding reachability graph. (c) Update to a new reachability graph.

**Lemma 1 ( [3] ).** *Given a $p \times q$ rectangle and a partial flow $\tilde{f}$, there exists a set of edge-disjoint paths inside the rectangle matching the flow entering and leaving the rectangle if and only if all the $hc_i$ for $1 \leq i \leq p-1$ and $vc_j$ for $1 \leq j \leq q-1$ are not oversaturated.*

### 2.2 The Reachability Graphs in Residual Networks

For a rectangle, the corresponding reachability graph retains all of the rectangle boundary vertices for connection with the grid vertices outside the rectangle and maintains a set of *internal vertices* to represent the connections between the boundary vertices. The characteristic of a reachability graph is that whenever a boundary vertex $\beta$ is connected to another boundary vertex $\beta'$, the flow can be augmented from $\beta$ to $\beta'$ through the rectangle such that none of the h-cuts nor v-cuts is oversaturated. The structure of a reachability graph depends on how the partial flow $\tilde{f}$ passes through the rectangle and eventually depends on the existence, quantity and location of the saturated cuts in the rectangle. There are three different kinds of structures for the reachability graphs, whose sizes are all linear to the number of boundary vertices (Figure 3).

**Case (a). Neither of the h-cut nor v-cut is saturated (Figure 3(a)):** The reachability graph is a star graph which has a path between any pair of boundary vertices.

**Case (b). Either some h-cuts or v-cuts (but not both) are saturated (Figure 3(b)):** The rectangle is partitioned into rectangular components by the saturated cuts. The part of the reachability graph representing a component is a star graph same as that in Case (a). The direction of an edge between two star graphs is opposite to that of the saturated cuts between the corresponding components. It forces an augmenting path to go only from one boundary vertex to another in the opposite direction of the saturated cuts.

**Case (c). Both saturated h-cuts and v-cuts exist (Figure 3(c)):** The rectangle is again partitioned into components by the saturated cuts. In this case, only the boundary components will be represented by a star graphs same as that in Case (a). Similar to the edges in Case (b), an edge between two star graphs has the direction opposing the corresponding saturated cut.

## 3 The Algorithm for Edge-Disjoint Paths

The algorithm starts by isolating a set of rectangles such that the number of uncovered vertices and edges are $O(\sqrt{nv})$.

### 3.1 Isolation of Rectangles

In the isolation procedure presented in [3], it might output rectangles of size $\sqrt{n} \times c$ for some constant $c$ which dominates the total update time of the reachability graphs. In this section, we present another algorithm which isolates the rectangles so that the size of each rectangle output is at most $\sqrt{n/v} \times \sqrt{n/v}$.

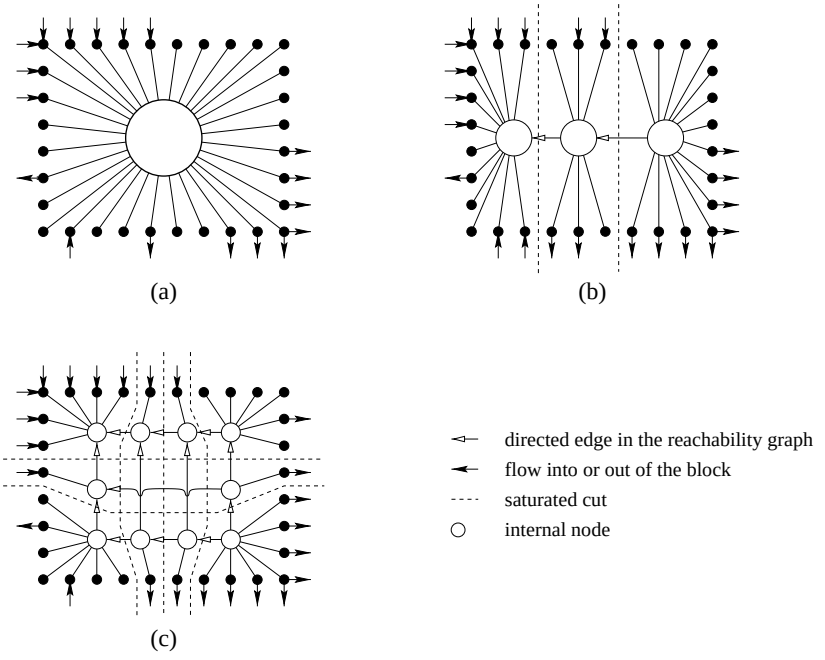



**Fig. 3.** The reachability graphs for Cases (a), (b) and (c)

Without loss of generality, assume the grid is $n_1$ by $n_2$ (i.e., $n = n_1 n_2$), and $n_1$ and $n_2$ are multiples of $\sqrt{n/v}$. Algorithm 1 presents the procedure which isolates a set of rectangles such that the number of uncovered vertices and edges is $O(\sqrt{nv})$. An example using the procedure to isolate rectangles is shown in Figure 1. Now we analyze the performance of the procedure. Firstly, the length of each side of the output rectangles is less than or equal to $\sqrt{n/v}$. Secondly, we realize that the uncovered edges are the edges removed in the procedure. As we have removed $\sqrt{n_1 v/n_2} - 1$ h-cuts of $n_2$ edges each, $\sqrt{n_2 v/n_1} - 1$ v-cuts of $n_1$ edges each, and at most $O(v)$ rows of edges (the rows that contains sources or the rectangles in $C$ that contains only one row) in the $\sqrt{n/v} \times \sqrt{n/v}$ rectangles, the number of uncovered edges is $O(\sqrt{nv})$. Since the number of uncovered vertices is linearly proportional to the number of uncovered edges, the number of uncovered vertices is also $O(\sqrt{nv})$.

### 3.2 The Algorithm

Denote $G_{\tilde{f}}$ the dynamic network induced by the partial flow $\tilde{f}$ (where each rectangle in the grid is replaced by a corresponding reachability graph). Since the structures of some reachability graphs may change drastically after augmenting merely one $s$-$t$ path, we may not find more than one augmenting path in the same $G_{\tilde{f}}$. Therefore, if we build a layered network $L$ from $G_{\tilde{f}}$ based on the conventional distance measure, $L$ may have to be rebuilt after adding only one path. To

Procedure: ISOLATE$(S, T)$
Input: an $n_1 \times n_2$ grid, a set of sources $S$ and a set of sinks $T$.
Output: a set of empty rectangles $C$.

Let $k = \sqrt{n/v}$;
Partition the grid into a set of $k \times k$ rectangles $C$ by removing
(1) the edges in the h-cuts $hc_k$, $hc_{2k}$, ..., $hc_{n_1-k}$ and
(2) the edges in the v-cuts $vc_k$, $vc_{2k}$, ..., $vc_{n_2-k}$;
**foreach** *rectangle* $R \in C$ **do**
  **if** $R$ *contains some sources or sinks* **then**
    Partition $R$ by removing the rows of vertices (and edges incident to the vertices) that contain the sources or sinks in $R$;
Remove the rectangles in $C$ that contain only one row of vertices;

**Algorithm 1:** ISOLATE$(S, T)$

improve the time complexity, we *define a proper distance measure* so that we can reuse (major part of) the layered network after adding a path. Let $p$ be a path between two uncovered vertices in $G_{\tilde{f}}$. Define the length of $p$ to be the number of uncovered edges in $p$ plus the number of rectangles that $p$ passes through. The distance of an uncovered vertices $w$ from $s$, denoted by $\delta_{\tilde{f}}(w)$, is the length of the shortest path from $s$ to $w$ in the new definition. The new layered network $L$ contains all uncovered vertices reachable from $s$ and the uncovered edges $(u, w)$ for $\delta_{\tilde{f}}(w) = \delta_{\tilde{f}}(u) + 1$. Moreover, $L$ also contains the path $h = (u_1, u_2, \ldots, u_k)$ of a reachability graph if $u_1, u_k$ are two boundary vertices of the corresponding rectangle with $\delta_{\tilde{f}}(u_k) = \delta_{\tilde{f}}(u_1) + 1$, and $u_2, u_3, \ldots, u_{k-1}$ are the internal vertices of the reachability graph. After $L$ is constructed, we find a blocking flow in $L$ (by NEW-BLOCKING in Algorithm 2) similar to the conventional method but we have to update (reconstruct) the affected reachability graphs once a flow is augmented.

Procedure: NEW-BLOCKING$(L)$
Input: the layered network $L$ defined above.

**repeat**
  Find the shortest $s$-$t$ path $p$ in $L$ based on the new distance measure;
  Augment the flow along $p$;
  Update (reconstruct) every reachability graph that $p$ passes through;
  Update $L$ by the new reachability graphs;
**until** $s$ *and* $t$ *in* $L$ *are disconnected*;

**Algorithm 2:** NEW-BLOCKING$(L)$

We need some new definitions for the correctness proof below. Given a path $h = (u_1, u_2, \ldots, u_k)$ in a reachability graph of $G_{\tilde{f}}$, suppose $u_1, u_k$ are two bound-

ary vertices of a rectangle and all $u_2, u_3, \ldots, u_{k-1}$ are the internal vertices in the corresponding reachability graph. We consider $(u_1, u_k)$ (and $h$) as a *virtual edge* of $G_{\tilde{f}}$. Also, we redefine a *blocking flow* $\tilde{f}'$ in $L$ as a partial flow such that every shortest $s$-$t$ path in $L$ contains either an uncovered edge $(u, w)$ with $\tilde{f}'(u, w) = 1$ or a virtual edge $(u, w)$ such that if $\tilde{f}'$ is added to the current flow, one of the h-cuts or v-cuts in the rectangle is saturated from $u$ to $w$.

Lemma 2 shows that the distance of all uncovered vertices is nondecreasing if current flow is augmented along one of the shortest paths in $G_{\tilde{f}}$. This enables the layered network to be used to find more than one augmenting paths. Lemma 3 argues that the distance of $t$ is strictly increasing in each round of finding blocking flow. Let $\tilde{f} + \tilde{f}'$ be the flow after adding $\tilde{f}'$ to $\tilde{f}$.

**Lemma 2.** *Let $\tilde{f}'$ be a partial flow along one of the shortest $s$-$t$ paths in $G_{\tilde{f}}$. We have $\delta_{\tilde{f}}(u) \leq \delta_{\tilde{f}+\tilde{f}'}(u)$ for all vertices $u$ reachable by $s$.*

*Proof.* Let $p$ be one of the shortest $s$-$t$ paths in $G_{\tilde{f}}$. Obviously, each edge (uncovered or virtual) $(u, w)$ in $G_{\tilde{f}}$ has $\delta_{\tilde{f}}(w) \leq \delta_{\tilde{f}}(u) + 1$. We shall show that each edge $(u, w)$ in $G_{\tilde{f}+\tilde{f}'}$ also has $\delta_{\tilde{f}}(w) \leq \delta_{\tilde{f}}(u) + 1$. Each uncovered edge in $G_{\tilde{f}+\tilde{f}'}$ is either an edge in $G_{\tilde{f}}$ or is the reverse of an edge in $p$. For each virtual edge $(u, w)$ in rectangle $R$ of $G_{\tilde{f}+\tilde{f}'}$, either $(u, w)$ is a virtual edge in $G_{\tilde{f}}$, or there were saturated cuts in $R$ of $G_{\tilde{f}}$ in the direction from $u$ to $w$ and $\tilde{f}'$ is augmented against all those cuts. For instance, assume $p$ contains a virtual edge $(w', u')$ in $R$ opposite to the direction of the only saturated cut which is from $u$ to $w$. There must also exist another two virtual edges $(w', w)$ and $(u, u')$ in $R$ of $G_{\tilde{f}}$, and thus $\delta_{\tilde{f}}(w) \leq \delta_{\tilde{f}}(w') + 1 = \delta_{\tilde{f}}(u') \leq \delta_{\tilde{f}}(u) + 1$. The situation can be generalized for more saturated cuts between $u$ and $w$.

Since $\delta_{\tilde{f}}(s) = \delta_{\tilde{f}+\tilde{f}'}(s) = 0$, we can deduce by a simple induction that $\delta_{\tilde{f}}(u) \leq \delta_{\tilde{f}+\tilde{f}'}(u)$ for all vertices $u$ reachable by $s$. □

**Lemma 3.** *Let $\tilde{f}'$ be a blocking flow in the layered network $L$ of $G_{\tilde{f}}$. We have $\delta_{\tilde{f}}(t) < \delta_{\tilde{f}+\tilde{f}'}(t)$.*

*Proof.* Since $\tilde{f}'$ is a partial flow on the edges of some of the shortest $s$-$t$ paths, we have (by Lemma 2) $\delta_{\tilde{f}}(u) \leq \delta_{\tilde{f}+\tilde{f}'}(u)$ for all vertices $u$ reachable by $s$. For instance, $\delta_{\tilde{f}}(t) \leq \delta_{\tilde{f}+\tilde{f}'}(t)$. Suppose $\delta_{\tilde{f}}(t) = \delta_{\tilde{f}+\tilde{f}'}(t)$. Let $p$ be a shortest $s$-$t$ path in $G_{\tilde{f}+\tilde{f}'}$. We have for all edge $(u, w)$ in $p$ that $\delta_{\tilde{f}+\tilde{f}'}(u) + 1 = \delta_{\tilde{f}+\tilde{f}'}(w)$ and hence $\delta_{\tilde{f}}(u) + 1 = \delta_{\tilde{f}}(w)$. That means $p$ is in $L$ and it contradicts to the property that at least one of the uncovered or virtual edges in $p$ do not appear in $G_{\tilde{f}+\tilde{f}'}$. Hence $\delta_{\tilde{f}}(t) < \delta_{\tilde{f}+\tilde{f}'}(t)$. □

By Lemma 3, we can bound the number of rounds in the algorithm by a similar approach as given in [5].

**Theorem 1.** *The algorithm terminates in at most $O(\sqrt[4]{nv})$ rounds.*

*Proof.* Consider after the first $\sqrt[4]{nv}$ rounds The shortest $s$-$t$ distance is at least $\sqrt[4]{nv}$. As the size of $G_{\tilde{f}}$ is $O(\sqrt{nv})$, there are at most $O(\sqrt{nv}/\sqrt[4]{nv})$, i.e., $O(\sqrt[4]{nv})$ edge-disjoint paths in $G_{\tilde{f}}$. Thus, there are at most $O(\sqrt[4]{nv})$ rounds left. □

## 4 Time Analysis

The following lemma reveals the running time of the algorithm in a round.

**Lemma 4.** *Procedure* NEW-BLOCKING *finds and augments a blocking flow in* $O(\sqrt{nv})+O(\alpha b)$ *time where* $\alpha$ *and* $b$ *are the number and sizes of the reachability graphs that have been reconstructed in the round respectively.*

*Proof.* Each uncovered edge is traversed once and then removed because of backtracking or saturation. After traversing a virtual edge, the algorithm either backtracks from that edge, or augments flow through the rectangle whose reachability graph will then need to be reconstructed. Thus, the time spent on the uncovered edges and the edges in the original reachability graphs is $O(\sqrt{nv})$, while the time spent on the edges in the reconstructed reachability graphs is $O(\alpha b)$. □

The following lemma and corollary bound the value of $\sum \alpha$ over all rounds in the algorithm.

**Lemma 5.** *The sum of path length of all the augmenting paths found in the algorithm is* $O(\sqrt{nv}\log v)$.

*Proof.* Let $e_i$ and $e^*$ be the number of augmenting paths (i.e., shortest edge-disjoint $s$-$t$ paths) found in the $i$th round and all rounds in the algorithm respectively. For instance in the first round, we find $e_1$ augmenting paths and each of them has length at most $\sqrt{nv}/e^*$. In the $i$th round, we find $e_i$ augmenting paths and each of them has length at most $\sqrt{nv}/(e^* - \sum_{j<i} e_j)$. Therefore the sum of path length is

$$\begin{aligned}\sum_{i\geq 1} \frac{e_i\sqrt{nv}}{e^* - \sum_{j<i} e_j} &= \sqrt{nv}\left(\underbrace{\frac{1}{e^*}+\cdots+\frac{1}{e^*}}_{e_1} + \underbrace{\frac{1}{e^*-e_1}+\cdots+\frac{1}{e^*-e_1}}_{e_2}+\cdots\right)\\ &\leq \sqrt{nv}\left(\frac{1}{e^*}+\frac{1}{e^*-1}+\frac{1}{e^*-2}+\cdots\right)\\ &\leq \sqrt{nv}\log e^* \leq \sqrt{nv}\log v.\end{aligned}$$

□

**Corollary 1.** *At most* $O(\sqrt{nv}\log v)$ *reachability graphs are reconstructed in the algorithm.*

*Proof.* Flow is augmented along at most $O(\sqrt{nv}\log v)$ virtual edges. □

**Lemma 6.** *The algorithm takes* $O(n^{3/4}v^{3/4} + b\sqrt{nv}\log v)$ *time to find the maximum partial flow.*

*Proof.* The running time of the algorithm includes the time to (1) isolate the rectangles ($O(n)$ time), (2) construct the $O(\sqrt[4]{nv})$ layered networks (Theorem 1) and find the blocking flows in each layered network, and (3) construct the edge-disjoint paths inside each rectangle after all rounds ($O(n)$ time). In the worst case, the running time is dominated by Step (2) which is $O(n^{3/4}v^{3/4} + b\sqrt{nv}\log v)$. □

If the size of a rectangle is $\sqrt{n} \times c$ (for some constants $c$) as given in [3], $b$ will be $O(\sqrt{n})$. Thus, no improvement on the time complexity can be achieved. However, with our new algorithm to isolate the rectangles in grid (Section 3.1), the size of each rectangle is at most $\sqrt{n/v} \times \sqrt{n/v}$, i.e., $b = O(\sqrt{n/v})$. In conclusion, we have the following theorem.

**Theorem 2.** *The algorithm solves the DP problem in* $O(n^{3/4}v^{3/4})$ *time.*

## References


1. Y. Birk and J. B. Lotspiech. On finding non-intersecting straightline connections of grid points to the boundary. *J. Algorithms*, 13(4):636–656, Dec. 1992.
2. J. Bruck and V. P. Roychowdhury. How to play bowling in parallel on the grid. *J. Algorithms*, 12(3):516–529, Sept. 1991.
3. W.-T. Chan and F. Y. L. Chin. Efficient algorithms for finding maximum number of disjoint paths in grids. *J. Algorithms*, To appear. (A preliminary version of this paper appeared in SODA'97.).
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1991.
5. S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, Dec. 1975.
6. L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.
7. A. V. Goldberg and R. Kennedy. Global price updates help. *SIAM J. Discrete Math.*, 10:551–572, 1997.
8. J. Hershberger and S. Suri. Efficient breakout routing in printed circuit boards. In *Algorithms and Data Structures, 5th International Workshop*, Lecture Notes in Computer Science, pages 462–471. Springer, 1997.
9. D. R. Karger and M. S. Levine. Finding maximum flows in undirected graphs seems easier than bipartite matching. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, 1998.
10. L. Palios. Connecting the maximum number of nodes in the grid to the boundary with nonintersecting line segments. *J. Algorithms*, 22(1):57–92, Jan. 1997.
11. V. P. Roychowdhury and J. Bruck. On finding non-intersecting paths in grids and its application in reconfiguring VLSI/WSI arrays. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 454–464, 1990.
12. V. P. Roychowdhury, J. Bruck, and T. Kailath. Efficient algorithms for reconfiguration in VLSI/WSI arrays. *IEEE Trans. Comput.*, 39(4):480–489, Apr. 1990.
13. T. A. Varvarigou, V. P. Roychowdhury, and T. Kailath. Reconfigurating processor arrays using multiple-track models: The 3-track-1-spare-approach. *IEEE Trans. Comput.*, 42(11):1281–1292, Nov. 1993.
14. M.-F. Yu and W. W.-M. Dai. Pin assignment and routing on a single-layer pin grid array. In *Proceedings of 1st Asia and South Pacific Design Automation Conference*, pages 203–208. IEEE, 1995.
15. M.-F. Yu and W. W.-M. Dai. Single-layer fanout routing and routability analysis for ball grid arrays. In *Proceedings of IEEE/ACM International Conference on Computer-aided Design*, pages 581–586, 1995.

# Output-Sensitive Algorithms for Uniform Partitions of Points*

Pankaj K. Agarwal[1], Binay K. Bhattacharya[2], and Sandeep Sen[3]

[1] Center for Geometric Computing, Department of Computer Science, Box 90129, Duke University, Durham, NC 27708-0129, USA.
pankaj@cs.duke.edu
[2] School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada.
binay@cs.sfu.ca
[3] Department of Computer Science and Engineering, IIT Delhi, New Delhi 110016, India.
ssen@cse.iitd.ernet.in

**Abstract.** We consider the following one- and two-dimensional bucketing problems: Given a set $S$ of $n$ points in $\mathbb{R}^1$ or $\mathbb{R}^2$ and a positive integer $b$, distribute the points of $S$ into $b$ equal-size buckets so that the maximum number of points in a bucket is minimized. Suppose at most $(n/b) + \Delta$ points lies in each bucket in an optimal solution. We present algorithms whose time complexities depend on $b$ and $\Delta$. No prior knowledge of $\Delta$ is necessary for our algorithms.
For the one-dimensional problem, we give a deterministic algorithm that achieves a running time of $O(b^4\Delta^2 \log n + n)$. For the two-dimensional problem, we present a Monte-Carlo algorithm that runs in sub-quadratic time for certain values of $b$ and $\Delta$. The previous algorithms, by Asano and Tokuyama [1], searched the entire parameterized space and required $\Omega(n^2)$ time in the worst case even for constant values of $b$ and $\Delta$.

## 1 Introduction

We consider geometric optimization problems that do not seem to have any nice properties like convexity and have a large number of distinct global optimal solutions. Consequently, it is hard to develop a search strategy that will avoid looking at all the optimum solutions (or more likely near-optimal solutions). However, if the number of optimal solutions are few, we may be able to prune the search-space. This may lead to more efficient algorithms that are "output-sensitive" where the notion of output is related to the number of optimal solutions. Since

* Work by the first author was supported by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants EIA–9870724, and CCR–9732787, by an NYI award, and by a grant from the U.S.-Israeli Binational Science Foundation. Work by the second author was supported by an NSERC grant. Part of this work was done while the last two authors were visiting Department of Computer Science, University of Newcastle, Australia.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 403–414, 1999.
© Springer-Verlag Berlin Heidelberg 1999

we do not know the optimum solution to begin with, we can try to estimate that by some means, say random-sampling, and then use that to prune the search space. The success of such an approach depends on how effectively we can prune the search space.

In this paper we consider the problem of partitioning a set of points in $\mathbb{R}^1$ or $\mathbb{R}^2$ into equal-size buckets, so that the maximum number of points in a bucket is minimized. The first problem that we consider is the following: Given a set $S$ of $n$ real numbers and an integer $1 \leq b \leq n$, partition $S$ uniformly into $b$ equal sized buckets, i.e., each bucket has the same width. The buckets are defined by real numbers $\beta_i = L + i \cdot w$, for $0 \leq i \leq b$ where $L$ is the left endpoint of the left-most bucket and $w$ is the width (size) of the buckets. The $i$-th bucket $B_i$ is defined by the interval $[\beta_i, \beta_{i+1})$ and $S \cap B_i$ is the *content* of the $i$-th bucket (for a fixed choice of $L$ and $w$). We wish to minimize the maximum size of the contents in buckets. Two version of this problem are studied: (i) the *tight* case in which $B_1$ and $B_b$ are required to be nonempty, and (ii) the *relaxed* case in which they are allowed to be empty.

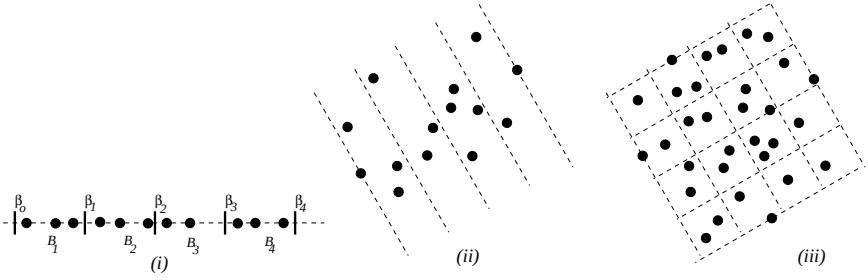



**Fig. 1.** (i) One-dimensional bucketing problem; (ii) uniform-projection problem; (iii) two-dimensional partitioning problem.

Next, we consider the two-dimensional problem. Given a set $S$ of $n$ points in $\mathbb{R}^2$ and an integer $b \leq n$, we again wish to partition $S$ into $b$ *equal-size* buckets so that the maximum number of points in a bucket is minimized. We consider two types of buckets. First, we consider the case in which the buckets are formed by equally spaced $b+1$ parallel lines, $\ell_0, \ldots, \ell_b$, with orientation $\theta$, for some $\theta \in \mathbb{S}^1$. We require $S$ to lie between $\ell_1$ and $\ell_b$ and each of $\ell_1, \ell_b$ contains at least one point of $S$. The *buckets* are $b$ strips defined by consecutive lines $\ell_{i-1}$ and $\ell_i$ $(1 \leq i \leq b)$; see Figure 1 (ii). This bucketing problem is known as the *uniform-projection* problem. We next define buckets to be the regions formed by two families of equally-spaced $\sqrt{b}+1$ lines. The extremal lines in both families are required to contain at least one point of $S$; see Figure 1 (iii). This problem is called the *two-dimensional partition* problem.

Asano and Tokuyama [1] describe $O(n^2)$ and $O(b^2n^2)$-time algorithms for the tight and relaxed cases of the one-dimensional problem. We are able to obtain an $O(b^2\Delta^2 n \log n)$ deterministic algorithm for the tight case and $O(b^3\Delta^2 n \log n)$ algorithm for the relaxed case. The running time of our algorithms is $O(n)$ for

constant values of $b$ and $\Delta$. The algorithm itself does not require the value of $\Delta$; the value is required only for the analysis. This problem has applications to construction of optimal hash functions [1].

Comer and O'Donnell [3] described an algorithm for the uniform-projection problem that runs in $O(bn^2 \log(bn))$ time using $O(n^2 + bn)$ spce. Asano and Tokuyama [1] gave an $O(n^2 \log n)$-time algorithm, which uses $O(n)$ space, by exploiting the dual transformation of the problem. They also give alternative implementations that could be better for smaller $b$, but the worst-case running time is $\Omega(n^2)$ even for constant values of $b$. Bhattacharya [2] also gave an alternate approach for this problem, using the *angle-sweep* method. We first describe a deterministic $O(n^{4/3} \log^{3/2} n)$-time algorithm for $b = 2$ for the uniform-projection problem, thus improving the quadratic upper-bound. For larger values of $b$, we describe a Monte Carlo algorithm that computes an optimal solution in time $O(\min\{n^{5/3}b^2 \log^{3/2} n + b^2\Delta n \log n, n^2\})$, with probability at least $1 - 1/n$. The running time increases to $O(\min\{bn^{7/4} \log^{3/2} n + b^2\Delta n \log n, n^2\}))$ if we restrict space to be linear. The dependence of running time on $\Delta$ is borne out by the fact that the number of possible optimal configurations (having the same value) increases.

Our overall approach for both the problems is similar. Namely, we use a sample to "localize" the search for the global optimum. Although intuitively, this is a good heuristic, analyzing the bound on the number of "potential" candidates for the global optimum, from the optima of the sample, is rather technical. In the one-dimensional problem, we can simply choose a a "deterministic" sample because the elements are linearly ordered, but the two-dimensional algorithms rely on random sampling.

## 2 Optimal One-Dimensional Cuts

For a set $S$ of real numbers $x_i$, $1 \leq i \leq n$ and an integer $b$, a pair $c = (L, w)$ is called a *cut* if the set of $b + 1$ real numbers $\beta_j$, $0 \leq j \leq b$ defined as $\beta_0 = L$ and $\beta_j = L + j \cdot w$ for $j > 0$ are such that $\beta_0 \leq x_1 \leq x_n \leq \beta_b$. The interval $[\beta_{j-1}, \beta_j)$ is called the $j$-th bucket and the set of $x_i$'s lying (strictly) in this interval is the *contents* of the $j$-th bucket. We will denote the $j$-th bucket by $B_j$ and the size of its contents, $|B_j \cap S|$, by $|B_j|^c$ for a cut $c$.

Let $\mathcal{C}$ be the set of all cuts. The optimal *cut-value*, $\Phi(S)$, is defined as $\min_{c \in \mathcal{C}}\{\max_{1 \leq j \leq b}\{|B_j|^c\}\}$ and any cut that achieves this is an *optimal cut*. If we restrict the cuts to satisfy the condition that $|B_1|, |B_b| \geq 1$, then it is called a *tight* cut. That is, the first and the last buckets cannot be empty. An *optimal tight cut* is defined analogously as above (restricted to the set of tight cuts). We will first describe an algorithm for finding an optimal tight cut.

**Definition 1** Two cuts $c_1$ and $c_2$ are *combinatorially distinct* iff there are buckets $B_i$ and $B_j$ such that $|B_i|^{c_1} \neq |B_i|^{c_2}$ and $|B_j|^{c_1} \neq |B_j|^{c_2}$.

We parameterize the problem as follows. We represent each cut $c = (w, L)$ as a point in the plane. Let $\mathcal{L} = \{x_i = L + jw \mid 1 \leq i \leq n,\ 0 \leq j \leq b\}$ be the set of $(b+1)n$ lines in the $(L, w)$-plane, which we refer to as the *event* lines. $\mathcal{L}$ consists of $b + 1$ families of parallel lines (one for each fixed $j$), each family containing $n$ lines; see Figure 2 (i). Hence, every face in $\mathcal{A}(\mathcal{L})$ contains at most $2(b + 1)$ edges. For all cuts $c = (w, L)$ lying in the same face $f$ of $\mathcal{A}(\mathcal{L})$ the cut-value remains the same; we will denote this value by $\Phi(f, S)$. Let $\Phi_j(f, S) = |B_j(S)|^c$ for any $c \in f$. The non-empty condition of extreme buckets implies that we have to consider only those cuts $(w, L)$ that lie in the quadrilateral $Q$ defined by the intersection of the following four constraints.

$$Q: \quad x_1 > L > x_1 - w \quad \text{and} \quad \frac{x_n - L}{b} \leq w \leq \frac{x_n - L}{b-1}. \tag{1}$$

The above constraint also leads to the following lemma whose proof is omitted from this version.

**Lemma 1.** *For every point $x_i \in S$ there exists an integer $1 \leq j \leq b - 1$, such that $x_i$ lies in one of the two buckets $B_j$ or $B_{j+1}$ for any tight cut.*

This lemma immediately implies that at most $n$ lines of $\mathcal{L}$ intersect $Q$ and thus $Q$ intersects $O(n^2)$ faces of $\mathcal{A}(\mathcal{L})$. The lines of $\mathcal{L}$ that intersect $Q$ can be determined in $O(bn)$ time. We can therefore search over $Q \cap \mathcal{A}(\mathcal{L})$ in $O(n^2)$ time to find all combinatorially distinct optimal cuts.

**Lemma 2.** *For a set of $m$ points, all the combinatorially distinct optimal cuts can be computed in $O(m^2)$ time.*

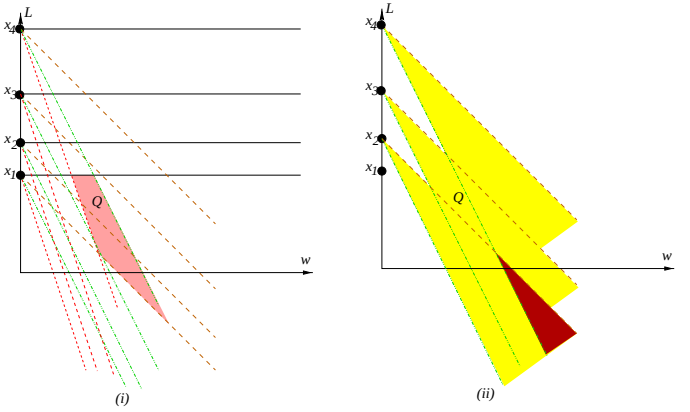



**Fig. 2.** (i) Set $\mathcal{L}$ and the feasible region $Q$; (ii) Shaded regions denote $C_{22}, C_{23}$, and $C_{24}$, and the dark region denotes $\mathcal{C}(2, 4; 2)$, the set of cuts for which $\{x_2, x_3, x_4\}$ lie in the second bucket $B_2$.

For an integer $r \geq 1$, let $R \subseteq S$ be the subset of $r$ points obtained by choosing every $n/r$-th point of $S$. From our previous observation about directly solving the problem, we can compute the optimal solution for $R$ in $O(r^2)$ time.

**Lemma 3.** *Let $n_0, r_0$ be the maximum size of a bucket in an optimal solution for $S$ and $R$, respectively. Then $\left|\frac{n_0}{n} - \frac{r_0}{r}\right| < \frac{1}{r}$.*

*Proof.* Let $c$ be an optimal cut for $R$. Each bucket of $c$ contains at most $r_0$ points. Since $R$ is chosen by selecting every $(n/r)$th point of $S$, each bucket of $c$ contains at most $(r_0+1)n/r - 1$ points of $S$. Therefore $n_0/n < (r_0+1)/r$. A similar argument shows that $n_0/n > (r_0-1)/r$. □

We now describe the algorithm for computing an optimal solution for $S$, assuming that we have already computed the value of $r_0$. Let $C_{ij}$ denote the set of points $c = (L, w)$ in the $(L, w)$-plane so that the point $x_j \in S$ lies in the bucket $B_i$ of the cut $c$. Then $C_{ij} = \{(L, w) \mid L + (i-1)w \le x_j < L + iw\}$ is the cone with apex at $(0, x_j)$. Given three integers $1 \le l \le r \le n$ and $1 \le i \le b$, the set of points in the $(L, w)$-plane for which the subset $\{x_l, x_{l+1}, \ldots, x_r\}$ of $S$ lies in the $i$th bucket $B_i$ is $\mathcal{C}(l, r; i) = \bigcap_{j=l}^{r} C_{ij}$. $\mathcal{C}(l, r; i)$ is a cone formed by the intersection of the halfplanes $L + (i-1)w \le x_l$ and $L + iw > x_r$.

By Lemma 3, $(r_0-1)n/r < n_0 < (r_0+1)n/r$. On the other hand $n_0 \ge n/b$. We perform a binary search in the interval $[\max\{\lceil (r_0-1)n/r \rceil, n/b\}, \lfloor (r_0+1)n/r \rfloor]$ to find an optimal cut for $S$. At each step of the binary search, given an integer $m$, we want to determine whether $n_0 \le m$.

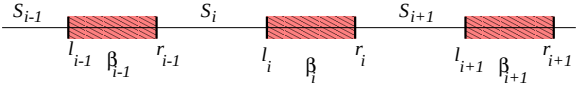



**Fig. 3.** The boundary $\beta_i$ can lie in the shaded interval $[l_i, r_i)$.

Fix an integer $m = (n/b) + \delta$ for $\delta \ge 0$. If $b^2\delta \ge n$, then we use the $O(n^2)$-time algorithm described earlier to compute an optimal cut, so assume that $b^2\delta < n$. If each $B_i$ in a cut $c$ contains at most $m$ points of $S$, then, for any $1 \le i \le b$, the first $i$ buckets in $c$ contain at most $mi$ points and the last $b - i$ buckets in $c$ contain at most $(b-i)m$ points, therefore $\beta_i$ lies in the interval $[x_{l_i}, x_{r_i})$, where $l_i = n - m(b-i)$ and $r_i = mi$. Set $r_0 = 1$; see Figure 3. Note that $r_i - l_i = b\delta$ for $1 \le i \le b$. This implies that the subset $S_i = \{x_j \mid r_{i-1} \le j < l_i\}$ always lies in the $i$th bucket $B_i$ (see Figure 3), for all $1 \le i \le b$. Hence, if there is cut $\xi = (L, w)$ so that all buckets in $\xi$ contain at most $m$ points, then $\xi$ lies in the region $P(m) = \bigcap_{i=1}^{b} \mathcal{C}(r_{i-1}, l_i - 1; i)$, which is the intersection of $b$ cones and is thus a convex polygon with at most $2b$ edges. Next, let $H_i \subseteq \mathcal{L}$ be a set of $l_i - r_i = b\delta$ lines defined as $H_i = \{L + iw = x_j \mid l_i \le j < r_i\}$. Set $H = \bigcup_{i=1}^{b} H_i$; $|H| = b^2\delta$. The same argument as in Lemma 1 shows that no line of $H \setminus \mathcal{L}$ intersects the interior of the polygon $P(m)$.

We construct the arrangement $\mathcal{A}(H)$ within the polygon $P(m)$ in $O(b^4\delta^2)$ time. (Actually, we can clip $\mathcal{A}(H)$ inside $P(m) \cap Q$, where $Q$ is the quadrilateral defined in (1).) Let $\mathcal{A}_P(H)$ denote this clipped arrangement. By the above discussion, $\mathcal{A}_P(H)$ is the same as $\mathcal{A}(L)$ clipped within $P(m)$. Therefore for any two points $(L, w)$ and $(L', w')$ in a face $f \in \mathcal{A}_P(H)$, the contents of all buckets in

the cuts $(L, w)$- and $(L', w')$ are the same. Let $\varphi(f) = \langle \Phi_1(f, S), \ldots, \Phi_b(f, S) \rangle$. If $f$ and $f'$ are two adjacent faces of $\mathcal{A}_P(H)$ separated by a line $L + iw = x_j$, then the only difference in the two cuts is that $x_j$ lies in $B_{i-1}$ in one of them and it lies in $B_i$ in the other. Therefore $\varphi(f')$ can be computed from $\varphi(f)$ in $O(1)$ time. By spending $O(n)$ time at one of the faces of $\mathcal{A}_P(H)$ and then traversing the planar subdivision $\mathcal{A}_P(H)$, we can compute $\Phi(f, S)$ for all of its faces in $O(b^4\delta^2 + n)$ time. If there is any face $f$ for which $\Phi(f, S) \leq m$, we can conclude that $n_0 \leq m$. Otherwise, $n_o > m$. Since $n_0 = (n/b) + \Delta \leq (r_0 - 1)n/r$ and $m = (n/b) + \delta < (r_0 + 1)n/r$, we have $\delta \leq \Delta + 2n/r$. The total time in performing the binary search is $O((b^4(\Delta + (n/r))^2 + n) \log(n/r))$. Hence, the total time spent in computing an optimal cut is

$$O\left(r^2 + b^4\left(\Delta + \frac{n}{r}\right)^2 \log\frac{n}{r} + n\log\frac{n}{r}\right).$$

Choosing $r = \left\lceil b\sqrt{n}\log^{1/4}(n/b^2)\right\rceil$, we obtain the following.

**Lemma 4.** *An optimal tight cut for $n$ points into $b$ buckets can be found in $O(b^4\Delta^2\log(n/b^2) + b^2 n\log^{1/2}(n/b^2))$ time.*

Instead of using the quadratic algorithm for computing $r_0$, we can compute $r_o$ recursively, then we obtain the following recurrence. Let $T(m)$ denote the maximum running time of the algorithm for computing an optimal cut for the subset of $S$ size $m$ chosen by selecting every $(n/m)$th point of $S$, then we have

$$T(n) = \begin{cases} T(r) + O\left(b^4\left(\Delta + \frac{n}{r}\right)^2 + n\right)\log\frac{n}{r} & \text{if } b^2(\Delta + \frac{2n}{r}) \leq n, \\ O(n^2) & \text{otherwise.} \end{cases}$$

Choosing $r = n/2$ and using the fact that $r_o \leq n_o r/n + 1$. we can show that $T(n) = O(b^4\Delta^2\log n + n)$.

**Theorem 1.** *An optimal tight cut for $n$ points into $b$ buckets can be found in $O(b^4\Delta^2\log n + n)$ time.*

**Remark.** By choosing the value of $r$ more carefully, we can improve the running time to $O(b^4\Delta^2\log_\Delta n + n\log\log\Delta)$.

We can use a similar analysis for finding optimal cuts, including relaxed cuts. We simply replace $n$ by $bn$ as there are $bn$ event lines. Another way to view this is that the optimal cut can be determined by trying out all non-redundant cuts for $\eta$ buckets for $2 \leq \eta \leq b$ and selecting the best one.

**Corollary 1.** *An optimal relaxed cut for a set of $n$ points into $b$ buckets can be found in $O(b^5\Delta^2\log n + bn)$ time.*

## 3 The Uniform-Projection Problem

In this section we describe the algorithms for the uniform projection problem. Let $S = \{p_1, \ldots, p_n\}$ be a set of $n$ points in $\mathbb{R}^2$ and $1 \leq b \leq n$ an integer. We

want to find $b+1$ equally spaced parallel lines so that all points of $S$ lie between the extremal lines, the extreme lines contain at least one points of $S$, and the maximum number of points in a bucket is minimized. If the lines have slope $\theta$, we refer to these buckets as a $\theta$-cut of $S$. We first describe a subquadratic algorithm for $b = 2$. Next, we show how the running time of the algorithm by Asano and Tokuyama can be improved, and then we describe a Monte Carlo algorithm that computes $\Phi(S)$, the optimum value, with high probability, in subquadratic time for certain values of $b$ and $\Delta$.

It will be convenient to work in the dual plane. The duality transform maps a point $p = (a, b)$ to the line $p^* : y = -ax + b$ and a line $\ell : y = \alpha x + \beta$ to the point $\ell^* = (\alpha, \beta)$. Let $\ell_i$ denote the line dual to the point $p_i \in S$, and let $\mathcal{L} = \{\ell_i \mid 1 \leq i \leq n\}$. Let $\mathcal{A}(\mathcal{L})$ denote the arrangement of $\mathcal{L}$. We define the *level* of a point $p \in \mathbb{R}^2$ with respect to $\mathcal{L}$, denoted by $\lambda(p, \mathcal{L})$, to be the number of lines in $\mathcal{L}$ that lie below $p$. The level of all points within an edge or a face of $\mathcal{A}(\mathcal{L})$ have the same level. For an integer $0 \leq k < n$, we define the $k$-level of $\mathcal{A}(\mathcal{L})$, denoted by $\mathcal{A}_k(\mathcal{L})$, to be the closure of the set of edges of $\mathcal{A}(\mathcal{L})$ whose levels are $k$. $\mathcal{A}_k(\mathcal{L})$ is an $x$-monotone polygonal chain with at most $O(n(k+1)^{1/3})$ edges [4]. The *lower* and *upper* envelopes of $\mathcal{A}(\mathcal{L})$ are the levels $A_0(\mathcal{L})$ and $\mathcal{A}_{n-1}(\mathcal{L})$.

Since we require the extreme bucket boundaries to contain a point of $S$, they correspond to points on the upper and lower envelopes of $\mathcal{L}$. For a fixed $x$-coordinate $\theta$, let $s(\theta)$ denote the vertical segment connecting the points on the lower and upper envelopes of $\mathcal{L}$ with the $x$-coordinate $\theta$. We can partition $s(\theta)$ into $b$ equal-length subsegments $s_1(\theta), \ldots, s_b(\theta)$. Let $\beta_0(\theta), \ldots, \beta_b(\theta)$ be the endpoints of these segments. These endpoint are dual of the bucket boundaries with slope $\theta$, and $s_i(\theta)$ is the dual of the $j$th bucket in the $\theta$-cut. The line $\ell_j$ intersects $s_i(\theta)$, $i \leq b$, iff the point $p_j$ lies in the bucket $B_i$ corresponding to the $\theta$-cut. Let $\beta_i$ denote the path traced by the endpoint $\beta_i(\theta)$ as we vary $\theta$ from $-\infty$ to $+\infty$. If we vary $\theta$, as long as the endpoints of $s(\theta)$ do not pass through a vertex of upper or lower envelopes, $\beta_i(\theta)$, $0 \leq i \leq b$, trace along line segments. Therefore each $\beta_i$ is an $x$-monotone polygonal chain with at most $2n$ vertices; see Figure 4 for an illustration. Since we will be looking at the problem in the dual plane from now, we will call $\beta_i$'s *bucket lines*. Let $\mathcal{B} = \{\beta_0, \ldots, \beta_b\}$. The intersection of a bucket line with a line (that is dual of point) marks an *event* where the point switches between the corresponding buckets.

For an $x$-coordinate $\theta$ and a subset $A \subseteq \mathcal{L}$, let $\mu_i(A, \theta)$ denote the number of lines of $A$ that intersect the vertical segment $s_i(\theta)$; $\mu_i(A, \theta)$ denotes the set of points dual to $A$ that lie in the $i$th bucket in the $\theta$-cut. Let $\Phi(A, \theta) = \max_i\{\mu_i(A, \theta)\}$. Set $n_o = \Phi(S) = \min_x \Phi(S, x)$.

### 3.1 Partitioning into Two Buckets

We will first describe a deterministic scheme that takes subquadratic time to find an optimal solution for partitioning $S$ into two buckets. By our convention, $\beta_0, \beta_2$ denote the upper and lower envelopes of $\mathcal{L}$, respectively. To determine $n_o$, we will search for an $x$ coordinate $x_o$, where $\beta_1(x_0)$ is closest to the middle level $\lambda_{n/2}$ of the arrangement $\mathcal{A}(\mathcal{L})$. Let $\lambda_o$ be the level of $\beta_1(x_o)$. We do a

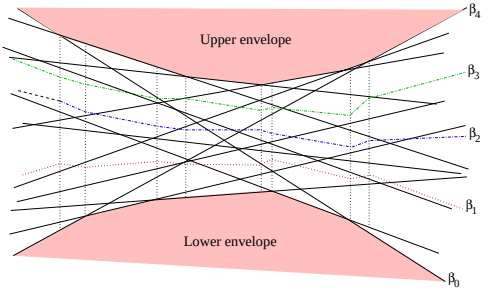



**Fig. 4.** The uniform-projection problem and the bucket lines in the dual setting.

binary-search to determine $\lambda_o$ in the following manner. We start by looking for an intersection between $\lambda_{n/2}$ and $\beta_1$. If there is one, then we report the optimum $n/2$ split. Otherwise we repeat the procedure for the levels between 0 and $n/2$ if $\beta_1$ lies below $\lambda_{n/2}$, and for the levels between $n/2$ and $n-1$ if $\beta_1$ lies above $\lambda_{n/2}$. Since the maximum size of a level of $n$ lines is $O(n^{4/3})$ [4] and it can be constructed in an output-sensitive manner, each phase of the search takes at most $O(n^{4/3}\log^{1+\varepsilon} n)$ time for any $\varepsilon > 0$. Note that the total number of intersections between $\beta_1$ which has size $n$ and any level of size $s$ is $O(n+s)$.

Hence for the case $b = 2$, we have a sub-quadratic algorithm.

**Lemma 5.** *The optimal uniform projection of $n$ points in $\mathbb{R}^2$ into two buckets can be computed in $O(n^{4/3}\log^{2+\varepsilon} n)$ steps, for any $\varepsilon > 0$, using $O(n)$ space.*

### 3.2 A Deterministic Algorithm

In this section we present a deterministic algorithm for the uniform-projection problem that has $O(bn\log n + K\log n)$ running time and uses $O(n+b)$ storage, where $K$ denotes the number of event points, i.e., the number of intersection points between $\mathcal{L}$ and $\mathcal{B}$. This improves the running times of $O(n^2+bn+K\log n)$ for general $b$ and $O(b^{0.610}n^{1.695} + K\log n)$ for $b < \sqrt{n}$ in [1].

As in Asano-Tokuyama's algorithm, we will sweep a vertical line through $\mathcal{A}(\mathcal{L})$, but unlike their approach we will not stop at every intersection point of $\mathcal{L}$ and $\mathcal{B}$. We first compute the lower and upper envelopes of $\mathcal{L}$, which are the bucket lines $\beta_0$ and $\beta_b$, respectively. We can then compute rest of the bucket lines $\beta_1, \ldots, \beta_{b-1}$ in another $O(bn)$ time. We preprocess each $\beta_i$ for answering ray-shooting queries in $O(n\log n)$ time so that a query can be answered in $O(\log n)$ time [5].

For any line $\ell \in \mathcal{L}$, we can compute all $K_\ell$ intersection points of $\ell$ with $\mathcal{B}$ in $O((K_\ell+1)\log n)$ time, sorted by their $x$-coordinates, using the ray-shooting data structure as follows. We traverse $\ell$ from left to right, stopping at each intersection point of $\ell$ with $\mathcal{B}$. Suppose we are at an intersection point $v$ of $\ell$ and $\beta_i$ and $\ell$ lies above $\beta_i$ immediately after $v$. Then the next intersection point $w$ of $\ell$ and $\mathcal{B}$, if it exists, lies on either $\beta_i$ or $\beta_{i+1}$. By querying the ray-shooting

data structures for $\beta_i$ and $\beta_{i+1}$ with the ray emanating from $v$ along $\ell$, we can compute $w$ in $O(\log n)$ time, if it exists. We can thus compute all intersection points of $\mathcal{L}$ and $\mathcal{B}$ and sort them by their $x$-coordinates in $O((n+K)\log n)$ time. After having computed all the event points, we can compute an optimal cut in another $O((n+K))$ time by sweeping from left to right, as in [1]. The space required by the algorithm is $O(bn+K)$. The space can be reduced to $O(n+b)$, without affecting the asymptotic running time, by not computing all the event points in advance; see [1] and the full version for details.

**Theorem 2.** *An optimum partitioning in the tight case can be determined in* $O((bn+K)\log n)$ *time using* $O(n)$ *storage, where* $K$ *is the number of event points.*

### 3.3 A Monte-Carlo Algorithm

We now present an algorithm that attains sub-quadratic running time for small values of $b$ and $\Delta$, where $n_o = (n/b) + \Delta$. The overall idea is quite straightforward. From the given set $\mathcal{L}$ of $n$ lines, we choose a random subset $R$ of size $r \geq \log n$ (a value that we will determine during the analysis). We compute $r_o = \min_\theta \Phi(R, \theta)$, where the minimum is taken over the $x$-coordinates of all the intersection points of $R$ and $\mathcal{B}$, the set of bucket lines with respect to $\mathcal{L}$. Note that we are not computing $\Phi(R)$ since we are considering buckets lines with respect to $\mathcal{L}$. $\mathcal{B}$ can be computed in $O(n\log n + bn)$ time and $r_o$ can be computed in additional $O(r(b+n)) = O(rn)$ time. We use $r_o$ to estimate the overall optimum $n_o$ with high likelihood. In the next phase, we will use this estimate and the ideas used in the one-dimensional algorithm to sweep only those regions of $\mathcal{B}$ that "potentially" contain the optimal. In our analysis, we will show that the number of such event points will be $o(n^2)$ if $b$ and $\Delta$ are small. The reader can also view this approach as being similar to the randomized selection algorithm of Floyd and Rivest.

We choose two parameter $r$ and $\mathrm{Var} = \mathrm{Var}(r)$ whose values will be specified in the analysis below. An *event* point with respect to $\mathcal{L}$ (resp. $R$) is a vertex of $\mathcal{B}$ or an intersection point of a line of $\mathcal{L}$ (resp. $R$) with a chain in $\mathcal{B}$. The event points with respect to $R$ partition the chains of $\mathcal{B}$ into disjoint segments, which we refer to as *canonical intervals*. Before describing the algorithm we state a few lemmas, which will be crucial for our algorithm.

Our first lemma establishes a relation between the *event* points of $\mathcal{A}(\mathcal{L})$ and those of $\mathcal{A}(R)$.

**Lemma 6.** *Let* $\alpha > 0$ *be a constant and let* $1 \leq i \leq b$ *be an integer. With probability at least* $1 - 1/n^\alpha$*, at most* $O(n\log n/r)$ *event points of* $\mathcal{A}(\mathcal{L})$ *lie on any canonical interval of* $\beta_i$*.*

In the following, we will assume that $R$ is a random subset of $\mathcal{L}$ of size $r \geq \log n$. Using Chernoff's bound, we establish a connection between the lines of $\mathcal{L}$ and of $R$ intersecting a vertical segment.

**Lemma 7.** *Let $e$ be a vertical segment and let $\mathcal{L}_e \subseteq \mathcal{L}$ be the subset of lines that intersect $e$. There is a constant $c$ such that with probability exceeding $1 - 1/n^2$,*

$$\left| \frac{|\mathcal{L}_e|}{n} - \frac{|\mathcal{L}_e \cap R|}{r} \right| \leq c\sqrt{\frac{|\mathcal{L}_e|}{n} \cdot \frac{\log n}{r}}.$$

An immediate corollary of the above lemma is the following.

**Corollary 2.** *There is a constant $c$ so that, with probability exceeding $1 - 1/n$,*

$$\left| \frac{n_o}{n} - \frac{r_o}{r} \right| \leq c\sqrt{\frac{\log n}{r}}.$$

**Corollary 3.** *Let $\xi$ be a $\theta$-cut so that every bucket of $\xi$ contains at most $m$ points of $S$. For $1 \leq i \leq b-1$, let*

$$l_i = r - (b-i)m\frac{r}{n} - c\sqrt{r \log n} \qquad \text{and} \qquad r_i = im\frac{r}{n} + c\sqrt{r \log n},$$

*where $c$ is a constant. Then with probability $> 1 - 1/n$, $l_i \leq \lambda(\beta_i(\theta), R) \leq r_i$.*

*Proof.* If each bucket of $\xi$ at most $m$ points, then the first $i$-buckets of $\xi$ contain at most $mi$ points of $S$ and the last $(b-i)$ buckets of $\xi$ contain at most $(b-i)m$ points of $S$. The lemma now follows from Lemma 7. □

We also need the following result by Matoušek on simplex range searching.

**Lemma 8 (Matoušek [6]).** *Given a set $P$ of points and a parameter $m$, $n \leq m \leq n^2$, one can preprocess $P$ for triangle range searching in time $O(m \log n)$, to build a data-structure of $O(m)$ space and then report queries in $O((n \log^2 n)/\sqrt{m} + K)$ time, for output size $K$, where $K$ is number of points in the query triangle.*

**Remark.** If $m = O(r^2 \log n)$ and $K \geq (n/r) \log n$, then the output size dominates the query time, so the query time becomes $O(K)$ in this case.

We now describe the algorithm in detail. Choose a random sample $R$ of size $r$, where $r \geq \log n$ is a parameter to be fixed later, and compute $r_o = \min_\theta \Phi(R, \theta)$, where $\theta$ varies over the $x$-coordinates of all the event points of $\mathcal{B}$ with respect to $R$. We can use a quadratic algorithm to compute $r_o$. By Corollary 2, $n_0 \leq m = nr_0/r + cn\sqrt{(\log n)/r}$. Suppose $n_0 = (n/b) + \Delta$ and $m = (n/b) + \delta$. Since $n_0 \geq nr_0/r - cn\sqrt{(\log n)/r}$, we have $\delta \leq \Delta + 2n\sqrt{(\log n)/r}$ and $m \leq (n/b)\Delta + 2n\sqrt{(\log n)/r}$.

We will describe an algorithm that searches over all $\theta$-cuts for which $\Phi(S, \theta) \leq m$ and computes the value of $n_o$. For each $1 \leq i < b$, let $X_i = \{\theta \mid l_i \leq \lambda(\beta_i(\theta), R) \leq r_i\}$. We will describe below how to compute $X_i$. Let $X = \cap_{i=1}^{b-1} X_i$. Next, for each $0 \leq i \leq b$, we compute the set $\mathcal{I}_i$ of canonical intervals of $\beta_i$ whose $x$-projections intersect $X$. Let $\mathcal{I} = \bigcup_{i=0}^{b} \mathcal{I}_i$, and set $\nu = |\mathcal{I}|$. We will describe below how to compute $\mathcal{I}$. Let $E_i$ denote the set of $x$-coordinates of

event points (with respect to $\mathcal{L}$) lying on $\mathcal{I}_i$. Set $E = \bigcup_{i=0}^{b} E_i$. By Lemma 6, $|E| = O((\nu n/r)\log r)$.

By Corollary 3, if $\xi$ is a $\theta$-cut so that all buckets in $\xi$ have at most $m$ points of $S$, then $\theta \in X$. Since the contents of buckets change only at event points, $\Phi(\mathcal{L}) = \min_{\theta \in E} \Phi(\mathcal{L}, \theta)$. We thus have to compute $\Phi(\mathcal{L}, \theta)$ for all $\theta \in E$.

We preprocess $S$ in $O(r^2 \log^2 n)$ time into a data structure of size $O(r^2 \log n)$ for answering triangle range queries using Lemma 8. For each canonical interval $I \in \mathcal{I}_i$, we compute the subset $\mathcal{L}_I \subseteq \mathcal{L}$ of lines that intersect $I$ in $O((n/r)\log r)$ time. We then compute the intersection points of $I$ and $\mathcal{L}_I$ — these are the event points with respect to $\mathcal{L}$ that lie on $I$. We repeat this step for all intervals in $\mathcal{I}$. The total time spent in computing these intersection points is $O(r^2 \log^2 n + \nu(n/r)\log r)$. We now have all points in $E$ at our disposal. For $1 \le i \le b$, $\mu_i(\mathcal{L}, \theta)$, for $\theta \in E$, changes only if $\theta \in E_{i-1} \cup E_i$, i.e., when a line of $\mathcal{L}$ intersects the bucket line $\beta_{i-1}$ or $\beta_i$. We compute $\mu_i(\mathcal{L}, \theta)$, for $\theta \in E_{i-1} \cup E_i$, as follows. If $\theta$ is the $x$-coordinate of the left endpoint of a canonical interval in $\mathcal{I}_{i-1} \cup \mathcal{I}_i$, we use the range-searching data structure to compute in $O((n/r)\log r)$ time the number $k_\theta$ of lines in $\mathcal{L}$ that intersect the segment $s_i(\theta)$; $\mu_i(\mathcal{L}, \theta) = k_\theta$. Otherwise, if $\theta$ lies on a canonical interval $I$ and if $\theta'$ is the $x$-coordinate of the event point lying before $\theta$ on $I$, then we compute $\mu_i(\mathcal{L}, \theta)$ from $\mu_i(\mathcal{L}, \theta')$ in $O(1)$ time. The total time spent in computing $\mu_i$'s is $O((|\mathcal{I}_{i-1}| + |\mathcal{I}_i|)(n/r)\log r)$. Summing over $1 \le i \le b$, we spent a total of $O((\nu n/r)\log r)$ time. By our construction, we now have $\mu_i(\mathcal{L}, \theta)$ all $\theta \in E$. By scanning all these $\mu_i$'s once more, we can compute $\min_{\theta \in E} \Phi(\mathcal{L}, \theta)$ in another $O(\nu(n/r)\log r)$ time.

It thus suffices to describe how to compute $\mathcal{I}_i$. Set $l_i = r - (b-i)mr/n - c\sqrt{r \log n}$ and $r_i = imr/n + c\sqrt{r \log n}$. Define

$$\sigma = r_i - l_i = bm\frac{r}{n} - r + 2c\sqrt{r \log n} \le b\Delta\frac{r}{n} + 4c\sqrt{r \log n}.$$

Since $l_i \le \lambda(\beta_i(\theta), R) \le r_i$ for all $\theta \in X_i$, we compute the levels $A_j(R)$, $l_i \le j \le r_i$. Let $M_i$ be the resulting planar subdivision induced by these levels. By a result of Dey [4], $|M_i| = O(r^{4/3}(r_i - l_i)^{2/3}) = O(r^{4/3}\sigma^{2/3})$. $M$ can be computed in time $O(|M| \log^{1+\varepsilon} n)$. Since $\beta_i$ is an $x$-monotone polygonal chain and $M$ consists of $\sigma$ edge-disjoint $x$-monotone polygonal chains, the number of intersection points between $\beta_i$ and $M_i$ is $O(n\sigma + |M|)$, and they can be computed within that time bound. We can thus compute the set $\mathcal{I}'_i$ of all canonical intervals of $\beta_i$ whose $x$-projections intersect $X_i$ in time $O(n\sigma + r^{4/3}\sigma^{2/3}\log^{1+\varepsilon} r)$. Let $X_i$ be the $x$-projection of the portion of $\beta_i$ that lies between $A_{l_i}(R)$ and $A_{r_i}(R)$. $X_i$ consists of $O(n + r^{4/3})$ intervals. We set $X = \bigcup_{i=1}^{b-1} X_i$; $|X| = O(b(n + r^{4/3}))$. Next we discard those canonical intervals of $\mathcal{I}'_i$ whose $x$-projections do not intersect $X$. The remaining intervals of $\mathcal{I}'_i$ gives the set $\mathcal{I}_i$. Therefore $\nu = \sum_i |\mathcal{I}'_i| = O(b(n\sigma + r^{4/3}\sigma^{2/3}))$. Repeating this procedure for $0 \le i \le b$, the total time in computing $\mathcal{I}$ is $O(b(n\sigma + r^{4/3}\sigma^{2/3}\log^{1+\varepsilon} r))$. The total time in computing $n_o$ is thus

$$O(n \log n + bn) + O(rn) + O(r^2 \log r) + O(b(n\sigma + r^{4/3}\sigma^{2/3})) \cdot (n/r)\log r.$$

Setting $r = \lceil (bn)^{2/3} \log n \rceil$ and using the fact that $b \le n$, we obtain the following.

**Theorem 3.** *There is a Monte Carlo algorithm to compute the optimal uniform projection of a set of $n$ points in $\mathbb{R}^2$ onto $b$ equal-sized buckets in time $O(\min\{b^{2/3}n^{5/3}\log n + (b^2\Delta)n\log n,\, n^2\})$, using $O((bn)n^{4/3}\log^3 n)$ space, with probability at least $1-1/n$, where the optimal value is $(n/b)+\Delta$. In particular, our algorithm can detect and report if there is a uniform projection (i.e., with $\Delta = 0$) in $O(b^{2/3}n^{5/3}\log n)$ time.*

## 4 Two-Dimensional Partitioning

Using the algorithm developed for the uniform projection, we can obtain a two-dimensional partitioning of a point set in the plane similar to the one-dimensional partitioning. More specifically, we overlay the plane with two orthogonal families of $\sqrt{b}+1$ equally spaced parallel lines. These lines generate $b$ rectangles. Our previous strategy can be used to obtain an efficient algorithm for the problem of obtaining a partitioning that minimizes the maximum bucket content with the restriction that the buckets lie between the extreme points.

As noted by Asano and Tokuyama, the dual plane parameterization gives a direct solution to the problem by sweeping the arrangement by two vertical lines that are a fixed distance apart (this corresponds to the fixed angle in the primal plane). Using our approach, we obtain a result analogous to Theorem 3, but has a time-bound that is a multiplicative $b$ factor more. This is because for a candidate-interval corresponding to one sweep line, there could be $b$ rectangles. For optima bounded by $n/b^2 + \Delta$, the candidate intervals correspond to the bucket lines lying within $b(\Delta r/n + O(\sqrt{r\log n}))$ levels of $\mathcal{A}(R)$. Omitting all the details, we conclude the following.

**Theorem 4.** *Given a set of $n$ points in the plane and an integer $b$, there exists a Monte-Carlo algorithm to find an optimal two-dimensional partition in time $O(\min\{b^{5/3}n^{5/3}\log n + (b^3\Delta)n\log n,\, n^2\})$, with probability at least $1-1/n$, where the optimal value is $n/b^2+\Delta$.*

## References

1. T. Asano and T. Tokuyama. Algorithms for projecting points to give the most uniform distribution and applications to hashing. *Algorithmica*, 9, pp. 572–590, 1993.
2. B. Bhattacharya. Usefulness of angle sweep. *Proc. of FST&TCS*, New Delhi, India, 1991.
3. D. Comer and M.J. O'Donnell Geometric problems with applications to hashing. *SIAM Journal on Computing*, 11, pp. 217–226, 1982.
4. T. K. Dey, Improved bounds on planar -sets and related problems, *Discrete Comput. Geom.*, 19 (1998), 373–382.
5. J. Hershberger and S. Suri, A pedestrian approach to ray shooting: Shoot a ray, take a walk, *J. Algorithms*, 18 (1995), 403–431.
6. J. Matoušek. Efficient Partition trees. *Discrete and Computational Geometry*, 8, pp. 315–334, 1992.

# Convexifying Monotone Polygons

Therese C. Biedl[1,⋆], Erik D. Demaine[1], Sylvain Lazard[2,⋆],
Steven M. Robbins[3], and Michael A. Soss[3]

[1] Department of Computer Science, University of Waterloo,
Waterloo, Ontario N2L 3G1, Canada,
{biedl, eddemaine}@uwaterloo.ca
[2] INRIA Lorraine – LORIA, Projet ISA, 615 rue du jardin botanique B.P. 101,
54602 Villers les Nancy Cedex, France,
lazard@loria.fr
[3] School of Computer Science, McGill University, 3480 University Street,
Montréal, Québec H3A 2A7, Canada,
{stever, soss}@cgm.cs.mcgill.ca

**Abstract.** This paper considers reconfigurations of polygons, where each polygon edge is a rigid link, no two of which can cross during the motion. We prove that one can reconfigure any monotone polygon into a convex polygon; a polygon is *monotone* if any vertical line intersects the interior at a (possibly empty) interval. Our algorithm computes in $O(n^2)$ time a sequence of $O(n^2)$ moves, each of which rotates just four joints at once.

## 1 Introduction

An interesting area in computational geometry is the reconfiguration of (planar) *linkages*: collections of line segments in the plane (called *links*) joined at their ends to form a particular graph. A *reconfiguration* is a continuous motion of the linkage, or equivalently a continuous motion of the joints, that preserves the length of each link. We further enforce that links do not cross, that is, do not intersect during the motion. For a survey of work on linkages where crossing is allowed, see the paper by Whitesides [9].

The case of noncrossing links has had a recent surge of interest. The most fundamental question [7] is still open: Can every chain be reconfigured into any other chain with the same sequence of link lengths? Here a *chain* is a linkage whose underlying graph is a path. Because reconfigurations are reversible, an equivalent formulation of the question is this: Can every chain be *straightened*, that is, reconfigured so that the angle between any two successive links is $\pi$? This question has been posed independently by several researchers, including Joseph Mitchell, and William Lenhart and Sue Whitesides [6]. It has several applications, including hydraulic tube and wire bending, and sheet metal folding [7].

At first glance, it seems intuitive that any chain can be "unraveled" into a straight line, but experimentation reveals that this is a nontrivial problem.

⋆ Research performed during a post-doctoral position at McGill University.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 415–424, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Indeed, no such "general unraveling" motions have been formally specified. Because the problem is so elusive, it is natural to look at special classes of linkages and prove that at least they can be straightened. For example, consider the class of *monotone* chains, where every vertical line intersects the chain at a point or not at all. Such a chain can easily be straightened by repeatedly (1) rotating the first link until it lines up with the second link, and (2) fusing these links together into a single "first link." This motion induces no crossings because it preserves monotonicity throughout.

This paper addresses the analogous question of straightenability for *polygons* (a linkage whose underlying graph is a cycle): Can every polygon be reconfigured into a convex polygon? In other words, can every polygon be *convexified*? This question was also raised by the researchers mentioned above. Note that a convex polygon can be reconfigured into any other convex polygon with the same clockwise sequence of link lengths, and hence the question is equivalent to the fundamental question for polygons [7]: Can every polygon be reconfigured into any other polygon with the same clockwise sequence of link lengths?

In this paper we focus on the case of monotone polygons. Similar to the case of chains, a polygon is *monotone* if the intersection of every vertical line with the interior of the polygon is an interval, that is, either a single vertical line segment, a point, or the empty set. See Fig. 1 for an example. A monotone polygon consists of two chains, the upper chain and the lower chain. Each chain is *weakly monotone* in the sense that the intersection with a vertical line is either empty, a single point, or a vertical edge.[1] The left [right] ends of the upper and lower chains may be identical (like point $A$ in Fig. 1), or they may be connected by a vertical edge (like edge $(B, C)$ in Fig. 1). The vertical edge $(B, C)$ belongs to neither chain.

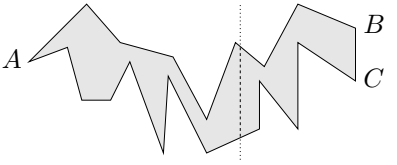



**Fig. 1.** A monotone polygon.

In contrast to monotone chains, it is nontrivial to convexify monotone polygons. In this paper, we show that this is possible by a fairly simple motion consisting of a sequence of $O(n^2)$ moves. We use just a single type of move, changing the angles of only four joints, which we show is the fewest possible. While the proof of correctness is nontrivial, our algorithm for computing the motion is simple and efficient, taking $O(n^2)$ time.

### 1.1 Related Work

Let us briefly survey the work on reconfiguring linkages whose links are not allowed to cross.

The most related result, by Bose, Lenhart, and Liotta [3], is that all monotone-separable polygons can be convexified. A *monotone-separable* polygon is a monotone polygon whose upper and lower chains are separated by a line segment (connecting the common ends of the chains). Their motion involves translating

[1] All straight (angle-$\pi$) vertices are removed, so there is no possibility of two adjacent vertical edges.

almost all joints in the upper chain at once, and appears not to extend to general monotone polygons.

The only other result about convexifying classes of polygons is that every star-shaped polygon can be convexified [5]. A polygon is *star-shaped* if its boundary is entirely visible from a single point. This motion rotates all joints simultaneously, and it seems difficult to find a motion involving few joints [10].

The remaining related results are for types of linkages other than polygons. For tree linkages, it is known that the answer to the fundamental problem is "no" [2]: There are some trees which cannot be reconfigured into other trees with the same link lengths and planar embedding. Indeed, there can be an exponential number of trees with the same link lengths and planar embedding that are pairwise unreachable. The complexity of determining whether a tree can be reconfigured into another remains open.

Another way to change the problem is to allow linkages in higher dimensions. If we start with a polygon in the plane, and allow motions in three dimensions, then every polygon can be convexified [8]. Indeed, a 1935 problem by Erdős asks whether a particular sequence of moves through 3D, each rotating only two joints (called a "flip"), converges in finite time. While the answer is positive, the number of moves is unfortunately unbounded in $n$. Recently, it was shown that $O(n)$ moves of a different kind suffice [1]; each rotates at most four joints.

If the polygon lies in three dimensions and we want to convexify it by motion through three dimensions, then it is surely not convexifiable if it is knotted. But there are unknotted polygons that cannot be convexified [1]. The complexity of determining whether a polygon in 3D can be convexified also remains open. Amazingly, Cocan and O'Rourke [4] have shown that every polygon in $d$ dimensions can be convexified through $d$ dimensions for any $d \geq 4$.

### 1.2 Outline

The rest of this paper is organized as follows. Section 2 begins with a more formal description of the problem. Section 3 describes our algorithm for computing the motion. Sections 4 and 5 prove its correctness and bound its performance, respectively. We conclude in Section 6.

## 2 Definitions

This section gives more formal definitions of the concepts considered in this paper: linkages, configurations, and motions.

Consider a graph, each edge labeled with a positive number. Such a graph may be thought of as a collection of distance constraints between pairs of points in a Euclidean space. A *realization* of such a graph maps each vertex to a point, also called a *joint*, and maps each edge to the closed line segment, called a *link*, connecting its incident joints. The link length must equal the label of the underlying graph edge. If a graph has one or more such realizations, we call it a *linkage*.

An embedding of a linkage in space is called a *configuration* of the linkage. In a *simple configuration*, any pair of links intersect only at a common endpoint, and in this case the links must be incident at this joint in the linkage. We consider only simple configurations in this paper. A *motion* of a linkage is a continuous movement of its joints respecting the link lengths such that the configuration of the linkage remains simple at all times.

In this paper, we consider linkages embedded in the plane whose graph is a single cycle. The configurations are simple polygons, and so divide the plane into the *exterior* and *interior* regions (distinguished by the fact that the interior region is bounded). Joints of an $n$-link linkage are labeled $j_0, j_1, \ldots, j_{n-1}$ in a counterclockwise manner: traversing the boundary in sequence $j_0, j_1, \ldots, j_{n-1}$ keeps the interior region on the left. The *joint angle* $\theta_i$ is the interior angle at joint $j_i$: $\theta_i = \angle j_{i+1} j_i j_{i-1} \in (0, 2\pi)$. We call a joint *straight* if the joint angle is $\pi$, *convex* if the joint angle is strictly less than $\pi$, and *reflex* otherwise. A configuration is *convex* if none of its joints are reflex.

The question considered is whether every monotone configuration of a cyclic linkage (or *polygon*) can be *convexified*, that is, moved to a convex configuration. We show this is true by giving an algorithm to compute such a motion.

## 3 Algorithm

As input, the algorithm requires a description (that is, the joint coordinates) of a polygon with $n$ links. In each main step, the algorithm computes a sequence of $O(n)$ moves that ultimately straighten a joint. This joint angle is then held fixed so that it remains straight forever after, effectively reducing the number of joints. As the algorithm continues, the configuration has fewer and fewer nonstraight joints. We stop when no reflex joint is left, and so the polygon is convex as desired.

First we need some notation for basic geometric concepts. Let $p$ and $q$ be two points in the plane. Define $\|pq\|$ to be the Euclidean distance between $p$ and $q$. If $p$ and $q$ are distinct, define $\mathrm{ray}(p, q)$ to be the ray originating at $p$ and passing through point $q$. We use "left" and "right" in two different senses, one for points and one for rays. A point $r$ is *left* or *right* of $\mathrm{ray}(p, q)$ if it is strictly left or right (respectively) of the oriented line supporting $\mathrm{ray}(p, q)$. A point $p$ is *left* or *right* of point $q$ if $p$ has a strictly smaller or larger $x$ coordinate than $q$, respectively. In both cases, we use *nonstrictly* left/right to denote left/right or equality, i.e., neither left nor right.

The algorithm works as follows:

**Algorithm Convexify**

- Until the polygon is convex:
  - Find a rightmost reflex joint (a joint with maximum $x$ coordinate, breaking ties arbitrarily).
  - Relabel joints counterclockwise along the polygon so that this rightmost reflex joint is $j_1$, and all straight joints are ignored.
  - If $j_1$ belongs to the lower chain:

1. Compute the largest index $p$ such that $j_2, \ldots, j_{p-1}$ are right of ray $(j_0, j_1)$.
2. Until a joint has straightened:
   a) Perform the following move (see Fig. 2) until either the joints $j_0$, $j_1$, and $j_{p-1}$ become collinear; or one of the joints $\{j_0, j_1, j_{p-1}, j_p\}$ straightens:
      i. Fix the positions of joints $j_p$, $j_{p+1}$, $\ldots$, $j_{n-1}$, and $j_0$.
      ii. Fix all joint angles except those at $j_0$, $j_1$, $j_{p-1}$, and $j_p$.
      iii. Rotate $j_1$ clockwise about $j_0$.
      iv. Move joint $j_{p-1}$ as uniquely defined by maintaining the distances $\|j_1 j_{p-1}\|$ and $\|j_{p-1} j_p\|$.
      v. Move joint $j_i$, $2 \leq i \leq p-2$, as uniquely defined by maintaining the distances $\|j_1 j_i\|$ and $\|j_i j_{p-1}\|$.

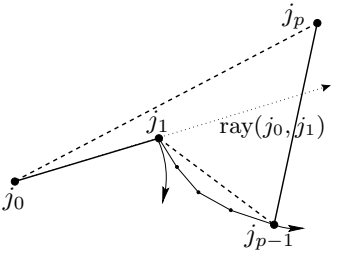



**Fig. 2.** The movement of joints $j_1$ and $j_{p-1}$. The thick dashed lines represent "virtual links" whose lengths are preserved.

   b) Update the coordinates of $j_1$ and $j_{p-1}$.
   c) If $j_0$, $j_1$, and $j_{p-1}$ are collinear, then decrement $p$, because with the new positions, $j_{p-1}$ is on ray$(j_0, j_1)$. Also update the coordinates of the new $j_{p-1}$.
3. Update the coordinates of any remaining joints that have moved.

– If $j_1$ belongs to the upper chain, the algorithm is similar.

First let us justify that the algorithm is well-defined.

**Lemma 1.** *The definition of p in Step 1 is well-defined and at least* 3.

*Proof.* Because $j_1$ is reflex, ray$(j_0, j_1)$ must intersect the polygon elsewhere than the segment $(j_0, j_1)$. This implies that there are joints on both sides of the ray, and hence $p$ is well-defined. Furthermore, because $j_1$ is reflex and the joints are oriented counterclockwise on the polygon, $j_2$ is right of ray$(j_0, j_1)$. Hence, 2 is a valid value for $p-1$, so $p \geq 3$. □

Note further that the number of simultaneously rotating joints (four) is the best possible, because any motion of a polygon that rotates just three joints reconfigures a virtual triangle, which is rigid.

## 4 Proof of Correctness

In this section, we prove the following theorem.

**Theorem 1.** *Given any monotone polygon, Algorithm Convexify computes a convexifying motion, during which the polygon remains simple and monotone.*

The difficulty is in showing that the polygon remains monotone and simple during the motion. For the remainder of this section, assume without loss of generality that the link $(j_0, j_1)$ is on the lower chain. First we need some trivial but important observations.

**Lemma 2.** *During each move, no reflex angle becomes convex and no convex angle becomes reflex.*

*Proof.* Because all other joint angles are fixed, any such transition means that a joint $j_0$, $j_1$, $j_{p-1}$, or $j_p$ straightens, which stops the move. □

**Lemma 3.** *Joints $j_2, \ldots, j_{p-1}$ are convex.*

*Proof.* Consider such a joint $j_i$. Because $j_1$ is a rightmost reflex vertex, $j_i$ is convex if it is right of $j_1$. By monotonicity and the property that $j_i$ is right of ray$(j_0, j_1)$, $j_i$ cannot be left of $j_1$. The only case that remains is when $j_i$ has the same $x$ coordinate as $j_1$. Because we ignore straight joints, $j_i$ must in fact be $j_2$ in this case. Because $j_1$ is reflex, $j_2$ must be below $j_1$. Now $j_3$ must be strictly right of $j_2$, and hence the angle at $j_2$ is convex. □

Next we need a general result about quadrangles.

**Lemma 4.** [1] *Consider a simple quadrangle $j_0, j_1, j_2, j_3$ (in counterclockwise order) with $j_1$ reflex. (See Fig. 3.) Let $\theta_i$ denote the interior angle at joint $j_i$. If the linkage moves so that $j_1$ rotates clockwise about $j_0$, then $\theta_1$ decreases and $\theta_i$ increases for all $i \in \{0, 2, 3\}$, until $\theta_1$ straightens. In other words, all of the angles approach $\pi$.*

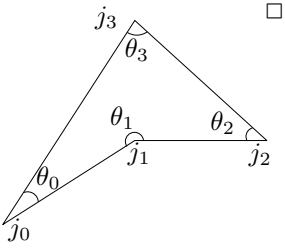



**Fig. 3.** Illustration of Lemma 4.

By the definition of $p$, $j_1$ is reflex in the quadrangle $Q = (j_0, j_1, j_{p-1}, j_p)$, so we can apply Lemma 4 to $Q$ and obtain the following result about our motion:

**Lemma 5.** *During each move in Step 2a, $j_{p-1}$ rotates counterclockwise about $j_p$, and the joint angles $\theta_1$ and $\theta_{p-1}$ both approach $\pi$.*

Next we analyze the movement of $j_1$ relative to $j_{p-1}$'s reference frame, determined by fixing the position of $j_{p-1}$ and keeping the axes parallel to the world frame's. This can be visualized by imagining that during the motion we translate the entire linkage so that $j_{p-1}$ stays in its original position.

**Lemma 6.** *$j_1$ rotates counterclockwise about $j_{p-1}$.*

*Proof.* Consider the relative movement of $j_1$ and $j_p$ about $j_{p-1}$. Joint $j_p$ is rotating counterclockwise about $j_{p-1}$ and the angle $\angle j_1 j_{p-1} j_p$ is increasing by Lemma 5. Hence, $j_1$ must also be rotating counterclockwise about $j_{p-1}$. □

We are now in the position to prove that the polygon remains simple and monotone throughout the motion.

*Proof (Theorem 1).* The only way that simplicity or monotonicity can be violated is that either a link intersects another link, or a vertical link rotates in the "wrong" direction. The wrong direction for link $(j_i, j_{i+1})$ on the lower [upper] chain is when $j_{i+1}$ becomes left [right] of $j_i$. Consider the first time at which a link intersects another, or a vertical link rotates in the wrong direction.

Suppose first that a vertical link $(j_i, j_{i+1})$ rotates in the wrong direction. Because only joints $j_1, \ldots, j_{p-1}$ move, we must have $0 \leq i \leq p-1$. We distinguish three cases:

**Case 1:** Link $(j_0, j_1)$ is vertical
Because $j_1$ is reflex and $j_2$ is nonstrictly right of $j_1$, $j_1$ must be above $j_0$. But $j_1$ rotates clockwise about $j_0$, so monotonicity is preserved.

**Case 2:** Link $(j_1, j_2)$ is vertical
Because $j_1$ is reflex, $j_1$ is above $j_2$. By Lemma 6, $j_1$ rotates counterclockwise about $j_{p-1}$. Hence, the rigid triangle $j_{p-1}j_1j_2$ rotates counterclockwise about $j_{p-1}$. Thus because the link $(j_1, j_2)$ is vertical, $j_1$ is above $j_2$, and $j_{p-1}$ is (nonstrictly) right of $j_1$, $j_1$ moves left of $j_2$. Thus, monotonicity is preserved.

**Case 3:** Link $(j_i, j_{i+1})$ is vertical, $2 \leq i \leq p-1$
We show that joints $j_i$ and $j_{i+1}$ are both convex. First, if $i \leq p-2$, then this follows by Lemma 3. Second, if $i = p-1$, then $j_{p-1}$ and $j_p$ must be right of $j_1$ in order for them to be on opposite sides of ray$(j_0, j_1)$. Hence, $j_{p-1}$ and $j_p$ must be convex because they are right of $j_1$ which is a rightmost reflex vertex.
Thus, joints $j_{i-1}$ and $j_{i+2}$ are both left of the link $(j_i, j_{i+1})$; that is, $(j_{i-1}, j_i)$ belongs to the lower chain and $(j_{i+1}, j_{i+2})$ belongs to the upper chain. This means that link $(j_i, j_{i+1})$ joins the top chain to the bottom chain (like link $BC$ in Fig. 1). The polygon remains monotone no matter which way the link moves.

Now suppose that two links intersect each other, but the polygon remained simple and monotone before this time. By Lemma 5, the joint angles $\theta_1$ and $\theta_{p-1}$ approach $\pi$, and hence the chain of moving links $(j_0, \ldots, j_p)$ cannot self-intersect. Hence, the only concern is whether any of these links could intersect the rest of the polygon.

In the following, refer to Fig. 4. Let $u$ denote the original position of ray$(j_0, j_1)$, and $v$ denote the downward vertical ray emanating from $j_0$. Let $W$ be the wedge right of ray $u$ and left of ray $v$.

Joint $j_1$ starts on the boundary of $W$; because it rotates clockwise about $j_0$, it enters region $W$ at the start of the move. By the choice of $p$, joints $j_2, \ldots, j_{p-1}$ all lie to the right of $u$. These joints must also lie to the left of $v$, because otherwise the polygon would not be monotone. Thus, after the move starts, the chain $j_1, \ldots, j_{p-1}$ lies inside $W$.

We now argue that the only joints that can be inside $W$ are $j_1, \ldots, j_{p-1}$. We know that $j_0$ and $j_p$ are not interior to $W$. If some other joint lies inside $W$, the chain must cross one of the boundaries of $W$. The chain cannot cross ray $v$, because that would violate monotonicity. Nor can the chain cross ray $u$, because that would require a reflex vertex to the right of $j_1$ or a violation of monotonicity. Because none of $j_p, \ldots, j_{n-1}, j_0$ are moving, this chain remains outside $W$ during the motion.

To establish that the polygon remains simple, we claim further that the joints $j_1, \ldots, j_{p-1}$ never leave $W$. Suppose to the contrary that one does. Let $j_i$ be the first such joint to leave $W$. It must reach either ray $u$ or ray $v$. Consider each possibility in turn.

$j_i$ **crosses** $v$**:** Because $j_1$ stays reflex (unless it straightens which stops the move), it cannot be the first to cross $v$. If $j_i$ crosses $v$ for $1 < i < p$, then we have both $j_0$ and $j_i$ left of $j_1$, so the chain is not monotone, a contradiction.

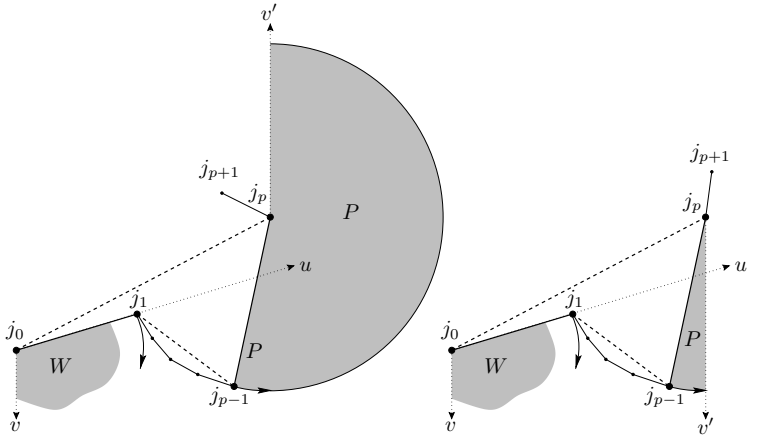



**Fig. 4.** Definition of $v$, $v'$, $W$, and $P$. (Left) When $(j_p, j_{p+1})$ is not on the lower chain. (Right) When $(j_p, j_{p+1})$ is on the lower chain.

$j_i$ **crosses** $u$**:** Because $j_1$ rotates clockwise about $j_0$, it never crosses $u$. If $j_2$ crosses, $j_1$ cannot be reflex, a contradiction. If $j_{p-1}$ crosses, the move must have already stopped from $j_0$, $j_1$, and $j_{p-1}$ becoming collinear because $j_1$ rotates clockwise about $j_0$. Finally, if $j_i$ crosses $u$ for $2 < i < p-1$, $j_i$ must be reflex because $j_{i-1}$ and $j_{i+1}$ are both right of $u$, contradicting Lemma 3.

Hence, the links of the chain $j_0, \ldots, j_{p-1}$ always remain inside $W$, so they cannot intersect links outside of $W$.

This leaves just one moving link, $(j_{p-1}, j_p)$. By Lemma 5, $j_{p-1}$ rotates counterclockwise about $j_p$. Define $v'$ to be the vertical ray emanating from $j_p$ that points away from the interior of the polygon, preferring upward if both directions are possible. Thus, $v'$ points downward [upward] when the link $(j_p, j_{p+1})$ is [not] on the lower chain.

Let $P$ be the pie wedge bounded by the link $(j_{p-1}, j_p)$, the ray $v'$, and the counterclockwise circular arc, centered at $j_p$, starting at $j_{p-1}$ and ending on $v'$ (at distance $\|j_p j_{p-1}\|$ from $j_p$). Because monotonicity is preserved up to this point, $P$ is empty of nonmoving links. $P$ also contains the entire sweep of $(j_{p-1}, j_p)$: if $v'$ points upward, $j_{p-1}$ cannot cross $v'$ without first becoming collinear with $j_0$ and $j_1$; if $v'$ points downward, it cannot cross without first straightening $j_p$. □

## 5 Time and Move Bounds

Finally we establish the time and move bounds on the algorithm. Our model of computation is a real random-access machine supporting comparisons, basic arithmetic, and square roots.

**Lemma 7.** *Each iteration of Step 2 takes constant time.*

*Proof.* The computations in this step are computing which of the five candidate events for stopping the motion occurs first (Step 2a), and updating $O(1)$ coordinate positions (Steps 2b and 2c).

In order to compute the halting event, we define the following (refer to Fig. 5). Here $x^*$ denotes that $x$ changes during the move.

$$a = \|j_0 j_1\|, \quad b = \|j_1 j_{p-1}\|, \quad c = \|j_{p-1} j_p\|,$$
$$d = \|j_p j_0\|, \quad m^* = \|j_1 j_p\|,$$
$$\varphi_0^* = \angle j_p j_0 j_1^*, \quad \varphi_1^* = \angle j_0 j_1 j_{p-1}^*,$$
$$\varphi_{p-1}^* = \angle j_1^* j_{p-1}^* j_p, \quad \varphi_p^* = \angle j_{p-1}^* j_p j_0.$$

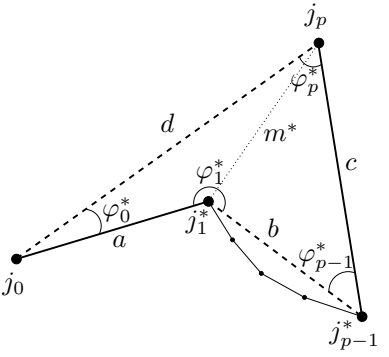



**Fig. 5.** Illustration of the proof of Lemma 7.

Note that $m^*$ increases during the move, because $j_1^*$ rotates clockwise about $j_0$, increasing $\varphi_0^*$ and thus (by law of cosines) $m^*$. Hence, we parameterize the move by the diagonal distance $m^*$. More precisely, we determine the halting event by computing the value of $(m^*)^2$ for each of the five candidate events, and choosing the event with smallest $(m^*)^2$.

Now the event that $j_i$ straightens happens when its angle $\theta_i^*$ is $\pi$, and the event that $j_0$, $j_1^*$, and $j_{p-1}^*$ become collinear happens when $\varphi_1^* = \pi$. Note however that $j_1^*$ can only straighten after $j_0$, $j_1^*$, and $j_{p-1}^*$ become collinear (because $\angle j_0, j_1^*, j_{p-1}^*$ is initially reflex), and hence we do not need to consider this event. Using the law of cosines, we obtain the following solutions to these event equations:

$j_0$ **straightens:** $(m^*)^2 = a^2 + d^2 + 2ad\cos(\angle j_{n-1} j_0 j_p)$ and $\angle j_{n-1} j_0 j_p < \pi$.
$j_{p-1}^*$ **straightens:** $(m^*)^2 = b^2 + c^2 + 2bc\cos(\angle j_{p-2} j_{p-1} j_1)$.
$j_p$ **straightens:** Provided $\angle j_0 j_p j_{p+1} < \pi$ (which is necessary for this event), $(m^*)^2$ is the solution of a quadratic polynomial involving $a$, $b$, $c$, $d$, and $\cos(\angle j_0 j_p j_{p+1})$. This reduces to an arithmetic expression involving square roots.
$j_0, j_1^*, j_{p-1}^*$ **collinear:** $(m^*)^2 = (ac^2 + bd^2 - ab(a+b))/(a+b)$, because $\cos\alpha_1^* = -\cos\beta_1^*$.

The new joint coordinates can be computed as follows: Suppose that we know the new coordinates of joints $j_{i-1}$ and $j_i$ (initially $i = 0$). Let $v$ be the vector from $j_i$ to $j_{i-1}$, rescaled to have the known length $\|j_i j_{i+1}\|$. Rotate this vector clockwise by the new $\theta_i$, which only involves cosines and sines of this angle, and then add it to the point $j_i$. The result is the new position of joint $j_{i+1}^*$. We can similarly compute the new coordinates of $j_{p-1}^*$ from the coordinates of $j_p$ and $j_{p+1}$. Thus, each update of $j_1^*$, $j_{p-1}^*$ (Step 2b), and possibly $j_{p-2}^*$ (Step 2c) takes constant time as desired. □

**Theorem 2.** *Algorithm Convexify computes $O(n^2)$ moves in $O(n^2)$ time.*

*Proof.* By Lemma 7, any one move takes $O(1)$ time to compute. Any execution of Step 2 therefore takes $O(n)$ time, because initially $p \le n-1$, and $p$ decreases with every move, until a joint straightens and Step 2 terminates. All other steps can also be implemented in $O(n)$ time. One iteration of the main loop thus takes $O(n)$ time. Because each iteration straightens a joint, the polygon becomes convex after $O(n)$ iterations. Hence, there are $O(n^2)$ moves, which are computed in $O(n^2)$ time. □

## 6 Conclusion

We have presented an $O(n^2)$-time algorithm to compute a sequence of $O(n^2)$ moves, each rotating the minimum possible number of four joints at once, that reconfigures a given monotone polygon into a convex polygon with the same link lengths. By running the algorithm twice we can find a motion between any two monotone polygons with the same clockwise sequence of link lengths.

Several interesting open problems remain. Can our algorithm be improved to use $o(n^2)$ moves each rotating $o(n)$ joints? More generally, what is the tradeoff between the number of simultaneously rotated joints and the number of moves?

Our result adds to the class of polygons that are known to be convexifiable; previously, the only nontrivial classes were star-shaped polygons [5] and monotone-separable polygons [3]. A natural area of research is to explore more general classes of polygons. Is there a convexifiable class containing both monotone and star-shaped polygons (other than trivial classes like the union)?

**Acknowledgement.** We thank William Lenhart, Anna Lubiw, Godfried Toussaint, and Sue Whitesides for helpful discussions. This work was partially supported by FCAR and NSERC.

## References

1. 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. Manuscript in preparation, 1999. A preliminary version appeared in Proc. 10th ACM-SIAM Sympos. Discrete Algorithms, 1999, 866-867.
2. Therese Biedl, Erik Demaine, Martin Demaine, Sylvain Lazard, Anna Lubiw, Joseph O'Rourke, Steve Robbins, Ileana Streinu, Godfried Toussaint, and Sue Whitesides. On reconfiguring tree linkages: Trees can lock. In *Proc. 10th Canadian Conf. Comput. Geom.*, Montréal, Aug. 1998.
3. Prosenjit Bose, William Lenhart, and Giuseppe Liotta. Personal comm., 1999.
4. Roxana Cocan and Joseph O'Rourke. Polygonal chains cannot lock in 4D. In *Proc. 11th Canadian Conf. Comput. Geom.*, Vancouver, Aug. 1999.
5. H. Everett, S. Lazard, S. Robbins, H. Schröder, and S. Whitesides. Convexifying star-shaped polygons. In *Proc. 10th Canadian Conf. Comput. Geom.*, Montréal, Aug. 1998.
6. W. J. Lenhart and S. H. Whitesides. Reconfiguring closed polygonal chains in Euclidean $d$-space. *Discrete Comput. Geom.*, 13:123–140, 1995.
7. Joseph O'Rourke. Folding and unfolding in computational geometry. In *Proc. Japan Conf. Discrete and Computational Geometry*, Tokyo, Dec. 1998. To appear.
8. Godfried Toussaint. The Erdős-Nagy theorem and its ramifications. In *Proc. 11th Canadian Conf. Comput. Geom.*, Vancouver, Aug. 1999.
9. Sue Whitesides. Algorithmic issues in the geometry of planar linkage movement. *Australian Computer Journal*, 24(2):42–50, May 1992.
10. Sue Whitesides. Personal communication, Oct. 1998.

# Bisecting Two Subsets in 3-Connected Graphs*

Hiroshi Nagamochi[1], Tibor Jordán[2], Yoshitaka Nakao[1], and Toshihide Ibaraki[1]

[1] Kyoto University, Kyoto, Japan 606-8501
{naga,ibaraki}@kuamp.kyoto-u.ac.jp
[2] BRICS, University of Aarhus, 8000 Aarhus C, Denmark.
jordan@daimi.au.dk

**Abstract.** Given two subsets $T_1$ and $T_2$ of vertices in a 3-connected graph $G = (V, E)$, where $|T_1|$ and $|T_2|$ are even numbers, we show that $V$ can be partitioned into two sets $V_1$ and $V_2$ such that the graphs induced by $V_1$ and $V_2$ are both connected and $|V_1 \cap T_j| = |V_2 \cap T_j| = |T_j|/2$ holds for each $j = 1, 2$. Such a partition can be found in $O(|V|^2)$ time. Our proof relies on geometric arguments. We define a new type of 'convex embedding' of -connected graphs into real space $\mathbf{R}^{k-1}$ and prove that for $= 3$ such embedding always exists.

## 1 Introduction

We define the following graph-partitioning problem: Given an undirected graph $G = (V, E)$ and $k$ subsets $T_1, \ldots, T_k$ of $V$, not necessarily disjoint, find a partition $V$ into $l$ subsets $V_1, \ldots, V_l$ such that $G[V_i]$ $(1 \leq i \leq l)$ are all connected and $a_{ij} \leq |V_i \cap T_j| \leq b_{ij}$ holds for each pair $i, j$, where $a_{ij}$ and $b_{ij}$ are prespecified lower and upper bounds. In this problem, we interpret each $T_i$ as a set of vertices which possess 'resource' $i$. In particular, one may ask for a partition where the vertices in each $T_j$ $(1 \leq j \leq k)$ are distributed among the subsets $V_1, \ldots, V_l$ as equally as possible.

In this paper, we consider the case of $l = 2$ and ask to distribute the resources equally: Given an undirected graph $G = (V, E)$ and $k$ subsets $T_1, T_2, \ldots, T_k$ of $V$, where $|T_j|$ is even for all $j$, find a bipartition $\{V_1, V_2\}$ of $V$ such that the graph $G[V_i]$ induced by each $V_i$ is connected and $|V_1 \cap T_j| = |V_2 \cap T_j|$ $(= |T_j|/2)$ holds for $j = 1, \ldots, k$. Let us call this problem *the $k$-bisection problem.* We prove that for every 3-connected graph $G$ and for every choice of resources $T_1$ and $T_2$, the 2-bisection problem has a solution.

To verify this, we reduce it to a geometrical problem. Our method is outlined as follows. We first prove that every 3-connected graph $G$ can be embedded in the plane in such a way that the convex hull of its vertices, which is a convex polygon, corresponds to a cycle $C$ of $G$, and every vertex $v$ in $V - V(C)$ is in the convex

* This research was partially supported by the Scientific Grant-in-Aid from Ministry of Education, Science, Sports and Culture of Japan, and the subsidy from the Inamori Foundation. Part of this work was done while the second author visited the Department of Applied Mathematics and Physics at Kyoto University, supported by the Monbusho International Scientific Research Program no. 09044160.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 425–434, 1999.
© Springer-Verlag Berlin Heidelberg 1999

hull of its neighbors (precise definition is given in Section 4). This will guarantee that, for any given straight line $L$ in the plane, each of the two subgraphs of $G$ separated by $L$ remains connected. Given such an embedding, we apply the 'ham-sandwich cut' algorithm, which is well known in computational geometry, to find a straight line $L^*$ that bisects the two subsets $T_1$ and $T_2$ simultaneously. Since the above embedding ensures that the two subgraphs separated by the ham-sandwich cut $L^*$ are connected, this bipartition of the vertices becomes a solution to the 2-bisection problem.

We give an algorithm which finds such a bisection in $O(|V|^2)$ time.

### 1.1 Related Results

If $l = 2$ and there is just one set $T$ of resources, the problem is NP-hard for general graphs, since it is NP-hard to test whether a given graph $G = (V, E)$ and an integer $n_1 < |V|$ have a partition of $V$ into two subsets $V_1$ and $V_2$ such that the graph induced by $V_i$ is connected for $i = 1, 2$, and $|V_1| = n_1$ holds [2]. When $G$ is 2-connected, it is known that such a partition $\{V_1, V_2\}$ always exists and it can be found in linear time [11,13]. More generally, the following result was shown independently by Győri [6] and Lovász [8].

**Theorem 1.1** [6,8] *Let $G = (V, E)$ be an $\ell$-connected graph, $w_1, w_2, \ldots, w_\ell \in V$ be different vertices and $n_1, n_2, \ldots, n_\ell$ be positive integers such that $n_1 + n_2 + \ldots + n_\ell = |V|$. Then there exists a partition $\{V_1, V_2, \ldots, V_\ell\}$ of $V$ such that $G[V_i]$ is connected, $|V_i| = n_i$ and $w_i \in V_i$ for $i = 1, \ldots, \ell$.* □

## 2 Preliminaries

Let $G = (V, E)$ stand for an undirected graph with a set $V$ of *vertices* and a set $E$ of *edges*, where we denote $|V|$ by $n$, $|E|$ by $m$. For a subgraph $H$ of $G$, the sets of vertices and edges in $H$ are denoted by $V(H)$ and $E(H)$, respectively. Let $X$ be a subset of $V$. The subgraph of $G$ induced by $X$ is denoted by $G[X]$. A vertex $v \in V - X$ is called a *neighbor* of $X$ if it is adjacent to some vertex $u \in X$, and the set of all neighbors of $X$ is denoted by $\Gamma_G(X)$. Let $e = (u, v)$ be an edge with end vertices $u$ and $v$. We denote by $G/e$ the graph obtained from $G$ by contracting $u$ and $v$ into a single vertex (deleting any resulting self-loop), and by $G - e$ the graph obtained from $G$ by removing $e$. *Subdividing* an edge $e = (u, v)$ means that we replace $e$ by a path $P$ from $u$ to $v$ where the inner vertices of $P$ are new vertices of the graph. If we obtain a graph $G'$ by subdividing some edges in $G$, then the resulting graph is called a *subdivision* of $G$. A graph $G$ is *k-connected* if and only if $|V| \geq k + 1$ and the graph $G - X$ obtained from $G$ by removing any set $X$ of $(k - 1)$ vertices remains connected. A singleton set $\{x\}$ may be simply written as $x$.

### 2.1 The Ham-Sandwich Theorem

Consider the $d$-dimensional space $\mathbf{R}^d$. For a non-zero $a \in \mathbf{R}^d$ and a real $b \in \mathbf{R}^1$, $H(a,b) = \{x \in \mathbf{R}^d \mid \langle a \cdot x \rangle = b\}$ is called a *hyperplane*, where $\langle a \cdot x \rangle$ denotes the inner product of $a, x \in \mathbf{R}^d$. Moreover, $H^+(a,b) = \{x \in \mathbf{R}^d \mid \langle a \cdot x \rangle \geq b\}$ (resp., $H^-(a,b) = \{x \in \mathbf{R}^d \mid \langle a \cdot x \rangle \leq b\}$) is called a *positive closed half space* (resp., *negative closed half space*) with respect to $H$.

Let $P_1, \ldots, P_d$ be $d$ sets of points in $\mathbf{R}^d$. We say that a hyperplane $H = H(a,b)$ in $\mathbf{R}^d$ *bisects* $P_i$ if $|H^+(a,b) \cap P_i| \leq \lfloor \frac{|P_i|}{2} \rfloor$ and $|H^-(a,b) \cap P_i| \leq \lfloor \frac{|P_i|}{2} \rfloor$. Thus, if $|P_i|$ is odd, then any bisector $H$ of $P_i$ contains at least one point of $P_i$. If $H$ bisects $P_i$ for all $i = 1, \ldots, d$, then $H$ is called a *ham-sandwich cut* with respect to the sets $P_1, \ldots, P_d$. The following theorem is well-known.

**Theorem 2.1** [3] *Given $d$ sets $P_1, \ldots, P_d$ of points in the $d$-dimensional space $\mathbf{R}^d$, there exists a hyperplane which is a ham-sandwich cut with respect to $P_1, \ldots, P_d$.* □

## 3 Bisecting $k$ Subsets in Graphs

Let $G = (V, E)$ be a graph and $T_1, T_2, \ldots, T_k$ be subsets of $V$, where $|T_j|$ is even for $j = 1, \ldots, k$. A bipartition $\{V_1, V_2\}$ of $V$ is a *$k$-bisection* if $G[V_i]$, $i = 1, 2$, are connected and $|V_1 \cap T_j| = |V_2 \cap T_j|$ $(= |T_j|/2)$ holds for $j = 1, 2, \ldots, k$.

One can observe that the $k$-connectivity of $G$ is not sufficient for the existence of a $k$-bisection. For $k = 1$ let $G = (V, E)$ be the complete bipartite graph $K_{2\ell-1,1}$ with $\ell \geq 2$ and let $T_1 = V$. Clearly, there exists no 1-bisection. For $k \geq 2$ consider the following example. Let $K_{2k-1,k} = (W \cup Z, E)$ be a complete bipartite graph with vertex sets $W = \{w_1, w_2, \ldots, w_{2k-1}\}$ and $Z = \{z_1, z_2, \ldots, z_k\}$, and let $T_i = W \cup \{z_i\}$ for $i = 1, \ldots, k$. Note that $|T_i| = 2k$ holds for all $i = 1, \ldots, k$. Suppose that $\{V_1, V_2\}$ is a $k$-bisection to $K_{k,2k-1}$ and $\{T_1, \ldots, T_k\}$. The set $T_1$ is bisected by $\{V_1, V_2\}$; $V_1 \cap T_1 = \{w_1, \ldots, w_k\}$ and $V_2 \cap T_1 = \{w_{k+1}, \ldots, w_{2k-1}, z_1\}$ can be assumed without loss of generality. Since $V_1 \cap T_i = \{w_1, \ldots, w_k\}$ also holds for each $i = 2, \ldots, k$, $V_1$ cannot contain any vertex in $Z$ and hence $V_1$ must be $\{w_1, \ldots, w_k\}$. However the induced subgraph $G[V_1]$ is not connected. Thus these graphs admit no $k$-bisection.

On the other hand, we propose the following conjecture.

**Conjecture 3.1** *Let $G = (V, E)$ be a $(k+1)$-connected graph and $T_1, T_2, \ldots, T_k$ be pairwise disjoint subsets of $V$, where $|T_j|$ is even for $j = 1, \ldots, k$. Then $G$ has a $k$-bisection.*

Conjecture 3.1 is true for $k = 1$ by using the so-called '$st$-numbering' of vertices, as observed in [13].

**Corollary 3.2** *Let $G = (V, E)$ be a 2-connected graph and $T$ be a subset of $V$ with even $|T|$. Then $G$ has a 1-bisection $\{V_1, V_2\}$, and such $\{V_1, V_2\}$ can be computed in $O(m)$ time.* □

The main result of this paper is an algorithmic proof for Conjecture 3.1 in the case of $k = 2$ (in a stronger form where the sets $T_1, T_2$ of resources need not be disjoint). Our algorithm finds a 2-bisection in a 3-connected graph in $O(n^2)$ time.

## 4 Strictly Convex Embeddings

In this section, we introduce a new way of embedding a graph $G$ in $\mathbf{R}^d$. The existence of such an embedding (along with the ham-sandwich cut theorem) will play a crucial role in the proof of Conjecture 3.1 for $k = 2$ in Section 5.

For a set $P = \{x_1, \ldots, x_k\}$ of points in $\mathbf{R}^d$, a point $x' = \alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_k x_k$ with $\sum_{i=1,\ldots,k} \alpha_i = 1$ and $\alpha_i \geq 0$, $i = 1, \ldots, k$ is called a *convex combination* of $P$, and the set of all convex combinations of $P$ is denoted by $Conv(P)$.

If $P = \{x_1, x_2\}$, then $Conv(P)$ is called a *segment* (connecting $x_1$ and $x_2$), denoted by $[x_1, x_2]$. A subset $S \subseteq \mathbf{R}^d$ is called a *convex set* if $[x, x'] \subseteq S$ for any two points $x, x' \in S$. For a convex set $S \subseteq \mathbf{R}^d$, a point $x \in S$ is called an *extreme point* if there is no pair of points $x', x'' \in S - x$ such that $x \in [x', x'']$. For two extreme points $x_1, x_2 \in S$, the segment $[x_1, x_2]$ is called an *edge* of $S$ if $\alpha x' + (1 - \alpha)x'' = x \in [x_1, x_2]$ for some $0 \leq \alpha \leq 1$ implies $x', x'' \in [x_1, x_2]$. The intersection $S$ of a finite number of closed half spaces is called a *convex polyhedron*, and is called a *convex polytope* if $S$ is non-empty and bounded.

Given a convex polytope $S$ in $\mathbf{R}^d$, the *point-edge graph* $G_S = (V_S, E_S)$ is defined to be an undirected graph with vertex set $V_S$ corresponding to the extreme points of $S$ and edge set $E_S$ corresponding to those pairs of extreme points $x, x'$ for which $[x, x']$ is an edge of $S$. For a convex polyhedron $S$, a hyperplane $H(a, b)$ is called a *supporting hyperplane* of $S$ if $H(a, b) \cap S \neq \emptyset$ and if $S \subseteq H^+(a, b)$ or $S \subseteq H^-(a, b)$. We say that a point $p \in S$ is strictly inside $S$ if there is no supporting hyperplane $H$ of $S$ containing $p$. If $S$ has a point strictly inside $S$ in $\mathbf{R}^d$, then $S$ is called *full-dimensional* in $\mathbf{R}^d$. We also denote by $Int(Conv(P))$ the set of points strictly inside $Conv(P)$.

Given a graph $G = (V, E)$, an *embedding* of $G$ in $\mathbf{R}^d$ is a mapping $f : V \to \mathbf{R}^d$, where each vertex $v$ is represented by a *point* $f(v) \in \mathbf{R}^d$, and each edge $e = (u, v)$ by a segment $[f(u), f(v)]$ (which may be written by $f(e)$). For two edges $e, e' \in E$, segments $f(e)$ and $f(e')$ may cross each other. For $\{v_1, v_2, \ldots, v_p\} = Y \subseteq V$, we denote by $f(Y)$ the set $\{f(v_1), \ldots, f(v_p)\}$ of points. For a set $Y$ of vertices, we denote $Conv(f(Y))$ by $Conv_f(Y)$.

We define a new kind of 'convex embedding' of a graph $G$ in the $d$-dimensional space:

**Definition 4.1** *Let $G$ be a graph without isolated vertices and let $G'$ be a subgraph of $G$. A* strictly convex embedding (*or* SC-embedding*, for short*) *of $G$ with* boundary *$G'$ is an embedding $f$ of $G$ into $\mathbf{R}^d$ in such a way that*

(i) *$\bigcup_{e \in E(G')} f(e)$ is the set of edges of a full-dimensional convex polytope $S$ in $\mathbf{R}^d$,*

(ii) $f(v) \in Int(Conv_f(\Gamma_G(v)))$ *holds for all vertices* $v \in V - V(G')$,
(iii) $f(v) \neq f(u)$ *for all pairs* $u, v \in V$.

It can be seen that the above definition implies that the extreme points of $Conv_f(V)$ are precisely the points in $f(V(G'))$. To satisfy (i), $G'$ must be a point-edge graph of the convex polytope $S$.

A similar concept of 'convex embeddings' of graphs, requiring only (ii) above, was introduced by Linial *et al.* in [7] and led to a new characterization of $k$-connected graphs. Their embedding, however, is not sufficient for our purposes.

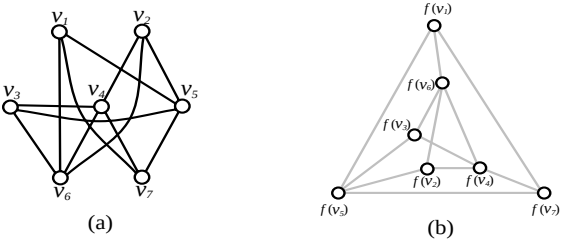



**Fig. 1.** An SC-embedding $f$ of the 3-connected graph $G_1$.

For a 3-connected graph $G_1$ and a cycle $C$ in Figure 1(a), Figure 1(b) illustrates an SC-embedding $f$ of $G$ with boundary $C$ in $\mathbf{R}^2$, where $V(C) = \{v_1, v_5, v_7\}$ and $E(C) = \{(v_1, v_5), (v_5, v_7), (v_1, v_7)\}$.

SC-embeddings into $\mathbf{R}^d$ have the following important property (the proof is omitted).

**Lemma 4.2** *Let* $G = (V, E)$ *be a graph without isolated vertices and let* $f$ *be an SC-embedding of* $G$ *into* $\mathbf{R}^d$. *Let* $f(V_1) = f(V) \cap H^+(a, b)$ *hold for some hyperplane* $H = H(a, b)$ *and suppose* $V_1 \neq \emptyset$. *Then* $G[V_1]$ *is connected.* □

## 5 SC-Embeddings in the Plane

In this section, we restrict ourselves to the embeddings in the two dimensional space $\mathbf{R}^2$. We prove that every 3-connected graph $G$ admits an SC-embedding $f$ with its boundary specified by an arbitrary cycle $C$, and consider how to find such an SC-embedding $f$ of $G$. For this, we use the following characterization of 3-connected graphs, due to Tutte.

**Lemma 5.1** [12] *Let* $G = (V, E)$ *be a* 3*-connected graph. For any edge* $e$, *either* $G/e$ *is* 3*-connected or* $G - e$ *is a subdivision of a* 3*-connected graph.* □

For two points $f(v')$ and $f(v'')$ in an embedding $f$ of $G$, we denote by $L(f(v'), f(v''))$ the half line that is obtained by extending segment $[f(v'), f(v'')]$ in the direction from $f(v')$ to $f(v'')$, and denote by $\hat{L}(f(v'), f(v''))$ the half line obtained from $L(f(v'), f(v''))$ by removing points in $[f(v'), f(v'')] - f(v'')$.

Let $f$ be an SC-embedding of a graph $G = (V, E)$, with boundary $C$, where we assume that for each $v \in V$, the order that the positions $f(u)$ for $u \in \Gamma_G(v)$ appear around $v$ is known. Let $v \in V - V(C)$, and consider another embedding $f'$ such that $f'(u) = f(u)$ for all $u \in V - v$ but $f'(v) \neq f(v)$, and also

$$f'(u) \in Int(Conv_{f'}(\Gamma_G(u))) \text{ for all } u \in V - V(C) - v. \tag{5.1}$$

(However $f'(v) \in Int(Conv_{f'}(\Gamma_G(v)))$ may not hold.) Clearly, the possible position $f'(v)$ depends only on the neighbors $u$ of $v$. For each neighbor $u \in \Gamma_G(v)$, we define a set $Cone_f(v, u) \subseteq \mathbf{R}^2$ as follows. If $f(u) \in Int(Conv_f(\Gamma_G(u) - v))$, then let $Cone_f(v, u) = \mathbf{R}^2$. Otherwise (if $f(u) \notin Int(Conv_f(\Gamma_G(u) - v))$), $f(u)$ becomes an apex (or on an edge) of $Conv_f(\Gamma_G(u) \cup \{u\} - v)$, and there are two edges $e_1 = (u, w_1)$ and $e_2 = (u, w_2)$ with $w_1, w_2 \in \Gamma_G(u) - v$ such that the acute angle $\alpha$ formed by segments $f(e_1)$ and $f(e_2)$ is the maximum (see Figure 2). Let $Cone_f(v, u)$ be the cone bounded by two half lines $\hat{L}(f(w_1), f(u))$ and $\hat{L}(f(w_2), f(u))$ (where points in these half lines are not included in $Cone_f(v, u)$). We define an open set

$$R_f^*(v) = \bigcap_{u \in \Gamma_G(v)} Cone_f(v, u).$$

Clearly, this is the set of the possible positions of $f'(v)$ if $f'$ satisfies (5.1). Observe that $R_f^*(v)$ is a non-empty open set, since $Cone_f(v, u)$ is an open set containing $f(v)$ for each $u \in \Gamma_G(v)$.

Note that $Cone_f(v, u)$ can be obtained in $O(|\Gamma_G(u)|)$ time since $Int(Conv_f(\Gamma_G(u) - v))$ can be computed in $O(|\Gamma_G(u)|)$ time based on the ordering that $f(w)$, $w \in \Gamma_G(u)$ appear around $u$. Summarizing the above argument, we have the next result.

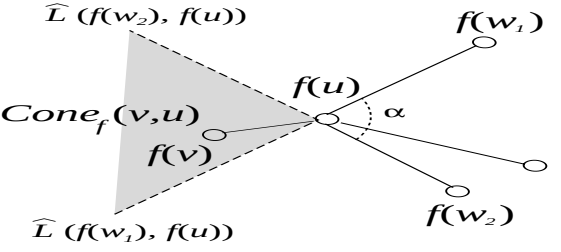



**Fig. 2.** Definition of $Cone_f(v, u)$.

**Lemma 5.2** *Given an SC-embedding $f$ of a graph $G = (V, E)$, with boundary $C$, and a vertex $v \in V - V(C)$, $R^*_f(v)$ is nonempty, and redefining $f(v)$ to any point in $R^*_f(v)$ preserves the property $f(u) \in Int(Conv_f(\Gamma_G(u)))$ for all $u \in V - V(C) - v$. Moreover $R^*_f(v)$ can be obtained in $O(\sum_{u \in \Gamma_G(v)} |\Gamma_G(u)|)$ time.* □

Based on Lemmas 5.1 and 5.2, we show the following theorem.

**Theorem 5.3** *For every 3-connected graph $G = (V, E)$ and every cycle $C$ of $G$, there exists an SC-embedding with boundary $C$. Such an SC-embedding can be found in $O(n^2)$ time.*

**Proof**. First we compute a sparse 3-connected spanning subgraph $G' = (V, E')$ of $G$ such that $E' \subseteq E$ and $|E'| = O(n)$ in linear time [9]. We put back all edges in $E(C)$ to $G'$ (if there is any missing edge in $C$). Clearly, an SC-embedding of $G'$ is also an SC-embedding of $G$.

We embed the vertices in $V(C)$ so that the cycle $C$ forms a convex polygon in the plane. An edge $e = (u, v)$ is called *internal* if $\{u, v\} \cap V(C) = \emptyset$. Then we apply the following procedure. We choose an arbitrary internal edge $e$ in $G'$, and if $G' - e$ is a subdivision of a 3-connected graph, then we remove the edge $e$ from $G'$. Moreover if there exist (one or two) vertices $v$ whose degree is two in $G' - e$, then, for each of such $v$, we replace the two incident edges $(u_1, v)$ and $(v, u_2)$ with a single edge $(u_1, u_2)$ and remove $v$. The resulting graph is denoted by $G' \overset{\bullet}{-} e$. Otherwise if $G' - e$ is not a subdivision of a 3-connected graph, then we contract the edge $e$. By Lemma 5.1, the resulting graph in both cases is 3-connected. Finally, we denote the resulting graph again by $G'$.

We repeat this procedure until there exists no internal edge in the current $G'$. In this case, each vertex $v \notin V(C)$ is easily embedded (independently) so that $f(v) \in Int(Conv_f(\Gamma_{G'}(v)))$ holds. Note that this procedure requires $O(n)$ 3-connectivity tests on graphs with $O(n)$ edges each (due to the sparsification). Thus it requires $O(n^2)$ time altogether.

Now we return the removed or contracted edges into $G'$ in the reverse order of the above procedure. As a general step, we consider how to construct an SC-embedding $f'$ of $G'$, assuming that an SC-embedding $f$ for a 3-connected graph $H = G' \overset{\bullet}{-} e$ or $H = G'/e$ is available, where $e = (v_1, v_2)$ is an internal edge in $G'$.

**Case (i)**: $H = G'/e$.

Let $v^*$ be the vertex in $G'/e$ created as a result of contracting the internal edge $e = (v_1, v_2)$. To regain $G'$, we insert $e$ in $G'/e$ after splitting $v^*$ into $v_1$ and $v_2$. To find an SC-embedding $f'$ of $G'$, we will determine the point $f'(v_2)$ while keeping $f'(v_1) = f(v^*)$ and $f'(u) = f(u)$ for $u \in V(G') - \{v_1, v_2\}$. Let $N_1 = \Gamma_{G'}(v_1) - v_2$ and $N_2 = \Gamma_{G'}(v_2) - v_1$. If $f(v^*) \in Int(Conv_f(N_1))$ in $f$ (hence $f'(v_1) = f(v^*) \in Int(Conv_{f'}(N_1))$ in $f'$), then let $R_2 = \mathbf{R}^2$. Otherwise (if $f(v^*) \notin Int(Conv_f(N_1))$), we let $R_2 = Cone_f(v_2, v_1)$ (see Figure 3(a)) as defined

before Lemma 5.2. This $R_2$ contains no points in $f(N_1)$, and if $f'(v_2) \in R_2$ is chosen, then $f'(v_1)$ $(= f(v^*)) \in Int(Conv_{f'}(\Gamma_{G'}(v_1)))$ holds.

Also to satisfy $f'(v_2) \in Int(Conv_{f'}(\Gamma_{G'}(v_2)))$ $(= Int(Conv_f(N_2 \cup \{v^*\})))$ (see Figure 3(b)), we choose a point $f'(v_2) \in R_2 \cap Int(Conv_f(N_2 \cup \{v^*\})))$ (which is not empty, since otherwise $f(v^*)$ would be an apex of the convex hull $Int(Conv_f(\Gamma_{G'/e}(v^*))))$.

Therefore by Lemma 5.2, if $f'(v_2)$ is chosen from $R_2 \cap Int(Conv_f(N_2 \cup \{v^*\}))) \cap R_f^*(v^*)$, then the resulting embedding $f'$ is an SC-embedding of $G'$. Observe that $R_2 \cap Int(Conv_f(N_2 \cup \{v^*\})) \cap R_f^*(v^*)$ is a non-empty open set, and hence such a choice (satisfying $f'(v_2) \neq f'(v)$ for all $v \in V(G') - v_2$, as well) is possible.

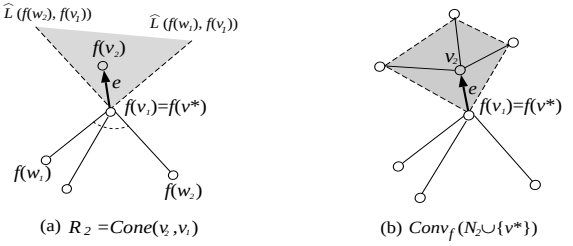



**Fig. 3.** Illustration for $R_2 = Cone_f(v_2, v_1)$ and $Int(Conv_f(N_2 \cup \{v^*\}))$.

**Case (ii)**: $H = G' \overset{\bullet}{-} e$. This case can be treated by a similar argument in (i) (the proof is omitted). From the above argument and by Lemma 5.2, given an SC-embedding $f$ of $G' \overset{\bullet}{-} e$ (or $G'/e$), we can construct an SC-embedding $f'$ of $G'$ in $O(|E'|)(= O(n))$ time. Since this procedure is executed $O(n)$ times to construct an SC-embedding of the original graph $G$, the entire running time for finding an SC-embedding $f$ of $G$ is $O(n^2)$. □

As shown by Theorem 5.3, the 3-connectivity is a sufficient condition for a graph $G$ to have an SC-embedding. We can show that the problem of testing whether a general graph $G$ admits an SC-embedding is NP-hard (the proof is omitted).

**Theorem 5.4** *The problem of deciding whether $G = (V, E)$ has an SC-embedding is NP-hard.* □

## 6 Finding a 2-Bisection in a 3-Connected Graph

By combining the algorithmic proof of Theorem 5.3 and the ham-sandwich cut algorithm in two dimensions, we are now able to obtain a polynomial time al-

gorithm that finds a 2-bisection in a 3-connected graph. This also implies that Conjecture 3.1 is true for $k = 2$.

**Algorithm BISECTION** $(G, T_1, T_2)$
**Input**: A 3-connected graph $G$ and subsets $T_1, T_2 \subseteq V$ such that $|T_i|$ $(i = 1, 2)$ are even.
**Output**: A 2-bisection $\{V_1, V_2\}$ of $(G, T_1, T_2)$.

1. Choose an arbitrary cycle $C$, and construct an SC-embedding $f$ with boundary $C$ in $G$ in $O(n^2)$ time by Theorem 5.3.
2. Let $W = \{f(v) \mid v \in T_1\}$ and $B = \{f(v) \mid v \in T_2\}$, and let $W' = W \cup \{a^*\}$ and $B' = B \cup \{a^*\}$ by adding a dummy point $a^* \in \mathbf{R}^2$ (hence both $|W'|$ and $|B'|$ are odd integers). By applying the ham-sandwich cut algorithm [1,4] to $W'$ and $B'$, compute a ham-sandwich cut $L$ such that each side of $L$ contains $(|W'|-1)/2$ points in $W'$ and $(|B'|-1)/2$ points in $B'$ (and hence one point $p_1 \in W'$ and one point $p_2 \in B'$ are on $L$).
3. If $p_1 = p_2 = a^*$, then output the current bipartition $\{V_1, V_2\}$ of $V$ generated by $L$.
4. Otherwise, if $p_1 = p_2 \neq a^*$ or $p_1 \neq p_2$ (hence $a^* \notin \{p_1, p_2\}$), then define $p_1$ and $p_2$ to be on that side of $L$ which contains $a^*$. Output the resulting bipartition $\{V_1, V_2\}$ of $V$, neglecting $a^*$.

Note that every point not on the boundary $C$ is located *strictly* inside the convex hull of its neighbors in an SC-embedding $f$ in Step 1. As mentioned in Section 2.3, the ham-sandwich algorithms [1,4] may perturb some points in an input $f(V)$ by a small amount $\epsilon$. This, however, does not affect the use of Lemma 4.2 because such $\epsilon$ can be chosen arbitrarily small. Therefore, by applying Lemma 4.2 to the SC-embedding in Step 3 or 4, we obtain a partition $V_1$ and $V_2$, where each $V_i$ induces a connected subgraph of $G$. Since both $T_1$ and $T_2$ are equally divided by $\{V_1, V_2\}$, the output $\{V_1, V_2\}$ is a 2-bisection of $G$ with respect to $T_1$ and $T_2$.

An SC-embedding $f$ in Step 1 can be obtained in $O(n^2)$ time by Theorem 5.3. Steps 2 and 3 can be done in $O(n+m)$ time using the linear time ham-sandwich cut algorithm [1]. From the above discussion, the next theorem is established.

**Theorem 6.1** *Let $G = (V, E)$ be a 3-connected graph. Then there exists a 2-bisection in $G$, and such bisection can be computed in $O(n^2)$ time.* □

## 7 Remarks

If a graph $G$ is isomorphic to a point-edge graph of a convex polytope in $\mathbf{R}^3$, then $G$ is called *polyhedral*. The following characterization of polyhedral graphs is well-known.

**Lemma 7.1** [10] *A graph $G$ is a polyhedral graph if and only if $G$ is 3-connected and planar.* □

In order to prove Conjecture 3.1 for $k = 3$ by a similar application of the 3-dimensional ham-sandwich theorem (see Theorem 2.1 and Lemma 4.2), it would be sufficient to prove that every 4-connected graph $G$ has an SC-embedding $f : V \to \mathbf{R}^3$. In such an embedding the boundary $G'$ should be a polyhedral subgraph of $G$ by Definition 4.1(i). We conjecture that such an embedding exists (for every proper choice of the boundary).

**Conjecture 7.2** *For a 4-connected graph $G = (V, E)$ and any polyhedral subgraph $G'$ of $G$, there is an SC-embedding $f : V \to \mathbf{R}^3$ with boundary $G'$.* □

As opposed to 3-connected graphs, it is not clear whether every 4-connected graph has a subgraph which can be chosen to be the boundary of an SC-embedding.

**Conjecture 7.3** *Every 4-connected graph $G = (V, E)$ has a 3-connected planar subgraph $G' = (V', E')$.* □

Conjectures 7.2 and 7.3 together would imply the existence of a 3-bisection in a 4-connected graph, that is, would verify Conjecture 3.1 for $k = 3$.

## References

1. L. Chi-Yuan, J. Matoušek and W. Steiger, *Algorithms for ham-sandwich cuts*, Discrete Comput. Geom., 11, 1994, 433–452.
2. M. E. Dyer and A. M. Frieze, *On the complexity of partitioning graphs into connected subgraphs*, Discrete Applied Mathematics, 10, 1985, 139–153.
3. H. Edelsbrunner, Algorithms in Combinatorial Geometry, Springer-Verlag, Berlin, 1987.
4. H. Edelsbrunner and R. Waupotitsch, *Computing a ham-sandwich cut in two dimensions*, J. Symbolic Computation, 2, 1986, 171–178.
5. M. R. Garey, D. S. Johnson and R. E. Tarjan, *The planar Hamiltonian circuit problem is NP-complete*, SIAM J. Comput., 5, 1976, 704–714.
6. E. Győri, *On division of connected subgraphs*, Combinatorics (Proc. Fifth Hungarian Combinatorial Coll, 1976, Keszthely), Bolyai-North-Holland, 1978, 485–494.
7. N. Linial, L. Lovász and A. Wigderson, *Rubber bands, convex embeddings and graph connectivity*, Combinatorica, 8, 1988, 91–102.
8. L. Lovász, *A homology theory for spanning trees of a graph*, Acta Math. Acad. Sci. Hungar, 30, 1977, 241–251.
9. H. Nagamochi and T. Ibaraki, *A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph*, Algorithmica, 7, 1992, 583–596.
10. E. Steinitz, *Polyeder und Raumeinteilungen*, Encyklopädie der mathematischen Wissenschaften, Band III, Teil 1, 2. Hälfte, IIIAB12, 1916, 1–139.
11. H. Suzuki, N. Takahashi and T. Nishizeki, *A linear algorithm for bipartition of biconnected graphs*, Information Processing Letters, 33, 1990, 227–232.
12. W.T. Tutte, Connectivity in Graphs, University of Toronto Press, 1966.
13. K. Wada and K. Kawaguchi, *Efficient algorithms for tripartitioning triconnected graphs and 3-edge-connected graphs*, Lecture Notes in Comput. Sci., 790, Springer, Graph-theoretic concepts in computer science, 1994, 132–143.

# Generalized Maximum Independent Sets for Trees in Subquadratic Time

B. K. Bhattacharya[1] and M. E. Houle[2]

[1] School of Computing Science, Simon Fraser University
Burnaby, BC, Canada V5A 1S6
binay@cs.sfu.ca
[2] Basser Department of Computer Science
The University of Sydney, Sydney, NSW 2006, Australia
meh@cs.usyd.edu.au

**Abstract.** In this paper we consider a special case of the *Maximum Weighted Independent Set* problem for graphs: given a vertex- and edge-weighted tree $\mathcal{T} = (V, E)$ where $|V| = n$, and a real number $b$, determine the largest weighted subset $P$ of $V$ such that the distance between the two closest elements of $P$ is at least $b$. We present an $O(n \log^3 n)$ algorithm for this problem when the vertices have unequal weights. The space requirement is $O(n \log n)$. This is the first known subquadratic algorithm for the problem. This solution leads to an $O(n \log^4 n)$ algorithm to the previously-studied *Weighted Max-Min Dispersion Problem*.

## 1 Introduction

Many problems in location theory deal with the placement of facilities on a network so as to maximize or minimize certain functions of distances between the facilities, or between facilities and the nodes of the network. One family of location problems deals with *dispersion* — the placement of sets of facilities within the network so as to maximize the distances among the facilities (see, for example, [4,10,13,16,18]), or the sizes of structures associated with these distances (such as the minimum spanning trees, Steiner trees, and Hamiltonian tours studied in [9]). Mathematical models for dispersion assume that the given network is represented by a graph $G = (V, E)$ where each edge $e \in E$ has edge length $d(e)$. For vertices $x, y \in V$, let $d(x, y)$ denote the shortest path between $x$ and $y$ in $G$.

The following *Weighted Max-Min Dispersion Problem* (WMMDP) was discussed in [3,4,11,13].

**Instance**
A graph $G = (V, E)$, vertex weights $\omega(v) \geq 0$ for all $v \in V$, edge lengths $d(e) \geq 0$ for all $e \in E$, and a real number $W \geq 0$.

**Requirement**
Find a subset $P$ of $V$ such that $\omega(P) \geq W$ and $f(P) = \min_{\forall x,y \in P; x \neq y} d(x, y)$ is maximized.

A. Aggarwal, C. Pandu Rangan (Eds.): ISAAC'99, LNCS 1741, pp. 435–445, 1999.
© Springer-Verlag Berlin Heidelberg 1999

Like most other variants of dispersion problems, the WMMDP problem is *NP-hard* for general graphs [3,5]. Recently, when the underlying graph is a tree, Bhattacharya and Houle [1] gave a simple $O(n)$ algorithm for the unweighted version of the problem. However, the weighted version takes $O(n^2)$ time and space to solve. Clearly, WMMDP is polynomially solvable for network $G$ if the following problem, called WMinGap, is polynomially solvable.

**Instance**
A graph $G = (V, E)$, vertex weights $\omega(v) \geq 0$ for all $v \in V$, edge lengths $d(e) \geq 0$ for all $e \in E$, and a real number $b > 0$.

**Requirement**
Find a largest weighted subset $P$ of $V$ such that $f(P) = \min_{\forall x,y \in P; x \neq y} d(x, y) \geq b$.

In this paper we present a solution to the WMinGap problem. The proposed WMinGap algorithm requires $O(n \log^3 n)$ time and $O(n \log n)$ space in the worst case, which implies that *Weighted Max-Min Dispersion Problem* (WMMDP) can be solved in $O(n \log^4 n)$ time for an acyclic network.

In the general setting, WMinGap and WMMDP were recently investigated by Rosenkrantz, Tayi and Ravi [14], along with a number of related problems in both graph and geometric settings. Although both WMinGap and WMMDP are NP-hard problems, in the case where the underlying network is a linked list, they showed that the problems may be solved in $O(n \log n)$ time and $O(n^2 \log n)$ time, respectively. A recent version of [14] indicates that both WMinGap and WMMDP can be solved in $O(n \log n)$ time for the linked list case.

In the next section, we shall present some of the terminology to be used in the paper. In Section 3, the algorithm for the WMinGap problem will be presented. Concluding remarks appear in Section 4.

## 2 Notation and Preliminaries

For our discussion of the WMinGap problem, we shall assume that the input $\mathcal{T} = (V, E)$ is a binary tree rooted at $r$ — otherwise, $\mathcal{T}$ can be converted into a binary tree by replacing each vertex $v$ of degree $\delta > 3$ by a binary tree of $\delta - 2$ dummy vertices joined by edges of length zero (the particular shape of this 'dummy tree' is unimportant).

For the WMinGap problem, a (valid) *placement* $P$ is a subset of $V$ such that no two vertices $x, y \in P$ have distance $d(x, y) < b$ in $\mathcal{T}$. The *weight* $\omega(P)$ of a placement $P \subseteq V$ is defined as the sum of the individual vertex weights $\omega(v)$ over all $v \in P$. If $P$ is a subset of the subtree of $\mathcal{T}$ rooted at $r$, then we define the *depth* $\Delta(P, r)$ of $P$ with respect to $r$ to be $\min_{v \in P} d(v, r)$, the minimum distance from $r$ to the vertices of $P$.

## 3 Weighted Min-Gap Problem

When the vertices of $\mathcal{T}$ have equal weight, Bhattacharya and Houle [1] showed that the greedy strategy of always building upon the deepest optimal placements

of subtrees leads to a simple and efficient algorithm for finding the deepest optimal placement of their parent. If the vertices of $\mathcal{T}$ are allowed to have arbitrary non-negative weights, the greedy strategy fails.

The strategy which will be used to solve the WMinGap problem is that of dynamic programming. Instead of maintaining a single 'best' choice of optimal placement for each subtree of $\mathcal{T}$, we will instead maintain a list of candidate valid placements for each subtree. The list associated with the subtree rooted at vertex $v$ will be constructed by merging the lists associated with its left and right children.

During each merge, placements for subtrees which cannot contribute to an optimal placement for $\mathcal{T}$ must be pruned. The following lemma states a condition by which such suboptimal placements may be identified.

**Lemma 1.** *Let $\mathcal{T}$ be a binary tree of $n$ vertices rooted at $r$ such that all vertices and all edges are assigned a non-negative real weight, and let $P$ be a valid placement for $\mathcal{T}$. Let $v$ be a vertex of $\mathcal{T}$, and let $P^v$ be the set of those vertices of $P$ located in the subtree $\mathcal{T}^v$ rooted at $v$. If $P'_v$ is a valid placement for $\mathcal{T}^v$ such that $\Delta(P'_v, v) \geq \Delta(P^v, v)$ and $\omega(P'_v) > \omega(P^v)$, then $P$ cannot be an optimal placement for $\mathcal{T}$.*

Lemma 1 allows us to maintain a list of candidate placements for the subtree $\mathcal{T}^v$ rooted at $v$ in which the depths of the placements appear in increasing order, and the weights of the placements appear in decreasing order. The lemma implies that if any placement for $\mathcal{T}^v$ cannot be inserted into the list without violating these orders, the placement cannot be part of an optimal placement for $\mathcal{T}$, and shall hence be referred to as *redundant*.

**Lemma 2.** *Let $\mathcal{T}$ be a binary tree of $n$ vertices rooted at $r$ such that all vertices and all edges are assigned a non-negative real weight, and let $P$ be a valid placement for $\mathcal{T}$. Let $v$ be a vertex of $\mathcal{T}$, and let $P^v$ be the set of those vertices of $P$ located in the subtree $\mathcal{T}^v$ rooted at $v$. If $\Delta(P^v, v) \geq b$, and if $P'_v$ is a valid placement for $\mathcal{T}^v$ such that $\Delta(P^v, v) \geq \Delta(P'_v, v) \geq b$ and $\omega(P^v) < \omega(P'_v)$, then $P$ cannot be an optimal placement for $\mathcal{T}$.*

Any placement $P^v$ as defined in Lemma 2 will also be said to be *redundant*. The lemma implies that at most one placement of depth $b$ or greater need be kept for any subtree of $\mathcal{T}$; moreover, if different placements of identical weight exist, it suffices to keep one placement of greatest depth.

At each vertex $v$ of $\mathcal{T}$, information regarding all possible non-redundant placements will be stored. By maintaining the invariant that each vertex traversed stores a representation of all non-redundant placements, when the algorithm terminates, the optimal placement will be one of the non-redundant placements associated with $r$, the root of $\mathcal{T}$. In addition to its weight $\omega(v)$, the information stored at node $v$ consists of:

- $\mathsf{dep}(v)$
  A list of the depths of candidate placements for $\mathcal{T}^v$, sorted in increasing order. Only the last entry of $\mathsf{dep}(v)$ is allowed to be greater than or equal to $b$. Initially, the list is empty.

- $\mathsf{wt}(v)$
  A list of the weights of candidate placements for $\mathcal{T}^v$, sorted in decreasing order. The $i$th entry in lists $\mathsf{wt}(v)$ and $\mathsf{dep}(v)$ correspond to the same candidate placement. Initially, the list is empty.
- $P^v$
  A list of candidate placements where $\mathsf{dep}(v)[i] = d(v, P^v[i])$.

In addition, the lengths of the edges of $\mathcal{T}$ adjacent to $v$ are assumed to be accessible from $v$.

### 3.1 Compact and Combine

Before presenting the main algorithm, we will first discuss two of the operations to be performed on the lists maintained at nodes of $v$: Compact and Combine. The Compact operation involves traversing the list $\mathsf{wt}(v)$, and deleting any entry whose successor in the list is greater than or equal to it, thus ensuring that $\mathsf{wt}(v)$ is kept in decreasing order. It also ensures that only one entry having depth $\mathsf{dep}(v)[i] \geq b$ is maintained — the entry having largest weight $\mathsf{wt}(v)[i]$. Whenever entry $i$ is to be purged, it is simultaneously removed from the three lists $\mathsf{wt}(v)$, $\mathsf{dep}(v)$ and $P^v$.

Compact can be performed in time proportional to the size of the lists. Lemmas 1 and 2 justify the use of Compact on lists whenever the entries of $\mathsf{dep}(v)$ are in non-decreasing order.

The Combine operation is somewhat more complex, and is at the heart of Algorithm WMinGap. Let $u$ and $w$ be the left and right children of vertex $v$, and consider the lists $\mathsf{dep}(u)$, $\mathsf{wt}(u)$ and $P^u$; and $\mathsf{dep}(w)$, $\mathsf{wt}(w)$ and $P^w$ as described above. The object of Combine is to construct lists $\mathsf{dep}(v)$, $\mathsf{wt}(v)$ and $P^v$ which reflects all non-redundant valid placements for $\mathcal{T}^v$ which do not include $v$ itself.

Let us assume that before performing Combine, Compact has been performed on the lists, and that at most one entry in $\mathsf{dep}(u)$ and $\mathsf{dep}(w)$ is greater than or equal to $b$ (the last entry in each).

The first step of Combine involves making copies of the above lists, which we denote $\mathsf{dep}^*(u)$, $\mathsf{wt}^*(u)$ and $P^*_u$; and $\mathsf{dep}^*(w)$, $\mathsf{wt}^*(w)$ and $P^*_w$. We next add $d(u, v)$ to each entry of $\mathsf{dep}^*(u)$, and $d(w, v)$ to each entry of $\mathsf{dep}^*(w)$, so that each list now reflect depths relative to $v$.

There are three ways in which a placement $P^u[i]$ under $u$ can be combined with a placement $P^w[j]$ under $w$ to yield placements for $\mathcal{T}^v$.

- $\mathsf{dep}^*(u)[i] \leq \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] \geq \frac{b}{2}$,
- $\mathsf{dep}^*(u)[i] \geq \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] \leq \frac{b}{2}$,
- $\mathsf{dep}^*(u)[i] > \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] > \frac{b}{2}$.

The fourth possibility, $\mathsf{dep}^*(u)[i] < \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] < \frac{b}{2}$, leads to an invalid placement for $\mathcal{T}^v$, and is not considered.

The cases can handled as follows:

1. $\mathsf{dep}^*(u)[i] \leq \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] \geq \frac{b}{2}$.
   Here, a placement $P^u[i]$ with $\mathsf{dep}^*(u)[i] \leq \frac{b}{2}$ can only be combined with placements $P^w[j]$ for those $j$ such that $\mathsf{dep}^*(w)[j] \geq b - \mathsf{dep}^*(u)[i]$. The combined weight of the placement would then be $\mathsf{wt}^*(u)[i] + \mathsf{wt}^*(w)[j]$, and the depth of the placement would be $\mathsf{dep}^*(u)[i]$. Since the entries of $\mathsf{wt}^*(w)$ appear in decreasing order, Lemma 1 implies that of all the placements $P^w[j]$ satisfying $\mathsf{dep}^*(w)[j] \geq b - \mathsf{dep}^*(u)[i]$, only the one with smallest index $j$ need be considered — all other choices of $j$ lead to redundant placements. Therefore, for each $P^u[i]$, we determine the smallest index $j$ such that $\mathsf{dep}^*(w)[j] \geq \mathsf{dep}^*(u)[i]$ and $\mathsf{dep}^*(w)[j] \geq b - \mathsf{dep}^*(u)[i]$.
2. $\mathsf{dep}^*(u)[i] \geq \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] \leq \frac{b}{2}$.
   In this case, a placement $P^w[j]$ with $\mathsf{dep}^*(w)[j] \leq \frac{b}{2}$ can only be combined with placements $P^u[i]$ for those $i$ such that $\mathsf{dep}^*(u)[i] \geq b - \mathsf{dep}^*(w)[j]$. This case is entirely symmetric to the previous case, and in a similar manner output lists can be constructed which describe all non-redundant valid placements resulting from combinations involving $P^w[j]$ where $\mathsf{dep}^*(w)[j] \leq \frac{b}{2}$.
3. $\mathsf{dep}^*(u)[i] > \frac{b}{2}$ and $\mathsf{dep}^*(w)[j] > \frac{b}{2}$.
   In the third case, any placement $P^u[i]$ with $\mathsf{dep}^*(u)[i] \geq \frac{b}{2}$ can be combined freely with placements $P^w[j]$ with $\mathsf{dep}^*(w)[j] \geq \frac{b}{2}$. However, if $P^u[i]$ has depth in $\mathcal{T}^v$ less than or equal to that of $P^w[j]$, then as in the first case, Lemma 1 indicates that of all the placements $P^w[j]$ satisfying $\mathsf{dep}^*(w)[j] \geq \mathsf{dep}^*(u)[i]$, only the one with smallest index $j$ need be considered — all other choices of $j$ lead to redundant placements.

It is clear from above that it is possible to combine $\mathsf{dep}(u)$, $\mathsf{wt}(u)$ and $P^u$ with $\mathsf{dep}(w)$, $\mathsf{wt}(w)$ and $P^w$ to obtain $\mathsf{dep}(v)$, $\mathsf{wt}(v)$ and $P^v$ in time proportional to the size of the lists involved. This results in a $O(n^2)$ algorithm for the WMinGap problem. The storage space requirement is also $O(n^2)$.

### 3.2 Modified Combine Algorithm

We now present a modification of Combine which runs in subquadratic time.

**Intervals of Placements** Let $v$ be a node of the tree, and let $u$ and $w$ be its left and right children of a vertex $v$. We assume that the lists $\mathsf{dep}(u)$, $\mathsf{wt}(u)$, $P^u$, $\mathsf{dep}(w)$, $\mathsf{wt}(w)$ and $P^w$ are available. Let $n_u$ and $n_w$ be the sizes of $P^u$ and $P^w$, respectively. We are interested in obtaining the lists $\mathsf{dep}(v)$, $\mathsf{wt}(v)$ and $P^v$.

Let $S^v$ be the sorted sequence of the distances from $v$ of the elements of $\mathcal{T}^v$. Let $s_i^v$ be the element with rank $i$ in $S^v$; that is, the distance of $s_i^v$ from $v$ is the $i$th smallest in $S^v$. Let $r_i^v$ be the rank of $\mathsf{dep}(u)[i] + d(u, v)$ in $S^v$. Clearly, $r_i^v < r_{i+1}^v$ for each $i$. Let $r_j^v$ be the rank of $\mathsf{dep}(w)[j] + d(w, v)$ in $S^v$. Again, $r_j^v < r_{j+1}^v$ for each such $j$. In general, there is no such relationship between $r_i^v$ and $r_j^v$.

Each placement $P^u[i]$ of $u$ and $P^w[j]$ of $w$ can be ranked according to the distances from $v$ of their elements closest to $v$. More precisely, the rank $\mathsf{rank}(P^u[i])$ of $P^u[i]$ is the rank in $S^v$ of the element $x \in P^u[i]$ such that $d(v,x) = d(v,u) + \mathsf{dep}(u)[i]$. The element $x$ shall be called the *depth element* $\mathsf{depelt}(P^u[i])$ of $P^u[i]$.

Consider now the set $X$ of placements $P^w[j]$ such that

1. $d(v,w) + \mathsf{dep}(w)[j] \leq d(v,u) + \mathsf{dep}(u)[i]$, and
2. $\mathsf{dep}(w)[j] + d(v,w) + d(v,u) + \mathsf{dep}(u)[i] \geq b$.

Clearly, any such placement $P^w[j]$ can be combined with $P^u[i]$. Moreover, the depth of the resulting set would be determined by $\mathsf{depelt}(P^w[j])$. Let $r^-_{(i)}$ and $r^+_{(i)}$ be the minimum and the maximum ranks in $S^v$ of placements $P^w[j]$ satisfying the above conditions with respect to $P^u[i]$. The placements of $P^w$ with ranks in the interval $[r^-_{(i)}, r^+_{(i)}]$ are precisely the set $X$.

Let $[r^-_{(h)}, r^+_{(h)}]$ be the interval of ranks obtained with respect to $P^u[h]$, for any $h > i$. It is possible that the intervals $[r^-_{(i)}, r^+_{(i)}]$ and $[r^-_{(h)}, r^+_{(h)}]$ overlap. Let $x$ be an element which belongs to both. Since a placement of $P^w$ with depth element $x$ can be combined with both $P^u[i]$ and $P^u[h]$, and since $\mathsf{wt}(u)[i] > \mathsf{wt}(u)[h]$, Lemma 1 implies that we need not consider combining $x$ with $P^u[h]$. Therefore, for each $P^u[i]$ we consider only the subset $X^u_{(i)} \subseteq P^w$ such that for any $x \in X^u_{(i)}$:

1. $d(v,w) + \mathsf{dep}(w)[j] \leq d(v,u) + \mathsf{dep}(u)[i]$,
2. $\mathsf{dep}(w)[j] + d(v,w) + d(v,u) + \mathsf{dep}(u)[i] \geq b$, and
3. $x$ is not an element of any $X_{(h)}, h < i$.

Redefined so that all three of these conditions are satisfied, the interval of ranks $[r^-_{(i)}, r^+_{(i)}]$ shall be denoted $I^u(i)$.

The interval $[r^-_{(i)}, r^+_{(i)}]$ represents the set of depth elements in $\mathcal{T}^w$ with which $P^u[i]$ can be profitably combined while keeping the depth element in $\mathcal{T}^w$. By attaching to this interval the weight of $P^u[i]$, we obtain a description of how $P^u[i]$ can contribute towards placements with depth elements located in $\mathcal{T}^w$. This will be particularly useful when considering a chain of subtrees as in Figure 1: best placements with depth elements in subtree $\mathcal{T}^{v_j}$ can be combined with placements from subtrees $\mathcal{T}^{v_i}$ (for $i > j$) by accumulating such intervals in an efficient structure, and summing the contributions once the accumulation process has terminated.

**The Interval Structure** We build a full binary tree $BT(v)$ with leaf nodes associated with the elements of $\mathcal{T}^v$, as shown in Figure 1. The leaves are ordered from left to right, with the leftmost leaf corresponding to $s^v_1$, and the rightmost leaf to $s^v_{n_v}$.

Each leaf $s^v_i$ will be assigned a weight $\omega'(i)$, which will accumulate a portion of the weight of the best placement for which $s^v_i$ determines the depth. The node $s^v_i$ can be either *active* or *inactive* within the structure: in the former case,

$\omega'(i)$ is positive; in the latter case $\omega'(i) = -\infty$. Initially, all leaves are set to be inactive. Whenever a leaf changes status from inactive to active, it receives a timestamp.

At the internal nodes of $BT(v)$, information will be stored so as to allow the following operations to be performed efficiently:

1. Determine whether $s_i^v$ is active.
2. Determine the closest active leaf nodes to the left and to the right of $s_i^v$.
3. Change the $s_i^v$ from inactive to active or vice versa. (This is done by changing its weight $\omega'(i)$.)

We augment the structure so that it can also be used as a segment tree (see Preparata and Shamos [12] for details). Subtree $\mathcal{T}^{v_k}$ will contribute intervals $I^{v_k}(i)$ for insertion into the segment tree, for all $i$ from 1 to $n_u$. Each interval will store the weight of $P^{v_k}[i]$, and will receive a timestamp when inserted into the tree. At any given internal node of the segment tree, intervals will be maintained in increasing order of their timestamps.

The tree will be maintained in such a way that the weight of the best placement for a depth element $s_i^v$ is the weight of $s_i^v$ plus the sum of the weights of intervals containing its rank $r_i^v$ having timestamp greater than that of $s_i^v$.

**Description of the Algorithm** The modified Combine algorithm, MCombine, processes the subtrees from $\mathcal{T}^{v_1}$ to $\mathcal{T}^{v_m}$, in order. For each subtree $\mathcal{T}^{v_k}$, each placement of $P^{v_k}$ is considered.

Algorithm MCombine handles placements $P^{v_k}[i]$ in two phases. In the first phase, its depth element is made active, and its weight is initialized to that of the best placement involving elements of $\mathcal{T}^{v_1} \cup \ldots \cup \mathcal{T}^{v_k}$. The tree will be assumed to have been compacted by the beginning of this phase, and as such the placement sought is simply $P^{v_k}[i] \cup P^{v_{k-1}}[j]$, where $\mathsf{depelt}(P^{v_{k-1}}[j])$ is the shallowest depth element such that $P^{v_k}[i]$ and $P^{v_{k-1}}[j]$ can be combined. The best placement weight when $v_k$ itself is the depth element is also determined in this manner.

After the first phase has terminated, the second phase updates the best placements with depth elements in $\mathcal{T}^{v_1} \cup \ldots \cup \mathcal{T}^{v_{k-1}}$, to take into account a possible merge with $P^{v_k}[i]$. This is done by inserting the interval $I^{v_k}(i)$ into the segment tree, weighted with $\mathsf{wt}(v_k)[i]$ and timestamped. The weight of placements with depth element in $\mathcal{T}^{v_1} \cup \ldots \cup \mathcal{T}^{v_{k-1}}$ corresponding to leaf $l$ in $BT(v)$ can thus be read by adding to its weight $\omega'(l)$ the weight $\omega'(I)$ of each interval $I$ containing $l$ and having timestamp greater than that of $l$.

In both phases, we compact $BT(v)$. This is done by determining the true weights of the active leaf nodes of $BT(v)$ immediately to the left and right of the node associated with each modified element. In the case of modifications which occurred in the first phase, let $x$ and $y$ be the active elements immediately to the left (lower rank) and the right (higher rank) of the leaf node $r_i^{v_k}$ associated with $P^{v_k}[i]$. It follows from Lemma 1 that, if the weight of $y$ is greater than $\omega'(r_i^{v_k})$, the corresponding leaf $s_i^{v_k}$ can be eliminated from $BT(v)$ by changing its status

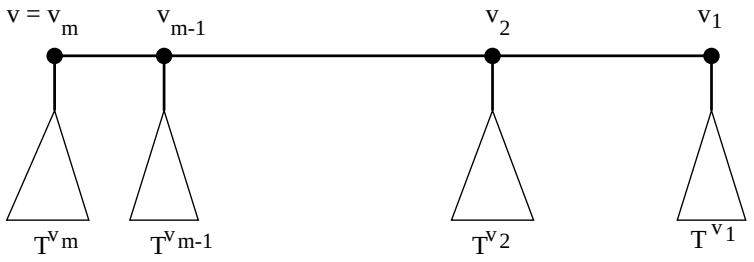



**Fig. 1.** Tree structure

to inactive. If the weight of $x$ is less than $\omega'(r_i^{v_k})$, we can eliminate $x$ instead. In this case, we repeat the process until no more elements can be eliminated from $BT(v)$.

The case of modifications occurring in the second phase is very similar to that of the first phase. However, note that we need only initiate this compaction process from the two endpoints of interval $I^{v_k}(i)$.

The modified combine algorithm can formally be described as follows:

**Algorithm** MCombine

1. Initialization
   a) Initialize the lists $P^v$ and $\mathsf{dep}(v)$.
   b) $t \leftarrow 0$.
   c) for each $i = 1, 2, \ldots, n_1$ do
      i. Let $\rho \leftarrow \mathsf{rank}(P^{v_1}[i])$.
      ii. Insert $P^{v_1}[i]$ in $BT(v)$ at the leaf node location $\rho$.
      iii. Set $\omega'(\rho) \leftarrow \mathsf{wt}(v_1)[i]$ and $\mathsf{stamp}(\rho) \leftarrow t$; $t \leftarrow t + 1$.
2. For $k \leftarrow 2, \ldots, m$ do {first phase}:
   a) For $i \leftarrow 1, 2, \ldots, n_{v_k}$, determine the element $x_i$ in $BT(v)$ with the minimum depth in $\mathcal{T}^{v_1} \cup \ldots \cup \mathcal{T}^{v_{k-1}}$ which can be combined with $P^{v_k}[i]$ in such a way that the result has depth element $\mathsf{depelt}(P^{v_k}[i])$. Let $y_i$ be this weight.
   b) For $i \leftarrow 1, 2, \ldots, n_{v_k}$,
      i. Let $\rho \leftarrow \mathsf{rank}(P^{v_k}[i])$.
      ii. Set $\omega'(\rho) \leftarrow \omega'(x_i) + y_i$ and $\mathsf{stamp}(\rho) \leftarrow t$; $t \leftarrow t + 1$.
      iii. Compact $BT(v)$ at $\rho$.
   c) As in the two previous steps, for depth element $v_k$ itself.
3. For $k \leftarrow 2, \ldots, m$ do {second phase}:
   a) For $i \leftarrow 1, 2, \ldots, n_{v_k}$, determine the interval $I = [r^-, r^+]$ of the active elements in $BT(v)$ satisfying the three conditions of Section 3.2.
   b) Set $\omega'(I) \leftarrow \mathsf{wt}(v_k)[i]$ and $\mathsf{stamp}(I) \leftarrow t$; $t \leftarrow t + 1$.
   c) Insert $I$ into $BT(v)$, and compact $BT(v)$ at $r^-$ and $r^+$.

After iteration $k$, taking into consideration only the elements of in $\mathcal{T}^{v_1} \cup \ldots \cup \mathcal{T}^{v_k}$, the weight of an active point (at leaf node location $l$ of $BT(v)$) can be determined by adding to $\omega'(l)$ the sum of $\omega'(I)$ over all intervals containing $I$ with $\mathsf{stamp}(I) > \mathsf{stamp}(l)$. Once the algorithm has terminated, the true weights of the active points can be determined in this way.

**Time and Space Complexity** We know that any given interval can appear in at most $O(\log n)$ internal nodes of a segment tree, and that the set of intervals appearing at a given internal node are maintained in increasing order of their time stamps. It is easy to show that

1. The structure requires $O(n \log n)$ space.
2. Each interval can be inserted into $BT(v)$ in $O(\log^2 n)$ time.
3. Each compaction operation requires $O((c+1)\log^2 n)$ time, where $c$ is the number of leaves eliminated (rendered inactive) as a result of the operation.
4. Given any leaf $l$ and a timestamp $t$, it is possible to determine in $O(\log^2 n)$ time the total weight of the intervals containing $l$ with time stamps greater than $t$.

Noting that a leaf can be eliminated by compaction at most once, the work associated with each insertion or deletion requires $O(\log^2 n)$ amortized time. The total time required for an execution of MCombine on a chain of subtrees is therefore $O(n \log^2 n)$.

### 3.3 Weighted Min-Gap Algorithm

The WMinGap algorithm for tree $\mathcal{T}$ with root $r$ can now be formally described, as follows:

**Algorithm** WMinGap

1. a) Determine the path from $r$ to the centroid $v_m$ of $\mathcal{T}$.
   {The path to the centroid decomposes $\mathcal{T}$ into a set of subtrees $\mathcal{T}^{v_i}$, $i = 1, 2, \ldots, m$ such that the size of each subtree is no more than two-thirds the size of the original tree $\mathcal{T}$.}
2. Solve the WMinGap problem for each of the subtrees $\mathcal{T}^{v_i}$, $i = 1, \ldots, k$.
   {The lists $\mathsf{dep}(u_i)$, $\mathsf{wt}(u_i)$, $P^{u_i}$ and the binary tree $BT(u_i)$ are known.}
3. Apply MCombine on the solutions of Step 2 to obtain $BT(v)$, from which the lists $\mathsf{dep}(v)$, $\mathsf{wt}(v)$, and $P^v$ can be calculated.

We can compute the centroid of $\mathcal{T}$ in $O(n)$ time, from which the subtrees $\mathcal{T}^{v_i}$, $i = 1, 2, \ldots, k$ can be determined in $O(n)$ time. We have seen that Step 3 takes $O(n \log^2 n)$ time.

It is not difficult to retrieve the final solution from $BT(v)$. The optimal placement is referenced by the first entries of the lists stored at $r$. This follows from the fact that all non-redundant placements for $\mathcal{T}$ are referenced by the lists. Let $p$ be the node of $\mathcal{T}$ corresponding to the first active leaf of $BT(v)$. Node $p$ must be included in the optimal placement. To obtain the other vertices of the optimal placement, it suffices to recursively traverse the subtrees of $p$ in $\mathcal{T}$. These vertices are identified by the intervals encountered in the path in $BT(r)$ from the root $r$ to the leaf node containing $p$. We can complete this process in $O(n \log^2 n)$ time.

From the preceding discussion, the following theorem holds.

**Theorem 1.** *Let $\mathcal{T}$ be a tree of $n$ vertices, whose vertices and edges are assigned non-negative real-valued weights. Given any positive real number $b$, Algorithm* WMinGap *computes in $O(n \log^3 n)$ worst-case time and $O(n \log n)$ space a structure from which in $O(n \log^2 n)$ additional time a placement $P$ can be read which is optimal for $\mathcal{T}$ with respect to $b$.*

## 4 Conclusion

In this paper, we have presented for acyclic networks an $O(n \log^3 n)$ time and $O(n \log n)$ space solution to the WMinGap facility location problem. The strategy of eliminating redundant placements which we described is similar in many respects to certain other problems in which a dominating subsequence of potentially-optimal solutions is maintained (for example, the applications involving optimal layouts of convex shapes studied in [2] and in [15]). For these problems the best solutions known take quadratic time in the worst case. It would be interesting to see if the techniques presented here lead to subquadratic solutions for these problems as well.

## References

1. B. K. Bhattacharya and M. E. Houle. Generalized maximum independent sets for trees. Proceedings of Computing: the Australasian Theory Symposium (CATS97), Sydney, Feb. 1997.
2. P. Eades, T. Lin and X. Lin. Minimum size h-v drawings. *Advanced Visual Interfaces (Proc. AVI '92)*, *World Scientific Series in Computer Science*, 36:386–394, 1992.
3. E. Erkut. The discrete $p$-dispersion problem. *Research Paper No. 87-5, Dept. of Finance and Management Science*, University of Calgary, Apr. 1989.
4. E. Erkut and S. Neuman. Analytical models for locating undesirable facilities. *European Journal of Operations Research*, 40:275–291, 1989.
5. E. Erkut, T. Baptie and B. von Hohenbalken. The discrete $p$-maxian location problem. *Computers in OR*, 17(1):51–61, 1990.
6. U. Friege, G. Kortsarz, D. Peleg. The dense k-subgraph problem. Manuscript, June 1998.
7. G. N. Frederickson and D. B. Johnson. Finding th paths and $p$-centers by generating and searching good data structures. *Journal of Algorithms*, 4:61–80, 1983.
8. R. Hassin, S. Rubinstein and A. Tamir. Approximation algorithms for maximum facility dispersion. *Operations Research Letters*, 21:133–137, 1997.
9. M. M. Halldórsson, K. Iwano, N. Katoh and T. Tokuyama. Finding subsets maximizing minimum structures. In *Proc. 6th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, San Francisco, pp. 150–157, 1995.
10. G. Kortsarz and D. Peleg. On choosing a dense subgraph. In *Proc. 34th IEEE FOCS*, Palo Alto, USA, pp. 692–701, 1993.
11. M. J. Kuby. Programming models for facility dispersion: the $p$-dispersion and maximum dispersion problems. *Geographical Analysis*, 19(4):315–329, 1987.

12. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
13. S. S. Ravi, D. J. Rosenkrantz and G. K. Tayi. Heuristic and special case algorithms for dispersion problems. *Operations Research*, 42(2):299–310, 1994.
14. D. J. Rosenkrantz, G. K. Tayi and S. S. Ravi. Capacitated facility dispersion problems. Manuscript, 1997.
15. L. Stockmeyer. Optimal orientation of cells in slicing floorplan design. *Information and Control*, 57:91–101, 1983.
16. A. Tamir. Obnoxious facility location on graphs. *SIAM J. Disc. Math.*, 4:550–567, 1991.
17. A. Tamir. Comments on the paper 'Heuristic and special case algorithms for dispersion problems' by S. S. Ravi. D. J. Rosenkrantz and G. K. Tayi. *Operations Research*, 46:157–158, 1998.
18. D. J. White. The maximal dispersion problem and the 'first point outside the neighbourhood' heuristic. *Computers and Operations Research*, 18:43–50, 1991.

## Author Index


Agarwal, P.K. 403
Arvind, V. 123
Asano, T. 317

Bansal, N. 247
Bhattacharya, B.K. 403, 435
Biedl, T.C. 415
Bordim, J.L. 215
Bose, P. 113, 269

Cai, M.-C. 279
Chan, W.-T. 393
Chang, J.-M. 163
Chin, F.Y.L. 393
Chwa, K.-Y. 195, 383
Cicerone, S. 205
Cui, J. 215

Damaschke, P. 27
Damian-Iordache, M. 56, 70
Das, A. 295
Demaine, E.D. 415
Deng, X. 153
Di Stefano, G. 205

Eidenbenz, S. 184

Flocchini, P. 93

Gaur, D.R. 49
Georgakopoulos, G.F. 4

Handke, D. 205
Hayashi, T. 215
Ho, C.-W. 163
Horiyama, T. 83
Houle, M.E. 435
Hsu, F.R. 173

Ibaraki, T. 83, 373, 425
Isobe, S. 347
Iwama, K. 133

Jordán, T. 425
Janssen, J. 327

Kaklamanis, C. 269
Katoh, N. 317
Kawashima, K. 317
Kirousis, L.M. 269
Ko, M.-T. 163
Kranakis, E. 269
Krishnamurti, R. 49
Krizanc, D. 269
Kusakari, Y. 337

Lazard, S. 415
Lee, J.-H. 195, 383
Lin, Y.-L. 173

Madhavan, C.E.V. 295
Maheshwari, A. 307
Makino, K. 259
Masubuchi, D. 337
McClurkin, D.J. 4
Mehlhorn, K. 1
Miyazaki, S. 133
Morin, P. 113

Nagamochi, H. 373, 425
Nakano, K. 215
Nakao, Y. 425
Narayanan, L. 327
Nishizeki, T. 337, 347

Olariu, S. 215

Park, C.-D. 195
Peleg, D. 269
Pemmaraju, S.V. 56, 70
Poon, C.K. 143, 153
Prencipe, G. 93

Raman, V. 18, 247
Rao, S.S. 18
Robbins, S.M. 415

Sanders, P. 37
Santoro, N. 93
Sen, S. 403
Shibuya, T. 225

Soss, M.A. 415
Subrahmanyam, K.V. 123
Sugihara, K. 357

Takaoka, T. 237
Tardos, E. 183
Ting, H.-F. 393
To, K.-K. 103
Tsai, Y.-T. 173

Vempala, S. 367
Vinodchandran, N.V. 123
Vöcking, B. 367

Widmayer, P. 93
Wong, W.-H. 103

Yang, X. 279

Zeh, N. 307
Zhang, J. 279
Zhang, Y. 153
Zhao, L. 373
Zhou, X. 347
Zhu, B. 143