Lecture Notes in Computer Science 1461
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Gianfranco Bilardi Giuseppe F. Italiano
Andrea Pietracaprina Geppino Pucci (Eds.)

Algorithms – ESA '98

6th Annual European Symposium
Venice, Italy, August 24-26, 1998
Proceedings

Springer

Volume Editors

Gianfranco Bilardi
Università di Padova, Dipartimento di Elettronica e Informatica
via Gradenigo 6/A, I-35131 Padova, Italy
and
The University of Illinois at Chicago
Department of Electrical Engineering and Computer Science
851 South Morgan Street - 1009 SEO, Chicago, IL 60607-7053, USA
E-mail: bilardi@artemide.dei.unipd.it

Giuseppe F. Italiano
Università "Cá Foscari" di Venezia
Dipartimento di Matematica Applicata e Informatica
via Torino 155, I-30173 Venezia Mestre, Italy
E-mail: italiano@dsi.unive.it

Andrea Pietracaprina
Università di Padova, Dipartimento di Matematica Pura e Applicata
via Belzoni 7, I-35131 Padova, Italy
E-mail: andrea@artemide.dei.unipd.it

Geppino Pucci
Università di Padova, Dipartimento di Elettronica e Informatica
via Gradenigo 6/A, I-35131 Padova, Italy
E-mail: geppo@artemide.dei.unipd.it

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algorithms : 6th annual European symposium ; proceedings / ESA '98, Venice, Italy, August 24 - 26, 1998. Gianfranco Bilardi . . . (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1461)
ISBN 3-540-64848-8

CR Subject Classification (1991): F.2, G.1-2, I.3.5, C.2, E.1

ISSN 0302-9743
ISBN 3-540-64848-8 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 1998
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10638295 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

This volume contains 40 contributed papers accepted for presentation at the 6th Annual European Symposium on Algorithms (ESA'98), Venice, Italy, August 24–26, 1998, and the invited lectures by Eli Upfal (Brown) and Je rey Vitter (Duke). ESA is an annual meeting, with previous meetings held in Bad Honnef, Utrecht, Corfu, Barcelona, and Graz, and covers research in the *use, design, and analysis of e cient algorithms and data structures* as it is carried out in computer science, discrete applied mathematics and mathematical programming. Proceedings of ESA have been traditionally part of the LNCS series of Springer-Verlag. The symposium is promoted and coordinated by a steering committee, whose current composition is:

Gianfranco Bilardi, *Padua & UIC*
Josep Diaz, *Barcelona*
Giuseppe F. Italiano, *Venice*
Kurt Mehlhorn, *Saarbrücken*
Ralf H. Möhring, *Berlin*
Gerhard Woeginger, *Graz*

The Program Committee of ESA'98 consisted of:

Gianfranco Bilardi, Co-chair, *Padua & UIC*
Jean Daniel Boissonnat, *Sophia Antipolis*
Kieran T. Herley, *Cork*
Dorit Hochbaum, *Berkeley*
Giuseppe F. Italiano, Co-Chair, *Venice*
Luděk Kučera, *Prague*
Richard J. Lipton, *Princeton*
Yishay Mansour, *Tel-Aviv*
Kurt Mehlhorn, *Saarbrücken*
Friedhelm Meyer auf der Heide, *Paderborn*
Grammati Pantziou, *Patras*
Mike Paterson, *Warwick*
José Rolim, *Geneva*
Erik M. Schmidt, *Aarhus*
Maria Serna, *Barcelona*
Paul Vitanyi, *Amsterdam*
Gerhard Woeginger, *Graz*

There were 131 extended abstracts submitted in response to a call for papers. Papers were solicited describing original results in *all areas of algorithmic research*, including but not limited to: approximation algorithms; combinatorial optimization; computational biology; computational geometry; databases and information retrieval; graph and network algorithms; machine learning; number theory and computer algebra; on-line algorithms; pattern matching and data compression; symbolic computation. Submissions that reported on experimental and applied research were especially encouraged.

The contributed papers were selected by the program committee, which met in Padova, Italy, on April 24 and 25, 1998. The selection of contributed papers was based on originality, quality, and relevance to the symposium. Considerable e ort was devoted to the evaluation of the submissions. Extensive feedback was provided to authors as a result, which we hope has proven helpful. However, submissions were not refereed in the thorough and detailed way that is customary

for journal papers. Many submissions represent reports of continuing research. It is expected that most of the papers in these proceedings will eventually appear in finished form in scientific journals. We believe that the final program of ESA'98, organized into ten sessions, represents an interesting cross section of research e orts in the area of algorithms.

We wish to thank the program committee, and all of those who submitted abstracts for consideration. We thank all of our many colleagues who helped the program committee in evaluating the submissions. Their names appear in a separate list, and we apologize to anyone whose name is omitted. ESA'98 could not have happened without the dedicated work of the organizing committee, consisting of:

Francesco Bombi, Chair, *Padua*
Dora Giammarresi, *Venice*
Salvatore Orlando, *Venice*
Marcello Pelillo, *Venice*
Andrea Pietracaprina, *Padua*
Fabio Pittarello, *Venice*
Geppino Pucci, *Padua*
Alessandro Roncato, *Venice*

The agency Key Congress & Communications acted as a secretariat for the symposium. We gratefully acknowledge the generous support of the ESA'98 sponsors:

Consiglio Nazionale delle Ricerche
InfoCamere
IRT
Telecom Italia
UNESCO
Università di Padova
Università Ca' Foscari di Venezia

Finally, Andrea Pietracaprina and Geppino Pucci have been invaluable co-editors of these proceedings.

Padua, June 1998 Gianfranco Bilardi

Venice, June 1998 Giuseppe F. Italiano

List of Reviewers

Stephen Alstrup
Carme Alvarez
Charles Anderson
Gyarfas Andras
Alexander Andreev
Lars Arge
Mike Atkinson
Yonatan Aumann
Yossi Azar
Amotz Bar-Noy
B. Beauquier
Vincent Berry
Hans Bodlaender
Karl F. Bohringer
Maria Luisa Bonet
Allan Borodin
J. Boyar
Peter Bro Miltersen
Gerth Brodal
Herve Bronnimann
Tiziana Calamoneri
Ioannis Caragiannis
Rafael Casas
Frederic Cazals
Otfried Cheong
Siu-Wing Cheng
Andrea Clementi
Pierluigi Crescenzi
Felipe Cucker
Artur Czumaj
James Davenport
Sergio De Agostino
Mark De Berg
Pavan Kumar Desikan
Tassos Dimitriou
Josep Diaz
Giuseppe Di Battista
Miriam Di Ianni
Yefim Dinitz
Amalia Duch
Ran El-Yaniv
David Eppstein
Funda Ergun
Martin Farach-Colton
Panagiota Fatourou
Uri Feige
Sandra Feisel
Afonso Ferreira
Michele Flammini
Dimitris Fotakis
Gudmund Frandsen
Joaquim Gabarro
Ra aele Giancarlo
Leslie Ann Goldberg
Paul Goldberg
Olivier Goldschmidt
Mordecai Golin
Jacek Gondzio
Roberto Grossi
Dan Halperin
Kostas Hatzis
Han Hoogeveen
Martin Hühne
Ferran Hurtado
Rob Irving
Tao Jiang
Ben Juurlink
Howard Karlo
Claire Kenyon
Hal Kierstead
Bettina Klinz
Stavros Koliopoulos
Spyros Kontogiannis
Giuseppe Lancia
Bruno Lang
Mike Langston
Kim S. Larsen
Jan van Leeuwen
Stefano Leonardi
Mauro Leoncini
Giuseppe Liotta
Martin Loebel
Tamas Lukovszki
Vasilis Mamalis
Joseph Manning
Conrado Martinez
Annalisa Massini
J. Matousek
Brian Mayoh
S. Muthukrishnan
Se Naor
Apostol Natsev
Stefan Nilsson
Michael Noecker
John Noga
Eli Olinick
Rasmus Pagh
Alessandro Panconesi
Jakob Pagter
Panos Pardalos
Boaz Patt-Shamir
Aleksandar Pekec
Christian N. S. Pedersen
George Pentaris
S. Perennes
Jordi Petit
Leonidas Pitsoulis
Guido Proietti
Teresa Przytycka
Yuval Rabani
Tomasz Radzik
Rajeev Raman
R. Ravi
Franz Rendl
Herman te Riele
Ingo Rieping
Gianluca Rossi
Guenter Rote
Vera Sacristan
Cenk Sahinalp
Christian Scheideler
Klaus Schröder
Petra Schuurman
Eli Shamir
Ron Shamir
Steve Seiden
Nir Shavit
Jop F. Sibeyn
Riccardo Silvestri
Alistair Sinclair
Michiel Smid
Danuta Sosnowska

Aravind Srinivasan
Yiannis Stamatiou
Andrea Sterbini
Michel Syska
Luca G. Tallini
Vasilis Tampakas
Luca Trevisan
John Tromp
Zsolt Tuza
Eli Upfal
Berthold Vöcking
Nicolai Vorobjov
Rolf Wanka
Todd Wareham
Matthias Westermann
Pawel Winter
Fatos Xhafa
Kathy Yelick
Christos Zaroliagis
Martin Ziegler
Paul Zimmerman
Uri Zwick
Thierry Zwissig

Table of Contents

Invited Lectures

Data Structures

Strings and Biology

Numerical Algorithms

Geometry

Randomized and On-Line Algorithms

Parallel and Distributed Algorithms I

Graph Algorithms

Parallel and Distributed Algorithms II

Optimization

External Memory Algorithms[?]

Jeffrey Scott Vitter[??]

Center for Geometric Computing, Department of Computer Science
Duke University, Durham, NC 27708–0129, USA
jsv@cs.duke.edu
http://www.cs.duke.edu/~jsv/

Abstract. Data sets in large applications are often too massive to fit completely inside the computer's internal memory. The resulting input/output communication (or I/O) between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck. In this tutorial, we survey the state of the art in the design and analysis of *external memory algorithms* (also known as EM algorithms or out-of-core algorithms or I/O algorithms). External memory algorithms are often designed using the parallel disk model (PDM). The three machine-independent measures of an algorithm's performance in PDM are the number of I/O operations performed, the CPU time, and the amount of disk space used. PDM allows for multiple disks (or disk arrays) and parallel CPUs, and it can be generalized to handle cache hierarchies, hierarchical memory, and tertiary storage.
We discuss a variety of problems and show how to solve them efficiently in external memory. Programming tools and environments are available for simplifying the programming task. Experiments on some newly developed algorithms for spatial databases incorporating these paradigms, implemented using TPIE (Transparent Parallel I/O programming Environment), show significant speedups over popular methods.

1 Introduction

The *Input/Output* communication (or simply *I/O*) between the fast internal memory and the slow external memory (such as disk) can be a bottleneck in applications that process massive amounts of data [51]. One approach to optimizing performance is to bypass the virtual memory system and to develop algorithms that explicitly manage data placement and movement, which we refer to as *external memory algorithms*, or more simply *EM algorithms*. The terms *out-of-core algorithms* and *I/O algorithms* are also sometimes used.

[?] Extended abstract. An earlier version appeared as an invited tutorial in *Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems (PODS '98)*, Seattle, WA, June 1998. A longer version appears in [106]. An updated version is available electronically on the author's web pageat http://www.cys.duke.edu/~jsv/. Feedback is welcome.

[??] Supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation research grant CCR–9522047.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 1{25, 1998.
c Springer-Verlag Berlin Heidelberg 1998

In order to be effective, EM algorithm designers need to have a simple but reasonably accurate model of the memory system's characteristics. Magnetic disks consist of one or more rotating platters and one read/write head per platter surface. The data are stored in concentric circles on the platters called *tracks*. To read or write a data item at a certain address on disk, the read/write head must mechanically *seek* to the correct track and then wait for the desired address to pass by. The seek time to move from one random track to another is often on the order of 5–10 milliseconds, and the average rotational latency, which is the time for half a revolution, has the same order of magnitude. In order to amortize this delay, it pays to transfer a large collection of contiguous data items, called a *block*.

Even if an application can structure its pattern of memory use to take full advantage of disk block transfer, there is still a substantial *access gap* between internal memory and external memory performance. In fact the access gap is growing, since the speed of memory chips is increasing more quickly than disk bandwidth and disk latency. The use of parallel processors further widens the gap. Storage systems such as RAID are being developed that use multiple disks to get more bandwidth [29, 59].

We can capture the main properties of magnetic disks and multiple disk systems by the commonly-used *parallel disk model* (PDM) introduced by Vitter and Shriver [107]:

N = problem size (in units of data items);
M = internal memory size (in units of data items);
B = block transfer size (in units of data items);
D = # independent disk drives;
P = # CPUs;

where $M < N$, and 1 DB M=2. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items. If P D, each of the P processors can drive about D=P disks; if $D < P$, each disk is shared by about P=D processors. The internal memory size is M=P per processor. The P processors are connected by an interconnection network.

It is often helpful to refer to some of the above PDM parameters in units of blocks rather than in units of data items. We define the lower-case notation

$$n = \frac{N}{B} \quad ; \qquad m = \frac{M}{B} \tag{1}$$

to be the problem size and internal memory size, respectively, in units of disk blocks. We assume that the input data are initially "striped" across the D disks, in units of blocks, as illustrated in Figure 1, and we require the output data to be similarly striped. Striped format allows a file of N data items to be read or written in $O(n$=$D) = O(N$=$DB)$ I/Os, which is optimal. We refer to $O(n$=$D)$ I/Os as a "linear number of I/Os" in the PDM model.

The primary measures of performance in PDM are the number of I/O operations performed, the internal (parallel) computation time, and the amount of

	D_0	D_1	D_2	D_3	D_4
stripe 0	0 1	2 3	4 5	6 7	8 9
stripe 1	10 11	12 13	14 15	16 17	18 19
stripe 2	20 21	22 23	24 25	26 27	28 29
stripe 3	30 31	32 33	34 35	36 37	38 39

Fig. 1. Initial data layout on the disks, $D = 5$, $B = 2$. The input data items are initially striped block-by-block across the disks. For example, data items 16 and 17 are stored in the second block (i.e., in stripe 1) of disk D_3.

disk space used. For reasons of brevity we will focus in this paper on the number of I/Os performed. Most of the algorithms discussed in the paper use optimal CPU time, at least for the single-processor case, and $O(n)$ blocks of disk space, which is optimal.

The track size is a parameter of the disk hardware and cannot be altered;[1] for most disks it is in the range 50–100 KB. Generally the block transfer size B in PDM is chosen to be a significant fraction of the track size so as to amortize seek time.[2]

PDM is a good generic programming model that facilitates the design of I/O-efficient algorithms, especially when used in conjunction with the programming tools discussed in Section 10. More complex and precise disk models have been developed, such as the ones by Ruemmler and Wilkes [86], Shriver et al. [91], and Barve et al. [20], which distinguish between sequential reads and random reads and consider the effects on throughput of features such as disk buffer caches and shared buses. In practice, the effects of more complex models can often be realized or approximated by PDM with an appropriate choice of parameters. The bottom line is that programs that perform well in terms of PDM will generally perform well when implemented on real systems.

The study of the complexity of algorithms using external memory devices began more than 40 years ago with Demuth's Ph.D. thesis on sorting [41, 69]. In

[1] The actual track size for any given disk varies with the radius of the track. Sets of adjacent tracks are usually formatted to have the same track size, so there are typically a small number of different track sizes for a given disk.

[2] The minimum block transfer size imposed by hardware is usually 512 bytes, but operating systems generally use a much larger block size, such as 8 KB. It is possible to use an even larger block size and further reduce the relative significance of seek and rotational latency, but the wall clock time per I/O will increase accordingly. For example, if we set B to be five times larger than the track size, the time per I/O will correspond to five revolutions of the disk plus the relatively insignificant seek time. (If the disk is smart enough, rotational latency can be avoided, since the block spans entire tracks.) Once the block transfer size becomes larger than the track size, the wall clock time per I/O grows linearly with the block size. It is thus often helpful to think of the block transfer size in PDM as a fixed hardware parameter roughly proportional to the track size. The particular block size that optimizes performance in an actual implementation will typically vary slightly from application to application.

the early 1970s, Knuth [69] did an extensive study of sorting using magnetic tapes and (to a lesser extent) magnetic disks. At about the same time, Floyd [48,69] considered a disk model akin to PDM for $D = 1$, $P = 1$, $B = M/2 = \Omega(N^c)$, for constant $c > 0$, and developed optimal upper and lower I/O bounds for sorting and matrix transposition. Savage and Vitter [90] developed an I/O version of pebbling for straightline computations, in which the data items are transferred in units of blocks. Aggarwal and Vitter [7] generalized Floyd's I/O model to allow simultaneous block transfers, but the model was unrealistic for a single disk. They developed matching upper and lower I/O bounds for all parameter values for a host of problems. Since the PDM model can be thought of as a more restrictive (and more realistic) version of their model, their lower bounds apply as well to PDM. Modified versions of PDM that integrate various aspects of parallel computation are developed in [75, 40]. Surveys of various aspects of I/O models and algorithms appear in [92, 11].

The same type of bottleneck that occurs between internal memory and external disk storage can also occur at other levels of the memory hierarchy, such as between data cache and level 2 cache, or between level 2 cache and DRAM, or between disk storage and tertiary devices. The PDM model can be generalized to multilevel memory hierarchies, but for reasons of brevity and emphasis, we do not discuss such models here. We refer the reader to [108] and its references.

In this paper we survey several useful paradigms for solving problems efficiently in external memory. In Sections 2–9, we discuss distribution and merging (for sorting-related problems), distribution sweep (for spatial join and finding all nearest neighbors), batched filtering and external fractional cascading (for GIS map overlay and red-blue line segment intersection), randomized incremental constructions (for intersection and other geometry problems), B-trees (for dictionaries and one-dimensional range queries), buffer trees (for batched dynamic problems and sweep line applications), interval trees (for stabbing queries), and R-trees (for spatial applications such as multidimensional range queries and contour line processing in GIS).

In Section 10, we describe experiments on two problems arising in spatial databases, for which speedups are possible over currently used techniques. We use TPIE (Transparent Parallel I/O programming Environment) for our implementations. In Section 11 we describe EM algorithms that can adapt optimally to dynamically changing memory allocations.

2 External Sorting and Related Problems

The problem of sorting is a central problem in the field of external memory algorithms, partly because sorting and sorting-like operations account for a significant percentage of computer use [69], and also because sorting is an important paradigm in the design of efficient EM algorithms. With some technical qualifications, many problems that can be solved easily in linear time in internal memory, such as list ranking, permuting, expression tree evaluation, and connected components in a sparse graph, require the same number of I/Os in PDM as sorting.

Theorem 1 ([7, 82]). *The average-case and worst-case number of I/Os required for sorting N data items using D disks is*

$$\frac{n}{D}\log_m n \quad = \quad \frac{n}{D}\frac{\log n}{\log m} \quad : \tag{2}$$

It is conceptually much simpler to program for the single-disk case ($D = 1$) than for the multi-disk case. *Disk striping* is a paradigm that can ease the programming task with multiple disks. I/Os are permitted only on entire stripes, one at a time. For example, in the data layout in Figure 1, data items 20–29 can be accessed in a single I/O step because their blocks are grouped into the same stripe. The net effect of striping is that the D disks behave as a single logical disk, but with a larger logical block size DB.

Let us consider what happens if we use the technique of disk striping in conjunction with an optimal sorting algorithm for one disk. The optimal number of I/Os using one disk is

$$n\frac{\log n}{\log m} \quad = \quad \frac{N}{B}\frac{\log(N/B)}{\log(M/B)} \quad : \tag{3}$$

The effect of disk striping with D disks is to replace B by DB in (3), which yields the I/O bound

$$\frac{N}{DB}\frac{\log(N/DB)}{\log(M/DB)} \quad = \quad \frac{n}{D}\frac{\log(n/D)}{\log(m/D)} \quad : \tag{4}$$

The striping I/O bound (4) can be larger than the optimal bound (2) by a multiplicative factor of $(\log m)/\log(m/D)$, which is significant when D is on the order of m, causing the $\log(m/D)$ term in the denominator to be very small. In order to attain the optimal sorting bound (2) theoretically, we must be able to control the disks independently, so that each disk can access a different stripe in the same I/O. Sorting via disk striping is often more efficient in practice than more complicated techniques that utilize independent disks, since the $(\log m)/\log(m/D)$ factor may be dwarfed by the additional overhead of using the disks independently [104].

In the following two subsections we discuss and analyze new external sorting algorithms based upon the distribution and merge paradigms. The SRM method, which uses a randomized merge technique, outperforms disk striping in practice for reasonable values of D (see Section 2.2). In the last two subsections we consider the related problems of permuting and Fast Fourier Transform. All the methods we discuss, with the exception of Greed Sort in Section 2.2, use the disks independently for parallel read operations, but parallel writes are done in a striped manner, which facilitates the writing of parity error correction information [29, 59]. The lower bounds are discussed in Section 3.

2.1 Sorting by Distribution: Simultaneous Online Load Balancings

Distribution sort is a recursive process in which the data items to be sorted are partitioned by a set of $S - 1$ partitioning elements into S buckets. All the items

in one bucket precede all the items in the next bucket. The individual buckets are sorted recursively and then concatenated together to form a single totally sorted list.

The $S - 1$ partitioning elements are chosen in such a way that the buckets are of roughly equal size. Hence, the bucket sizes decrease by a (S) factor from one level of recursion to the next, and there are $O(\log_S n)$ levels of recursion. During each level of recursion, the data are streamed through internal memory, and the S buckets are written to the disks in an online manner. A double buffer of size $2B$ is allocated to each of the S buckets. When one half of the double buffer fills, its block is written to disk in the next I/O, and the other half is used to store the incoming items. Therefore, the maximum number of buckets (and partitioning elements) is $S = (M/B) = (m)$, and the resulting number of levels of recursion is $(\log_m n)$.

It seems difficult to find $S = (m)$ partitioning elements using (n/D) I/Os and guarantee that the bucket sizes are within a constant factor of one another. But efficient deterministic methods exist for choosing $S = \sqrt{m}$ partitioning elements [107, 81], which has the effect of doubling the number of levels of recursion. Probabilistic methods based upon random sampling can be found in [45].

In order to meet the sorting bound (2), the formation of the buckets at each level of recursion must be done in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each read step and each write step during the bucket formation must involve on the average (D) blocks. The file of items being partitioned was itself one of the buckets formed in the previous level of recursion. In order to read that file efficiently, its blocks must be spread uniformly among the disks, so that no one disk is a bottleneck. The challenge in distribution sort is to write the blocks of the buckets to the disks in an online manner and achieve a global load balance by the end of the partitioning, so that the bucket can be read efficiently during the next level of the recursion.

Partial striping is a useful technique that can reduce the amount of information that must be stored in internal memory in order to manage the disks. The disks are grouped into clusters of size C and data are written in "logical blocks" of size CB, one per cluster. Choosing $C = \sqrt{D}$ won't change the optimal sorting time by more than a constant factor, but as pointed out earlier, full striping (in which $C = D$) can be nonoptimal.

In addition to partial striping, Vitter and Shriver [107] use two randomized online techniques during the partitioning so that with high probability each bucket is well balanced across the D disks. (Partial striping is used so that the pointers needed to keep track of the layout of the buckets on the disks can fit in internal memory.) The first technique is used when the size N of the file to partition is sufficiently large or when DB M. Each parallel write operation writes its D blocks in random order to a stripe, with all $D!$ orders equally likely. If N is not large enough, however, the technique breaks down and the distribution of each bucket tends to be uneven. For these smaller values of N, Vitter and Shriver use a different technique: In one pass, the file is read, one memoryload

at a time. Each memoryload is randomly permuted and written back to the disks in the new order. In a second pass, the file is accessed in a "diagonally striped" manner. They show that with very high probability each individual "diagonal stripe" contributes about the same number of items to each bucket, so an oblivious shuffling pattern can achieve the desired global balance.

DeWitt et al. [42] present a randomized distribution sort algorithm in a similar model to handle the case when sorting can be done in two passes. They use a sampling technique to find the partitioning elements and route the items in each bucket to a particular processor. The buckets are sorted individually in the second pass.

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [81]. During the partitioning process, the algorithm tracks how evenly each bucket has been distributed so far among the disks. For each $1 \quad b \quad S$ and $1 \quad d \quad D$, let num_b be the total number of items in bucket b processed so far during the partitioning and let $num_b(d)$ be the number of those items written to disk d; that is, $num_b = \sum_{1 \quad d \quad D} num_b(d)$. The algorithm is able to write at least half of any given memoryload to the disks and still maintain the invariant for each bucket b that the $\lfloor bD/2 \rfloor$ largest values of $num_b(1)$, $num_b(2)$, ..., $num_b(D)$ differ by at most 1, and hence each $num_b(d)$ is at most about twice the ideal value num_b/D.

2.2 Sorting by Merging

The merge paradigm is somewhat orthogonal to the distribution paradigm discussed above. A typical merge sort algorithm works as follows: In the "run" formation phase, the n blocks of data are streamed into memory, one memoryload at a time; each memoryload is sorted into a "run," which is then output to stripes on disk. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, the "replacement-selection" technique can be used to get runs of $2M$ data items, on the average, when $M \quad B$ [69].)

After the initial runs are formed, the merging phase begins. In each pass of the merging phase, groups of R runs are merged together. During each merge, one block from each run resides in internal memory. When the data items of a block expire, the next block for that run is input. Double buffering is used to keep the disks busy. Hence, at most $R = \quad (m)$ runs can be merged at a time; the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (2), each merging pass must be done in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each parallel read operation during the merging must on the average bring in the next $\quad (D)$ blocks needed for the merging. The challenge is to ensure that those blocks reside on different disks so that they can be read in a single I/O (or a small constant number of I/Os). The difficulty lies in the fact that the runs being merged were themselves formed during the previous merge pass. Their blocks were written to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

A perfect solution, in which the next D blocks needed for the merge are guaranteed to be on distinct disks, can be devised for the binary merging case $R = 2$ based on the Gilbreath principle [50, 69]: The first run is striped in ascending order by disk number, and the other run is striped in descending order. The next D blocks needed for the output can always be read in by a single I/O operation, and thus the amount of buffer space needed for binary merging can be kept to a minimum. Unfortunately there is no analog to the Gilbreath principle for $R > 2$, and as we have seen above it is necessary to use a large value of R in order to get an optimal sorting algorithm.

The Greed Sort method of Nodine and Vitter [82] was the first optimal deterministic EM algorithm for sorting with multiple disks. It handles the case $R > 2$ by relaxing the condition on the merging process. In each step, the following two blocks from each disk are brought into internal memory: the block b_1 with the smallest data item value and the block b_2 whose largest item value is smallest. It may be that $b_1 = b_2$, in which case only one block is read into memory, and it is added to the next output stripe. Otherwise, the two blocks b_1 and b_2 are merged in memory; the smaller B items are written to the output stripe, and the remaining items are prepended back to the run originally containing b_1. The resulting run that is produced is only an "approximately" merged run, but its saving grace is that no two inverted items are too far apart. A final application of Columnsort [73] in conjunction with partial striping restores total order.

An optimal deterministic merge sort, with somewhat higher constant factors than those of the distribution sort algorithms, was developed by Aggarwal and Plaxton [6], based upon the Sharesort hypercube sorting algorithm [39]. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious.

The most practical method for sorting is the simple randomized merge sort (SRM) algorithm of Barve et al. [21] (referred to as "randomized striping" by Knuth [69]). Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is used.) During the merging process, the next block needed from each disk is read into memory, and if there is not enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space. The expected performance of SRM is not optimal for all parameter values, but it significantly outperforms the use of disk striping for reasonable values of the parameters, as shown in Table 1. Barve et al. [21] derive an upper bound on the I/O performance; the precise analysis is an interesting open problem.

2.3 Permuting and Transposition

Permuting is the special case of sorting in which the key values of the N data items form a permutation of $\{1, 2, \ldots, N\}$.

Theorem 2 ([7]). *The average-case and worst-case number of I/Os required for permuting N data items using D disks is*

$$\min \left\{ \frac{N}{D}, \frac{n}{D} \log_m n \right\} \quad . \tag{5}$$

	$D = 5$	$D = 10$	$D = 50$
$k = 5$	0:56	0:47	0:37
$k = 10$	0:61	0:52	0:40
$k = 50$	0:71	0:63	0:51

Table 1. The ratio of the number of I/Os used by simple randomized merge sort (SRM) to the number of I/Os used by merge sort with disk striping, during a merge of kD runs. The figures were obtained by simulation; they back up the (more pessimistic) analytic upper bound in [21].

Matrix transposition is the special case of permuting in which the permutation can be represented as a transposition of a matrix from row-major order into column-major order.

Theorem 3 ([7]). *The number of I/Os required using D disks to transpose a p q matrix from row-major order to column-major order is* $(n/D) \log_m \min\{M, p, q, n\}$.

Matrix transposition is a special case of a more general class of permutations called *bit-permute/complement* (BPC) permutations, which in turn is a subset of the class of *bit-matrix-multiply/complement* (BMMC) permutations. BPC permutations include matrix transposition, bit-reversal permutations (which arise in the FFT), vector-reversal permutations, hypercube permutations, and matrix reblocking. Cormen et al. [37] characterize the optimal number of I/Os needed to perform any given BMMC permutation in terms of its matrix representation, and they give an optimal algorithm for implementing it.

2.4 Fast Fourier Transform

Computing the Fast Fourier Transform (FFT) in external memory consists of a series of I/Os that permit each computation implied by the FFT (or butterfly) directed graph to be done while its arguments are in internal memory. A permutation network computation consists of a fixed pattern of I/Os such that any of the $N!$ possible permutations can be realized; data items can only be reordered when they are in internal memory. A permutation network can be realized by a series of three FFTs [111].

Theorem 4. *The number of I/Os using D disks required for computing the N-input FFT digraph or an N-input permutation network is given by the same bound* (2) *as for sorting.*

Cormen and Nicol [36] give some practical implementations for one-dimensional FFTs based upon the optimal PDM algorithm of [107]. The algorithms for FFT are simpler than for sorting because the computation is non-adaptive in nature, and thus the communication pattern is oblivious.

3 Lower Bounds on I/O

In this section we prove the lower bounds from Theorems 1–4 in Section 2. The most trivial batched problem is that of *scanning* (or *streaming* or *touching*) a file of N data items, which can be done in a linear number $O(N/DB) = O(n/D)$ of I/Os. Permuting is a simple problem that can be done in linear time in the (internal memory) RAM model, but requires a nonlinear number of I/Os in PDM because of the locality constraints imposed by the block parameter B.

The following proof of the permutation lower bound (5) of Theorem 2 is due to Aggarwal and Vitter [7]. The idea of the proof is to measure, for each $t \geq 0$, the number of distinct orderings that are realizable by at least one sequence of t I/Os. The value of t for which the number of distinct orderings first exceeds $N!/2$ is a lower bound on the average number of I/Os (and hence the worst-case number of I/Os) needed for permuting.

We assume for the moment that there is only one disk, $D = 1$. Let us consider how the number of realizable orderings can change when we read a given disk block into internal memory. There are at most B data items in the block, and they can intersperse among the M items in internal memory in at most $\binom{M}{B}$ ways, so the number of realizable orderings increases by a factor of $\binom{M}{B}$. If the block has never before resided in internal memory, the number of realizable orderings increases by an extra $B!$ factor, since the items in the block can be permuted among themselves. (This extra contribution of $B!$ can only happen once for each of the N/B original blocks.) The effect of writing the disk block is considerably less than that of reading it. There are at most $n + t \leq N \log N$ ways to choose which disk block is involved in the I/O. Hence, the number of distinct orderings that can be realized by some sequence of t I/Os is at most

$$(B!)^{N/B} \cdot N(\log N) \cdot \binom{M}{B}^{t}. \tag{6}$$

Setting the expression in (6) to be at least $N!/2$, and simplifying by taking the logarithm, we get

$$N \log B + t \cdot \log N + B \log \frac{M}{B} = \Omega(N \log N). \tag{7}$$

We get the lower bound for the case $D = 1$ by solving for t. The general lower bound (5) follows by dividing by D.

Permuting is a special case of sorting, and hence, the permuting lower bound applies also to sorting. In the unlikely case that $B \log m = o(\log n)$, the permuting bound is only $\Omega(N/D)$, and we must resort to the comparison model to get the full lower bound (2) of Theorem 1 [7]. Arge et al. [14] show for the comparison model that any problem with an $\Omega(N \log N)$ lower bound in the RAM model requires $\Omega(n \log_m n)$ I/Os in PDM. However, in the typical case in which $B \log m = \Omega(\log n)$, the comparison model is not needed to prove the sorting lower bound; the difficulty of sorting in that case arises not from determining the order of the data but from permuting (or routing) the data.

The same proof used above for permuting also applies to permutation networks, in which the communication pattern is oblivious. Since the choice of disk block is fixed for each t, there is no $N \log N$ term as there is in (6) (and correspondingly there is no additive $\log N$ term as there is in (7)). Hence, when we solve for t, we get the lower bound (2) rather than (5). In contrast with sorting, the lower bound follows directly from the counting argument, without any dependence upon the comparison model for the case $B \log m = o(\log n)$. The same lower bound also applies to the FFT, since permutation networks can be built from a series of three FFTs. The lower bound for transposition involves a potential argument based on a togetherness relation [7].

Chiang et al. [30] and Arge [10] discuss lower bound reductions for several graph problems. Problems like list ranking and expression tree evaluation have the same nonlinear PDM lower bound as permuting. This situation is in contrast with the RAM model, in which the same problems can all be done in linear time.

The lower bounds mentioned above assume that the data items are in some sense "indivisible" in that they are not split up and reassembled in some magic way to get the desired output. It is conjectured that the sorting lower bound (2) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, Adler [2] showed that removing the indivisibility assumption can lead to faster algorithms. A similar result is shown by Arge and Miltersen [15] for the problem of determining if the N data item values are distinct.

4 Matrix and Grid Computations

Dense matrices are generally represented in memory in row-major or column-major order. Matrix transposition, which is the conversion of a matrix from one representation to the other, was discussed in Section 2.3. For certain operations such as matrix addition, both representations work well. However, for standard matrix multiplication and LU decomposition, neither representation is appropriate; a better representation is to block the matrix into square $\sqrt{B} \times \sqrt{B}$ submatrices.

Theorem 5 ([60, 90, 107, 110]). *The number of I/Os required to multiply two $k \times k$ matrices or to compute the LU factorization of a $k \times k$ matrix is $\Theta(k^3 / \min\{k, \sqrt{M}\} DB)$.*

Hong and Kung [60] and Nodine et al. [80] give optimal EM algorithms for iterative grid computations, and Leiserson et al. [74] show how to reduce the number of I/Os of naive multigrid implementations by a $\Theta(M^{1/5})$ factor. Gupta et al. [56] show how to derive efficient EM algorithms automatically for computations expressed in tensor form. Different techniques are called for when the matrices are sparse. The reader is referred to [35, 76, 87, 97, 96, 104, 106] for further study of EM matrix algorithms.

5 Batched Problems in Computational Geometry

Problems involving massive amounts of geometric data are ubiquitous in spatial databases [72, 88, 89], geographic information systems (GIS) [72, 88, 100], constraint logic programming [65, 66], object-oriented databases [112], statistics, virtual reality systems, and computer graphics [88]. NASA's Earth Observing System, for example, produces petabytes (10^{15} bytes) of raster data per year. A major challenge is to develop mechanisms for processing the data, or else much of it will be wasted.

For systems of this size to be efficient, we need efficient EM algorithms and data structures for basic problems in computational geometry. Luckily, many problems on geometric objects can be reduced to a small number of base problems, such as computing intersections, convex hulls, or nearest neighbors. Useful paradigms have been developed for solving these problems.

For brevity in the remainder of this paper, we deal only with the single-disk case $D = 1$. The I/O bounds for the batched problems can often be cut by a factor of (D) using the techniques of Section 2.

Theorem 6. *The following batched problems and several related problems involving N items and Q queries can be solved using*

$$O(n + q) \log_m n + z \tag{8}$$

I/Os, where $q = dQ/Be$ (or else $q = 0$ if Q is not defined) and $z = dZ/Be$ for output size Z:

1. *Answering Q orthogonal range queries on N points,*
2. *Finding all intersections between N nonintersecting red line segments and N nonintersecting blue line segments.*
3. *Computing the pairwise intersections of N segments,*
4. *Constructing the 2-D and 3-D convex hull of N points,*
5. *Triangulation of N points,*
6. *Performing Q point location queries in a planar subdivision of size N,*
7. *Finding all nearest neighbors for a set of N points,*

Goodrich et al. [52], Arge et al. [18], Arge et al. [17], and Crauser et al. [38] develop EM algorithms for these problems using the following EM paradigms for batched problems:

Distribution sweeping: a generalization of the distribution paradigm of Section 2 for externalizing plane sweep algorithms;

Persistent B-trees: an offline method for constructing an optimal-space persistent version of the B-tree data structure (see Section 7), yielding a factor of B improvement over the generic persistence techniques of Driscoll et al. [43].

Batched filtering: a general method for performing simultaneous external memory searches in data structures that can be modeled as planar layered directed acyclic graphs and in external fractionally cascaded data structures; useful for 3-D convex hulls and batched point location.

External fractional cascading: an EM analog to fractional cascading.

Online filtering: a technique based upon the work of Tamassia and Vitter [95] for online queries in data structures with fractional cascading.

External marriage-before-conquest: an EM analog to the technique of Kirkpatrick and Seidel [68] for output-sensitive convex hull constructions.

Randomized incremental construction with gradations, a localized version of the incremental construction paradigm of Clarkson and Shor [34].

The distribution sweep paradigm is fundamental to sweep line processes. For example, we can compute the pairwise intersections of N orthogonal segments in the plane by the following recursive distribution sweep: At each level of recursion, the plane is partitioned into (m) vertical strips, each containing (N/m) of the segments' endpoints. We use a horizontal sweep line to process the N segments from top to bottom. When a vertical segment is encountered by the sweep line, the segment is inserted into the appropriate strip. When a horizontal segment h is encountered, we report h's intersections with all the "active" vertical segments in the strips that are spanned completely by h. (A vertical segment is "active" if it is intersected by the current sweep line; vertical segments that are found to be no longer active are deleted from the strips.) The remaining end portions of h (which partially span a strip) are passed recursively to the next level, along with the vertical segments. After the initial sorting preprocessing, each of the $O(\log_m n)$ levels of recursion requires $O(n)$ I/Os, yielding the desired bound (8). Arge et al. [17] develop a unified approach to distribution sweep in higher dimensions.

Crauser et al. [38] use an incremental randomized construction to attain the I/O bound (8) for several other geometry problems.

6 Batched Problems on Graphs

The first work on EM graph algorithms was by Ullman and Yannakakis [98] for the problem of transitive closure. Chiang et al. [30] consider a variety of graph problems, several of which have upper and lower I/O bounds related to permuting. One key idea Chiang et al. exploit is that efficient EM algorithms can often be developed by a sequential simulation of a parallel algorithm for the same problem. Sorting is done periodically to reblock the data. In list ranking, which is used as a subroutine in the solution of several other graph problems, the number of working processors in the parallel algorithm decreases geometrically with time, so the number of I/Os for the entire simulation is proportional to the number of I/Os used in the first phase, which is given by the sorting bound $(n/D)\log_m n$. Dehne et al. [40] and Sibeyn and Kaufmann [94] show how to achieve the same I/O bound if $\log B = O(\log(M/B))$ by exploiting coarse-grained parallel algorithms whose data access characteristics permit the periodic sortings to be done with a linear number of I/Os.

For list ranking, the optimality of the EM algorithm in [30] assumes that $D\sqrt{m}\log m = \Omega(\log n)$, which is usually true. That assumption can be removed by use of the buffer tree data structure [9] (see Section 7.1). A practical, randomized implementation of list ranking appears in [93]. Recent work on other EM graph algorithms appears in [1, 10, 71, 54]. The problem of how to store graphs on disks

for efficient traversal is discussed in [79, 4]. Constructing classification trees in external memory for data mining is discussed in [109].

The I/O complexities of several of the basic graph problems considered in [30, 98] remain open, including connectivity, topological sorting, shortest paths, breadth-first search, and depth-first search. For example, for a graph with V vertices and E edges, the best-known EM algorithms for depth-first search and transitive closure require $(\frac{V}{M}\frac{E}{B} + V)$ and $(V^2 \sqrt{E/M}/B)$ I/Os, respectively. Connected components can be determined in $O(\min\{\frac{V^2}{B}\log_m \frac{V^2}{B}, (\log \frac{V}{M})\frac{E}{B}\log_m \frac{E}{B}\})$ I/Os deterministically and in only $O(\frac{E}{B}\log_m \frac{V}{B})$ I/Os probabilistically. The interesting connection between the parallel domain and the EM domain suggests that there may be relationships between computational complexity classes related to parallel computing (such as P-complete problems) and those related to I/O efficiency.

7 Dynamic Multiway Tree Data Structures

Tree-based data structures arise naturally in the dynamic online setting, in which the data can be updated and queries must be processed immediately. Binary trees have a host of applications in the RAM model. In order to exploit block transfer, trees in external memory generally use a block for each node, which can store $\Theta(B)$ pointers and data values. A tree of degree B^c, for fixed $c > 0$, with n leaf nodes has height $\lceil c \log_B n \rceil$. The well-known *B-tree* due to Bayer and McCreight [23, 69] is a balanced multiway tree with height $\Theta(\log_B n)$ in which each node has degree $\Theta(B)$. It supports dynamic dictionary operations and one-dimensional range queries in $O(\log_B n + t)$ I/Os per operation. Persistent versions of B-trees have been developed by Becker et al. [24] and Varman and Verma [101]. Lomet and Salzberg [77] explore mechanisms to add concurrency and recovery to B-trees.

Arge and Vitter [19] give a useful variant of B-trees called *weight-balanced B-trees* with the property that the number of data items in any subtree of height h is $\Theta(a^h)$, for some fixed parameter a of order B. (By contrast, the sizes of subtrees at level h in a regular B-tree can differ by a multiplicative factor that is exponential in h.) When a node on level h gets rebalanced, no further rebalancing is needed until its subtree is updated $\Omega(a^h)$ times. Weight-balanced B-trees were originally developed as part of an optimal dynamic EM data structure for stabbing queries (interval trees) and segment trees, but they can also be used to get improvements for algorithms developed in the RAM model [19, 54].

Grossi and Italiano [55] develop a multidimensional version of B-trees, called *cross trees*, that combine the data-driven partitioning of B-trees at the upper levels of the tree with the space-driven partitioning of methods like quad trees at the lower levels of the tree. The data structure also supports the dynamic operations of split and concatenate.

7.1 Buffer Trees

Many batched problems in computational geometry can be solved by plane sweep techniques. For example, in Section 5 we showed how to compute or-

thogonal segment intersections by keeping track of the active vertical segments during the sweep process. If we use a B-tree to store the active vertical segments, each insertion and query uses $O(\log_B n)$ I/Os, resulting in a huge I/O bound of $O(N \log_B n)$, which is more than B times larger than the desired bound. One solution suggested in [105] is to use a binary tree in which items are pushed lazily down the tree in blocks of B items at a time. The binary nature of the tree results in a data structure of height $\log_2 n$, yielding a total I/O bound of $O(n \log_2 n)$, which is still nonoptimal by a significant $\log m$ factor.

Arge [9] developed the elegant *buffer tree* data structure to support *batched dynamic* operations such as in this example, where the queries do not have to be answered right away or in any particular order. The buffer tree is a balanced multiway tree, but with degree (m). Its key distinguishing feature is that each node has a buffer that can store M items. Items in a node are not pushed down to the children until the buffer fills. Emptying the buffer requires $O(m)$ I/Os, which amortizes the cost of distributing the items to the respective children. Each item incurs an amortized cost of $1=B$ I/Os per level. Buffer trees can be used as a subroutine to the standard sweep line algorithm to get an optimal EM algorithm for orthogonal segment intersection. Arge showed how to extend buffer trees to implement external memory segment trees by reducing the node degrees to $(\sqrt{m})$ and by introducing "multislabs" in each node.

Buffer trees provide a natural amortized implementation of priority queues for use in applications like discrete event simulation, sweeping, and list ranking. Brodal and Katajainen [27] develop a worst-case optimal priority queue, in the sense that every sequence of B *insert* and *delete_min* operations require only $O(\log_m n)$ I/Os.

7.2 R-trees

The *R-tree* of Guttman [57] and its many variants are an elegant generalization of the B-tree for storing geometric objects. Each node in the tree has associated with it a bounding box (or bounding polygon) of all the elements in its subtree. The bounding boxes of sibling nodes can overlap. If the tree is being used for point location, for example, a point may lie within the bounding box of several children of the current node in the search. In that case, the search must proceed to all such children.

Several heuristics for where to insert new items into the R-tree and how to rebalance have been proposed. Extensive surveys appear in [53, 49]. The *R*-tree* of Beckmann et al. [25] seems to give best overall query performance. Constructing an R*-tree by repeated insertions, however, is extremely slow. A faster alternative is to use the Hilbert R-tree of Kamel and Faloutsos [62, 63]. Each item is labeled with the position of its center on the Hilbert space-filling curve, and a B-tree is built on the totally ordered labels. Bulk loading a Hilbert R-tree is therefore easy once the center points are presorted, but the quality of the Hilbert R-tree in terms of query performance is not as good as that of an R*-tree, especially for higher-dimensional data [26, 64].

Arge et al. [13] and Bercken et al. [99] have independently devised fast bulk loading methods for R*-trees that are based upon buffer trees. The former

method is especially efficient and can even support dynamic batched updates and queries. Experiments with this technique are discussed in Section 10.

8 Range Searching

Multidimensional range search is a fundamental primitive in several large geometric applications and it provides indexing support for new constraint data models and object-oriented data models. (See [66] for background.)

Range searching in a batched setting has been discussed in Section 5, so in this section we concentrate on the important online case. R-trees and the methods mentioned in Section 7.2 perform well in many typical cases but have no worst-case bounds. Unfortunately, for many search problems it is very difficult to develop theoretically optimal algorithms; many open problems remain. The primary theoretical challenges are four-fold:

1. to get a query complexity close to $O(\log_B n)$ I/Os,
2. to get an output complexity of $O(z)$ I/Os, where $z = \lceil Z/B \rceil$ and Z is the output size,
3. to use close to linear disk storage space, and
4. to support dynamic updates.

Arge and Vitter [19] design an interval tree based upon the weight-balanced B-tree that meets all four goals. It solves the problems of stabbing queries and dynamic interval management, following up on earlier work by Kanellakis et al. [66]. The EM interval tree is used by [32] to extract at query time the boundary components of the isosurface (or contour) of a surface. A data structure for a related problem, which has optimal output complexity, appears in [4].

Ramaswamy and Subramanian [85] introduce the notion of *path caching* to develop EM algorithms for two-sided and three-sided 2-D range queries. Their algorithms are optimal in criteria 1 and 2 but with higher storage overheads and amortized and/or nonoptimal update bounds. Subramanian and Ramaswamy [85] present the *P-range tree* data structure for the three-sided and (general) four-sided 2-D range queries. The update times are amortized and slightly nonoptimal, and the space overhead for four-sided queries is $O(n(\log n)/\log\log_B n)$.

On the other hand, Subramanian and Ramaswamy [85] prove for an external memory version of the pointer machine model that no EM algorithm can achieve both criteria 1 and 2 using less than $O(n(\log n)/\log\log_B n)$ space, and thus their algorithm is near-optimal. Hellerstein et al. [58] consider a generalization of the layout-based lower bound argument of Kanellakis et al. [66] for studying the tradeoff between disk space overhead and query performance; the model does not distinguish, however, between query complexity and output complexity. Further work appears in [70].

When the data structure is restricted to contain only a single copy of each item, Kanth and Sing [67] show for a restricted class of index-based trees that range queries in the worst case require $\Omega(n^{1-1/d} + z)$ I/Os; a matching upper bound is provided by the cross tree data structure [55] of Section 7.

Fundamentally different techniques are needed for higher dimensions and for non-orthogonal queries. Vengroff and Vitter [103] use a partitioning approach and compressed representation to get the first EM algorithms for three-dimensional searching that are within a polylog factor of optimal. Agarwal et al. [3] give near-optimal bounds for halfspace range searching in two dimensions and some variants in higher dimensions using another type of partitioning structure.

Callahan et al. [28] develop a dynamic EM data structure for use in several online problems such as finding an approximately nearest neighbor and maintaining the closest pair of vertices. Numerous other data structures have been developed for range queries and related problems on spatial data. We refer to [78, 5] for a survey.

9 String Processing

Digital trie-based structures, in which branching decisions at each node are made based upon the values of particular bits in strings, are effective for string processing in internal memory. In EM applications, what is needed is a multiway digital structure. Unfortunately, if the strings are long, there is no space to store them completely in each node, and if pointers to strings are used, the number of I/Os per node access will be large.

Ferragina and Grossi [46, 47] develop an elegant generalization of the B-tree, called the *SB-tree*, for storing strings. The key problem they address is how to represent a single node that does B-way branching. In a B-tree, $B - 1$ items are stored in the node to guide the searching, and each item is assumed to be unit-sized. However, strings can occupy many characters, so there is not enough space to store $B - 1$ strings in each node. Pointers to the $B - 1$ strings could be stored, but access to them during search would require more than a constant number of I/Os.

Their solution, called the *blind trie*, is based on the data structure of Ajtai et al. [8] that achieves B-way branching with a total storage of $O(B)$ characters. The resulting query time to search in an SB-tree for a string of length ℓ is $O(\log_B n + \ell/B)$, which is optimal. Ferragina and Grossi apply SB-trees to string matching, prefix search, and substring search. Farach and Ferragina [44] show how to construct SB-trees, suffix trees, and suffix arrays on strings of length N using $O(n \log_m n)$ I/Os, which is optimal. Clark and Munro [33] give an alternate approach to suffix trees.

Arge et al. [12] consider several models for the problem of sorting K strings of total length N in external memory. They develop efficient sorting algorithms in these models, making use of the SB-tree, buffer tree techniques, and a simplified version of the SB-tree for merging called the *lazy trie*. They show somewhat counterintuitively that for sorting short strings (i.e., strings whose length is less than B) the complexity depends upon the total *number of characters*, whereas for long strings the complexity depends upon the total *number of strings*.

10 Empirical Comparisons

In this section we examine the empirical performance of algorithms for two problems that arise in spatial databases. TPIE (Transparent Parallel I/O programming Environment) [102, 104] is used as the common implementation platform.[3] TPIE is a comprehensive software package that helps programmers develop high-level, portable, and efficient implementations of EM algorithms.

In the first experiment, three algorithms are implemented in TPIE for the problem of rectangle intersection, which is often the first step in a spatial join computation. The first method, called Scalable Sweeping-Based Spatial Join (SSSJ) [16], is a robust new algorithm based upon the distribution sweep paradigm of Section 5. The other two methods are Partition-Based Spatial-Merge (QPBSM) used in Paradise [84] and a new modification called MPBSM that uses an improved dynamic data structure for intervals [16].

The algorithms were tested on several data sets. The timing results for the two data sets in Figures 2(a) and 2(b) are given in Figures 2(c) and 2(d), respectively. The first data set is the worst case for sweep line algorithms; a large fraction of the line segments in the file are active (i.e., they intersect the current sweep line). The second data set is a best case for sweep line algorithms. The two PBSM algorithms have the disadvantage of making extra copies. SSSJ shows considerable improvement over the PBSM-based methods. On more typical data, such as TIGER/line road data, experiments indicate that SSSJ and MPBSM run about 30% faster than QPBSM.

In the second experiment, three methods for building R-trees are evaluated in terms of their bulk loading time and the resulting query performance. The three methods tested are a newly developed buffer R-tree method [13] (labeled "buffer"), the naive sequential method for construction into R*-trees (labeled "naive"), and the best update algorithm for Hilbert R-trees (labeled "Hilbert") [64].

The experimental data came from TIGER/line road data sets from four U.S. states. The starting point in the experiment was a single R-tree for each of the four states, containing 75% of the road data objects for that state. Using each of the three algorithms, the remaining 25% of the objects were inserted into the R-tree, and the construction time was measured. The query performance of each resulting R-tree was tested by posing rectangle intersection queries, using rectangles taken from TIGER hydrographic data. The results in Table 2 show that the buffer R-tree has faster construction time than the Hilbert R-tree (the previous best method for construction time) and similar or better query performance than repeated insertions (the previous best method for query performance).

Other recent experiments involving the paradigms discussed in this paper appear in [31, 61].

[3] The TPIE software distribution is available at no charge on the World Wide Web at http://www.cs.duke.edu/TPIE/.

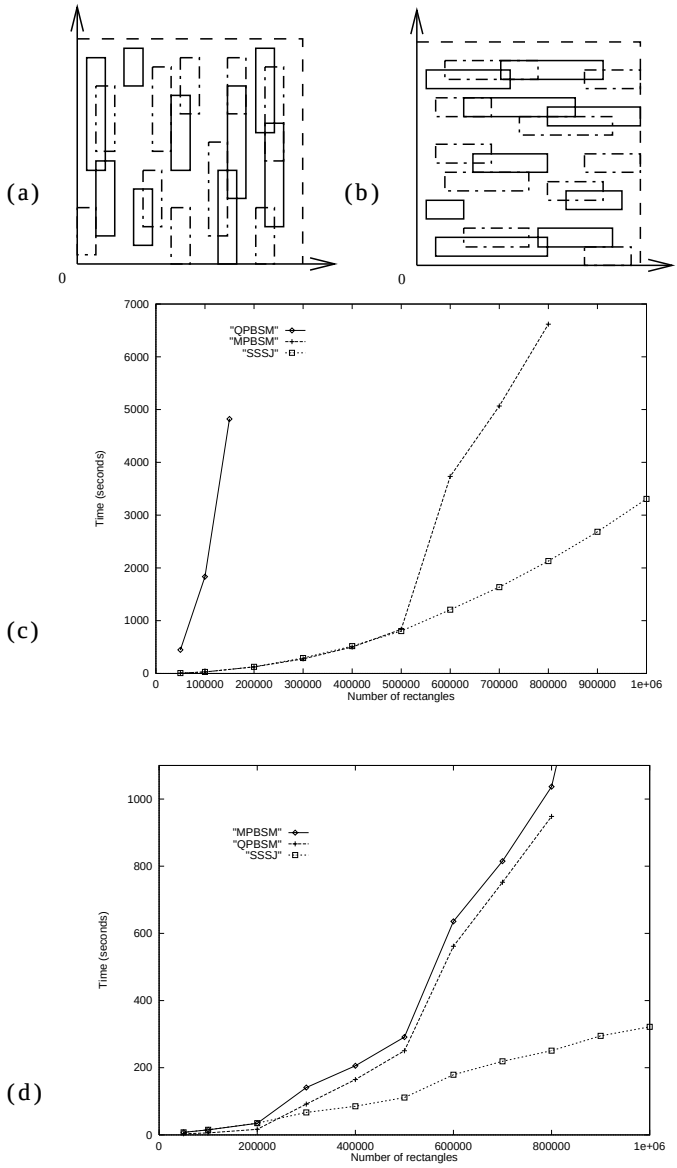


Fig. 2. Comparison of Scalable Sweeping-Based Spatial Join (SSSJ) with the original PBSM (QPBSM) and a new variant (MPBSM) (a) Data set 1 consists of tall and skinny (vertically aligned) rectangles. (b) Data set 2 consists of short and wide (horizontally aligned) rectangles. (c) Running times on data set 1. (d) Running times on data set 2.

11 Dynamic Memory Allocation

It is often the case that operating systems dynamically change the amount of memory allocated to a program. The algorithms in the earlier sections assume a

Data Set	Update Method	Update with 25% of the data Building	Querying	Packing
RI	naive	259; 263	6; 670	64%
	Hilbert	15; 865	7; 262	92%
	buffer	13; 484	5; 485	90%
CT	naive	805; 749	40; 910	66%
	Hilbert	51; 086	40; 593	92%
	buffer	42; 774	37; 798	90%
NJ	naive	1; 777; 570	70; 830	66%
	Hilbert	120; 034	69; 798	92%
	buffer	101; 017	65; 898	91%
NY	naive	3; 736; 601	224; 039	66%
	Hilbert	246; 466	230; 990	92%
	buffer	206; 921	227; 559	90%

Table 2. Summary of the I/O costs for R-tree update.

fixed memory allocation; they must make use of virtual memory if the memory allocation is reduced, often causing a severe performance hit.

Barve and Vitter [22] discuss the design and analysis of EM algorithms that adapt gracefully to changing memory allocations. In their model, without loss of generality, a program P is allocated memory in phases: During the ith phase, P is allocated m_i blocks of internal memory, and this memory remains allocated to P until P completes $2m_i$ I/O operations, at which point the next phase begins. The process continues until P finishes execution. The duration for each memory allocation phase is long enough to allow all the memory in that phase to be used.

For sorting, the lower bound approach of (6) implies that $\sum_i 2m_i \log m_i = \Omega(n \log n)$. We say that P is *dynamically optimal* for sorting if $\sum_i 2m_i \log m_i = O(n \log n)$ for all possible sequences $m_1, m_2, \ldots$ of memory allocation. Intuitively, if P is dynamically optimal, no other program can perform more than a constant number of sorts in the worst-case for the same allocation sequence.

Barve and Vitter define the model in generality and give dynamically optimal strategies for sorting, matrix multiplication, and buffer trees operations. Their work represents the first theoretical model of dynamic allocation for EM algorithms. Pang et al. [83] and Zhang and Larson [113] give memory-adaptive merge sort algorithms, but their algorithms handle only special cases and can be made to perform poorly for certain patterns of memory allocation.

12 Conclusions

In this paper we have described several useful paradigms for the design and implementation of efficient external memory algorithms. The problem domains we have considered include sorting, permuting, FFT, scientific computing, computational geometry, graphs, databases, geographic information systems, and text and string processing. Algorithmic challenges remain in virtually all these problem domains. One example is the design and analysis of new algorithms

for efficient use of multiple disks. Optimal bounds are not yet determined for basic graph problems like topological sorting, shortest paths, breadth-first and depth-first search, and connectivity. There is an interesting connection between problems that have good I/O speedups and problems that have fast and work-efficient parallel algorithms.

For some of the problems that can be solved optimally up to a constant factor, the hidden constant is too large for the algorithm to be of use in practice, and simpler approaches are needed. A continuing goal is to develop optimal EM algorithms and to translate theoretical gains into observable improvements in practice. New architectures such as networks of workstations and hierarchical storage devices present interesting challenges and opportunities. Work is beginning, for example, on extensions of TPIE to such domains. The ability to adapt to changing memory allocations may be important for the use of EM algorithms in practice.

References

1. J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external memory graph algorithms. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, 1998.
2. M. Adler. New coding techniques for improved bandwidth utilization. In *37th IEEE Symp. on Foundations of Computer Science*, 173–182, Burlington, VT, October 1996.
3. P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. In *Proc. 17th ACM Symp. on Princ. of Database Systems*, 1998.
4. P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998.
5. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, E. Goodman, and R. Pollack, editors, *Discrete and Computational Geometry: Ten Years Later*, 63–72. American Mathematical Society Press, to appear.
6. A. Aggarwal and C. G. Plaxton. Optimal parallel sorting in multi-level storage. *Proc. Fifth Annual ACM-SIAM Symp. on Discrete Algorithms*, 659–668, 1994.
7. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1116–1127, 1988.
8. M. Ajtai, M. Fredman, and Komlos. Hash functions for priority queues. *Information and Control*, 63(3), 217–225, 1984.
9. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, 334–345, 1995. A complete version appears as BRICS technical report RS–96–28, University of Aarhus.
10. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, 82–91, 1995.
11. L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, LNCS 1340, 1997.
12. L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symposium on Theory of Computation*, 540–548, 1997.
13. L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees, 1998. Manuscript.
14. L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. 3rd Workshop on Algorithms and Data Structures*, volume 709, 83–94. Lecture Notes in Computer Science, Springer-Verlag, 1993.
15. L. Arge and P. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, 1998.

16. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. 24th Intl. Conf. on Very Large Databases*, New York, August 1998.
17. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998.
18. L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, to appear. Special issue on cartography and geographic information systems.
19. L. Arge and J. S. Vitter. Optimal interval management in external memory. In *Proc. 37th IEEE Symp. on Found. of Computer Sci.*, 560–569, Burlington, VT, October 1996.
20. R. Barve, P. B. Gibbons, B. Hillyer, Y. Matias, E. Shriver, and J. S. Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus: the long version. Technical report, Bell Labs, 1997.
21. R. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4), 601–631, 1997.
22. R. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory, 1998. Manuscript.
23. R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Inform.*, 1, 173–189, 1972.
24. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4), 264–275, Dec. 1996.
25. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD International Conf. on Management of Data*, 322–331, 1990.
26. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *International Conf. on Extending Database Technology*, 1998.
27. G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. Technical Report DIKU Report 97/25, University of Copenhagen, October 1997.
28. P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, 381–392, 1995.
29. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Comp. Surveys*, 26(2), 145–185, June 1994.
30. Y.-J. Chiang, , M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 139–149, January 1995.
31. Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. 1995 Work. Algs. and Data Structures*, 1995.
32. Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization Conf.*, 1997.
33. D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 383–391, Atlanta, June 1996.
34. K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete and Computational Geometry*, 4, 387–421, 1989.
35. P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snir, B. Traversat, and P. Wong. Overview of the MPI-IO parallel I/O interface. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, volume 362 of *The Kluwer International Series in Engineering and Computer Science*, chapter 5, 127–146. Kluwer Academic Publishers, 1996.
36. T. H. Cormen and D. M. Nicol. Performing out-of-core FFTs on parallel disk systems. *Parallel Computing*, 1998. To appear; available as Dartmouth Report PCS-TR96-294.
37. T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM J. Computing*, to appear.
38. A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for geometric problems. In *Proc. 14th ACM Symp. on Computational Geometry*, June 1998.

39. R. Cypher and G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *J. Computer and System Sci.*, 47(3), 501–548, 1993.
40. F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proc. 9th ACM Symp. on Parallel Algorithms and Architectures*, 106–115, June 1997.
41. H. B. Demuth. *Electronic Data Sorting*. PhD thesis, Stanford University, 1956. A shortened version appears in *IEEE Transactions on Computing*, C-34(4):296–310, April 1985, special issue on sorting, E. E. Lindstrom, C. K. Wong, and J. S. Vitter, editors.
42. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. First International Conf. on Parallel and Distributed Information Systems*, 280–291, December 1991.
43. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38, 86–124, 1989.
44. M. Farach and P. Ferragina. Optimal suffix tree construction in external memory, November 1997. Manuscript.
45. W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, New York, third edition, 1968.
46. P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. 27th Annual ACM Symp. on Theory of Computing*, 693–702, Las Vegas, 1995.
47. P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 373–382, Atlanta, June 1996.
48. R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, 105–109. Plenum, 1972.
49. V. Gaede and O. Günther. Multidimensional access methods. *Computing Surveys*, 1998.
50. M. Gardner. *Magic Show*, chapter 7. Knopf, New York, 1977.
51. G. A. Gibson, J. S. Vitter, and J. Wilkes. Report of the working group on storage I/O issues in large-scale computing. *ACM Computing Surveys*, 28(4), December 1996. Also available as http://www.cs.duke.edu/~jsv/SDCR96-IO/report.ps.
52. M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Computer Science*, 714–723, 1993.
53. D. Greene. An implementation and performance analysis of spatial data access methods. In *Proc. IEEE International Conf. on Data Engineering*, 606–615, 1989.
54. R. Grossi and G. F. Italiano. Efficient splitting and merging algorithms for order decomposable problems. In *24th International Colloquium on Automata, Languages and Programming*, volume 1256 of *LNCS*, 605–615, Bologna, Italy, July 1997.
55. R. Grossi and G. F. Italiano. Efficient cross-trees for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, 1998.
56. S. K. S. Gupta, Z. Li, and J. H. Reif. Generating efficient programs for two-level memories from tensor-products. In *Proc. Seventh IASTED/ISMM International Conf. on Parallel and Distributed Computing and Systems*, 510–513, Washington, D.C., October 1995.
57. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf. on Management of Data*, 47–57, 1985.
58. J. M. Hellerstein, E. Koutsoupias, and C. H. Papadimitriou. On the analysis of indexing schemes. In *Proc. 16th ACM Symp. on Principles of Database Systems*, 249–256, Tucson, Arizona, May 1997.
59. L. Hellerstein, G. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2–3), 182–208, 1994.
60. J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. *Proc. 13th Annual ACM Symp. on Theory of Computation*, 326–333, May 1981.
61. D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. Workshop on Algorithm Engineering, 1997. Electronic proceedings available at http://www.dsi.unive.it/~wae97/proceedings/.
62. I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. 2nd International Conf. on Information and Knowledge Management (CIKM)*, 490–499, 1993.
63. I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proc. 20th International Conf. on Very Large Databases*, 500–509, 1994.

64. I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proc. 4th International Symp. on Spatial Data Handling*, 3B, 31–42, 1996.
65. P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *Proc. 9th ACM Conf. on Princ. of Database Systems*, 299–313, 1990.
66. P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Systems*, 233–243, 1993. To appear in a special issue of *Journ. Comput. Sys. Science*.
67. K. V. R. Kanth and A. K. Sing. Optimal dynamic range searching in non-replicating index structures. Technical Report CS97–13, UCSB, July 1997.
68. D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM Journal on Computing*, 15, 287–299, 1986.
69. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second edition, 1998.
70. E. Koutsoupias and D. S. Taylor. Tight bounds for 2-dimensional indexing schemes. In *Proc. 17th ACM Conf. on Princ. of Database Systems*, Seattle, WA, June 1998.
71. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE Symp. on Parallel and Distributed Processing*, 169–176, October 1996.
72. R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, 1992.
73. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34(4), 344–354, April 1985. Special issue on sorting, E. E. Lindstrom and C. K. Wong and J. S. Vitter, editors.
74. C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *IEEE Foundations of Comp. Sci.*, 704–713, 1993.
75. Z. Li, P. H. Mills, and J. H. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications*, 8, 35–59, 1996.
76. J. W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3), 249–264, Sept. 1986.
77. D. B. Lomet and B. Salzberg. Concurrency and recovery for index trees. *The VLDB Journal*, 6(3), 224–240, 1997.
78. J. Nievergelt and P. Widmayer. Spatial data structures: Concepts and design choices. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*. Springer-Verlag, LNCS 1340, 1997.
79. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2), 181–214, August 1996.
80. M. H. Nodine, D. P. Lopresti, and J. S. Vitter. I/O overhead and parallel vlsi architectures for lattice computations. *IEEE Transactions on Computers*, 40(7), 843–852, July 1991.
81. M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures*, 120–129, June–July 1993.
82. M. H. Nodine and J. S. Vitter. Greed Sort: An optimal sorting algorithm for multiple disks. *J. ACM*, 42(4), 919–933, July 1995.
83. H. Pang, M. Carey, and M. Livny. Memory-adaptive external sorts. *Proc. 19th Conf. on Very Large Data Bases*, 1993.
84. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. ACM SIGMOD International Conf. on Management of Data*, volume 25, 2 of *ACM SIGMOD Record*, 259–270, June 1996.
85. S. Ramaswamy and S. Subramanian. Path caching: a technique for optimal external searching. *Proc. 13th ACM Conf. on Princ. of Database Systems*, 1994.
86. C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 17–28, Mar. 1994.
87. J. Salmon and M. Warren. Parallel out-of-core methods for N-body simulation. In *Proc. Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.
88. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, 1989.
89. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

90. J. E. Savage and J. S. Vitter. Parallelism in space-time tradeoffs. In F. P. Preparata, editor, *Advances in Computing Research, Volume 4*, 117–146. JAI Press, 1987.
91. E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Joint International Conf. on Measurement and Modeling of Computer Systems*, June 1998.
92. E. A. M. Shriver and M. H. Nodine. An introduction to parallel I/O models and algorithms. In R. Jain, J. Werth, and J. C. Browne, editors, *Input/Output in Parallel and Distributed Computer Systems*, chapter 2, 31–68. Kluwer Academic Publishers, 1996.
93. J. F. Sibeyn. From parallel to external list ranking. Technical Report MPI–I–97–1–021, Max-Planck-Institut, Sept. 1997.
94. J. F. Sibeyn and M. Kaufmann. BSP-like external-memory computation. In *Proc. 3rd Italian Conf. on Algorithms and Complexity*, 229–240, 1997.
95. R. Tamassia and J. S. Vitter. Optimal cooperative search in fractional cascaded data structures. *Algorithmica*, 15(2), 154–171, February 1996.
96. S. Toledo. Out of core algorithms in numerical linear algebra, a survey. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, 1998.
97. S. Toledo and F. G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In *Proc. Fourth Workshop on Input/Output in Parallel and Distributed Systems*, 28–40, Philadelphia, May 1996.
98. J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3, 331–360, 1991.
99. J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings 23rd VLDB Conf.*, 406–415, 1997.
100. M. van Kreveld, J. Nievergelt, T. Roos, and P. W. (Eds.). *Algorithmic Foundations of GIS*. LNCS 1340. Springer-Verlag, 1997.
101. P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), 391–409, May/June 1997.
102. D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1995. The manual and software distribution are available on the web at http://www.cs.duke.edu/TPIE/.
103. D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. 28th Annual ACM Symp. on Theory of Computing*, Philadelphia, PA, May 1996.
104. D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. Goddard Conf. on Mass Storage Systems and Technologies*, volume II of *NASA Conf. Publication 3340*, 553–570, College Park, MD, September 1996.
105. J. S. Vitter. Efficient memory access in large-scale computation. In *Proc. 1991 Symp. on Theor. Aspects of Comp. Sci.*, LNCS. Springer-Verlag, 1991.
106. J. S. Vitter. External memory algorithms. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, 1998.
107. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3), 110–147, 1994.
108. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3), 148–169, 1994.
109. M. Wang, J. S. Vitter, and B. R. Iyer. Scalable mining for classification rules in relational databases. In *Proc. International Database Engineering & Application Symp.*, Cardiff, Wales, July 1998.
110. D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer I/O practical. In *Proc. 1993 DAGS/PC Symp.*, 56–63, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.
111. C. Wu and T. Feng. The universality of the shuffle-exchange network. *IEEE Transactions on Computers*, C-30, 324–332, May 1981.
112. S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kauffman, 1990.
113. W. Zhang and P.-A. Larson. Dynamic memory adjustment for external mergesort. *Proc. 23rd Intl. Conf. on Very Large Data Bases*, 1997.

Design and Analysis of Dynamic Processes: A Stochastic Approach
(Invited Paper)

Eli Upfal

Computer Science Department, Brown University
Box 1910, Providence, RI 02912.
E-mail: eli@cs.brown.edu
http://www.cs.brown.edu/people/eli

Abstract. Past research in theoretical computer science has focused mainly on static computation problems, where the input is known before the start of the computation and the goal is to minimize the number of steps till termination with a correct output. Many important processes in today's computing are dynamic processes, whereby input is continuously injected to the system, and the algorithm is measured by its long term, steady state, performance. Examples of dynamic processes include communication protocols, memory management tools, and time sharing policies. Our goal is to develop new tools for the design and analyzing the performance of dynamic processes, in particular through modeling the dynamic process as an infinite stochastic processes.

1 Introduction

Rigorous analysis of the dynamic performance of computer processes is one of the most challenging current goals in theory of computation. Past research in theoretical computer science has focused mainly on static computation problems. In static computation the input is known at the start of the computation and the goal is to minimize the number of steps till the process terminates with a correct output. Many important processes in today's computing are dynamic processes, whereby input is continuously injected to the system, and the algorithm (which is not supposed to terminate at all) is measured by its long term (steady state) performance. Examples of dynamic computer processes include:

- Contention resolution protocols.
- Routing and communication algorithms.
- Memory management tools such as cashing and paging.
- Time sharing and load balancing protocols.

1.1 Performance measures for dynamic processes

In evaluating the performance of dynamic algorithms one needs to distinguish between algorithms that satisfy all incoming requests and algorithms that may

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 26{34, 1998.
c Springer-Verlag Berlin Heidelberg 1998

drop requests. In the first case the primary performance measure of interest is the algorithm's stability conditions. Roughly speaking, a system is stable if in the long run the number of new arriving requests is no larger than the number of requests processed by the system. The goal of the analysis here is to characterize the (most general) input conditions (deterministic or stochastic) under which the system is stable. The corresponding measure for algorithms that may drop requests is the number of requests (as a function of the incoming stream of requests) that the algorithm successfully satisfied (or equivalently, the number of requests that the algorithm needs to reject in order to keep the system stable). A second criteria, which is important for both type of algorithms is speed, measured by the (maximum or expected) time to satisfy a request.

An algorithm is usually analyze in terms of its (worst case or average) performance with respect to a given class of inputs. However, in many dynamic algorithms it is useful to study instead the *competitive ratio* performance of the algorithm. Competitive ratio analysis compares the performance of the dynamic (on-line) algorithm to the performance of an "off-line" algorithm that "knows" in advance the whole sequence of requests. The quality of an on-line algorithm is measured by the maximum, over all possible input sequences, of the ratio between its performance and the performance of an off-line on the same input sequence. Thus, competitive analysis measures the quality of the algorithm with respect to the optimal one, rather then measuring the actual performance of the system. This measure is particular useful in cases when no algorithm can perform well on all instances of the set of inputs.

1.2 Stochastic analysis

As in the case of static algorithms, randomness is introduced into dynamic computation through either the algorithm, the input or both. Many interesting dynamic protocols are random, a well known example is the Aloha contention resolution protocol [32] which is used in the Ethernet and other similar applications. An execution of a dynamic random algorithm, even on a fixed input sequence, defines an infinite stochastic process in which a state at a given step depends on the history of the process. Analysis of such a process requires different approach and tools from the ones used for analyzing the finite execution of a randomized static computation.

Worst case analysis rarely gives an interesting insight on the actual performance of a dynamic algorithm. A worst case adversary can generate extremely hard sequences of requests, and the performance of the algorithm on these "pathological" cases does not accurately represent the efficiency of the algorithm. To offset the affect of rare cases it is useful to analyze the performance of dynamic systems under some stochastic assumptions on the stream of inputs. Such assumptions are more realistic in the dynamic setting, in particular when requests are originated by a number of independent processors, than in the case of static analysis. The stochastic process that control the stream of requests might be stationary, periodic, or even bursty. The goal is to obtain results that are valid under the weakest set of assumptions. The advantage and practicality

of this approach has been well demonstrated by the achievements of queueing theory [26]. Our goal is to applied similar techniques to dynamic computer processes that do not fit the queueing theory settings. Competitive ratio analysis is another alternative to standard worst case analysis. However, even in the case of competitive analysis, stochastic input may lead to better evaluation of the algorithm's performance. One example is greedy load balancing. This procedure is very efficient in practice, while its worst case competitive ratio is high. On the other hand it can be shown that under some stochastic assumptions the average competitive ratio of greedy load balancing is very low [5], thus giving one possible explanation for the "real-life" performance of this procedure.

Stochastic analysis of dynamic processes builds on the rich theory of stochastic processes, in particular queueing theory, and the theory of stationary processes. However, in many cases one needs to develop new tools to address the specific problems pose by computer related processes, which are discrete and involved complicated dependency conditions.

2 Dynamic Balanced Allocations

In [5] we studied a surprising combinatorial phenomena which we term *balanced allocations*. In the dynamic version of balanced allocation we have n boxes and n balls. In each step one ball, chosen uniformly at random, is removed from the system, and a new ball appears. The new ball comes with d possible destinations, chosen independently at random, and it is placed into the least full box, at the time of the placement, among the d possible destinations.

The case $d = 1$ corresponds to the classic occupancy problem. It is easy to show that in the stationary distribution, with high probability the fullest box has $(\log n / \log\log n)$ balls. The analysis of the case $d \quad 2$ is significantly harder, since the locations of the current n balls might depend on the locations of balls that are no longer in the system. A surprising result which we proved in [5] shows that for $d \quad 2$, in the stationary distribution, with high probability no box contains more than $\ln\ln n / \ln d + O(1)$ balls. Thus, an apparent minor change in the random allocation process results in an exponential decrease in the maximum occupancy per location. Several recent works have built on our original result. Mitzenmacher [34] studied a continuous time variant of our model in which balls arrive according to a Poisson process and are removed with an exponential distribution. Czumaj and Stemann [14] showed that a more efficient variant of our placement procedure results in similar maximum load.

The dynamic balanced allocation process has a number of algorithmic applications:

Dynamic dictionary. The efficiency of a hashing technique is measured by two parameters: the expected and the maximum access time. Our approach suggests a simple hashing technique, similar to hashing with chaining. We call it *2-way chaining*. It has $O(1)$ expected, and $O(\log\log n)$ maximum access time. We use two random hash functions. The two hash functions define two possible entries in the table for each key. The key is inserted to the least full location, at

the time of the insertion. Keys in each entry of the table are stored in a linked list. The expected insertion and look-up time is $O(1)$, and with high probability the maximum access time is $\ln n / \ln \ln 2 + O(1)$, versus the $(\log n / \log \log n)$ time when one random hash function is used.

Dynamic Resource Allocation. Consider a scenario in which a user or a process has to choose on-line between a number of identical resources (choosing a server to use among the servers in a network; choosing a disk to store a directory; etc.). To find the least loaded resource, users may check the load on all resources before placing their requests. This process is expensive, since it requires sending an interrupt to each of the resources. A second approach is to send the task to a random resource. This approach has minimum overhead, but if all users follow it, the difference in load between different servers will vary by up to a logarithmic factor. The balanced allocation process suggests a more efficient solution. If each user samples the load of two resources and sends his request to the least loaded, the total overhead is small, and the load on the n resources varies by only a $O(\log \log n)$ factor, an exponential improvement over the pure random allocation.

3 Dynamic Flow Control for Packet Routing With Bounded Bu ers

Most theoretical work on routing algorithms has focused on static routing: A set of packets is injected into the system at time 0, and the routing algorithm is measured by the time it takes to deliver all the packets to their destinations, assuming that no new packets are injected in the meantime (see Leighton [29] for an extensive survey). In practice however, networks are rarely used in this "batch" mode. Most real-life networks operate in a *dynamic* mode whereby new packets are continuously injected into the system. Each processor usually controls only the rate at which it injects its own packets and has only a limited knowledge of the global state. This situation is better modeled by a stochastic paradigm whereby packets are continuously injected according to some inter-arrival distribution, and the routing algorithm is evaluated according to its long term behavior. goal is to develop algorithms that perform close to optimal for a variety of inter-arrival distributions.

Several recent articles have addressed the dynamic routing problem, in the context of packet routing on arrays [28,33,27,8], on the hypercube and the butterfly [38] and general networks [37]. Except for [8], the analyzes in these works assumes a Poisson arrival distribution and requires unbounded queues in the routing switches (though some works give a high probability bound on the size of the queue used [28,27]). Unbounded queues allow the application of some tools from queuing theory (see [20,21]) and help reduce the correlation between events in the system, thus simplifying the analysis at the cost of a less realistic model.

In [9] we focused on the design and analysis of efficient flow control mechanism for dynamic packet routing in networks with bounded buffers at the switching nodes, a setting that more accurately models real networks. Our goal was to

build on the vast amount of work that has been done for static routing in order to obtain results for the dynamic setting. In [9] we developed a general technique in which a static algorithm for a network with bounded buffers is augmented with a simple flow control mechanism to obtain a provably efficient dynamic routing algorithm on a similar network with bounded buffer. The crucial step in [9] is a general theorem showing that any communication scheme (a routing algorithm and a network) that satisfies a given set of conditions, defined only with respect to a *nite history* is stable up to a certain inter-arrival rate. The theorem also bounds the expected routing time under these conditions.

This technique was applied in [9] to a number of dynamic routing scenarios on the butterfly network. In particular we gave the first provable routing protocol for the butterfly network with bounded buffer that is stable for injection rate that is within a constant factor of the hardware bandwidth. Similar results are obtained in [11] for dynamic routing on the mesh. Another result in [9] shows that our general technique can be applied to the *adversarial* input model of [7]. In that model, instead of probabilistic assumptions on the input there is an absolute bound on the number of packets generated in any time interval and must traverse any particular edge. Our technique gives a dynamic randomized routing algorithm for a butterfly with bounded buffer that is optimal (up to constant factors) in that model [9].

4 Dynamic Virtual Circuit Allocation

Communication protocols for high-speed high bandwidth networks (such as the ATM protocol) are based on *virtual circuit switching.* The speed of the network does not allow for on-line routing of individual packets. Instead, upon establishing a connection, bandwidth is allocated along a path connecting the two endpoints for the duration of the connection. This "virtual circuits" are set up on a per-call basis and are disconnected when the call is terminated. Efficient utilization of the network depends on the allocation of virtual circuits between pairs of nodes so that no link is overloaded beyond its capacity.

As in other routing problems we distinguish between a static and a dynamic version. In the static version all the requests are given at once and must be simultaneously satisfied. In practice network are rarely used in the "batch mode" modeled by the static problem. Real-life network performance is better modeled by a dynamic process whereby requests for connection are continuously arriving at the nodes of the network. A connection has a duration time, and once the communication has terminated its bandwidth can be used for another connection. A dynamic circuit switching problem is thus characterized by a number of parameters: the network topology, the channels physical bandwidth, the injection rate distribution of new requests and the distribution of the duration of connections.

Using a random walk approach we develop in [10] a simple and fully distributed protocol for dynamic path selection on bounded degree expander graphs. For the analysis we adopt the stochastic model assumed in the design of most

long-distance telephone networks [24]. Requests arrives according to a Poisson process, and the duration of a connection is exponentially distributed. Our solution achieves an almost optimal utilization of the edges under this stochastic model.

Out approach differs from the work on admission control [3,23,4] in that we do not reject requests. All requests are eventually satisfied in our model, but not immediately. In contrast, in the admission control model a request is either immediately satisfied or it is rejected. Our approach better models most types of computer communication, while the admission control approach is a better model for human (telephone) communication.

5 Stochastic Contention Resolution with Short Delays

One of the few processes that has been studied in the past in dynamic settings are contention resolution protocols and in particular the backoff based solutions.

The contention resolution problem is best defined in terms of multiple access channels. There are *n senders* and *m receivers*. at each of a series of time steps, one or more senders may generate a *packet*. A packet when generated has a *destination* — a unique receiver to which it must be delivered. Any sender may attempt to send a packet to any receiver at any step, but a receiver may only receive one packet in a step. If a receiver is sent more than one packet in a step (a *collision*), all packets sent to that receiver are lost and the senders notified of the loss. The senders must then try to send these packets again at a future step. There is no explicit communication between the senders for coordination the transmissions; the only information that senders have is the packet(s) they have waiting for transmission, and the history of losses. A packet can only be transmitted directly from its sender to its receiver; intermediate hops are disallowed. The case $m = 1$ is a classical instance of sharing a common resource such as a bus or an Ethernet channel (the shared bus is modeled by the single "receiver"). The case with $m = n$ has been studied recently in the static setting under the name *Optically Connected Parallel Computer*, or OCPC [31].

Of primary interest, in the dynamic setting, is whether a protocol can sustain packet arrivals at some rate without *instability.* For stable protocols, two quantities are of interest: (1) the *maximum arrival rate* that can be sustained stably (measured as a fraction of the hardware capacity), and (2) the *delay*, defined to be the maximum over all senders of the expected number of steps from the generation of a packet to its delivery, in the steady state. Delay is of particular importance in high-speed communications applications such as video and ATN networks.

Most previous analysis focused on backoff protocols. The binary exponential backoff protocol used in the Ethernet was proposed by Metcalfe and Boggs [32]. Aldous [1] showed that for any positive constant , binary exponential backoff is unstable if the number of senders is infinite. Kelly [25] showed that any polynomial backoff protocol is unstable for infinitely many senders. Håstad, Leighton and Rogoff [19] studied systems with a finite number of senders. they showed that

binary exponential backoff is unstable for slightly larger than 0:567 even for a system with a finite number of senders, and that polynomial backoff protocol is stable for any < 1 and for any finite number of senders.

A major drawback of all the backoff protocols is the long expected delay in delivering packets (at least linear in the number of senders) as was shown in [19]. In [36] we present the first contention resolution protocol with $o(n)$ expected delay, our protocol is stable for a constant injection rate and the expected delay is logarithmic in the number of senders. Two recent works [35,16] have build on our results to obtain constant expected delay.

References

1. D. Aldous. Ultimate instability of exponential back-off protocol for acknowledgment based transmission control of random access communication channels. *IEEE Transactions on Information Theory,* Vol. IT-33, 1987, pp. 219–223.
2. M. Andrew, B. Awerbuch, A Fernandez, J. Kleinberg, T. Leighton, and Z. Liu. Universal stability results for greedy contention-resolution protocols. *Proceedings of the 37th Annual Synp. on Foundations of Computer Science*, pages 380–389, 1997.
3. B. Awerbuch, Y. Azar, and S. Plotkin. Throughput competitive online routing. In *Proceedings of the 34th IEEE Conference on Foundations of Computer Science*, pages 32–40, 1993.
4. B. Awerbuch, R. Gawlick, T. Leighton, and Y. Rabani. On-line admission control and circuit routing for high-performance computing and communication. *Proceedings of the 35th Annual Synp. on Foundations of Computer Science*, October 1994, pp 412–423.
5. Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 593–602, 1994.
6. Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992.
7. A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan and D. P. Williamson Adversarial queuing theory. *Proceedings of the 28th Annual ACM Symposium on Theory of Computing,* pp. 376–385, 1996.
8. A. Z. Broder and E. Upfal. Dynamic Deflection Routing in Arrays. *Proceedings of the 28th Annual ACM Symposium on Theory of Computing,* pp. 348–355, 1996.
9. A.Z. Broder, A.M. Frieze, and E. Upfal. "A general approach to dynamic packet routing with bounded buffers." *Proceedings of the 37th IEEE Symp. on Foundations of Computer Science.* Burlington, 1996, pp. 390–399.
10. A.Z. Broder, A.M. Frieze, and E. Upfal. "Static and dynamic path selection on expander graphs: a random walk approach". *Proceedings of the 29th ACM Symp. on Theory of Computing.* El Paso, 1997.
11. A. Broder, A. Frieze, and E. Upfal. "Dynamic packet routing on arrays with bounded buffers". *Third Latin American Symposium on Theoretical Informatics - LATIN '98* Campinas, Brazil. April 1998. In *Springer-Verlag Lecture Notes in Computer Science 1380*, pp 273–281, 1998.
12. R.L. Cruz. A calculus for network delay, part I: Network elements in isolation. *IEEE Trans. on Information Theory*, Vol. 37, pages 114–131, 1991

13. R.L. Cruz. A calculus for network delay, part II: Network analysis in isolation. *IEEE Trans. on Information Theory*, Vol. 37, pages 132–141, 1991
14. A. Czumaj and V. Stemann. "Randomized Allocation Processes". Preprint, 1997.
15. M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *Proceedings of the 29th IEEE Conference on Foundations of Computer Science*, pages 524–531, 1988.
16. L.A. Goldberg and P.D. MacKenzie. Contention Resolution with Guaranteed Constant Expected Delay. Preprint, 1997.
17. J. Goodman, A.G. Greenberg, N. Madras, and P. March. Stability of Binary Exponential Backoff. *J. of the ACM*, Vol. 35 (1988) pages 579–602.
18. A.G. Greenberg, P. Flajolet, and R.E. Ladner. Estimating the Multiplicities of Conflicts to Speed Their Resolution in Multiple Access Channels. *J. of the ACM*, Vol. 34, 1987, pp. 289–325.
19. J. Håstad, T. Leighton, and B. Rogoff. Analysis of backoff protocols for multiple access channels. *Proceedings of the 19th ACM Symp. on Theory of Computing,* 1987, pp. 241–253.
20. M. Harcol-Balter and P. Black. Queuing analysis of oblivious packet routing networks. *Procs. of the 5th Annual ACM-SIAM Symp. on Discrete Algorithms*. Pages 583–592, 1994.
21. M. Harcol-Balter and D. Wolf. Bounding delays in packet-routing networks. *Procs. of the 27th Annual ACM Symp. on Theory of Computing,* 1995, pp. 248–257.
22. R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 318–326, 1992.
23. A. Kamath, O. Palmon, and S. Plotkin. Routing and Admission Control in General Topology Networks with Poisson Arrivals. *SODA 96*.
24. F.P. Kelley. Blocking probabilities in large circuit-switching networks. *Adv. Appl. Prob.*, 18:573–505, 1986.
25. F.P. Kelly. Stochastic models of computer communication systems. *J. Royal Statistical Soc. B,* Vol. 47, 1985, pp. 379–395.
26. L. Kleinrock. *Queueing systems*. Wiley, New York, 1975.
27. N. Kahale and T. Leighton. Greedy dynamic routing on arrays. *Procs. of the 6th Annual ACM-SIAM Symp. on Discrete Algorithms*. Pages 558–566, 1995.
28. T. Leighton. Average case analysis of greedy routing algorithms on arrays. *Procs. of the Second Annual ACM Symp. on Parallel Algorithms and Architectures.* Pages 2–10, 1990.
29. F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*. Morgan-Kaufmann, San Mateo, CA 1992.
30. F.T. Leighton and S. Rao. Circuit switching: a multi-commodity flow based approach. *Proc. of 9th International Parallel Processing Symposium*, 1995.
31. P.D. MacKenzie, C.G. Plaxton, and R. Rajaraman. On contention resolution protocols and associated probabilistic phenomena. *Proceedings of the 26th ACM Symposium on Theory of Computing,* 1994, pp. 153–162.
32. R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Communication of the ACM*, Vol. 19, 1976, pp. 395–404.
33. M. Mitzenmacher. Bounds on the greedy algorithms for array networks. *Procs. of the 6th Annual ACM Symp. on Parallel Algorithms and Architectures.* Pages 346–353, 1994.

34. M. Mitzenmacher. Load balancing and density dependent jump Markov processes. *Procs. of the 37th IEEE Annual Symp. on Foundations of Computer Science*, pages 213–222, October 1996.
35. M. Paterson and A. Srinivasan. Contention resolution with bounded delay. *Proc. of the 34th Annual IEEE Symp. on Foundation of Computer Science*, pages 104–113, 1995.
36. P. Raghavan and E. Upfal. St9chastic contention resolution with short delays. *Proc. of 24th ACM Symp. on Theory of Computing*, pages 229–237, 1995.
37. C. Scheideler and B. Voecking Universal continuous routing strategies. *Procs. of the 8th Annual ACM Symp. on Parallel Algorithms and Architectures.* 1996.
38. G. D. Stamoulis and J. N. Tsitsiklis. The efficiency of greedy routing in hypercubes and butterflies. *Procs. of the 6th Annual ACM Symp. on Parallel Algorithms and Architectures.* Pages 346–353, 1994.

Car-Pooling as a Data Structuring Device: The Soft Heap*

Bernard Chazelle

Department of Computer Science
Princeton University
Princeton, NJ 08544, USA
chazelle@cs.princeton.edu

Abstract. A simple variant of a priority queue, called a *soft heap*, is introduced. The data structure supports the usual operations: insert, delete, meld, and findmin. In order to beat the standard information-theoretic bounds, the soft heap allows errors: occasionally, the keys of certain items are artificially raised. Given any $0 < \varepsilon < 1/2$ and any mixed sequence of n operations, the soft heap ensures that at most εn keys are raised at any time. The amortized complexity of each operation is constant, except for insert, which takes $O(\log 1/\varepsilon)$ time. The soft heap is optimal. Also, being purely pointer-based, no arrays are used and no numeric assumptions are made on the keys. The novelty of the data structure is that items are moved together in groups, in a data-structuring equivalent of "car pooling." The main application of the data structure is a faster deterministic algorithm for minimum spanning trees.

1 Introduction

We design a simple variant of a priority queue, called a *soft heap*. The data structure stores items with keys from a totally ordered universe, and supports the operations:

- `create`(S): Create a new, empty soft heap.
- `insert`(S, x): Add new item x to S.
- `meld`(S_1, S_2): Form a new soft heap with the items stored in S_1, S_2 (assumed to be disjoint), and destroy S_1 and S_2.
- `delete`(S, x): Remove item x from S.
- `findmin`(S): Return an item in S with minimum key.

The soft heap may, at any time, increase the value of certain keys. Such keys, and by extension, the corresponding items, are called *corrupted*. Corruption is entirely at the discretion of the data structure and the user has no control over it. Naturally, `findmin` returns the minimum *current* key, which might be corrupted. The benefit is speed: during heap updates, items travel together in packets in a form of "car pooling."

* This work was supported in part by NSF Grant CCR-93-01254, NSF Grant CCR-96-23768, ARO Grant DAAH04-96-1-0181, and NEC Research Institute.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 35–42, 1998.
© Springer-Verlag Berlin Heidelberg 1998

Theorem 1. *Fix any parameter $0 < \varepsilon < 1/2$, and beginning with no prior data, consider a mixed sequence of operations that includes n inserts. There is a soft heap such that the amortized complexity of each operation is constant, except for insert, which takes $O(\log 1/\varepsilon)$ time. At most εn items are corrupted at any given time. In a comparison-based model, these bounds are optimal.*

Note that this does not mean that only εn items are corrupted in total. Because of deletes, in fact, most items could end up corrupted. If we set $\varepsilon < 1/n$, then the soft heap behaves like a regular heap with logarithmic insertion time. The soft heap is purely pointer-based: no arrays are used, and no numeric assumptions on the keys are required. In this regard, it is fundamentally different from previous work on approximate data structures, eg, [5]. The soft heap was designed with a specific application in mind, minimum spanning trees: it is the main vehicle for an algorithm [1] that computes a minimum spanning tree of a connected graph in time $O(m \log \alpha)$, where $\alpha = \alpha(m, n)$ is a functional inverse of Ackermann's function and n (resp. m) is the number of vertices (resp. edges).

A variant of the soft heap allows the corruption to be *independent* of the number of insertions. The proof is quite complicated and involves a lengthy amortized analysis omitted from this extended abstract.

Theorem 2. *Fix any parameter $0 < \varepsilon < 1/2$, and beginning with no prior data, consider a mixed sequence of operations that includes f findmins and d deletes. There is a soft heap such that the amortized complexity of each operation is constant, except for insert, which takes $O(1/\varepsilon^2)$ time. At most $\varepsilon(f + d)$ items are corrupted at any given time.*

2 The Data Structure

Recall that a binomial queue [6] of rank k is a rooted tree of 2^k nodes: it is formed by the combination of two binomial queues of rank $k-1$, where the root of one becomes the new child of the other root. Just as a Fibonacci heap [3] is a collection of binomial queues, a soft heap is a sequence of modified binomial queues, called *soft queues*. The modifications come in two ways.

- A soft queue h is a binomial queue with subtrees possibly missing. The binomial queue from which it is derived is called the *master* queue of h. The *rank* of a node of h is its number of children in the master queue. It is an upper bound on the number of children in h.
- A node v may store several items, in fact, a whole *item-list*, denoted L_v. The *node key* of v, denoted $\text{key}(v)$, indicates the value of all the current keys of the items in L_v: it is an upper bound on the original keys. We fix a parameter $r = r(\varepsilon)$, and we require that all corrupted items be stored at nodes of rank greater than r *and* in item-lists of size at least two.

The heap[1] consists of a doubly-linked list $s_1, \ldots, s_m$: each cell s_i has two extra pointers: one to the root r_i of a distinct queue, and another to the root of

[1] For brevity we drop the "soft."

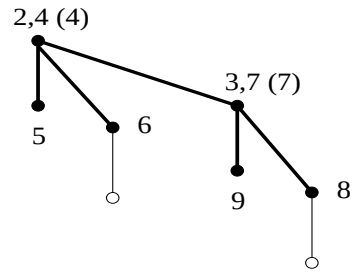


Fig. 1. Two binomial queues of rank 2 combine to make one binomial queue of rank 3. The soft queue is missing the two light edges. Corrupted item-lists are both of size two; node keys are in parentheses.

minimum key among all r_j's ($j \quad i$). The latter is called the *pre x-min* pointer of s_i. We require that rank(r_1) < < rank(r_m). By extension, the rank of a queue (resp. heap) refers to the rank of its root (resp. r_m).

3 The Heap Operations

To create a new, empty heap requires no work. To insert a new item, we create an uncorrupted one-node queue, and we meld it into the heap (see below). We delete an item "lazily" by marking it. Finally, the two remaining operations work like this:

{ `meld` (S_1; S_2): We dismantle the heap of lesser rank, say S_2, by melding each of its queues into S_1. To meld a queue of rank k into S_1, we look for the smallest index i such that rank(r_i) $\quad k$. If there is no such i, we add a new cell past the last one in S_1, with a pointer to the queue's root. If rank(r_i) > k, we insert the cell right before s_i, instead. Otherwise, we meld the two queues into one of rank $k+1$, by making the root with the larger key a new child of the other root. If rank(r_{i+1}) = $k+1$, a new conflict arises. We repeat the process as long as necessary like carry propagation in binary addition. Finally, we update the prefix-min pointers between s_1 and the last cell visited. When melding not a single queue but a whole heap, the last step can be done at the very end in just one pass through S_1.

{ `findmin`: The prefix-min pointer at s_1 leads to the smallest current key in the heap. The trouble is, all the items at that node r_j might be marked *deleted*. In that case, we must discard this item-list and refill it with other "live" items. To do that, we call the procedure succ(h), where h is the queue rooted at r_j, and follow suit with another `findmin`.

```
                        succ(h)

    L_root(h)  T  ;;
    if h has no child
        then set key(root(h)) to 1 and return;
1.  succ(A);
    if key(root(A)) > key(root(B))
        then exchange A and B, and clean up;
    T  T [ L_root(A);
    if loop-condition holds then goto 1;
    L_root(h)  T.
```

The procedure succ moves items up the queue by grouping them together to achieve the "car pooling" effect mentioned earlier. The price to pay for increased efficiency is corruption. In the pseudo-code below, B denotes the subtree rooted at the highest-ranking child of root(h), while A refers to h deprived of B and the edge leading to it; note that rank(B) might be much smaller than rank(h). The "loop-condition" statement is what makes soft heaps special. Without it, succ would be indistinguishable from the standard delete-min operation of a binomial queue.

Here is a line-by-line explanation. The item-list at root(h) contains only deleted items, so it is emptied out. The (local) variable T is initialized for the car-pooling about to take place. If h has no child, then having just lost the item-list at its root, we assign the root an infinite key in anticipation of its imminent removal. Otherwise, we recurse within A, treating it as a queue of same rank as B. This provides root(A) with a brand-new item-list (possibly empty) and a new node key (possibly 1). If the heap ordering is now violated at root(h), we swap A and B, ie, we move B to become rooted at root(h) and finally we exchange the names A and B to go back to the original notation. Next, we clean up: if the key of root(B) is now infinite, we discard B from h and from the heap. Note that although root(h) loses a child, its rank remains unchanged. We append the new item-list of A to T.

Next, we identify the new A and B; recall that B is the subtree rooted at the highest-ranking child of root(h). If no cleanup was necessary then rank(B) remains unchanged, else it drops. In the extreme case, h no longer has a child after the cleanup: then the new B is not defined. The most interesting instruction comes next. The loop-condition holds if it is being tested for the first time in the call to succ(h) (ie, branching is only binary), if the new B is well defined, and if

$$0 \quad \text{rank}(A) - r < 2^{\left\lfloor \frac{\text{rank}(h) - r}{2} \right\rfloor};$$

These inequalities refer to the old A. If the loop-condition holds, then the next time we append $L(A)$ to T, all items in T will be corrupted as they adopt the

new key of root(A). Note that this is the only spot where corruption takes place. Finally, if h is a bona-fide queue, and not a sub-queue called recursively, we must also clean up the heap. If the key of root(h) is infinite, we discard h. We also update all the prefix-min pointers between s_1 and the cell pointing to h. Recall that even if the root of h has lost children its rank does not decrease, and so h stays in place within the soft heap.

4 Fighting Corruption

To bound the number of corrupted items in the soft heap, we prove that

$$jL_v j \quad \max f\, 1\,;\, 2^{b(\mathrm{rank}(v)-r)=2c}\, g: \tag{1}$$

Until the first call to `succ`, all item-lists have size one, and the inequality holds. Afterwards, simple inspection shows that all operations have either no effect on (1) or sometimes a favorable one (eg, meld). All of them, that is, except for `succ`, which could potentially cause a violation. We show that this is not the case. If `succ`(h) calls `succ`(A) only once, the item-list of A migrates to a higher-ranking node by itself and by induction (1) is satisfied. Otherwise, the item-list at root(h) becomes $L_{\mathrm{root}(A)}\ [\ L_{\mathrm{root}(A^0)}$, and $0 \quad ' - r < 2b(k - r)=2c$, where $' = \mathrm{rank}(A) \quad \mathrm{rank}(A^0)$ and $k = \mathrm{rank}(h)$. By induction, the new item-list at h is, therefore, of size at most $2 \quad 2^{b(' - r)=2c} \quad 2^{b(k-r)=2c}$, which proves (1).

Lemma 1. *At any time during a mixed sequence of operations that includes n inserts, the soft heap contains at most $n=2^{r-3}$ corrupted items.*

Proof. We begin with a simple observation. If N is the node set of a binomial queue of rank k then (trivial proof by induction),

$$\sum_{v2N} 2^{\mathrm{rank}(v)=2} \quad 2^{k+2} - 3 \quad 2^{k=2}: \tag{2}$$

Now, recall that the ranks of the nodes of a queue h are derived from the corresponding nodes in its master queue h^0. So, the set R (resp. R^0) of nodes of rank at least r in h (resp. h^0) is such that $jRj \quad jR^0j$. Within h^0, the nodes of R^0 number a fraction at most $1=2^{r-1}$ of all the leaves. Some of these leaves may not appear in h, but they once existed in the queue, with each one holding a distinct item. Summing over all master queues, we find that

$$\sum_{h^0} jR^0j \quad n=2^{r-1}: \tag{3}$$

There is no corrupted item at any rank $\quad r$, and so by (1) their total number does not exceed

$$\sum_{h^0} \sum_{v2R^0} 2^{(\mathrm{rank}(v)-r)=2}: \tag{4}$$

Each R^0 forms a binomial queue by itself, where the rank of node v becomes $\text{rank}(v) - r$. Since a binomial queue of rank k has 2^k nodes, then by (2, 3), the sum in (4) at most $\sum_{h^0} 4|R^0| \quad n/2^{r-3}$.

5 The Running Time

Only `meld` and `succ` need to be looked at, all other operations being trivially constant-time. Assigning one credit per queue takes care of the carry-like ripples during a `meld`. Indeed, two queues of the same rank combine into one, which releases one credit to pay for the work. Updating prefix-min pointers can take time, however. Specifically, ripples aside, the cost of melding S_1 and S_2 is proportional to the smaller rank of the two. The entire sequence of melds can be modeled as a binary tree. A leaf z denotes a heap (its cost is 1). An internal node z indicates the melding of two heaps. Since heaps can grow only through melds, the size of the resulting heap at z is proportional to the number $N(z)$ of descendents. The cost of node z (ie, of the meld) is $1 + \log \min\{N(x); N(y)\}$, where x and y are the left and right children of z.[2] It is a staple of algorithm analysis [4] that adding together all these costs gives a total melding cost linear in the size of the tree.

Finally, we show that the cost of all calls to `succ` is $O(rn)$. Consider an execution of `succ`(h). If root(h) has at least two children, then after the first call `succ`(A), no cleanup is necessary and the new B is well defined. By looking at the computation tree, it follows immediately that, excluding the updating of prefix-min pointers, the running time is $O(rC)$, where C is the number of times the loop-condition succeeds. Each time the loop-condition is satisfied, both calls `succ`(A) bring finite node keys to the root and two nonempty item-lists are merged into T. There can be at most $n - 1$ such merges, therefore $C \quad n$ and our claim holds.

We ignored the cost of updating prefix-min pointers after each call to `succ`. To keep it in $O(n)$, we must make a few minor modifications to the algorithm. We maintain the following invariant on the root of any soft queue: *its number of children should be at least half its rank.* This makes the cost of prefix-min updating negligible. Indeed, each update takes $O(\text{rank}(h))$ time, which is now dominated by the cost of `succ`(h) itself.

Only `succ`, by removing nodes, can upset the new invariant. To restore it for h is easy: we meld back every sub-queue of h whose root is a child of root(h). If the root of h is not corrupted, then we meld it back, too. Otherwise, we set its node key to 1 and we store its item-list separately in a *reservoir*. For example, we can add to the list $s_1; \ldots; s_m$ a cell s_0 pointing to the reservoir. Of course, after all the melding is done, we check for the invariant again (which might still be violated because of nodes becoming roots) and we repeat as long as necessary. Note that melding two soft heaps is done as before; in addition, the two reservoirs are linked together in constant time.

[2] We use the fact that the rank is exactly the logarithm of the number of nodes in the master queue.

Our previous analysis of melds shows that their cost is linear in their number. How many new melds do we create now? Consider the missing children of root(h): in the master queue, at least one of these children is of rank at least $\lfloor \mathrm{rank}(h)/2 \rfloor$. The leaves at or below that child have lost their items: this cannot happen to them again, so they can be charged for the melding costs. Their number is at least to $2^{\mathrm{rank}(h)/2-2}$, which is more than enough to account for the (at most) $\mathrm{rank}(h)/2$ new melds.

By (1) the number of corrupted items migrating into the reservoir after dismantling h is at most $2^{(\mathrm{rank}(h)-r)/2}$, which is at most a fraction $2^{2-r/2}$ the number of leaves of the previous paragraph. It follows from Lemma 1 that the total number of corrupted items is bounded (conservatively) by $n/2^{r/2-2} + n/2^{r-1}$. Setting $r = 5 + 2\lceil \log(1/\varepsilon) \rceil$ proves Theorem 1 (except for the optimality claim).

Remark: The storage is linear in the number of insertions n, but not necessarily in the actual number of items present. If this is a problem, here is a simple fix: as soon as the number of live items falls below a fixed fraction of n, we dismantle the data structure entirely, raise all corrupted keys to infinity, and move their items into the reservoir. A charging scheme similar to the one above easily accounts for the resulting increase in corruption. This modified soft heap is optimal in storage, and as shown below, in time.

6 Optimality

To complete the proof of Theorem 1, we show that the soft heap is optimal. Consider a sequence of n inserts (with distinct keys), followed by n pairs of the form: findmin, delete returned item. Assume that at most εn items are corrupted at any given time; n is taken large enough, and without loss of generality, $1/\varepsilon$ lies between a large constant and $\sqrt{n}$. We also assume that each item knows at which time it is corrupted (if at all). We show that in linear time enough information can be collected that would take superlinear time to gather from scratch. Divide the sequence of n pairs into time intervals $T_1, \ldots, T_l$, each consisting of $\lceil 2\varepsilon n \rceil$ pairs; without loss of generality, we may assume that $n = \lceil 2\varepsilon n \rceil l$. Let S_i be the set of items deleted during T_i, and let U_i be the subset of S_i that includes only items that were at some point uncorrupted during T_i. Finally, let x_i be the smallest original key among the items of U_i, and let s_i be the corresponding item; for convenience, we put $x_0 = -\infty$ and $x_{l+1} = \infty$. Given an item $s \in S_i$ whose original key lies in $[x_j, x_{j+1})$, we define $\delta(s) = |i - j|$. Some simple facts:

(1) $|U_i| \geq \lceil 2\varepsilon n \rceil - \varepsilon n \geq \varepsilon n$: Because at most εn items are corrupted at the beginning of T_i.

(2) The x_i's appear in increasing order, and the original key of any item in U_i lies in $[x_i, x_{i+1})$: Since s_{i+1} is uncorrupted during T_i, its original key x_{i+1} is at least the current (and hence, the original) key of any item deleted during T_i.

(3) $\sum_s \delta(s) \mid s \in S_i \cap U_i \} < 2n$: Given $s \in S_i \cap U_i$, let $[x_j, x_{j+1})$ be the interval containing the original key of s. As we just observed, the original key of s

is less than x_{i+1}, and therefore, $j \quad i$. To avoid being selected by findmin, the item s must have been in a corrupted state during the deletion of x_k, for any $j < k < i$ (if any such k exists). The total number of corrupted items during the deletions of $x_1; : : : ; x_l$ is at most $"nl$, and therefore so is the sum of distances $i - j - 1 = \sum_{(s)} (s) - 1$ over all such items s. It follows that $(s) \quad "nl + n < 2n$, hence our claim.

(4) The number of items whose original keys fall in $[x_i; x_{i+1})$ is less than $6"n$: Suppose that the item does not belong to $S_i \, [\, S_{i+1}$. It cannot be selected by findmin and deleted before the beginning of T_i, since s_i was not corrupted yet. By the time s_{i+1} was deleted, the item in question must have been corrupted (it cannot have been deleted yet). So, there can be at most $"n$ such items. Thus, the total number of items with original keys in $[x_i; x_{i+1})$ is at most $2d2"ne + "n$.

Next, for each item $s \, 2 \, S_i \, n \, U_i$, we search which interval $[x_j; x_{j+1})$ contains its original key, which we can do in $O(\ (s) + 1)$ time by sequential search. By (1–4), this means that in $O(n)$ postprocessing time we have partitioned the set of original keys into disjoint intervals, each containing between $"n$ and $6"n$ keys. Next, in $O(l)$ time, we locate which interval contains the original key of rank $d2"nek$, for $1 \quad k \quad l$. Finally, by using linear selection-finding, we find the original key of that rank, all of which takes $O(n)$ time. Let $n_1 = \quad = n_l = d2"ne$; a standard counting argument shows that any comparison tree for computing these percentiles has height at least

$$\log \binom{n}{n_1; : : : ; n_l} = \Omega(n \log 1="):$$

References

1. Chazelle, B. *A faster deterministic algorithm for minimum spanning trees*, Proc. 38th Ann. IEEE Symp. Found. Comp. Sci. (1997), 22–31.
2. Chazelle, B. *A deterministic algorithm for minimum spanning trees*, manuscript, 1998.
3. Fredman, M.L., Tarjan, R.E. *Fibonacci heaps and their uses in improved network optimization algorithms*, J. ACM 34 (1987), 596–615.
4. Hoffman, K., Mehlhorn, K., Rosenstiehl, P., Tarjan, R.E. *Sorting Jordan sequences in linear time using level-linked search trees*, Inform. and Control 68 (1986), 170–184.
5. Matias, Y., Vitter, J.S., Young, N. *Approximate Data Structures with Applications*, Proc. 5th Ann. SIAM/ACM Symp. Disc. Alg. (SODA '94), 1994.
6. Vuillemin, J. *A data structure for manipulating priority queues*, Commun. ACM 21 (1978), 309–315.

Optimal Pre x-Free Codes for Unequal Letter Costs: Dynamic Programming with the Monge Property*

Phil Bradford[1], Mordecai J. Golin[2], Lawrence L. Larmore[3], and Wojciech Rytter[4]

[1] Max-Planck-Institut für Informatik, 66123 Saarbruecken, Germany
[2] Hong Kong UST, Clear Water Bay, Kowloon, Hong Kong. golin@cs.ust.hk
[3] Department of Computer Science, University of Nevada, Las Vegas, NV 89154-4019. larmore@cs.unlv.edu
[4] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02–097 Warszawa, Poland, and Department of Computer Science, University of Liverpool. rytter@csc.liv.ac.uk

Abstract. In this paper we discuss a variation of the classical *Huffman coding problem*: finding optimal prefix-free codes for unequal letter costs. Our problem consists of finding a minimal cost prefix-free code in which the encoding alphabet consists of unequal cost (length) letters, with lengths and . The most efficient algorithm known previously required $O(n^{2+\max(\ ,\ )})$ time to construct such a minimal-cost set of n codewords. In this paper we provide an $O(n^{\max(\ ,\ )})$ time algorithm. Our improvement comes from the use of a more sophisticated modeling of the problem combined with the observation that the problem possesses a "Monge property" and that the SMAWK algorithm on monotone matrices can therefore be applied.

1 Introduction

The problem of finding optimal prefix-free codes for unequal letter costs (and the associated problem of constructing optimal lopsided trees) is an old and hard classical one. The problem consists of finding a minimal cost prefix-free code in which the encoding alphabet consists of unequal cost (length) letters, of lengths and , *:* The code is represented by a *lopsided tree*, in the same way as a Huffman tree represents the solution of the Huffman coding problem. Despite the similarity, the case of unequal letter costs is much harder then the classical Huffman problem; no polynomial time algorithm is known for general letter costs, despite a rich literature on the problem, *e.g.*, [1,7]. However there are known polynomial time algorithms when and are integer constants [7].

The problem of finding the minimum cost tree in this case was first studied by Karp [9] in 1961 who solved the problem by reduction to integer linear programming, yielding an algorithm exponential in both n and *:* Since that time

* The work of the second author was partially supported by Hong Kong RGC CERG grant 652/95E, that of the third author by NSF grant CCR-9503441

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 43{54, 1998.
c Springer-Verlag Berlin Heidelberg 1998

there has been much work on various aspects of the problem such as; bounding the cost of the optimal tree, Altenkamp and Mehlhorn [2], Kapoor and Reingold [8] and Savari [15]; the restriction to the special case when all of the weights are equal, Cot [5], Perl Gary and Even [14], and Choi and Golin [4]; and approximating the optimal solution, Gilbert [6]. Despite all of these efforts it is still, surprisingly, not even known whether the basic problem is polynomial-time solvable.

The only technique other than Karp's for solving the problem is due to Golin and Rote [7] who describe an $O(n^{+2})$-time dynamic programming algorithm that constructs the tree in a top-down fashion. This is the the most efficient known algorithm for the case of small ; in this paper we apply a different approach by constructing the tree in a bottom-up way and describing more sophisticated attacks on the problem. The first attack permits reducing the search space in which optimal trees are searched for. The second shows how, surprisingly, monotone-matrix concepts, e.g., the *Monge property* [13] and the SMAWK algorithm [3] can be utilized.

Combining these two attacks improves the running time of of [7] by a factor of $O(n^2)$ down to $O(n$):

Our approach requires a better understanding of the combinatorics of lopsided trees; to achieve this we also introduce the new crucial concept of *characteristic sequences*.

Let 0 : A tree T is a *binary lopsided ; tree* (or just a *lopsided tree*) if every non-leaf node u of the tree has two sons, the length of the edge connecting u to its left son is , and the length of the edge connecting u to its right son is : Figure 1 shows a 2-5 lopsided tree. Let T be a lopsided tree and $v \in T$ some node. Then

$$\begin{aligned} \mathit{depth}(T; v) &= \text{ sum of the lengths of the edges connecting } \mathit{root}(T) \text{ to } v \\ \mathit{depth}(T) &= \max\{\mathit{depth}(T; v) : v \in T\} \end{aligned}$$

For example, the tree in Figure 1 has depth 20. Now suppose we are given a sequence of nonnegative weights $P = \{p_1; p_2; \ldots; p_n\}$. Let T be a lopsided tree with n leaves labeled $v_1; v_2; \ldots; v_n$: The weighted external path length of the tree is

$$\mathit{cost}(T; P) = \sum_i p_i \quad \mathit{depth}(T; v_i):$$

Given P; the problem that we wish to solve is to construct a labeled tree T that minimizes $\mathit{cost}(T; P)$:

As was pointed out quite early [9] this problem is equivalent to finding a minimal cost prefix-free code in which the encoding alphabet consists of two (or generally, more) unequal cost (length) letters, of lengths and . Also note that if = = 1 then the problem reduces directly to the standard Huffman-encoding problem.

Notice that, given any particular tree T, the cost actually depends upon the labeling of the leaves of T; the cost being minimized when $p_1 \quad p_2 \quad p_n$ and $\mathit{depth}(T; v_1) \quad \mathit{depth}(T; v_2) \quad \mathit{depth}(T; v_n)$: We therefore will always

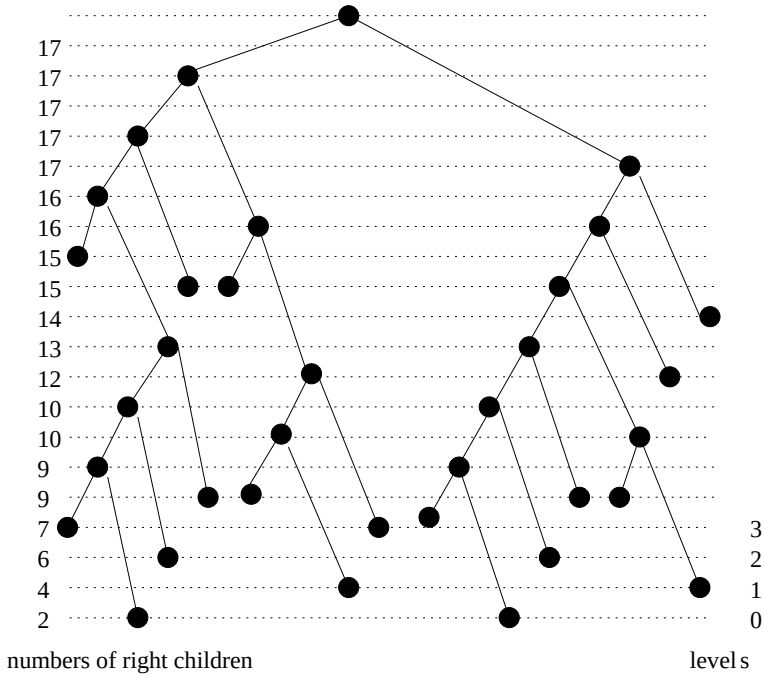


Fig. 1. An example 2-5 tree T. The characteristic sequence $B = sequence(T)$ is (2,4,6,7,9,9,10,10,12,13,14,15,16,16,17,17,17,17,17).

assume that the leaves of T are labeled in nonincreasing order of their depth. We will also assume that $p_1 \quad p_2 \quad \ldots \quad p_n$.

Note: In this extended abstract we omit many technical proofs.

2 Combinatorics of Lopsided Trees and Monotonic Sequences

The first crucial concept in this paper is the *characteristic sequence* of a tree T. Denoted by $sequence(T)$ this is the vector $B_T = (b_0; b_1; \ldots; b_{d-1})$ in which b_i is the number of right children on or below level i for $0 \quad i < d$, where d is the height of the tree. and the levels are enumerated from bottom to top (See Figure 1).

Let n and P be fixed. Now let $B = b_0; b_1; \ldots; b_{d-1}$ be any sequence, not necessarily one of the form $B = B_T$ defined by some tree T: B is said to be *monotonic* if d and

$$0 \quad b_0 \quad b_1 \quad b_2 \quad \ldots \quad b_{d-1}:$$

Note that the number of right children on or below level i of tree T can not decrease with i so for all trees T, B_T is a monotonic sequence.

A monotonic sequence B of length d *terminates* in a -tuple $(y\ ;\ y_{-1}; :::; y_1)$ if $8j;\ 0 \quad j < \ ;\ b_{d-j} = y_j$: Note that if T is a lopsided tree with n leaves then T must have $n-1$ internal nodes and thus $n-1$ right children. Furthermore the top levels of T can not contain any right children. Thus if $B = sequence(T)$ for some T then B terminates in a tuple $(n-1; n-1; :::; n-1)$:

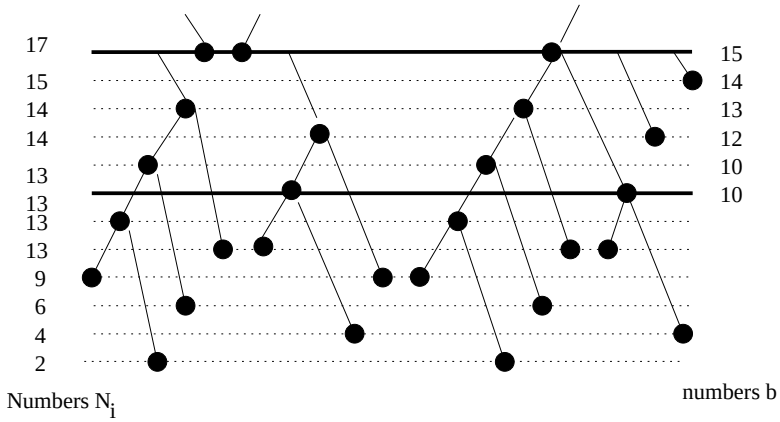


Fig. 2. The bottom forest F_{11} of the tree T from Figure 1.

For a monotonic sequence $B = b_0; b_1; :::; b_{d-1}$ define

$$N_k(B) = b_k + b_{k-(\ -\ )} - b_{k-\ };\qquad S_i = \sum_{j\ \ i} p_j;\qquad cost(B;P) = \sum_{0\ \ k<d} S_{N_k(B)} \tag{1}$$

If $i < 0$ or $i > n$ then $S_i = 1$: For a tree T, denote by $F_k = forest_k(T)$ the forest resulting by taking all nodes at level k and below (See Figure 2). Denote by $N_k(T)$ the number of leaves in $forest_k(T)$. (Note that we have overloaded the notation $N_k(\ )$ to apply to both trees and sequences.)

The following lemma collects some basic facts:

Lemma 1 *Let T be a lopsided tree and $B = sequence(T)$: Then*
(P1) $cost(T;P) = \sum_{0\ \ k<depth(T)} S_{N_k(T)}$,
(P2) $8\ 0 \quad i < d = depth(T);\quad N_i(T) = b_i + b_{i-(\ -\ )} - b_{i-}$
(where $8\ j < 0$; we set $b_j = 0$).
(P3) $cost(T;P) = cost(B;P)$,

Proof.
We omit the proof of (P1) which is straightforward but tedious. To prove (P2),

note that F_i is a forest, hence

$$N_i(T) = \{u \in F_i : u \text{ is a leaf in } F_i\} \quad (2)$$

$$= \text{Number of internal nodes in } F_i + \text{Number of trees in } F_i \quad (3)$$

The first summand in the last line is easily calculated. A node at height k is internal in F_i if and only if it is the father of some right son at level $k-1$: Thus

$$\text{Number of internal nodes in } F_i = b_{i-1}: \quad (4)$$

The second summand is only slightly more complicated to calculate. The number of trees in F_i is exactly the same as the number of *tree-roots* in F_i: Now note that a node in F_i is a tree-root in F_i if and only if its father is not in F_i: Thus a right son at height k in F_i is a tree-root if and only if $i-1 < k \le i$ and there are exactly $b_i - b_{i-1}$ such nodes.

Similarly a left son at height k is a tree-root if and only if $i-1 < k \le i$: This may occur if and only if the left son's right brother is at height k, where $i-1 < k \le i-(1-)$: The number of such nodes is therefore $b_{i-(1-)} - b_{i-1}$:

We have therefore just seen that

$$\text{Number of trees in } F_i = (b_i - b_{i-1}) + (b_{i-(1-)} - b_{i-1}): \quad (5)$$

Combining (4) and (5) completes the proof of (P2). (P3) follows from (P1) and (P2).

Now define a sequence B to be *legal* if B is monotonic and $B = \text{sequence}(T)$ for some lopsided tree T: The lemma implies that minimizing cost over all legal sequences is exactly the same as minimizing cost over all lopsided trees.

However, not all sequences are legal so this knowledge does not at first seem to help us. In the next section we sketch a proof of the following fact. Given any minimum-cost *monotonic* sequence that terminates in the -tuple $(n-1; n-1; \ldots; n-1)$ it is possible to build a legal sequence with the same cost. Since all legal sequences are monotonic this legal sequence must be a minimal-cost legal sequence and thus correspond to a minimum-cost tree. In other words, to find a minimal-cost tree it will suffice to find a minimum-cost monotonic sequence terminating in $(n-1; n-1; \ldots; n-1)$:

3 Relation Between Minimum Sequences and Optimal Trees

We start by assuming that $B = \text{sequence}(T)$ for some T: In T the weight p_1 is associated with some lowest leaf at level 0. The left sibling of this leaf is associated with some other weight p_k: How can such a k be identified?

Observe that this sibling can be the lowest leaf in the tree which is a left-son, i.e., the lowest left node in T: Such a node appears on level $-$ (see the left tree in Figure 3). The number of leaves below this level is b_{-1}, so assuming that

we list items consecutively with respect to increasing levels, the lowest left-son leaf has index $k = \mathit{FirstLeft}(B)$, where

$$\mathit{FirstLeft}(B) = b \; - \; _{-1} + 1$$

We state without proof the intuitive fact that if T is an optimal tree in which $p_1; p_k$ label sibling leaves, then the tree T^0 that results by (i) removing those leaves and (ii) labeling their parent (now a leaf) with $p_1 + p_k$ will also be an optimal tree for the leaf set $P^0 = P \; [\; fp_1 + p_k g - fp_1; p_k g$ (see the right tree in Figure 3), Also, calculation shows that

$$\mathit{cost}(T; P) = \mathit{cost}(T^0; P^0) + \quad p_1 + \quad p_k: \tag{6}$$

The rest of this section will be devoted to translating this intuition into facts about trees and sequences.

If $p_1; \; p_k$ are siblings in a tree T then denote by $T^0 = \mathit{merge}(T; 1; k)$ the tree in which leaves $p_1; \; p_k$ are removed and their parent is replaced by a leaf with weight $p_1 + p_k$ (see Figure 3). We also write $\mathit{unmerge}(T^0; 1; k) = T$. Thus

$$\mathit{cost}(\mathit{unmerge}(T^0; 1; k); P) = \mathit{cost}(T^0; P^0) + \quad p_1 + \quad p_k: \tag{7}$$

For the sequence $B = (b_0; b_1; : : : b_d)$ denote

$$\mathit{dec}(B) = B^0 = (b_0 - 1; b_1 - 1; b_2 - 1; : : : b_d - 1):$$

Note that (after any leading zeros are deleted) this sequence is the characteristic sequence of $T^0 = \mathit{merge}(T; 1; k)$:

Assume Γ is a sorted sequence of positive integers, x is a positive integer, and $\mathit{insert}(\Gamma; x)$ is the sequence Γ with x inserted and sorted (as in insertion sort). Now denote by $\mathit{delete}(P; p_1; p_k)$ the sequence P with elements p_1 and p_k deleted, and define

$$P^0 = \mathit{package\_merge}(P; 1; k) = \mathit{insert}(\mathit{delete}(P; p_1; p_k); p_1 + p_k):$$

For example if $P = f2; 3; 4; 5; 10g$ then

$$\begin{aligned} P^0 = \mathit{delete}(P; 2; 4) &= f3; 5; 10g \\ \mathit{insert}(P^0; 6) &= f3; 5; 6; 10g \\ \mathit{package\_merge}(P; 1; 3) &= f3; 5; 6; 10g \end{aligned}$$

After appropriate manipulations (deleted in this abstract) we derive the following essential fact:

Lemma 1 ((key-lemma)).
Let $k = \mathit{FirstLeft}(B)$, $P^0 = \mathit{package\_merge}(P; 1; k)$ *and* $B^0 = \mathit{dec}(B)$. *Then*

$$\mathit{cost}(B^0; P^0) \quad \mathit{cost}(B; P) - \quad p_1 - \quad p_k:$$

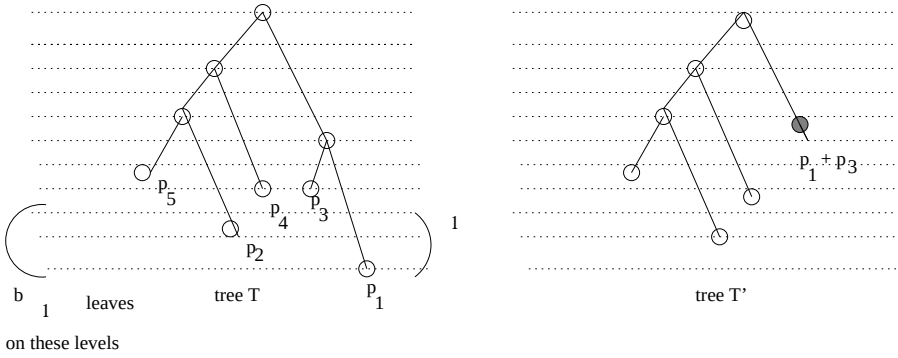


Fig. 3. The correspondence between trees T, T^0 and their sequences: $T^0 = merge(T; 1; 3)$ and $sequence(T) = B = (1; 2; 2; 3; 3; 4; 4; 4; 4)$ $sequence(T^0) = dec(B) = B^0 = (0; 1; 1; 2; 2; 3; 3; 3; 3; 3)$; $FirstLeft(B) = b_{-_{-1}} + 1 = b_{5-2-1} + 1 = 3$ and $cost(T) = cost(T^0) + 2p_4 + 5p_1$.

This lemma permits us to prove that minimum-cost monotonic sequences have the same cost as minimum cost trees and permit the construction of such trees:

Theorem 1 ((correctness)).
Assume B is a minimum cost monotonic sequence terminating in $(n - 1; n - 1; : : : ; n - 1)$ for the sequence P: Then there is a tree T such that:
(1) $cost(T; P) = cost(B; P)$.
Furthermore if $n > 2$ then
(2) $FirstLeft(B)$ *is the index of the left brother of* p_1 *in* T,
(3) $B^0 = dec(B)$ *is a minimum cost sequence for* $P^0 = package\_merge(P; 1; k)$.

Proof.
The proof is by induction with respect to the number n of items in P: If $n = 2$ then all legal sequences have the form

$$b_0 = b_1 = \quad = b_{d-1} = 1$$

where d : The sequence with $d =$ has minimum cost and this sequence is also the minimum-cost monotonic sequence.

So now suppose that $n > 2$: Let $B^0 = dec(B)$ and T^0 be a minimum cost tree for P^0. P^0 has $n - 1$ items, so by the induction hypothesis $cost(T^0; P^0)$ equals the minimum cost of a monotonic sequence for P^0. In particular, by Lemma 1, we have

$$cost(T^0; P^0) \quad cost(B^0; P^0) \quad cost(B; P) - \quad p_k - \quad p_1: \tag{8}$$

Take $T = unmerge(T^0; 1; k)$. Then by Equality (6) and Inequality (8) we have:

$$cost(sequence(T); P) = cost(T; P) = cost(T^0; P^0) + \quad p_k + \quad p_1 \quad cost(B; P):$$

B was chosen to be a minimal cost sequence so all of the inequalities must be equalities and, in particular, we find that $cost(T;P) = cost(B;P)$. Hence T is the required tree, and this completes the proof of (1).

We also find that

$$cost(T^0;P^0) + \quad p_k + \quad p_1 = cost(B;P)$$

so plugging back into (8) we find that $cost(T^0;P^0) = cost(B^0;P^0)$: Since T^0 is a minimal cost tree for P^0 the induction hypothesis implies B^0 is a minimum cost sequence for P^0 proving (3). The proof of (2) follows from the details of the construction.

Note that this theorem immediately implies that, given a minimum-cost sequence B for P; we can construct a minimum-cost tree for P: If $n = 2$ the tree is simply one root with two children. If $n > 2$ calculate $B^0 = dec(B)$ and $P^0 = package\_merge(P;1;k)$ in $O(n)$ time. Recursively build the optimal tree T^0 for P^0 and then replace its leaf labelled by $p_1 + p_k$ by an internal node whose left child is labelled by p_k and whose right child is labelled by p_1: This will be the optimal tree. Unwinding the recursion we find that the algorithm uses $O(n^2)$ time (but this can easily be improved down to $O(n \log n)$ with a careful use of data structures).

4 The Monge Property and the Algorithm

We now introduce the weighted directed graph G whose vertices are monotonic -tuples of nonnegative integers in the range $[0 :: n-1]$. There is an edge between two vertices if and only they "overlap" in a (-1)-tuple, precisely defined below.

Suppose $i_0 \quad i_1 \quad i_2 \quad ::: i_{-1} \quad i$. Let $u = (i_0;i_1;i_2;:::;i_{-1})$ and $v = (i_1;i_2;:::;i_{-1};i)$. Then there is an edge from u to v if $u \neq v$, and furthermore, the weight of that edge is

$$Weight(u;v) = EdgeCost(i_0;i_1;:::;i) = S_{i+i-i_0}$$

Observe that if $(u;v)$ is an edge in G then the monotonicity of $(i_0;i_1;i_2;:::;i_{-1};i)$ guarantees that u is lexicographically smaller as a tuple than v: In other words the lexicographic ordering on the nodes is a topological ordering of the nodes of G; the existence of such a topological ordering implies that G is acyclic. Note that the -tuple of zeros, $(0;:::0)$, is a source. We refer to this node as the *initial node* of the graph. Note also that the -tuple $(n-1;:::;n-1)$ is a sink; we refer to it as the *final node* of the graph.

For any vertex u in the graph, define $cost(u)$ to be the weight of a shortest (that is, least weight) path from the initial node to u.

Suppose we follow a path from the source to the sink and, after traversing an edge $(u;v)$, output i , the final element of v: The sequence thus outputted is obviously a monotonic sequence terminating in the -tuple $(n-1;n-1;:::;n-1)$ and from the definition of $Weight(u;v)$ the cost of the path is exactly the cost

of the sequence. Similarly any monotonic sequence terminating in the -tuple $(n-1; n-1; \ldots; n-1)$ corresponds to a unique path from source to sink in G.

In particular, given a tree T and $B = \mathit{sequence}(T)$ Lemma 1 implies that the cost of the path corresponding to B is exactly the same as $\mathit{cost}(T)$.

Example.
The tree T from Figure 3 has $B = \mathit{sequence}(T) = (1;2;2;3;3;4;4;4;4)$ and its corresponding path in the graph G

$$(0;0;0;0;0) \xrightarrow{S_1} (0;0;0;0;1) \xrightarrow{S_2} (0;0;0;1;2)$$
$$\xrightarrow{S_2} (0;0;1;2;2) \xrightarrow{S_4} (0;1;2;2;3) \xrightarrow{S_5} (1;2;2;3;3) \xrightarrow{S_5} (2;2;3;3;4)$$
$$\xrightarrow{S_5} (2;3;3;4;4) \xrightarrow{S_5} (3;3;4;4;4) \xrightarrow{S_5} (3;4;4;4;4) \xrightarrow{S_5} (4;4;4;4;4)$$

The cost of this path and also of the tree T is

$$S_1 + 2\ S_2 + S_4 + 6\ S_5$$

The above observations can be restated as

Observation 2 *Assume T is a tree and $B = \mathit{sequence}(T)$. Then $\mathit{cost}(T) = \mathit{cost}(B)$ equals the cost of the path in G corresponding to B.*

The correctness theorem and the algorithm following it can can now be reformulated as follows:

Theorem 2. *The cost of a shortest path from the initial node to the final node is the same as the cost of a minimum cost tree. Furthermore given a minimum cost path a minimum-cost tree can be reconstructed from it in $O(n^2)$ time.*

Observe that G is acyclic and has $O(n^{+1})$ edges. The standard dynamic-programming shortest path algorithm would therefore find a shortest path from the source to the sink, and hence a min-cost tree, in $O(n^{+1})$ time. We now discuss how to find such a path in $O(n)$ time. Our algorithm obviously cannot construct the entire graph since it is too large. Instead we use the fact that, looked at in the right way, our problem possesses a Monge property.

A 2-dimensional matrix A is defined to be a *Monge matrix* [13] if for all $i; j$ in range

$$A(i;j) + A(i+1;j+1) \quad A(i;j+1) + A(i+1;j) \tag{9}$$

Now let $= (i_1; i_2; \ldots; i_{-1})$ be any monotonic (-1)-tuple of integers. For $0 \quad i \quad i_1$ and $i_{-1} \quad j \quad n-1$, define

$$\mathit{EdgeCost}\ (i;j) = \mathit{EdgeCost}(i; i_1; \ldots; i_{-1}; j) = S_{j+i\ -i}$$

$$A\ (i;j) = \mathit{cost}(i; i_1; \ldots; i_{-1}) + \mathit{EdgeCost}\ (i;j).$$

The important observation is that

Theorem 3 ((Monge-property theorem)).
For xed , the matrix A is a two-dimensional Monge matrix.

Proof.
Let $= (i_1; i_2; : : : ; i\ _{-1})$. We prove Equation (9), where $A = A\ $. If the right hand side of Equation (9) is infinite, we are done. Otherwise, by the definitions of the S_k, and of $A\ $, canceling terms when possible, we have

$$A\ (i; j+1) + A\ (i+1; j) - A\ (i; j) - A\ (i+1; j+1) = p_{j+i\ -i+1} - p_{j+i\ -i} \quad 0$$

which completes the proof.

A 2 2 matrix A is defined to be *monotone* if either A_{11} A_{12} or A_{21} A_{22}. An n m matrix A is defined to be *totally monotone* if every 2 2 submatrix of A is monotone. The SMAWK algorithm [3] takes as its input a function which computes the entries of an n m totally monotone matrix A and gives as output a non-decreasing function f, where 1 $f(i)$ m for 1 i n, such that $A_{i;f(i)}$ is the minimum value of row i of A. The time complexity of the SMAWK algorithm is $O(n+m)$, provided that each computation of an A_{ij} takes constant time. Note that every Monge matrix is totally monotone so our matrices A are totally monotone. This fact permits us to prove:

Theorem 4 ((Shortest-path theorem)).
A shortest path from a source node to the sink node in G can be constructed in $O(n\ )$ time.

Proof.
The case where $= 1$ requires an exceptional proof, because the proof below fails if the sequence is a 0-tuple. However, that case is already proved in [11]. Thus, we assume 2.

In this extended abstract we actually only show how to calculate the cost of the shortest path. Transforming this calculation into the *construction* of the actual path uses standard dynamic programming backtracking techniques that we will leave to the reader.

Our approach is actually to calculate the values of $cost(u)$ for *all* monotonic -tuples $u = (i_0; i_1; : : : ; i\ _{-1})$: In particular, this will calculate the value of $cost(n - 1; n - 1; : : : ; n - 1)$ which is what is really required.

For fixed $= (i_1; i_2; : : : ; i\ _{-1})$ note that

$$8j \quad i\ _{-1};\ cost(\ ; j) = \min\{A\ (i; j)\ :\ i \quad i_1\}$$

Also note that $A\ (i; j)$ can be calculated in constant time provided the values of $cost(i;\ )$ is known. This means that, given a fixed ; if the values of $cost(i;\ )$ are already known for all i then the values of $cost(\ ; j)$ for *all* j can be calculated in total time $O(n)$ using the SMAWK algorithm. We call this $O(n)$ step *processing* .

Our algorithm to calculate $cost(i_0; i_1; : : : ; i\ _{-1})$ for all -tuples is simply to process all of the ($- 1$) tuples in lexicographic order. Processing in this order ensures that at the time we process the values of $cost(i;\ )$ are already known for all i.

Using the SMAWK algorithm it thus takes $O(n)$ time to process each of the $O(n\ ^{-1})$ ($- 1$)-tuples so the entire algorithm uses $O(n\ )$ time as stated.

Algorithm *Optimal_Tree_Construction*
  sequence construction phase:
    compute a shortest path from source to sink in G;
    let B be the sequence corresponding to ;
  tree reconstruction phase:
    construct optimal tree T from B using
    recursive algorithm described following the
    Correctness Theorem
end of algorithm.

Theorem 5 ((main result)).
We can construct a minimum cost lopsided tree in $O(n\ )$ time.

Proof.
If $= 1$ use the basic Huffman encoding algorithm which runs in $O(n)$ time. Otherwise apply the algorithm *Optimal_Tree_Construction*. Theorem 4 tells us that B can be computed in $O(n\ )$ time.

The algorithm described following the Correctness Theorem for constructing an optimal tree from B runs in $O(n^2) = O(n\ )$ time completing the proof of the theorem.

We conclude by pointing out, without proof, that the algorithm *Optimal_Tree_Construction* can be straightforwardly extended in two different directions:

Theorem 6.
We can construct a minimum cost lopsided tree in $O(n \log^2 n)$ time with $O(n^{\ -1})$ processors of a PRAM.

Theorem 7 ((height limited trees)).
We can construct a minimum cost lopsided tree with height limited by L in $O(n \quad L)$ time.

(A tree with height limited by L is one in which no node has depth greater than L.)

References

1. Julia Abrahams, "Code and Parse Trees for Lossless Source Encoding," *Sequences'97,* (1997).
2. Doris Altenkamp and Kurt Mehlhorn, "Codes: Unequal Probabilities, Unequal Letter Costs," *J. Assoc. Comput. Mach.* 27 (3) (July 1980), 412–427.
3. A. Aggarwal, M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric applications of a matrix-searching algorithm, *Algorithmica* 2 (1987), pp. 195–208.

4. Siu-Ngan Choi and M. Golin, "Lopsided trees: Algorithms, Analyses and Applications," *Automata, Languages and Programming*, Proceedings of the 23rd International Colloquium on Automata, Languages, and Programming (ICALP 96)*:*
5. N. Cot, "A linear-time ordering procedure with applications to variable length encoding," *Proc. 8th Annual Princeton Conference on Information Sciences and Systems*, (1974), pp. 460–463.
6. E. N. Gilbert, "Coding with Digits of Unequal Costs," *IEEE Trans. Inform. Theory,* **41** (1995).
7. M. Golin and G. Rote, "A Dynamic Programming Algorithm for Constructing Optimal Prefix-Free Codes for Unequal Letter Costs," *Proceedings of the 22nd International Colloquium on Automata Languages and Programming (ICALP '95),* (July 1995) 256-267. Expanded version to appear in *IEEE Trans. Inform. Theory.*
8. Sanjiv Kapoor and Edward Reingold, "Optimum Lopsided Binary Trees," *Journal of the Association for Computing Machinery* **36** (3) (July 1989), 573–590.
9. R. M. Karp, "Minimum-Redundancy Coding for the Discrete Noiseless Channel," *IRE Transactions on Information Theory,* 7 (1961) 27-39.
10. Abraham Lempel, Shimon Even, and Martin Cohen, "An Algorithm for Optimal Prefix Parsing of a Noiseless and Memoryless Channel," *IEEE Transactions on Information Theory*, **IT-19**(2) (March 1973), 208–214.
11. L. L. Larmore, T. Przytycka, W. Rytter, Parallel computation of optimal alphabetic trees, SPAA93.
12. K. Mehlhorn, "An Efficient Algorithm for Constructing Optimal Prefix Codes," *IEEE Trans. Inform. Theory ,* **IT-26** (1980) 513-517.
13. G. Monge, *Déblai et remblai*, Mémoires de l' Académie des Sciences, Paris, (1781) pp. 666-704.
14. Y. Perl, M. R. Garey, and S. Even, "Efficient generation of optimal prefix code: Equiprobable words using unequal cost letters," *Journal of the Association for Computing Machinery* **22** (2) (April 1975), 202–214,
15. Serap A. Savari, "Some Notes on Varn Coding," *IEEE Transactions on Information Theory*, **40** (1) (Jan. 1994), 181–186.
16. Robert Sedgewick, *Algorithms,* Addison-Wesley, Reading, Mass.. (1984).

Finding All the Best Swaps of a Minimum Diameter Spanning Tree Under Transient Edge Failures*

Enrico Nardelli[1,2], Guido Proietti[1,3], and Peter Widmayer[4]

[1] Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila
Via Vetoio, 67010 L'Aquila, Italy
nardelli,proietti@univaq.it.

[2] Ist. di Analisi dei Sistemi e Informatica, CNR, V.le Manzoni 30, 00185 Roma, Italy

[3] On leave to Computer Science Dept., Carnegie Mellon University, 15213 Pittsburgh, PA, supported by the CNR under the fellowship N.215.29

[4] Institut für Theoretische Informatik, ETH Zentrum, 8092 Zürich, Switzerland
widmayer@inf.ethz.ch.

The work of this author was partially supported by grant "Combinatorics and Geometr" of the Swiss National Science Foundation.

Abstract. In network communication systems, frequently messages are routed along a minimum diameter spanning tree (MDST) of the network, to minimize the maximum delay in delivering a message. When a transient edge failure occurs, it is important to choose a temporary replacement edge which minimizes the diameter of the new spanning tree. Such an optimal replacement is called the *best swap*. As a natural extension, the *all-best-swaps (ABS) problem* is the problem of finding the best swap for every edge of the MDST. Given a weighted graph $G = (V; E)$, where $jV j = n$ and $jEj = m$, we solve the ABS problem in $O(n\sqrt{m})$ time and $O(m + n)$ space, thus improving previous bounds for $m = o(n^2)$.

1 Introduction

For communication networks, it is important to remain operational even if individual network components fail. In the past few years, the ability of a network to survive a transient failure (its *survivability*) has been studied intensely (an excellent survey paper on survivable networks is [5]). From the practical side, this has largely been a consequence of the replacement of metal wire meshes by fiber optic networks: Their extremely high bandwidth makes it economically attractive to make networks as sparse as possible. In the extreme, a network might be designed as a spanning tree of some underlying graph of all possible links. A sparse network, however, is less likely to survive a transient edge (or node) failure than a mesh with a multitude of connections that can be used as

* This research was carried out while the rst two authors visited the third author within the CHOROCHRONOS TMR program of the European Community.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 55{66, 1998.
c Springer-Verlag Berlin Heidelberg 1998

detours. Therefore, it is important for sparse networks to also take survivability into account from the very beginning.

On the theoretical side, this gave rise to an interesting family of problems on graphs. In particular, it is important to know in advance how some specific, significant feature of the spanning tree changes as a consequence of a transient edge failure. For example, if the spanning tree has minimum length (i.e., is a minimum spanning tree (MST)), the most immediate feature of interest is the total length of the tree itself, and we are interested in finding the edge whose removal results in the greatest increase of the length of the MST. This problem has been studied (at least implicitly) for a decade, and the fastest solution known to date is an $O(m + n \log n)$ time algorithm [8], where $|V| = n$ and $|E| = m$. Another meaningful class of problems arises when the spanning tree is a single-source shortest path tree (SPT), which is another popular network architecture. In such a case, the problem of finding an edge in the tree whose removal results in the largest increase of the distance between the source node r and a given node s has been solved efficiently by Malik et al. [9], who gave an $O(m+n \log n)$ time algorithm. Recently, Nardelli et al. [10] defined a different parameter for measuring the vitality of an edge along a shortest path, looking for the edge whose removal will result in the worst detour to reach the destination node, and they solved the problem in $O(m + n \log n)$ time.

However, in several applications, the used spanning tree is neither an MST nor a SPT. Rather, many network architectures look for minimizing the *diameter* of the spanning tree, that is the length of the longest path in the tree between any two nodes, so that the maximum propagation delay in the network will be as low as possible. Hence, a *minimum diameter spanning tree* (*MDST*) is a spanning tree having minimum diameter among all the spanning trees. For an MDST subject to transient edge failures in the way just described, it makes sense to temporarily substitute the failing edge with a single edge which makes the diameter of the new spanning tree as low as possible (for practical motivations, see [7]). Such an optimal replacement is called a *best swap*. As a natural extension, the *all-best-swaps (ABS) problem* is the problem of finding a best swap for every edge of the MDST.

The related problem of maintaining a spanning tree of small diameter in a fully dynamic context, where a tree evolves due to repeated insertions and deletions of edges, has been solved in [7]. This problem belongs to the family of problems that is concerned with updating the solution of a graph problem after dynamic changes of the graph; for a recent survey, consult [2]. The approach in [7] is more general than what is needed for solving the ABS problem. Hence, if we use it for solving the ABS problem, we spend $O(n^2)$ time and $O(m+n)$ space and preprocessing time. We get these bounds by computing a best swap in $O(n)$ time for each deleted edge. Recently, Alstrup et al. [1] improved the runtime for computing a best swap in an incremental context (i.e., when no deletions are allowed) to $O(\log^2 n)$. By using some of the results in [7], their approach can be adapted to solve the ABS problem, and the obtained runtime is $O(n\sqrt{m}\log n)$, with $O(m + n)$ space.

In this paper we solve the ABS problem in $O(n\sqrt{m})$ time and $O(m+n)$ space, thus strictly improving previous bounds for $m = o(n^2)$. This can be done by adapting the well-known *path halving* compression technique [11] for answering in $O(1)$ amortized time (over a total of $(n\sqrt{m})$ queries) the following query: Given a rooted tree T and a pair of nodes y and v in T such that v belongs to the subtree of T rooted at y, what is the length of a longest simple undirected path in T, starting at v and staying within the subtree of T rooted at y?

The paper is organized as follows: in Section 2 we define the problem more precisely and formally. Section 3 proposes the algorithm for solving the problem. Finally, in Section 4, we present the adaptation of the path halving compression technique which allows to efficiently solve the ABS problem.

2 Problem Statement

Let $G = (V; E)$ be a biconnected, undirected graph, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges, with a nonnegative real length $|e|$ associated with each edge $e \in E$. Let n and m denote the number of vertices and the number of edges, respectively. A graph $H = (V'; E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. If $V' = V$ then H is called a *spanning subgraph* of G. A connected acyclic spanning subgraph of G is called a *spanning tree* of G.

A *simple path* (or a *path* for short) in $G = (V; E)$ is a subgraph $(V'; E')$ with $V' = \{v_1; \ldots; v_k\}$ $v_i \neq v_j$ for $i \neq j$ and $E' = \{(v_i; v_{i+1}) \mid 1 \leq i < k\}$, also denoted as $\langle v_1; \ldots; v_k\rangle$. Path $\langle v_1; \ldots; v_k\rangle$ is said to go from v_1 to v_k. Because there is exactly one path between any two nodes in a tree, let us denote in this case the length $|\langle v_1; \ldots; v_k\rangle|$ of the path as $d(v_1; v_k)$.

Let $D(T) = \langle d_1; d_2; \ldots; d_k\rangle$ denote a *diameter path* of T, that is a path whose length $|D(T)|$ is maximum in T. We call $|D(T)|$ the *diameter* of T. For simplicity, we will use the term diameter also for the diameter path, whenever no confusion arises. A *minimum diameter spanning tree* of G is a spanning tree of G having minimum diameter. If $e \in E_T$ is a tree edge, a *replacement edge* (or *swap edge*) for e is an edge $f \in E \setminus E_T$ such that $T_{e;f} = (V; E_T - e + f)$ is a spanning tree of G. Let R_e denote the set of replacement edges for e. A *best swap* for e is an edge $r(e) \in R_e$ such that

$$|D(T_{e;r(e)})| = \min_{f \in R_e} \{|D(T_{e;f})|\}.$$

The *all-best-swaps (ABS) problem* is the problem of finding a best swap for every edge $e \in E_T$.

3 Solving the ABS Problem

To solve the ABS problem efficiently, we will exploit relationships among the original spanning tree T and the replacing ones. Let d_c, with $1 \leq c \leq k$, be the *center* of the diameter path, that is the node in $D(T)$ for which $|d(d_1; d_c) -$

$d(d_c; d_k)|$ is minimum. If this node is not unique, we take the node farther from d_1. Let Φ denote a source directed tree obtained by rooting T in d_c and orienting the edges towards the leaves. Following [7], we maintain an augmented topology tree and an augmented 2-dimensional topology tree [3,4] to efficiently retrieve only $O(\sqrt{m})$ *selected edges* among the $O(m)$ replacement edges, whenever an edge e in T is deleted. In fact, among the selected edges, a best swap is contained.

3.1 The algorithm

The general outline of our algorithm is the following:

`Step 0:` Perform preliminary computations.
`Step 1:` For each edge $e \in \Phi$ as considered by a postorder visit
`Step 2:` Delete e; update the topology and the 2-dimensional topology tree.
`Step 3:` Compute the set of selected replacement edges $S_e \subseteq R_e$.
`Step 4:` For each edge $f \in S_e$, compute $|D(T_{e;f})|$ and select a best swap.
`Step 5:` Insert e and update the topology and the 2-dimensional topology tree.

Step 0 requires $O(n+m)$ time and space, as we show next. Notice that in Step 1 we consider all the $O(n)$ edges of the tree, in the order they are generated by a postorder tree visit (this order is needed for a correct path halving compression, as is shown in Section 4). Steps 2, 3 and 5 can be accomplished in $O(\sqrt{m})$ time, and $|S_e| = O(\sqrt{m})$ [7]. Concerning Step 4, we will show that $|D(T_{e;f})|$ can be computed in $O(1)$ amortized time. This can be done using the *path halving compression* technique that will be presented and analyzed in detail in Section 4. This technique, given a rooted tree Φ and a pair of nodes u and v in Φ such that u belongs to the subtree of Φ rooted at v, allows to find in $O(\log_{(1+Q/n)} n)$ amortized time per query (over a total of Q queries) the length $Findpath(u; v)$ of a longest simple undirected path starting from u and staying within the subtree rooted at v. Since for solving the ABS problem we will prove that $Q = (n\sqrt{m})$, $Findpath(u; v)$ can be computed in $O(1)$ amortized time. Thus, Step 4 costs $O(\sqrt{m})$ amortized time, and the global time for solving the ABS problem is $O(n\sqrt{m})$, using $O(n+m)$ space.

3.2 Preliminary computations

Let $L_D = \langle d_1; \ldots; d_{c-1}; d_c \rangle$ and $R_D = \langle d_c; d_{c+1}; \ldots; d_k \rangle$. W.l.o.g., let us assume $|L_D| \geq |R_D|$. We mark in $O(n)$ time all the nodes in Φ rooted at d_{c-1}, say N_l, and all the nodes rooted at d_{c+1}, say N_r, with the nearest node on the diameter from which they descend (if a node is on the diameter, then it is marked with itself). N_c will denote the remaining nodes, marked with d_c. Figure 1 depicts this notation.

Then, we associate with each node $v \in \Phi$ its distance from d_c, and the lengths $h_i(v); i = 1; 2;$ of the two longest directed paths in Φ emanating from v and making use of two different subtrees of v, if they exist. For each of these

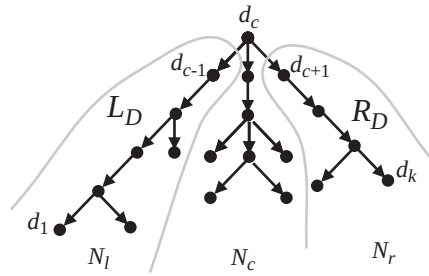


Fig. 1. The oriented minimum spanning tree

paths we also store the nodes $a_i(v); i = 1; 2;$ adjacent to v. Let $[h_1(v); a_1(v)]$ and $[h_2(v); a_2(v)]$ be these pairs of values, with $h_1(v)$ $h_2(v)$. With the root, we also associate a further value, say $h_3(d_c)$, corresponding to the length of a longest path in Φ starting from d_c and not using d_{c-1} and d_{c+1}. After, we associate with each node d_j 2 L_D (d_j 2 R_D) a further value, say (d_j), containing the length of a longest path in Φ starting from d_c and containing neither d_{c+1} (d_{c-1}) nor the edge $(d_j; d_{j-1})$ $((d_j; d_{j+1}))$. Note that since d_c belongs to both L_D and R_D, (d_c) coincides with $h_3(d_c)$. We express (d_j) recursively as follows:

$$(d_c) = h_3(d_c)$$

$$(d_j) = \max(\ (d_{j+1}); d(d_c; d_j) + h_2(d_j)) \qquad \text{for } j = c - 1; : : : ; 1$$

$$(d_j) = \max(\ (d_{j-1}); d(d_c; d_j) + h_2(d_j)) \qquad \text{for } j = c + 1; : : : ; k:$$

Next, we associate with each node d_j 2 L_D (d_j 2 R_D) a further value, say (d_j), containing the length of a longest path in T starting from d_1 (d_k) and staying within the subtree of Φ rooted at d_j. We express (d_j) recursively as follows:

$$(d_1) = \ (d_k) = 0$$

$$(d_j) = \max(\ (d_{j-1}); d(d_1; d_j) + h_2(d_j)) \qquad \text{for } j = 2; : : : ; c - 1$$

$$(d_j) = \max(\ (d_{j+1}); d(d_k; d_j) + h_2(d_j)) \qquad \text{for } j = k - 1; : : : ; c + 1:$$

We also associate with each node on the diameter a further value, say (d_j), containing the nearest node along the diameter of the path stored in (d_j) (this can be done during the computation of (d_j)). It is easy to see that all the above computations cost $O(n)$ time.

Finally, we convert G into a graph G^0 with maximum vertex degree 3 [6], and we derive from T a spanning tree T^0 of G^0 (see [7] for further details). Then, we associate to T^0 a topology tree and an augmented 2-dimensional topology tree [3,4,7], which can be initialized in $O(m)$ time and space. It is easy to see that an edge swap in T corresponds to an edge swap in T^0, and vice versa. Therefore, in the following we will continue to refer to the original spanning tree T, even

though our algorithm makes use of the topology tree and the 2-dimensional topology tree of T^0.

Summarizing, preliminary computations have an overall cost of $O(m+n)$ time and use $O(m+n)$ space.

3.3 Computing $|D(T_{e,f})|$ in $O(1)$ amortized time

In the rest of the paper, two paths will be considered *adjacent* if they share the root d_c only. When the edge $e=(x,y)$ is removed, T is split into two subtrees, say T_x and T_y, which will be later connected by means of a replacement edge $f=(u,v)$. As a consequence

$$|D(T_{e,f})| = \max\{|D(T_x)|, |D(T_y)|, |P_f|\} \tag{1}$$

where $|P_f|$ is the length of a longest path in $T_{e,f}$ passing through f. We now analyze different cases that can arise in solving the ABS problem. For the sake of clarity, we perform a different analysis depending on whether the removed edge is located on the diameter or not.

3.3.1 The removed edge is not on the diameter

Assume the edge $e=(x,y)$ is removed, where x is closer to d_c than y and $e \notin D(T)$. In this case, neither L_D nor R_D are affected. Trivially, $D(T_x)=D(T)$. Moreover, $|D(T_y)| \leq |D(T)|$, since a diameter in T_y is a diameter in T too. It then remains to compute $|P_f|$, for any selected replacement edge $f \in S_e$. Let $f=(u,v)$, where $u \in T_x$ and $v \in T_y$. It is clear that

$$|P_f| = |L_u| + |f| + |L_v|$$

where L_u is a longest path in T_x starting from u and L_v is a longest path in T_y starting from v. Since v is a descendant of y in T, by using the path halving compression technique $|L_v| = \mathit{Findpath}(v,y)$ can be computed in $O(1)$ amortized time, while $|f|$ is clearly available in $O(1)$ time. It remains to compute $|L_u|$. The following claim is easy to prove:

Lemma 1. *At least one of the longest paths in T_x starting from u contains d_c.*

Proof. Suppose, for the sake of contradiction, that none of the longest paths in T_x starting from u contains d_c. Let us restrict our attention to any one of such longest paths, say P_u. We will show that such a path can be modified into another path at least as long as P_u and passing through d_c, from which the claim will follow. Let w be the node in P_u nearest to d_c, and let z be the ending node of P_u other than u. Three cases are possible:

1. $w \in N_l$: let $q \in L_D$ be the node on $D(T)$ nearest to w (if w is on the diameter, then $q = w$). It is trivial to see that in this case, being $q \neq d_c$ since $w \in N_l$, it must be

$$d(q,z) \geq d(q,d_k)$$

since otherwise $d(d_1;q)+d(q;z) > d(d_1;q)+d(q;d_k) = jD(T)j$. Being $d(q;z) = d(q;w) + d(w;z)$, it then follows that P_u can be modified into the path $P_u{}^0 = hu;:::;w;:::;q;:::;d_k i$ containing d_c and such that

$$jP_u{}^0 j = d(u;w) + d(w;q) + d(q;d_k) \quad d(u;w) + d(q;d_k)$$

$$d(u;w) + d(q;z) \quad d(u;w) + d(w;z) = jP_u j$$

that is a contradiction.

2. $w \in N_r$: this case is symmetric to the first one.
3. $w \in N_c$: in this case, it must be clearly $d(w;z) \quad d(w;d_c) + d(d_c;d_1)$, since otherwise $d(d_c;d_1)+d(d_c;w)+d(w;z) > jD(T)j$. It then follows that P_u can be modified into the path $P_u{}^0 = hu;:::;w;:::;d_c;:::;d_1 i$, containing d_c and such that

$$jP_u{}^0 j = d(u;w) + d(w;d_c) + d(d_c;d_1) \quad d(u;w) + d(w;z) = jP_u j$$

that is a contradiction. □

¿From the above analysis and from the fact that L_D is one of the longest paths emanating from d_c in T and that R_D is one of the longest paths emanating from d_c in T which does not make use of d_{c-1} it follows that

$$jL_u j = \begin{array}{l} d(d_c;u) + jR_D j \text{ if } u \in N_l \\ d(d_c;u) + jL_D j \text{ otherwise} \end{array}$$

and therefore, it follows that $jL_u j$ is available in $O(1)$ time. Summarizing, $jP_f j$ can be computed in $O(1)$ amortized time, and

$$jD(T_{e;f})j = \max(jD(T)j; jP_f j):$$

Once this value is computed for the $O(\sqrt{m})$ selected edges identified by the augmented topology and 2-dimensional topology tree, a best replacement edge is available. Therefore, the case $e \in D(T)$ can be managed in $O(\sqrt{m})$ amortized time.

3.3.2 The removed edge is on the diameter

We will analyze the case in which $e = (d_i;d_{i-1}) \in L_D$ is removed, since the case $e \in R_D$ is symmetric. When e is removed, T is split into two subtrees, say T_{d_i} and $T_{d_{i-1}}$, which will be later connected by means of a replacement edge $f = (u;v) \in S_e$. Equation (1) becomes

$$jD(T_{e;f})j = \max(jD(T_{d_i})j; jD(T_{d_{i-1}})j; jP_f j):$$

Let us now analyze the value of these three terms.

$jD(T_{d_i})j$: We start by proving the following fact:

Lemma 2. *At least one of the diameters of T_{d_i} contains d_k 2 R_D.*

Proof. In fact, for the sake of contradiction, suppose that none of the diameters of T_{d_i} contains d_k. Let us restrict our attention to any one of such diameters, say P. We will show that such diameter can be modified into a path containing d_k and at least as long as P, from which the claim will follow. Let w be the node in P nearest to d_c. Let P_1 and P_2 be the two subpaths in which P splits with respect to w, with ending nodes z_1 and z_2, respectively, and suppose that z_1 precedes z_2 in a preorder traversal of T. Three cases are possible:

1. w 2 N_l: this case is similar to the case 1 of the proof of Lemma 1. In fact, the modified path there built also contains d_k, apart from d_c.
2. w 2 N_r: in this case, let q 2 R_D be the node on $D(T)$ nearest to w (if w is on the diameter, then q w). Clearly, any path emanating from q in T is no longer than $d(q; d_k)$. In particular

$$d(q; d_k) \quad d(q; z_1) \quad d(w; z_1)$$

and

$$d(q; d_k) \quad d(q; z_2) \quad d(w; z_2):$$

Since either P_1 [$\langle w; \ldots; q\rangle$ or P_2 [$\langle w; \ldots; q\rangle$ (or both of them) is adjacent to $\langle q; \ldots; d_k\rangle$, it follows that P can be modified into a no shorter path containing d_k, that is a contradiction.

3. w 2 N_c: in this case, since z_2 descends from d_c in T, it must be clearly $d(w; z_2)$ $d(w; d_c)+d(d_c; d_k)$, since otherwise $d(d_c; d_1)+d(d_c; w)+d(w; z) > |D(T)|$. It then follows that P can be modified into the path $P^0 = \langle z_1; \ldots; d_c; \ldots; d_k\rangle$ containing d_k and such that

$$|P^0| = d(z_1; w) + d(w; d_c) + d(d_c; d_k) \quad d(z_1; w) + d(w; z_2) = |P|$$

that is a contradiction. □

¿From the above result, it follows that a longest path starting from d_k and not using the edge e just removed can be computed $O(1)$ time as

$$|D(T_{d_i})| = \max(\ (d_{c+1}); \ (d_i) + |R_D|):$$

$|D(T_{d_{i-1}})|$: analogously to the previous case, it can be proved that at least one of the diameters of $T_{d_{i-1}}$ must contain the node d_1. Therefore, it will be

$$|D(T_{d_{i-1}})| = \ (d_{i-1})$$

which can be computed in $O(1)$ time.

$|P_f|$: let be $f = (u; v)$, where u 2 T_{d_i} and v 2 $T_{d_{i-1}}$, and $|P_f| = |L_u|+|f|+|L_v|$. $|L_v| = Findpath(v; d_{i-1})$ can be computed in $O(1)$ amortized time and $|f|$ is available in $O(1)$ time. It remains to analyze $|L_u|$. Remember that to the node u is associated the nearest node q on the diameter from which it descends. The following three situations are possible for u:

1. $u \in N_l$: in this case, still using the same arguments as for the point 1 of the proof of Lemma 1, it can be easily proved that at least one of the longest paths in T_{d_i} starting from u must contain d_c, and then $|L_u|$ can be obtained in $O(1)$ time as

$$|L_u| = d(d_c, u) + |R_D|.$$

2. $u \in N_r$: In this case, still using the same arguments as for the point 2 of the proof of Lemma 2, it can be proved that at least one of the longest paths in T_{d_i} starting from u must contain q. Therefore, it follows that $|L_u|$ can be obtained in $O(1)$ time as (note that the following is equivalent to compute $Findpath(u, d_c)$)

$$|L_u| = \max \quad d(u,q) + d\ q,\ (d_{c+1}) \quad + \quad (d_{c+1}) - d\ d_k,\ (d_{c+1}) \ ;$$

$$d(u, d_k), d(u, d_c) + \quad (d_i) \ :$$

3. $u \in N_c$: in this case, it is easy to see that at least one of the longest paths in T_{d_i} starting from u must contain d_c. In fact, for the sake of contradiction, suppose that none of the longest paths in T_{d_i} starting from u contains d_c. Let us restrict our attention to any one of such longest paths, say P_u. Let w be the node in P_u nearest to d_c, and let z be the ending node of P_u other than u. Clearly, $d(w, z) \quad d(d_c, d_k)$, and therefore P_u can be modified into the path $P_u{}' = \langle u, \ldots, w, \ldots, d_c, \ldots, d_k \rangle$, containing d_c and such that

$$|P_u{}'| = d(u, w) + d(w, d_c) + d(d_c, d_k) \quad d(u, w) + d(w, z) = |P_u|$$

that is a contradiction. Given that L_u contains d_c, it remains to compute the length of a longest path starting from d_c and not containing u, and this can be done by looking at $|R_D|$ and at (d_i) (note that if (d_i) is exactly the length of a path passing through u, then it follows that $|R_D| \quad (d_i)$). Thus, $|L_u|$ can be computed in $O(1)$ time as

$$|L_u| = \max(d(d_c, u) + |R_D|, d(d_c, u) + \quad (d_i)).$$

Summarizing, the case $e \in L_D$ can be managed in $O(1)$ amortized time for any of the $O(\sqrt{m})$ selected edges. Since the case $e \in R_D$ is symmetric to the previous one, it can be managed with the same amortized runtime. Repeating the above for all the $n - 1$ edges of T we therefore have the following:

Theorem 1. *The ABS problem for a minimum diameter spanning tree T of a graph G with n vertices and m edges can be solved in $O(n\sqrt{m})$ time, using $O(m + n)$ space.* □

4 Constructing and Using Compressed Paths

In this section we use an adaptation of the well-known *path halving* compression technique to prove that a $Findpath(v;y)$ operation can be satisfied in $O(1)$ amortized time, as required to solve efficiently the ABS problem.

We start creating a *virtual forest* F of trees. Initially, F is composed of n singletons, each one associated to a node $v \in V$. With each node v in F the following values are associated: $[h_i(v); a_i(v)]; i = 1; 2$, as defined in Section 3.2 and $up(v)$, which will contain an estimation of the length of a longest path emanating from v and ascending towards d_c in Φ, now considering the edges of the path from d_c to v as directed from v towards d_c. At the beginning, $up(v) = 0; \forall v \in F$. The following instructions manipulate F:

- $Link(u;v)$: combine the trees with roots u and v into a single tree rooted in u, adding the edge $e = (u;v)$ of length $|e|$;
- $Eval(v)$: Return the length of a longest path starting from v in the tree containing it and apply a suited path halving compression technique.

Note that $Eval(v)$ assumes that a pointer to element v is obtained in constant time. The sequence of $Link()$ operations in F is determined by the sequence of edge removals from Φ. Remember that we sequentially consider all the edges $e \in E_T$ in postorder fashion. When the edge $e = (x;y)$ is (temporarily) removed, we perform a sequence of $Link(y;z_i)$ in F, where $z_i; i = 1;\ldots;k$ are all the sons of y in Φ. This means that for any given node $v \in V$, whenever a $Findpath(v;y)$ in Φ occurs, the node y is exactly the root of v in F. In fact, remember that we only ask $Findpath(v;y)$ on nodes descending from the currently removed edge. This observation is crucial for the correctness of the method. We implement a $Findpath(v;y)$ operation in Φ by means of an $Eval(v)$ operation in F, that examines all the nodes along the path from v to the root y of the tree containing it and compresses such a path. The compression technique used is an adaptation of the *path halving* technique [11], which will guarantee the *associativity* of $Eval(v)$. Let us describe how the path halving works. Given a couple of nodes $u;v \in V$, with u ancestor of v ($u \quad v$) in Φ, we define the following function

$$h(u;v) = \begin{array}{l} h_2(u) \text{ if } a_1(u) \quad v \\ h_1(u) \text{ otherwise.} \end{array}$$

It is easy to see that $h(u;v)$ can be computed in $O(1)$ time, for any $u;v \in V$, after $O(n)$ time of preprocessing of Φ. Assume an $Eval(v)$ operation occurs and this is the first time an $Eval()$ operation takes place. For the sake of simplicity, let us focus on a path of three nodes $\langle y;u;v\rangle$, with $y \quad u \quad v$ in Φ and such that $|(u;v)| = \quad$ and $|(y;u)| = \quad$. We have

$$Eval(v) = \max \{ h_1(v); up(v); \quad + h(u;v); \quad + up(u); \quad + \quad + h(y;v); \quad + \quad + up(y) \ :$$

The compression takes place as part of this operation, and replaces the edge $(u; v)$ with an edge $(y; v)$ of length $+$ and the label $up(v)$ of v with

$$up(v) = \max \quad up(v); \quad + h(u; v); \quad + up(u) \ :$$

After the compression we hence have

$$Eval(v) = \max \quad h_1(v); \max \quad up(v); \quad + h(u; v); \quad + up(u) \ ; \quad + \quad + h(y; v); \quad + \quad + up(y)$$

exactly as before the compression. Therefore, we can conclude that $Eval(v)$ compresses paths while correctly maintaining longest path information.

In the general case, let $hv_0 = v; v_1; : : : ; v_k = yi$ be the path from v to the root y of the tree containing v in F, having edges $e_i = (v_i; v_{i+1}); i = 0; : : : ; k - 1$. We have

$$Eval(v) = \max_{i=1;:::;k} \quad h_1(v); up(v); h(v_i; v) + \sum_{j=0}^{i-1} je_j j; up(v_i) + \sum_{j=0}^{i-1} je_j j \ : \quad (2)$$

W.l.o.g., let k be even. The path halving technique makes every other node along the path (except the last and the next to last) point to its grandfather, i.e., replaces the edge $(v_i; v_{i+1}); i = 0; 2; : : : ; k - 2$ with the edge $(v_i; v_{i+2})$ of length $je_i j + je_{i+1} j$ and sets

$$up(v_i) = \max \quad up(v_i); je_i j + h(v_{i+1}; v_i); je_i j + up(v_{i+1}) \ ; i = 0; 2; : : : ; k - 2: \quad (3)$$

Hence, after the halving, the path between v and y is $hv_0 = v; v_2; : : : ; v_{k-2}; v_k = yi$, with edges $e'_i = (v_i; v_{i+2}); i = 0; 2; : : : ; k - 2$ of length $je_i j + je_{i+1} j$. Therefore, after the halving, we have

$$Eval(v) = \max_{i=1;:::;k=2} \left(h_1(v); up(v); h(v_{2i}; v) + \sum_{j=0}^{2i-1} je_j j; up(v_{2i}) + \sum_{j=0}^{2i-1} je_j j \right)$$

which, from (3), is equivalent to (2). Therefore, it turns out that $Eval(v)$ before and after the halving is invariant. Remembering the order the edges are removed, it is clear that the compression of the paths works correctly (i.e., the compression incrementally proceeds towards the upper levels of T, according to the edge removals). Therefore, we conclude that $Findpath(v; y) \quad Eval(v)$.

Since naive linking in F must be applied to preserve paths of T, a sequence of $Q \quad n$ $Eval(v)$ queries in F can be satisfied in $O(Q \log_{(1+Q=n)} n)$ time [11]. Therefore, to establish that a single query can be satisfied in $O(1)$ amortized time, it remains to prove that $Q = \Omega(n^{1+})$ for some constant > 0. We now informally show that $Q = (n\sqrt{m})$, i.e., $1=2$.

Let us distinguish the edges of T in *basic edges* (i.e., edges contained inside a basic cluster [7] and therefore associated to a node at level $\ell = 0$ in the corresponding topology tree) and *spanning edges* (i.e., edges joining two clusters at the same level $\ell \quad 0$ of the topology tree). If an edge removed from T is a basic edge, then we have $|S_e| = \quad (\sqrt{m})$. On the other hand, if an edge removed from T is a spanning edge and joins two clusters corresponding to nodes at level ℓ in the topology tree, $0 \quad \ell \quad d\log \sqrt{m}e - 1$, then we have $|S_e| = \quad (\sqrt{m}/2^{\ell})$. For each edge in S_e a *Findpath*() query is issued. Hence the minimum overall number of queries is obtained when as many as possible of the $n - 1$ failing edges of T are spanning edges at the highest possible level in the topology tree. Since there will be $(\sqrt{m}/2^{\ell+1})$ spanning edges for nodes at level ℓ of the topology tree, i.e., a total of $(\sqrt{m})$ spanning edges, it follows that the minimum number of queries is posed when $(\sqrt{m})$ edges of T are spanning and the remaining $(n - \sqrt{m})$ are basic. Therefore, we have

$$Q = \Omega\left((n - \sqrt{m})\sqrt{m} + \sum_{\ell=0}^{d\log \sqrt{m}e-1} \frac{\sqrt{m}}{2^{\ell+1}}\frac{\sqrt{m}}{2^{\ell}}\right) = \Omega(n\sqrt{m})$$

and since $Q = O(n\sqrt{m})$, it follows $Q = \quad (n\sqrt{m})$, which implies $\quad 1/2$.

Acknowledgements { The authors would like to thank anonymous referees for their helpful suggestions.

References

1. S. Alstrup, J. Holm, K. de Lichtenberg and M. Thorup, Minimizing diameters of dynamic trees, *Proc. 24th Int. Coll. on Automata, Languages and Programming (ICALP)*, (1997) 270–280.
2. D. Eppstein, Z. Galil and G.F. Italiano, Dynamic graph algorithms, Tech. Rep. CS96-11, Univ. Ca' Foscari di Venezia (1996).
3. G.N. Frederickson, Data structures for on-line updating of minimum spanning trees, *SIAM J. Computing*, **14** (1985) 781–798.
4. G.N. Frederickson, Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees. *Proc. 32nd IEEE Symp. on Foundations of Computer Science (FOCS)*, (1991) 632–641.
5. M. Grötschel, C.L. Monma and M. Stoer, Design of survivable networks, in: *Handbooks in OR and MS, Vol. 7*, Elsevier (1995) 617–672.
6. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.
7. G.F. Italiano and R. Ramaswami, Maintaining spanning trees of small diameter, *Proc. 21st Int. Coll. on Automata, Languages and Programming (ICALP)*, (1994) 212–223. A revised version will appear in *Algorithmica*.
8. K. Iwano and N. Katoh, Efficient algorithms for finding the most vital edge of a minimum spanning tree, *Info. Proc. Letters*, **48** (1993) 211–213.
9. K. Malik, A.K. Mittal and S.K. Gupta, The k most vital arcs in the shortest path problem, *Oper. Res. Letters*, **8** (1989) 223–227.
10. E.Nardelli, G.Proietti and P.Widmayer, Finding the detour-critical edge of a shortest path between two nodes, *Info. Proc. Letters*, to appear.
11. R.E. Tarjan and J. van Leeuwen, Worst-case analysis of set union algorithms, *JACM*, **31** (2) (1984) 245–281.

Augmenting Su x Trees, with Applications

Yossi Matias[1] [?], S. Muthukrishnan[2] [??], Süleyman Cenk Ṣahinalp[3] [???], and Jacob Ziv[4] [y]

[1] Tel-Aviv University, and Bell Labs, Murray Hill
[2] Bell Labs, Murray Hill
[3] University of Warwick and University of Pennsylvania
[4] Technion

Abstract. Information retrieval and data compression are the two main application areas where the rich theory of string algorithmics plays a fundamental role. In this paper, we consider one algorithmic problem from each of these areas and present highly efficient (linear or near linear time) algorithms for both problems. Our algorithms rely on augmenting the *su x tree*, a fundamental data structure in string algorithmics. The augmentations are nontrivial and they form the technical crux of this paper. In particular, they consist of adding extra edges to suffix trees, resulting in Directed Acyclic Graphs (DAGs). Our algorithms construct these "suffix DAGs" and manipulate them to solve the two problems efficiently.

1 Introduction

In this paper, we consider two algorithmic problems, one from the area of Data Compression and the other from Information Retrieval. Our main results are highly efficient (linear or near linear time) algorithms for these problems. All our algorithms rely on the *su x tree* [McC76], a versatile data structure in combinatorial pattern matching. Suffix trees, with suitably simple augmentations, have found numerous applications in string processing [Gus98,CR94]. In our applications too, we augment the suffix tree with extra edges and additional information. In what follows, we describe the two problems and provide some background information, before presenting our results.

[?] Department of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel; and Bell Labs, Murray Hill, NJ, 07974, USA; matias@math.tau.ac.il. Partly supported by Alon Fellowship.
[??] Bell Labs, Murray Hill, NJ, 07974, USA; muthu@research.bell-labs.com.
[???] Department of Computer Science, University of Warwick, Coventry, CV4-7AL, UK; and Center for BioInformatics, University of Pennsylvania, Philadelphia, PA, 19146, USA; cenk@dcs.warwick.ac.uk. Partly supported by ESPRIT LTR Project no. 20244 - ALCOM IT.
[y] Department of Electrical Engineering, Technion, Haifa 32000, Israel; jz@ee.technion.ac.il.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 67{78, 1998.
c Springer-Verlag Berlin Heidelberg 1998

1.1 Problems and Background

We consider the *document listing problem* of interest in Information Retrieval and the HYZ *compression problem* from context-based Data Compression.

The Document Listing Problem. We are given a set of documents $T = \{T_1, \ldots, T_k\}$ for preprocessing. Given a query pattern P, the problem is to output a list of all the documents that contain P as a substring.

This is different from the standard query model where we are required to output all the *occurrences* of P. The standard problem can be solved in time proportional to the number of occurrences of P in T using a suffix tree. In contrast, our goal in solving the document listing problem is to generate the output with a running time that depends on the number of documents that contain P. Clearly, the latter may be substantially smaller than the former if P occurs multiple times in the documents.

A related query is where we are required to merely report the *number* of documents that contain P. An algorithm that solves this problem in $O(|P|)$ time is given in [Hui92], which is based on data structures for computing lowest common ancestor (LCA) queries.

The document listing problem is of great interest in information retrieval and has independently been formulated in many scenarios (see pages 124-125 in [Gus98] for a "morbid" application), for example in discovering gene homologies [Gus98].

The HYZ Compression Problem. Formally the (;)-*HYZ* compression problem is as follows. We are given a binary string T of length t. We are asked to replace disjoint *blocks* (substrings) of size with desirably shorter codewords. The codewords are selected in a way that it would be possible for a corresponding decompression algorithm to compute the original T out of the string of codewords. This is done as follows. Say the first $i-1$ such blocks have been compressed. To compute the codeword c_j for block j, we first determine its *context*. The context of a block $T[i : l]$ is the longest substring $T[k : i-1]; k < i;$ of size at most such that $T[k : l]$ occurs earlier in T. The codeword c_j is the ordered pair $\langle \gamma; \; i$ where γ is the length of the context of block j and is rank of block j with respect to the context, according to some predetermined ordering. For instance, one can use the lexicographic ordering of all distinct substrings of size exactly that follow any previous occurrence of the context of block j in the string. The (;)-*HYZ* compression scheme is based on the intuition that similar symbols in data appear in similar contexts. At the high level, it achieves compression by sorting context-symbol pairs in lexicographic order, and encoding each symbol according to its context and its rank. Thus it is a *context-based* scheme.

The (;)-*HYZ* compression problem has been proposed recently in [Yok96] and [HZ95,HZ98]. The case considered in [Yok96] is one where $= O(1)$ and is unbounded. During the execution of this algorithm, the average length of a codeword for representing a -sized block is shown to approach the *conditional entropy* for the block, $H(C)$, within an additive term of $c_1 \log H(C) + c_2$ for

constants c_1 and c_2, provided that the input is generated by a limited order Markovian source.[1] An independent result [HZ95,HZ98] is more general since it applies to all ergodic sources and yields essentially the best possible non-asymptotic compression even for a moderate amount of memory (at least for some ergodic sources). Of relevance to us is the fact that even with limited context of $= O(\log t)$, this scheme achieves the optimal compression (in terms of the conditional entropy) provided $\log\log t$, and in fact, this is true for all ergodic sources.[2]

Lempel-Ziv compression schemes and their variants [ZL77,ZL78,Wel84] that are popular in practice (and are used in tools such as `UNIX compress, compact, gnuzip, pkzip, winzip`, the gif image compression format and the current modem compression standard V42bis) do not achieve optimality under the refined notion of informational content (namely, conditional entropy) stated above. Hence, the *HYZ* compression scheme is more powerful.

Lempel-Ziv schemes and their variants are popular because they have (efficient) online/linear time implementations [RPE81]. However, such efficient algorithms for the HYZ compression scheme have not been known thus far. Our paper addresses the problem of efficiently implementing the (;)-*HYZ* compression scheme.[3]

1.2 Our Results

We present the following algorithmic results in this paper. Throughout we assume that the strings are drawn from a binary alphabet[4].

1. For the document listing problem, we present an algorithm that preprocesses the k documents in linear time (that is, $O(\sum_i t_i)$) and space. The time to answer a query with pattern P is $O(|P| \log k + out)$ where out is the number of documents that contain P. The algorithm relies on an augmentation of the suffix tree that we call the *Su x-DAG*.[5]

[1] See [Yok96] for the definition of the conditional entropy, its relevance in context-dependent compression schemes and details of the limited order Markovian source in the claim.

[2] See [HZ95,HZ98] for requirements on the source type and the result. In particular, this results are valid for large n but they are not asymptotic in nature – unlike, for example, [ZL77] – and hence might be of additional practical interest.

[3] There are many other context-based data compression methods in the literature [CW84,BW94,WRF95]. These have not been shown to be optimal under refined information-theoretic notion such as the conditional entropy. Experimental studies have confirmed that these context-based methods compress English text better than Lempel-Ziv variants [BCW90]. However, these context-based methods are either not very fast (they usually require time and sometimes space superlinear with the size of the input), or not online. For a survey and comparison of these methods, see [BCW90].

[4] The analyses in [HZ95,HZ98] are for binary strings.

[5] The suffix-DAG data structure also helps solve the count query in time $O(|P|)$, which matches the best known complexity [Hui92]. We present a new lemma for

Although this problem is natural, no nontrivial algorithmic results were known before our paper. The fastest algorithms for solving this problem run in time proportional to the number of occurrences of the pattern in all documents; clearly, this could be much larger than the number of documents that contain P, if P occurs multiple times in a document.

2. For the (;)-*HYZ* compression problem, we provide two algorithms. The first algorithm takes time $O(t$) (for compression and decompression) and works for any maximum context size . This gives a linear time algorithm for $= O(1)$, the case considered in [Yok96]. The only previously known algorithm for solving this problem is in [Yok96], where for $= O(1)$, the author presents an $O(t$) time algorithm. This algorithm is optimal only for small contexts, that is, $= O(1)$, and for unbounded , this running time $O(t^2)$.
   The second algorithm is tuned for the optimal case identified in [HZ95,HZ98]: we provide an $O(t)$ time algorithm to compress and decompress, provided that $= \log\log t$, and $= O(\log t)$. It is the first efficient algorithm for this problem. Notice that for both cases, our algorithms are optimal.
   The first algorithm is similar to the McCreight's method for building the suffix tree of the string online; however, we need to appropriately augment each node with at most units of information. The second algorithm maintains a trie on all strings within the specified context size together with "suffix links" on this trie (as in a suffix tree), and additionally, auxiliary edges on each node to some of its descendants. The technical details of the latter algorithm are more involved than the former, but in both cases the crux is to design algorithms for maintaining and utilizing the augmented information and the structural properties of the suffix links are used extensively to achieve this.
   The efficient algorithmic results supplement the information-theoretic properties of the HYZ schemes proved in [Yok96,HZ95,HZ98]; hence these schemes may prove to be a promising alternative to the existing ones in practice. We are currently experimenting with our algorithms.

2 The Su x-DAG Data Structure

Consider a set of document strings $T = fT_1; T_2; : : : ; T_k g$, of respective sizes $t_1; t_2; : : : ; t_k$ to be preprocessed. Our goal is to build a data structure which would support the following queries on an on-line pattern P of size p: (1) *list query* – the list of the documents that contain P, and (2) *count query* – the number of documents that contain P.

Theorem 1. *Given T and P, there is a data structure which responds to a count query in $O(p)$ time, and a list query in $O(p \log k + \mathtt{out})$ time, where* out *is the number of documents in T that contain P.*

computing count queries in bottom-up fashion, eliminating dependancy on the use of data structures for handling LCA queries, hence our scheme might work faster in practice.

Proof Sketch. We build a data structure , that we call the suffix-DAG of documents $T_1;:::;T_k$, in $O(t) = O(\ \ t_k)$ time using $O(t)$ space. The suffix-DAG of T, denoted by $SD(T)$, contains the generalized suffix tree, $GST(T)$, of the set T at its core. A *generalized su x tree* of a set of documents is defined to be the compact trie of all the suffixes of each of the documents in T ([Gus98]). Each leaf node l in $GST(T)$ is labeled with the list of documents which have a suffix represented by the path from the root to l in $GST(T)$. We denote the substring represented by a path from the root to any given node n by $P(n)$.

The nodes of $SD(T)$ are the nodes of $GST(T)$ themselves. The edges of $SD(T)$ are of two types: (1) the *skeleton edges* of $SD(T)$ are the edges of $GST(T)$; (2) the *supportive edges* of $SD(T)$ are defined as follows: given any two nodes n_1 and n_2 in $SD(T)$, there is a pointer edge from n_1 to n_2, if and only if (i) n_1 is an ancestor of n_2, and (ii) among the suffix trees $ST(T_1); ST(T_2);:::; ST(T_k)$ of respective documents $T_1; T_2;:::; T_k$, there exists at least one, say $ST(T_i)$, which has two nodes, $n_{1;i}$ and $n_{2;i}$ such that $P(n_1) = P(n_{1;i})$, $P(n_2) = P(n_{2;i})$, and $n_{1;i}$ is the parent of $n_{2;i}$. We label such an edge with i, for all relevant documents T_i.

In order to respond to the count and list queries, we build one of the standard data structures that support least common ancestor (LCA) queries on $SD(T)$ in $O(1)$ time [SV88]. Also, for each of the internal node n of $SD(T)$, we keep an array that stores its supportive edges in pre-order fashion, and the number of documents which include $P(n)$ as a substring.

The following lemmas state the complexities of the procedures for responding to list and count queries.

Lemma 1. *The su x-DAG is su cient to respond to the count queries in $O(p)$ time, and to list queries in $O(p \log k + out)$ time.*

Proof Sketch. The procedure for responding to count queries is as follows. With P, trace down $GST(T)$ until the highest level node n is reached for which P is a prefix of $P(n)$. We simply return the number of documents that contain $P(n)$ as a substring; recall that this information is stored in node n.

The procedure for responding to list queries is as follows. We locate node n (defined above) in $SD(T)$ and traverse $SD(T)$ backwards from n to the root. At each node u on the path, we determine all supportive edges out of u have their endpoints in the subtree rooted at n. The key observation is that all such edges will form a consecutive segment in the array of supportive edges residing with node u. This segment can be identified with two binary searches using an oracle that determines if a given edge has its endpoint in the subtree rooted at node n. Performing an LCA query with the endpoint of that edge and node n provides such an oracle taking $O(1)$ time. We can prove that the maximum size of the array of supportive edges attached to any node is at most $kj \ \ j$, where $j \ \ j = O(1)$ is the size of the alphabet of the string. Thus this procedure takes $O(\log k)$ time at each node u on the path from n to the root to identify the segment of supportive edges; the entire segment of such edges is output at each node u. The output of all such segments contains duplicates, however, we can

prove that the total size of the output is $O(\mathtt{out}|P|) = O(\mathtt{out})$, where $\mathtt{out}$ is the number of occurrences of P in t. We provide the proof of the observations above in the full version of this paper. □

Lemma 2. *The suffix-DAG of the document set T can be constructed in $O(t)$ time and $O(t)$ space.*

Proof Sketch. The construction of the generalized suffix tree $GST(T)$ with all suffix links, and a data structure to support constant time LCA queries are standard (see [CR94]). What remains is to describe for each node n of $SD(T)$, (1) how its supportive edges are constructed, (2) how its supportive edge array is built, and (3) how the number of documents that include $P(n)$ is computed.

The supportive edges with, say, label i, can be built by emulating McCreight's construction for $ST(T_i)$ (the suffix tree of T_i) on $SD(T)$. Notice that for each node in $ST(T_i)$, there is a corresponding node in $SD(T)$ with the appropriate suffix link. This implies that one can iteratively construct the supportive edges by using the suffix links as follows: Suppose that at some given step of the construction, a supportive edge between two nodes n_1 and n_2 with label i has been established. Let the suffix links from n_1 and n_2 reach nodes n_1^0 and n_2^0 respectively. Then, either there is a supportive edge between n_1^0 and n_2^0, or there exists one intermediate node to which there is a supportive edge from n_1^0, and there exists one intermediate node from which there is a supportive edge to n_2^0. The time to compute such intermediate nodes can be charged to the number of characters in the substring represented by the path between n_1 and n_2 in T_i. We leave the details of the non-trivial construction of the supportive edge arrays to the full paper.

Given a node n of $GST(T)$, the number of documents, $\#(n)$, which contain the substring of n, can be computed as follows: The only information we use about each node n is (1) the number of supportive edges from n to its descendants, $\#''(n)$, and (2) the number of supportive edges to n from its ancestors, $\#\#(n)$. □

Lemma 3. *For any node n, $\#(n) = \sum_{n^0 \text{2children of } n} \#(n^0) + \#''(n) - \#\#(n)$.*

Proof. The proof follows from the following key observations:

- if a document T_i includes the substrings of more than one descendant of n, then there should exist a node in $ST(T_i)$ whose substring is identical to that of n;
- given two supportive edges from n to n_1 and n_2, the path from n to n_1 and the path from n to n_2 do not have any common edges.

□

This concludes the proof of the lemma for suffix-DAG construction and hence the proof of the theorem. □

We note that using the suffix-DAG structure and additional edges between the arrays of supportive edges at endpoints of the edges, we can prove the following. There is an algorithm that preprocesses T in $O(t \log\log t)$ time and $O(t)$ space following which a list query can be answered in time $O(p + \mathtt{out})$. The details will be presented in the final version.

3 The Compression Algorithms

Consider a compression scheme with parameters and . We refer to it as C ; , and whenever and will be understood we may use C instead. The input to C ; is a string T of t characters. In this discussion, we assume that the alphabet is binary, however our results trivially generalize to any constant size alphabet. We denote by $T[i]$, the ith character of T (1 i t), and by $T[i : j]$ the substring of T which begins at $T[i]$ and ends at $T[j]$. The parameters and are (possibly constant) functions of t.

Recall the compression scheme from Section 1.1. The compression scheme C ; performs compression by partitioning the input T into contiguous substrings, or blocks of characters, and replacing each block by a corresponding codeword. To compute the codeword c_j for block j, C ; first computes the *context* for block j. The context of a block $T[i : l]$ is the longest substring $T[k : i-1]$ for $k < i$, for which $T[k : l]$ occurs earlier in T. If the context size exceeds , the context is truncated to contain only the rightmost characters. The codeword c_j is the ordered pair $\langle y;\ i\rangle$. We denote by y the size of the context for block j. We denote by the lexicographic order of block j amongst all possible substrings of size immediately following earlier occurrences of the context of block j.[6]

Our results are as follows.

Theorem 2. *There is an algorithm to implement the compression scheme C ; which runs in $O(t$) time and requires $O(t$) space, independent of .*

Proof. We employ McCreight's method for the construction of the suffix tree, during which we augment the suffix tree as follows. For each node v, we store an array of size in which for each $i = 1;:::;$, we store the number of distinct paths rooted at v (ending at a node or within an edge) of precisely i characters minus the number of such distinct paths of precisely $i - 1$ characters; note that these numbers may be negative.

We now show how to efficiently construct this data structure.

Lemma 4. *There is an algorithm to construct the augmented su x tree of T in $O(t$) time.*

[6] For instance, for $T = 010011010:::$ and $= 2$, $= 1$, the context of block 9 (which consists of $T[9] = 0$ only) is $T[7 : 8] = 01$, since 010 appears earlier but 1010 does not. The two substrings which follow earlier occurrences of this context are $T[3] = 0$ and $T[6] = 1$. The lexicographic order of block 9 among these substrings is 1.

Proof. Consider the suffix tree construction due to McCreight. While inserting a new node v into the suffix tree, we update the subtree information, if necessary, of the ancestors of v which are at most characters higher than v. The number of these ancestors whose information will need to be changed is at most . We can argue that at most one of the fields of information at any ancestor of v needs to be updated. That completes the proof. □

We now show that the resulting date structure facilitates the implementation of the compression scheme C in desired time and space bounds.

Lemma 5. *The augmented su x tree is su cient to compute the codeword for each block of input T in amortized $O(\ ^2)$ time.*

Proof Sketch. Recall that the codeword for a block j consists of the pair $h\gamma;\ i$, where γ is the context size and is the lexicographic order of block j among substrings following its context. The computation of γ can be performed by locating the node in the suffix tree which represents the longest prefix of the context. This can be achieved in an inductive fashion by using the suffix links of McCreight in amortized $O(\ )$ time (details omitted).

The computation of can be performed as follows. We traverse the path between nodes v and w in the suffix tree. The node v represents the longest prefix of the context of block j. The node w, a descendant of v, represents the longest prefix of the substring formed by concatenating the context of block j to block j itself. During this traversal, we compute the size of the relevant subtrees representing substrings lexicographically smaller and lexicographically greater than the substring represented by this path.

We present the details of the amortization argument used in the computation of γ, and the details of subtree traversal used in the computation of in the full version. □

This completes the proof of the theorem. □

For $= O(1)$ as in [Yok96], the algorithm above takes linear time independent of .

We now turn our focus to our second algorithm for implementing the compression method $C_{\ ;}$ for the optimal choice of parameters $= O(\log t)$ and $= \log\log t$. In what follows, we restrict ourselves to the case of $= \log t$. Later we will describe how to extend this to larger .

Theorem 3. *There is an algorithm to implement the compression method $C_{\ ;}$ for $= \log t$ and $= \log\log t$ in $O(t)$ time using $O(t)$ space.*

Proof Sketch. Our algorithm is based on a suffix tree-like data structure which supports the following feature. Consider a node v in the suffix tree and the set of descendants of v which are characters apart from v. Our data structure enables one to compute the lexicographic order of the path between any such descendant and the node v. Once the lexicographic order of a given node w is known, the codeword of the block represented by the path between v and w of size is easily computed.

Our algorithm exploits the fact that the context size is bounded, and it seeks similarities between suffixes of the input up to a small size. For this purpose, we build the *trie* of the $\log t$-sized prefixes of all suffixes of the input T. We note that this data structure is similar to a suffix tree except that (i) every edge contains exactly one character, and (ii) for any pair of suffixes, it represents common prefixes of size at most $\log t$. We will refer this data structure as the *limited su x trie* of input T. In order to support the lexicographic order queries we augment the limited suffix trie by keeping a search data structure for each node v to maintain is descendants which are characters away from v.

The lemma below states that one can compute the codeword of any input block in optimal $O( )$ time with our data structure.

Lemma 6. *The augmented limited su x trie of input T is su cient to compute the codeword for any input block j in $O( )$ time.*

Proof Sketch. Given a block j of the input, suppose that the node v represents its context and that among the descendants of v, the node w represents the substring obtained by concatenating the context of block j and block j itself. Because $= \log\log t$, the maximum number of elements in the search data structure for v is $2 = O(\log t)$. There is a simple data structure that maintains k elements and computes the rank of any given element in $O(\log k)$ time; hence, one can compute the lexicographic order of node w in only $O(\log\log t) = O( )$ time. We leave the details of the proof to the full paper. □

Lemma 7. *The augmented limited su x trie of input T can be built and maintained in $O(t)$ time and space.*

Proof Sketch. The depth of our augmented limited suffix trie is bounded by $\log t$, hence the total number of nodes in the trie is only $O(t)$. This suggests that one can adapt McCreight's suffix tree construction in $O(t)$ time - without being penalized for building a suffix trie rather than a suffix tree.

To complete the proof what we need to do is to show that it is possible to construct and maintain the search data structures of all nodes in $O(t)$ time. This follows from the fact that each node v in our data structure is inserted to the search data structure of at most one of its ancestors. Therefore, the total number of elements maintained by all search data structures is $O(t)$. The insertion time of an element e to a search data structure, provided that the element e' in the data structure which is closest to e in rank, is $O(1)$. As the total number of nodes to be inserted in the data structure is bounded by $O(t)$, one can show that the total time for insertion of nodes in the search data structures is $O(t)$.

We leave the details of how the limited suffix trie is built and how the search data structures are constructed to the full paper. □

This completes the proof of the theorem. □

We use several new ideas to extend the result above to $= O(\log t)$, or more generally, to the arbitrary case. These include encoding all possible length paths into a constant number of machine words of size $\log t$, and performing

bit-wise operations on these words. The final step involves showing how a table indexed by all machine words can be built which will replace bit operations on machine words by mere table lookups. The end result is an $O(t)$ time and space compression and uncompression algorithm for general and $= \log \log t$.

4 Acknowledgements

We thank an anonymous referee for very fruitful suggestions.

References

BCW90. T. Bell, T. Cleary, and I. Witten. *Text Compression*. Academic Press, 1990.

Bro98. G. S. Brodal. Finger search trees with constant insertion time. In *ACM-SIAM Symposium on Discrete Algorithms*, 1998.

BW94. M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, DEC SRC, 1994.

CR94. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford Press, 1994.

CW84. J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

Gus98. D. M. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Addison Wesley, 1998.

Hui92. J. Hui. Color set size problem with applications to string matching. In *Combinatorial Pattern Matching*, 1992.

HZ95. Y. Hershkovits and J. Ziv. On sliding window universal data compression with limited memory. In *Information Theory symposium*, pages 17–22, September 1995.

HZ98. Y. Hershkovits and J. Ziv. On sliding window universal data compression with limited memory. *IEEE Trans. on Information Theory*, 44:66–78, January 1998.

McC76. E. M. McCreight. A space economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

RPE81. M. Rodeh, V. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, January 1981.

SV88. B. Schieber and U. Vishkin. On finding lowest common ancestors:simplification and parallelization. *SIAM Journal of Computing*, 17:1253–1262, 1988.

Wel84. T.A. Welch. A technique for high-performance data compression. *IEEE Computer*, pages 8–19, January 1984.

WRF95. M. J. Weinberger, J. J. Rissanen, and M. Feder. A universal finite memory source. *IEEE Transactions on Information Theory*, 41(3):643–652, 1995.

Yok96. H. Yokoo. An adaptive data compression method based on context sorting. In *IEEE Data Compression Conference*, 1996.

ZL77. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.

ZL78. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, IT-24(5):530–536, September 1978.

A Appendix

In this section we demonstrate the construction of the suffix-DAG of a set of documents by means of figures.

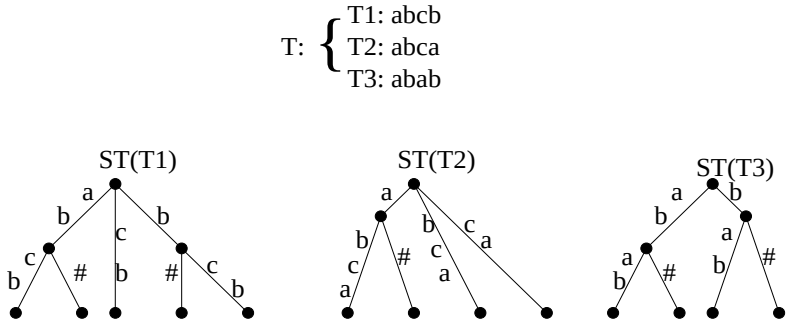


Fig. 1. The independent suffix trees of a given set of documents.

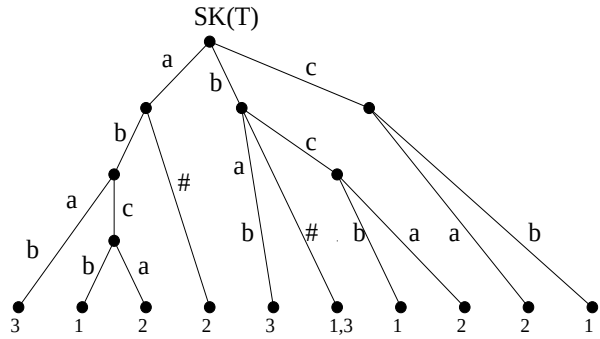


Fig. 2. The generalized suffix tree of the set of documents in Figure 1.

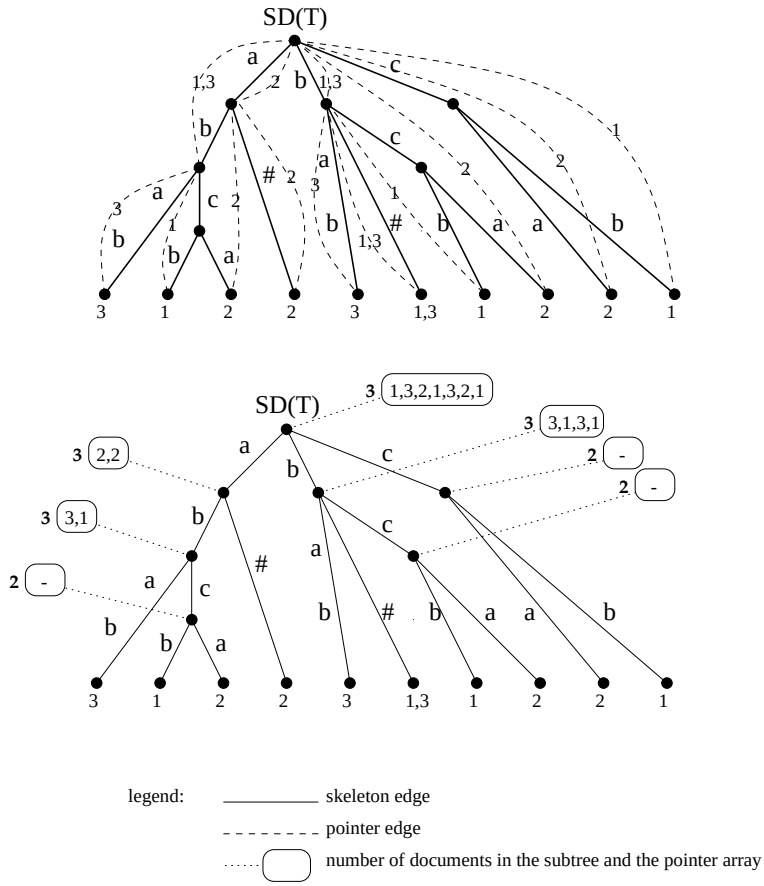


Fig. 3. The suffix-DAG of the set of documents in Figure 1.

Longest Common Subsequence from Fragments via Sparse Dynamic Programming

Brenda S. Baker[1] and Raffaele Giancarlo[2?]

[1] Bell Laboratories, Lucent Technologies,
700 Mountain Avenue, Murray Hill, NJ 07974, USA
bsb@research.bell-labs.com
[2] Dipartimento di Matematica ed Applicazioni, Universitá di Palermo,
Via Archirafi, 34-90123 Palermo - Italy
raffaele@altair.math.unipa.it

Abstract. Sparse Dynamic Programming has emerged as an essential tool for the design of efficient algorithms for optimization problems coming from such diverse areas as Computer Science, Computational Biology and Speech Recognition [7,11,15]. We provide a new Sparse Dynamic Programming technique that extends the Hunt-Szymanski [2,9,8] paradigm for the computation of the Longest Common Subsequence (LCS) and apply it to solve the LCS from Fragments problem: given a pair of strings X and Y (of length n and m, resp.) and a set M of matching substrings of X and Y, find the longest common subsequence based only on the symbol correspondences induced by the substrings. This problem arises in an application to analysis of software systems. Our algorithm solves the problem in $O(jMj \log jMj)$ time using balanced trees, or $O(jMj \log\log \min(jMj; nm=jMj))$ time using Johnson's version of Flat Trees [10]. These bounds apply for two cost measures. The algorithm can also be adapted to finding the usual LCS in $O((m+n)\log j \quad j + jMj \log jMj)$ using balanced trees or $O((m+n)\log j \quad j + jMj \log\log \min(jMj; nm=jMj))$ using Johnson's Flat Trees, where M is the set of maximal matches between substrings of X and Y and is the alphabet.

1 Introduction

Sparse Dynamic Programming [4,5] is a technique for the design of efficient algorithms mainly with applications to problems arising in Sequence Analysis. As in [4,5], here we use the term Sequence Analysis in a very broad sense, to include problems that share many common aspects and come from such diverse areas as Computer Science, Computational Biology and Speech Recognition [7,11,15]. A typical problem in Sequence Analysis is as follows: we are given two strings and we would like to find the "distance" between those strings under some cost

? Work Supported in part by Grants from the Italian Ministry of Scienti c Research and by the Italian National Research Council. Part of this work was done while the author was visiting Bell Laboratories of Lucent Technologies

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 79{90, 1998.
c Springer-Verlag Berlin Heidelberg 1998

assumptions. The technique can be concisely described as follows. We are given a set of Dynamic Programming (**DP** for short) recurrences to be computed using an associated **DP** matrix. The recurrences maximize (or minimize) some objective function. However, only a *sparse* set of the entries of the **DP** matrix really matters for the optimization of the objective function. The technique takes advantage of this sparsity to produce algorithms that have a running time dependent on the size of the sparse set rather than on the size of the **DP** matrix. To the best of our knowledge, the idea of using sparsity to speed up the computation of Sequence Analysis algorithms can be traced back to the algorithm of Hunt and Szymanski [9] for the computation of the Longest Common Subsequence (LCS for short) of two strings. However, the first systematic study of sparsity in **DP** algorithms for Sequence Analysis is due to Eppstein et al. [4,5]. Over the years, additional contributions have been provided by several authors (see for instance [6,12]).

The main contribution of this paper is to provide new tools for Sparse **DP**. Namely, we generalize quite naturally the well known Hunt-Szymanski [2,9,8] paradigm for the computation of the LCS of two strings. Our generalization, called LCS from Fragments, yields a new set of Sparse **DP** recurrences and we provide efficient algorithms computing them.

As a result, we obtain a new algorithm for the computation of the LCS of two strings that compares favorably with the known algorithms in the Hunt-Szymanski paradigm and, in its most basic version, is as simple to implement as Hunt-Szymanski.

In addition, our techniques solve an algorithmic problem needed for an application to finding duplication within or between software systems. In this case, a database of "matching" sections of code is obtained via a program such as dup [3]. The notion of "matching" code sections may be either exact textual matches or parameterized matches in the sense of [3], in which parameterized matches between code sections indicate textual similarity except for a systematic renaming of parameters such as variable or function names. Typically, to avoid finding matches too short to be meaningful, dup is used to find maximal matches over a threshold length. It would be convenient to have a graphical user interface that would enable scrolling simultaneously through two source programs, in such a way that matching sections are aligned. While there is a natural line-by-line alignment within a single pair of matching sections of code, the problem is how to handle alignment upon scrolling forward or backward from one pair of matching sections to other matching sections. A solution to LCS from Fragments optimizes the alignment across multiple matching segments.

We first present, at an informal level, the problem we have considered and then compare our results with the state of the art in Sparse **DP** and LCS algorithms.

Let X and Y be two strings of length n and m, respectively, over some alphabet . Without loss of generality, assume that $j \quad j \quad n$. The LCS problem is to find the longest string $s_1 s_2 \cdots s_k$ such that s_1, s_2, ... , s_k occur in that order (possibly with other symbols intervening) in both X and Y. The dual of

the LCS problem is to find the minimum edit distance between X and Y, where an insertion or deletion of a character has cost 1 (Levenshtein distance). It is straightforward to transform a solution to the LCS problem into a solution to the minimum edit distance problem, and vice versa [2,8,13].

For the LCS from Fragments problem, we are given strings X and Y and a set M of pairs of equal-length substrings of X and Y that "match". The notion of "match" is somewhat general in the sense that the two substrings of X and Y are equal according to some definition of "equality", e.g., standard character equality or parameterized match [3]. Each pair of substrings, called a "fragment", is specified in terms of the starting positions in X and Y and a length. The LCS from Fragments problem is to find a minimal-cost pair of subsequences of X and Y in which successive symbols correspond based on corresponding positions in a pair in M. For example, if $X = abcddaBC$ and $Y = ABCdabC$, and the fragments are $(1; 1; 4)$ (representing the first four characters of both strings) and $(5; 4; 4)$ (representing the last four characters of both strings), then (depending on the cost measure) subsequences $abcdaBC$ of X and $ABCdabC$ of Y might be a solution to the LCS from Fragments problem, since abc corresponds to ABC in the first fragment and $daBC$ corresponds to $dabC$ in the second fragment. We consider two cost measures based on edit distance.

When the notion of match is the standard one, i.e., the two strings are equal for each pair in M, we use the Levenshtein edit distance.

When the notion of match is that of parameterized match, i.e., for each pair in M, the two strings exibit a one-to-one correspondence as defined in [3], we treat the inclusion of each parameterized match in the LCS as a new "edit" operation. This naturally leads to our second cost measure: a cost of one is incurred for each insertion, deletion, or segment (of any length) of a fragment. Extension of this cost measure to deal with the full-fledged application is discussed in Section 5.

For both cost measures, LCS from Fragments can be computed in $O(jMj \log jMj)$ time. With sophisticated data structures, such as Johnson's version of Flat Trees [10], that bound reduces to $O(jMj \log\log \min(jMj; nm=jMj))$. The algorithm for the second cost measure is more complex than that for the Levenshtein cost measure.

If the set M consists of all pairs of maximal equal substrings of X and Y, and the Levenshtein cost measure is used, the solution of the LCS from Fragments problem is the usual LCS. This generalizes the Hunt-Szymanski paradigm, where the basis for the algorithm is the set of pairs of positions with the same symbols in X and Y. The LCS can be computed via LCS from Fragments in $O((m + n) \log j \quad j + jMj \log jMj)$ time, including the time for the computation of M. Moreover, using Johnson's version of Flat Trees [10], that bound reduces to $O((m + n) \log j \quad j + jMj \log\log \min(jMj; nm=jMj))$. Since $jMj \quad nm$, the time bounds that we obtain compare favorably with the best available algorithms for the computation of the LCS [2].

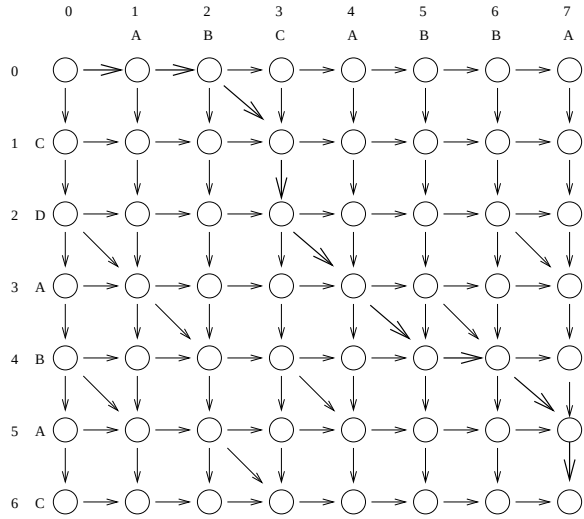


Fig. 1. An edit graph for the strings $X = CDABAC$ and $Y = ABCABBA$. It naturally corresponds to a **DP** matrix. The bold path from $(0;0)$ to $(6;7)$ gives an edit script from which we can recover the LCS between X and Y

2 Definitions and Preliminary Lemmas

Finding the LCS of X and Y is equivalent to finding the Levenshtein edit distance between the two strings [11], where the "edit operations" are insertion and deletion. Following Myers [13], we phrase the LCS problem as the computation of a shortest path in the edit graph for X and Y, defined as follows. It is a directed grid graph (see Fig. 1) with vertices $(i;j)$, where $0 \quad i \quad n$ and $0 \quad j \quad m$. We refer to the vertices also as *points*. There is a vertical edge from each non-bottom point to its neighbor below. There is a horizontal edge from each non-rightmost point to its right neighbor. Finally, if $X[i] = Y[j]$, there is a diagonal edge from $(i-1;j-1)$ to $(i;j)$. Assume that each non-diagonal edge has weight 1 and the remaining edges weight 0. Then, the Levenshtein edit distance is given by the minimum cost of any path from $(0;0)$ to $(n;m)$; the vertical and horizontal edges of this path correspond to the insertions and deletions of a minimal edit script, while the diagonal edges correspond to the LCS.

Our LCS from Fragments problem also corresponds naturally to an edit graph. The vertices and the horizontal and vertical edges are as before, but the diagonal edges correspond to a given set of fragments. Each fragment, formally described as a triple $(i;j;k)$, represents a sequence of diagonal edges from $(i-1;j-1)$ (the *start* point) to $(i+k-1;j+k-1)$ (the *end* point). For a fragment f, the start and end points of f are denoted by $start(f)$ and $end(f)$, respectively. In Figure 1, the fragments correspond to the maximal matching substrings of X and Y. In Figure 2, the fragments are the sequences of at least 2 diagonal

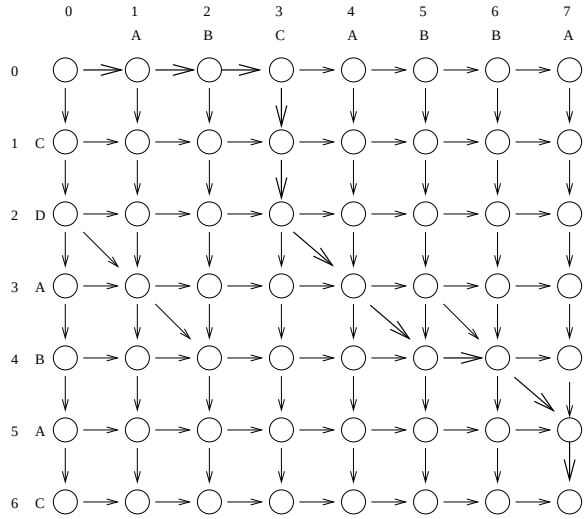


Fig. 2. An LCS from Fragments edit graph for the same strings as in Figure 1, where the fragments are the sequences of at least two diagonal edges of Figure 1. The bold path from $(0;0)$ to $(6;7)$ corresponds to a minimum-cost path under the Levenshtein edit distance

edges of Figure 1. The LCS from Fragments problem is equivalent to finding a minimum-cost path in the edit graph from $(0;0)$ to $(n;m)$. A minimum-cost path may use only part of certain fragments, as happens with fragment $(3;5;2)$ in the two figures. We consider two cost measures. As before, in the Levenshtein cost measure, each diagonal edge has weight 0 and each non-diagonal edge has weight 1. (However, all of our results generalize to the case of weighted Levenshtein edit distance. To simplify the presentation of our ideas, we restrict attention to the unweighted case.) To suit the software application, we also consider a cost measure in which a cost of 1 is incurred for traversing a horizontal edge, a vertical edge, or a segment (of any nonzero length) of a fragment.

For ease of presentation, we assume that the sequences of edges corresponding to different fragments are disjoint and do not even touch. However, the algorithms can be easily modified to handle overlapping fragments, which actually arise commonly in the application.

For a point p, define $x(p)$ and $y(p)$ to be the x- and y- coordinates of p, respectively. We also refer to $x(p)$ as the *row* of p and $y(p)$ as the *column* of p. Define the diagonal number of f to be $d(f) = y(start(f)) - x(start(f))$.

It will be helpful to show that we need only consider paths of restricted form. We say that a segment of a fragment f is a *pre* x of f if it includes $start(f)$.

Lemma 1. *Consider a cost measure in which horizontal and vertical edges have cost 1 and traversing a fragment segment of any length has a cost of 0 or 1. For*

any point p in the edit graph, there is a min-cost path from $(0;0)$ to p such that every fragment segment traversed is a prefix.

Proof. Suppose P is a min-cost path from $(0;0)$ to p that does not satisfy the lemma. We construct a new path P' with the same cost and fewer non-prefix segments; the argument is applied inductively to prove the lemma.

If the last non-prefix segment S of P is part of fragment f, let R be a rectangle with corners $(0;0)$ and $start(f)$; P begins within R and then exits it at a point q. The new path P' follows P except between q and S, where it instead follows the side of R to $start(f)$ and a prefix of f to S. The cost does not increase because the replaced part of P had to cross the same number of diagonals via horizontal or vertical edges, and P' has fewer non-prefix segments than P. □

Between successive fragment segments in a min-cost path satisfying Lemma 1, either there are only horizontal edges, only vertical edges, or both. In the last case, we can make a stronger statement about the preceding fragment segment.

Lemma 2. *In a min-cost path that satisfies Lemma 1, if there are at least one horizontal edge and at least one vertical edge between two successive fragment segments in the path, the earlier fragment segment is the entire fragment.*

Proof. If the lemma fails, the path cannot be a min-cost path because a lower-cost path is obtained by following the earlier fragment for an additional diagonal edge and reducing the number of horizontal and vertical edges. □

3 Levenshtein Cost Measure

In this section, we consider a cost measure corresponding to Levenshtein edit distance: diagonal edges are free, while insertions and deletions of characters have a cost of 1 each. For any point p, define $mincost_0(p)$ to be the minimum cost of any path from $(0;0)$ to p under this cost measure. Since diagonal edges are free, the proof of Lemma 1 yields the following corollary.

Corollary 1. *For any fragment f and any point p on f, $mincost_0(p) = mincost_0(start(f))$.*

By Corollary 1, it is reasonable to define $mincost_0(f) = mincost_0(start(f))$.

We say a fragment f' is *left of* $start(f)$ if some point of f' besides $start(f')$ is to the left of $start(f)$ on a horizontal line through $start(f)$. Similarly, a fragment f' is *above* $start(f)$ if some point of f' besides $start(f')$ is above $start(f)$ on a vertical line through $start(f)$.

Define $visl(f)$ to be the first fragment to the left of $start(f)$ if such exists, and undefined otherwise. Define $visa(f)$ to be the first fragment above $start(f)$ if such exists, and undefined otherwise.

We say that fragment f *precedes* fragment f' if $x(end(f)) < x(start(f'))$ and $y(end(f)) < y(start(f'))$, i.e. if the end point of f is strictly inside the rectangle with opposite corners $(0;0)$ and $start(f')$.

Suppose that fragment f precedes fragment f'. The shortest path from $end(f)$ to $start(f')$ with no diagonal edges has cost $x(start(f')) - x(end(f)) + y(start(f')) - y(end(f))$, and the minimum cost of any path from $(0;0)$ to $start(f')$ through f is that value plus $mincost_0(f)$. It will be helpful to separate out the part of this cost that depends on f by the definition $Z(f) = mincost_0(f) - x(end(f)) - y(end(f))$. Note that $Z(f) \quad 0$ since $mincost_0(f)$ $x(start(f)) + y(start(f))$.

Lemma 3. *For a fragment f, $mincost_0(f)$ is the minimum of $x(start(f)) + y(start(f))$ and any of c_p, c_l, and c_a that are de ned according to the following:*

1. *If at least one fragment precedes f, $c_p = x(start(f)) + y(start(f)) + \min\{Z(f') : f' \text{ precedes } f\}$.*
2. *If $visl(f)$ is de ned, $c_l = mincost_0(visl(f)) + d(f) - d(visl(f))$;*
3. *If $visa(f)$ is de ned, $c_a = mincost_0(visa(f)) + d(visa(f)) - d(f)$;*

Proof. Cases (1)-(3) represent the minimum costs of the various possible ways a path that can reach $start(f)$ from a preceding fragment: via both horizontal and vertical edges, horizontal edges only, or vertical edges only. ☐

Next, we develop an algorithm based on Lemma 3.

We assume the existence of a data structure of type D that stores integers j in some range $[0;u]$ and supports the following operations: (1) insert, (2) delete, (3) member, (4) min, (5) successor: given j, find the next larger value than j in D, and (6) max: given j, find the max value less than j in D.

If d elements are stored, D could be implemented via balanced trees [1] with $O(\log d)$ time per operation or via the van Emde Boas Flat Trees [14] with $O(\log\log u)$ time per operation. If we use Johnson's version of Flat Trees [10], the time for all of those operations becomes $O(\log\log G)$, where G is the length of the gap between the nearest integers in the structure below and above the priority of the item being inserted, deleted or searched for.

The general approach of the algorithm will be a sweepline approach where successive rows are processed, and within rows, points are processed from left to right. Lexicographic sorting of the $(x;y)$-values to enable this can be done in $O(jMj \log jMj)$ time. Alternatively, using data structure D, implemented via Johnson's version of Flat Trees as just discussed, sorting can be accomplished via a sequence of insertions, a min, and a sequence of successor operations in $O(jMj \log\log \min(jMj; nm/jMj))$ time, with analysis as in Eppstein et al. [5]. Not all the rows and columns need contain a start point or end point, and we generally wish to skip empty rows and columns for efficiency. From now on, we assume our algorithm processes only nonempty rows and columns, renumbered to exclude empty ones.

First, compute $visl(f)$ and $visa(f)$ for each fragment f via a sweepline algorithm. We describe the computation of $visl(f)$; that for $visa(f)$ is similar. For $visl(f)$, the sweepline algorithm sweeps along successive rows. Assume that we have reached row i. We keep all fragments crossing row i sorted by diagonal number in a data structure V of type D. For each fragment f such that

$x(start(f)) = i$, we calculate the fragment f^0 to the left of $start(f)$ from the sorted list of fragments; in this case, $visl(f) = f^0$. Then, for each fragment f with $x(start(f)) = i$, we insert f into V. Finally, we remove fragments $\hat{f}$ such that $y(end(\hat{f})) = i$.

If the data structure D is implemented as a balanced search tree, the total time for this computation is $O(jMj \log jMj)$. If van emde Boas Flat Trees are used, the total time is $O(jMj \log\log jMj)$ (we can "renumber" the diagonals so that the items stored in the data structure are in the range $[0; 2jMj]$). Even better, notice that we perform on the data structure three homogeneous sequences of operations per row: first a sequence of max operations (to identify visibility information for the start points of fragments), then a sequence of insertions, and finally a sequence of deletions. In that case, we can use Johnson's version of Flat Trees [10] to show that the sweep can be implemented to take $O(jMj \log\log \min(jMj; nm/jMj)$ time. The analysis is as in [4] (Lemma 1).

The main algorithm takes a sweepline approach of processing successive rows. It follows the same paradigm as the Hunt-Szymanski LCS algorithm [9] and the computation of the RNA secondary structure (with linear cost functions) [5].

We use another data structure B of type D, but this time B stores column numbers (and a fragment associated with each one). The values stored in B will represent the columns at which the minimum value of $Z(f)$ decreases compared to any columns to the left, i.e. the columns containing an end point of a fragment f for which $Z(f)$ is smaller than $Z(f^0)$ for any f^0 whose end point has already been processed and which is in a column to the left. Notice that, once we fix a row, D gives a partition of that row in terms of columns.

Within a row, first process any start points in the row from left to right. For each start point of a fragment, compute $mincost_0$ using Lemma 3. Note that when the start point of a fragment f is computed, $mincost_0$ has already been computed for each fragment that precedes f and each fragment that is $visa(f)$ or $visl(f)$. To find the minimum value of $Z(f^0)$ over all predecessors f^0 of f, the data structure B is used. The minimum relevant value for $Z(f^0)$ is obtained from B by using the max operation to find the $\max j < y(start(f))$ in B; the fragment f^0 associated with that j is one for which $Z(f^0)$ is the minimum (based on endpoints processed so far) over all columns to the left of the column containing $start(f)$, and in fact this value of $Z(f^0)$ is the minimum value over all predecessors of f.

After any start points for a row have been processed, process the end points. When an end point of a fragment f is processed, B is updated as necessary if $Z(f)$ represents a new minimum value at the column $y(end(f))$; successor and deletion operations may be needed to find and remove any values that have been superseded by the new minimum value.

Given M precomputed fragments, the above algorithm can be implemented in $O(jMj \log jMj)$ time via balanced trees, or $O(n + jMj \log\log jMj)$ time if van Emde Boas data structures are used. Moreover, using the same ideas as in Eppstein et al. [5], we can "re-schedule" the operations on D so that, for each processed row, we perform four sequences of homogeneous operations of the type

max, insert, delete, member. Using this fact and the analysis in Eppstein et al. [5], we can show that this phase of the algorithm takes $O(|M| \log\log \min(|M|, nm/|M|))$ time.

We note that for each fragment f, a pointer may be kept to the fragment from which a min-cost path arrived at $start(f)$, and hence both the LCS and a minimal edit script are easily recovered in $O(|M|)$ space and time.

Theorem 1. *Suppose $X[1:n]$ and $Y[1:m]$ are strings, and a set M of fragments relating substrings of X and Y is given. One can compute the LCS from Fragments in $O(|M| \log |M|)$ time and $O(|M|)$ space using standard balanced search tree schemes. When one uses Johnson's data structure, the time reduces to $O(|M| \log\log \min(|M|, nm/|M|))$.*

Proof. Correctness follows from Lemma 3 and the time analysis from the discussion preceding the statement of the theorem. □

Two substrings $X[i:i+k-1]$ and $Y[j:j+k-1]$ are a *maximal match* if and only if they are equal and the equality cannot be extended to the right or to the left. A maximal match between two substrings is conveniently represented by a triple (i, j, k), corresponding to a sequence of diagonal edges in the edit graph starting at $(i-1, j-1)$ and ending at $(i+k-1, j+k-1)$.

When M is the set of maximal matches between X and Y, a solution to the LCS from Fragments problem also gives a solution to the usual LCS problem. Using techniques in [3], we can compute the set of maximal matches in $O((m+n) \log |j\ j+|M|)$ time and $O(m+n+|M|)$ space. Thus, we obtain the following corollary.

Corollary 2. *Given two strings $X[1:n]$ and $Y[1:m]$ one can compute the LCS of those two strings in $O((m+n) \log |j\ j+|M| \log |M|)$ time and $O(m+n+|M|)$ space using standard balanced search tree schemes. When one uses Johnson's data structure, the time reduces to $O((m+n) \log |j\ j+|M| \log\log \min(|M|, nm/|M|))$.*

4 Cost Measure with Unit Cost for Fragment Segments

In this section, we consider a cost measure in which any segment of a fragment can be traversed at a cost of 1, regardless of length. Each insertion and deletion still incurs a cost of 1. For any point p, define $mincost_1(p)$ to be the minimum cost of a path from $(0, 0)$ to p under this cost measure.

The solution of the previous section breaks down under the new cost measure. The problem is that a minimum-cost path to a point on a fragment f may not be able to traverse f because of the cost of traversing f itself. The failure has ramifications because the proof of Lemma 3 assumed that the cost of a path through a point on $visl(f)$ could be computed from $mincost_0(visl(f))$ without concern about whether $visl(f)$ itself was used in the path, and similarly for $visa(f)$. Fortunately, the minimum cost of reaching a point on a fragment f

cannot be more than one less than the cost of reaching it through f; this fact follows from the proof of Lemma 1.

For any fragment f, we define $mincost_1(f) = 1 + mincost_1(start(f))$. This is the minimum cost of reaching any point on f (other than $start(f)$) via a path that uses a nonempty segment of f.

We wish to consider the minimum cost of reaching $start(f)$ from a previous fragment as we did before, but will take into account the complications of the new cost measure. Define $visl(f)$ and $visa(f)$ as before. Corresponding to Z from before, we define $Z_p(f) = mincost_1(f) - x(end(f)) - y(end(f))$. Since the cost of a sequence of horizontal edges or a sequence of vertical edges is the change in diagonal number, it will be convenient to define $Z_l(f) = mincost_1(f) - d(f)$ and $Z_a(f) = mincost_1(f) + d(f)$ to separate out costs dependent only on f. The lemma that corresponds to Lemma 3 is the following.

Lemma 4. *For a fragment f, $mincost_1(start(f))$ is the minimum of $x(start(f)) + y(start(f))$ and any of c_p, c_l, and c_a that are defined via the following:*

1. *if at least one fragment precedes f, then $c_p = x(start(f)) + y(start(f)) + \min\{Z_p(f^0) : f^0$ precedes $f\}$.*
2. *if $visl(f)$ is defined, then c_l is defined as follows: if there exists at least one fragment f^0 to the left of $start(f)$ with $Z_l(f^0) = Z_l(visl(f)) - 1$ then $c_l = Z_l(visl(f)) - 1 + d(f)$ else $c_l = Z_l(visl(f)) + d(f)$.*
3. *if $visa(f)$ is defined then c_a is defined as follows: if there exists at least one fragment f^0 above $start(f)$ with $Z_a(f^0) = Z_a(visa(f)) - 1$ then $c_a = Z_a(visa(f)) - 1 - d(f)$ else $c_a = Z_a(visa(f)) - d(f)$.*

Proof. The proof is similar to that of Lemma 3 except that cases (2) and (3) are more complicated.

In particular, (2) covers two cases for paths that reach $start(f)$ via only horizontal edges from the previous fragment. In one case, the path traverses $visl(f)$, and in the other case, it passes through a point p on $visl(f)$ but does not traverse a segment of $visl(f)$. Since the cost to p can be lower by at most one in the latter case than in the former, we need consider only the fragments to the left of $start(f)$ that will result in a cost of one less than in the former case. Since the cost of a sequence of horizontal edges is the change in diagonal number, we need consider only fragments f^0 to the left for which $Z_l(f^0) = Z_l(f) - 1$. For condition (3), the argument is similar. □

Lemma 4 allows us to restrict our attention to fragments f^0 with particular values of Z_a and Z_l.

We describe the approach for a particular value of Z_l; for Z_a, the approach is similar. For the set S of fragments f^0 with this value of $Z_l(f^0)$, we need to store information about the relevant fragments in a way that enables answering queries about whether one of the fragments in S is left of $start(f)$ for a particular fragment f, i.e. to ask whether there exists an f^0 in S with $x(start(f^0)) < x(start(f))$ $x(end(f^0))$ and $d(f) > d(f^0)$. Our strategy is to keep track of the minimum diagonal number of all fragments crossing each row

with this value of Z_l. To do this, we use a data structure of type D (as defined as in the previous section) to store the endpoints of intervals where the minimum diagonal number changes, and with each such endpoint, we store the diagonal number; to determine whether any fragment in the set is to the left of $start(f)$, we query to find the maximum stored value less than $x(start(f))$, retrieve the corresponding diagonal number, and compare it to $d(f)$. The construction for case (2) is similar. The data structure must be built dynamically as fragments are processed and the values of Z_l become available.

A separate data structure is kept for each distinct value of Z_l or Z_a. One possibility is to keep a pointer to each of these data structures in two arrays A_l and A_a; these would use $O(n+m)$ space since the range of values of Z_l and Z_a is $[-(n+m); 0]$. Alternatively, two instances of D can be used to store the distinct values of Z_l and Z_a that occur, together with the associated pointers. The total space used is $O(jMj)$.

The algorithm proceeds as follows. First, compute $visl(f)$ and $visa(f)$ as before. As before, the main sweepline algorithm sweeps across successive rows and has a data structure B that keeps track of the columns in which the minimum value of $Z_l(f)$ decreases compared to columns for the left. However, when a new value is to be computed for $mincost_1(start(f))$, the new algorithm applies Lemma 4, using the data structure for $z = Z_l(visl(f))$. After the values of $mincost_1(f)$ for the fragments with start points in the row have been computed, the next task is to update the data structures storing the sets of used values of $Z_l(f)$ and $Z_a(f)$ and the individual data structure for each such value. Then B is updated as before based on the end points in the row and the values of $Z_p(f)$ for the fragments containing these end points.

As before, data structures of type D can be implemented via either balanced trees or Johnson's version of van Emde Boas Flat Trees with the same time bounds as discussed in the previous section, and there are $O(jMj)$ operations to perform. Consequently, we obtain the following result.

Theorem 2. *Suppose $X[1:n]$ and $Y[1:m]$ are strings, and a set M of fragments relating substrings of X and Y is given. One can use standard balanced search trees to compute the LCS from Fragments in $O(jMj \log jMj)$ time and $O(jMj)$ space for a cost measure where each insertion, deletion, or fragment segment has cost 1. When one uses Johnson's data structure, the time reduces to $O(jMj \log\log \min(jMj; nm=jMj))$.*

Proof. Correctness follows from Lemma 4 and the time and space analysis in the discussion of the algorithm. □

5 Extensions to Other Cost Measures

For the software application, the database of matches may include both exact matches (where the corresponding substrings are identical) and parameterized matches (where the corresponding substrings contain different variable names). It may desirable to assign a cost of 0 to exact matches and a cost of 1 to

parameterized matches that are not exact. It is straightforward to modify the algorithm of the previous section to allow for this modification, without changing the time bounds. Other weights may also be used as long as the cost of a fragment is less than the cost of an insertion plus a deletion; otherwise Lemma 2 would fail.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading, MA., 1983.
2. A. Apostolico. String editing and longest common subsequence. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 2*, pages 361–398, Berlin, 1997. Springer Verlag.
3. B. S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proc. 25th Symposium on Theory of Computing*, pages 71–80. ACM, 1993.
4. D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano. Sparse dynamic programming I: Linear cost functions. *J. of ACM*, 39:519–545, 1992.
5. D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano. Sparse dynamic programming II: Convex and concave cost functions. *J. of ACM*, 39:546–567, 1992.
6. M. Farach and M. Thorup. Optimal evolutionary tree comparison by sparse dynamic programming. In *Proc. 35th Symposium on Foundations of Computer Science*, pages 770–779. IEEE, 1994.
7. D. Gusfield. *Algorithms on Strings, Trees and Sequences-Computer Science and Computational Biology.* Cambridge University Press, Cambridge, 1997.
8. D.S. Hirschberg. Serial computations of Levenshtein distances. In A. Apostolico and Z. Galil, editors, *Pattern Matching Algorithms*, pages 123–142, Oxford, 1997. Oxford University Press.
9. J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. of the ACM*, 20:350–353, 1977.
10. D. B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Math. Sys. Th.*, 15:295–309, 1982.
11. J.B. Kruskal and D. Sankoff, editors. *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.
12. W. Miller and E. Myers. Chaining multiple alignment fragments in sub-quadratic time. In *Proc. of 6-th ACM-SIAM SODA*, pages 48–57, 1995.
13. E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.
14. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. *Info. Proc. Lett.*, 6:80–82, 1977.
15. M.S. Waterman. *Introduction to Computational Biology. Maps, Sequences and Genomes.* Chapman Hall, Los Angeles, 1995.

Computing the Edit-Distance Between Unrooted Ordered Trees

Philip N. Klein[?]

Department of Computer Science, Brown University

Abstract. An ordered tree is a tree in which each node's incident edges are cyclically ordered; think of the tree as being embedded in the plane. Let A and B be two ordered trees. The *edit distance* between A and B is the minimum cost of a sequence of operations (contract an edge, uncontract an edge, modify the label of an edge) needed to transform A into B. We give an $O(n^3 \log n)$ algorithm to compute the edit distance between two ordered trees.

1 Introduction

A tree is said to be *ordered* if each node is assigned a cyclic ordering of its incident edges. Such an assignment of cyclic orderings constituted a combinatorial planar embedding of the tree (and is called a *rotation system*; see [1]). Several application areas involve the comparison between planar embedded trees. Two examples are biochemistry (comparing the secondary structures of different RNA molecules) and computer vision (comparing trees that represent different shapes).

One way of comparing such trees is by their edit distance: the minimum cost to transform one tree into another by elementary operations. The edit distance between two trees can be computed using dynamic programming. This paper provides a faster dynamic-programming algorithm than was previously known.

Let A and B be ordered trees. We assume in this paper that the edges are labeled; node labels can be handled similarly. Two kinds of elementary operations are allowed: label modification and edge contraction. We assume that two subroutines (or tables) have been provided. The first subroutine, given two labels, outputs the cost of changing one label into the other. The second, given a label, outputs the cost of contracting an edge with that label. We assume that the costs are all nonnegative. The algorithmic goal is to find a minimum-cost set of operations to perform on A and B to turn them into the same tree. The left of Figure 1 gives an example.

In stating the time bounds for algorithms, we assume that the cost subroutines take constant time.

A more familiar edit-distance problem is computing *string edit-distance* [7]. The edit-distance between two strings is the minimum cost of a set of symbol-deletions and symbol-modifications required to turn them into the same string.

[?] research supported by NSF Grant CCR-9700146

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 91{102, 1998.
c Springer-Verlag Berlin Heidelberg 1998

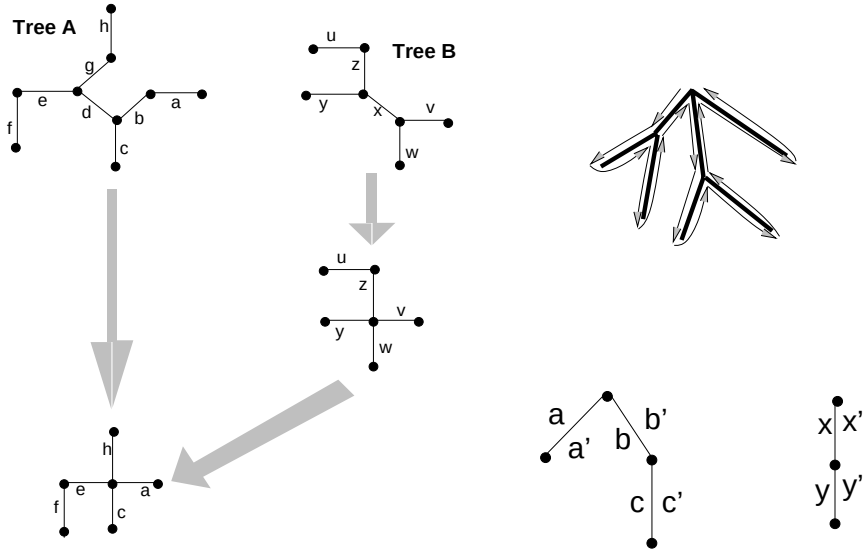


Fig. 1. The diagram on the left shows the comparison between two planar-embedded trees A and B. They can be transformed into the same tree as follows. In A, contract the edges f, d, b. In B, contract the edge x, and then change labels $u; v; w; y; z$ to $f; h; a; c; e$.

On the top-right is shown a rooted tree (in bold) and the corresponding Euler string of darts (indicated by arrows).

On the bottom-right, two small trees are shown with labeled darts. The Euler tour of the left tree is $aa^0bcc^0b^0$ and that of the second is xyy^0x^0.

Edit-distance between trees that are simple paths is equivalent to the string edit-distance problem. There is a simple $O(ab)$-time algorithm to compute the distance between a length-a string and a length-b string. Thus the worst-case time bound is $O(n^2)$.

A modification of the string edit-distance problem is the *cyclic* string edit-distance problem. In this problem, the strings are considered to be cyclic, and an algorithm must determine the best alignment of the beginnings of the two strings. One can arbitrarily fix the beginning of the first string, and try all possibilities for the beginning of the second string: for each, one computes an ordinary edit-distance. This brute-force approach reduces a cyclic edit-distance instance to n ordinary edit-distance instances, and hence yields an $O(n^3)$-time algorithm. An algorithm due to Maes [2] takes $O(n^2 \log n)$ time.

The problem of *rooted* ordered tree edit-distance has previously been considered. This problem arises in settings where, e.g., parse trees need to be compared, such as in natural language processing and image understanding.

For this problem, the input consists of two rooted trees A and B where each node's children are ordered left to right. The fastest previously known algorithm is due to Zhang and Shasha; it runs in time

$$O(|A|\,|B|\,\text{LR\_colldepth}(A)\,\text{LR\_colldepth}(B))$$

where $|T|$ denotes the size of a tree T and LR_colldepth(T) is a quantity they define, called the *collapsed depth*. Zhang and Shasha bound the collapsed depth of a tree T by

$$\min\{\text{depth of } T;\ \text{number of leaves of } T\}$$

However, in the worst case, the collapsed depth of a tree T is $\Omega(|T|)$. Thus in the worst case their algorithm runs in time $O(n^4)$ where n is the sum of the sizes of the two trees.

Unrooted ordered trees are to rooted ordered trees as cyclic strings are to ordinary strings, and it is possible to use a similar brute-force reduction from edit-distance on unrooted trees to edit-distance on rooted trees. The brute-force reduction would yield an algorithm that in the worst case required $O(n^5)$ time.

We give an algorithm that runs in $O(n^3 \log n)$ time. It solves both the rooted and the unrooted tree edit-distance problems. Thus it improves the worst-case time on rooted trees by nearly a factor of n (although depending on the input trees Zhang and Shasha's algorithm may be faster). It beats the naive $O(n^5)$-time algorithm for the unrooted case by nearly an n^2 factor.

In particular, for trees A and B, our algorithm runs in time $O(|A|^2|B| \log |B|)$. Our algorithm uses essentially the same approach as the algorithm of Zhang and Shasha. We define a variant of collapsed depth that is always at most logarithmic, and we generalize their algorithm to work with this variant. Loosely speaking, the complexity of analyzing B is thus reduced from $|B|$ LR_colldepth(B) to $|B| \log |B|$. The price we pay, however, is that the complexity of analyzing A goes from $|A|$LR_colldepth(A) to $|A|^2$. The consolation is that within this bound we can consider all possible roots of A; thus we can solve the unrooted problem within the same bounds.

1.1 Notation

For a rooted tree T, the root of T is denoted root(T). For any node v in T, the subtree of T consisting of v and its descendents is called a *rooted subtree* of T, and is denoted $T(v)$. A special case arises in which the tree is a descending path P (the first node of P is taken as the root of the tree): in this case, $P(v)$ denotes the subpath beginning at v.

For an edge e of T, we let $T(e)$ denote the subtree of T rooted at whichever endpoint of e is farther from the root of T. Note that e does not occur in $T(e)$.

Given an ordered, rooted tree T, replace each edge $\{x, y\}$ of T by two oppositely directed arcs (x, y) and (y, x), called *darts*. The depth-first search traversal of T (visiting each node's children according to their order) defines an Euler tour of the darts of T. Each dart appears exactly once. (See the top-right of Figure 1.) We interpret the tour as a string, the *Euler string* of T, and we denote this string

by $E(T)$. The first dart of the string goes from the root to the leftmost child of the root.

For a dart a, the oppositely directed arc corresponding to the same edge will be denoted a^M (here M stands for "mate") and will be called the *mate* of a.

A *substring* of a string is defined to be a consecutive subsequence of the string.

The *reverse* of a string s is denoted s^R. Thus s^R contains the same elements as s but in the reverse order. If the dart a is the first or last symbol of s, we use $s - a$ to denote the substring obtained from s by deleting a.

We use to denote the empty string, and we use $\log x$ to denote $\log_2 x$.

2 Euler Strings: Parenthesized Strings for Representing Trees

For now, we take as our goal calculating the edit-distance between two *rooted* trees. For this purpose, it is notationally and conceptually useful to represent the trees by their Euler strings. We can thus interpret the edit-distance problem on trees as an edit-distance problem on strings. However, this string edit-distance problem is not an ordinary one; each dart occurring in a tree's Euler string has a mate, and the pairing of darts affects the edit-distance calculation.

Think of each pair of darts as a pair of parentheses. The Euler string of a tree is then a nesting of parentheses. In comparing one Euler string to another, we must respect the parenthesization. Contracting an edge in one tree corresponds to deleting a pair of paired parentheses in the corresponding string. Matching an edge in one tree to an edge in the other corresponds to matching the pairs of parentheses and then matching up what is inside one pair against what is inside the other pair.[1] (See Figure 1.)

3 Comparing Substrings of Euler Strings

The subproblems arising in comparing two Euler strings involve comparing substrings of these strings.

For the purpose of comparing one substring s to another, we ignore each dart in s whose mate does not also occur in s.

We are interested in measuring the distance between substrings s and t of the Euler strings of trees A and B. The operations allowed are: delete a pair of parentheses in one of the strings (i.e. contracting the corresponding edge), and match a pair of parentheses in one string to a pair in the other (i.e. match an edge in one tree to an edge in the other).

In this subsection, we give a recurrence relation for the edit distance $\text{dist}(s; t)$ between two substrings. This recurrence relation implies a dynamic program—in

[1] Shapiro [3] compares trees by comparing their Euler strings. However, he does not seem to treat paired darts in any special way; he compares the strings using an ordinary string-edit distance algorithm. Thus he does not compute the true tree edit-distance.

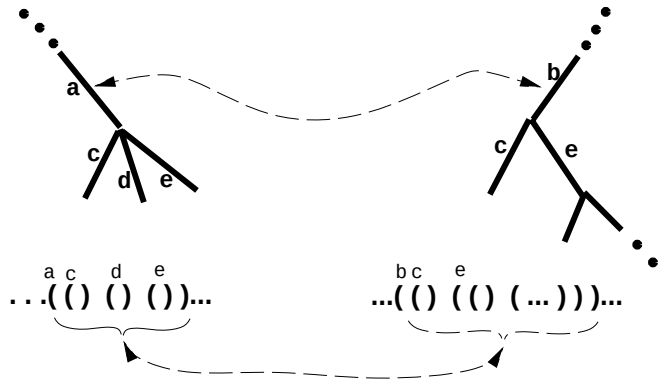


Fig. 2. Matching an edge in one tree to an edge in the other corresponding to matching a pair of parentheses in one string to a pair in the other. Note that if we are to match edge a to edge b, then we must somehow match the interior of the pair of parentheses corresponding to the edge a to the interior of the pair corresponding to b.

fact, it is a simplified (and less efficient) version of the algorithm of Zhang and Shasha[2]

We first give an auxiliary definition

match($s; t$)=
If s has the form $s_1(s_2); t_1[\![t_2]\!]$
then dist($s_1; t_1$) + dist($s_2; t_2$) + cost(change $(\ )$ to $[\![\ ]\!]$)
else 1

where the "cost" term represents the cost of changing the label of the edge in tree A corresponding to $(\ )$ into the label of the edge in tree B corresponding to $[\![\ ]\!]$.
Now we give the recurrence relation. The base case is

$$\text{dist}(\ ;\ ) = 0$$

The recursive part is

$$\text{dist}(s; t) = \min\{\text{match}(s; t),$$
$$\text{if } t = \quad \text{then } 1 \text{ else dist}(s; t - \text{last}(t)) + \text{cost(delete last dart of } t),$$
$$\text{if } s = \quad \text{then } 1 \text{ else dist}(s - \text{last}(s); t) + \text{cost(delete last dart of } s)\}$$

where the cost of the deletion is zero if the last dart's mate does not appear in the string. We use the notation $t - \text{last}(t)$ to denote the string obtained from t by deleting its last dart.

[2] Note, however, that we use notation very different from that of Zhang and Shasha. They described their algorithm in terms of (disconnected) forests induced by subsequences of the nodes ordered by preorder, whereas we use substrings of Euler strings.

As an example of how the recurrence is applied, consider the two trees at the bottom-right of Figure 1. Applying the recurrence relation to the Euler strings of these trees, we obtain dist($aa^0bcc^0b^0; xyy^0x^0$)=min$\{$dist($aa^0bcc^0; xyy^0x^0$)+cost(delete b^0), dist($aa^0bcc^0b^0; xyy^0$) + cost(delete x^0), match($aa^0bcc^0b^0; xyy^0x^0$)$\}$.

Invoking the definition of *match*, the last term is equal to dist($aa^0;\ $) + dist($cc^0; yy^0$) + cost(change bb^0 to xx^0).

The correctness of the recurrence relation is based on the following observation, which in turn is based on the fact that deletions and label modifications do not change the order among remaining symbols in a string.

Proposition 1 (Zhang and Shasha). *Consider the cheapest set of operations to transform s and t into the same string x. If the last dart of s is not deleted and the last dart of t is not deleted, then these two darts must both correspond to the last dart in x.*

The value of dist($s; t$) is the minimum over at most three expressions, and each depends on the distance between smaller substrings. Therefore, to compute the distance between Euler strings $E(A)$ and $E(B)$, we can use a dynamic program in which there is a subproblem "compute dist($s; t$)" for every pair of substrings $s; t$ of $E(A)$ and $E(B)$. The subproblems are solved in increasing order of $|s|$ and $|t|$. The number of pairs $s; t$ of substrings is $O(|A|^2|B|^2)$, and each value dist($s; t$) can be calculated in constant time. Thus the time required is $(|A|^2|B|^2)$, which is $O(n^4)$.

4 Obtaining a Faster Dynamic Program

Zhang and Shasha take advantage of the fact that not all substring pairs $s; t$ need be considered, and thereby obtain an algorithm that, depending on the input trees, can be much faster than the naive algorithm. However, in the worst case their algorithm takes $\Omega(n^4)$ time like the naive algorithm.

We modify some of their ideas to obtain an algorithm that takes $O(n^3 \log n)$ time. In this section, we present our new ingredients and the algorithm that employs them. In the next section, we will discuss how this algorithm relates to that of Zhang and Shasha.

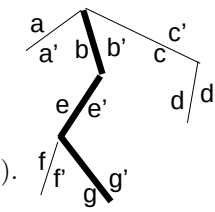
We start by presenting the ingredients, a sequence of substrings of a tree's Euler string and a decomposition of a tree into paths. Then we show how to combine these ingredients to obtain the algorithm.

4.1 Special substrings

Let T be a tree, and let P be a path starting at the root of T and descending to a leaf. We shall define a sequence of substrings of $E(T)$, called the *special substrings*. The first special substring is simply $E(T)$ itself; each subsequent substring is obtained by deleting either the first or last dart of the previous substring. It is therefore convenient to define the sequence of substrings by defining

the sequence of darts to be deleted, which we call the *difference sequence*. The recursive procedure below defines this sequence. (We use $\diamond$ to denote concatenation of sequences.)

Diff($T; P$):
  Let $r :=$ root(T).
  If r has no children then return the empty sequence
  Else let v be r's child in P.
    Let T_{left} denote the prefix of $E(T)$ ending on $(r; v)$.
    Let T_{right} denote the suffix of $E(T)$ beginning on $(v; r)$.
    Return $T_{\text{left}} \diamond T^R_{\text{right}} \diamond$ Diff($T(v); P(v)$)



Let $e_1; \ldots; e_{jE(T)j+1}$ denote the difference sequence. For example, for the tree shown to the right of the procedure, the difference sequence is $aa^0bc^0d^0dcb^0ee^0ff^0gg^0$. Now we can define the special substrings $t_0; t_1; \ldots; t_{jE(T)j+1}$ of T with respect to P. Substring t_i is obtained from $E(T)$ by deleting $e_1; \ldots; e_i$.

Lemma 1. *The sequence of special substrings t_i of T with respect to P has the following properties.*

1. *For $i = 1; \ldots; m$, the substring t_i is a substring of t_{i-1} and is shorter than t_{i-1} by one dart e_i.*
2. *Suppose e_i is an dart not on P and $e_j = e_i^M$ $(j > i)$. Then t_{i-1} is either $t_j\ e_j\ E(T(e_i))\ e_i$ or $e_i\ E(T(e_i))\ e_j\ t_j$.*
3. *For each node v of P, the string $E(T(v))$ is one of the special strings.*

Definition 1. *For a nonempty special substring t_i, define the* successor *of t_i to be t_{i+1}, and define the* difference dart *of t_i to be e_{i+1}.*

Thus the successor of t_i is obtained from t_i by deleting the difference dart of t_i. Note that the difference dart of t_i is either the leftmost or the rightmost dart occuring in t_i.

4.2 Decomposition of a rooted tree into paths

The next idea is the employment of a tree decomposition into *heavy paths*. This decomposition is used, e.g., in the dynamic-tree data structure [4]. Given a rooted tree T, define the *weight* of each node v of T be the size of the subtree rooted at v. For each nonleaf node v, let heavy(v) denote the child of v having greatest weight (breaking ties arbitrarily). The sequence of nodes

$$r; \text{heavy}(r); \text{heavy}(\text{heavy}(r)); \ldots$$

defines a descending path which is called the *heavy path*; we denote this path by $P(T)$.

Each of the subtrees hanging off of $P(T)$ has size at most $jTj=2$ (for otherwise the heavy path would enter the subtree). We recursively define the tree decomposition of T to include the path $P(T)$ together with the union of the tree decompositions of all the subtrees hanging off of $P(T)$.

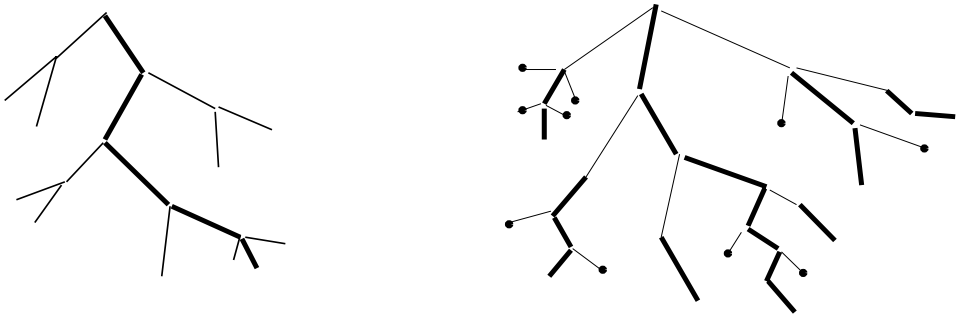


Fig. 3. In the left picture, a tree's heavy path is indicated in bold. On the right is depicted the decomposition of a tree into heavy paths and associated special subtrees. The dots indicate trivial, one-node heavy paths.

Let $P_1, \ldots, P_k$ be the descending paths comprising the tree decomposition of T, and let $r_1, \ldots, r_k$ be the first nodes of these paths. For example, r_1 is the root of T. Define the *collapsed depth* of a node v in T to be the number of ancestors of v that are members of $\{r_1, \ldots, r_k\}$.

Lemma 2 (Sleator and Tarjan, 1983). *For any node v, the collapsed depth of v is at most* $\log |T|$.

For $i = 1, \ldots, k$, let T_i be the subtree of T rooted at r_i. We call each T_i a *special subtree*. We use $P(T_i)$ to denote the heavy path P_i that starts at r_i.

4.3 Special substrings of special subtrees

For a tree T equipped with a decomposition into heavy paths, we define the *relevant* substrings of $E(T)$ to be the union, over all special subtrees T' of T, of the special substrings of T' with respect to $P(T')$.

Lemma 3. *The number of relevant substrings of T is at most* $2|T| \log |T|$

Proof. The proof consists in combining a slight modification of Lemma 7 of Zhang and Shasha [10] with our Lemma 2.

The analogue of Zhang and Shasha's lemma states that

$$\sum_{\text{special subtree } T'} |T'| = \sum_{v \in T} \text{collapsed depth of } v \tag{1}$$

To prove this equality, note that for each node v, the number of special subtrees T' containing v is the collapsed depth of v. Thus v contributes the same amount to the left and right sides.

For each special subtree T', the number of special substrings is one plus the number of darts in T', which is $1 + 2(|T'| - 1)$. Thus the total number of relevant substrings is at most the sum, over all special subtrees T', of $2|T'|$. By combining (1) with Lemma 2, we bound this sum by $2|T| \log |T|$.

Lemma 4. *For every node v of T, $E(T(v))$ (the Euler string of the subtree rooted at v) is a relevant substring.*

Proof. Every node v occurs in the heavy path P of some special subtree T^0. By part 3 of Lemma 1, $E(T^0(v))$ is a special substring of T^0, hence a relevant substring of T.

4.4 Unrooted, ordered trees

Given an unrooted, ordered tree T, let $E(T)$ denote the Euler tour of the darts of T, interpreted as a cyclic string. For each dart d, we can obtain a non-cyclic string from $E(T)$ by designating d as the starting dart of the string. Each non-cyclic string thus obtained is the Euler string of one of the *rooted* versions of T, and conversely each rooted version of T can be obtained in this way. Let $R(T)$ denote the set of these Euler strings. Note that $|R(T)| = O(|T|)$.

4.5 The new dynamic program

We finally give the new algorithm for computing the edit distance between trees A and B. Essentially the same algorithm is used for the rooted case and the unrooted case.

- If B is unrooted, root it arbitrarily.
- Find a heavy-path decomposition of B, and then identify the relevant substrings of each special subtree of B.
- By dynamic programming, calculate $\mathrm{dist}(s, t)$ for every substring s of the cyclic string $E(A)$ and every relevant substring t of B.
- For the rooted distance, output $\mathrm{dist}(\bar{s}, \bar{t})$, where $\bar{s}$ is the Euler string of the (rooted) tree A, and $\bar{t}$ is that of B.
- For the unrooted distance, output $\min_{s \in R(A)} \mathrm{dist}(s, \bar{t})$, where $\bar{t} = E(B)$. Note that the min is over all Euler strings of rooted versions of A.

For the unrooted edit-distance between A and B, we see that the algorithm arbitrarily roots B and compares it (using *rooted* edit-distance) to every rooted version of A. The correctness of this approach is intuitively evident, and has been formally proved by Srikanta Tirthapura [6]

Now we consider the analysis. The dominant step is the dynamic programming. The number of substrings s of $E(A)$ is $O(|A|^2)$. By Lemma 3, the number of relevant substrings t of B is $O(|B| \log |B|)$. We show below how each value $\mathrm{dist}(s, t)$ can be calculated in constant time from a few "easier" values $\mathrm{dist}(s^0, t^0)$. Hence the time (and space) required is $O(|A|^2 |B| \log |B|)$, which is $O(n^3 \log n)$.

We must show that the answer to every subproblem can be computed in constant time from the answers to "easier" subproblems.

Note that the recurrence relation in Section 3, whose correctness is based on Observation 1, relies on deletion of the rightmost darts of substrings. We can invoke a symmetric version of Observation 1 to justify an alternative recurrence relation based on deletion of the leftmost darts.

The ability to delete from the left gives us freedom which we exploit as follows. Our goal is to compute dist($s; t$) from a few "easier" values dist($s^0; t^0$) (i.e. where $|s^0| + |t^0|$ is smaller than $|s| + |t|$). We need to ensure that for each such value we use, the substring t^0 is a relevant substring of B. Since t is itself relevant, the successor of t is such a relevant substring t^0. However, the successor of t is either the substring obtained from t by deleting the last dart in t or the substring obtained by deleting the first dart of t.

We now give a formula for computing dist($s; t$). It is computed as the minimum of three terms.

$$\text{dist}(s; t) = \min\{\text{Delete-From-}s(s; t); \text{Delete-From-}t(s; t); \text{Match}(s; t)\}$$

We proceed to define the terms.

Delete-From-t($s; t$):
If t is the empty string, return ∞
Let e be the difference dart of t.
If e^M occurs in t, return dist($s; t - e$) + cost(delete e from t)
else return dist($s; t - e$)

Delete-From-s($s; t$):
If s is the empty string, return ∞.
If t is the empty string, let e be the rightmost dart of s
else if the difference dart of t is the rightmost dart of t
then let e be the rightmost dart of s.
else let e be the leftmost dart of s.
If e^M occurs in s, return dist($s - e; t$) + cost(delete e from s)
else return dist($s - e; t$)

Match($s; t$):
If s or t is empty, return ∞.
Let e be the difference dart of t.
If t has the form $t^0\, e^M\, t^{00}\, e$ then write s as $s^0\, e'^M\, s^{00}\, e'$.
If t has the form $e\, t^{00}\, e^M\, t^0$ then write s as $e'\, s^{00}\, e'^M\, s^0$.
Return dist($s^0; t^0$) + dist($s^{00}; t^{00}$) + cost(match e in t with e' in s)

The correctness of the formula follows from Observation 1 and its symmetric version.

It remains to verify that, for each "easier" distance subproblem dist($s^0; t^0$) appearing in the formula, t^0 is a relevant substring.

In Delete-From-t, the substring t^0 is $t - e$ where e is the difference dart of t. Hence $t - e$ is the successor of t, and is therefore relevant.

In Match, we have to check the substrings t^0 and t^{00}. Since t^{00} is the Euler string of a rooted subtree, it is relevant by Lemma 4. We use part 2 of Lemma 1 to show that t^0 is relevant. Assume t has the form $t^0\, e^M\, t^{00}\, e$ (the other case is symmetric). Say e is the i^{th} difference dart, so t is the $i - 1^{st}$ special substring, t_{i-1}. Then by part 2, t^0 is the j^{th} special substring t_j, where e^M is the j^{th} difference dart.

5 Concluding Remarks

5.1 The algorithm of Zhang and Shasha

We have presented an algorithm that solves the edit-distance problem for both rooted and unrooted ordered trees. Our algorithm has a better worse case bound than the previous algorithm for rooted trees, that of Zhang and Shasha, and is especially suitable for the unrooted case, for which Zhang and Shasha's algorithm alone is insufficient.

However, for the rooted case Zhang and Shasha's algorithm may run faster depending on the structure of the trees being compared. For this reason, it may be useful for future research to interpret their algorithm in the present framework; perhaps someone can combine the advantages of the two algorithms.

In the algorithm of Zhang and Shasha, the analogue of decomposition into heavy paths might be called decomposition into *leftmost* paths. The leftmost path descends via leftmost children. The disadvantage of this decomposition is that it does not guarantee small collapsed depth; indeed, the collapsed depth can be $\Omega(n)$.

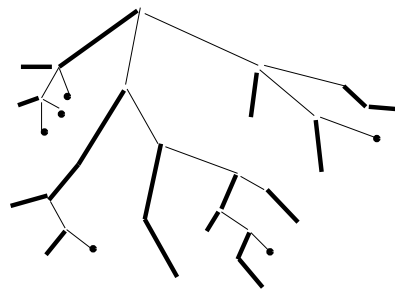


Fig. 4. The decomposition of a tree into leftmost paths is depicted. The dots indicate trivial, one-node leftmost paths.

The leftmost decomposition has a considerable benefit, however. One can define the special substrings of a subtree to be the prefixes of the Euler string of the subtree. The successor of a special substring is obtained by deleting its last symbol. The advantage is that only rightmost deletes are needed in the algorithm. For this reason, the decomposition idea can be applied to *both* trees A and B being compared, not just B. The number of subproblems is therefore $O(|A| \text{ LR\_colldepth}(A) \; |B| \text{ LR\_colldepth}(B))$ instead of $O(|A|^2 \; |B| \text{ LR\_colldepth}(B))$.

One must verify that the analogue of Lemma 1 holds for this set of special substrings. Part 1 holds trivially. Part 2 is easy to verify. Part 3 does not hold; however, because of the leftmost decomposition, something just as useful does hold. Say two substrings of $E(T)$ are *equivalent* if upon removal of unmatched darts the strings become equal. Note that we ignore such darts in computing

edit-distance; thus the edit-distance between equivalent substrings is zero. The notion of equivalence gives us a variant of part 3: for each node v in the leftmost path of a subtree T^0, there is a special substring (a prefix of $E(T^0)$) that is equivalent to $E(T(v))$.

5.2 Related problems on trees

Zhang and Shasha point out that their dynamic program can be adapted to solve similar problems, and give as examples two possible generalizations of approximate string matching to trees; these problems involve finding a modified version of a pattern tree in a text tree.

For some applications, comparison of *unordered* trees would make more sense. Unfortunately, computing edit-distance on unordered trees is NP-complete, as shown by Zhang, Statman, and Shasha [12]. Zhang has given an algorithm [9] for computing a kind of constrained edit-distance between unordered trees.

One might consider generalizing from edit-distance between ordered trees to edit-distance between planar graphs. However, the problem of finding a Hamiltonian path in a planar graph can be reduced to finding the edit-distance between planar graphs (by using the dual graph).

5.3 Acknowledgements

Many thanks to Srikanta Tirthapura for his perceptive remarks and his skepticism. He has been a great help in this research.

References

1. J. L. Gross and and T. W. Tucker, *Topological Graph Theory*, Wiley, 1987.
2. M. Maes, "On a cyclic string-to-string correction problem," *Information Processing Letters* 35 (1990), pp. 73-78.
3. B. A. Shapiro, "An algorithm for comparing multiple RNA secondary structures," *Computer Applications in the Biosciences* (1988), pp. 387-393.
4. D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of Computer and System Sciences* 26 (1983), pp. 362-391.
5. K.-C. Tai, "The tree-to-tree correction problem", *Journal of the Association for Computing Machinery* 26 (1979), pp. 422-433.
6. Srikanta Tirthapura, personal communication, 1998.
7. R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the Association for Computing Machinery* 21, (1974), pp. 168-173.
8. J. T.-L. Wang, K. Zhang, K. Jeong, and D. Shasha, "A system for approximate tree matching," *IEEE Transactions on Knowledge and Data Engineering* 6 (1994), pp. 559-571. 5
9. K. Zhang, "A constrained edit distance between unordered labeled trees," *Algorithmica* 15 (1996), pp. 205-222.
10. K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems, *SIAM Journal on Computing* 18 (1989), pp. 1245-1262.
11. K. Zhang and D. Shasha, "Approximate tree pattern matching," Chapter 14 of *Pattern Matching Algorithms*, Oxford University Press (1997)
12. K. Zhang, R. Statman and D. Shasha, "On the editing distance between unordered labeled trees," *Information Processing Letters* 42 (1992), pp. 133-139

Analogs and Duals of the MAST Problem for Sequences and Trees

Michael Fellows[1], Michael Hallett[2], Chantal Korostensky[2], and Ulrike Stege[2]

[1] Department of Computer Science, University of Victoria
Victoria, B.C. Canada V8W 3P6, mfellows@csr.uvic.ca
[2] Computational Biochemistry Research Group, E.T.H. Zürich
Ch-8092 Zürich, Switzerland {hallett,korosten,stege}@inf.ethz.ch

Abstract. Two natural kinds of problems about "structured collections of symbols" can be generally refered to as the Largest Common Subobject and the Smallest Common Superobject problems, which we consider here as the dual problems of interest. For the case of rooted binary trees where the symbols occur as leaf-labels and a subobject is defined by label-respecting hereditary topological containment, both of these problems are *NP*-complete, as are the analogous problems for sequences (the well-known Longest Common Subsequence and Shortest Common Supersequence problems). However, when the trees are restricted by allowing each symbol to occur as a leaf-label at most once (which we call a *phylogenetic tree* or *p-tree*), then the Largest Common Subobject problem, better known as the Maximum Agreement Subtree (MAST) problem, is solvable in polynomial time. We explore the complexity of the basic subobject and superobject problems for sequences and binary trees when the inputs are restricted to p-trees and p-sequences (*p-sequences* are sequences where each symbol occurs at most once). We prove that the sequence analog of MAST can be solved in polynomial time. The Shortest Common Supersequence problem restricted to inputs consisting of a collection of p-sequences (pSCS) remains *NP*-complete, as does the analogous Smallest Common Supertree problem restricted to p-trees (pSCT). We also show that both problems are hard for the parameterized complexity classes $W[1]$ where the parameter is the number of input trees or sequences. We prove fixed-parameter tractability for pSCS and pSCT when the k input sequences (trees) are restricted to be complete: every symbol of occurs exactly once in each object and the question is whether there is a common superobject of size bounded by $j \quad j+r$ and the parameter is the pair $(k; r)$. We show that without this restriction, both problems are harder than Directed Feedback Vertex Set, for which parameterized complexity is famously unresolved. We describe an application of the tractability result for pSCT in the study of gene duplication events, where k and r are naturally small.

1 Introduction

Typically algorithms used for computing phylogenetic relationships are concerned, naturally, with trees where the leaves are labeled from a set of species

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 103{114, 1998.
c Springer-Verlag Berlin Heidelberg 1998

and each label occurs at most once. However, the study of gene duplication events leads to models that make use of trees where the species labels may occur more than once.

Example 1: Modeling Gene Duplication With Repeat-Labeled Trees. When trying to resolve the species tree for a set of n taxa, one typically creates a set of k gene trees. It is not always the case that the gene trees agree. One such reason is due to paralogous duplications of genes followed by subsequent loss of genes. The papers [GCMRM79,GMS96,Z97] provide a formal model for evaluating the "cost" of rectifying the k gene trees with the correct species (phylogenetic) tree. This model implicitly makes use of trees with repeated leaf labels. For problems about sequences, we usually assume that the sequences of interest will contain occurrences of the same symbol many times. But there are some applications where attention may be restricted to sequences x where any symbol occuring in x occurs at most once.

Example 2: Scheduling Manufacturing Operations. The SHORTEST COMMON SUPERSEQUENCE problem can be applied to scheduling. Each symbol of the alphabet corresponds to an operation in the sequential process of manufacturing an object. The input to the problem corresponds to the manufacture sequences for a number of different objects that share the same production line. A common supersequence then corresponds to a schedule of operations for the production line that allows all of the different objects to be manufactured. It may reasonably be the case that each object to be manufactured requires any given operation to be applied at most once.

Restricting attention to trees or sequences without repeated symbols seems to have a significant effect on problem complexity. In order to discuss the issue we introduce the following terminology. All trees in this paper are binary unless stated otherwise. A *p-tree* is a rooted tree where the leaves are labeled from an alphabet , and where no symbol in is used more than once as a label. An *rl-tree* is a rooted tree with leaves labeled from , where labels may be repeated. Unless stated otherwise n is used to denote the size of , j $j = n$. Say that a string of symbols (or sequence) x 2 is a *p-string* (*p-sequence*) if no symbol of occurs more than once in x. We call x a *complete* p-sequence if each symbol of the alphabet occurs exactly once in x. To emphasize the analogy with trees, say that x is an *rl-string* if it is a string in the usual sense, where symbols of may be repeated. If x and y are strings, then we write x y to denote that x is a subsequence of y. If X and Y are rl-trees, then X Y denotes that X is contained in Y by topological containment that respects ancestry with label isomorphism at the leaves. For a given tree T, the *size* jTj of T is defined by the number of leaves in T.

The following quartet of fundamental computational problems are natural and important for various applications.

Two Problems About Trees

LARGEST COMMON SUBTREE (LCT)

Input: rl-trees $T_1; :::; T_k$ and a positive integer m

Question: Is there an rl-tree T of size jTj m leaves, with T T_i for $i = 1; :::; k$?

SMALLEST COMMON SUPERTREE (SCT)
Input: rl-trees $T_1; :::; T_k$ and a positive integer m
Question: Is there an rl-tree T of size $jTj \quad m$ leaves, with $T_i \quad T$ for $i = 1; :::; k$?

Two Problems About Sequences

LONGEST COMMON SUBSEQUENCE (LCS)
Input: rl-sequences $x_1; :::; x_k$ and a positive integer m
Question: Is there an rl-sequence x, with $jxj \quad m$ and $x \quad x_i$ for $i = 1; :::; k$?

SHORTEST COMMON SUPERSEQUENCE (SCS)
Input: rl-sequences $x_1; :::; x_k$ and a positive integer m
Question: Is there an rl-sequence x, with $jxj \quad m$ and $x_i \quad x$ for $i = 1; :::; k$?

All four of these problems are *NP*-complete (see [M78] for the sequence problems; we prove *NP*-completeness for the tree problems in Section 3). Both LCS and SCS can be solved in $O(n^k)$ time [M78]. We prove that LCT and SCT, the tree analogs of LCS and SCS, have similar polynomial complexity for fixed k (Section 3). The question we explore is

What happens to the complexity of these basic problems when the inputs are restricted to be p-trees and p-sequences?

The LARGEST COMMON SUBTREE problem restricted to p-trees is better known as the MAXIMUM AGREEMENT SUBTREE (MAST) problem. This is an important problem for which useful polynomial-time algorithms (for trees of bounded degree) have recently been developed. In the study of evolution the situation frequently arises that one has k sets of gene sequence data for n species. A typical approach would be to compute *gene trees* for each of the data sets. For various reasons, these trees will frequently not be in agreement. One may then use an algorithm for MAST to compute a maximum subtree (*species tree*) on which the gene trees agree. Farach et al. have described an algorithm for MAST for k rooted p-trees for n species of maximum degree d that runs in time $O(kn^3 + n^d)$ [FPT95]. A different (and simpler) algorithm with the same time complexity was developed by Bryant [B97]. Przytycka has described a modification of the algorithm of [FPT95] that runs in time $O(kn^3 + k^d)$ [P97].
The above described results on MAST answer our question for one of the four basic problems, but what of the other three?

Main Results. (1) We describe an $O(j \quad j^2)$ algorithm to solve the LONGEST COMMON SUBSEQUENCE problem for inputs restricted to p-sequences. (2) We prove that SHORTEST COMMON SUPERSEQUENCE and SMALLEST COMMON SUPERTREE for inputs restricted to p-sequences and p-trees remain *NP*-complete, and are hard for $W[1]$ when the parameter is the number of input objects. (3) We prove fixed-parameter tractability for SHORTEST COMMON SUPERSEQUENCE for p- and rl-sequences when the question is whether there is a common supersequence of length at most $j \quad j + r$ parameterized by the number of sequences and r. We present an algorithm with running time $O(k^r \quad)$. (4) We prove fixed-parameter tractability for SHORTEST COMMON SUPERSEQUENCE for complete p-sequences and parameter r when the question is whether there is a common supersequence of length $j \quad j+r$. (In fact, we prove tractability for a more general, weighted alphabet form of the problem.) (5) We prove fixed-parameter tractabil-

ity for SHORTEST COMMON SUPERTREE for k complete p-trees and parameter is the pair $(k; r)$, when the question is whether there is a common supertree of size $j \quad j + r$. Readers not familiar with parameterized complexity are referred to [DF93,DF95a,DF95b,DF95c,DF98].

2 Sequence Analogs of MAST

We consider the following analog of MAST for p-sequences. While not difficult, the positive result shows an interesting parallel between the complexity of sequence problems and the complexity of leaf-labeled tree problems that we will see holds up also for the superobject problems.

LONGEST COMMON SUBSEQUENCE FOR p-SEQUENCES (pLCS)
Input: p-sequences $x_1; :::; x_k$ over an alphabet , and $m \; 2 \; Z^+$.
Question: Is there a p-sequence x, $jxj \quad m$, such that $x \quad x_i$ for $i = 1; :::; k$?
Theorem. pLCS can be solved in time $O(k \quad n \log n)$ for an alphabet of size n.
Proof (sketch): The basic structure of the algorithm is based on the partial ordering of the elements of defined: $a \quad b$ if and only if $8i$: a precedes b in x_i. (Note that the input being p-sequences is crucial to having this be well-defined.) We simply compute a longest chain in this poset by topological sorting. Including the time to compare elements in the poset, this can be accomplished in time $O(k \quad n \log n)$ using the appropriate data structures. ∎

3 Tree Analogs of SCS and LCS

The complexity situation for the tree analogs SCT and LCT of the sequence problems SCS and LCS turns out to be almost identical to the situation for these well-studied problems. Both SCT and LCT are *NP*-complete, but can be solved in time $O(n^{k+1})$ for k trees by dynamic programming. A straightforward reduction from LCS encoding sequences as caterpillar trees proves the *NP*-completeness of LCT. An $O(n^{k+1})$-time algorithm for LCT can be given by dynamic programming. Both proofs are omitted.

Theorem 1. *LCT restricted to binary trees is* NP*-complete and, for each fixed k, it can be solved in time $O(n^{k+1})$.*

Theorem 2. *SCT restricted to binary trees is* NP*-complete.*

Proof (sketch): We reduce from the *NP*-complete problem DIRECTED FEEDBACK VERTEX SET (DFVS) (see [GJ79]) in two steps. In the first step, we reduce to the SCS problem. In the second step, we transform the SCS instance into an instance of SCT. The DFVS problem takes as input a directed graph $D = (V; A)$ and a positive integer k and asks whether there is a set $V^0 \quad V$ with $jV^0j \quad k$ such that $D - V^0$ is acyclic. (As a parameterized problem, we take k to be the parameter.)

Let $D = (V; A)$ be a digraph for which we wish to determine if it has a directed feedback vertex set of size k. The instance of SCS to which we transform this is described as follows. The alphabet is V. Each arc uv of D becomes the length 2 sequence uv. The set of sequences S thus has the same size as the set A of arcs of D. The parameter k^0 of the image instance is equal to k.
Let x be a solution for the sequence problem, and let V^0 be the set of vertices (which double as the symbols of the alphabet) that occur more than once in x. Clearly $jV^0j \quad k$. We argue that V^0 covers all the directed cycles in D. If not, then there is a directed cycle C in D involving vertices that occur exactly once in x. Let a be the first vertex of C that occurs in x. But then there is some vertex b of C such that ba is an arc of C and therefore ba is a sequence in S. Since b occurs after a in x this contradicts that x is a solution to the SCS problem.
Conversely, let V^0 be a k-element directed feedback vertex set for D.
A common supersequence x for the sequences in S can be written $x = x_1x_2x_3$ where $x_1 = x_3$ is any permutation of V^0 and x_2 is a topological sort of $V - V^0$. If uv is an arc of D where both u and v belong to V^0, then clearly uv is a subsequence of x, either by finding u in x_1 and v in x_3 or vice versa. If both u and v belong to $V - V^0$ then uv is a subsequence of x_2. The two remaining cases (where exactly one of u and v belongs to V^0) are equally easy to check.

The second step of theorem transforms an instance of SCS to SCT. An instance of SCS consists of a set X of sequences over an alphabet and a positive integer l. A sequence $x_i \; 2 \; X$ of length t, where x_i consists of the sequence of symbols of $x_i = x_i[1] \quad x_i[t]$ is transformed to a tree T_i represented by the parenthesized expression $(((\quad((F; x_i[1]); x_1[2]) \quad)x_i[t]); \$)$ where F is a complete binary tree having more than l leaves labeled from a set of new symbols that we will also denote F, and where \$ is another new symbol. The alphabet of leaf labels of the trees is thus $^0 = \quad [F [f\$g$. The theorem follows from Lemma 1, for which the proof is omitted. ∎

Lemma 1. *If X is a set of p-sequences $X = fx_i : 1 \quad i \quad mg$ over an alphabet , $j \; j = n$, where each symbol of occurs in at least one sequence, r is a positive integer, and $T = fT_i : 1 \quad i \quad mg$ is the set of p-trees that are the images of the transformation from* SCS *to* SCT, *^{0} is the transformed alphabet, $j \; ^0j = n^0$, then there is a common supersequence x of the sequences in X of length at most $n + r$ if and only if there is an rl-tree T having size at most $n^0 + r$ that is a common supertree for the trees in T.*

We remark that part 1 above gives a much simpler proof of the *NP*-completeness for SCS than [M78].

Theorem 3. *For fixed k, SCT on binary trees can be solved in time $O(n^{k+1})$.*

Proof (sketch): Using dynamic programming we can calculate, for each coordinate k-tuple of vertices $c = (u_1; :::; u_k)$, where u_i is a vertex of T_i for $i = 1; :::; k$, the number of leaves $l(c)$ (and a representative, for the search algorithm) of a smallest common binary supertree for the subtrees rooted at the vertices u_i. A solution for the given input can be found in the resulting table at the coordinate

k-tuple of roots of the T_i.
The case of $k = 2$ is instructive and generalizes easily to larger values of k. Suppose the vertices u_i of T_i, for $i = 1;2$ have children u_i^L and u_i^R. The basis for the dynamic programming recurrence is the observation that if T is a common supertree of T_1 and T_2, then there are essentially 7 different ways that the T_i can be embedded in a binary supertree T: (1) with T_1 embedded entirely in one branch of T and T_2 embedded entirely in the other branch, giving $l(u_1;u_2) = l(u_1;;) + l(u_2;;)$, (2) with T_1 and the "half" of T_2 rooted at u_2^L embedded in one branch of T and the other half of T_2 embedded in the other branch of T, giving $l(u_1;u_2) = l(u_1;u_2^L) + l(;;u_2^R)$, (3-5) similarly to (2), (6,7) with half of each of the T_i embedded in each subtree of T, giving $l(u_1;u_2) = l(u_1^L;u_2^L) + l(u_1^R;u_2^R)$ or $l(u_1;u_2) = l(u_1^L;u_2^R) + l(u_1^R;u_2^L)$. The value of $l(u_1;u_2)$ is obtained as the minimum over these 7 possibilities. ∎

4 The Shortest Common Supersequence Problem for p-Sequences

We show $W[1]$-hardness for pSCS parameterized by the number of input sequences. Furthermore we show that the problem is fixed parameter tractable if we allow the supersequence to be a bit longer (by parameter r) than the size of the alphabet. We also give an *FPT* algorithm for the version of the problem where we only parameterize by r and have complete p-sequences as input.

4.1 Hardness Results

We define the following problem

p-Sequence Shortest Common Supersequence (pSCS I)
Input: p-sequences $x_1;:::;x_k$ and a positive integer M
Parameter: k. *Question:* Is there an rl-sequence x, with jxj M and x_i x for $i = 1;:::;k$?

Theorem 4. pSCS I *is (1) hard for* $W[1]$ *and (2)* NP*-complete.*

Proof (sketch): We reduce from the Clique problem which is proven $W[1]$-complete in [DF95b,DF98] when parameterized by the size of the clique set. The proof is rather lengthy and technical so we have chosen not to include it here. Interested readers are referred to [FHKS98] for the complete details. ∎

4.2 Tractable Parameterizations

We say that a sequence S contains r *duplication events* if S is not a p-sequence but the exactly r symbols need be removed from S to result in a p-sequence. We define a *duplication event* for trees in a similar manner except we must remove r leaves which result in a tree homeomorphic to a p-tree. We define the following problem

Bounded Duplication SCS I for p-Sequences (BDSCS I)
Input: A family of k p-sequences x_i 2 for $i = 1;:::;k$ and a positive integer r representing the number of duplication events. We assume that j $j = n$ and that each symbol of occurs in at least one of the input sequences.
Parameter: $(k; r)$
Question: Is there a common rl-supersequence x of length at most $n + r$?

Theorem 5. Bounded Duplication SCS I for p-Sequences *is fixed-parameter tractable.*

Proof (sketch): We describe the argument in terms of a one-person game that provides a convenient representation of a supersequence of a set of p-sequences. At each move of the game, a symbol a 2 is chosen, and those sequences in the collection that have a as their first symbol are modified by deleting the first symbol. We will call this an a-move. The game is completed when all of the symbols have been deleted from all of the sequences. It is straightforward to verify that a set of p-sequences has a common supersequence x of length m if and only if the game can be completed in m moves, where the sequence of symbols chosen for the moves of the game is x. When the game is played for the set of sequences x_i we may refer to the *current* x_i, meaning the suffix of x_i that remains at an understood (current) point in the game.
Suppose the symbol a is chosen for a move of the game. For each of the sequences x_i, one of the following statements must be true: (1) The symbol a is the first symbol of the current x_i and does not otherwise occur in the current x_i. (2) The symbol a does not occur in the current x_i. (3) The symbol a occurs in the current x_i, but is not the first symbol. If for an a-move of the game only (1) and (2) occur, we call this a *good* a-move. A move that is not good is *bad*. Our algorithm is based on the following claims (proofs omitted).
Claim 1. If for the input to the problem, a game is played that involves at least r bad moves, then the corresponding common supersequence x has length at least $n + r$. *Claim 2.* For any yes-instance of the problem, there is a game that can be completed with at most r bad moves.
We can now describe an *FPT* algorithm based on the method of search trees [DF95c]. By Claim 2, if the answer is "yes" then there is a game that completes with no more than r bad moves.
Algorithm. (0) The root node of the search tree is labeled with the given input. (1) A node of the search tree is expanded by making a sequence of good moves (arbitrarily) until no good move is possible. For each possible nontrivial bad move (i.e., one that results in at least one deletion), create a child node labeled with the set of sequences that result after this bad move. (2) If a node is labeled by the set of empty sequences, then answer "yes". (3) If a node has depth r in the search tree, then do not expand any further.
The correctness of the algorithm follows from Claim 2 and the following, which justifies that the sequence of good moves in (1) can be made in any order without increasing the length of the game.

Claim 3. If two different good moves, an a-move and a b-move, for $a; b$ 2 , are possible at a given point in a game, then there is a completion of the game in a total of l moves including m bad moves and starting with the a-move if and only if there is a completion of the game in a total of l moves including m bad moves and starting with the b-move. The proof is omitted due to space limitations. The running time of the algorithm is bounded by $O(k^r \quad n)$. ∎

It is easy to see that we can extend this *FPT*-algorithm to the case that the input consists of rl-sequences instead of p-sequences.

Corollary 1. Bounded Duplication SCS I *is fixed-parameter tractable.*

We next consider the question of what happens to the complexity of the above problem if the parameter is considered to be just r, rather than the pair $(k; r)$. If we restrict the input sequences to be complete, then we can show fixed parameter tractability for a more general weighted problem. Indeed, as is frequently the case for *FPT* algorithms, solving this more general problem seems to be the key to proving tractability for the unweighted form of the problem.

Bounded Duplication SCS II for Complete p-Sequences (BDCSC II)
Input: Complete p-sequences x_i over an alphabet of size n, a positive integer r, and a cost function c : ! Z^+.
Parameter: r
Question: Is there a common supersequence x of duplication cost $jjxjj_c$ r where the duplication cost is defined as $jjxjj_c = \sum_{a2} (n_a(x) - 1)c(a)\ n_a(x),\ a\ 2$, denotes the number of occurrences of symbol a in x.

Theorem 6. Bounded Duplication SCS II for Complete p-Sequences *is fixed-parameter tractable.*

Proof (sketch): Say that two symbols $a; b$ 2 are *combinable* if ab is a substring of each x_i. We argue that if $j \quad j > 3r + 1$ then either there is a combinable pair of symbols or the answer is "no". Suppose the answer is "yes" and let x be any solution. Let x^0 be the subsequence of x consisting of those symbols of that are not repeated in x. There is a natural factorization of x^0 into at most $2r + 1$ substrings of x. Let ^{0} be the set of symbols occuring in x^0. Then $j \quad ^0 j > 2r + 1$ and at least one of the substrings of the factorization of x^0 must have length at least two. Let ab denote a substring of x of two symbols in ^{0}. We argue that a and b are combinable. If not, then in at least one of the x_i we do not have the substring ab, and since the x_i are complete, one of the following must hold: (1) x_i contains the substring ba, (2) x_i contains a subsequence acb, or (3) x_i contains a subsequence bca. It is easy to check that in each of these cases it is impossible for x to be a supersequence of x_i.
The algorithm is described as follows, based on the method of problem kernels. The argument above shows that we can efficiently determine a reason to answer "no" (if $j \quad j > 3r + 1$), or we can find a combinable pair ab of symbols. If we find a combinable pair, then we replace the substring ab in each x_i with a new

symbol s_{ab} and give this symbol the duplication cost $c(s_{ab}) = c(a) + c(b)$. Given that the answer to this modified input is "yes" if and only if the answer for the original input is "yes", this gives us a way of reducing to a problem kernel in time $O(N^2)$ where N is the total size of the input to the problem. The proof of correctness is omitted due to space limitations. ∎

It would be very nice to settle the parameterized complexity of Bounded Duplication SCS III for p-Sequences (BDSCS III), defined in exactly the same way as BDSDS I expect that the parameter is simply r rather than the pair $(k; r)$. The proof of Theorem 2 shows the following

Theorem 7. *If BDSCS III is fixed parameter tractable, then so is DFVS.*

The interest of this theorem is that DFVS has resisted a number of attempts to settle its parametric complexity and is considered a major open problem [DF98], with most expert opinion favoring the conjecture that it is in *FPT*.

5 The Smallest Common Supertree Problem for Phylogenetic Trees

In computational biology the question arises how to resolve the species tree for a given set of gene trees such that the number of paralogous duplications is minimized. A *species tree* (or phylogenetic tree) is a rooted binary leaf labeled tree where the labels correspond to a set of taxa. Ma et al. [MLZ98] consider this question under the cost model introduced by Goodman et. al. [GCMRM79] and prove that the problem is *NP*-complete if the gene trees are rl-trees. An easy reduction leads to the *NP*-completeness of the same problem when the gene trees in the input are p-trees (proof omitted).

Optimal Species Tree I for p-Trees (pOST I)
Input: p-trees $G_1; \ldots; G_k$
Question: Does there exist a species tree S with minimum duplication cost $\sum_{i=1}^{k} m(G_i; S)$? (Here $m(G_i; S)$ is the number of duplications under the least common ancestor mapping of the gene tree G in the species tree S [MLZ98].)

Theorem 8. *pOST I is NP-complete.*

We consider in this section the problem of finding the smallest common supertree for a given set of p-trees (pSCT). We could interpret the resulting rl-tree in the following sense. A duplication node d in an rl-tree R is an internal node of R where the subtrees $R_1; R_2$ of R induced by the children $d_1; d_2$ of d have leaf labels in common. For given gene trees $G_1; \ldots; G_k$ we ask for the smallest rl-tree containing $G_1; \ldots; G_k$ as subtrees. Via the duplication nodes we can extract a p-tree (species tree) S which "explains" $G_1; \ldots; G_k$ in terms of the duplication nodes. That S is an approximation for pOST I is a consequence of the following consideration. The pSCT-problem asks for an rl-tree R containing $G_1; \ldots; G_k$. The minimum size of R gives a lower bound on the number of gene duplications. To see this, note that there can exist gene trees $G_i; G_j; i \neq j;$ using the same

duplication node, although two different gene duplications are represented. (We say a duplication node is *used* if the embedding of the gene tree in the rl-tree contains subtrees of the gene trees in both subtrees of the duplication node.) The exact number of gene duplications for $G_1;:::;G_k$ and R is $\sum_{i=1}^{k}$(minimum number of used nodes for an embedding G_i is embedded in R). Although pSCT is $W[1]$-hard (Theorem 9) the problem is fixed parameter tractable for complete p-trees if we ask for a common supertree of size at most $n+r$ for parameter pair $(k;r)$.

5.1 Hardness Results

We define the following problem

Shortest Common Supertree for p-Sequences (pSCT I)
Input: binary p-trees $T_1;:::;T_k$ and a positive integer m
Parameter: k.
Question: Is there an rl-tree T, with jTj m and T_i T for $i = 1;:::;k$?

Noting that the construction in the second part of Theorem 2 also applies to pSCT I, we receive the following theorem.

Theorem 9. pSCT I *(1) hard for* $W[1]$ *and (2)* NP*-complete.*

5.2 Tractable Parameterizations

The parameterization of SCT that we next consider is defined as follows.

Bounded Duplication SCT for binary p-Trees (BDSCT)
Input: A family of k complete binary p-trees T_i with leaf label set , j $j = n$, and a positive integer r representing the number of duplication events.
Parameter: $(k;r)$
Question: Is there a common binary supertree T of the T_i of size at most $n+r$?

As discussed above the definition is reasonable for applications in the study of gene duplication events in the sense that both k and r may be small and the input trees complete when complete sequence data is available for all of the species under consideration.

Theorem 10. Bounded Duplication SCT for binary p-Trees *is fixed parameter tractable.*

Proof (sketch): The algorithm is based on a combination of the methods of search trees and reduction to a problem kernel described in [DF95c,DF98]. To a given tree T_i and a 2 , we associate an ordered partition $_i(a)$ of $-fag$ in a natural way by considering the path $_i(a) = (r_i = v_0;v_1;:::;v_s = l)$ in T_i from the root r_i or T_i to the leaf l labeled by a. Since the tree is binary, each vertex of this path other than l has another child v_j^0 for $j = 0;:::;s-1$. Let $T_i(j)$ denote the subtree rooted at v_j^0 and let $L_i(j)$ denote the set of leaf labels of the subtree $T_i(j)$. Then $_i(a) = (L_i(0);:::;L_i(s-1))$.
We say that a 2 is *good* if for all $i;i^0$, 1 $i;i^0$ k, $_i(a) = _{i^0}(a)$, and

otherwise we say that a is *bad*. Note that if a is good then there is an obvious possibility of breaking the problem down into subproblems, since for each class of the ordered partition (a) each T_i yields a subtree having precisely this set of leaf labels. We will refer to this as the subproblem *induced* by the class of the ordered partition.

The algorithm proceeds by attempting to find a good $a\ 2$ (this is easy by just computing and comparing the partitions), and if a good a is found, then branching in a search tree by trying all possibilities for allocating the "budget" of k duplications among the nontrivial subproblems induced by classes of (a). Here by *nontrivial* we mean the situation where the induced subtrees of the subproblem are not all isomorphic, so that at least one duplication event is required for a common supertree. Note that determining whether a subproblem is trivial is easy. If there are more than k nontrivial subproblems then we answer "no".

We continue in this way to build a search tree until there are no more good symbols that allow further breakdowns of the problem. When the relevant subproblems are *small* (bounded in size by an appropriate function of k and r) then we answer by exhaustive search. The correctness and fixed parameter tractability of the procedure follows from the following two claims.

Claim 1. If there is a good $a\ 2$ then an optimal solution can be computed by solving the subproblems and gluing these together along the path (a). The proof of this claim is nontrivial and omitted for reasons of length.

Claim 2. There is a function $f(r)$ such that if $j\ \ j > f(r)$ and $k\ \ 2$ and all $a\ 2$ are bad, then the answer is "no".

The proof of this claim makes use of Bunemann's Theorem [B71] on the reconstructibility of p-trees from triples, and on being able to find more than k disjoint sets of conflicting triples when $j\ \ j$ is large and all $a\ 2$ are bad. ∎

6 Open Problems

Our main contribution has been to explore the complexity of basic problems about sets of symbols structured by binary trees and sequences. These are related to the MAST problem, which has an important, nontrivial polynomial-time algorithm, a situation that suggests that the other three problems that are naturally similar to MAST might also be solvable in polynomial-time. One of these, the sequence analog of MAST, we have shown to be polynomial, while the problems dual to MAST concerning common supersequences and supertrees remain NP-complete. One interesting aspect of our results are the strong parallels exhibited between the complexity of tree and sequence problems.

A number of open problems remain. In particular: (1) The fixed parameter tractability results are presently slightly stronger for sequences. We conjecture that tree analogs of Theorems 5 and 6 should hold. (2) Does Theorem 6 still hold if the input sequences are not required to be complete? This seems to be a reasonable conjecture, but we have shown that this would imply that Directed Feedback Vertex Set is in *FPT*, also a reasonable conjecture, but apparently a difficult one [DF98]. (3) Because in biological applications the p-sequences and

p-trees are often complete, the complexity of the four basic problems LCS, SCS, LCT, and SCT for complete p-objects are interesting.
(4) Can a guaranteed performance bound be establish for the application of Theorem 10 to problems of paralogous gene duplications discussed in Section 5? A possible heuristic for computing parsimonious annotations is to: (i) Compute an rl-supertree T of minimum size using the algorithm of Theorem 10, and then (ii) Extract an annotated p-tree from T.

References

B97. D. Bryant. "Building Trees, Hunting for Trees, and Comparing Trees – Theory and Methods in Phylogenetic Analysis," Ph.D. Thesis. Department of Mathematics. University of Canterbury, 1997.

B71. P. Buneman. "The Recovery of Trees from Measures of Dissimilarity," In F. R. Hodson, D. G. Kendall, P. Tautu, editors, *Mathematics in the Archaeological and Historical Sciences*, pp. 387–395. Edinburg University Press, Edinburgh, 1971.

DF93. R. Downey and M. Fellows. "Fixed Parameter Tractability and Completeness III: Some Structural Aspects of the W-Hierarchy," in: K. Ambos-Spies, S. Homer and U. Schöning, editors, *Complexity Theory: Current Research*, Cambridge Univ. Press (1993), 166–191.

DF95a. R. G. Downey and M. R. Fellows. "Fixed Parameter Tractability and Completeness I: Basic Theory," *SIAM Journal of Computing* 24 (1995), 873-921.

DF95b. R. G. Downey and M. R. Fellows. "Fixed Parameter Tractability and Completeness II: Completeness for $W[1]$," *Theoretical Computer Science A* 141 (1995), 109-131.

DF95c. R. G. Downey and M. R. Fellows. "Parametrized Computational Feasibility," in: *Feasible Mathematics II, P. Clote and J. Remmel (eds.)* Birkhauser, Boston (1995) 219-244.

DF98. R. G. Downey and M. R. Fellows. *Parameterized Complexity*, Springer-Verlag, 1998.

FPT95. M. Farach, T. Przytycka, and M. Thorup. "On the agreement of many trees" *Information Processing Letters 55* (1995), 297–301.

FHKS98. M. R. Fellows, M. T. Hallett, C. Korostensky, U. Stege. "The complexity of problems on sequences and trees." Technical Report, ETH-Zurich, 1998.

GJ79. M. R. Garey and D. S. Johnson. " Computers and Intractability: A Guide to the Theory of NP-Completeness," W. H. Freeman, San Francisco, 1979.

GCMRM79. M. Goodman, J. Czelusniak, G. W. Moore, A. E. Romero-Herrera and G. Matsuda. "Fitting the Gene Lineage into its Species Lineage: A parsimony strategy illustrated by cladograms constructed from globin sequences," *Syst. Zool.*(1979), 28, 132–163.

GMS96. R. Guigó, I. Muchnik, and T. F. Smith. "Reconstruction of Ancient Molecular Phylogeny," *Molecular Phylogenetics and Evolution* (1996),6:2, 189–213.

MLZ98. B. Ma, M. Li, and L. Zhang. "On Reconstructing Species Trees from Gene Trees in Term of Duplications and Losses," *Recomb 98*, to appear.

M78. D. Maier. "The Complexity of Some Problems on Subsequences and Supersequences," *J. ACM*, 25,2(1978), 322–336.

P94. R. D. M. Page. "Maps between trees and cladistic analysis of historical associations among genes, organisms, and areas," *Syst. Biol. 43* (1994), 58–77.

P97. T. Przytycka. private communication, 1997.

Z97. L. Zhang. "On a Mirkin-Muchnik-Smith Conjecture for Comparing Molecular Phylogenies," *Journal of Computational Biology* (1997) 4:2, 177–187.

Complexity Estimates Depending on Condition and Round-O Error

Felipe Cucker and Steve Smale

Department of Mathematics
City University of Hong Kong
83 Tat Chee Avenue, Kowloon
HONG KONG
fmacucker,masmaleg@math.cityu.edu.hk

1 Introduction

Solving systems of polynomial equations by computer has become a principal task in many areas of science. In most of this usage of the machine, there is a loss of precision at many steps of the process. For example a round-off error occurs when two numbers with 10 digit precision are multiplied to obtain a new number with 10 digit precision. The main goal of this paper is to give a theoretical foundation for solving systems of equations (with inequality constraints permitted as well) which takes into account this loss of precision. We give an algorithm with reasonable complexity estimates for solving general real polynomial systems of equalities and inequalities and which succeeds in the presence of round-off error, provided the precision of the computation satisfies a bound polynomial in some parameters (to be described). Moreover we strive to give some foundations for the round-off error in the setting of real number machines as in [4]. Modifying these "real Turing machines" to include round-off error helps to make a more compelling case for their use in the foundations of Numerical Analysis, developing a suggestion in [20].

Our Main Theorem (Theorem 2 below) states that the feasibility of any system of equalities and inequalities in n variables can be decided in time polynomial in the condition number of the input system and singly exponential in n. Moreover, a polynomial bound is given for the required precision.

Algorithms for deciding the feasibility of systems of polynomial equations and inequalities exist after the work of Tarski [21]. In more recent times new algorithms were devised to improve Tarski's since the latter has hyperexponential complexity. In the seventies, Collins [7] devised an algorithm whose complexity has a doubly exponential dependence on n. A breakthrough was made later by Grigoriev and Vorobjov [10] with the introduction of the *critical points method* and an algorithm working in single exponential time. This algorithm is sequential and only works for systems of polynomials with rational coefficients. The cost measure considered is the bit cost. Subsequent developments appear in [11,14,3]. These articles provide algorithms which work also with arbitrary real numbers at unit cost and can be efficiently parallelized.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 115–126, 1998.
c Springer-Verlag Berlin Heidelberg 1998

All these algorithms assume that the computations are exact, i.e. no round-off is produced during the computation. The ones using the critical points method (i.e. those working in single exponential time) have a fairly complicated description and to prove their correctness requires deep arguments from differential topology. In contrast, the algorithm in the Main Theorem is quite simple to describe and works under round-off computations for all well-posed inputs.

2 Feasibility of Homogeneous Sparse Polynomial Systems

We discuss the homogeneous algebraic case first and subsequently show how this can be used in the affine semi-algebraic case in the Theorem 2.

Let $N > 0$ and $\mathbb{R}^N$ be the Euclidean space of dimension N. The norm $\|\ \|_2$ in $\mathbb{R}^N$, which is the norm associated to the dot product in $\mathbb{R}^N$, is defined by

$$\|x\|_2 = \sqrt{\sum_{i=1}^{N} x_i^2}.$$

Another norm on $\mathbb{R}^N$ is $\|x\|_1 = \max_{1 \le i \le N} |x_i|$. In this paper we shall consider both norms. If no confusion is possible, we denote $\|x\|_2$ simply by $\|x\|$.

Denote by $B(\mathbb{R}^N)$ the unit box in $\mathbb{R}^N$, that is

$$B(\mathbb{R}^N) = \{x \in \mathbb{R}^N \mid \max_{1 \le i \le N} |x_i| = 1\}.$$

Recall that a function $f : \mathbb{R}^{n'+1} \times \mathbb{R}^{n+1} \to \mathbb{R}$ is *bihomogeneous* of degrees c and d when $f(\lambda a, \mu x) = \lambda^c \mu^d f(a, x)$ for any $(a, x) \in \mathbb{R}^{n'+1} \times \mathbb{R}^{n+1}$ and $\lambda, \mu \in \mathbb{R}$.

Definition 1. $[H_{(c;d)}]$ Let $f_1, \ldots, f_m \in \mathbb{Z}[a_0, \ldots, a_{n'}, x_0, \ldots, x_n]$ be bihomogeneous polynomials of degrees $c_1, \ldots, c_m$ in the a's and of degrees $d_1, \ldots, d_m$ in the x's respectively. Let $f = (f_1, \ldots, f_m)$. For a given $a \in \mathbb{R}^{n'+1}$ consider these polynomials as polynomials in the variables $x_0, \ldots, x_n$ with coefficients in $\mathbb{Z}[a_0, \ldots, a_{n'}] \subset \mathbb{R}$ and denote this system by $f_a : \mathbb{R}^{n+1} \to \mathbb{R}^m$ so that $f_a(x) = f(a, x)$. We say that a pair (f, a) is *feasible* (or that f_a is feasible) if there is a point $x \in \mathbb{R}^{n+1}$, $x \neq 0$, such that $f_a(x) = 0$.

We call $f = (f_1, \ldots, f_m)$ a *sparse polynomial system*. Also, given $c = (c_1, \ldots, c_m) \in \mathbb{N}^m$ and $d = (d_1, \ldots, d_m) \in \mathbb{N}^m$ we shall denote the set of all sparse systems by $H_{(c;d)}$.

Remark 1. Let $D = \max\{d_1, \ldots, d_m\}$. In the rest of this paper we will assume that $D \geq 2$.

Example 1. The dense case may be thought of as a special case. Consider $f = (f_1, \ldots, f_m)$ with each f_i of the form

$$f_i = \sum_{j,\ j=d_i} a_{ij} x^j.$$

Here α denotes a multiindex $(\alpha_0, \ldots, \alpha_n) \in \mathbb{N}^{n+1}$ with $|\alpha| = \alpha_0 + \ldots + \alpha_n = d_i$ and $x^\alpha = x_0^{\alpha_0} \cdots x_n^{\alpha_n}$. In this case the number of a variables is

$$\ell + 1 = \sum_{i=1}^{N} \binom{n+d_i}{n}.$$

Moreover, for each $i = 1, \ldots, m$, $c_i = 1$, that is the polynomials f_i are linear in the a's.

To decide whether such a system f has a non-zero real root can be seen as a decision version over the reals of the projective Hilbert Nullstellensatz.

Example 2. Consider $m = 1$, $\ell = 3$, $n = 1$, and polynomials f of degree d with the form

$$f(a, x) = a_0 x_1^d + a_1 x_1^{d-1} x_0 + a_2 x_1 x_0^{d-1} + a_3 x_0^d.$$

This is a simple example of sparsity in which some monomials are not allowed. Again, f is linear in a_0, a_1, a_2, a_3.

In the two examples above, f is linear in the variables a_i. But of course, this needs not be so.

For some time to come we will consider the problem:

Decide on input (f, a) with $f \in \mathcal{H}_{(c;d)}$ and $a \in B(\mathbb{R}^{\ell+1})$ if f_a is feasible.

3 Condition Numbers of Homogeneous Sparse Systems

The goal of this section is to introduce the condition number $\mu(f_a)$ of the pair (f, a).

For $a \in B(\mathbb{R}^{\ell+1})$ and $x \in B(\mathbb{R}^{n+1})$ define

$$\kappa(f_a, x) = \max\{1, \|f\| \|D f_a(x)^\dagger\|\}$$

where $Df_a(x) : \mathbb{R}^{n+1} \to \mathbb{R}^m$ is the derivative of f_a at x and we have taken its Moore-Penrose inverse (see [6,1,19]). The norm in the expression $\|Df_a(x)^\dagger\|$ is the operator norm

$$\|Df_a(x)^\dagger\| = \max_{v \in \mathbb{R}^m} \frac{\|Df_a(x)^\dagger(v)\|}{\|v\|}$$

where we are using the Euclidean norm on $\mathbb{R}^m$ as well as on $\mathbb{R}^{n+1}$. If $Df_a(x)^\dagger$ is not defined (i.e. $Df_a(x)$ is not surjective) we take $\|Df_a(x)^\dagger\| = \infty$.

The quantity $\|f\|$ in the above expression for $\kappa(f_a, x)$ is the norm of f we now describe.

Let $\alpha = (\alpha_0, \ldots, \alpha_\ell) \in \mathbb{N}^{\ell+1}$ and denote by $|\alpha|$ the sum $\alpha_0 + \ldots + \alpha_\ell$. Define similarly $|\beta|$ for $\beta \in \mathbb{N}^{n+1}$. For $f = (f_1, \ldots, f_m) \in \mathcal{H}_{(c;d)}$ with

$$f_i(a, x) = \sum_{\substack{|\alpha| = c_i \\ |\beta| = d_i}} f_{i\alpha\beta}\, a^\alpha x^\beta,$$

we define

$$\|f\|^2 = \sum_{i=1}^{m} \left(\sum_{|j|=c_i;\ |j|=d_i} |f_{ij}| \right)^2 .$$

For f in Example 1, $\|f\| \le \sqrt{m}\binom{D+n}{n}$ where $D = \max_{i \le m} d_i$.

For f in Example 2, $\|f\| = 4$.

For f_a feasible define

$$\kappa(f_a) = \min_{\substack{x \in B(\mathbb{R}^{n+1}) \\ f_a(x)=0}} \kappa(f_a, x).$$

For f_a infeasible define

$$\Delta(f_a) = \frac{1}{\|f\|} \min_{x \in B(\mathbb{R}^{n+1})} \|f_a(x)\|_\infty .$$

The condition number we consider is

$$\mu(f_a) = \begin{cases} \kappa(f_a) & \text{if } f_a \text{ is feasible} \\ \frac{1}{\Delta(f_a)} & \text{if } f_a \text{ is infeasible} \end{cases} \tag{1}$$

If $\kappa(f_a) = \infty$ we say that $\mu(f_a) = \infty$. Note that, because of the compactness of $B(\mathbb{R}^{n+1})$, the minimum in the definition of $\Delta(f_a)$ is attained and we have $\Delta(f_a) \neq 0$. Therefore, $\mu(f_a) < \infty$ if f_a is infeasible.

The number $\mu(f_a)$ is a condition number in the spirit of [15,16,17,18,19] but in special cases goes back to Wilkinson [23]. Its finiteness for an input a is a condition which allows algorithms with round-off to correctly decide that input.

Remark 2. 1) For simplicity above and below, we use extensively the normalizations $\|a\|_\infty = 1$ and $\|x\|_\infty = 1$. To a certain extent we could have defined scale invariant quantities on $\mathbb{R}^{\ell+1} \times \mathbb{R}^{n+1}$ in their place. A more satisfying development could perhaps follow Dedieu-Shub [9] "multihomogeneous Newton."

2) We also remark that $\mu(f_a)$ is an homogeneous expression of degree 0 in f.

3) It is not difficult to prove that for all $f \in H_{(c;d)}$ and all $a \in B(\mathbb{R}^{\ell+1})$, $\mu(f_a) \ge 1$.

4) The choice of the box $B(\mathbb{R}^{n+1})$ instead of the spheres $S(\mathbb{R}^{n+1})$ (that is, of the norm $\|\ \|_\infty$ instead of $\|\ \|_2$) is motivated in part by the following example.

Example 3. Let $f(a,x) = a_0 x_0^d - a_1 x_1^d$, $d \ge 2$, and let a be in the line given by the equation $a_0 = a_1$. Then, f_a has a line of zeros which is given by the equation $x_0 = x_1$.

If we take representatives of a and x on $S(\mathbb{R}^2)$ we get

$$a = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right) \qquad \text{and} \qquad x = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right).$$

Then, $Df_a(x)$ is given by the matrix

$$\frac{d}{2^{d=2}}\,(1,-1)$$

and therefore,

$$Df_a(x)^{\dagger} = \frac{2^{d=2}}{2d}\begin{matrix}1\\-1\end{matrix} \qquad \text{and} \qquad \|Df_a(x)^{\dagger}\| = \frac{2^{\frac{d+1}{2}}}{2d}.$$

Had we defined $\mu(f_a)$ using the points (a,x) in unit spheres we would have $\mu(f_a) = \|Df_a(x)^{\dagger}\|\|f\| = \frac{2^{\frac{d+1}{2}}}{d}$.

On the other hand, if we take representatives of a and x on $B(\mathbb{R}^2)$ we get $a = (1,1)$ and $x = (1,1)$.

Then, $Df_a(x)$ is given by the matrix $(d, -d)$ and therefore,

$$Df_a(x)^{\dagger} = \frac{1}{2d}\begin{matrix}1\\-1\end{matrix} \qquad \text{and} \qquad \|Df_a(x)^{\dagger}\| = \frac{1}{d}\sqrt{\frac{2}{2}}.$$

Thus $\mu(f_a) = \max\{1, \|Df_a(x)^{\dagger}\|\|f\|\} = \max\{1, \frac{\sqrt{2}}{d}\} = 1$.

Note the exponential dependence on d for $\mu(f_a)$ in the first case and the lack of such dependence in the second.

4 Round-O Machines

Computations over the reals deal with two kinds of data: discrete and continuous. This is explicit in computer programs in `FORTRAN` or `C` in which both `integer` and `float` are primitive data types.

Integer values appear in continuous computations as variables controlling loops, addressing registers or describing discrete parameters of objects such as the dimension of a real matrix. An important feature of these variables is that they can be assumed free of round-off errors. If N is such a value an instruction

$$\texttt{FOR } i = 1 \texttt{ TO } N \texttt{ DO}$$

will execute the content of the loop N times and at the kth execution the value of i will be exactly k.

We now informally describe a machine model formalizing round-off computations (for a formal definition see [8]). It is an extension of the machines over $\mathbb{R}$ introduced in [5]. In this extension integers can be declared as such and will be distinguished from the reals occurring during the computation.

Let $\mathbb{R}^{\infty} = \bigsqcup_{n\ 1} \mathbb{R}^n$ be the disjoint union of the Euclidean spaces $\mathbb{R}^n$, $n \quad 1$ and define analogously $\mathbb{Z}^{\infty}$. An element in $\mathbb{R}^{\infty}$ has the form $(z_1, \ldots, z_n)$ with $z_i \ 2\ \mathbb{R}$, $i = 1, \ldots, n$ for some n. We say n is its *length*.

A *round-off machine* M is a RAM with two kinds of registers holding integers and reals respectively. The machine allows for indirect addressing and operations

$+$, $-$ and $\cdot$ in both kinds of registers. In addition, divisions and square roots may be performed on real registers. The machine branches on sign tests.

A function $\delta : \mathbb{Z}^{\infty} \times \mathbb{R}^{\infty} \to \mathbb{R}_+$ is attached to the machine. For any input $(v, a) \in \mathbb{Z}^{\infty} \times \mathbb{R}^{\infty}$ a machine M as above performs a *computation with round-off function* $\delta(v, a)$ by "following its program" until (if ever) a HALT instruction is reached. When this happens, a point in $\mathbb{Z}^{\infty} \times \mathbb{R}^{\infty}$ is produced by returning the contents of some registers.

The main feature of round-off computations is that arithmetic operations yielding a real number are affected by an error. If the operation to be done is

$$z_0 \leftarrow z_0 \circ z_{-1}$$

then the computation actually performed is

$$\begin{aligned} z_0 &\leftarrow (z_0 \circ z_{-1}) + \rho \quad \text{if } \circ \in \{+, -, \cdot\} \\ z_0 &\leftarrow \frac{z_0}{z_{-1}} + \frac{\rho}{z_{-1}} \quad \text{if } \circ = /. \end{aligned}$$

For a square root the computation actually performed is $z_0 \leftarrow \sqrt{z_0} + \rho$. In all these cases, ρ is the *round-off* made by the machine and it is required that $|\rho| \le \delta(v, a)$. The same error affects the reading of the input, such that for an input $((v_1, \ldots, v_s), (a_1, \ldots, a_\ell))$ the point actually read by M is $((v_1, \ldots, v_s), (a_1^0, \ldots, a_\ell^0))$ with $a_i^0 = a_i + \rho$ for $i = 1, \ldots, \ell$.

Notice that the only requirement on ρ is that $|\rho| \le \delta(v, a)$. We will not suppose any knowledge of ρ besides this bound.

Let $\Omega_M \subseteq \mathbb{Z}^{\infty} \times \mathbb{R}^{\infty}$ be the set of the (v, a) for which the computation of M reaches an output node. Then, M defines an *input-output function*

$$\Phi_M : \Omega_M \to \mathbb{Z}^{\infty} \times \mathbb{R}^{\infty}.$$

5 Complexity Measure

Complexity theory studies computational problems with respect to some measure of cost, usually the "running time." Classically, a notion of *size* is attached to the set of inputs and a *cost* is associated to each elementary step of the machine model under consideration. Then the complexity of a machine M is the function associating to each input size n the maximum, over all inputs x of size n, of the cost of the computation performed by M on input x. More on this is in [2] or [13] for computations over $\mathbb{Z}_2$ and in [4] for computations over arbitrary rings, especially $\mathbb{R}$.

A central theme in Numerical Analysis is the dependence of the accuracy of the computed value on a number, the condition number, whose logarithm measures how much precision is lost by small perturbations of the input in an *exact* computation. For "good" (backward stable) algorithms the loss of precision introduced by round-off errors is equivalent to that produced by a small perturbation of the input (see [24] for more on this) and therefore, the logarithm

of the condition number appears to measure the loss of precision produced by round-off errors as well.

Poorly-conditioned inputs are those having large condition number and ill-posed inputs are those having infinite condition number. If we want to allow round-off errors in the computations, poorly-conditioned inputs will require much precision to be solved and ill-posed inputs are unlikely to be solved at all. In other words, the precision function δ will have to consider the condition of its argument.

An additional property of condition numbers is that they play a central role in the complexity of iterative algorithms by controlling their speed of convergence. Again, poorly-conditioned inputs will require large amounts of time to be solved and ill-posed inputs are unlikely to be solved at all. We will not develop this here but refer the reader to [12,20,22].

Therefore, the above scheme (in which cost only depends on input size) needs to be modified (otherwise, the cost function would be infinite for all n). A natural idea is to consider the cost as a function of both the input size and the condition number.

Definition 2. The *size* of $x \in \mathbb{R}$ is 1. The *size* of $x \in \mathbb{R}^n$ is n. The *height* of $x \in \mathbb{Z}$, $x \neq 0$, is $\lceil \log(1+|x|) \rceil$ and the *height* of 0 is 1. The *size* of $x \in \mathbb{Z}^*$ is the sum of the heights of $x_1, \ldots, x_*$. Finally, for an element $(v, a) \in \mathbb{Z}^1 \times \mathbb{R}^1$ we define size(v, a) to be the sum of the sizes of v and a.

Elements in $\mathbb{Z}^1$ usually encode some discrete object and their size measures how big this encoding is. The main case of interest here is the encoding of many variable polynomials with integer coefficients. Let

$$f = \sum f_\alpha x^\alpha$$

be an integer polynomial of degree d. The *dense encoding* of f consists of the list

$$\{f_\alpha \mid |\alpha| \leq d\}.$$

This list contains *all* the coefficients f_α (including the zero coefficients) with $|\alpha| \leq d$.

The *sparse encoding* of f is the list of pairs

$$\{(\alpha, f_\alpha) \mid f_\alpha \neq 0\}.$$

Note that while the size of the dense encoding of f is always at least d, the size of its sparse encoding can be logarithmic in d for some polynomials f.

Polynomial systems are encoded using the encodings of their defining polynomials.

Definition 3. Let M be a round-off machine. We associate to each instruction η of M a cost function cost : $\mathbb{Z}_1 \times \mathbb{R}_1 \to \mathbb{N}$ as follows. Sign tests for the branchings cost 1. The same holds for any operation involving only reals. Operations involving integers (as well as indirect addressing) cost the maximal height

of the involved integers. The *cost of a computation* is the addition of the costs of all the performed instructions. Notice that the above defines a function

$$T_M : \mathbb{Z}^1 \quad \mathbb{R}^1 \; ! \; \mathbb{N} [f 1 g$$

the value 1 corresponding to inputs whose computation does not halt.

Remark 3. For the way we have defined cost, the cost of operating with integer numbers can be much higher that the cost of doing the same operation to these numbers considered as real numbers. On the other hand, the latter is subject to round-off errors. The convenience to "convert" integer values into real numbers will then depend on the round-off functions of the machine as well as on the problem at hand.

We have already defined the notions of size and cost. We now introduce the third ingredient we need to define complexity and state the main results of this section.

De nition 4. A *conditioned decision problem* is a subset $S \quad \mathbb{Z}^1 \quad \mathbb{R}^1$ together with a function $\mu : \mathbb{Z}^1 \quad \mathbb{R}^1 \; ! \; \mathbb{R}_+ [f 1 g$. The set Σ of elements $(v, a) 2 \mathbb{Z}^1 \quad \mathbb{R}^1$ such that $\mu(v, a) = 1$ is the set of *ill-posed* inputs. A round-off machine M *decides* S if

(i) $\Phi_M(v, a) = 1$ for $(v, a) 2 S - \Sigma$,
(ii) $\Phi_M(v, a) = 0$ for $(v, a) 2 (\mathbb{Z}^1 \quad \mathbb{R}^1) - (S [\Sigma)$,
(iii) if $\Phi_M(v, a) = 1$ then $(v, a) 2 S$, and
(iv) if $\Phi_M(v, a) = 0$ then $(v, a) \not 2 S$.

Here Φ_M is the input-output function of M. Notice that for $(v, a) 2 \Sigma$, M may not halt.

An example of conditioned decision problem is the following:

Decide on input (f, a), with $f 2 H_{(c;d)}$ and $a 2 B(\mathbb{R}^{'+1})$, if f_a is feasible.

Here S is the subset of $\mathbb{Z}^1 \quad \mathbb{R}^1$ of pairs (v, a) such that v is the encoding of a sparse system f, $a 2 B(\mathbb{R}^{'+1})$ and f_a is feasible. Also, μ is the function which associates to the pair (v, a) the number $\mu(f_a)$ defined in Section 3 if v encodes a sparse system f and 1 otherwise.

De nition 5. A round-off machine M is said to work with *polynomial precision* (or to be a *polynomial precision machine*) if there exists a polynomial p such that, for each input (v, a),

$$j \log \delta(v, a) j \quad p(\text{size}(v, a), \log \mu(v, a)).$$

A conditioned decision problem (S, μ) can be *decided with polynomial precision* if there exists a polynomial precision machine M deciding S.

We define *exponential precision* in a similar way by replacing the polynomial bound $p(\text{size}(v, a), \log \mu(v, a))$ above by

$$2^{p(\text{size}(v;a);\log \quad (v;a))}.$$

Remark 4. The precision can be seen as the number of bits or digits after the floating point necessary to correctly decide an input. A problem can be decided with polynomial precision if this number of bits is polynomially bounded in $\text{size}(v, a)$ and $\log \mu(v, a)$.

We can now state our first theorem.

Theorem 1. *There exists a polynomial precision machine M which with input (f, a) halts correctly with output:*

(a) "f_a is infeasible", or
(b) "f_a is feasible",

or else

(c) doesn't halt.

Moreover the last case occurs if and only if $\mu(f_a) = \infty$. The halting time satisfies

$$T_M(f, a) \quad \mu(f_a)^{2n} 2^{c\,\text{size}(f)^2}$$

if $\text{size}(f)$ *is considered for the sparse encoding and*

$$T_M(f, a) \quad \mu(f_a)^{2n} \text{size}(f)^{cn}$$

if $\text{size}(f)$ *is considered for the dense encoding. In both cases, c denotes a universal constant.*

6 Feasibility of A ne Semi-Algebraic Systems

Let $f_i, g_j, h_k \in \mathbb{Z}[a_1, \ldots, a_{\cdot}, x_1, \ldots, x_n]$ be arbitrary polynomials for $i = 1, \ldots, m$, $j = 1, \ldots, r$, and $k = 1, \ldots, q$. Our goal is the study of the problem

Given $a \in \mathbb{R}^{\cdot}$ decide whether there is a point $x \in \mathbb{R}^n$ such that

$$\begin{aligned} f_i(a, x) &= 0 \qquad \text{for } i = 1, \ldots, m, \\ g_j(a, x) \quad 0 &\qquad \text{for } j = 1, \ldots, r, \\ h_k(a, x) &> 0 \qquad \text{for } k = 1, \ldots, q. \end{aligned} \tag{2}$$

We may write these conditions as $f_a(x) = 0$, $g_a(x) \quad 0$, and $h_a(x) > 0$, and we call the triple $\varphi_a = (f_a, g_a, h_a)$ a *basic semi-algebraic system.*

De nition 6. We say that φ_a *is feasible* if a point x exists as in (2) and we say it is *infeasible* otherwise. Such an x is said to *satisfy* φ_a.

We reduce this problem to the context of Theorem 1. First "bihomogenize".

If $p \in \mathbb{Z}[a, x]$ is one of the polynomials above, $a = (a_1, \ldots, a_{\cdot})$ and $x = (x_1, \ldots, x_n)$, denote by p the polynomial in $\mathbb{Z}[a_0, a, x_0, x]$ obtained by bihomogenizing p with respect to the a's and the x's respectively. That is, if

$$p = \sum_{\;;} p \quad a \; x$$

then

$$\widehat{p} = \sum_{\alpha,\beta} p_{\alpha\beta}\, a^{\alpha} x^{\beta} a_0^{c-|\alpha|} x_0^{d-|\beta|}.$$

Here c and d are the degrees of p with respect to the a's and the x's respectively, α and β are multiindexes, and $|\alpha|$ denotes the sum of the components of α.

Thus, $\widehat{f}_i$ is the bihomogenization of f_i for $i = 1, \ldots, m$ and we write $\widehat{f} = (\widehat{f}_1, \ldots, \widehat{f}_m)$ and similarly for $\widehat{g}$ and $\widehat{h}$.

We now eliminate the inequalities. Let

$$\begin{aligned} F_i &= \widehat{f}_i(a,x) & i &= 1,\ldots,m \\ G_j &= \widehat{g}_j(a,x) - \|\widehat{g}_j\| D y_j^2 & j &= 1,\ldots,r \\ H_k &= \widehat{h}_k(a,x) - \|\widehat{h}_k\| D z_k & k &= 1,\ldots,q \end{aligned}$$

where $y_1, \ldots, y_r, z_1, \ldots, z_q$ are new variables and D is the maximum of the degrees of f_i, g_j, and h_k. Again, $F = (F_1, \ldots, F_m)$, $G = (G_1, \ldots, G_r)$ and $H = (H_1, \ldots, H_q)$.

Let $\Phi = (F, G, H)$. By abuse of notation, we shall consider Φ both as a system and as a map

$$\begin{aligned} \Phi : \mathbb{R}^{\ell+1} \times \mathbb{R}^{n+1} \times \mathbb{R}^{r+q} &\to \mathbb{R}^{m+r+q} \\ (a,x,y,z) &\mapsto (F_a(x,y,z), G_a(x,y), H_a(x,z)). \end{aligned} \tag{3}$$

Remark 5. Note that Φ is determined by φ. Moreover, if

$$\overline{a} = \frac{(1,a)}{\|(1,a)\|_1}$$

then $\overline{a} \in B(\mathbb{R}^{\ell+1})$, and $\Phi_{\overline{a}}$ is determined by φ_a. The relationship between feasibility of φ_a and feasibility of $\Phi_{\overline{a}}$ is given in the next proposition.

Let $P = \{(x,y,z) \in B(\mathbb{R}^{n+1}) \times \mathbb{R}^{r+q} \mid x_0 > 0$ and $z_k > 0$ for $k = 1, \ldots, q.\}$.

Proposition 1. *For any $a \in \mathbb{R}^{\ell}$, φ_a is feasible if and only if there exists $(x,y,z) \in P$ such that $\Phi_{\overline{a}}(x,y,z) = 0$.*

If a zero (x,y,z) of $\Phi_{\overline{a}}$ as in Proposition 1 exists we say that $\Phi_{\overline{a}}$ is *feasible on* P. Otherwise, we say that $\Phi_{\overline{a}}$ is *infeasible on* P.

We now introduce an appropriate version of the condition number φ_a. The condition number for φ_a will be defined, up to some minor modifications, as the condition number of the associated system $\Phi_{\overline{a}}$. Define $\|\Phi\|$ by

$$\|\Phi\|^2 = \sum_{i=1}^{m} \|F_i\|^2 + \sum_{j=1}^{r} \|G_j\|^2 + \sum_{k=1}^{q} \|H_k\|^2.$$

For $a \in B(\mathbb{R}^{\ell+1})$, $x \in B(\mathbb{R}^{n+1})$ and $(y,z) \in \mathbb{R}^{r+q}$ define

$$\kappa(\Phi_a, (x,y,z)) = \max\{1, \|\Phi\| \|D\Phi_a(x,y,z)^{\dagger}\|\}.$$

Now, let $a \in \mathbb{R}'$. If φ_a is feasible (i.e. $\Phi_{\overline{a}}$ is feasible on P) define

$$\kappa(\varphi_a) = \min_{\substack{(x;y;z)\in P \\ \overline{a}(x;y;z)=0}} \frac{\kappa(\Phi_{\overline{a}}, (x,y,z))}{\partial(x,y,z)} \tag{4}$$

where $\partial(x,y,z) = \min\{x_0, z_1, \ldots, z_q\}$. Note that the minimum in (4) is achieved even though the set is not compact.

If φ_a is infeasible (i.e. $\Phi_{\overline{a}}$ is infeasible on P) define

$$\Delta(\varphi_a) = \frac{1}{\|\Phi\|} \inf_{(x;y;z)\in P} \|\Phi_{\overline{a}}(x,y,z)\|_1 .$$

Again, the condition number we consider is

$$\mu(\varphi_a) = \begin{cases} \kappa(\varphi_a) \text{ if } \varphi_a \text{ is feasible} \\ \frac{1}{\Delta(\varphi_a)} \text{ if } \varphi_a \text{ is infeasible.} \end{cases}$$

If $\kappa(\varphi_a) = \infty$ or $\Delta(\varphi_a) = 0$ we say that $\mu(\varphi_a) = \infty$.

We can now extend Theorem 1 to semi-algebraic systems.

Theorem 2. *There exists a polynomial precision machine M which with input (φ, a) halts correctly with output:*

(a) "φ_a is infeasible", or
(b) "φ_a is feasible",

or else

(c) doesn't halt.

Moreover the last case occurs if and only if $\mu(\varphi_a) = \infty$. The halting time satisfies

$$T_M(\varphi, a) \quad \mu(\varphi_a)^{2n} 2^{c\,\mathrm{size}(\varphi)^2}$$

if $\mathrm{size}(\varphi)$ *is considered for the sparse encoding and*

$$T_M(\varphi, a) \quad \mu(\varphi_a)^{2n} \mathrm{size}(\varphi)^{cn}$$

if $\mathrm{size}(\varphi)$ *is considered for the dense encoding. In both cases, c denotes a universal constant.*

Remark 6. A model of round-off parallel machine can be defined by modifying the parallel machines over $\mathbb{R}$ in Chapter 18 of [4] to allow for round-off as in Section 4. Then, in Theorem 2 (and in Theorem 1 as well) we can replace sequential halting time by parallel halting time and bound the latter by

$$(\log \mu(\varphi_a) \mathrm{size}(\varphi))^c$$

with c a universal constant. The number of processors is bounded by the expression for the sequential time in Theorem 2. The polynomial bound for the precision remains the same.

References

1. E.L. Allgower and K. Georg. *Numerical Continuation Methods.* Springer-Verlag, 1990.
2. J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I.* EATCS Monographs on Theoretical Computer Science, 11. Springer-Verlag, 1988.
3. S. Basu, R. Pollack, and M.-F. Roy. On the combinatorial and algebraic complexity of quantifier elimination. In *35th annual IEEE Symp. on Foundations of Computer Science*, pages 632–641, 1994.
4. L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation.* Springer-Verlag, 1998.
5. L. Blum, M. Shub, and S. Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the Amer. Math. Soc.*, 21:1–46, 1989.
6. S.L. Campbell and C.D. Meyer. *Generalized Inverses of Linear Transformations.* Pitman, 1979.
7. G.E. Collins. *Quantifier elimination for real closed fields by cylindrical algebraic deccomposition*, volume 33 of *Lect. Notes in Comp. Sci.*, pages 134–183. Springer-Verlag, 1975.
8. F. Cucker and S. Smale. Complexity estimates depending on condition and round-off error. Preprint, 1997.
9. J.-P. Dedieu and M. Shub. Multihomogeneous Newton methods. Preprint, 1997.
10. D.Yu. Grigoriev and N.N. Vorobjov. Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5:37–64, 1988.
11. J. Heintz, M.-F. Roy, and P. Solerno. Sur la complexité du principe de Tarski-Seidenberg. *Bulletin de la Société Mathématique de France*, 118:101–126, 1990.
12. N. Higham. *Accuracy and Stability of Numerical Algorithms.* SIAM, 1996.
13. C.H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
14. J. Renegar. On the computational complexity and geometry of the first-order theory of the reals. Part I. *Journal of Symbolic Computation*, 13:255–299, 1992.
15. M. Shub and S. Smale. Complexity of Bezout's theorem I: geometric aspects. *Journal of the Amer. Math. Soc.*, 6:459–501, 1993.
16. M. Shub and S. Smale. Complexity of Bezout's theorem II: volumes and probabilities. In F. Eyssette and A. Galligo, editors, *Computational Algebraic Geometry*, volume 109 of *Progress in Mathematics*, pages 267–285. Birkhäuser, 1993.
17. M. Shub and S. Smale. Complexity of Bezout's theorem III: condition number and packing. *Journal of Complexity*, 9:4–14, 1993.
18. M. Shub and S. Smale. Complexity of Bezout's theorem V: polynomial time. *Theoretical Computer Science*, 133:141–164, 1994.
19. M. Shub and S. Smale. Complexity of Bezout's theorem IV: probability of success; extensions. *SIAM J. of Numer. Anal.*, 33:128–148, 1996.
20. S. Smale. Complexity theory and numerical analysis. In A. Iserles, editor, *Acta Numerica*, pages 523–551. Cambridge University Press, 1997.
21. A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951.
22. L.N. Trefethen and D. Bau III. *Numerical Linear Algebra.* SIAM, 1997.
23. J. Wilkinson. *Rounding Errors in Algebraic Processes.* Prentice Hall, 1963.
24. J. Wilkinson. Modern error analyis. *SIAM Review*, 13:548–568, 1971.

Intrinsic Near Quadratic Complexity Bounds for Real Multivariate Root Counting

J. Maurice Rojas[?]

Department of Mathematics, Y6508
City University of Hong Kong
83 Tat Chee Avenue
Kowloon, HONG KONG
FAX: (Hong Kong) [852] 2788-8561
mamrojas@math.cityu.edu.hk
http://www.cityu.edu.hk/ma/staff/rojas

We give a new algorithm, with three versions, for computing the number of real roots of a system of n polynomial equations in n unknowns. The first version is of Monte Carlo type and, neglecting logarithmic factors, runs in time **quadratic** in the average number of complex roots of a closely related system. The other two versions run nearly as fast and progressively remove a measure zero locus of failures present in the first version. Via a slight simplification of our algorithm, we can also count complex roots, with or without multiplicity, within the same complexity bounds. We also derive an even faster algorithm for the special case $n=2$, which may be of independent interest.

Remark 1 *For technical reasons, we will only consider roots with all coordinates nonzero. This restriction will be lifted in forthcoming work of the author, and most of the theory necessary for this extension has already appeared in [Roj97c,Roj98a]. We will also assume that all our input polynomial systems have only finitely many complex roots with all coordinates nonzero. This mild restriction can also be lifted, but the details will be covered in a later paper. In any case, a feature of our algorithm is that we* **do** *allow infinitely many roots at infinity.*[1]

Our Main Theorem, along with an independent approach in [MP98], represent the first complexity bounds for real root counting which are **intrinsic** in the sense that they depend mainly on the geometric properties of the underlying zero set. Indeed, it was an open question whether any older algorithms for real root counting satisfied these more refined complexity bounds.[2]

[?] This research was completed at City University of Hong Kong and partially funded by a National Science Foundation Mathematical Sciences Postdoctoral Fellowship.

[1] Most prior resultant-based algorithms for equation solving assume finiteness of the number of solutions (typically in projective space) of a homogeneous version of the input system.

[2] We should also point out that the existence of algorithms for real root counting with complexity bounds solely in terms of the number of real roots is an open problem of the utmost interest.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 127–138, 1998.
c Springer-Verlag Berlin Heidelberg 1998

Before stating our results, we give a quick illustration of the benefits of intrinsic bounds. Consider the following bivariate polynomial system:

$$f_1(x;y) = a_1 + a_2x^8y^2 + a_3x^9y^2 + a_4x^{10}y^2 + a_5x^{10}y^3 + a_6x^9y^3$$

$$f_2(x;y) = b_1 + b_2x + b_3xy^3 + b_4x^2y^2 + b_5x^5y^3 + b_6x^5y^4 + b_7x^4y^4$$

Suppose we choose complex constants for the coefficients $\{a_i\}$ and $\{b_i\}$. Recall that the classical Bézout's Theorem [Mum82] gives the loose upper bound of $13\ 9 = 117$ (the product of the total degrees of the polynomials) on the number of isolated complex roots, for **any** choice of the coefficients. We say "loose" because the **maximal** number of isolated complex roots, for any choice of coefficients, is actually 35. We will return to this example in section 3, but our point for the moment is that exploiting the monomial term structure, as opposed to working solely in terms of d (the maximum of the total degrees of the input systems), can result in much better bounds.

To state our complexity results, let F be an n n polynomial system with support contained in an n-tuple of point sets $E := (E_1;:::;E_n)$ [Roj97a]. (So the n-tuple E determines precisely which monomial terms are allowed to appear in F.) Let us briefly recall Bernshtein's Theorem [Ber75,GKZ94,Ewa96,Roj98a], which asserts the following: (1) The number, $N_{\mathbb{C}}$, of isolated complex roots of F with all coordinates nonzero is bounded above by the **mixed volume**, $M(E)$, of the convex hulls of $E_1;:::;E_n$, (2) $M(E)$ d^n when the support of F **is** E, and (3) for fixed E, $N_{\mathbb{C}} = M(E)$ for all choices of coefficients outside of a codimension 1 locus. For most purposes, the last statement can be interpreted as "$N_{\mathbb{C}} = M(E)$ with probability 1."

Definition 1 *For any fixed k n and any n-tuple of point sets in $\mathbb{R}^k$, $(E_1;:::;E_n)$, define $M_{\mathbf{E}}^{\text{ave}}$ to be the* **average** *of the k-dimensional mixed volumes $\{M(E)\}$, as E ranges over all* **ordered** *k-tuples $(E_1;:::;E_k)$ with E_i 2 $\{E_1;:::;E_n\}$ for all i 2 $\{1;:::;k\}$. Also, for any a 2 $\mathbb{Z}^n \setminus \{0\}$, let p_a denote the orthogonal projection mapping $\mathbb{R}^n$ onto the hyperplane perpendicular to the vector a.*

Main Theorem *Suppose F is an n n polynomial system with real coefficients, a total of m monomial terms, and support contained in E. Then there is a Monte Carlo algorithm (cf. section 2, Main Algorithm) for counting the exact number of real roots of F, with the following versions and run times (counting arithmetic steps over $\mathbb{R}$):*

Version	*Run Time*$=O($ $)$	codim
1	$M(E)S_1 \log^2 S_1$	1
2	$M(E)S_m^2 \log^2 S_m$	2
Safe	$M(E)S_m^3 \log^2 S_m$	*1*

where[3] $S_t :=$ $p_{\overline{n}e^n}(M_E^{\text{ave}} + t^2 p_{\overline{n}} \max_{a \in [-t^2;t^2]^n} \{M_{p_a(E)}^{\text{ave}}\})$. *The codimension entries refer to the codimension of a locus of coefficient values (depending only*

[3] The first and second "average mixed volume" terms are respectively n-dimensional and $(n-1)$-dimensional. Also, $p_a(E) := (p_a(E_1);:::;p_a(E_n))$.

on E) where the algorithm may not work. In particular, the safe version will count the real roots of **any** *input F and, with a controllably small probability, will at worst underestimate the number of real roots.*

Furthermore, when $n=2$, we can make the following improvements:

Version	*Run Time*$=O(\quad)$	codim
1	$m \log m + M(E) S_1^0 \log^2 S_1^0$	1
Safe	$m \log m + M(E) S_m^0 \log^2 S_m^0$	*1*

where $S_t^0 := t \max_{a \in [-t;t]^2} \{\text{Length}(p_a(E_1)) + \text{Length}(p_a(E_2))\}$.

Via a slight simplification of the above algorithms, the same complexity bounds hold for **complex** *root counting as well, with or without multiplicity.*

When $n > 2$ and E is such that an algebraic formula of a special type exists (cf. section 1), we can remove a factor of S_m from the complexities of versions 2 and "Safe," and replace the parameter S_t by the smaller quantity $M(E) + t^2 \overline{n} \max_{a \in [-t^2;t^2]^n} \{\sum_{i=1}^{n} M(p_a(E_1); \ldots; p_a(E_{i-1}); p_a(E_{i+1}); \ldots; p_a(E_n))\}$. Also, in practice, one can drastically reduce S_t and S_t^0 once E is fixed.

We also point out that when $n > 2$ our algorithm relies on a preprocessing step. However, this preprocessing need only be done once per monomial term structure. In applications where the monomial term structure is fixed and the coefficients vary many times, the preprocessing step has negligible complexity (cf. section 2).

Our algorithm for real root counting will use the **toric** (a.k.a. **sparse**) resultant[4] [Stu93,Stu94,GKZ94,EC95,Stu97], Sturm sequences [Roy95], some fast matrix arithmetic algorithms [BP94], and a bit of convex geometry [Ewa96]. After the description and complexity analysis of our algorithm in section 2, we give a simple example of our algorithm in section 3. Finally, in section 4, we briefly compare our complexity bounds with some older algorithms, as well as the new approach of [MP98].

1 An Executive Summary of Resultants

We will mainly follow the notation of [Roj97a]. In particular, let $\bar{F} := (f_1; \ldots; f_{n+1})$ be an $(\mathbf{n+1})$ **n indeterminate** polynomial system with support $\bar{E}$. Then there is a corresponding **toric** (a.k.a. **sparse**) **resultant**, $\text{Res}_{\bar{E}}(\ )$, which is a polynomial in the coefficients $\{c_{i;e} \mid i \in \{1; \ldots; n+1\}, e \in E_i\}$. Let $C_{\bar{E}}$ be the **vector** consisting of all these indeterminate coefficients, and define[5] $|\bar{E}| := \sum_{i=1}^{n+1} |E_i|$. Then, up to sign, $\text{Res}_{\bar{E}}(\ )$ can be defined as the unique polynomial of lowest degree (with relatively prime integer coefficients) satisfying the following property: $\bar{F}|_{C_{\bar{E}}=C}$ has a root in $(\mathbb{C}\ )^n$ for some $C \in \mathbb{C}^{|\bar{E}|} \Rightarrow \text{Res}_{\bar{E}}(C) = 0$. (We

[4] Other commonly used prefixes for this modern extension of the classical resultant include: A-, $(A_1; \ldots; A_k)$-, mixed, sparse mixed, and Newton.

[5] We use $|\ |$ for set cardinality. In a completely analogous way, we will also let C_E be the vector of coefficients of F.

let $\mathbb{C}^*$ denote the set of **nonzero** complex numbers.) For convenience, we will usually write $\mathrm{Res}_{\bar{E}}(\bar{F})$ instead of $\mathrm{Res}_{\bar{E}}(C)$ whenever $C_{\bar{E}} = C$.

Example 1 *Suppose we pick* $E_1 = \cdots = E_{n+1} = \{\mathbf{O}, \hat{e}_1, \ldots, \hat{e}_n\}$, *where* $\mathbf{O} \in \mathbb{R}^n$ *is the origin, and* $\hat{e}_i \in \mathbb{R}^n$ *is the* $i^{\underline{\text{th}}}$ *standard basis vector. (So each* E_i *is the vertex set of a standard* n*-simplex.) Then we are dealing with a polynomial condition for* $n+1$ *affine forms in* n *variables to have a common root. In particular, it is easily verified that* $\mathrm{Res}_{\bar{E}}$ *is just the* **determinant** *of the* $(n+1) \times (n+1)$ *coefficient matrix of this* $\bar{F}$.

The toric resultant is the theoretical foundation for some of the fastest current methods for solving polynomial equations [MC92,Emi94]. Also, whereas the classical resultant really measured the existence of roots in projective space, the toric resultant actually measures the existence of roots in a suitable **toric variety** [GKZ94,Roj98a] — hence the author's preference for this particular name.

From basic elimination theory, one can deduce the following facts about the polynomial $\mathcal{R}_i := \mathrm{Res}_{\bar{E}}(F, u_i + x_i)$, where $\bar{E} := (E, \{\mathbf{O}, \hat{e}_i\})$ and F has constant coefficients: (1) $\mathcal{R}_i$ is a polynomial in the single variable u_i, and (2) $\mathcal{R}_i(-\zeta_i) = 0$ for any root $(\zeta_1, \ldots, \zeta_n)$ of F. This would give a general method to reduce root counting to the univariate case, were it not for two problems: $(\mathbf{P_1})$ The polynomial $\mathcal{R}_i$ can be identically zero, and $(\mathbf{P_2})$ two roots of F might have the same $i^{\underline{\text{th}}}$ coordinate. However, there is a way out via better binomials.

Theorem 1 *[Roj97a] Let* F *be an* $n \times n$ *polynomial system with support* $E := (E_1, \ldots, E_n)$ *such that* $M(E) > 0$. *Also let* P_E *be the sum of the convex hulls of the* E_i, *and pick* $a \in \mathbb{Z}^n \setminus \{\mathbf{O}\}$ *such that the segment* $[\mathbf{O}, a]$ *is* **not** *parallel to any facet of* P_E. *Define* $\mathcal{R}_{\boldsymbol{a}}(u_+, u_-) := \mathrm{Res}_{E_{\boldsymbol{a}}}(F, u_+ + u_- x^{\boldsymbol{a}})$ *where* $\{u_\pm\}$ *is a pair of parameters and* $E_{\boldsymbol{a}} := (E, \{\mathbf{O}, a\})$. *Then* $\mathcal{R}_{\boldsymbol{a}}$ *is either identically zero or a homogeneous polynomial of degree* $M(E)$. *Furthermore, the following hold:*

1. *Let* $\varepsilon_\pm$ *be the lowest exponent of* $u_\pm$ *occuring in any monomial of* $\mathcal{R}_{\boldsymbol{a}}$. *Then* F *has* **exactly** $N := M(E) - \varepsilon_+ - \varepsilon_-$ *roots in* $(\mathbb{C}^*)^n$, *counting multiplicities, provided* $N < \infty$.
2. *Let* $\mathcal{R}^1_{\boldsymbol{a}}$ *be the square-free part of* $\mathcal{R}_{\boldsymbol{a}}$ *and let* $\varepsilon^1_\pm$ *be the lowest exponent of* $u_\pm$ *occuring in any monomial of* $\mathcal{R}^1_{\boldsymbol{a}}$. *Then* F *has at least* $N^1 := \deg(\mathcal{R}^1_{\boldsymbol{a}}) - \varepsilon^1_+ - \varepsilon^1_-$ **distinct** *roots in* $(\mathbb{C}^*)^n$. *Furthermore, this count is exact provided the projection of roots* $\zeta \mapsto \zeta^{\boldsymbol{a}}$ *is one to one.*
3. *The set of all* C_E *such that* $\mathcal{R}_{\boldsymbol{a}}$ *is identically zero (i.e.,* $N = \infty$*) forms a locus of codimension* ≥ 2. *More generally, for* **arbitrary** $a \in \mathbb{Z}^n \setminus \{\mathbf{O}\}$, *the set of* C_E *for which (1) and (2) fail to hold is of codimension* ≥ 1.
4. *If some monomial multiple of* $\mathcal{R}_{\boldsymbol{a}}$ *is square-free, then no root of* F *or* $\mathcal{R}_{\boldsymbol{a}}(u_+, -1)$ *has multiplicity* > 1.
5. $\mathcal{R}_{\boldsymbol{a}}(\zeta^{\boldsymbol{a}}, -1) = 0$ *for any root* $\zeta \in (\mathbb{C}^*)^n$ *of* F.
6. *When* $n = 2$ *the first locus in (3) is always empty.* ■

Remark 2 *The convex geometric hypothesis on* a *is necessary in order to avoid problems* $(\mathbf{P_1})$ *and* $(\mathbf{P_2})$. *(In particular, if we only use* $a \in \{\hat{e}_i\}$, *then the best we can hope for is a codimension* 1 *locus in assertion 3.) In particular, combining* **binomials** *with toric resultants is better (and faster) than the usual trick of combining generic linear forms with resultants.*

In the above theorem, the **mixed volume** is a function assigning a nonnegative integer to every input n-tuple of polytopes with integral vertices [Ewa96]. In particular, the hypothesis $M(E) > 0$ simply rules out those supports giving a system F having, under a suitable change of coordinates, a subsystem with too many equations in too few variables. This hypothesis can actually be checked in time polynomial in n, by using a slight variant of Edmond's matroid intersection algorithm [GK94].

We will also need the following technical result to build the "safe" version of our algorithm.

Theorem 2 *[Roj98b] Following the notation of Theorem 1, suppose* $\quad_{\boldsymbol{a}}$ *is identically zero. Let us then form the* **double toric perturbation of** $\quad_{\boldsymbol{a}}$, $\tilde{}_{\boldsymbol{a}}$, *as follows: Define* $\tilde{}_{\boldsymbol{a}}$ *to be the gcd of the coefficients of the lowest power of* s *in the following two polynomials*

$$\mathrm{Res}_{E_{\boldsymbol{a}}}(F - sF \;; u_+ + u_- x^{\boldsymbol{a}}) \text{ and } \mathrm{Res}_{E_{\boldsymbol{a}}}(F - sF \;; u_+ + u_- x^{\boldsymbol{a}});$$

where F *and* F *are distinct generic polynomial systems with support contained in* E. *Then* $\tilde{}_{\boldsymbol{a}}$ *is a homogeneous polynomial of degree* $M(E)$, **not** *identically equal to zero, satisfying all but one assertion of Theorem 1: in place of assertion 5, we instead have the following:*

5′. $\tilde{}_{\boldsymbol{a}}(\ ^{\boldsymbol{a}}; -1) = 0$ *for any* **isolated** *root* $\quad$ *of* F. ■

It immediately follows from Theorem 1 that the set of all eligible $(F \;; F \quad)$ for Theorem 2 lies outside of a locus of codimension 2 in coefficient space. So picking random number coefficients is actually a reliable way of generating such $(F \;; F \quad)$. Unfortunately, an efficient **deterministic** way of picking F and F is still unknown.

We will also use some basic results from the geometry of numbers, but for the sake of brevity, we will not enter into a detailed description. So to state our final technical lemma, we will instead close this section by recalling a salient fact about the current state of the art of toric resultant computation: For now, the most reliable way of computing $\mathrm{Res}_{\bar{E}}$ is to first find a matrix $R_{\bar{E}}$, with each entry either 0 or a coefficient of $\bar{F}$, such that (1) $\det(R_{\bar{E}})$ is not identically zero, and (2) $\mathrm{Res}_{\bar{E}}(\bar{F})$ divides $\det(R_{\bar{E}})$. (The remainder of this computation is covered in [Emi94,EC95], among other references.) In particular, we say that E is **determinantal** iff, for all $\boldsymbol{a} \in \mathbb{Z}^n$, we can always construct the matrix $R_{E_{\boldsymbol{a}}}$ so that the excess factor from (2) is a constant. This is the condition sufficient for the improved value of S_t in our complexity bounds, as mentioned just after the statement of our Main Theorem. Our final lemma also explains these values of S_t.

Lemma 1. *Suppose* $\boldsymbol{a} \in \mathbb{Z}^n \setminus \{\mathbf{0}\}$ *and* F *is an indeterminate* $n \quad n$ *polynomial system with support* E. *Then the total degree of the toric resultant* $\mathrm{Res}_{E_{\boldsymbol{a}}}(C_{\bar{E}})$ *is* $M(E_1 + \{\mathbf{0}, \boldsymbol{a}\}, \ldots, E_n + \{\mathbf{0}, \boldsymbol{a}\})$, *and the degree with respect to the coefficients of* F *is exactly* $M(E)$. *Furthermore,* $R_{E_{\boldsymbol{a}}}$ *is* **quasi-Toeplitz** *[BP94] and its size* S *is bounded above by*

$$\overline{n}e^n\left(M_E^{\mathrm{ave}} + \mathrm{Length}(\boldsymbol{a})\, M_{p_{\boldsymbol{a}}(E)}^{\mathrm{ave}}\right)$$

in general, and this upper bound improves to

$$M(E) + \text{Length}(a) \sum_{i=1}^{\#} M(p_a(E_1); \cdots; \widehat{p_a(E_i)}; \cdots; p_a(E_n))g$$

if E is determinantal. ■

In the above, we use the natural addition for point sets in $\mathbb{R}^n$: $A+B := fa+b \mid a2 A; b2 Bg$ for any $A; B \quad \mathbb{R}^n$.

2 The Power of Sparsity in Real Root Counting

The following is an overview of our algorithm. The three versions stated earlier will follow from varying the execution of Steps 3–5 (and how many times this is done), while our preprocessing step is essentially the execution of Steps 1–2.

Main Algorithm

Input: *An $n \quad n$ polynomial system F with support E, such that $M(E) > 0$.*[6] *(We also assume F has only finitely many roots in $(\mathbb{C} \ )^n$, but allow infinitely many roots at infinity.)*

Output: *The number of real roots of F with all coordinates nonzero.*

Description:

1. *Pick a point $a \ 2 \ \mathbb{Z}^n$ such that the coordinates of a have no nontrivial common divisor and the line segment $[\mathbf{O}; a]$ is* **not** *parallel to any facet of the polytope $\sum_{i=1}^{n} \text{Conv}(E_i)$.*
2. *Define E_a to be the $(n+1)$-tuple $(E; f\mathbf{O}; ag)$, and compute R_{E_a}.*
3. *Following the notation of Theorem 1, evaluate the polynomial $_a(\ )$ at $M(E)+1$ distinct choices of .*
4. *Using the values from Step 3, solve for the coefficients of $_a$.*
5. *Find the number $N_{\mathbb{R}}$ of nonzero real roots of the polynomial $_a(u_+; 1)$ via Sturm sequences.*
6. *Output $N_{\mathbb{R}}$ as the number of real roots of F.*

Remark 3 *We actually obtain more than just the number of roots: If we simply stop at Step 4, Theorem 1 tells us that $(x; -1)$ is a polynomial whose roots are precisely the values of ^{a} as ranges over all roots of F in $(\mathbb{C} \ )^n$.*

The approach taken by our algorithm (univariate reduction and counting sign alternations in a sequence of polynomial evaluations) is essentially classical. However, the use of the toric resultant, and a precise understanding of the potential degeneracies of this method, are new. The correctness of our algorithm will be derived below, so let us begin by describing our preprocessing step.

First note that a satisfies the hypotheses of our Main Algorithm if and only if a is in the complement of a finite union of hyperplanes in $\mathbb{Z}^n$. Furthermore, the number of such hyperplanes is $O(m^{2n})$, when $n > 2$ [GS93], or

[6] As we remarked in the last section, this condition merely rules out certain overdetermined systems, and can easily be checked in polynomial time.

$O(m)$ when $n=2$. It is then easily shown that if one picks a uniformly random a in $[-O(m^{2=}\ );O(m^{2=}\ )]^n$, and removes common factors from the coordinates of a, then the probability that a avoids the hyperplanes is at least $1-\ $.

To compute R_{E_a}, it immediately follows from [Emi96] that this will take $O(Sn^{9:5}m^{6:5}\log^2\frac{d}{\pi})$ bit operations for an error probability of π, where S is as stated in Lemma 1 and d is an upper bound on the absolute values of the coordinates of the E_i. (Note that we are also implicitly using the fact that the $(n+1)^{\underline{\text{st}}}$ support consists of just two points.) Thus, if we chose a randomly as detailed above, we immediately obtain the above complexity bound for a Monte Carlo success probability of $1-\ -\pi$ for our preprocessing step.

Alternatively, if we want to preprocess deterministically, let us make the following more refined observation on where a must lie: a satisfies the hypotheses of our algorithm if and only if a lies in the interior of some n-dimensional cone in the complement of the hyperplane arrangement defined by the facets of P_E. To check the latter condition, it clearly suffices to know the faces of the convex hulls of all the E_i to generate such a cone . To then generate R_{E_a} without randomization, we can make use of the Cayley trick [GKZ94] to find a mixed subdivision of E_a, and then enumerate shifted lattice points in our mixed subdivision. (That this suffices to build a resultant matrix follows from the development of [Stu93,Emi94].) From the well-known complexity bounds for computing convex hulls and simplicial subdivisions [PS85], it is not hard to thus derive a deterministic complexity bound of $O(S+(m+2)^{bn-\frac{1}{2}c})$ real arithmetic steps for our preprocessing. Furthermore, we can always pick a constant number of good a in $[-O(m^2);O(m^2)]^n$ within the same asymptotic complexity bound.

Before describing the three versions of our algorithm, we will briefly detail the complexity of Steps 4–5 of our Main Algorithm. This will then completely reduce our complexity analysis to how we compute $\ _a$ (or $\tilde{}_a$), and whether we do this more than once.

First recall that the coefficients of a polynomial of degree d can be recovered via interpolation in time $O(d\log^2 d)$ [BP94]. Also, via the same reference, we know that the signs at 1 of the Sturm sequence of such a polynomial can be computed within time $O(d\log^2 d)$ as well. Furthermore, the gcd of two polynomials of degree d can also be computed within $O(d\log^2 d)$ arithmetic steps [BP94]. Thus, since $d\quad M(E)$ in our case at hand (by Lemma 1), we need only show that Step 3 can be done quickly enough.

Recall that the determinant of a $k\quad k$ quasi-Toeplitz matrix can be evaluated within time $O(k\log^2 k)$ [BP94]. Also recall that the condition that two distinct roots of F lie on a hypersurface of the form $f\quad j\quad ^a=\text{constant}g$ is a codimension 1 condition in the space of coefficients. (This follows easily from considering resultants and the univariate discriminant.) We now describe our three versions.

Version 1: Here, since we are only interested in a codimension 1 locus, we can be quite lazy. First note that by Theorem 1 (assertion 3), we can just pick $a=\hat{e}_i$ for any i. (So we can avoid the construction of if we choose to be deterministic in our preprocessing.) Next, note that R_{E_a} is a nonzero constant multiple of $\ _a$,

outside of a codimension 1 locus. Thus, recalling our observations on the map $\mapsto \ ^{\boldsymbol{a}}$, it suffices to work with $M(E)+1$ evaluations of $\det(R_{E_{\boldsymbol{a}}})$.

To conclude our complexity analysis, simply note that size of $R_{E_{\boldsymbol{a}}}$ is bounded above as specified by Lemma 1. Since $R_{E_{\boldsymbol{a}}}$ is quasi-Toeplitz, our first complexity bounds follow. ■

Remark 4 *Our first algorithm* **can** *detect failure, but only if no two distinct complex roots of F have the same value of $\ ^{\boldsymbol{a}}$. (This is part of the codimension 1 locus of failure.) We make this failure even less likely in our next version.*

Version 2: First note that in order to maintain a failure locus of codimension 2, we can no longer rely solely on determinants of $R_{E_{\boldsymbol{a}}}$. So we must must add an extra indeterminate and perturb, in order to compute our necessary $O(M(E))$ values of $\ _{\boldsymbol{a}}(\ )$. This perturbation technique is described in [EC95], and results in an increase of our last complexity bound by an additional factor of S_t. (Note also that this particular step is necessary **only** if $E_{\boldsymbol{a}}$ is not determinantal.)

The only other worry is the projection of roots defined by $\mapsto \ ^{\boldsymbol{a}}$. However, we can perform the following trick: Run the algorithm twice, but now using two distinct values of $a \in [-O(m^2), O(m^2)]^n$ satisfying the hypotheses of our Main Algorithm. One then uses the **largest** putative root count, and it is easily shown (for fixed E and a fixed pair of a's) that the set of C_E for which both of these root counts is wrong forms a codimension 2 locus. Adding up the preceding complexities, we obtain our desired bound. ■

Safe: Here, we now need only worry about the case where $n > 2$ and $\ _{\boldsymbol{a}}$ vanishes identically. By Theorem 2, we can then simply work with $\tilde{\ }_{\boldsymbol{a}}$ instead of $\ _{\boldsymbol{a}}$.

This entails working with a slight modification of $R_{E_{\boldsymbol{a}}}$ with an **extra** indeterminate s, appearing in exactly S_m rows of our new matrix. We must then work over $\mathbb{R}[s]$ instead of $\mathbb{R}$, and it is easily checked (via the techniques of [Can89]) that this adds only one more factor of S_m to our complexity. The remaining gcd computation falls well within the asymptotic bounds of the preceding computational work. ■

Note that to calculate the number of **complex** roots, we can essentially use the same algorithm: We need only omit the Sturm sequence calculation, and simply calculate the exponent of the lowest order term of $\ _{\boldsymbol{a}}(u_{+}, 1)$. Furthermore, if we only want to count distinct roots without multiplicities, we need only work with the square-free part of $\ _{\boldsymbol{a}}(u_{+}, 1)$. This square-free part can be isolated in time $O(M(E) \log M(E))$ by using a fast gcd algorithm [BP94]. So we have now proved Main Theorem, except for our speedier algorithm in the case of $n=2$.

For lack of space, we will simply give an abbreviated description of the improved algorithm (deferring a complete analysis to the full version of this paper): First, pick a $U \in \mathbb{GL}_2(\mathbb{Z})$ such that the first column of U is not parallel to any edge of P_E, and the entries of U have absolute value $O(t)$. Next, we make the monomial change of variables $x \mapsto u^U$. We then use our Main Algorithm, but with a slight change: We instead compute the Sylvester resultant, $\ (u_2)$, of $f_1(u_1)$ and $f_2(u_1)$. (So f_1 and f_2 are considered as polynomials with coefficients in $\mathbb{R}[u_2]$.) Also, we use the corresponding Sylvester matrix in place of $R_{E_{\boldsymbol{a}}}$, and in place of $\ _{\boldsymbol{a}}$.

By using a slightly modified version 1, and a simplified version "safe," we can obtain the stated reduced complexity bounds. In particular, we respectively take $t=1$ and $t=m$ in these two versions. ■

3 A Simple Example of Our Algorithm

Suppose one would like to count the number of real roots of the following bivariate polynomial system:

$$f_1(x;y) = 1 - 2x^8y^2 - 3x^9y^2 + x^{10}y^2 + x^{10}y^3 + x^9y^3$$

$$f_2(x;y) = 1 + x + 2xy^3 - 5x^2y^2 + 2x^5y^3 + 2x^5y^4 + 2x^4y^4$$

Note that the above system has **no** roots with either coordinate zero. (If we set x and/or y to zero, the first equation becomes $1 = 0$.) So to count the real roots of this system, it suffices to count roots in $(\mathbb{R}\ )^2$ via our Main Algorithm.[7]

Following the notation of our Main Algorithm, the convex hulls $\mathrm{Conv}(E_1)$ and $\mathrm{Conv}(E_2)$ (which indeed satisfy the hypotheses of our algorithm) are illustrated in Figure 1. It is then easily verified that $\sum_{i=1}^{2} \mathrm{Conv}(E_i)$ is a heptagon with no

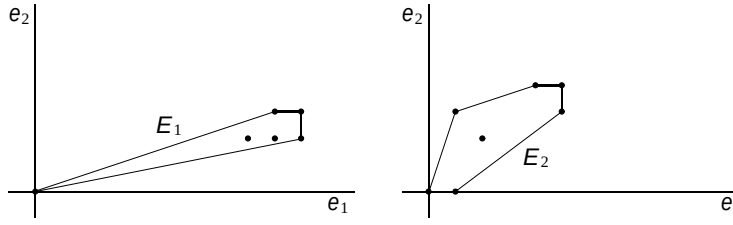


Fig. 1. The supports E_1 and E_2, along with their convex hulls. Note that each of the three "upper" edges of $\mathrm{Conv}(E_1)$ is parallel to some edge of $\mathrm{Conv}(E_2)$.

edge parallel to the line segment $[0;(1;1)]$. So we can pick $a := (1;1)$ for Step 1.

At this point, we can use the author's `Matlab` software package[8] `res2.m` to compute R_{E_a}. We omit a description of the resulting 63 63 matrix[9], but for the sake of completeness, we will display part of $_a(u_+;1)$:

$$-38400u_+^{31} + 7808u_+^{30} + \quad + 10u_+^3 + 2u_+^2 + 1$$

This polynomial is square-free (easily verified with `Maple`) so by Theorem 1, $N_{\mathbb{R}}$ (from our Main Algorithm) will indeed be the number of real roots of our 2 2

[7] We will illustrate the general Main Algorithm, without the special optimizations for $n=2$.

[8] An alpha version of this program is publically available from the author's webpage.

[9] The 63 (well within the given estimate for S_{13}) is but an artifact of our particular algorithm for computing R_{E_a}. If E were determinantal (which we don't know at this point), we would be able to obtain a 47 47 matrix instead.

system. `Maple` can then do the necessary Sturm sequence calculation for us via the command `realroot` in a fraction of a second to obtain that $_{a}(u_{+};1)$ has exactly **3 real roots**.

Finally, note that our Main Algorithm (via Theorem 1) also gives us the exact number of complex roots: $\underline{\mathbf{31}} = 35 - 4$. This is already a significant improvement over the classical Bézout's theorem (which gave an upper bound of 117) and even Bernshtein's theorem [Ber75] (which gives an upper bound of 35).

In closing, we point out that experimental evidence with polynomial system solving algorithms based on the **classical** resultant [MC92] suggests that we can expect a decent `C` implementation of our algorithms to run within milliseconds when $M(E)$ is less than 1000 (once the preprocessing step has been done). Thus our algorithms, which are based on the more efficient toric resultant, have considerable practical potential.

4 Conclusion

We have seen a fast algorithm for counting the number of real and complex solutions of a system of n polynomial equations in n unknowns. A feature of our algorithm is that its worst-case complexity can be stated in purely geometric terms. By avoiding the use of generic linear projections, our algorithm thus avoids many classically occuring degeneracies and computational bottlenecks.

We now briefly compare our worst-case complexity bounds with those of other algorithms. It appears that before this paper, the best complexity bound for real root counting was $d^{O(n)}$ [Can93,Roy95,BPR96], and work of Renegar [Ren87] suggests that the implied asymptotic constant is at worst 4. However, regardless of this asymptotic constant, there are many infinite families of sparse systems of equations for which d^n exceeds our complexity bounds by orders of magnitude. For instance, in [Roj97b], a family of systems $\{F_n\}_{n=1}^{1}$ is presented where F_n is n n and has exactly r complex roots for all $n\,2\,\mathbb{N}$. However, the corresponding value of d^n for F_n is $n^n r^n$. With a bit more geometric work, we can see that our complexity bound becomes $O(^{p}\overline{n}e^n r)$ while the older bound is $(nr)^{O(n)}$. Alternatively, to get another family of examples, we can simply specialize our results to the case of n equations of total degree d to obtain a complexity bound of $O(d^{2n+\frac{n+\frac{3}{2}\log n+\log\log d}{\log d}})$.

The author then discovered the paper [MP98] by Mourrain and Pan, after this paper was first submitted to these proceedings. In [MP98] the authors discuss the related problem of approximating all real and complex roots of F, as well as real root counting. For the latter problem, a Las Vegas bound of $O(3^n D^2 \log D)$ is stated in [MP98, Proposition 7.2], where D is an upper bound on the number of complex roots. For the case of counting roots in $(\mathbb{R}\ )^n$, D can be taken to be $M(E)$. So here, the comparison is a bit more subtle.

First, we mention that the preprocessing complexity appears to be the same in both approaches, as both methods make use of the toric resultant. Secondly, the loci of failure for the [MP98] algorithm and version 1 of our algorithm both

have codimension 1.[10] (While there is overlap in these loci, neither locus contains the other.) Thirdly, no mention is made in [MP98] of how the complexity rises when the failure loci are removed.

We are thus reduced to comparing $O(M(E)S_1 \log^2 S_1)$ (the complexity of version 1 of our algorithm) with $O(3^n M(E)^2 \log M(E))$. Considering the case where all the Newton polytopes are identical and equal to some polytope P $\mathbb{R}^n$, we can obtain an infinite family of supports for which our bound is better, and another infinite family where the bound from [MP98] is better. (For the first family, one can assume P is any cube. For the second family, one can assume that P is a small neighborhood of an $(n-1)$-dimensional polytope of a certain type.) For lack of space, we leave the details to the full version of this paper.

In conclusion, it appears that a combination of both these techniques can be quite profitable. Also, considerable computational experimentation with these algorithms would be quite valuable for gaining further insight into such a combination. We hope to address this point in the future.

References

Ber75. Bernshtein, D. N., *"The Number of Roots of a System of Equations,"* Functional Analysis and its Applications (translated from Russian), Vol. 9, No. 2, (1975), pp. 183–185.

BPR96. Basu, S., Pollack, R., and Roy, M.-F., *"On the Combinatorial and Algebraic Complexity of Quantifier Elimination,"* J. ACM 43 (1996), no. 6, pp. 1002–1045.

BP94. Bini, Dario and Pan, Victor Y. *Polynomial and Matrix Computations, Volume 1: Fundamental Algorithms,* Progress in Theoretical Computer Science, Birkhäuser, 1994.

Can89. Canny, John F., *"Generalized Characteristic Polynomials,"* Symbolic and Algebraic Computation (Rome, 1988), pp. 293–299, Lecture Notes in Comput. Sci., 358, Springer, Berlin, 1989.

Can93. _____, *"Improved Algorithms for Sign Determination and Existential Quantifier Elimination,"* Comput. J. **36** (1993), no. 5, pp. 409–418.

Emi94. Emiris, Ioannis Z., *"Sparse Elimination and Applications in Kinematics,"* Ph.D. dissertation, Computer Science Division, U. C. Berkeley (December, 1994).

Emi96. Emiris, Ioannis Z., *"On the Complexity of Sparse Elimination,"* J. Complexity **12** (1996), no. 2, pp. 134–136.

EC95. Emiris, Ioannis Z. and Canny, John F., *"Efficient Incremental Algorithms for the Sparse Resultant and the Mixed Volume,"* Journal of Symbolic Computation, vol. 20 (1995), pp. 117–149.

Ewa96. Ewald, Günter, *Combinatorial Convexity and Algebraic Geometry,* Graduate Texts in Mathematics, 168, Springer-Verlag, New York, 1996.

GKZ94. Gel'fand, I. M., Kapranov, M. M., and Zelevinsky, A. V., *Discriminants, Resultants and Multidimensional Determinants,* Birkhäuser, Boston, 1994.

GK94. Gritzmann, Peter and Klee, Victor, *"On the Complexity of Some Basic Problems in Computational Convexity II: Volume and Mixed Volumes,"* Polytopes: Abstract, Convex, and Computational (Scarborough, ON, 1993), pp. 373–466, NATO Adv. Sci. Inst. Ser. C Math. Phys. Sci., 440, Kluwer Acad. Publ., Dordrecht, 1994.

[10] Note that by expanding our locus of failure slightly (still keeping codimension 1), we can change version 1 from Monte Carlo type to Las Vegas type (cf. remark 4).

GS93. Gritzmann, Peter and Sturmfels, Bernd, *"Minkowski Addition of Polytopes: Computational Complexity and Applications to Gröbner Bases,"* SIAM J. Discrete Math. 6 (1993), no. 2, pp. 246–269.

MC92. Manocha, Dinesh and Canny, John *"Real Time Inverse Kinematics for General 6R Manipulators,"* Technical Report, University of California, Berkeley, 1992.

MP98. Mourrain, Bernard and Pan, Victor, *"Asymptotic Acceleration of Solving Multivariate Polynomial Systems of Equations,"* STOC '98, ACM Press, 1998.

Mum82. Mumford, David, *"Algebraic Geometry I: Complex Algebraic Varieties,"* reprint of 1976 edition, Classics in Mathematics, Springer-Verlag, Berlin, 1995, x+186 pp.

PS85. Preparata, Franco P. and Shamos, Michael Ian, *Computational Geometry: An Introduction,* Texts and Monographs in Computer Science, Springer-Verlag, New York-Berlin, 1985.

Ren87. Renegar, Jim, *"On the Worst Case Arithmetic Complexity of Approximating Zeros of Systems of Polynomials,"* Technical Report, School of Operations Research and Industrial Engineering, Cornell University.

Roj97a. Rojas, J. Maurice, *"Toric Laminations, Sparse Generalized Characteristic Polynomials, and a Refinement of Hilbert's Tenth Problem,"* Foundations of Computational Mathematics, selected papers of a conference, held at IMPA in Rio de Janeiro, January 1997, Springer-Verlag (1997).

Roj97b. _____, *"Linear Algebra for Large Nonlinear Algebra over the Reals,"* extended abstract, Sixth SIAM Conference on Applied Linear Algebra, Snowbird, Utah, October, 1997.

Roj97c. _____, *"Affine Elimination Theory,"* extended abstract, Proceedings of a Conference in Honor of David A. Buchsbaum, October, 1997, Northeastern University.

Roj98a. _____, *"Toric Intersection Theory for Affine Root Counting,"* Journal of Pure and Applied Algebra, to appear.

Roj98b. _____, *"Twisted Chow Forms and Toric Perturbations for Degenerate Polynomial Systems,"* submitted.

Roy95. Roy, Marie-Françoise, *"Basic Algorithms in Real Algebraic Geometry and their Complexity: from Sturm's Theorem to the Existential Theory of Reals,"* Lectures in Real Geometry (Madrid, 1994), pp. 1–67, de Gruyter Exp. Math., 23, de Gruyter, Berlin, 1996.

Stu93. Sturmfels, Bernd, *"Sparse Elimination Theory,"* In D. Eisenbud and L. Robbiano, editors, Proc. Computat. Algebraic Geom. and Commut. Algebra 1991, pages 377–396, Cortona, Italy, 1993, Cambridge Univ. Press.

Stu94. _____, *"On the Newton Polytope of the Resultant,"* Journal of Algebraic Combinatorics, 3: 207–236, 1994.

Stu97. _____, *"Introduction to Resultants,"* Notes from a lecture presented at an AMS short course on Applications of Computational Algebraic Geometry, San Diego, January 6–7, 1997.

Fast Algorithms for Linear Algebra Modulo N [?]

Arne Storjohann and Thom Mulders

Institute of Scientific Computing
ETH Zurich, Switzerland
fstorjoha,mulders@inf.ethz.ch

Abstract. Many linear algebra problems over the ring $\mathbb{Z}_N$ of integers modulo N can be solved by transforming via elementary row operations an $n \times m$ input matrix A to Howell form H. The nonzero rows of H give a canonical set of generators for the submodule of $(\mathbb{Z}_N)^m$ generated by the rows of A. In this paper we present an algorithm to recover H together with an invertible transformation matrix P which satisfies $PA = H$. The cost of the algorithm is $O(nm^{\omega-1})$ operations with integers bounded in magnitude by N. This leads directly to fast algorithms for tasks involving $\mathbb{Z}_N$–modules, including an $O(nm^{\omega-1})$ algorithm for computing the general solution over $\mathbb{Z}_N$ of the system of linear equations $xA = b$, where $b \in (\mathbb{Z}_N)^m$.

1 Introduction

The reduction of a matrix A over a field to reduced row echelon form H is a central topic in elementary linear algebra. The nonzero rows of H give a canonical basis for the row span $\mathrm{S}(A)$ of A, that is, the set of all linear combinations of rows of A. Being able to compute H quickly leads directly to fast algorithms for problems such as: testing whether two matrices have the same row span; testing whether a vector belongs to the row span of a matrix; computing the nullspace of a matrix; solving systems of linear equations.

This paper gives fast algorithms for solving similar linear algebra problems over the ring $\mathbb{Z}_N$ of integers modulo N. If N is prime, then $\mathbb{Z}_N$ is a field and $\mathrm{S}(A)$ is a vector space. Otherwise, $\mathrm{S}(A)$ forms a $\mathbb{Z}_N$-module but not a vector space. Computing over $\mathbb{Z}_N$ is a more subtle problem than computing over a field because of the existence of zero divisors. A classical result states that over a field two matrices in echelon form with the same row span will have the same number of nonzero rows — the rank. Over $\mathbb{Z}_N$ this is not the case. For example, the matrices

$$A = \begin{bmatrix} 4 & 1 & 0 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 8 & 5 & 5 \\ 0 & 9 & 8 \\ 0 & 0 & 10 \end{bmatrix} \quad \text{over } \mathbb{Z}_{12}$$

[?] This work has been supported by grants from the Swiss Federal Office for Education and Science in conjunction with partial support by ESPRIT LTR Project no. 20244 — ALCOM–IT.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 139{150, 1998.
c Springer-Verlag Berlin Heidelberg 1998

have the same row span but not the same number of nonzero rows. Moreover, considered as matrices over $\mathbb{Z}$, both A and B are in Hermite normal form — a canonical form for row spans over $\mathbb{Z}$ (see [5]). A canonical echelon form for row spans in the module $(\mathbb{Z}_N)^m$ is defined by Howell in [4]. Two matrices A and B over $\mathbb{Z}_N$ will have $\mathrm{S}(A) = \mathrm{S}(B)$ if and only if their Howell forms coincide. In the previous example, the Howell form of A and B is

$$H = \begin{bmatrix} 4 & 1 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

In addition to being an echelon form, the Howell form has some additional properties which make it particularly well suited to solving tasks involving $\mathbb{Z}_N$-modules.

We give an asymptotically fast algorithm for computing the Howell form of an $A \in \mathbb{Z}_N^{n \times m}$ with $n \geq m$. The algorithm requires $O(nm^{\omega-1})$ operations with integers bounded in magnitude by N to compute the following: the Howell form H of A; an invertible transformation matrix $P \in (\mathbb{Z}_N)^{n \times n}$ with $PA = H$; a kernel $Y \in (\mathbb{Z}_N)^{n \times *}$ of A. Here, ω is the exponent for matrix multiplication over rings: two $n \times n$ matrices over a ring can be multiplied in $O(n^\omega)$ ring operations. Standard matrix multiplication has $\omega = 3$ whereas the current record in [3] allows $\omega < 2.376$. In this paper we assume that $\omega > 2$.

An important feature of our algorithm is that we return a representation for the $n \times n$ transforming matrix P which allows matrix-vector products involving P to be computed in $O(nm)$ instead of $O(n^2)$ operations when P is dense. Let $A, B \in (\mathbb{Z}_N)^{n \times m}$ be given. We get $O(nm^{\omega-1})$ algorithms for: determining if $\mathrm{S}(A) = \mathrm{S}(B)$, and, if so, returning transformation matrices $P, P^{-1} \in (\mathbb{Z}_N)^{n \times n}$ such that $PA = B$ and $A = P^{-1}B$; computing a canonical spanning set for the union or intersection of the modules generated by the rows of A and B; determining inconsistency or computing a general solution to a system $xA = b$ of linear equations.

The rest of this paper is organised as follows. In Sect. 2 we define some *basic operations* from $\mathbb{Z}_N$ and bound their cost in terms of bit operations. In Sect. 3 we recall the definition and properties of the Howell form and give an iterative algorithm for its computation. In Sect. 4 we give an asymptotically fast algorithm to compute the Howell form of a matrix. Finally, in Sect. 5 we show how to apply the algorithm of the previous section to solving a variety of linear algebra problems over $\mathbb{Z}_N$.

We will frequently write matrices using a block decomposition. An unlabeled block has all entries zero and the generic label $*$ indicates a block with possibly nonzero entries. For convenience, we allow the row and/or column dimension of a matrix or block to be zero.

2 Basic Operations

In this section we define certain basic operations from $\mathbb{Z}_N$ that we will use in the rest of this paper to describe algorithms. Our main complexity results will

be given in terms of numbers of basic operations. We also bound in this section the bit complexity of these basic operations. We will represent elements from the ring $\mathbb{Z}_N$ as integers from the set $S = \{0, 1, \ldots, N-1\}$ and henceforth identify elements from $\mathbb{Z}_N$ with elements from S. For $a, b \in S$ the *basic operations* are:

b1) Basic arithmetic operations: return $c \in S$ such that $c = a + b$, $c = a - b$, $c = ab$ or $c = -a$ in $\mathbb{Z}_N$;
b2) Quo(a, b): when $b \neq 0$, return $q \in S$ such that $a - qb = r$ with $0 \leq r < b$;
b3) Gcd(a, b): return $\gcd(a, b)$;
b4) Div(a, b): when $\gcd(b, N) | a$, return a $c \in S$ such that $bc = a$ in $\mathbb{Z}_N$;
b5) Gcdex(a, b): return $g, s, t, u, v \in S$ such that $g = \gcd(a, b) = sa+tb$, $ua+vb = 0$ and $sv - tu = 1$ in $\mathbb{Z}_N$;
b6) Ann(a): return $c \in S$ such that $c = N/\gcd(a, N)$ in $\mathbb{Z}_N$. Then c generates the ideal of all elements which annihilate a in $\mathbb{Z}_N$;
b7) Stab(a, b): return $c \in S$ such that $\gcd(a + cb, N) = \gcd(a, b, N)$. Then $a + cb$ generates the ideal generated by a and b in $\mathbb{Z}_N$;
b8) Unit(a): when $a \neq 0$, return $c \in S$ such that $\gcd(c, N) = 1$ and $ca = \gcd(a, N)$ in $\mathbb{Z}_N$;

Let $M(t)$ be a bound on the number of bit operations required to multiply two $\lceil t \rceil$-bit integers. Standard arithmetic has $M(t) \in O(t^2)$ while the current record is $M(t) \in O(t \log t \log\log t)$ (see [6]).

¿From [1] we know that we can perform operations b1, b2, b3, b4, b5 and b6 in $O(M(\log N)\log\log N)$ bit operations. We now show that operations b7 and b8 can be performed in the same time. First we give algorithm Split, that for integers D and a will split off the factors in D which are common to a. In this algorithm all operations are over $\mathbb{Z}$.

Algorithm Split
Input: $D, a \in \mathbb{Z}$ such that $D > 0$ and $0 \leq a < D$
Output: $M \in \mathbb{Z}$ such that $M > 0$, $M | D$ and for all prime numbers p we have:

$$p | M \Rightarrow p | a$$
$$p | D/M \Rightarrow p \nmid a$$

for i **to** $\lceil \log\log D \rceil$ **do**
    $a := a^2 \bmod D$
od;
$D/\gcd(a, D)$

Theorem 1. *The previous algorithm is correct. The cost of the algorithm is* $O(M(\log D)\log\log D)$ *bit operations.*

Proof. Let p be a prime number and define $\mathrm{ord}_p(n) = \max\{i \mid p^i | n\}$ for $n \in \mathbb{Z}$. When $a = 0$ it is clear that the algorithm is correct, so assume that $a \neq 0$. Define $a_0 = a$ and $a_{i+1} = a_i^2 \bmod D$. Then for all i, $\mathrm{ord}_p(a_i) \geq \min(\mathrm{ord}_p(D), 2^i \mathrm{ord}_p(a))$. Since $\mathrm{ord}_p(D) \leq \log D$ the correctness of the algorithm now follows.

The complexity of the algorithm follows easily. □

Lemma 1. *We can perform operations b7 and b8 in $O(M(\log N)\log\log N)$ bit operations.*

Proof. To compute $\mathrm{Stab}(a,b)$ first compute $g=\gcd(a,b,N)$. Let $c \in S$ such that $c \equiv \mathrm{Split}(N/g, a/g) \pmod N$. It is easy to see that $\gcd(a/g+cb/g, N/g)=1$. Return c.

To compute $\mathrm{Unit}(a)$ first compute $s,g \in S$ such that $g=\gcd(a,N)$ and $sa \equiv g \pmod N$. Then $\gcd(s,N/g)=1$. When $g=1$, return s. Otherwise, compute $d=\mathrm{Stab}(s,N/g)$. Then $\gcd(s+dN/g,N)=1$ and $(s+dN/g)a \equiv g \pmod N$. Return $c \in S$ such that $c \equiv s+dN/g \pmod N$. ∎

In the sequel we also need to perform the extended stabilization operation: Given $a_0,a_1,\ldots,a_n \in S$, compute $c_1,\ldots,c_n \in S$ such that $\gcd(a_0+c_1a_1+\cdots+c_na_n,N)=\gcd(a_0,a_1,\ldots,a_n,N)$.

Lemma 2. *The extended stabilization operation requires $O(n)$ basic operations.*

Proof. To compute $c_1,\ldots,c_n$ we can apply the stabilization operation in sequence on $(a_0,a_1),(a_0+c_1a_1,a_2),\ldots,(a_0+c_1a_1+\cdots+c_{n-1}a_{n-1},a_n)$. ∎

3 Spans in the Module $(\mathbb{Z}_N)^m$ and the Howell Form

In this section we recall the definition of the Howell form of a matrix which was introduced in [4]. Let $R=\mathbb{Z}_N$ and $A \in R^{n \times m}$ be given. The row span $\mathrm{S}(A)$ of A is the R–submodule of R^m generated by the rows of A, that is, the set of all R-linear combinations of rows of A. The nonzero rows of the Howell form of A give a canonical set of generators for $\mathrm{S}(A)$. For two matrices A and B we have $\mathrm{S}(A)=\mathrm{S}(B)$ if and only if A and B have the same Howell form.

If $U \in R^{n \times n}$ is invertible over R, then $\mathrm{S}(UA)=\mathrm{S}(A)$. Recall that a square matrix U is invertible precisely when U has determinant a unit from R. The matrix U corresponds to a sequence of elementary row operations: interchanging two rows; multiplying a row by a unit; adding a multiple of one row to a different row. Any matrix A over R can be transformed using only elementary row operations to a matrix H that satisfies the following conditions:

e1) Let r be the number of nonzero rows of H. Then the first r rows of H are nonzero.
e2) For $1 \le i \le r$ let $H[i,j_i]$ be the first nonzero entry in row i. Then $j_1<j_2<\ldots<j_r$.

A matrix H having properties e1 and e2 is said to be in row echelon form. For example, the first matrix in (1) is in row echelon form. Further elementary row operations can be applied so that H satisfies:

e3) $H[i,j_i] \mid N$ for $1 \le i \le r$.
e4) For $1 \le k<i \le r$, $0 \le H[k,j_i]<H[i,j_i]$.

A matrix H which satisfies e1, e2, e3 and e4 is said to be in reduced row echelon form. For example, in (1) the first matrix can be transformed to the second matrix which is in reduced row echelon form. Finally, a matrix is said to be in Howell form if it is in reduced row echelon form and also satisfies:

e5) (Howell property) When $a \in \mathrm{S}(H)$ has zeros as its first $(j_i - 1)$ components then $a \in \mathrm{S}(H[i \ldots m])$, where $H[i \ldots m]$ denotes the matrix comprised of the ith through mth row of H.

For example, in (1) the third matrix is the Howell form of the first two matrices.

$$\begin{bmatrix} 4 & 1 & 10 \\ 0 & 0 & 5 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 4 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 4 & 1 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ over } \mathbb{Z}_{12} \ . \tag{1}$$

Theorem 2. *For any $A \in R^{n \times m}$ there exists a unique $H \in R^{m \times m}$ that is in Howell form and such that $\mathrm{S}(H) = \mathrm{S}(A)$.*

Proof. See [4].

The matrix H from Theorem 2 is called the Howell form of A. The theorem shows that the Howell form of matrices can be used to determine whether two matrices have the same row span.

A matrix H which satisfies e1, e2 and e5 is said to be in weak Howell form. We say that H is a (weak) Howell basis for A if H has no zero rows, H is in (weak) Howell form and $\mathrm{S}(H) = \mathrm{S}(A)$.

Definition 1. *Let $A \in R^{n \times m}$. We define the kernel $\ker(A)$ of A as $\{r \in R^n \mid rA = 0\}$. The kernel of A is an R-submodule of R^n. $K \in R^{n \times n}$ is called a kernel for A if $\mathrm{S}(K) = \ker(A)$.*

The next two lemmas, for which we omit the proofs, give two key properties of the Howell form.

Lemma 3. *Let $H = \left[\begin{array}{c|c} H_1 & F \\ \hline & H_2 \end{array}\right]$, K_i a kernel for H_i $(i = 1, 2)$ and $K_1 F = S H_2$. Then $K = \left[\begin{array}{c|c} K_1 & -S \\ \hline & K_2 \end{array}\right]$ is a kernel for H.*

Lemma 4. *Let $H = \left[\begin{array}{c|c} H_1 & F \\ \hline & H_2 \end{array}\right]$ such that H_1 and H_2 are in weak Howell form and H_1 contains no zero rows. Let K be a kernel for H_1 and suppose that $\mathrm{S}(KF) \subseteq \mathrm{S}(H_2)$. Then H is in weak Howell form.*

Next we will give an algorithm to compute the Howell form of a matrix. The algorithm is in the same spirit as Howell's constructive proof for the existence of the form (see [4]). Following [2], we first triangularize the input matrix and then keep the work matrix in triangular form. This saves space by avoiding the

need to augment the work matrix with m extra rows as in [4]. The correctness of the algorithm follows from Lemma 4.

Algorithm Howell
Input: $A \in R^{n \times m}$
Output: The Howell form of A
if $n < m$ **then**
    Augment A with zero rows to make it square;
    $n := m$
;
Comment Put A in upper triangular form
for j **from** 1 **to** m **do**
    for i **from** $j+1$ **to** n **do**
        $(g, s, t, u, v) := \mathrm{Gcdex}(A[j,j], A[i,j])$;
        $\begin{matrix} A[j,\ ] \\ A[i,\ ] \end{matrix} := \begin{matrix} s & t \\ u & v \end{matrix} \begin{matrix} A[j,\ ] \\ A[i,\ ] \end{matrix}$
    od
od;
Comment Put A in Howell form
Augment A with one zero row;
for j **from** 1 **to** m **do**
    if $A[j,j] \neq 0$ **then**
        $A[j,\ ] := \mathrm{Unit}(A[j,j])A[j,\ ]$;
        for i **from** 1 **to** $j-1$ **do**
            $A[i,\ ] := A[i,\ ] - \mathrm{Quo}(A[i,j], A[j,j])A[j,\ ]$
        od;
        $A[n+1,\ ] := \mathrm{Ann}(A[j,j])A[j,\ ]$
    else
        $A[n+1,\ ] := A[j,\ ]$
    ;
    for i **from** $j+1$ **to** m **do**
        $(g, s, t, u, v) := \mathrm{Gcdex}(A[i,i], A[n+1,i])$;
        $\begin{matrix} A[i,\ ] \\ A[n+1,\ ] \end{matrix} := \begin{matrix} s & t \\ u & v \end{matrix} \begin{matrix} A[i,\ ] \\ A[n+1,\ ] \end{matrix}$
    od
od;
Move all nonzero rows to the top of A;
Output first m rows of A

A straightforward count proves the following theorem.

Theorem 3. *Algorithm Howell requires* $O(m^2 \max(n, m))$ *basic operations.*

4 Asymptotically Fast Computation of the Howell Form

All matrices will be over the ring $R = \mathbb{Z}_N$ and complexity results are given in terms of basic operations. Let $M(a, b, c)$ be a bound on the number of ba-

sic operations required to multiply an $a \times b$ by a $b \times c$ matrix over R. Using $M(n,n,n) \in O(n^{\omega})$ together with a block decomposition we have

$$M(a,b,c) \in \begin{cases} O(abc^{\omega-2}) & \text{if } c = \min(a,b,c) \\ O(acb^{\omega-2}) & \text{if } b = \min(a,b,c) \\ O(bca^{\omega-2}) & \text{if } a = \min(a,b,c) \end{cases} \tag{2}$$

Definition 2. *Let $A \in R^{m \times n}$ and k be such that $0 \le k \le n$ and $n-k \le r$ where r is the number of nonzero rows in the Howell form of A. An* index k transform *of A is a 6-tuple of matrices (Q, U, C, H, K, S) which satisfy and can be written using a conformal block decomposition as*

$$\begin{bmatrix} Q & \\ & I \end{bmatrix} \begin{bmatrix} I & \\ & U \end{bmatrix} \begin{bmatrix} I & \\ & C \end{bmatrix} \begin{bmatrix} A \\ \bar{A} \end{bmatrix} = \begin{bmatrix} T \\ \bar{A} \\ H \end{bmatrix} \tag{3}$$

with U invertible, $\bar{A}$ the first k rows of A, H a weak Howell basis for A, K a kernel for H and S such that $\bar{A} = SH$.

Remark 1. When (Q, U, C, H, K, S) is an index k transform of A, then a kernel for T is given by

$$W = \begin{bmatrix} I & -S & \\ & K & \\ & & I \end{bmatrix}. \tag{4}$$

A kernel for A is given by $WQUC$.

Note that if $k = 0$ then T is a weak Howell form of A. Later we will show how to transform a weak Howell form to a Howell form. First we bound the cost of computing an index k transform. Define $T(n,m,r)$ to be a bound on the number of basic operations required to compute an index k transform for an $A \in R^{m \times n}$ which has a Howell basis with r rows. Our result is the following.

Theorem 4. *$T(n,m,0) \in O(nm)$. If $r > 0$ then $T(n,m,r) \in O(nmr^{\omega-2})$.*

Before proving Theorem 4 we give some intermediate results. The algorithm supporting the theorem is recursive and the following technical result will be used to bound the cost of the merge step. Let I_s denote the $s \times s$ identity matrix.

Lemma 5. *If*

$$Q_1 = \begin{bmatrix} I_k & & & \\ & I_{r_1} & & \\ & \bar{q}_1 & I_{r_2} & \\ & q_1 & & I \end{bmatrix}, \quad U_1 = \begin{bmatrix} I_k & & & \\ & u_1 & & \\ & & I_{r_2} & \\ & & & I \end{bmatrix}, \quad C_1 = \begin{bmatrix} I_k & & & & \\ c_1 & I_{r_1} & \bar{d}_1 & d_1 & \\ & & I_{r_2} & & \\ & & & & I \end{bmatrix},$$

$$Q_2 = \begin{array}{|c|c|c|c|}\hline I_k & & & \\ \hline & I_{r_1} & & \\ \hline & & I_{r_2} & \\ \hline & & q_2 & I \\ \hline \end{array}, \quad U_2 = \begin{array}{|c|c|c|c|}\hline I_k & & & \\ \hline & I_{r_1} & & \\ \hline & & u_2 & \\ \hline & & & I \\ \hline \end{array}, \quad \bar{C}_2 = \begin{array}{|c|c|c|c|}\hline I_k & & & \\ \hline & I_{r_1} & & \\ \hline c_2 & \bar{c}_2 & I_{r_2} & d_2 \\ \hline & & & I \\ \hline \end{array},$$

are all in $R^{n \times n}$ *and the block decomposition is conformal, then*

$$Q_2U_2\bar{C}_2Q_1U_1C_1 = \begin{array}{|c|c|c|c|}\hline I_k & & & \\ \hline & I_{r_1} & & \\ \hline & & I_{r_2} & \\ \hline & q_1 & q_2 & I \\ \hline \end{array}_{Q} \begin{array}{|c|c|c|c|}\hline I_k & & & \\ \hline & u_1 & u_{12} & \\ \hline & u_{21} & u_{22} & \\ \hline & & & I \\ \hline \end{array}_{U} \begin{array}{|c|c|c|c|}\hline I_k & & & \\ \hline c_{11} & I_{r_1} & & c_{12} \\ \hline c_2 & & I_{r_2} & d_2 \\ \hline & & & I \\ \hline \end{array}_{C} \tag{5}$$

where

$$\begin{aligned} c_{11} &= c_1 - \bar{d}_1 c_2 \\ c_{12} &= d_1 - \bar{d}_1 d_2 \\ u_{21} &= u_2(\bar{q}_1 + d_2 q_1 + \bar{c}_2)u_1 \\ u_{12} &= u_1 \bar{d}_1 \\ u_{22} &= u_2 + u_2(\bar{q}_1 + d_2 q_1 + \bar{c}_2)u_1 \bar{d}_1 \\ &= u_2 + u_{21}\bar{d}_1 \end{aligned}$$

Moreover, the computation of Q, U *and* C *requires at most* $O(n(r_1 + r_2)^{\omega-1})$ *basic operations.*

Proof. The first part of the lemma follows from a straightforward computation. The cost of computing Q, U and C is $O(M(a, b, c))$ where at least two of the dimensions a, b and c equal r_1 or r_2 and the third dimension is bounded by n. The second part of the lemma now follows from (2). ∎

Lemma 6. *There exists an absolute constant* c*, such that if* $m = m_1 + m_2$*, then*

$$T(n, m, r) \quad T(n, m_1, r_1) + T(n, m_2, r_2) + cnm_2r^{\omega-2}$$

for some $r_1, r_2 \quad 0$ *with* $r = r_1 + r_2$*.*

Proof. Let $A \in R^{n \times m}$ and k be such that $0 \quad k \quad n$ and $n - k \quad r$ where r is the number of rows in a Howell basis of A. We will compute an index k transform (Q, U, C, H, K, S) for A by merging transforms for matrices A_1 and A_2 of column dimension m_1 and m_2 respectively. The result will follow if we bound by $O(nm_2r^{\omega-2})$ basic operations the cost of constructing A_1 and A_2 and merging their transforms. Choose A_1 to be the first m_1 columns of A. Compute an index k transform $(Q_1, U_1, C_1, H_1, K_1, S_1)$ for A_1 at a cost bounded by $T(n, m_1, r_1)$ basic operations where r_1 is the row dimension of H_1. We now have

$$W_1Q_1U_1C_1 \begin{array}{|c|c|}\hline \bar{A}_1 & E \\ \hline \end{array}_{A} = \begin{array}{|c|c|}\hline I & -S_1 \\ \hline & K_1 \\ \hline \end{array}_{W_1} \begin{array}{|c|c|}\hline \bar{A}_1 & E \\ \hline H_1 & F \\ \hline \end{array} = \begin{array}{|c|}\hline E - S_1F \\ \hline K_1F \\ \hline \end{array} \tag{6}$$

where E and F are new labels. Note that the submatrix of A comprised of blocks $\bar{A}_1$ and E is $\bar{A}$ of (3). Choose A_2 as the last m_2 columns of the matrix on the right hand side of (6). Note that the last $n-k-r_1$ rows of A_2 is the unmodified trailing $(n - k - r_1) \quad m_2$ block of A. Because of the special structure of the matrices involved in the multiplications, we can recover A_2 and F in $O(nm_2r_1^{l-2})$ basic operations. In particular, note that S_1, K_1 and every subblock of Q_1, U_1 and C_1 which is neither the identity nor the zero block has column and/or row dimension equal to r_1. Compute an index $k + r_1$ transform $(Q_2, U_2, C_2, H_2, K_2, S_2)$ for A_2 at a cost bounded by $T(n, m_2, r_2)$ basic operations where r_2 is the row dimension of H_2. Now we show how to recover an index k transform (Q, U, C, H, K, S) for A. Define

$$H = \left[\begin{array}{c|c} H_1 & F \\ \hline & H_2 \end{array}\right], \quad K = \left[\begin{array}{c|c} K_1 & -S_{22} \\ \hline & K_2 \end{array}\right] \text{ and } S = \left[\begin{array}{c|c} S_1 & S_{21} \end{array}\right] \text{ where } S_2 = \left[\begin{array}{c} S_{21} \\ \hline S_{22} \end{array}\right].$$

That K is a kernel for H follows from Lemma 3. That H is in weak Howell form follows from Lemma 4. That $SH = \bar{A}$ follows from direct computation. We now show how to recover Q, U and C such that $QUCA = T$ where T is as in (3). At a cost of $O(nr^{l-1})$ basic operations compute

$$\bar{C}_2 = \left[\begin{array}{c|c|c} I & & \\ \hline \bar{a} & I & b \\ \hline & & I \end{array}\right] \text{ where } C_2 = \left[\begin{array}{c|c|c} I & & \\ \hline a & I & b \\ \hline & & I \end{array}\right] \text{ and } \bar{a} = a \left[\begin{array}{c|c} I & -S_2 \\ \hline & K_1 \end{array}\right].$$

A straightforward multiplication verifies that $Q_2U_2\bar{C}_2Q_1U_1C_1A = T$. The matrices Q, U and C of the index k transform for A can now be recovered in $O(nr^{l-1})$ basic operations using Lemma 5. That H is a weak Howell basis for A follows from the fact that H is a weak Howell basis for T. This also shows that $r_1 + r_2 = r$. ∎

Lemma 7. *$T(n, 1, r) \in O(n)$.*

Proof. Let $A \in R^{n \quad 1}$, $0 \quad k \quad n$ and $n - k \quad r$ where r is 0 if A is the zero matrix and 1 otherwise. If $r = 0$ then H is $0 \quad 1$, K is $0 \quad 0$, S is $k \quad 0$ and we can choose $Q = U = C = I_n$. Assume henceforth that $r = 1$. Using the extended stabilization operation from Lemma 2, recover a C of the correct shape such that $\gcd((CA)_{k+1;1}, N)$ is the gcd of all entries in A and N. Set $H = [a]$, where $a = (CA)_{k+1;1}$. Set $K = [\mathrm{Ann}(a)]$. Set $Q = I_n$ except with $Q_{i;k+1} = -\mathrm{Div}(A[i, 1], a)$ for $k + 2 \quad i \quad n$. Set S to be the $k \quad 1$ matrix with $S[i, 1] = \mathrm{Div}(A[i, 1], a)$ for $1 \quad i \quad k$. Set $U = I_n$. ∎

Lemma 8. *For $\alpha, x, y \in \mathbb{R}$ such that $0 < \alpha < 1$ and $x, y > 0$ we have $x \quad + y \quad 2^{1-} \quad (x + y)$.*

Proof. The function $z \mapsto \frac{1+z}{(1+z)}$ has for $z > 0$ an absolute maximum at $z = 1$ with value 2^{1-}. By substituting y/x for z into this function, the lemma follows easily. ∎

We now return to the proof of Theorem 4.

Proof. (of Theorem 4) Let $A \in R^{n \times m}$. By augmenting A with at most $m-1$ zero columns we may assume without loss of generality that m is a power of two. This shows that it will be sufficient to prove the theorem for the special case when m is a power of 2. Let c be the absolute constant of Lemma 6. By Lemma 7 we have that $T(n,1,r) \in O(n)$. Choose an absolute constant e such that $T(n,1,r) \leq en$. From Lemma 6 it now follows easily that $T(n,m,0) \leq enm$. Now choose an absolute constant d such that $c/2 \leq d(1-2^{2-\omega})$ and $d/2 \geq e/2 + c/2$. We claim that for $r > 0$

$$T(n,m,r) \leq dnmr^{\omega-2} . \tag{7}$$

We will prove (7) by induction on $\log_2 m$. Note that (7) is true for $m = 1$. Assume that (7) is true for some power of two m. Then

$$T(n,2m,r) \leq T(n,m,r_1) + T(n,m,r_2) + cnmr^{\omega-2}$$

for some $r_1, r_2 \geq 0$ with $r_1 + r_2 = r$. When r_1 and r_2 are both nonzero we get the following:

$$\begin{aligned} T(n,2m,r) &\leq dnm(r_1^{\omega-2} + r_2^{\omega-2}) + cnmr^{\omega-2} \\ &\leq 2nmr^{\omega-2}(2^{2-\omega}d + c/2) \quad \text{(Lemma 8)} \\ &\leq dn(2m)r^{\omega-2}. \end{aligned}$$

When $r_1 = 0$ we get the following:

$$\begin{aligned} T(n,2m,r) &\leq enm + dnmr^{\omega-2} + cnmr^{\omega-2} \\ &\leq 2nmr^{\omega-2}(e/2 + d/2 + c/2) \\ &\leq dn(2m)r^{\omega-2}. \end{aligned}$$

The case $r_2 = 0$ is similar. ∎

Now we will show how we can transform a matrix in weak Howell form to Howell form.

Corollary 1. *Let (Q,U,C,H,K) be an index 0 transform for an $A \in R^{n \times m}$. An index 0 transform (Q^0,U^0,C,H^0,K^0) for A which has H^0 in Howell form can be computed in $O(nr^{\omega-1})$ basic operations where r is the number of rows in H.*

Proof. If $r = 0$ then nothing needs to be done. Assume $r > 0$. Recover an invertible and diagonal $D \in R^{r \times r}$ such that DH satisfies property e3. This requires r basic operations of type b7. Let T be the submatrix of DH comprised of columns $[j_1, j_2, \ldots, j_r]$ where j_i is as in property e2 of the Howell form. Recover a unit upper triangular matrices R such that RDH satisfies property e4. Next recover R^{-1}. Both R and R^{-1} can be recovered in $O(r^\omega)$ basic operations using the algorithm in [7, Theorem 3]. Set $Q^0 = \mathrm{diag}(RD, I_{n-r})\, Q\, \mathrm{diag}(D^{-1}R^{-1}, I_{n-r})$, $U^0 = \mathrm{diag}(RD, I_{n-r})\, U$ and $K^0 = KD^{-1}R^{-1}$. Correctness follows easily and the cost follows from (2). ∎

Since the Howell form of an invertible matrix is the identity matrix we can obtain the inverse of an invertible matrix U as the transforming matrix from U to its Howell form. We get the following corollary:

Corollary 2. *The inverse of an invertible matrix in $R^{n \times n}$ can be computed in $O(n^{\omega})$ basic operations.*

5 Some Applications

Let $R = \mathbb{Z}_N$ and $A_i \in R^{n_i \times m}$ be given for $i = 1, 2$. By augmenting with 0-rows, we may assume without loss of generality that $n = n_1 \geq n_2 \geq m$. Let A denote a copy of A_1 and let r be the number of rows in a Howell basis for A. Note that $r \leq m$ and in the worst case $r = m$. For brevity we give space and time bounds in terms of n and m only.

We propose solutions to some basic tasks involving modules. A worst case running time bound of $O(nm^{\omega-1})$ basic operations for all the tasks follows from (2), Theorem 4 and Corollaries 1 and 2. For some tasks we state in addition the cost under the assumption that some quantities have been pre-computed. The correctness of the proposed solutions are left as an exercise.

Task 1: Kernel computation [Find a kernel $Y \in R^{n \times n}$ for A.]
Compute an index 0 transform (Q, U, C, H, K) for A. By Remark 1 we can choose

$$Y = \left[\begin{array}{c|c} WQU & C \\ \hline I_r & \\ \hline I_{n-r} & I_{n-r} \end{array}\right]. \tag{8}$$

Return the decomposition for Y as the product of the two matrices shown in (8). This has two advantages. First, both WQU and C will have only $O(nr)$ nonzero entries; their product may have $O(n^2)$ nonzero entries. Second, premultiplying a vector by C and then by WQU costs only $O(nr)$ basic operations.

Task 2: Equality of spans. [Determine if $\mathrm{S}(A_1) = \mathrm{S}(A_2)$.]
By augmenting A_2 with 0-rows, we may assume without loss of generality that $n_1 = n_2$. Compute an index 0 transform $(Q_i, U_i, C_i, H_i, K_i)$ for A_i which has H_i in Howell form $(i = 1, 2)$. Then $\mathrm{S}(A_1) = \mathrm{S}(A_2)$ if and only if $H_1 = H_2$. A transformation matrix P such that $A_1 = PA_2$ and $P^{-1}A_1 = A_2$ is given by $P = (Q_1U_1C_1)^{-1}Q_2U_2C_2$. A straightforward multiplication will verify that

$$P = \left[\begin{array}{c|c} (2I - C_1) & ((Q_2 - Q_1)U_1 + I) \\ \hline I_r & \\ \hline & I_{n-r} \end{array}\right] \left[\begin{array}{c|c} I_r & \\ \hline & I_{n-r} \end{array}\right] \left[\begin{array}{c|c} U_1^{-1}U_2 & \\ \hline & I_{n-r} \end{array}\right] \left[\begin{array}{c|c} I_r & C_2 \\ \hline & I_{n-r} \end{array}\right]. \tag{9}$$

As was the case for the kernel, return the decomposition for P as the product of the four matrices shown in (9). In particular, if $X_1 \in R^{k \times n}$, then an X_2 such that $X_1A_1 = X_2A_2$ can be recovered as $X_2 \leftarrow X_1P$ in $O(nmk^{\omega-2})$ basic operations for $k \leq m$ and $O(nkm^{\omega-2})$ basic operations for $k > m$.

Task 3: Union of modules. [Find the Howell basis for $\mathrm{S}(A_1) \cap \mathrm{S}(A_2)$.]
Return the Howell basis for $[\, A_1^T \mid A_2^T \,]^T$. Here, X^T denotes the transpose of X.

Task 4: Intersection of modules. [Find the Howell basis for $\mathrm{S}(A_1) \setminus \mathrm{S}(A_2)$.]
Compute a kernel Y for $[\, A_1^T \mid A_2^T \,]^T$ as in (8). Return the Howell basis for $Y[\, A_1^T \mid 0 \,]^T$.

Task 5: Testing containment. [Determine whether or not $b \in \mathrm{S}(A)$.]
Compute an index 0 transform (Q, U, C, H, K) for A. Recover a row vector y such that

$$\left[\begin{array}{c|c} 1 & y \\ \hline & I \end{array}\right] \left[\begin{array}{c|c} 1 & b \\ \hline & H \end{array}\right] = \left[\begin{array}{c|c} 1 & b^0 \\ \hline & H \end{array}\right]$$

with the right hand side in Howell form. Then $b \in \mathrm{S}(A)$ if and only if $b^0 = 0$. If $b \in \mathrm{S}(A)$, then $xA = b$ where $x = [\, y \mid 0 \,]UC$. Assuming that the index 0 transform for A is precomputed, testing containment and recovering x requires $O(m^2)$ and $O(nm)$ basic operations respectively.

Task 6: Solving systems of linear equations. [Find a general solution to $xA = b$.]
Determine containment of b in $\mathrm{S}(A)$ using Task 5. If $b \notin \mathrm{S}(A)$ then return "no solution exists". If $b \in \mathrm{S}(A)$, find a y such that $yA = b$ using Task 5. Return (y, Y) where Y is a kernel for A as in (8). Every solution x to $xA = b$ can be expressed as y plus some linear combination of the rows of Y.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. E. Bach. Linear algebra modulo N. Unpublished manuscript., December 1992.
3. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.
4. J. A. Howell. Spans in the module $(\mathbb{Z}_m)^s$. *Linear and Multilinear Algebra*, 19:67—77, 1986.
5. M. Newman. *Integral Matrices*. Academic Press, 1972.
6. A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing*, 7:281–292, 1971.
7. A. Storjohann and G. Labahn. Asymptotically fast computation of Hermite normal forms of integer matrices. In Y. N. Lakshman, editor, *Proc. Int'l. Symp. on Symbolic and Algebraic Computation: ISSAC '96*, pages 259–266. ACM Press, 1996.

A Probabilistic Zero-Test for Expressions Involving Roots of Rational Numbers

Johannes Blömer

Institut für Theoretische Informatik
ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland

Abstract. Given an expression E using $+, -, \cdot, /$ with operands from $\mathbb{Z}$ and from the set of real roots of integers, we describe a probabilistic algorithm that decides whether $E = 0$. The algorithms has a one-sided error. If $E = 0$, then the algorithm will give the correct answer. If $E \neq 0$, then the error probability can be made arbitrarily small. The algorithm has been implemented and is expected to be practical.

1 Introduction

In this paper we consider the following problem. Given a real radical expression without nested roots, that is, an expression E defined with operators $+, -, \cdot, /$, with integer operands and operands of the form $\sqrt[d]{n}; d \geq 2 \in \mathbb{N}; n \geq 2 \in \mathbb{Z}; \sqrt[d]{n} \geq 2 \in \mathbb{R}$. We want to decide whether the expression E is zero. We describe an efficient, probabilistic algorithm to solve this problem. If the expression is zero, the algorithm will give the correct answer. If E is non-zero, the probability that the algorithms declares E to be zero, can be made arbitrarily small. The algorithm is not based on root separation bounds. Unlike algorithms based on root separation bounds, the algorithm has a worst-case running time that does not depend exponentially on the number of input roots. Similarly, the algorithm improves the algorithm in [2]. In that paper expressions are restricted to sums of roots. Of course, turning an arbitrary expression with k input roots into a sum of roots, creates a sum with up to 2^k terms. Again, the new algorithms avoids this behavior.

Tests in computer programs often can be reduced to determining the sign of a radical expression as described above. This is particularly true for problems in computational geometry (see for example [6],[12], [15],[16]). Computing the sign of a radical expression E obviously is a harder problem than deciding whether the expression is zero. Currently, any sign detecting algorithm is based on root separation bounds. That is, the algorithm first computes a bound b such that if E is non-zero then $|E| > 2^{-b}$ (see [7],[13] for the best bounds currently available). In a second step, it approximates E with absolute error less than 2^{-b}. However, experiments often show that if the expression E is close to zero, then E actually is zero. Here, by "close to zero" we mean that computing E with ordinary floating-point arithmetic does not allow to infer the sign of E. In these situations an efficient zero-test can be used as follows. To determine the sign of an expression E, first compute E using machine-provided floating-point arithmetic. If this

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 151{162, 1998.
c Springer-Verlag Berlin Heidelberg 1998

allows you to detect the sign, stop. Otherwise, use the zero-test to determine whether E is zero. If this is the case, stop. Otherwise, approximate E with accuracy 2^{-b} to detect the sign of E. Here 2^{-b} is the accuracy required by the root separation bound. As mentioned, experiments indicate that in many situations the most expensive, third step hardly ever will be necessary.

A second application for a zero-test is in detecting degeneracies in geometric configurations. Here one needs to distinguish between two different types of degeneracies. One is caused by the use of finite-precision arithmetic. These degeneracies one usually wants to remove. The other degeneracies are problem-inherent degeneracies which one may want to keep. The problem-inherent degeneracies can often be detected by a zero-test as provided by the algorithm described in this paper. Degeneracies caused by finite-precision arithmetic then can be removed by some perturbation scheme.

Let us briefly outline the algorithm described in this paper. The basic idea, which originates in [8], is as follows. If α is an algebraic integer and α_i are its conjugates, then either $\alpha = 0$ or α and all its conjugates are non-zero. Therefore, rather than testing whether $\alpha \stackrel{?}{=} 0$, we may choose any conjugate α_i of α and check whether $\alpha_i \stackrel{?}{=} 0$. The simple but fundamental observation of [8] is, that although $|\alpha| \stackrel{?}{=} 0$ may be small, with high probability the absolute value of a random conjugate α_i of α is not too small. Hence a moderately accurate approximation to α_i will reveal that α_i, and therefore α, is non-zero.

Let us apply this idea to a radical expression E. For the sake of simplicity, we restrict ourselves to division-free expressions E involving only square roots $\sqrt{n_1}, \ldots, \sqrt{n_k}$. To apply the method described above we need to be able to generate a random conjugate $\overline{E}$ of E. It is well-known that the conjugates of E can be obtained be replacing the roots $\sqrt{n_1}, \ldots, \sqrt{n_k}$ by roots $\varepsilon_1\sqrt{n_1}, \ldots, \varepsilon_k\sqrt{n_k}$, where $\varepsilon_j = \pm 1$. Unfortunately, not all sign combinations lead to a conjugate. To see this, consider the following toy example, $E = \sqrt{2}\sqrt{3} - \sqrt{6}$. Interpreting all square roots as positive real numbers, we see that $E = 0$. However, choosing the sign combination $-1, -1, -1$ leads to $\sqrt{2}\sqrt{3} + \sqrt{6} \neq 0$. In particular, this sign combination does not lead to a conjugate of E. Of course, this combination fails to generate a conjugate because $\sqrt{2}\sqrt{3} = \sqrt{6}$. In general, we need to find multiplicative dependencies between the input roots. To determine these multiplicative dependencies we use a well-known procedure called factor refinement (see for example [1]). Its running time is $O(\log^2(n))$, where $n = \prod n_i$. Once the dependencies have been determined and a random conjugate $\overline{E}$ has been generated, the algorithm to check whether $E = 0$ simply approximates $\overline{E}$.

What is the accuracy required for this approximation in order to guarantee an error probability less than 1/2, say? By the result of [8] the approximation needs to have accuracy 2^{-B}, where 2^B is an upper bound on the absolute value of the conjugates of E. To obtain such an estimate we use a bound $\mathrm{u}(E)$ first introduced in [7]. A similar, but slightly worse bound is obtained in [13]. $\mathrm{u}(E)$ is easy to compute and, in the worst case, is the best possible upper bound on $|E|$ itself. Summarizing, except for an overhead of $O(\log^2(n))$, the running time of our algorithm will be the time needed to compute $\overline{E}$ with absolute error $2^{-\mathrm{u}(E)}$.

Since E and $\overline{E}$ differ only in the signs of the input radicals, in the worst case this is the time needed to compute E with absolute error $2^{-u(E)}$.

As mentioned above, this compares favorably to algorithms based on separation bounds. Take the bounds in [7], which are the best bounds currently available. To decide whether an expression E as above is zero, an approximation to E with absolute error less than $2^{-2^k u(E)}$ is computed, where k is the number of input square roots. Hence the quality of the approximation required by the algorithm in [7] differs by a factor of 2^k from the quality required by our algorithm. We achieve this reduction by a preprocessing step requiring time $O(\log^2(n))$. Even for moderately small values of k this is time well spent.

This leads to the main question this paper raises. The bounds in [7] not only apply to expressions as defined above, these bounds also apply to expressions with nested roots[1]. That is, in the expression we not only allow the operations $+; -; \ ; =$, we also allow operators of the form $\sqrt[q]{}$. Although we feel that the class of expressions the algorithm described in this paper can handle is the most important subclass of the expressions dealt with in [7], it would be very interesting to generalize the algorithm to the class of nested radical expressions. Note that denesting algorithms as in [10],[9], and [4] implicitly provide a zero-test for these expressions. But denesting algorithms solve are a far more general problem than testing an expression for zero. Accordingly, denesting algorithms, if used as zero-tests, are less efficient than algorithms based on roots separation bounds.

The algorithm described in this paper has been implemented. So far no effort has been made to optimize its running time, but the algorithm seems to be practical. The main objective of the implementation was to compare the probabilistic behavior observed in practice with the probabilistic guarantees provided by the theory. The data set is still rather small. As expected, the algorithms performs better than predicted by theory. To give some specific numbers, we tested the algorithms on the determinant of 3 3 matrices whose entries are sums of square roots. The integers involved are 5-digit integers. We generated matrices whose determinant is less than 10^{-6}. Random conjugates of these determinants consistently fell in the range from $10^5 - 10^7$. We never found an example where the random conjugate of a determinant was smaller than the determinant itself.

The paper is organized as follows. In Section 2 the main definitions are given, and we formally state the main result. In Section 3 we describe the algorithm for division-free expression and analyze its running time. In Section 4 we analyze the error probability of the algorithm. In Section 5 we briefly show how to generalize the algorithm to expressions with divisions.

2 De nitions and Statement of Results

Throughout this paper, we only deal with roots of integers $\sqrt[d]{n}; d \in \mathbb{N}; n \in \mathbb{Z}$. The symbol $\sqrt[d]{n}$ does not specify a unique complex number. However, when

[1] It should be noted, that it is unknown whether the bounds in [7] can be improved, if nested roots are not allowed.

we use this symbol, we will always assume that some specific d-th root of n is referred to. How this particular root is specified is irrelevant, except that the specification must allow for an efficient approximation algorithm. Usually we will require that a root $\sqrt[d]{n}$ is a real number. In this case, we assume that n is positive.

Our definition of a radical expression is the same as the definition of a straight-line program over the integers, except that we allow roots of integers as inputs. To be more specific, for a directed acyclic graph (dag) the nodes of in-degree 0 will be called *input nodes*. The nodes of out-degree 0 will be called *output nodes*. Nodes that are not inputs are called *internal nodes*.

Definition 1. *A depth 1 radical expression E over the integers is a directed acyclic graph (dag) with a unique output node and in-degree exactly 2 for each internal node. Each input node is labeled either by an integer or by a root of an integer. Each internal node is labeled by one of the arithmetic operations $+, -, \cdot, /$. If no internal node is labeled by $/$, then E is called a division-free radical expression. If the inputs are labeled by integers and real roots of integers, then the expression is called a real radical expression. In either case, the input labels that are integers are called the input integers and the remaining input labels are called the input radicals.*

These expressions are called depth 1 expressions, since we do not allow operators of the form $\sqrt[d]{}$ for the internal nodes. Hence there are no nested roots in the expression. In this paper all expressions are depth 1 expression. In the sequel, we will omit the prefix "depth 1". Similarly, the suffix "over the integers" will be omitted.

For a radical expression E with e edges and with input integers $m_1, \ldots, m_k$ and input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_l]{n_l}$ the *size of* E, denoted by size(E), is defined as

$$e + \sum_{i=1}^{k} \log|m_i| + \sum_{j=1}^{l} \log|n_j| + \sum_{j=1}^{l} d_j,$$

where the logarithms are base 2 logarithms. Remark that size(E) depends linearly on d_i rather than on $\log(d_i)$.

To each node v of a radical expression we can associate a complex number, called the *value* val(v) of that node. The value of an input node is the value of its label. If v is an internal node labeled with $\circ \in \{+, -, \cdot, /\}$ and edges from nodes v_1, v_2 are directed into v, then $\mathrm{val}(v) = \mathrm{val}(v_1) \circ \mathrm{val}(v_2)$. The *value* val($E$) *of a radical expression* is the value of the output node of E.

It is easy to construct an expressions with $O(n)$ edges whose value is double-exponential in n. This shows that in general one cannot even write down val(E) in time polynomial in size(E). One way to avoid this problem is to restrict expressions E to trees. In this case, $\log(|\mathrm{val}(E)|)$ is bounded by a polynomial in size(E). In this work we follow a different approach. For a radical expression E we define an easily computable bound u(E) such that for a division-free expression u(E) is an upper bound on $|\mathrm{val}(E)|$. Later we will see that arbitrary expressions

E can be written as the quotient of two division-free radical expressions $E_1; E_2$ such that $|\mathrm{val}(E_1)| \quad \mathrm{u}(E)$. The definition of the bound $\mathrm{u}(E)$ follows [7].

Let E be a radical expression and let v be a node of E. For an input node v the bound $\mathrm{u}(v)$ is the absolute value of its label. $\mathrm{l}(v)$ is defined to be 1. If v is an internal node and edges from $v_1; v_2$ are directed into v, then $\mathrm{u}(v); \mathrm{l}(v)$ are defined as follows:

$$\begin{array}{ll} \mathrm{u}(v) = \mathrm{u}(v_1)\mathrm{l}(v_2) + \mathrm{u}(v_2)\mathrm{l}(v_1) & \\ \mathrm{l}(v) \ = \mathrm{l}(v_1)\mathrm{l}(v_2) & \text{if } v \text{ is labeled with } +; - \\ & \\ \mathrm{u}(v) = \mathrm{u}(v_1)\mathrm{u}(v_2) & \\ \mathrm{l}(v) \ = \mathrm{l}(v_1)\mathrm{l}(v_2) & \text{if } v \text{ is labeled with } \\ & \\ \mathrm{u}(v) = \mathrm{u}(v_1)\mathrm{l}(v_2) & \\ \mathrm{l}(v) \ = \mathrm{u}(v_2)\mathrm{l}(v_1) & \text{if } v \text{ is labeled with } =: \end{array}$$

Finally, we define $\mathrm{u}(E)$ as the corresponding value of the output node of E. If E is division-free, then $|\mathrm{val}(E)| \quad \mathrm{u}(E)$. With these definitions we can state the main result of this paper.

Theorem 1. *Let E be a real radical expression. There is a probabilistic algorithm with one-sided error $1=2$ that decides whether $\mathrm{val}(E) = 0$. If the algorithm outputs $\mathrm{val}(E) \neq 0$, then the answer is correct. The running time of the algorithm is polynomial in $\mathrm{size}(E) + \log \mathrm{u}(E)$.*

We did not state the running time explicitly, because the running time depends on the way specific values for the input radicals are represented. As will be seen later, the running time of the algorithm is usually dominated by the running time of an algorithm approximating E with absolute error $2^{-d \log \mathrm{u}(E) e}$.

By running the algorithm e times with independent random bits, the error probability can be reduced to $= 2^{-e}$. We will see later that there is a better way to achieve this error probability, if e is small.

3 The Algorithm for Division-Free Expressions

In this section we will describe a probabilistic algorithm that decides whether a division-free radical expression is zero. We will also analyze the running time of the algorithm. In the following section we will analyze the error probability of the algorithm.

Before we describe the algorithm recall that a *d-th root of unity*, $d \; 2 \; \mathbf{N}$, is a solution of $X^d - 1 = 0$. The d-th roots of unity are given by $\exp(ik \; =d); k = 0; : : : ; d - 1$. Therefore, a random d-th root of unity corresponds to a random number between 0 and $d - 1$.

Algorithm Zero-Test

`Input:` A real division-free expression E with input radicals $\sqrt[d_1]{n_1}; : : : ; \sqrt[d_k]{n_k}$.

Output: "zero" or "non-zero"

Step 1: Compute $m_1, \ldots, m_l \in \mathbb{Z}$, such that $\gcd(m_i, m_j) = 1$ for all $i \neq j$, $i, j = 1, \ldots, l$, and such that each n_i can be written as $n_i = \prod_{j=1}^{l} m_j^{e_{ij}}$, $e_{ij} \in \mathbb{N}$. Compute this representation for each n_i.

Step 2: For all (i, j), $i = 1, \ldots, k$, $j = 1, \ldots, l$, compute the minimal $d_{ij} \in \mathbb{N}$ such that $\sqrt[d_i]{m_j}^{d_{ij}} \in \mathbb{Z}$. For $j = 1, \ldots, l$, compute $t_j = \operatorname{lcm}(d_{1j}, \ldots, d_{kj})$.

Step 3: Compute $d = \operatorname{lcm}(d_1, \ldots, d_k)$ and choose l d-th roots of unity $\zeta_1, \ldots, \zeta_l$ uniformly and independently at random.

Step 4: In E replace the radical $\sqrt[d_i]{n_i}$ by

$$\prod_{j=1}^{l} \zeta_j^{d^2 e_{ij}/t_j d_i} \sqrt[d_i]{m_j}^{e_{ij}} = \left(\prod_{j=1}^{l} \zeta_j^{d^2 e_{ij}/t_j d_i} \right) \sqrt[d_i]{n_i}.$$

Call this new radical expression $\overline{E}$.

Step 5: Compute $u(E)$ and approximate $\operatorname{val}(\overline{E})$ with absolute error less than $= 2^{-d \log(u(E)) e - 1}$. If in absolute value this approximation is smaller than output "zero", otherwise output "non-zero".

In the remainder of this section we will analyze the running time of this algorithm. For **Step 1** we can use a well-known procedure called factor-refinement. At any time during its execution factor-refinement maintains a list of integers m_j such that each n_i can be written as a product of the m_j's. Initially the list contains the n_i's. If there are two list elements m_s, m_t that are not relatively prime, factor-refinement computes $d = \gcd(m_s, m_t)$, replaces m_s and m_t by m_s/d and m_t/d, respectively, and adds d to its list. It is clear that eventually the list will contain integers that are relatively prime. An amortized analysis of factor-refinement was given by Bach et al. [1].

Lemma 1. *Let $n_1, \ldots, n_k$ be integers, $n = \prod_{i=1}^{k} n_i$. In time $O(\log^2(n))$ integers $m_1, \ldots, m_l$ can be computed such that*

(i) $\gcd(m_i, m_j) = 1$ *for all* $i \neq j$, $1 \leq i, j \leq l$,
(ii) Each n_i *can be written as* $n_i = \prod_{j=1}^{l} m_j^{e_{ij}}$, $e_{ij} \in \mathbb{N}$.

Within the same time bound the factorizations $n_i = \prod_{j=1}^{l} m_j^{e_{ij}}$, $e_{ij} \in \mathbb{N}$, *can be computed.*

Observe that l, the number of m_j's, can not be bounded by a function depending only on k, the number of input radicals. However, l is bounded by $\sum_{i=1}^{k} \log(n_i) \leq \operatorname{size}(E)$.

In **Step 2** we are asked to compute for each $\sqrt[d_i]{m_j}$ the smallest d_{ij} such that $\sqrt[d_i]{m_j}^{d_{ij}} \in \mathbb{Z}$. For fixed i and j this can done in time polynomial in d_i and $\log m_j$ as follows. For $e = 1, \ldots, d_i - 1$, first approximate $\sqrt[d_i]{m_j}^{e}$ with absolute error less than $1/2$ to obtain the unique integer m with $|\sqrt[d_i]{m_j}^{e} - m| < 1/2$. Then check whether $m^{d_i} = m_j^e$.

`Step 3` and `Step 4` can clearly be done in time polynomial in size(E). To analyze `Step 5` observe that although we change the input radicals, the corresponding input radicals in E and $\overline{E}$ have the same absolute value. Therefore $u(E) = u(\overline{E})$. By definition of $u(\overline{E})$, this is an upper bound for val(v) of each internal node v of the expression $\overline{E}$. A straightforward error analysis shows that approximating the input radicals of $\overline{E}$ with absolute error less than 2^{-w}, where

$$w = 3\text{size}(E) + 2d\log(|u(E)|)e + 1;$$

leads to an approximation of val($\overline{E}$) with absolute error less than $2^{-d\log(u(E))e-1}$, as required in `Step 5` of Algorithm Zero-Test. We assume that the radicals $\sqrt[n_i]{p_i}$ are represented in a way that allows for efficient approximation algorithms. The input radicals in $\overline{E}$ differ from the input radicals in E by powers of roots of unity. It follows from Brent's approximation algorithms for exp; log; and the trigonometric functions (see [5]) that these powers of roots of unity can be efficiently approximated. Hence, the input radicals of $\overline{E}$ can be approximated with absolute error $2^{-w}; w = 3\text{size}(E) + 2d\log(|u(E)|)e + 1;$ in time polynomial in size(E) and $\log(|u(E)|)$. As mentioned, this implies that val($\overline{E}$) can be approximated with absolute error $2^{-d\log(u(E))e-1}$ in time polynomial in $\log(|u(E)|)$ and size(E). Summarizing, we have shown

Lemma 2. *On input E the running time of Algorithm Zero-Test is polynomial in* size(E) *and* $\log(|\text{val}(E)|)$.

4 The Error Analysis

In this section we will analyze the error probability of Algorithm Zero-Test. We recall some basic facts and definitions from algebraic number theory. For readers not familiar with algebra and algebraic number theory we recommend [11]. A number $\alpha \in \mathbb{C}$ is called *algebraic* if α is the root of some polynomial $f(X) \in \mathbb{Q}[X]$. A polynomial $f(X) = \sum_{i=0}^{n} f_i X^i \in \mathbb{Q}[X]$ is called monic, if $f_n = 1$. An algebraic number $\alpha \in \mathbb{C}$ is called an *algebraic integer*, if it is the root of a monic polynomial with coefficients in $\mathbb{Z}$. The *minimal polynomial* of an algebraic number $\alpha \in \mathbb{C}$ is the smallest degree monic polynomial in $\mathbb{Q}[X]$ with root α. If $f(X)$ is the minimal polynomial of α, then the roots $\alpha_0 = \alpha; \alpha_1; \ldots; \alpha_{n-1}$ of f are called the *conjugates* of α. Product and sum of algebraic integers are algebraic integers. Product, sum, and quotients of algebraic numbers are algebraic numbers. Since arbitrary roots of integers are algebraic integers, we see that the value of an arbitrary radical expression is an algebraic number and that the value of a division-free algebraic expression is an algebraic integer.

The error analysis for Algorithm Zero-Test is based on the following two lemmas. The first one was originally formulated and used by Chen and Kao in [8].

Lemma 3. *Let α be an algebraic integer with conjugates $\alpha_0 = \alpha; \alpha_1; \ldots; \alpha_{n-1}$. Assume that $|\alpha_i| \neq 2^B; B \in \mathbb{N}$. For $b \in \mathbb{N}$, with probability at most $B/(b+B)$ a random conjugate α_i of α satisfies $|\alpha - \alpha_i| \leq 2^{-b}$.*

Lemma 4. *Let E be a real division-free radical expression with input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_k]{n_k}$ and let $\overline{E}$ be constructed as in Algorithm Zero-Test. Then* $\mathrm{val}(\overline{E})$ *is a random conjugate of* $\mathrm{val}(E)$ *chosen according to the uniform distribution.*

Both lemmas will be proven below. Let us show that they imply

Corollary 1. *Let E be a real division-free radical expression with input radicals $\sqrt[d_1]{n_1}, \ldots, \sqrt[d_k]{n_k}$. If* $\mathrm{val}(E) = 0$*, then on input E Algorithm Zero-Test will output "zero". If* $\mathrm{val}(E) \neq 0$*, then Algorithm Zero-Test will output "non-zero" with probability at least* $1/2$*.*

Proof: By Lemma 4 Algorithm Zero-Test generates an expression $\overline{E}$ whose value is a random conjugate of $\mathrm{val}(E)$. We already noted that $\mathrm{u}(E) = \mathrm{u}(\overline{E})$. In particular, the conjugates of $\mathrm{val}(E)$ are bounded in absolute value by $\mathrm{u}(E)$.

If $\mathrm{val}(E) = 0$, then the only conjugate of $\mathrm{val}(E)$ is 0 itself. Hence the approximation in `Step 5` will result in a number bounded in absolute value by $2^{-d\log(\mathrm{u}(E))e-1}$. Therefore, the answer of Algorithm Zero-Test will be "zero".

If $\mathrm{val}(E)$ is non-zero, then the approximation to $\mathrm{val}(\overline{E})$ is bounded in absolute value by $2^{-d\log(\mathrm{u}(E))e-1}$ if and only if $\mathrm{val}(\overline{E}) \geq 2^{-d\log(\mathrm{u}(E))e}$. Applying Lemma 3 to $\mathrm{val}(E)$ with $B = b = d\log(\mathrm{u}(E))e$ proves that this happens with probability at most $1/2$. □

Together with Lemma 2, Corollary 1 proves Theorem 1 for division-free real radical expressions.

We mentioned earlier that if the required error probability $\varepsilon = 2^{-e}$ is not too small, in practice we can do better than run Algorithm Zero-Test e times with independent random bits. We now want to make this statement more precise.

Set $b = d1/e - 1$. Assume that in `Step 5` of Algorithm Zero-Test instead of approximating $\mathrm{val}(\overline{E})$ with absolute error $2^{-d\mathrm{u}(E)e-1}$ we approximate it with absolute error $2^{-bd\mathrm{u}(E)e-1}$. Furthermore, we output "zero" if and only if the approximation is in absolute value less than $2^{-bd\mathrm{u}(E)e-1}$. With these parameters, a non-zero $\mathrm{val}(E)$ will be declared 0 by Algorithm Zero-Test if and only if $|\mathrm{val}(\overline{E})| \geq 2^{-bd\mathrm{u}(E)e}$. By Lemma 3 this happens with probability less than ε.

The running time of this algorithm will be polynomial in $1/\varepsilon$ rather than $\log 1/\varepsilon$. But for small ε it should be more practical than running $\log 1/\varepsilon$ times Algorithm Zero-Test with error probability $1/2$. Moreover, with this approach we save on the number of random bits (see [8] for a more detailed discussion).

In the remainder of this section we prove Lemma 3 and Lemma 4.

Proof of Lemma 3: Let d be the number of conjugates that are at most 2^{-b} in absolute value. $\prod_{i=0}^{n} |\alpha_i|$ is the absolute value of the constant term of the minimal polynomial of α. Hence the product is at least 1. Together with the upper bound 2^B on $|\alpha_i|$ we obtain

$$1 \leq \left|\prod_{i=0}^{n} \alpha_i\right| \leq 2^{(n-d)B} 2^{-db}.$$

This implies $d=n \quad B=(b+B)$. □

To prove Lemma 4 we need a few more definitions and facts from algebraic number theory. Again we refer to [11] for readers unfamiliar with algebra and algebraic number theory. For an algebraic number field F an isomorphism of F into a subfield of $\mathbb{C}$ whose restriction to $\mathbb{Q}$ is the identity, is called an *embedding* of F. The basic fact about embeddings is the following lemma.

Lemma 5. *Let F be an algebraic number field and let γ be an algebraic number whose minimal polynomial f over F has degree n. Every embedding of F can be extended in exactly n different ways to an embedding of $F(\gamma)$. An extension is uniquely determined by the image of γ, which must be one of the n distinct roots of (f).*

From this lemma one can deduce

Corollary 2. *Let be an algebraic number and let F be an algebraic number field containing . If is an embedding of F chosen uniformly at random from the set of embeddings of F, then ( ) is a conjugate of chosen uniformly at random from the set of conjugates of .*

We specialize these facts to radical expressions. If E is a radical expression with input radicals $\sqrt[d_1]{n_1},\ldots,\sqrt[q_k]{n_k}$, then val($E$) is contained in $\mathbb{Q}(\sqrt[d_1]{n_1},\ldots,\sqrt[q_k]{n_k})$, that is, the smallest field containing $\sqrt[d_1]{n_1},\ldots,\sqrt[q_k]{n_k}$. It does not seem easy to directly generate a random embedding of this field. However, we also have val(E) $\subseteq \mathbb{Q}(\sqrt[q]{m_1},\ldots,\sqrt[q]{m_l})$; where $d=\mathrm{lcm}(d_1,\ldots,d_k)$ and the integers m_j are as constructed in `Step 1` of Algorithm Zero-Test. For this extension we have

Lemma 6. *Let $m_1,\ldots,m_l$ be positive integers that are pairwise relatively prime. Assume that the radicals $\sqrt[q]{m_1},\ldots,\sqrt[q]{m_l}$ are real numbers. Let t_i be the smallest positive integer such that there is an integer c_i with $\sqrt[q]{m_i}=\sqrt[t_i]{c_i}$. Then t_i divides d and the minimal polynomial of $\sqrt[q]{m_i}$ over the field $\mathbb{Q}(\sqrt[q]{m_1},\ldots,\sqrt[q]{m_{i-1}})$ is given by $X^{t_i}-c_i$.*

Proof: It was shown by Siegel [14] that the minimal polynomial of $\sqrt[q]{m_i}$ over $\mathbb{Q}(\sqrt[q]{m_1},\ldots,\sqrt[q]{m_{i-1}})$ has the form

$$X^{t_i}-q_i\prod_{j=1}^{i-1}\sqrt[q]{m_j}^{e_{ij}};\ q_i\in\mathbb{Q};\ 0\le e_{ij}<t_j;\ t_i \text{ divides } d.$$

Hence

$$m_i^{t_i}=q_i^d\prod_{j=0}^{i-1}m_j^{e_{ij}}.$$

Write $q_i=c_i/p_i;\ c_i,p_i\in\mathbb{Z};\ \gcd(c_i,p_i)=1$. Since $m_i^{t_i}$ is an integer, p_i^d must divide $\prod_{j=0}^{i-1}m_j^{e_{ij}}$. Since the m_j's are pairwise relatively prime, $p_i^d=\prod_{j=0}^{i-1}m_j^{e_{ij}}$.

Hence $m_i^{t_i} = c_i^d$, and the minimal polynomial of $\sqrt[q]{m_i}$ over $\mathbb{Q}(\sqrt[q]{m_1}, \ldots, \sqrt[q]{m_{i-1}})$ is given by

$$X^{t_i} - c_i.$$

The lemma follows. □

For the embeddings of $\mathbb{Q}(\sqrt[q]{m_1}, \ldots, \sqrt[q]{m_{i-1}})$ this translates to

Corollary 3. *Let $m_i, \sqrt[q]{m_i}, t_i, c_i$ be as above. An embedding σ of the radical extension $\mathbb{Q}(\sqrt[q]{m_1}, \ldots, \sqrt[q]{m_l})$ is uniquely determined by l d-th roots of unity $\zeta_1, \ldots, \zeta_l$ such that $\sigma(\sqrt[q]{m_i}) = \zeta_i^{d/t_i} \sqrt[q]{m_i}$. Moreover, if the ζ_i are chosen uniformly and independently at random, then σ is chosen uniformly at random.*

Proof: By Lemma 5 and Lemma 6, an embedding σ of $\mathbb{Q}(\sqrt[q]{m_1}, \ldots, \sqrt[q]{m_l})$ is defined by mapping $\sqrt[q]{m_i}$ onto some root of $X^{t_i} - c_i; i = 1, \ldots, l$. These roots are given by $\zeta_i \sqrt[q]{c_i} = \zeta_i \sqrt[q]{m_i}$, where ζ_i is an arbitrary t_i-th root of unity. Since every t_i-th root of unity ζ_i can be written as ζ_i^{d/t_i} for a d-th root of unity ζ_i, the first part of the lemma follows.

To prove the second part, observe that for each t_i-th root of unity ζ_i there are exactly d/t_i d-th roots of unity ζ_i such that $\zeta_i^{d/t_i} = \zeta_i$. □

Proof of Lemma 4: From the previous corollary we know that by choosing d-th roots of unity $\zeta_j; j = 1, \ldots, l$, uniformly and independently at random, we choose a random embedding σ of $\mathbb{Q}(\sqrt[q]{m_1}, \ldots, \sqrt[q]{m_l})$. By choice of d, every radical $\sqrt[d_i]{m_j}$ is an element of this field. As before, t_j is defined as the smallest integer such that there is an integer c_j with $\sqrt[q]{m_j} = \zeta_j \sqrt[q]{c_j}$. Hence

$$\sqrt[d_i]{m_j} = \sqrt[q]{m_j}^{d/d_i} = \zeta_j \sqrt[q]{c_j}^{d/d_i}.$$

This implies

$$\sigma(\sqrt[d_i]{m_j}) = \sigma(\sqrt[q]{m_j})^{d/d_i} = \zeta_j^{d^2/d_i t_j} \sqrt[d_i]{m_j}.$$

We need to show that Algorithm Zero-Test correctly computes t_j. The algorithm computes t_j as $t_j = \mathrm{lcm}(d_{1j}, \ldots, d_{kj})$; where d_{ij} is the smallest integer such that $\sqrt[d_i]{m_j}^{d_{ij}} \in \mathbb{Z}$. Let $m_j = \prod_h p_h^{e_{hj}}$ be the prime factorization of m_j. Let e_j be the greatest common divisor of the exponents e_{hj}. Any positive integer t with $\sqrt[d_i]{m_j}^{t} \in \mathbb{Z}$ must satisfy

$$\frac{e_{hj}\, t}{d_i} \in \mathbb{N}, \text{ for all } h.$$

Hence t must be a multiple of $d_i / \gcd(e_{hj}, d_i)$ for all h. Therefore d_{ij} is the least common multiple of the integers $d_i / \gcd(e_{hj}, d_i)$. This least common multiple is given by

$$d_{ij} = d_i / \gcd(d_i, e_j) = \mathrm{lcm}(d_i, e_j)/e_j.$$

Similarly $t_j = \mathrm{lcm}(d; e_j)/e_j$: We obtain

$$\begin{aligned} e_j \quad & \mathrm{lcm}(d_{1j}; \ldots; d_{kj}) && = \mathrm{lcm}(e_j d_{1j}; \ldots; e_j d_{kj}) \\ = \mathrm{lcm}(\mathrm{lcm}(d_1; e_j); \ldots; & \mathrm{lcm}(d_k; e_j)) && = \mathrm{lcm}(\mathrm{lcm}(d_1; \ldots; d_k); e_j) = \mathrm{lcm}(d; e_j): \end{aligned}$$

Hence

$$\mathrm{lcm}(d_{1j}; \ldots; d_{kj}) = \mathrm{lcm}(d; e_j)/e_j = t_j:$$

So far, we have shown that $\prod_j^{d^2 = d_i t_j} \sqrt[d_i]{m_j};\ i = 1; \ldots; k;\ j = 1; \ldots; l;$ is the image of $\sqrt[d_i]{m_j}$ under a random embedding of $\mathbb{Q}(\sqrt[d]{m_1}; \ldots; \sqrt[d]{m_l})$ of . Then

$$(\sqrt[d_i]{n_i}) = \left(\prod_{j=1}^{\forall} \sqrt[d_i]{m_j}^{e_{ij}}\right) = \left(\prod_{j=1}^{\forall} \prod_j^{d^2 e_{ij} = t_j d_i} \sqrt[d_i]{n_i};\ i = 1; \ldots; k;\right)$$

as constructed in Algorithm Zero-Test, is the image of $\sqrt[d_i]{n_i}$ under the random embedding . Hence, $(\mathrm{val}(E)) = \mathrm{val}(\overline{E})$. Corollary 2 shows that $\mathrm{val}(\overline{E})$ is a random conjugate of $\mathrm{val}(E)$ chosen uniformly at random from the set of conjugates of $\mathrm{val}(E)$. □

5 Expressions with Divisions

If a radical expression E contains divisions, Lemma 3 is not applicable, since $\mathrm{val}(E)$ need not be an algebraic integer. However, if E contains divisions, then E can be transformed into an expression E^0, in which only the output node is labeled with $=$. This can be done by separately keeping track of the numerator and denominator of E and applying the usual arithmetic rules for adding, multiplying, and dividing quotients.

The size of E^0 is $O(\mathrm{size}(E))$ and $\mathrm{val}(E^0) = \mathrm{val}(E)$. Furthermore, if v is the output node of E^0, and if w is the node where the numerator of v is computed, then $\mathrm{val}(E) = 0$ if and only if $\mathrm{val}(w)$ of node w in E^0 is zero. Restricting E^0 to the subgraph induced by the edges lying on paths from the input nodes to w, we obtain a division-free expression D with $\mathrm{val}(D) = \mathrm{val}(w)$. To check whether $\mathrm{val}(E) = 0$, we can use Algorithm Zero-Test with input D. By definition of $u(E)$, we get $u(D)$ $u(E)$. Since E^0 and D can easily be constructed in time polynomial in $\mathrm{size}(E)$, the analysis for Algorithm Zero-Test given in the previous section proves Theorem 1 in the general case.

Acknowledgments I would like to thank Helmut Alt, Emo Welzl and Hans-Martin Will for stimulating and clarifying discussions.

References

1. E. Bach, J. Driscoll, J. O. Shallit, "Factor Refinement", *Journal of Algorithms*, Vol. 15, pp. 199-222, 1993.
2. J. Blömer, "Computing Sums of Radicals in Polynomial Time", *Proc. 32nd Symposium on Foundations of Computer Science* 1991, pp. 670-677.
3. J. Blömer, "Denesting Ramanujan's Nested Radicals", *Proc. 33nd Symposium on Foundations of Computer Science* 1992, pp. 447-456.
4. J. Blömer, "Denesting by Bounded Degree Radicals", *Proc. 5th European Symposium on Algorithms*, Lecture Notes in Computer Science, Vol. 1284, pp. 53-63, 1997.
5. R. P. Brent, "Fast Multiple-Precision Evaluation of Elementary Functions", *Journal of the ACM*, Vol. 23, pp. 242-251, 1976.
6. C. Burnickel, K. Mehlhorn, S. Schirra, "How to Compute the Voronoi Diagrams of Line Segments", *Proc. 2nd European Symposium on Algorithms*, Lecture Notes in Computer Science, Vol. 855, pp. 227-239, 1994.
7. C. Burnikel, R. Fleischer, K. Mehlhorn, S. Schirra, "A Strong and Easily Computable Separation Bound for Arithmetic Expressions Involving Radicals", *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997, pp. 702-709.
8. Z.-Z. Chen. M.-Y. Kao, "Reducing Randomness via Irrational Numbers", *Proc. 29th Symposium on Theory of Computing*, 1997, pp. 200-209.
9. G. Horng, M.-D. Huang, "Simplifying Nested Radicals and Solving Polynomials by Radicals in Minimum Depth", *Proc. 31st Symposium on Foundations of Computer Science* 1990, pp. 847-854.
10. S. Landau, "Simplification of Nested Radicals", *SIAM Journal on Computing* Vol. 21, No. 1, pp 85-110, 1992.
11. S. Lang, *Algebra*, 3rd edition, Addison-Wesley, 1993.
12. G. Liotta, F. P. Preparata, R. Tamassia, "Robust Proximity Queries in Implicit Voronoi Diagrams", *Technical Report RI 02912-1910*, Center for Geometric Computation, Department of Computer Science, Brown University, 1996.
13. M. Mignotte, "Identification of Algebraic Numbers", *Journal of Algorithms*, Vol. 3(3), 1982.
14. C. L. Siegel, "Algebraische Abhängigkeit von Wurzeln", *Acta Arithmetica*, Vol. 21, pp. 59-64, 1971.
15. C. K. Yap, "Towards Exact Geometric Computation", *Proc. Canadian Conference on Computational Geometry*, 1993, pp. 405-419.
16. C. K. Yap, T. Dubé "The Exact Computation Paradigm", in D. Z. Du, F. Hwang, editors, *Computing in Euclidean Geometry*, World Scientific Press, 1995.

Geometric Searching in Walkthrough Animations with Weak Spanners in Real Time[?]

Matthias Fischer, Tamás Lukovszki, and Martin Ziegler

Department of Computer Science and Heinz Nixdorf Institute, University of Paderborn
D-33102 Paderborn, Germany,
{mafi,talu,zieglerg}@uni-paderborn.de

Abstract. We study algorithmic aspects in the management of geometric scenes in interactive walkthrough animations. We consider arbitrarily large scenes consisting of unit size balls. For a smooth navigation in the scene we have to fulfill hard real time requirements. Therefore, we need algorithms whose running time is independent of the total number of objects in the scene and that use as small space as possible. In this work we focus on one of the basic operations in our walkthrough system: reporting the objects around the visitor within a certain distance.

Previously a randomized data structure was presented that supports reporting the balls around the visitor in an output sensitive time and allows insertion and deletion of objects nearly as fast as searching. These results were achieved by exploiting the fact that the visitor moves "slowly" through the scene. A serious disadvantage of the aforementioned data structure is a big space overhead and the use of randomization.

Our first result is a construction of weak spanners that leads to an improvement of the space requirement of the previously known data structures. Then we develop a deterministic data structure for the searching problem in which insertion of objects are allowed. Our incremental data structure supports $O(1 + k)$ reporting time, where k is a certain quantity close to the number of reported objects. The insertion time is similar to the reporting time and the space is linear to the total number of objects.

1 Introduction

A walkthrough animation is a simulation and visualization of a three-dimensional scene. The visitor of such a scene can see a part of the scene on the screen or a special output device. By changing the orientation of the camera she can walk to an arbitrary position of the scene. In todays computer systems, the scene is modeled by many triangles. The triangles are given by the three-dimensional position of their points. For every position of the visitor the computer has to compute a view of the scene. It has to eliminate hidden

[?] Partially supported by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), DFG Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik" Me872/4-1, and DFG Research Cluster "Efficient Algorithms for Discrete Problems and their Applications" Me872/7-1.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 163–174, 1998.
c Springer-Verlag Berlin Heidelberg 1998

triangles (hidden surface removal), to compute the color and brightness of the triangles (the objects resp.), and so on. This process is called rendering.

For a smooth animation we have hard *real time* requirements. The computer has to render at least 20 pictures (frames) per second. If the animation is computed with less than 20 frames per second, navigation in the scenes is hard or impossible. The time for the rendering of a picture depends on the complexity of the scene, i.e., the number of triangles and the number of pixels which are needed for drawing a triangle. Therefore the graphic workstation cannot guarantee the 20 frames per second for large geometric scenes. In order to control this situation real time and approximation algorithms are necessary to reduce the complexity of those parts of the scene, which are far away and thus have a low influence on the quality of the rendered image.

Our goal is to develop real time algorithms for managing large and dynamic geometric scenes. Our scene is *dynamic* in the sense that a visitor can insert and/or delete objects. We are interested in theoretical and experimental aspects of the problem. The basis for our considerations is an abstract modeling of the problem introduced by Fischer et al. [11]. One of the most important problems to be solved in the walkthrough system is the *search problem*: In order to guarantee the real time behavior of our algorithms it is important that the time for the search, insertion, and deletion of objects is independent of the scene size.

This work is focused on the search problem. It is motivated by the fact that the visitor can only see a relatively small piece of the scene. Only the objects appearing from the position of the visitor in an angle of at least a fixed constant α are potentially visible. Our goal is to develop data structures that support the selection of the objects potentially visible for the visitor. We give efficient solutions to the following problems.

The static searching problem: We assume that our scene consists of n unit size balls. In this case the position of the visitor q and the angle α define a circle in the plane or an d-dimensional sphere in the d-dimensional Euclidean space E^d. All objects in this sphere are potentially visible and all objects outside are not.

Representing the objects by points of E^d we have to solve a *circular range searching problem*, in which we are given a set S of n points in E^d. For a query $query(q; r)$ we have to report the points of S in the interior of the sphere with center q and radius r. We want to develop data structures with $O(n)$ space requirement which support the reporting of these points efficiently. Our goal is that, after a certain point location for q, the reporting takes time not much larger than the output size. In addition we want to use as little space as possible.

The searching problem with moving visitor: We assume that the visitor moves slowly through the scene. We say that the visitor moves *slowly* if the quotient $\frac{\delta}{x}$ is a constant, where δ is the maximum distance between two consecutive query positions, and x is the distance between a *closest pair* of S. We utilize the slow motion of the visitor in order to obtain $O(1)$ time for the point location for the query positions.

The dynamic searching problem: In the dynamic case the visitor can insert or delete an object at her current position. The problem is called *fully dynamic* if insertion and deletion are allowed; *incremental* if only insertion; and *decremental* if only deletion is allowed. Our goal is a linear space data structure with the same query time as in the static case and update time similar to the query time.

1.1 Related problems, state of the art

The objects potentially visible from the visitor are lying in the interior of a sphere whose center is the visitor's position q and radius r is the distance from which an object is in an angle exactly α visible, i.e., $r = \frac{w}{2\sin(\alpha/2)}$.

In the circular range searching problem we are given a set S of n points in E^d. A query has the form $query(q;r)$, and we have to report the points of S which are lying in the sphere with the center q and radius r. Representing the objects of the scene by points in E^d, we can report the potentially visible objects by solving a circular range searching problem. Furthermore, the results about *nearest neighbor queries* and *graph spanners* are of particular interest.

Data structures for circular range searching in the plane: For an overview of different kinds of range searching problems we refer to the survey article of Agarwal and Erickson [1]. Time optimal solutions of the 2-dimensional circular range searching problem use *higher order Voronoi diagrams*. Bentley and Maurer [4] presented a technique which extracts the points of the Voronoi cell containing q in the kth order Voronoi diagram of S for $k = 2^0; 2^1; 2^2; :::$ consecutively. It stops, if a point in the kth order Voronoi cell of q is found whose distance from q is greater than r, or all the n points are extracted. An $O(\log n \log\log n + t)$ query time is obtained, where t is the number of the points of S lying in the query disc. The space requirement of the data structures is $O(n^3)$. Chazelle et al.: [8] improved the query time to $O(\log n + t)$ and the space requirement to $O(n(\log n \log\log n)^2)$ by the aid of the algorithmic concept *filtering search*. Aggarwal et al.: [2] reduced the space requirement with *compacting* techniques to $O(n \log n)$. This method results an optimal query time for the circular range searching problem, but it has a superlinear space requirement already in the two dimensional case.

Nearest neighbor queries: Finding the *nearest neighbor* of a query point q in a set S of n points is one of the oldest problems in Computation Geometry. As we will see, the solution of this and related problems are very important to answer the range queries in our walkthrough problem. The following results show that it is not possible to get a query time independent of n without any restrictions on this problem.

Finding the *nearest neighbor* of a query point q in S has an $\Omega(\log n)$ lower bound in the so called *algebraic computation tree* model (c.f.: [13]). The planar version of this problem has been solved optimally with $O(\log n)$ query time and $O(n)$ space. But in higher dimensions no data structure of size $O(n \log^{O(1)} n)$ is known that answers queries in polylogarithmic time. The *approximate nearest neighbor* problem, i.e., finding a point $p \in S$ to the query point q such that $dist(q;p) \leq (1+\varepsilon) dist(q;q')$, where $q' \in S$ is the exact nearest neighbor of q, was solved optimally by Arya et al.: [3]. They give a data structure in dimension $d \geq 2$ of size $O(n)$ that answers queries in $O(\log n)$ time and can be built in $O(n \log n)$ time. (The constant factor in the query time depends on ε.) For proximity problems on point sets in $\mathbb{R}^d$ a comprehensive overview is given by Smid [16].

Spanners: Spanners were introduced to computational geometry by Chew [10]. Let $f > 1$ be any real constant. Let the *weight* of an edge $(p;q)$ be the Euclidean distance between p and q and let the *weight* of a path be the sum of the weights of its edges. A graph with vertex set $S \subseteq E^d$ is called a spanner for S with *stretch factor* f, or an f-spanner for S, if for every pair $p;q \in S$ there is a path in the graph between p and q

of weight at most f times the Euclidean distance between p and q. We are interested in spanners with $O(n)$ edges. By aid of an f-spanner of S we can answer a circular range query $query(q;r)$ by finding a "near" neighbor and performing a *breadth first search (BFS)* on the edges of the spanner. The BFS procedure visits all the edges having at least one endpoint not farer from q than fr. (The problem how we can find an appropriate near neighbor will be discussed later.)

Chen et al.: [9] proved that the problem of constructing any f-spanner for $f > 1$ has an $\Omega(n \log n)$ lower bound in the algebraic computation tree model. The first optimal $O(n \log n)$ time algorithm for constructing an f-spanner on a set of n points in E^d for any constant $f > 1$, were done by Vaidya [17] and Salowe [15]. Their algorithms use a hierarchical subdivision of E^d. Callahan and Kosaraju [7] gave a similar algorithm that constructs an f-spanner based on a special hierarchical subdivision and showed that the edges can be directed such that the spanner has bounded out-degree.

Weak spanners: Fischer et al.: [11] presented a fully dynamic data structure for real time management of large geometric scenes. They use a property of a certain dense graph weaker than the spanner property to answer circular range queries performing a point location and a BFS. We call this property *weak spanner* property (defined below).

The basic data structure in [11] for the searching problem is the graph $G_\gamma(S)$ called the γ-angle graph for S. (The same construction was called Θ-*graph* by Keil and Gutwin [12] and Ruppert and Seidel [14].) Fischer et al.: [11] proved that for $\gamma \leq \frac{\pi}{3}$ the graph $G_\gamma(S)$ is a weak spanner. They applied $G_\gamma(S)$ and perfect hashing to provide a randomized solution for the "moving visitor searching problem" and the dynamic searching problem.

1.2 New results

Consider a point set $S \subseteq E^d$ of n points. A (directed) graph G with vertex set S is a *weak spanner* for S with stretch factor f, if, for any two points $p;q \in S$, there is a (directed) path P from p to q in G such that, for each point $x \in P$, the Euclidean distance $dist(p;x)$ between p and x is at most $f \cdot dist(p;q)$. (Note that each weak spanner is strongly connected.)

Our first contribution is a new, improved variant of the weak spanner for $S \subseteq E^2$ constructed in [11]. We construct a graph $G_{\pi=2}(S)$ for S whose outdegree is bounded by 4. (The weak spanner used in [11] has outdegree 6.) On the other side, our weak spanner has stretch factor $3 + \sqrt{5} \approx 2{:}288$, whereas the one from [11] is 2. Nevertheless, we argue that our weak spanner yields a faster static data structure for our search problem: If the points of S are distributed uniformly in a square range of E^2, the graph $G_{\pi=2}(S)$ is not only a more space efficient data structure for circular range queries than some graph $G_\gamma(S)$ for $\gamma \leq \frac{\pi}{3}$ but it yields also a better expected query time: If we have to report the points of S in a disc with radius r, then we must traverse the directed edges having the origin in a concentric disc with radius fr, where f is the stretch factor of the graphs. The query time is determined by the number of traversed edges. The number of traversed edges is the number of points in the disc multiplied by the outdegree of the points. The expected number of points in the disc is quadratic in the radius. For example, the graph $G_{\pi=3}(S)$ has a stretch factor 2 and outdegree 6. $G_{\pi=2}(S)$ has a stretch factor $3 + \sqrt{5}$

and outdegree 4. Therefore, the expected query time in $G_{\pi=2}(S)$ is $\frac{3+\sqrt{5}}{6}$ 0:87 times the expected query time in $G_{\pi=3}(S)$.

Our second contribution is the development of a *deterministic* dynamic data structure for the "moving visitor searching problem", that guarantees running times for $query(q;r)$ and $insert(p)$ independent of n. The data structure from [11] yields such results only using randomization (perfect hashing). Our result is based on a careful, deterministic choice of $O(n)$ Steiner points. They replace the randomized approach from [11], that uses a grid as Steiner points and applies perfect hashing for compacting it.

The paper is organized as follows. In Section 2 we present a variant of the γ-angle graph [11], for a point set $S \subset E^2$ of n points, which outdegree bounded by four. We prove that this graph is a weak spanner for S with stretch factor $\sqrt{3+\sqrt{5}}$, and we give a plane sweep algorithm computing it in $O(n \log n)$ time.

In Section 3 we place $O(n)$ carefully chosen Steiner points into the scene supporting the point location for the visitor in $O(1)$ time. Here we assume that the visitor moves slowly, i.e., the distance between two consecutive positions of the visitor is at most a constant times the distance of the closest pair of S.

Finally, in Section 4, we show how we can insert a new object into the scene (a new point into S resp.) at the visitor's position q in $O(1+k)$ time, where k is the number of points (original points plus Steiner points) in the disc with center q and radius $r \cdot \sqrt{3+\sqrt{5}}$.

2 The Two Dimensional $\frac{\pi}{2}$-Angle Graph

In this section we construct a weak spanner $G_{\pi=2}(S)$ for the set $S \subset E^2$ of n points. The graph $G_{\pi=2}(S)$ contains at most $4n$ directed edges. For the definition and the construction of $G_{\pi=2}(S)$ we use a slightly more complicated distance measure than Fischer et al.[11]. This distance measure leads to some trickier plane sweep algorithm to obtain an $O(n \log n)$ construction time.

In the second part of the subsection we show that the graph $G_{\pi=2}(S)$ is a weak spanner with stretch factor $\sqrt{3+\sqrt{5}}$ 2:288 for the point set S. The proof uses different arguments than the proofs for the γ-angle graph for $\gamma = \frac{\pi}{3}$.

2.1 Construction

We define the graph $G_{\pi=2}(S)$ as follows: Rotate the positive x-axis over angles $i\frac{\pi}{2} - \frac{\pi}{4}$ for $0 \le i < 4$. This gives 4 rays $h_0;\ldots;h_3$. Let $c_0;\ldots;c_3$ be the cones that are bounded by two successive rays. The ith ray belongs to cone c_i and the $(i+1)$st ray does not. For $0 \le i < 4$, let l_i be the ray that emanates from the origin and halves the angle in the cone c_i, i.e. l_i is coincides with the positive/negative x-/y-axis respectively. For each point $p \in S$ translate the cones $c_0;\ldots;c_3$ and the corresponding rays $l_0;\ldots;l_3$ such that its apexes and starting points are at p. Denote by $c_0^p;\ldots;c_3^p$ and $l_0^p;\ldots;l_3^p$ these translated cones and rays, respectively. Now we build $G_{\pi=2}(S)$ such that for each $p \in S$ and $0 \le i < 4$, if the cone c_i^p contains a point of S then add a directed edge pq from p to some closest point q in c_i^p w.r.t. the distance measure $dist_i : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} \times \mathbb{R}$ defined as follows:

$dist_i(p;q) = (|x(p) - x(q)|;|y(p) - y(q)|)$ if $i = 0;2$, i.e: $q \in c_0^p \cap c_2^p$ and $dist_i(p;q) = (|y(p) - y(q)|;|x(p) - x(q)|)$ if $i = 1;3$. The total order under the pairs is defined by the lexicographical order.

Now we describe how the graph $G_{\pi/2}(S)$ can be efficiently built. Our construction consists of four phases. In the ith phase, $i = 0;\ldots;3$, we perform a plane sweep in the direction $i\frac{\pi}{2}$ in order to compute the neighbor in the ith translated cone c_i^p for each $p \in S$. Our plane sweep is a modified version of the plane sweep of Ruppert and Seidel [14]. We describe the 0th phase, the other phases perform analogously.

The Algorithm: First we sort the points of S non decreasing w.r.t: the x-coordinate. The sweepline moves from left to right. We maintain the invariant that for each point q left to the sweepline, its neighbor p in c_0^q has been computed if p is also left to the sweep line.

We initialize a data structure $D = \emptyset$. D will contain the points left to the sweepline whose neighbor has not yet been computed. The points in D are sorted increasing w.r.t: the y-coordinate. To handle the distance function $dist_i$ correctly, when the sweepline reaches a point p we put all points with the same x-coordinate as p in a data structure A and we work up these points in the same main step:

1. Let A be the set of points with the same x-coordinate as p ordered increasing w.r.t: the y-coordinate and let $A' = A$.
2. While $A \neq \emptyset$ we do the following:
   (a) Let p be the point of A with minimum y-coordinate. Determine the set of points $B(p) = \{q \in D : p \in c_0^q\}$. Let $B(p)$ be ordered also increasing w.r.t: the y-coordinate. Set $p' = p$. (The variable p' contain at each time the point of A with highest y-coordinate, such that the points of A with lower y-coordinate than p' cannot be a neighbor of any point of D.)
   (b) For each $q \in B(p)$ (in increasing order w.r.t: the y-coordinate) determine the point $p \in A$ with $|y(q) - y(p)|$ minimal and join q with p by a directed edge. Then delete q from D and set $p' = p$.
   (c) Delete p from A. Then delete all points from A having a lower y-coordinate than p'.
3. Insert the points of A' into D.

Correctness: Now we prove by induction that the invariant is satisfied after the main step of the plane sweep. In the data structure D we maintain the points left to the sweepline l whose neighbor is not left to l. At the beginning of the algorithm no point of S is left to l, so $D = \emptyset$. In the main step we must determine the points of D whose neighbor is on the sweepline l, compute the neighbor for these points, and update D, i.e: delete from D the points whose neighbor has been found and insert the points lying on l into D. In A we maintain the points on l, for which it is not yet decided if it is a neighbor of a point of D. Therefore, at the beginning A must contain all points of S on l. This is satisfied after step 1. We show that in step 2, for each point $q \in D$ with $c_0^q \setminus A \neq \emptyset$ the neighbor is computed correctly. For the points $q \in D$ doesn't exist any point $q \in c_0^q$ left to l. On the other side, if a point $q \in D$ has a point $q \in A$ in c_0^q then q is contained in $B(p)$ in step 2.a for a $p \in A$, its neighbor computed correctly in step 2.b and then deleted from D. In step 2.c the points are deleted from A which are surely not a neighbor of a point of D. Therefore, after step 2 D contains the points left to l whose

neighbor is not left to l and not on l. If we want to move the sweepline right to the next point, D mustn't contain any point whose neighbor is already computed (these points are deleted in step 2.b), but D must contain the points on l, because their neighbor is not yet computed. The points on l are inserted into D in step 3. So we can move the sweepline, the invariant is satisfied.

Analysis: For D, A and $B(p)$ we need data structures which support checking emptiness; insertion, deletion of a point; finding the next and the previous element w.r.t. y-coordinate for a given element of the data structure; and finding for a real number the point with nearest y-coordinate in the data structure (query). Using balanced search trees we obtain an $O(\log n)$ time for insertion, deletion, query and for finding the next and previous element, and $O(1)$ time for checking emptiness. If we link the nodes of the search tree in a doubly linked sorted list then we can find the next and the previous element in $O(1)$ time, too. For a point $q \in D$, let $next(q)$ and $prev(q)$ denote the next and the previous element of q in D w.r.t. the y-coordinate. In step 2.a we can determine $B(p)$ in the following way: Determine the nearest point $q \in D$ to p w.r.t. the y-coordinate. If q belongs to c_2^p (the reflected cone) then insert q into $B(p)$. Let $q^0 = q$. While $next(q)$ belongs to c_2^p, insert $next(q)$ into $B(p)$ and set $q = next(q)$. Than Let $q = q^0$. While $prev(q)$ belongs to c_2^p insert $prev(q)$ into $B(p)$ and set $q = prev(q)$.

Note that each point $p \in S$ is inserted and deleted from D, A and $B(p)$ exactly once. For each point q the nearest point $p \in A$ is computed once, too. Therefore, we can summarize the above analysis in the following theorem:

Theorem 1. *The graph $G_{\pi=2}(S)$ can be computed in $O(n \log n)$ time and $O(n)$ space.* □

2.2 The weak spanner property

Theorem 2. *$G_{\pi=2}(S)$ is a weak spanner with stretch factor $\sqrt{3+\sqrt{5}}$.*

Proof. We prove that for each $s, t \in S$ there is a directed path P from s to t in $G_{\pi=2}(S)$, such that for each point $v \in P$ the Euclidean distance $dist(s, v)$ is at most $\sqrt{3+\sqrt{5}}\, dist(s, t)$.

Consider the following simple algorithm.

- Let $s_0 = s$ and $j = 0$. While $s_j \neq t$ do the following:
- Let $c_i^{s_j}$ be the cone of s_j containing t. Set s_{j+1} to the neighbor of s_j in $G_{\pi=2}(S)$ in this cone and set $j = j + 1$.

The above algorithm finds a path with the properties we wish. To prove this we first define a potential function Φ on the points of S and show that Φ decreases in 'almost' every step of the above algorithm. Let $\phi_1(s)$ be the Manhattan distance $dist_M(t, s)$ from t to s and let $\phi_2(s)$ be $|x(t) - x(s)|$ if $t \in c_0^s \cup c_2^s$ and $|y(t) - y(s)|$ if $t \in c_1^s \cup c_3^s$. We define $\Phi(s)$ to be the pair $(\phi_1(s), \phi_2(s))$. The total order on the values of Φ is the lexicographical order of the pairs.

We show that the points of the path created by the above algorithm have the following property:

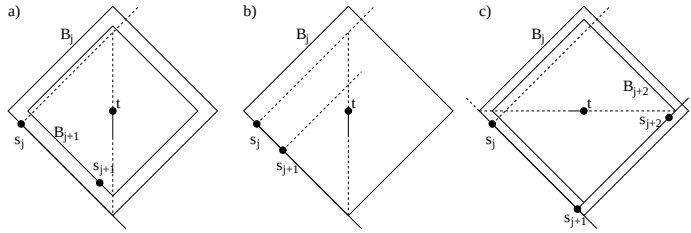


Fig. 1. The potential Φ during the construction of P.

1. either $\Phi(s_j) > \Phi(s_{j+1})$,
2. or $\phi_1(s_j) = \phi_1(s_{j+1})$, $\phi_2(s_j) \quad \phi_2(s_{j+1})$ and $\phi_1(s_j) > \phi_1(s_{j+2})$.

This property implies that the algorithm terminates and finds t. Note, if $s_{j+1} = t$ then $\Phi(s_j) > \Phi(t) = (0;0)$. Let $c_i^{s_j}$ be the cone at s_j that contains t. By the definition of $G_{\pi=2(S)}$ either there is a directed edge $< s_j; t >$ in $G_{\pi=2}(S)$ or an other point $s_{j+1} \in c_i^{s_j}$ and a directed edge $< s_j; s_{j+1} >$ with $dist_i(s_j; s_{j+1}) \quad dist_i(s_j; t)$. Let $B_j = \{x \in \mathbb{R}^2 : dist_M(t; x) \quad dist_M(t; s_j)\}$. The point s_{j+1} is clearly contained in B_j (Figure 1) and so $\phi_1(s_j) \quad \phi_{j+1}$. We distinguish three cases.

Case 1: $dist_M(s_j; t) > dist_M(s_{j+1}; t)$. Then $\phi_1(s_j) > \phi_1(s_{j+1})$ (Figure 1.a). Therefore, property 1 holds.

Case 2: $dist_M(s_j; t) = dist_M(s_{j+1}; t)$ and $t \in c_i^{s_{j+1}}$. Then $\phi_1(s_j) = \phi_1(s_{j+1})$ and $\phi_2(s_j) > \phi_2(s_{j+1})$ (Figure 1.b). Therefore, property 1 holds.

Case 3: $dist_M(s_j; t) = dist_M(s_{j+1}; t)$ and $t \in c_{i+1 \bmod 4}^{s_{j+1}}$. Then it can be happen, that $\phi_1(s_j) = \phi_1(s_{j+1})$ and $\phi_2(s_j) \quad \phi_2(s_{j+1})$. But in this case the next point s_{j+2} cannot be on the boundary of B_j (Figure 1.c). Therefore, property 2 holds.

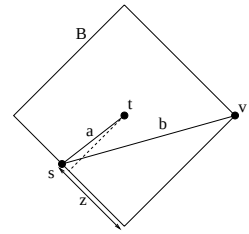


Fig. 2. Definitions for the computing the weak spanner factor.

Now we prove that for each $v \in P$ the Euclidean distance $dist(s; v)$ is at most $\sqrt{3 + \sqrt{5}}\, dist(s; t)$. We have seen, that $B = B_0$ contains each point s_j of the path P. Let $a = dist(s; t)$ and $b = dist(s; v)$. We have to maximize $\frac{b}{a}$ such that t is the center of B, s is on the boundary of B and x is contained in B. For a fixed point $p \in B$ one of the corners u of B satisfy $dist(p; u) = max(dist(p; x) : x \in B)$. So we can fix v at a corner of B and. Consider Figure 2. By aid of the theorem of Pythagoras and differentiation we obtain, that $\frac{b}{a}$ is maximal, when $z = (\sqrt{5} - 1)l$, where l is the side length of B, and this maximum is $\sqrt{3 + \sqrt{5}}$. □

Remark 1. Note that without the second component of $dist_i$, $0 \quad i < 4$ the obtained graph would be not necessary strongly connected. Consider the following example: let $S = \{p_1; p_2; p_3; p_4; p_5\}$ and let $(1;0)$, $(0;1)$, $(-1;0)$, $(0;-1)$ and $(0;0)$ be the coordinates of the points. Then the point in the center is not necessary reachable from the other points by a directed path. If we only have three cones (i.e: $\gamma = \frac{2\pi}{3}$) the construction of a non strongly connected example is quite easy.

3 Steiner Points for Navigation in the Scene

In this subsection we present a deterministic method to find the nearest neighbor of the query point q in c_i^q w.r.t: $dist_i$, $i = 0;::;3$. The BFS procedure in a range query $query(q;r)$ starts with these points. In order to find a nearest neighbor we exploit that the visitor moves through the scene slowly. Furthermore, we extend the original point set S with $O(n)$ carefully placed Steiner points. In the extended point set S' we can take advantage of the fact that the nearest neighbors of the query position q is close to the nearest neighbors of the previous query position q_{prev} and we can find it in constant time.

3.1 The "moving visitor" structure

First we describe how we place the Steiner points. We proceed similarly to the *mesh generation* technique of Bern et al: [5][6]: First we produce a linear size, balanced quadtree and we take the corners of the boxes of this quadtree as Steiner points. We describe this technique briefly below.

Definitions [5]: A *quadtree* is a recursive subdivision of the plane into square boxes. The nodes of the quadtree are the boxes. Each box is either a *leaf* of the tree or is *split* into four equal-area children. A box has four possible *neighbors* in the four cardinal directions; a neighbor is a box of the same size sharing a side. A *corner* of a box is one of the four vertices of its square. The corners of the quadtree are the points that are corners of its boxes. A side of a box is *split* if either of the neighboring boxes sharing it is split. A quadtree is called *balanced* if each side of an unsplit box has at most one corner in its interior. An *extended neighbor* of a box is another box of the same size sharing a side or a corner with it.

Building balanced quadtree for S [5]: We start with a root box b concentric with and twice as large as the smallest bounding square of S. We recursively split b as long as b has a point of S in its interior and one of the following conditions holds: (i) b has at least two points or (ii) b has side length l and contains a single point $p \in S$ with nearest neighbor in S closer than $2\sqrt{2}l$ or (iii) one of the extended neighbors of b is split. Then we balance the quadtree. We remark that the balancing increases the space requirement of the quadtree only by a constant factor. After all splitting is done, every leaf box containing a point of S is surrounded by eight empty leaf boxes of the same size.

Building linear size balanced quadtree [5]: The only nonlinear behavior of the above algorithm occurs, when a nonempty box is split without separating points of S. If this happens, we need to "shortcut" the quadtree construction to produce small boxes

around a "dense" cluster without passing through many intermediate size of boxes. We construct the quadtree for the cluster recursively and we treat the cluster as an individual point. In this way we obtain a linear size quadtree for S, i.e: the number of the boxes is linear to n. The linear size quadtree for S can be constructed in $O(n \log n)$ time and $O(n)$ space. For some algorithmic details we refer to [6].

Here ends the part borrowed from Bern et al: [5].

We call a dense cluster with the containing box and the eight extended neighbor boxes an *extended cluster*. We remark that in the extended clusters the balance property is maintained by the above description. It will be crucial for the fast navigation that an extended cluster has only constant number of corners on the boundary.

Building $G_{\pi=2}$ from the quadtree: We extend S with the $O(n)$ corners of the linear size quadtree and construct the graph $G_{\pi=2}(S^0)$ for the extended point set S^0. If we have the quadtree, we can determine for each point of S^0 its four neighbors in constant time. It follows from the fact that the neighbors of a point $p \in S^0$ in $G_{\pi=2}(S^0)$ are in the interior of the leaf box $b(p)$ containing p or in the interior of a neighboring box of $b(p)$.

Point location in constant time beyond slow motion of the visitor: If we have $G_{\pi=2}(S^0)$, we can determine for each point p^0 of S^0 the leaf box $b(p^0)$ of the quadtree containing p^0 in constant time. Furthermore, if x is the distance between the closest pair of S then the side length of the smallest box of the quadtree is a constant fraction of x. These observations imply that for the query position q the box $b(q)$ of the quadtree containing q can be computed in constant time via $G_{\pi=2}(S^0)$, if we know the box $b(q_{prev})$ containing the previous query position q_{prev}, since the line segment $(q_{prev}; q)$ intersects only constant many leaf boxes of the quadtree. The intersections and so the box $b(q)$ can be computed in constant time. Then the nearest neighbor of q in S^0 in each cone c_i^q, $i = 0; :::; 3$ can be determined in constant time, too. Knowing these nearest neighbors we can start the BFS in order to answer the range query $query(q; r)$. We conclude:

Theorem 3. *Let S be a scene of n equal size objects. Assume that the visitor moves slowly. There is a data structure which provides to report the potentially visible objects in time $O(1 + k)$ time, where k is the number of the edges of $G_{\pi=2}(S^0)$ having the origin in the interior of a circle which is concentric with the query circle and has a radius $\frac{p}{3 + \sqrt{5}}$ times of the query circle. The space requirement of the data structure is $O(n)$ and can be built in $O(n \log n)$ time.* □

4 The Incremental Data Structure, Lazy Updates

In this subsection we study the incremental version of the search problem. Here insertion of a new object at the current position of the visitor is allowed. We show how we can insert a point into the graph $G_{\pi=2}(S^0)$ in a time which match to the query time.

If we insert a new point into $G_{\pi=2}(S^0)$ we first must maintain the balance property in the quadtree. Then we must update the adjacencies at the affected vertices of $G_{\pi=2}(S^0)$ corresponding to corners of the changed boxes or they neighboring boxes. Updating $G_{\pi=2}(S^0)$ at the affected vertices costs only a constant time per box even in the case of a changed short cut. Therefore the update time is determined by the number of affected boxes. The main problem is to maintain the balance property. Let $l(b)$ be denote the

level of box b, it is the length of the tree path from the root box of the quadtree to the box b. If a leaf box b is splitted then the balance condition can be violated in each level higher than $l(b)$. Therefore, the worst case update time for the rebalancing of the quadtree depends on the depth of the quadtree and so on the total scene.

Let C_r be query disc having a radius r and C_{rf} the disc concentric with C_r and having a radius $rf = r(3+\sqrt{5})$. In order to made the rebalancing independent from the total size of the scene we make *lazy updates*, i.e: we split only the boxes intersecting C_{rf}. After the splittings we update only the vertices of $G_{\pi=2}(S^0)$ in C_{rf}. The remainder splittings will be performed later, when the current C_{rf} intersects the according boxes. So, at the beginning of each query we must check, whether the disc C_{rf} affects new leaf boxes violating the balance property. In this way the update time is at most linear to the number of vertices of the updated graph $G_{\pi=2}(S^0)$ in the disc C_{rf}. This time matches to the time of the searching. We summarize:

Theorem 4. *Let S be a scene of n equal size objects. Assume that the visitor moves slowly. There is a incremental data structure which provides inserting a new object at the position of the visitor in time $O(1+k)$, where k is the number of the edges of $G_{\pi=2}(S^0)$ having the origin in the interior of a circle which is concentric with the query circle and has a radius $(3+\sqrt{5})$ times of the query circle.* □

5 Conclusions

We considered special range searching problems motivated by a walkthrough simulation of a large geometric scene.

We constructed a directed graph $G_{\pi=2}(S)$ for a set $S \subset E^2$ of n points whose outdegree is bounded by 4. We gave a plane sweep algorithm to compute $G_{\pi=2}(S)$ in $O(n \log n)$ time. Then we proved that $G_{\pi=2}(S)$ is a weak spanner with stretch factor $(3+\sqrt{5}) \approx 2{:}288$. Our weak spanner decreases the space requirement of the data structures of Fischer et al: [11] drastically and yields a faster static data structure for the search problem in expected sense.

To solve the incremental "moving visitor searching problem" we presented a deterministic linear space data structure. Our data structure guarantees running times for $query(q;r)$ and $insert(p)$ independent of n. It is based on careful choice of $O(n)$ Steiner points.

The main directions of the future research are the following. We want to generalize the graph $G_{\pi=2}(S)$ in $d \geq 3$-dimensions, such that we obtain a weak spanner in E^d.

Further, we are interested in other linear size data structures to solve our searching problems. In particular, does a data structure exist solving the "moving visitor search problem" without Steiner points?

Acknowledgment: We would like to thank Artur Czumaj and Friedhelm Meyer auf der Heide for the helpfull comments and suggestions.

References

1. P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. Technical Report CS-1997-11, Duke Univ., Dep. of Comp. Sci., 1997. To appear: in Discrete and Computational Geometry: Ten Years Later.
2. A. Aggarwal, M. Hansen, and T. Leighton. Solving query-retrival problems by compact voronoi diagrams. In *22nd ACM Symposium on Theory of Computing (STOC'90)*, pages 331–340, 1990.
3. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 573–582, 1994.
4. J. L. Bentley and H. A. Maurer. A note on the euclidean near neighbor searching in the plane. *Information Processing Letters*, 8:133–136, 1979.
5. M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *J. Comp. Syst. Sci.*, 48:384–409, 1994.
6. M. Bern, D. Eppstein, and S. H. Teng. Parallel construction of quadtrees and quality triangulations. In *3rd Workshop on Algorithms and Data Structures (WADS'93)*, pages 188–199, 1993.
7. P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 291–300, 1993.
8. B. Chazelle, R. Cole, F. P. Preparata, and C. Yap. New upper bounds for neighbor searching. *Information and Control*, 68:105–124, 1986.
9. D. Z. Chen, G. Das, and M. Smid. Lower bounds for computing geometric spanners and approximate shortest paths. In *8th Canadian Conference in Computational Geometry (CCCG'96)*, pages 155–160, 1996.
10. L. P. Chew. There is a planar graph almost as good as the complete graph. In *2nd Annual ACM Symposium on Computational Geometry*, pages 169–177, 1986.
11. M. Fischer, F. Meyer auf der Heide, and W.-B. Strothmann. Dynamic data structures for realtime management of large geometric scenes. In *5th Annual European Symposium on Algorithms (ESA'97)*, pages 157–170, 1997.
12. J. M. Keil and C. A. Gutwin. Classes of graphs which approximate the complete euclidean graph. *Discrete and Computational Geometry*, 7:13–28, 1992.
13. F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction*. Springer Verlag, New York, 1985.
14. J. Ruppert and R. Seidel. Approximating the d-dimensional complete euclidean graph. In *3rd Canadian Conference in Computational Geometry (CCCG'91)*, pages 207–210, 1991.
15. J. S. Salowe. Constructing multidimensional spanner graphs. *International Journal of Computational Geometry and Applications*, 1:99–107, 1991.
16. M. Smid. *Closest-Point Problems in Computational Geometry*. To appear in: Handbook on Computational Geometry, edited by J.-R. Sack, North Holand, Amsterdam.
17. P. M. Vaidya. A sparse graph almost as good as the complete graph on points in k dimensions. *Discrete and Computational Geometry*, 6:369–381, 1991.

A Robust Region Approach to the Computation of Geometric Graphs[?]

(Extended Abstract)

Fabrizio d'Amore, Paolo G. Franciosa, and Giuseppe Liotta

Università di Roma "La Sapienza"
Dipartimento di Informatica e Sistemistica,
Via Salaria 113, I-00198 Roma, Italy
fdamore,pgf,liottaG@dis.uniroma1.it

Abstract. We present robust algorithms for computing the minimum spanning tree, the nearest neighbor graph and the relative neighborhood graph of a set of points in the plane, under the L_2 metric. Our algorithms are asymptotically optimal, and use only double precision arithmetic. As a side effect of our results, we solve a question left open by Katajainen [11] about the computation of relative neighborhood graphs.

1 Introduction

In the last two decades an impressive body of combinatorial results, data structures and algorithmic techniques have been developed in computational geometry. However, the expected technology transfer to application areas such as computer graphics, robotics, pattern recognition and GIS has been unsatisfactory. In particular, the assumption of real number arithmetic supporting infinite precision computations and the exclusive reliance on asymptotic analysis as a design principle can be identified as main obstacles to the adoption of the computational geometry viewpoint on the part of practitioners. Thus, the design of robust geometric algorithms, i.e. algorithms that are not affected by numerical round-off errors and that can handle degenerate configurations of the input has been motivating a great deal of research in recent years (see, [23] for a survey).

In this paper we investigate the problem of the robust computation of some well-known geometric graphs [16] of a set of points S $\mathbb{R}^2$ under the L_2 metric: *nearest neighbor graphs* (NNG(S)), *Euclidean minimum spanning trees* (EMST(S)), and *relative neighborhood graphs* (RNG(S)).

NNG, EMST and RNG are often referred to as members of the family of *proximity graphs*, i.e. graphs that attempt to capture different closeness relationships between points in a point set. Proximity graphs have applications in several areas of computer science, including pattern recognition, image processing, computational morphology, networking, computational biology and computational geometry. It is therefore crucial to design efficient algorithms that compute such graphs by properly handling degenerate configurations of the input (such as co-

[?] Research partially supported by the EC ESPRIT Long Term Research Project ALCOM-IT under contract 20244.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 175{186, 1998.
c Springer-Verlag Berlin Heidelberg 1998

circularities or collinearities) and that are not affected by errors in the flow of control due to round-off approximations.

The well-known relationship $\overline{\mathrm{NNG}}(S) \subseteq \mathrm{EMST}(S) \subseteq \mathrm{RNG}(S) \subseteq \mathrm{DT}(S)$ [16], where $\overline{\mathrm{NNG}}(S)$ denotes the undirected graph underlying $\mathrm{NNG}(S)$, is at the basis of well-known asymptotically optimal algorithms for computing the aforementioned geometric graphs for a set of points S: first $\mathrm{DT}(S)$ is computed in $O(|S| \log |S|)$ and then the appropriate subgraph is extracted in $O(|S| \log |S|)$ time (see, e.g., [13,16,18]). However, this approach presents two challenges when adopted in practice. The first one is that the design of an asymptotically optimal algorithm for Delaunay triangulations that correctly handles degeneracies is not an easy task (see, e.g., [8]). The second challenge concerns the size of the numbers that are involved in the computation. Any algorithm that computes $\mathrm{DT}(S)$ must be able to solve the so-called `in-circle` test (i.e. answering the question "is a given point inside the disk defined by other three points?"); to robustly handle this test on input data points represented as pairs of b-bits integers an algorithm must handle polynomials whose value may need $4b + O(1)$ bits to be represented.

However, the cost of facing the above two challenges sharply contrasts with the observation that it is trivial to design robust brute-force algorithms for computing $\mathrm{NNG}(S)$, $\mathrm{EMST}(S)$, and $\mathrm{RNG}(S)$, based on simple distance comparisons, thus requiring only double precision integer arithmetics. Although suboptimal with respect to asymptotic time complexity, such algorithms are very easy to implement, have small constant factors, and thus may be more convenient to adopt in practice on small input instances. Other algorithms have been proposed in the literature that compute $\mathrm{EMST}(S)$, and $\mathrm{RNG}(S)$ avoiding the computation of $DT(S)$ (see, e.g., [1,4,12,19,22]). However, such algorithms are either suboptimal or cannot be implemented with double precision arithmetic. Also, in some cases either degeneracies are not explicitly taken into account or the algorithm efficiency relies on the usage of complex data structures that are hard to implement. Robust algorithms for solving the all nearest neighbors problem are given in [9,17] and [20] using respectively 3-fold and double precision arithmetics. However, the solution in [20] uses rather complex data structures, and experimental results in [9] show that it is less effective in practice than the 3-fold precision solution. Furthermore, the approach in [20] cannot be easily generalized to the computation of other proximity graphs.

In this paper we propose a unified approach to the computation of $\mathrm{NNG}(S)$, $\mathrm{EMST}(S)$, and $\mathrm{RNG}(S)$, that gives rise to algorithms with optimal asymptotic performance, that are not affected by degenerate input configurations and that require evaluating polynomials whose value can be represented in at most $2b + O(1)$ bits. Furthermore, the algorithms are simple and make use of standard data structures. In order to design algorithms that guarantee the output correctness in spite of a limitation on the number of bits needed to represent intermediate computation results, we follow the *degree-driven* approach introduced in [14] and further refined in [2]. A geometric algorithm typically makes decisions by executing tests (predicates) that correspond to evaluating the sign

of a homogeneous multivariate polynomial on the input values with a constant number of terms. While approximations in producing the output data are admissible (and even desirable) as long as they do not exceed the resolution of the problem at hand, approximations in tests could give rise to topologically inconsistent output or to program crashes. According to the degree-driven approach, the numerical complexity of a test is measured in terms of the arithmetic degree of the corresponding polynomial, and the degree of an algorithm is the maximum degree of its predicates. A geometric problem has degree d if any algorithm that solves the problem has degree at least d.

Experimental results have shown that most of the CPU time of geometric algorithms is devoted to multiprecision numerical computations (see, e.g., [10,15]). Thus, optimizing the degree of an algorithm has to be considered a design goal as relevant as optimizing its asymptotic performances. The degree-driven approach to robust geometric computing has been applied to the problem of answering proximity queries in two and three dimensions in [14], to the problem of checking several geometric structures in [6] and to various segment intersection problems in [2].

The main contributions of this paper can be listed as follows.

- We use a *region approach* (see, e.g., [7,22]) to the robust computation of proximity graphs. More precisely, we introduce a variant of the *geographic neighbor graph* (first defined by Yao [22]) and investigate the relationship between such variant and relative neighborhood graphs, Euclidean minimum spanning trees and nearest neighbor graphs. We call the variant 4-*neighbors graph*.
- We propose an algorithm that computes the 4-neighbors graph of S in $(jSj \log jSj)$ time, (jSj) space and with optimal degree 2. The algorithm is simple and correctly handles degenerate input configurations.
- By exploiting the relationship between 4-neighbors graph and proximity graphs, and by using the optimal-degree algorithm mentioned above as a building block, we derive both asymptotically optimal and degree-optimal algorithms for the robust computation of the following geometric graphs: nearest neighbor graph, relative neighborhood graph, and Euclidean minimum spanning tree.

The above results also solve a question left open by Katajainen [11] about the existence of a subquadratic algorithm, based on the region approach, that computes the relative neighborhood graph of a set of points in the plane under the L_2 metric.

Our low-degree algorithm for the computation of the 4-neighbors graph is based on the partial computation of a Voronoi diagram defined on a suitable distance function. We use a sweep line technique that updates the sweep status only in correspondence of input data points. Intersections of bisectors in the Voronoi diagram are not explicitly computed, but their detection is used to deduce a change in the topology of the diagram between two consecutive sweep events.

Due to the lack of space, proofs are omitted in this extended abstract.

2 Degree of Geometric Algorithms and Problems

The notion of degree as a measure of the maximum number of bits that are needed to represent the values of the variables involved during an algorithm computation was introduced in [14]. An equivalent definition was given by Burnikel [3] who adopts the term *precision* instead of degree.

We borrow some terminology from [14]. Geometric algorithms are characterized on the basis of the complexity of their test computations. Any such computation consists of evaluating the sign of an algebraic expression over the input variables, constructed using an adequate set of operators, such has $f+; -; \ ; \ ; \frac{p}{2}-g$. This can be reduced to the evaluation of the sign of multivariate polynomials derived from the expression. A primitive variable is an input variable of the algorithm and has conventional arithmetic degree 1. The arithmetic degree of a (homogeneous) polynomial expression is the common arithmetic degree of its monomials. The arithmetic degree of a monomial is the sum of the arithmetic degrees of its variables. The *degree of an algorithm* is the maximum arithmetic degree of the irreducible factors of the multivariate polynomials that occur in its predicates and that change sign over their domain. The *degree of a problem* is the minimum degree of any algorithm that solves the problem. We say that an algorithm is (degree-)optimal if it has the same degree as the problem.

Boissonnat and Preparata [2] recently fine-tuned the notion of degree distinguishing between two arithmetic models. The *predicate arithmetic of degree* d is the arithmetic model where the only numerical operations that are allowed are the evaluations of predicates of degree at most d. The *exact arithmetic of degree* d assumes that also the value (and not just the sign) of a polynomial of degree d can be computed and exactly represented. However, rounding of higher degree operations, such as multiplications having a factor of degree d, are also allowed, with the caveat that the error introduced by the approximate results must not affect the correctness of the algorithm. Typical rounding can be to the closest d-fold precision integer or to the floor of the exact result. Although this second model is more demanding, it is also more realistic, since the floating point arithmetic available on almost any hardware platform meets the IEEE 754 standard. In the rest of this paper we evaluate the degree of our algorithms referring to the exact arithmetic of degree d model. The degree of an algorithm characterizes (up to a small additive constant) the arithmetic precision that is sufficient for a robust implementation. Namely, if the coordinates representing the input of a degree-d algorithm are b-bit integers, an algorithm of degree d that evaluates predicates with a constant number of terms is implementable with bit precision at most $d(b + O(1))$.

3 Octant Neighborhood

In this section we define the 4-*neighbors graph* and discuss some basic properties of the *octant Voronoi diagram*, which are Voronoi diagram based on a non-

standard distance function. In what follows, S denotes a set of n distinct points (*sites*) in the Euclidean plane.

3.1 4-neighbors graphs

Let p and q be two points in the plane. Point q is said to be h-*visible* from point p if either they coincide or the angle between the x-axis and vector $\vec{qp}$ belongs to interval $(\frac{(h-1)}{4}\pi; \frac{h}{4}\pi]$, $h = 1; \ldots; 8$. We introduce the distance $d_h(p;q)$ that coincides with the L_2 distance $d(p;q)$ if p is h-visible from q, and is $+\infty$ otherwise. Note that in general $d_h(p;q) \neq d_h(q;p)$, since at most one of them is finite.

The h-*octant neighbor graph* $\mathrm{ONG}_h(S)$ is a geometric graph on the vertex set S such that every site $p_i \in S$ $(i = 1; \ldots; n)$ is connected to its closest site p_j by a directed edge $(p_i; p_j)$ (break ties arbitrarily), where closeness is measured in terms of d_h-distance. The 4-*neighbors graph* of S is the directed graph $\mathrm{ONG}_{1-4}(S) = \bigcup_{i=1}^{4} \mathrm{ONG}_i(S)$. Clearly $\mathrm{ONG}_{1-4}(S)$ has $O(n)$ edges, although it is easy to construct examples where the 4-neighbors graph is not planar. The 4-neighbors graph of S is a subgraph of $\mathrm{ONG}(S) = \bigcup_{i=1}^{8} \mathrm{ONG}_i(S)$, which is often referred to as the *octant neighbor graph* of S (see, e.g., [11]). In what follows we denote by $\overline{G}$ the undirected graph underlying a directed graph G.

The relationship between octant neighbor graphs and minimum spanning trees has been studied by Yao [22], who proved that $\mathrm{EMST}(S) \subseteq \overline{\mathrm{ONG}}(S)$; other interesting and well-known relationships (see, e.g, [16]) are the following: $\overline{\mathrm{NNG}}(S) \subseteq \mathrm{EMST}(S) \subseteq \mathrm{RNG}(S) \subseteq \mathrm{DT}(S)$. In [21] it has been observed that $\mathrm{EMST}(S)$ is a subgraph of a proximity graph defined when the h-visibility is considered with respect to angles of size $\pi/3$ instead of $\pi/4$ and only three neighbors for each point of S are considered. The following theorem establishes the relationship between 4-neighbors graphs and the proximity graphs of interest in this paper.

Theorem 1. $\overline{NNG}(S) \subseteq EMST(S) \subseteq RNG(S) \subseteq \overline{ONG_{1-4}}(S) \subseteq \overline{ONG}(S)$:

3.2 Octant Voronoi diagrams

Our algorithm for computing 4-neighbors graphs exploits the properties of h-*octant Voronoi diagrams*. The h-octant Voronoi diagram of $S = \{p_1; p_2; \ldots; p_n\}$, denoted by $\mathrm{OVD}_h(S)$ $(h = 1; \ldots; 4)$, is a partition of the plane into regions such that there is a bijection between regions and sites, and such that the region associated with p_i contains all the points q of the plane for which $d_h(p_i; q) = \min_{1 \le j \le n}\{d_h(p_j; q)\}$. We denote by $\mathrm{reg}_h(p_i; S)$ the region associated with p_i in $\mathrm{OVD}_h(S)$.

In order to simplify the description of the properties of h-octant Voronoi diagrams, we concentrate on 4-octant Voronoi diagrams. Analogous properties can be easily derived for different values of h. When it does not give rise to ambiguities, we denote $\mathrm{OVD}_4(S)$ by $\mathrm{OVD}(S)$ and $\mathrm{reg}_4(p_i; S)$ by $\mathrm{reg}_S(p_i)$.

Property 1. The vertices of $\mathrm{OVD}(S)$ are a superset of S. They have at least two incident edges. Also, sites are left end-point of exactly two edges.

Property 2. Every edge e of OVD(S) is either a line segment or a half-line and its slope belongs to $[-\frac{1}{4}; 0]$. If the left endpoint of e is a site then the slope of e is either $-\frac{1}{4}$ or 0.

In Fig. 1, we note that adjacent regions in OVD(S) are separated by a polyline. Given a set of points S and two points $p; q \in S$, we define the *boundary* $bnd_S(p; q)$ *between* p *and* q *in OVD*(S) as the set of all edges e in OVD(S) such that $\mathrm{reg}_S(p)$ abuts on e from above and $\mathrm{reg}_S(q)$ abuts on e from below. According to this definition, we note that $\mathrm{reg}_S(q)$ and $\mathrm{reg}_S(r)$ in Fig. 1 are separated by two different boundaries, namely $\mathrm{bnd}_S(q; r)$ and $\mathrm{bnd}_S(r; q)$. Moreover, if the regions associated with two sites are not adjacent, as in the case of sites $p; r$ in Fig. 1, we have $\mathrm{bnd}_S(p; r) = ;$.

One can easily observe that the boundary between two adjacent regions consists of at most two consecutive edges, such that the first one (from left) is either horizontal or has slope $-\frac{1}{4}$ while the second one is the perpendicular bisector of p_i and p_j, denoted by $\mathrm{bis}_{p_i p_j}$.

Property 3. Let p and q be two sites of S. The boundary between $\mathrm{reg}_S(p)$ and $\mathrm{reg}_S(q)$ in OVD(S) is a subset of the boundary between $\mathrm{reg}_{fp;qg}(p)$ and $\mathrm{reg}_{fp;qg}(q)$. Namely, $\mathrm{bnd}_S(p; q)$ $\mathrm{bnd}_{fp;qg}(p; q)$.

In what follows we use the notation $\mathrm{bnd}(p; q)$ as a shorthand for $\mathrm{bnd}_{fp;qg}(p; q)$.

Property 4. For any $z \in \mathbb{R}$, let OVD($S_<(z)$) denote the set $fp_i \in S \mid x_i < zg$. We have that OVD(S) and OVD($S_<(z)$) coincide in the half-plane $x < z$.

Property 5. For any $S' \subseteq S$ and for each $p \in S'$, the region associated with p in OVD(S) is contained in the region associated with p in OVD(S'). Namely, $\mathrm{reg}_S(p) \subseteq \mathrm{reg}_{S'}(p)$.

Theorem 2. *For* $n > 1$*, the number of segments in OVD*(S) *is at most* $5n - 4$.

Theorem 3. *Let* S *be a set of points in the plane. Computing the nearest neighbor graph, the Euclidean minimum spanning tree, the relative neighborhood graph, and the octant neighbor graph of* S *has degree* 2*. Computing the Delaunay triangulation, the Voronoi diagram, and the octant Voronoi diagram of* S *has degree* 4*.*

4 Robust Computation of 4-Neighbors Graphs

The 4-neighbors graph of a set of points (sites) is obtained as the (undirected) union of $\mathrm{ONG}_i(S)$, $i = 1; : : : ; 4$. In what follows, we only refer to the computation of $\mathrm{ONG}_4(S)$, since any $\mathrm{ONG}_i(S)$, $i = 1; : : : ; 4$, can be computed by the same algorithm applied to a different set of sites, obtained from S after a suitable sequence of sign changes and coordinate swaps. We assume that each site is represented by a pair of b-bit integer coordinates.

Sites are ordered from left to right, and sites with the same abscissa can be arranged in any order. If all sites in S have the same abscissa, then $\mathrm{ONG}_4(S)$ is trivially computed. Otherwise, the plane can be partitioned into vertical *slabs* defined by $x_l < x \quad x_r$, where x_l and x_r are distinct abscissae of consecutive sites in S (i.e., no site has abscissa in $(x_l; x_r)$). Hence, each slab contains sites (at least one) only at abscissa $x = x_r$.

Our algorithm for computing $\mathrm{ONG}_4(S)$ is based on a sweep line technique, where the plane is swept from left to right. After a subset $S^0 \quad S$ of sites has been swept, suppose the next site to be encountered is $p_i = (x_i; y_i)$. The algorithm determines the region R of $\mathrm{OVD}(S)$ that contains p_i and defines a new edge of $\mathrm{ONG}_4(S)$. By Property 4, in order to determine the region R containing p_i, it is sufficient to know the boundaries of $\mathrm{OVD}(S^0)$ that are crossed by the sweep line at abscissa $x = x_i$. Such information is stored in the so-called *sweep status*, a data structure storing the list of boundaries of the octant Voronoi intersected by the sweep line. The updates to the sweep status at abscissa $x = x_i$ are computed by keeping track of the changes in the topology of the octant Voronoi diagram that occurred to the left of p_i, in the vertical slab H that contains p_i.

Clearly, if all the intersections between pairs of boundaries that occur in H were explicitly computed, the update would be immediate. However, this approach would imply the explicit computation of $\mathrm{OVD}(S^0)$, and thus it would require degree at least 4, as shown in Theorem 3. The main point of our optimal degree approach is to avoid the computation of the topology of $\mathrm{OVD}(S^0)$ and directly obtain the sweep status at $x = x_i$.

4.1 Sweep status, pockets and bundles

The sweep status is an interleaved sequence $(b_0; s_0; \ldots; b_i; s_i; b_{i+1}; \ldots; s_k; b_{k+1})$, where each s_i is the occurrence of a site in S whose region intersects the sweep line, and b_i is $\mathrm{bnd}(s_{i-1}; s_i)$; an example is shown in Fig. 1. The sweep status is intended ordered from top to bottom, i.e. b_0 is the topmost boundary crossed by the sweep line. We say that boundaries b_i and b_{i+1} are *adjacent* in the current sweep status. Also, we say that b_i is *above* b_{i+1} (and that b_{i+1} is *below* b_i).

A single site can occur more than once in the sweep status, if its region is not vertically monotone. See for example Fig. 1, where only the sequence of sites in the sweep status is shown. We also assume that there is a *dummy* site with suitable coordinates, such that it is 4-visible from all sites. Sites s_0 and s_k in the sweep status are always occurrences of the dummy site, while b_0 and b_k are respectively the horizontal and $-\frac{1}{4}$ sloped half-lines from the dummy site. In Fig. 1 the dummy site is named d.

A *pocket* is a triple $(b_i; s_i; b_{i+1})$ in the sweep status such that b_i and b_{i+1} intersect to the right of the sweep line. Such intersection point is called the *potential apex* of the pocket and is denoted by $\mathrm{pot\_apex}(b_i; s_i; b_{i+1})$. The *pocket region* is the subset of $\mathrm{reg}_S(s_i)$ that lies to the right of the sweep line, below b_i and above b_{i+1}. In what follows, we always consider a pocket associated with one of the vertical slabs defined by the input points. We say that a pocket $(b_i; s_i; b_{i+1})$ *belongs* to a slab $H = (x_i; x_k]$ if $x_i < \mathrm{pot\_apex}(b_i; s_i; b_{i+1}) \quad x_k$. Note that in

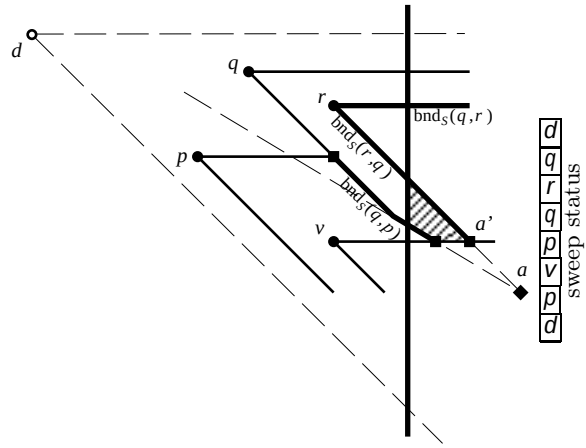


Fig. 1. An octant Voronoi diagram and a sweep line. Multiple occurrences of sites in the sweep status (only site occurrences are shown in the sweep status). Due to the intersection of boundary bnd($q;p$) with boundary bnd($p;v$), apex(bnd($r;q$)$;r;$bnd($q;p$)) = a' does not coincide with pot_apex(bnd($r;q$)$;r;$bnd($q;p$)) = a.

some cases the rightmost point of the pocket region (called *apex* and denoted apex($b_i;s_i;b_{i+1}$)) does not coincide with the potential apex of the corresponding pocket, and the slab containing the apex of a pocket may be different from the slab containing its potential apex. This situation is correctly handled by our algorithm. Fig. 1 depicts a pocket, its apex and its potential apex; the shaded area is the pocket region.

The proof of the following lemma is based on Property 5. In the statement, $x(\ )$ denotes the abscissa of a point.

Lemma 1. *For each pocket* $(b_i;s_i;b_{i+1})$ *we have that* $x(pot\_apex(b_i;s_i;b_{i+1}))$ $x(apex(b_i;s_i;b_{i+1}))$.

In our sweeping approach to the robust computation of 4-neighbors graphs, the exact $x(\ )$ values of the apices are never computed. Namely, as we are going to show in the next section, in order to correctly update the sweep status during the sweep, it suffices to know the slab to which the pockets in the sweep status belong. Hence, we can only rely on the position of potential apices. Before being ready for a detailed description of the algorithm, we need a last definition.

A *bundle* is a portion $(s_{i-1};b_i;s_i;b_{i+1};\ldots;b_m;s_m)$ of consecutive sites and boundaries in the sweep status such that all triples $(b_{j-1};s_j;b_j)$, $i < j < m$ are pockets, and they all belong to the same slab. In the simplest case, a bundle only consists of a single pocket. A bundle is *maximal* if it is not contained in any bundle. A bundle B *belongs* to a slab H if the pockets of B belong to H. A bundle can be seen as a "cloud" of intersections of boundaries in a same slab; it can be seen that to the right of such intersections all the involved boundaries

disappear, and a single new boundary appears in the sweep status. When a new slab H is spanned by the sweep line, first all maximal bundles that belong to H are processed, and then the sites in H are taken into account. While processing a bundle, we do not investigate the exact topology of OVD(S) within the slab, but we build in a single step the sweep status after all intersections in the bundle occur.

4.2 Algorithm `Robust-4-neighbors`

Algorithm `Robust-4-neighbors` (see next page) receives as input a set S of distinct points of the plane ordered by non-decreasing abscissa, and computes $\mathrm{ONG}_4(S)$. If two input points have the same abscissa, they belong to the same slab; such points are processed by the algorithm in any order.

During the plane-sweep, Algorithm `Robust-4-neighbors` maintains the following information:

- the sweep status;
- for each slab H such that H is to the right of the sweep line, a set of maximal bundles that belong to H.

At the beginning of the computation, the set of edges of $\mathrm{ONG}_4(S)$ is empty (line 1), the sweep status is set to $(b_0; s_0; b_1)$ (line 2), where s_0 is an occurrence of the dummy site and b_0 (b_1) is the 0 ($-\frac{}{4}$) sloped ray from s_0. All sets of maximal bundles associated with the slabs are initially empty (line 3).

Algorithm `Robust-4-neighbors` sweeps the plane from left to right. For each slab that is encountered, it updates the sweep status, updates the set of maximal bundles, and updates the set of edges of $\mathrm{ONG}_4(S)$. Suppose $H = (x_j; x_r]$ is the last visited slab and suppose $H^0 = (x_r; x_j]$ is the next slab to be visited. Before processing H^0, we can prove that the following invariants are preserved (the proof is omitted for lack of space).

I1: All maximal bundles in the current sweep status are correctly associated with the slab which they belong to.

I2: The current sweep status contains all the boundaries of OVD(S) that intersect the vertical line $x = x_r + \varepsilon$, for a suitably small positive ε.

I3: Let S^0 S be the set of sites whose abscissa is at most x_r. For any point p in S^0, the octant neighbor of p has been correctly computed.

Algorithm `Robust-4-neighbors` exploits two complementary procedures, named `INSERT_POCKET` and `DELETE_POCKET`. Both procedures receive as input a triple $(b_i; s; b_k)$ (where b_i and b_k are two boundaries and s is a site) and verify whether such triple is a pocket. If this is the case, the first procedure determines the slab which the pocket belongs to and updates the set of maximal bundles of that slab accordingly. Conversely, the second procedure removes the pocket from the maximal bundle that contains it, and updates the set of maximal bundles of the slab which the pocket belonged to before the removal.

`INSERT_POCKET` is invoked when a new pair of adjacent boundaries is detected in the sweep status. This happens when a new boundary is inserted in the

sweep status (lines 6, 13 and 19). Conversely, DELETE_POCKET is invoked when two boundaries are no longer adjacent in the sweep status because some new boundary has been inserted in-between (line 13) or one of the boundaries has been removed (line 19).

```
Algorithm Robust-4-neighbors(S)
input: a set of sites S
output: the 4-octant neighbor graph ONG4(S)
begin
1.    ONG4(S) := (S; ;)
2.    set the sweep status to (b0; d; b1)
3.    set to ; the set of maximal bundles associated with each slab
4.    for each slab, from the leftmost one to rightmost one, do
/* let (b0; s0; b1; s1; : : : ; sk; bk+1) be the current sweep status */
/* PROCESS BUNDLES: */
5.        while there is a bundle B = (si-1; bi; si; : : : ; bm; sm) in the current slab
6.            replace B by (si-1; bnd(si-1; sm); sm) in the sweep status
7.            INSERT_POCKET(bi-1; si-1; bnd(si-1; sm))
8.            INSERT_POCKET(bnd(si-1; sm); sm; bm+1)
/* PROCESS SITES: */
/* move the sweep line to the right-end of the current slab */
9.        for each site p in the current slab
10.           locate p in the sweep status
11.           if p lies between two boundaries bi and bi+1 then
12.               add edge (p; si) to ONG4(S)
13.               replace (bi; si; bi+1) in the sweep status with
                      (bi; si; bnd(si; p); p; bnd(p; si); si; bi+1)
14.               DELETE_POCKET(bi; si; bi+1)
15.               INSERT_POCKET(bi; si; bnd(si; p))
16.               INSERT_POCKET(bnd(p; si); si; bi+1)
17.           else (p lies on boundary bi)
18.               add edge (p; si) to ONG4(S)
19.               replace (bi-1; si-1; bi; si; bi+1) in the sweep status with
                      (bi-1; si-1; bnd(si-1; p); p; bnd(p; si); si; bi+1)
20.               DELETE_POCKET(bi-1; si-1; bi)
21.               DELETE_POCKET(bi; si; bi+1)
22.               INSERT_POCKET(bi-1; si-1; bnd(si-1; p))
23.               INSERT_POCKET(bnd(p; si); si; bi+1)
end
```

Lemma 2. *Algorithm* Robust-4-neighbors *correctly computes* $ONG_4(S)$.

By a careful analisys of the geometric primitives required by our algorithm, exploiting techniques similar to those presented in [2], and based on Theorems 3 and 2 we can prove the following:

Theorem 4. *Let* S *be a set of* n *distinct points in the plane. There exists an algorithm that computes the* 4*-neighbors graph of* S *with optimal time* $O(n \log n)$*, optimal space* $O(n)$ *and that has optimal degree* 2.

5 Robust Computation of Proximity Graphs

Algorithm `Robust-4-neighbors` can be used as a building block for devising robust algorithms that compute nearest neighbor graphs, Euclidean minimum spanning trees and relative neighborhood graphs in asymptotically optimal time and space and with optimal degree 2.

By Theorems 1 and 4 an algorithm that extracts $\mathrm{NNG}(S)$ from $\mathrm{ONG}_{1-4}(S)$ simply consists of selecting for each vertex v of $\mathrm{ONG}_{1-4}(S)$ the shortest edge incident on it.

Theorem 5. *Let S be a set of n distinct points in the plane. There exists an algorithm that computes the nearest neighbor graph of S in optimal time $O(n \log n)$, optimal space $O(n)$ and that has optimal degree 2.*

Combining Theorems 1 and 4 with an $O(n \log n)$ technique for extracting the minimum spanning tree from a weighted graph with n vertices and $O(n)$ edges (an $O(n \log \log n)$ time algorithm is presented in [5]) one can conclude the following.

Theorem 6. *Let S be a set of n distinct points in the plane. There exists an algorithm that computes the Euclidean minimum spanning tree of S in optimal time $O(n \log n)$, optimal space $O(n)$ and that has optimal degree 2.*

Supowit [18] has shown an algorithm that computes the relative neighborhood graph of a set S of n points in the plane in optimal $O(n \log n)$ time by deleting from the Delaunay triangulation of S the edges that are not in the relative neighborhood graph. A variant of Supowit's algorithm can be applied to each $\mathrm{ONG}_i(S)$, thus giving:

Theorem 7. *Let S be a set of n distinct points in the plane. There exists an algorithm that computes the relative neighborhood graph of S in optimal time $O(n \log n)$, optimal space $O(n)$ and that has optimal degree 2.*

The above theorem answers a question posed by Katajainen [11] about the existence of a subquadratic algorithm, based on the region approach, for computing relative neighborhood graphs in the plane under the L_2 metric.

Acknowledgments

We are grateful to Franco Preparata for helpful discussions.

References

1. P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.*, 6(5):407–422, 1991.
2. J. D. Boissonnat and F. Preparata. Robust plane sweep for intersecting segments. Technical Report 3270, INRIA, Sophia Antipolis, 1997.
3. C. Burnikel. *Exact Computation of Voronoi Diagrams and Line Segment Intersections*. Ph.D thesis, Universität des Saarlandes, Mar. 1996.

4. P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 291–300, 1993.
5. D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742, 1976.
6. O. Devillers, G. Liotta, R. Tamassia, and F. P. Preparata. Cecking the convexity of polytopes and the planarity of subdivisions. In *Algorithms and Data Structures (Proc. WADS 97)*, volume 1272 of *Lecture Notes Comput. Sci.*, pages 186–199. Springer-Verlag, 1997.
7. H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. 16th Annu. ACM Sympos. Theory Comput.*, pages 135–143, 1984.
8. L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.
9. K. Hinrichs, J. Nievergelt, and P. Schorn. An all-round sweep algorithm for 2-dimensional nearest-neighbor problems. *Acta Informatica*, 29:383–394, 1992.
10. M. Karasick, D. Lieber, and L. R. Nackman. Efficient Delaunay triangulations using rational arithmetic. *ACM Trans. Graph.*, 10:71–91, 1991.
11. J. Katajainen. The region approach for computing relative neighborhood graphs in the L_p metric. *Computing*, 40:147–161, 1987.
12. D. Krznaric, C. Levcopoulos, and B. J. Nilsson. Minimum spanning trees in d dimensions. In *Proc. of ESA 97*, number 1284 in Lecture Notes in Computer Science, pages 341–349. Springer, 1997.
13. A. Lingas. A linear-time construction of the relative neighborhood graph from the Delaunay triangulation. *Comput. Geom. Theory Appl.*, 4:199–208, 1994.
14. G. Liotta, F. P. Preparata, and R. Tamassia. Robust proximity queries: an illustration of degree-driven algorithm design. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 156–165, 1997.
15. E. P. Mücke. Detri 2.2: A robust implementation for 3D triangulations. Manuscript, available at URL: `http://www.geom.umn.edu:80/software/cglist/lowdvod.html`, 1996.
16. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.
17. P. Schorn. *Robust Algorithms in a Program Library for Geometric Computation*, volume 32 of *Informatik-Dissertationen ETH Zürich.* Verlag der Fachvereine, Zürich, 1991.
18. K. J. Supowit. The relative neighborhood graph with an application to minimum spanning trees. *J. ACM*, 30:428–448, 1983.
19. P. M. Vaidya. Minimum spanning trees in k-dimensional space. *SIAM J. Comput.*, 17:572–582, 1988.
20. P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.
21. Y. C. Wee, S. Chaiken, and D. E. Willard. General metrics and angle restricted voronoi diagrams. Technical Report 88-31, Comp. Sci. Dpt., Univ. at Albany, November 1988.
22. A. C. Yao. On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM J. Comput.*, 11:721–736, 1982.
23. C. K. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 35, pages 653–668. CRC Press LLC, Boca Raton, FL, 1997.

Positioning Guards at Fixed Height Above a Terrain { An Optimum Inapproximability Result?

Stephan Eidenbenz, Christoph Stamm, and Peter Widmayer

Institute for Theoretical Computer Science ETH, 8092 Zürich, Switzerland
<lastname>@inf.ethz.ch

Abstract. We study the problem of minimizing the number of guards positioned at a fixed height h such that each triangle on a given 2.5-dimensional triangulated terrain T is completely visible from at least one guard. We prove this problem to be NP-hard, and we show that it cannot be approximated by a polynomial time algorithm within a ratio of $(1 -)\frac{1}{35} \ln n$ for any > 0, unless $NP \quad TIME(n^{O(\log \log n)})$, where n is the number of triangles in the terrain. Since there exists an approximation algorithm that achieves an approximation ratio of $\ln n + 1$, our result is close to the optimum hardness result achievable for this problem.

1 Introduction and Problem De nition

We study the problem of positioning a minimum number of guards at a fixed height above a terrain. The *terrain* is given as a finite set of points in the plane, together with a triangulation (of its convex hull), and a height value is associated with each point (a triangulated irregular network (TIN), see e.g. [5]). The TIN defines a bivariate, continuous function; this surface in space is also called a 2.5-dimensional terrain. A *guard* is a point in space above the terrain. A guard can see a point of the terrain if the straight line segment between the guard and the point does not intersect the terrain. That is, a particular guard point can see some parts of the terrain, while others might be hidden. We ask for a smallest set of guards at a fixed height that together see the whole terrain. More precisely, we study a problem we call Terrain Cover (TC), where the input is a 2.5-dimensional terrain, given as a TIN, and a height h, and where the goal is to find a smallest set of guard points at height h such that every triangle can be seen from at least one guard. We assume h to be such that all points in the terrain are below h. Note that our requirement that each triangle be completely covered by one guard is a particular version of the problem, different from the version in which a triangle may also be covered by several guards together, with each guard covering only a part of the triangle. This problem models a question that arises after the liberalization of the telecommunications market

? This work is partially supported by the Swiss National Science Foundation

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 187–198, 1998.
c Springer-Verlag Berlin Heidelberg 1998

in Switzerland. Companies are planning to place communication stations above the Swiss mountains in extremely low position - balloons at a height of 20 km above sea level - and hold them in geo-stationary position. If we simply model electromagnetic wave propagation at high frequencies (GHz) by straight lines of sight, ignoring reflection and refraction, the Terrain Cover problem asks for a cover with the smallest number of balloons.

Related problems have been considered previously. Guarding a polygon has been studied in depth; for an overview, see the surveys [6], [7], [8] and [9], or any textbook on computational geometry. More specifically, [2] deals with optimum guarding of polygons and monotone chains. When guards can only be positioned directly on a given 1.5-dimensional terrain that must be completely covered, the problem of finding the minimum number of guards is NP-hard. A shortest watchtower for a given 2.5 dimensional terrain, i.e., a single guard position closest to the terrain (in vertical direction) that sees all of the terrain, can be found in $O(n \log n)$ time [10]. The related problem of finding the lowest watchtower, i.e., a single guard position with smallest height value that sees all of the terrain, can be solved in linear time using linear programming. Some upper and lower bounds on the number of guards needed to guard a terrain, when guards can only be positioned at the vertices of a 2.5-dimensional terrain, have been derived [1]. Our problem of guarding (covering) a terrain at a fixed height has not been studied in the literature so far. However, a previous result [6] implies that the Terrain Cover problem for a 1.5-dimensional terrain can be solved in linear time.

We proceed as follows in this paper. We first propose an approximation algorithm for the Terrain Cover problem for a 2.5-dimensional terrain that guarantees an approximation ratio of $\ln n + 1$, where n is the number of triangles in the TIN, and ln is the natural logarithm. We do so by showing (in Sect. 2) how a solution of the Set Cover problem can be used to solve Terrain Cover approximately. In Set Cover (SC), we are given a finite universe $E = fe_1; \quad ; e_n g$ of elements e_i and a collection of subsets $S = fs_1; \quad ; s_m g$ with $s_i \quad E$, and we need to find a subset $S^0 \quad S$ of minimum cardinality such that every element e_i belongs to at least one member in S^0. For ease of discussion, let E and S have an arbitrary, but fixed order.

Our proposed approximate solution brings up the question of whether this approximation is the best possible. It is the main contribution of this paper to show that indeed a better approximation is impossible, up to a constant factor, unless NP has $n^{O(\log \log n)}$-time deterministic algorithms. To this end, we propose a reduction from SC to TC (Sect. 3). Our reduction constructs a (planar) polygon with holes from a given instance of SC; in a second step a terrain is built by turning the inside of the polygon into a canyon, with steep walls on the polygon boundary and columns for the holes. Recall that a reduction from the problem SC to the problem TC is a pair $(f; g)$ of two functions such that for any instance I of SC, $f(I)$ is an instance of TC and such that for every feasible solution z of $f(I)$, $g(z)$ is a feasible solution of I. Furthermore, if z^0 is an optimum solution of $f(I)$, then $g(z^0)$ is an optimum solution of I. In addition,

both functions must be computable in time polynomial in the size of the SC-instance, i.e. polynomial in jIj. We show that the reduction has all the desired properties and can be computed efficiently (Sect. 4). In Sect. 5, we show how an inapproximability result for SET COVER by Feige [4] carries over with the proposed construction to TERRAIN COVER. Precisely, we prove that TERRAIN COVER cannot be approximated with ratio $((1 -)=35) \ln n$, for any > 0, by a polynomial time algorithm, unless $NP \quad TIME(n^{O(\log \log n)})$. Section 6 contains some concluding remarks and discusses implications for other problems.

2 An Approximation Algorithm

TC can be approximated with a ratio $\ln n + 1$, where n is the number of triangles, by constructing an SC-instance for a given TC-instance as follows: Each triangle is an element of the SC-instance. For each triangle determine the area on the plane $z = h$ from where the triangle is fully visible. This area is a polygon of descriptional complexity $O(n^2)$, that can be computed in time $O(n^4)$ by interpreting the points of the polygon as special points of an arrangement. At each point, where two of these polygons intersect, determine which triangles are visible from this point and define the set of visible triangles as one set for SC. There are $O(n^6)$ such intersections. Now solve the SC-instance approximately, by applying the well-known greedy algorithm for SC: add to the solution the set that covers a maximum number of elements not yet covered. This solution is not more than $\ln n + 1$ times larger than the optimum solution for SC. To see that this reduction is approximation-ratio-preserving, consider that the n polygons partition the plane $z = h$ into cells. Observe that the set of visible triangles is the same throughout the area of a cell. On the boundary of the cell, however, a few more triangles might be visible since the boundary may be part of the visibility area of another triangle. Therefore, any solution of the TC-instance can be transformed to a solution of the SC-instance by moving guards that are in the interior of a cell to an appropriate intersection point on the boundary of the cell.

3 Construction of the Reduction

In order to prove our inapproximability result for TERRAIN COVER (TC), we show how to construct an instance of TC for every instance of SET COVER (SC), i.e., we describe the function f of the reduction. The construction first leads to a polygon (with holes); we then construct a terrain by simply letting the area inside the polygon have height 0 and letting the area outside the polygon (including the holes) have height h', where h' is slightly less than h.

We construct the polygon in the $x-y$-plane; Figure 1 shows this construction. For the sequence of sets $s_1; \quad ; s_m$, place on the horizontal line $y = y_0$ the sequence of points $((i - 1)d'; y_0)$ from left to right for $i = 1; \quad ; m$, with d' a constant distance between two adjacent points. For ease of description, call the i-th point s_i. For each element $e_i \ 2 \ E$, place on the horizontal line $y = 0$ two

points $(D_i; 0)$ and $(D_i^0; 0)$, with $D_i^0 = D_i + d$ for a positive constant d. Arrange the points from left to right for $i = 1; \quad ; n$, with distances $d_i = D_{i+1} - D_i^0$ to be defined later. Call the points also D_i and D_i^0, for $i = 1; \quad ; n$.

For every element e_i, draw a line g through s_j and D_i, where s_j is the first set of which e_i is a member. Also draw a line g^0 through s_l and D_i^0, where s_l is the last set of which e_i is a member [1]. Let the intersection point of g and g^0 be I_i. Then draw line segments from every s_k that has e_i as a member to D_i and to D_i^0. Two lines connecting an element e_i with a set s_j form a cone-like feature; the area between these two lines will therefore be called a *cone*. Call the triangle $D_i I_i D_i^0$ a *spike*. We have only constructed one part of the polygon thus far: Among all the lines described, only the spikes and the line segments of the horizontal line $y = 0$ that are between two spikes are part of the polygon boundary, all other lines merely help in the construction. In our construction the guards of an optimum solution will have to be placed at the points s_j, therefore we need to make sure that a guard at s_j only sees the spikes of those elements e_i that are a member of the set s_j. This is achieved by introducing a barrier-line at $y = b$, see Fig. 1. Only line segments on the horizontal line $y = b$ that are outside the cones are part of the polygon. We draw another barrier-line with distance b^0 from the first barrier at $y = b + b^0$. Define holes of the polygon by connecting endpoints of line segments of the two barrier lines that belong to the same cone-defining line. We call the area between the two lines at $y = b$ and $y = b + b^0$ (including all holes) the *barrier*. Thus, the barrier contains a small part of all cones.

As a next step in the construction of the polygon, draw a vertical line segment at $x = -d^{00}$, where d^{00} is a positive constant, from $y = 0$ to $y = y_0$. This line segment is part of the polygon boundary except for the segment between the two barrier lines. Assume that the rightmost spike is farther right than the rightmost set, i.e. $D_n^0 > s_m$, and draw another vertical line segment from $y = 0$ to $y = y_0$ at $x = D_n^0 + d^{00}$, again taking a detour at the barrier. The boundary lines of the polygon defined so far are shown as solid lines in Fig. 1. The thickness b^0 of the barrier is defined such that all segments of all holes except for those on the line $y = b + b^0$ are visible from two guards at $P_1 = (-d^{00}; 0)$ and $P_2 = (D_n^0 + d^{00}; 0)$. To achieve this, the thickness b^0 is determined by intersecting (for each pair of adjacent holes) a line from P_1 through the lower right corner of the left hole (of the pair of adjacent holes) with a line from P_2 through the lower left corner of the right hole as shown in Fig. 2. Now, the barrier line $y = b + b^0$ is defined to go through the lowest of all these intersection points. (We will show in Sect. 4 that all intersection points actually lie on this line.)

In order to simplify our proof, we attach another feature, which is called an ear, to the corners P_1 and P_2, forcing one guard each to P_1 and P_2. Ears are shown in Fig. 2. Our construction aims at forcing guards for element spikes at points for sets, but there is a potential problem if a guard is placed in an area where two cones intersect: Such a guard may see the spikes of two elements that are not both a member of the same set. Therefore, we duplicate the whole

[1] We assume w. l. o. g. that each element is a member of at least two sets.

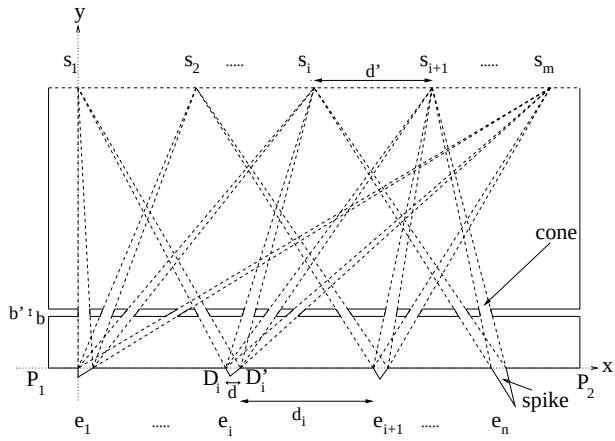


Fig. 1. Basic construction

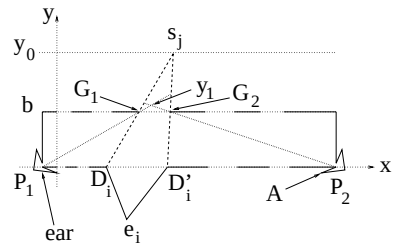


Fig. 2. Thickness of the barrier and ears

construction by flipping it over at the horizontal line $y = y_0$. The result is shown in Fig. 3. We denote the mirror image spike of e_i by e_i^0 and the mirror image points of $P_1; P_2$ by $P_1^0; P_2^0$. It is important to note that the cones, drawn as dashed lines in the figures, are not part of the polygon.

Given the polygon, the terrain is defined by placing the interior of the polygon at height $z = 0$ and the exterior at height $z = h^0$, with $h = h^0 +$ and a small positive constant, with vertical walls along the polygon boundary. The latter is for simplicity of the presentation only; the terrain can easily be modified to have steep, but not vertical, walls such that the terrain actually is a continuous function in two variables and such that our proofs still work. The resulting terrain is triangulated in such a way that the total number of triangles is polynomial in the input size, i.e., the size of the SC-instance, and such that each spike is triangulated as one triangle only. We set the parameters of the reduction as follows: d^0 and y_0 are arbitrary positive constants; d and b are positive constants as well, where $d = \frac{d^0}{2}$ and $b = \frac{5}{12}y_0$. We let $b^0 = \frac{\frac{35}{12^2}y_0}{2\sum_{i=1}^{n} m^i + 2\frac{d^0}{d} - \frac{7}{12}}$ and $D_l = d + 2d\sum_{i=1}^{l} m^i$ for $l = 1; \quad ; n$. We will prove in Sect. 4 that the reduction is feasible and runs in polynomial time with these parameter values.

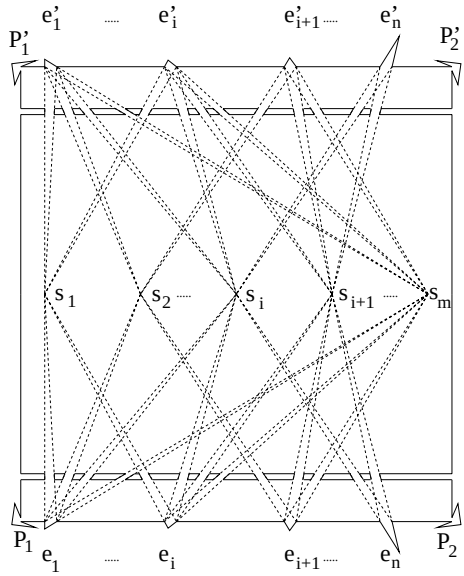


Fig. 3. Final construction

4 Properties of the Reduction

4.1 The Reduction is Feasible

In order to make the reduction work, we request that at no point a guard sees three or more spikes except if it is placed at some s_i. A guard that is placed at some point with y-value between 0 and b, i.e., between the barrier and the spikes, sees at most one spike, provided the barrier is placed such that no cones of two different elements intersect in the area below the barrier. A guard that is placed at some point with y-value between $b + b^0$ and y_0, but not equal to y_0, sees at most two spikes, provided that the spikes are placed such that no three cones intersect in the area above the barrier, and provided that the view of the guard is blocked by the barrier as described. A guard with y-value greater than y_0 does not see any of the spikes at $y = 0$ since the view is blocked by the barrier. A guard that is placed at some point with y-value less than 0, covers at most one spike, if it is ensured that no two spikes intersect. Thus, we need to prove the following:

- No three cones from different elements intersect.
- The barrier is such that all intersections of cones from the same element e_i are below b and such that all intersections of cones from different elements are above $b + b^0$ and such that all of the barrier except for the walls at $y = b + b^0$ is visible from at least one of two guards at P_1 and P_2.
- No two spikes intersect.

No Three Cones from Different Elements Intersect

Lemma 1. *For e_l 2 s_{l^0}, let:*

$$D_l \quad \max \quad \frac{s_{i^0} - s_{l^0}}{s_{i^0} - s_{j^0}}(D_j + d - D_i) + D_i + d$$

where the maximum is taken over all e_i 2 s_{i^0} and e_j 2 s_{j^0}, for which $i < j < l$ and $l^0 < j^0 < i^0$ holds. Then the three cones from e_l to s_{l^0}, from e_i to s_{i^0} and from e_j to s_{j^0}, with $i < j < l$ do not have a common intersection point.

Proof. Assume that the positions of the elements, i.e., the values D_v, have been set for all $v < l$ such that no three cones intersect. We show how to set D_l such that no three cones intersect; see Fig. 4. Let S be an intersection point with maximum y-value among any two cones of elements to the left of e_l. For each set s_{l^0} of which e_l is a member, draw a line through S, determine where it intersects the line $y = 0$, and let $D^S_{l;l^0}$ be the x-value of this intersection point. Let $D^S_l = \max_{l^0} D^S_{l;l^0}$ be the maximum x-value of all intersection points defined this way. For any pair of cones in "inverse position" to the left of e_l, with which a cone at e_l forms a "triple inversion", compute the corresponding D^S_l and let $D^{\max}_l$ be the maximum D^S_l. Finally, we let $D_l = D^{\max}_l + d$ to ensure that no three cones have one common intersection point at some point S. Figure 4 shows the situation for an intersection and explains the notation. The

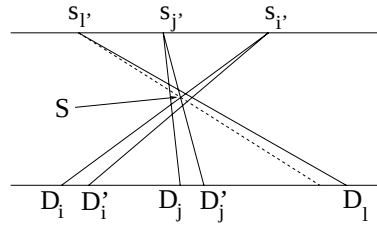


Fig. 4. Intersection of three cones

point S is the intersection point of the lines g_1 from s_{i^0} to D_i and g_2 from s_{j^0} to D^0_j. Simple geometric calculations yield: $S = ((1 - t_1)s_{i^0} + t_1 D_i; y_0(1 - t_1))$ with $t_1 = \frac{s_{i^0} - s_{j^0}}{D^0_j - D_i + s_{i^0} - s_{j^0}}$. Let g_3 be the line from s_{l^0} to S, and simple geometric calculations show: $D^S_{l;l^0} = \frac{s_{i^0} - s_{l^0}}{s_{i^0} - s_{j^0}}(D_j + d - D_i) + D_i$. The lemma follows. □

Lemma 1 implies:

$$\max\left(\frac{s_{i^0} - s_{l^0}}{s_{i^0} - s_{j^0}}(D_j + d - D_i) + D_i + d\right) \quad \max\left(\frac{md^0}{d^0}(D_j + d) + d\right) ; 8j < l$$

$$m(D_{l-1} + d) + d$$

Now, let $D_l = m(D_{l-1} + d) + d$. It is easy to see that this is consistent with our definition of D_l, since: $d + 2d \sum_{j=1}^{l} m^j = m((d + 2d \sum_{j=1}^{l-1} m^j) + d) + d$

The Barrier is in Good Position

Lemma 2. *Any two cones that belong to the same element e_i intersect only at points with y-values at most $y_0 \frac{d}{d+d'}$.*

Proof. Let e_i be a member of s_j and s_l, and let $s_j < s_l$. The intersection point of the lines g_j from s_j to D_i' and g_l from s_l to D_i is the point in the intersection area of the two cones that has the largest y-value. The lemma follows by simple geometric calculations. □

Lemma 3. *Any two cones that belong to elements e_i, e_j, respectively, with $i < j$, intersect only at points with y-values at least $y_0 \frac{d_i}{d_i + md'}$.*

Proof. Let $e_i \in s_{i'}$ and let $e_j \in s_{j'}$. Furthermore, let $D_i < D_j$ and $s_{j'} < s_{i'}$. This is the exact condition for the corresponding two cones to intersect. The intersection point of the lines g_1 from $s_{j'}$ to D_j and g_2 from $s_{i'}$ to D_i' is the point in the intersection area of the two cones with minimum y-value. The lemma follows by simple geometric calculations. □

Lemma 4. *Let $b' = \frac{bd(y_0 - b)}{y_0(p_2 - p_1) - d(y_0 - b)}$, where p_1 and p_2 are the x-values of the points P_1 and P_2. Then all of the barrier including the segments of the cones except for the segments at $y = b + b'$ are visible from the two guards at P_1 and P_2.*

Proof. Let $e_i \in s_j$ and let G_1 and G_2 be the two points where the corresponding cone intersects with the barrier line $y = b$ (see Fig. 2). We find an expression for y_1, which is the y-value of the intersection point of the two lines from P_1 to G_1 and from P_2 to G_2, and the lemma follows by simple geometric calculations. □

If we substitute $b = \frac{5}{12} y_0$ and $p_2 - p_1 = d + 2d \sum_{i=1}^{n} m^i + d'' - (-d'') = d + 2d \sum_{i=1}^{n} m^i + 2d''$ in the equation for b', we obtain $b' = \frac{\frac{35}{12^2} y_0}{2 \sum_{i=1}^{n} m^i + 2\frac{d''}{d} - \frac{7}{12}}$. A simple calculation shows that $b' < \frac{y_0}{12}$, if $m \geq 2$ and $n \geq 2$, which must be the case since there were no intersections otherwise.

Because of $d = \frac{d'}{2}$ and because of Lemma 2, any two cones from the same element intersect only at points with y-value at most $\frac{1}{3} y_0$, which is less than b. Because of $d_i \geq md'$ for all d_i and because of Lemma 3, any two cones from different elements intersect only at points with y-value at least $\frac{1}{2} y_0$, which is at most $b + b'$.

Spikes of Two Elements Do Not Intersect

Lemma 5. *The spikes of any two elements do not intersect.*

Proof. We determine the x-value x_l of the point I_l in the spike of e_l. Note that $x_l > D_l$. Simple calculations show that $x_l \geq 2D_l$. Since $D_{l+1} = m(D_l + d) + d$ and since we can assume that $m \geq 2$, the lemma follows. □

4.2 The Reduction Preserves Optimality

In this section we will show how our reduction maps solutions of the TC-instance $f(I)$ to solutions of the SC-instance I. We prove that optimum solutions are mapped to optimum solutions by showing each of both directions in a lemma.

Lemma 6. *If there exists a feasible solution of the SC-instance I with k sets, then there exists a feasible solution of the TC-instance $f(I)$ with $k+4$ guards.*

Proof. For each set s_j in the solution of the SC-instance, place a guard at height h at point s_j, and place four additional guards at height h at the points $P_1; P_2; P_1^0; P_2^0$. ⊓⊔

Lemma 7. *If there exists a feasible solution of the TC-instance $f(I)$ with $k+4$ guards, then there exists a feasible solution of the SC-instance I with k sets.*

Proof. We describe the function g that maps a solution for TC to a solution for SC. Given a solution of the TC-instance $f(I)$, proceed as follows: Move the guard that covers point A (at height 0) of the ear at P_1 (see Fig. 2) to P_1. For the remaining three ears, proceed accordingly.

Observe that a guard that covers the spike of some element e_i must lie in a cone that leads from this spike to some point s_j. For each spike, there must be at least one guard that completely covers the spike, since the spike is one triangle in the terrain. Move each guard that lies in only one cone (i.e. not in an intersection area of several cones) to the endpoint s_j of the cone. Move all guards that lie in an area where at least two cones intersect and that are below the barrier line $y = b$ (or above the barrier line $y = 2y_0 - b$) to the endpoint of s_j of any of the intersecting cones.

For guards that lie in an intersection of two cones from different elements $e_q; e_r$, proceed as follows: Note that in this case we have one guard available to cover two elements. Determine how the spikes e_q^0 and e_r^0 of the mirror image are covered. If they are covered by a guard that lies in an intersection of two cones from e_q^0 and e_r^0, we have two guards available to cover two elements and the problem is resolved by moving one of the two guards available to any s_j of which e_q is a member, and by moving the other guard to any s_i of which e_r is a member. If e_q^0 is covered by a guard that lies in an intersection of the two cones of e_q^0 and some $e_{q'}^0$ and if $e_{q'}^0$ is also covered by a guard at some s_i, then there are four guards available to cover four elements and the problem is resolved by moving the available guards to appropriate s_i's. If e_q^0 is only covered by a guard that lies in the intersection of two cones of e_q^0 and some $e_{q'}^0$ that is not covered by a guard at any s_i and if e_r^0 is only covered by a guard that lies in the intersection of two cones of e_r^0 and some $e_{r'}^0$ that is not covered by a guard at any s_i, then we let $M' = fq; rg$ and determine how the mirror images of $e_{q'}^0$ and $e_{r'}^0$, which are $e_{q'}$ and $e_{r'}$, are covered. If they are covered by a guard that lies in the intersection of two cones of $e_{q'}$ and $e_{r'}$, then we have four guards available to cover four elements and the problem is resolved by moving the available guards. If $e_{q'}$ is covered by a guard that lies in an intersection of two cones of $e_{q'}$ and

some $e_{q''}$ and if $e_{q''}$ is also covered by a guard at some s_i or if $q'' \in M'$, then we have five guards available to cover five elements and the problem is resolved by moving the available guards. If e'_q is only covered by a guard that lies in the intersection of two cones of $e'_{q'}$ and some $e'_{q''}$ that is not covered by a guard at any s_i and if $e'_{r'}$ is only covered by a guard that lies in the intersection of two cones of $e'_{r'}$ and some $e'_{r''}$ that is not covered by a guard at any s_i, and if neither q'' nor r'' are in M', then we add q' and r' to M' and proceed accordingly for the mirror images of $e_{q''}$ and $e_{r''}$, which are $e'_{q''}$ and $e'_{r''}$. This procedure will stop after n=2 iterations at the latest, since two indices are added to M' in each step. After n=2 steps, the number of guards available will be greater or equal to the number of elements to be covered.

Guards that lie inside the polygon but outside the cones cannot cover any spikes completely and are therefore removed. Guards that lie outside the polygon are also removed.

This rearrangement of guards correctly guards the terrain without increasing the number of guards. A solution for the SC-instance can be determined by including set s_j in the SC solution if and only if there is a guard at point s_j. □

Lemmas 6 and 7 establish the following theorem:

Theorem 1. *An optimum solution of the SC-instance I contains k sets, if and only if an optimum solution of the TC-instance $f(I)$ contains $k+4$ guards.*

The description of the function g also shows that we are able to efficiently find an optimum solution of the SC-instance I, if we are given an optimum solution of the TC-instance $f(I)$.

4.3 The Reduction is Polynomial

Note that $d; d'; y_0; h; b$ are all constants in our reduction. The values for b' and for all D_i are computable in polynomial time and can be expressed with $O(n \log m)$ bits. Therefore, the function f runs in time polynomial in the size of the input SC-instance, since it only produces a polynomial number of triangles from which each corner can be computed in polynomial time and each corner takes at most $O(n \log m)$ bits to be expressed. It is obvious that the function g runs in polynomial time, since it only involves moving around a polynomial number of guards. If the number of guards is super-polynomial, we have an immediate transformation by selecting every set in the SC-instance. It takes polynomial time to move each guard, since it needs to be determined in which cone(s) a guard lies.

The polynomiality of the reduction and Theorem 1 establish the following corollary:

Corollary 1. Terrain Cover *is NP-hard.*

5 An Inapproximability Result

In order to get a strong inapproximability result, we take advantage of a property of the SC-instances produced in the reduction in [4], used to prove an optimum inapproximability result for SC.

Lemma 8. *Let N be the number of elements and let M be the number of sets in any SC-instance produced by the reduction in [4]. Then $M \quad N^5$ holds.*

Proof. $N = mR = m(5n)^l$ according to and adopting the notation of [4]. There are k provers. Each of them can be asked $Q = n^{\frac{l}{2}}(\frac{5n}{3})^{\frac{l}{2}}$ questions. An answer contains $\frac{l}{2} + \frac{3l}{2} = 2l$ bits, therefore there are 2^{2l} possible answers for each question. Since for each prover and each question/answer-pair a set is added, there are $M = k \ Q \ 2^{2l}$ sets. We prove that there is a constant t such that $N^t > M$, which is equivalent to $t > \frac{\log M}{\log N}$, where log denotes the base 2 logarithm. To do so, observe that $\frac{\log M}{\log N} = \frac{\log k + \log Q + 2l}{\log m + l \log 5 + l \log n}$. Since $Q = n^{\frac{l}{2}}(\frac{5n}{3})^{\frac{l}{2}} = (\frac{5n^2}{3})^{\frac{l}{2}}$, we get $\frac{\log M}{\log N} = \frac{\log k + \frac{l}{2}\log\frac{5}{3} + l\log n + 2l}{\log m + l\log 5 + l\log n}$. Since $m = n^{(l)}$, there must be a constant $c > 0$ with $m \quad n^{cl}$ for large enough values of l. Since $k < l$ we get $\frac{\log M}{\log N} = \frac{l(\frac{\log k}{l} + \frac{1}{2}\log\frac{5}{3} + \log n + 2)}{l(c\log n + \log 5 + \log n)} \quad \frac{1+1+2+\log n}{\log n}$. Since n is the number of variables in the input instance of 5-Occurrence-3-Sat, we can assume $n \quad 2$. Therefore, we get $\frac{\log M}{\log N} \quad 5$. ∎

Now consider only those SC-instances that are produced in the reduction in [4] and their corresponding TC-instances. Then, an approximation ratio of $(1 -)\ln n$ for any > 0 cannot be guaranteed by a polynomial algorithm for those SC-instances unless $NP \quad TIME(n^{O(\log\log n)})$, since this would imply that 5-Occurrence-3-Sat could be solved efficiently.

Theorem 2. *For all SC-instances I produced in the reduction in [4] and their corresponding TC-instances $f(I)$, there is a constant $c > 0$ such that, if TC for all considered instances can be approximated by a polynomial algorithm with an approximation ratio better than $c(1 -)\ln|f(I)|$ for any > 0, then SC for all considered instances can be approximated with an approximation ratio better than $(1 -)\ln n$, where n is the number of elements in the SC-instance.*

Proof. If TC can be approximated with ratio better than $c(1 -)\ln|f(I)|$, then we can find an approximate solution A' for each TC-instance that satisfies $\frac{|A'|}{|OPT'|} \quad c(1 -)\ln|f(I)|$, where OPT' is an optimum solution of the TC-instance. Let $A = g(A')$ be the corresponding approximate solution for the SC-instance and let $OPT = g(OPT')$ be the optimum solution for the SC-instance. Because of Theorem 1 and the description of the function g, $|A'| = |A| + 4$ and $|OPT'| = |OPT| + 4$. We have $\frac{|A|+4}{|OPT|+4} \quad c(1 -)\ln|f(I)|$ and therefore $\frac{|A|}{|OPT|} \quad c(1 -)\ln|f(I)| + \frac{4}{|OPT|}(c(1 -)\ln|f(I)|) - \frac{4}{|OPT|}$. With $|OPT| \quad 1$, we get $\frac{|A|}{|OPT|} \quad 5(c(1 -)\ln|f(I)|)$ We need to express the number of triangles $|f(I)|$ of the TC-instance through the number of elements n in the SC-instance. Observe that the terrain of the TC-instance can always be triangulated such that the number of triangles is $O(nm)$. Therefore, $|f(I)| \quad nm\gamma$ for some constant γ. Because we can assume $\gamma < n$ and because of Lemma 8, we get $|f(I)| \quad nn^5 n = n^7$. (Note that if we had not restricted the set of possible SC-instances, then $m = 2^n$ would be possible and we would get a much weaker result.) Therefore, $\frac{|A|}{|OPT|} \quad 5 \quad 7c(1 -)\ln n = 35c(1 -)\ln n$. Thus, $c = \frac{1}{35}$. ∎

Thus, if TC could be approximated with a ratio $\frac{1-}{35} \ln jf(I)j$, then SC could be approximated with a ratio $(1 -) \ln n$ for any > 0. The contraposition of this sentence establishes our main result. Since SC cannot be approximated with a ratio $(1 -) \ln n$ according to [4], we get:

Theorem 3. *TC cannot be approximated by a polynomial time algorithm with an approximation ratio of* $\frac{1-}{35} \ln n$ *for any* > 0*, where* n *is the number of triangles, unless* $NP \quad TIME(n^{O(\log \log n)})$.

6 Conclusion

Theorem 3 together with our approximation algorithm with ratio $\ln n + 1$ settles the approximability of Terrain Cover up to a constant factor. It shows that Terrain Cover belongs to the relatively small family of NP-optimization problems with an approximation threshold of a non-trivial nature. Unfortunately, the approximation algorithm has an excessive running time and excessive space requirements, far too much for practical purposes if we take into account that the solution obtained might be far off the optimum. Therefore, it remains open how to solve the Terrain Cover problem in a practical situation. Our inapproximability result carries over to the problem of guarding a 2.5-dimensional terrain with guards on the terrain. As an aside, note that the restriction that each triangle must be covered completely by a single guard can be dropped without any consequences for the inapproximability result [3]. In that case, however, the proposed approximation algorithm cannot be applied.

References

1. P. Bose, T. Shermer, G. Toussaint and B. Zhu; *Guarding Polyhedral Terrains*; Computational Geometry 7, pp. 173-185, Elsevier Science B. V., 1997.
2. D. Z. Chen, V. Estivill-Castro, J. Urrutia; *Optimal Guarding of Polygons and Monotone Chains (Extended Abstract)*; Proc. Seventh Canadian Conference on Computational Geometry, August 1995, 133-138.
3. St. Eidenbenz, Ch. Stamm, P. Widmayer; *Inapproximability of Some Art Gallery Problems*; manuscript, 1998.
4. Uriel Feige; *A Threshold of* $\ln n$ *for Approximating Set Cover*; STOC'96, pp. 314-318,1996.
5. 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).
6. B. Nilsson; *Guarding Art Galleries - Methods for Mobile Guards*; doctoral thesis, Department of Computer Science, Lund University, 1994.
7. J. O'Rourke; *Art Gallery Theorems and Algorithms*; Oxford University Press, New York (1987).
8. T. Shermer; *Recent Results in Art Galleries*; Proc. of the IEEE, 1992.
9. J. Urrutia; *Art Gallery and Illumination Problems*; to appear in Handbook on Computational Geometry, edited by J.-R. Sack and J. Urrutia, 1998.
10. B. Zhu; *Computing the Shortest Watchtower of a Polyhedral Terrain in* $O(n \log n)$ *Time*; in Computational Geometry 8 (1997), pp. 181-193, Elsevier Science B.V.

Two-Center Problems for a Convex Polygon
(Extended Abstract)

Chan-Su Shin[1], Jung-Hyun Kim[1], Sung Kwon Kim[2], and Kyung-Yong Chwa[1]

[1] Dept. of Comp. Sci., KAIST, Taejon, 305-701, KOREA
{cssin, jhkim, kychwa}@jupiter.kaist.ac.kr
[2] Dept. of Comp. Sci. Eng., Chung-Ang U., Seoul, 156-756, KOREA
ksk@point.cse.cau.ac.kr

1 Introduction

Let S be a set of n points in the plane. The *standard 2-center problem* for S is to cover S by the union of two congruent closed disks whose radius is as small as possible. The standard 2-center problem has been studied extensively studied. The best algorithm, due to Sharir [15] and slightly improved by Eppstein [9], runs in randomized expected $O(n \log^2 n)$ time. As a variant of the standard 2-center problem, we can consider the *discrete 2-center problem* for S that finds two congruent closed disks whose union covers S, and whose centers are two points of S. Recently, this problem was solved in $O(n^{4/3} \log^5 n)$ time [1].

In this paper, we consider some variants of the 2-center problems. Let G be a convex polygon of n vertices in the plane. We consider the *standard 2-center problem* for G: find two congruent closed disks whose union covers G (its boundary and interior) and whose radius is minimized. We also consider its *discrete* version with centers at two vertices of G. Our convex-polygon 2-center problem differs from the point-set problem: (1) the input is a convex polygon and (2) the union of two disks should cover the edges of G as well as its vertices. The former means our problem is easier than the point-set problem, but the latter tells us that it might be more difficult.

We assume that the vertices of G are in *general circular position*, meaning that no four or more vertices are co-circular. The model of computation used throughout the paper is the extended real RAM (for details refer to [14]). Our results are summarized as follows.

- We first consider the **discrete** 2-center problem for a convex polygon. We show that the problem can be solved in $O(n \log^3 n)$ time. This result can be used to solve the discrete point-set 2-center problem where the points form the vertices of a convex polygon in $O(n \log^3 n)$ time. The discrete 2-center problem for a set of the points in arbitrary positions is solved in $O(n^{4/3} \log^5 n)$ time [1], thus our result says that the convexity plays a crucial role in improving the time bound.
- We next consider the **standard** 2-center problem for a convex polygon, and present an $O(n^2 \log^3 n)$-time algorithm. The best known deterministic 2-center algorithm for a point set runs in $O(n \log^4 n / \log \log n)$ time; this bound can be obtained by combining the results presented by Eppstein [9] and by Chan [7].

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 199–210, 1998.
© Springer-Verlag Berlin Heidelberg 1998

Thus there is a room for improvement here, however we have been not able to reduce our bound further.

All algorithms presented in this paper are based on the parametric search technique proposed by Megiddo [13]. Many of point-set center problems have been solved by the technique [15,9,1]. In Section 2, we briefly review the parametric search technique. We explain how to solve the discrete and standard 2-center problems for the convex polygon in Section 3 and Section 4, respectively.

2 Parametric Search Technique

Parametric search is an optimization technique which can be applied in situations where we seek a minimum parameter r satisfying the *monotone* condition that is met by all $r \geq r$ but not by any $r < r$. The strategy of the parametric search is to design efficient sequential and parallel algorithms for the corresponding decision problem: decide whether a given parameter r is smaller than or larger than or equal to the minimum parameter r.

Assume that we have both an efficient sequential algorithm A_s that, given an input data and a fixed radius r, decides if $r < r$ or $r \geq r$, and a parallel algorithm A_p, running on Valiant's model of computation [16]. We will denote the running time of A_s by T_s, the running time of A_p by T_p, and the number of processors it uses by W_p. Assume moreover that the flow of execution of A_p depends only on comparisons, each of which is resolved by testing the sign of some low-degree polynomial in r. The parametric search technique executes A_p "generically", without specifying the value of r, with the intention of simulating its execution at the unknown value r.

At each step of A_p, where there are at most W_p independent comparisons to resolve, the roots of the associated polynomials are computed. If we know between which two of them r lies, we can resolve the comparisons, and continue to the next parallel step. Thus we search r in the list of $O(W_p)$ roots, using binary search and utilizing A_s to decide if $r_i < r$ or $r_i \geq r$. This search requires $O(W_p + T_s \log W_p)$ time per parallel step, for a total of $O(W_p T_p + T_s T_p \log W_p)$ time.

Throughout this execution of the generic A_p, we obtain a sequence of progressively smaller intervals, each known to contain r. When the entire algorithm terminates, it will have produced a left-closed interval I so that r is either its left endpoint or an interior point. However, the latter case is impossible, because the output of the generic algorithm is the same for all r in the interior of I, but the output must change at r, by definition. Hence, r is the left endpoint of I. Clearly, all problems considered in this paper satisfy the monotone condition that G is covered by two disks of all radius $r \geq r$, but not by two disks of any radius $r < r$.

3 Discrete Two-Center Problem for a Convex Polygon

Let G be a convex polygon of n vertices in the plane. The problem is to find two congruent closed disks with centers at some vertices of G whose union covers G and whose radius is minimized. To apply the parametric search technique to this problem,

we have to solve the following decision problem: Given a fixed radius $r > 0$, decide whether G is covered by (the union of) two congruent disks of radius r. As it will be proved below, for this decision problem, $T_s = O(n \log n)$, $T_p = O(\log n)$, and $W_p = O(n \log n)$. Thus the overall algorithm runs in $O(n \log^3 n)$ time.

For convenience, we assume that the vertices of G cannot be covered by any single disk of radius r with its center at any vertex of G; this can be checked in $O(n)$ time by computing the farthest vertex for each vertex of G [3]. To help describe the decision algorithm, some definitions and notation are needed. Consider two points p and q on the boundary of G, not necessarily vertices of G. The *vertex interval* $hp; qi$ of G is defined to be the set of vertices of G that we meet when we move in the clockwise direction on the boundary of G from p to q. Note that $hp; qi \neq hq; pi$ and $hp; qi \cup hq; pi$ is equal to the set of vertices of G. For a vertex v of G, we shall denote the disk of radius r centered at v by $D(v)$. Let $a(v)$ (resp., $b(v)$) be the first intersection point of $@D(v)$[1] with $@G$ as we follow $@G$ starting from v in the clockwise (resp., counterclockwise) direction. Let us define $A(v)$ to be the subchain of $@G$ that is defined when we move in the clockwise direction on $@G$ from $b(v)$ to $a(v)$. Then $A(v)$ contains v.

Lemma 1. *The convex polygon G is covered by two disks of radius r if and only if there exists a pair of vertices, u and w, of G such that $@G \subseteq A(u) \cup A(w)$.*

Proof. If such a pair of vertices, u and w, exists, then G is clearly covered by two disks of radius r; one disk covering $A(u)$ and the other disk covering $A(w)$. Suppose now that G is covered by two disks D_1 and D_2. Without loss of generality, we assume that D_1 is to the left of D_2. It suffices to prove that if the center of D_1 is located at some vertex v, then the complementary boundary $@G - A(v)$ should be covered by D_2. For a contradiction, assume that there is a point $z \in @G - A(v)$ that is not covered by D_2. Consider two vertex intervals $ha(v); zi$ and $hz; b(v)i$. Since G cannot be covered by a single disk of radius r and G is convex, there must be at least one vertex in each vertex interval that is not contained in D_1. Let v' and v'' be such vertices of $ha(v); zi$ and $hz; b(v)i$, respectively. Of course, $v' \neq v''$. It is easily checked that v and z must be covered only by D_1, and v' and v'' must be covered only by D_2. This directly implies that $@D_1$ and $@D_2$ intersect at least four times, which is a contradiction. □

We are now ready to give an algorithm for the decision problem. First, compute $A(v)$ for each vertex v of G. Next, find a pair of vertices, u and w, such that the union of $A(u)$ and $A(w)$ covers $@G$. Such a pair can be found by solving the *minimum circle cover problem* [12]. The problem is defined as follows: Given n circular-arcs on a circle of radius r, find a minimum number of circular-arcs whose union covers the circle. We can easily transform $A(v)$'s into circular-arcs in linear time. Provided that the circular-arcs are sorted according to their starting points in the clockwise direction, the minimum circle cover problem is solved by a sequential algorithm [12] which runs in $O(n)$ time and by parallel algorithms [6,11] which run in $O(\log n)$ time with $O(n)$ processors. Thus, if the minimum number is equal to 2, then G is covered by two disks of radius r. Otherwise (i.e., > 2), G cannot be covered by any two disks of radius

[1] To denote the boundary of a closed, bounded region R, we will use the notation $@R$.

r. The most time-consuming step in our algorithm is a step of computing $A(v)$'s. In what follows, we show that the step can be done in $O(n \log n)$ sequential-time, and in $O(\log n)$ parallel-time with $O(n)$ processors.

Let p and q be the vertices of G whose distance is the diameter of G. Assume that p and q lie on x-axis, and p is to the left of q. The vertices divide $@G$ into two chains; a lower chain $G_L = (u_1; u_2; \quad ; u_{n_1})$ and an upper chain $G_U = (w_1; w_2; \quad ; w_{n_2})$, where $u_1 = w_1 = p$ and $u_{n_1} = w_{n_2} = q$. Then the chains are all x-monotone. In the remainder of this section, we explain the method of computing $A(u_i)$'s only.

To compute $A(u_i)$ for a vertex u_i of G_L, we compute $a(u_i)$ and $b(u_i)$, separately. Since the definition of $b(u_i)$ is symmetrical to that of $a(u_i)$, we will explain only the method of computing $a(u_i)$ for each $u_i \in G_L$. We partition the set of vertices of G_L into two subsets, $S_1 = \{u_i \mid a(u_i) \in G_L\}$ and $S_2 = \{u_i \mid a(u_i) \in G_U\}$. To determine to which subset each u_i belongs, it suffices to test whether or not $D(u_i)$ contains p, because p and q are the farthest vertex pair of G.

Consider two vertices u_i and u_j of S_1 such that $i < j$. Since p and q are the farthest pair of G and G_L is x-monotone, $a(u_j)$ appears after $a(u_i)$ when we move from p to q. Thus, we can compute each $a(u_i)$ for each $u_i \in S_1$, in a total of $O(n)$ time, as we traverse the edges of G_L, from left to right, one by one.

Let us now consider how to compute $a(u_i)$ for each $u_i \in S_2$. Suppose that w_k is a vertex of G_U immediately to the right of $a(u_i)$. The vertex w_k is the first vertex that is not contained by $D(u_i)$ when we follow G_U from p to q. In other words, $D(u_i)$ covers the vertex interval $[u_i; w_{k-1}]$ but not $[u_i; w_k]$. We will compute the vertex w_k instead of $a(u_i)$; $a(u_i)$ is the intersection point between the edge $(w_{k-1}; w_k)$ and $@D(u_i)$. For a vertex interval L, we denote by $I(L)$ the intersection of disks of radius r centered at the vertices of L, that is, $I(L) = \bigcap_{v \in L} D(v)$. The intersection $I(L)$ is a closed, bounded region in the plane, whose boundary consists of at most $|L|$ arcs each of which is a circular-arc bounding $D(v)$ for $v \in L$. Since $D(u_i)$ covers $[u_i; w_{k-1}]$ and does not cover $[u_i; w_k]$, it holds that $u_i \in I([u_i; w_{k-1}])$ and $u_i \notin I([u_i; w_k])$. However, since $[u_i; w_1]$ is always contained in $D(u_i)$ for any vertex $u_i \in S_2$, it is also true that $u_i \in I([w_1; w_{k-1}])$ and $u_i \notin I([w_1; w_k])$.

To find w_k, we maintain a complete binary tree T. It stores the vertices $w_1; \quad ; w_{n_2}$ in its leaves from left to right. A node $\quad$ of T represents $I(\ )$, where $I(\ )$ denotes the intersection of the disks centered at the vertices that are stored at the leaves of the subtree rooted at $\quad$. If a node $\quad$ is the leaf storing w_j, then $I(\ )$ is $D(w_j)$ itself. If $\quad$ is an internal node with left child $!_l$ and right child $!_r$, then $I(\ ) = I(!_l) \setminus I(!_r)$. The boundary $@I(\ )$ is divided into an upper chain $@I^+(\ )$ and a lower chain $@I^-(\ )$ by the leftmost and rightmost points on $@I(\ )$. Actually, each node $\quad$ maintains $@I(\ )$ in two doubly linked lists; one for $@I^+(\ )$ and the other for $@I^-(\ )$. We show in Appendix A that T can be computed in $O(n \log n)$ sequential-time and in $O(\log n)$ parallel-time with $O(n)$ processors. If we can find the vertex w_k with the help of T in $O(\log n)$ time, then we can find all w_k's (i.e., all $a(u_i)$'s) for each $u_i \in S_2$ within the bound we want.

¿From the definition of w_k, it holds that $u_i \in I([w_1; w_{k-1}])$ but $u_i \notin I([w_1; w_k])$. To search it, traverse T in a top-down fashion. At the root $\quad$ of T, check whether u_i is contained in $I(\ )$. The inclusion test is easily done in $O(\log n)$ time through binary

searches in $@I^+(\ )$ and in $@I^-(\ )$. If $\ 2\ I(\ )$, then the vertices of G_U (including q) are contained in $D(u_i)$. If q is contained in $D(u_i)$, then the vertices of G_L are also contained in $D(u_i)$. That is, G is covered by the disk $D(u_i)$. But this situation cannot occur because of the assumption that G cannot be covered by a single disk. Thus we automatically know that $u_i\ 2\ I(\ )$. Next, do the same test for the left child $!_l$ of . If $u_i\ 2\ I(!_l)$, then this implies that w_k is not stored at some leaf in the subtree rooted at $!_l$, so we go to the right child $!_r$ of ; otherwise, go to the left child $!_l$. Continue this process until one reaches at some leaf in T. Then the vertex stored at the leaf is exactly w_k. Since the inclusion test at a node takes $O(\log n)$ time, we need $O(\log^2 n)$ time totally. However, it does not match the bound that we want.

To reduce the bound to $O(\log n)$, we employ the *fractional cascading technique* [8]. Consider the following problem: Given a binary tree of n nodes such that every node contains a sorted list $X(\ )$, construct a data structure that, given a walk of length h from the root to a leaf and an arbitrary element x, enables a single processor to locate x quickly in each $X(\ )$ for each on the walk. The fractional cascading technique provides an efficient solution to this problem as follows:

Lemma 2. [8,4] *One can construct such a data structure, in $O(m)$ time or in $O(\log m)$ time using $O(m/\log m)$ processors, that enables to locate x in each $X(\ )$ in $O(\log m + h)$ time, where $m = n + \sum jX(\ )j$.*

The tree T has two sorted lists $@I^+(\ )$ and $@I^-(\ )$ at each node . The sum of size of $@I^+(\ )$'s for nodes at each level is $O(n)$, so the total size m is $O(n \log n)$. Applying Lemma 2 to the sorted lists $@I^+(\ )$'s in T, we can construct a data structure in $O(n \log n)$ sequential-time and $O(\log n)$ parallel-time with $O(n)$ processors. Construct another data structure for $@I^-(\ )$'s in T. Then we can perform all inclusion tests to find w_k in $O(\log m + h) = O(\log n)$ time by Lemma 2. As a result, we can compute $a(u_i)$'s for all vertices $u_i\ 2\ S_2$ in $O(n \log n)$ sequential-time and $O(\log n)$ parallel-time with $O(n)$ processors.

Theorem 1. *Given a convex polygon G of n vertices in the plane, we can solve the discrete 2-center problem for G in $O(n \log^3 n)$ time.*

Theorem 2. *Given a set S of n points which form a convex polygon, we can solve the discrete 2-center problem for S in $O(n \log^3 n)$ time.*

4 Standard 2-Center Problem for a Convex Polygon

In this section, we present an $O(n^2 \log^3 n)$-time algorithm for finding two optimal disks that cover a convex polygon G. To apply the parametric search technique, we solve the decision problem: Given a fixed radius $r > 0$, decide whether G is covered by two disks of radius r. The decision problem is solved below as follows; $T_s = O(n^2 \log n)$, $T_p = O(\log n)$, and $W_p = O(n^2)$. Thus, the overall algorithm runs in $O(n^2 \log^3 n)$ time.

Let D_1 and D_2 be two closed disks of radius r. We want to decide whether $G \quad D_1 \ [\ D_2$ by locating D_1 and D_2 properly. By $G \quad D_1 \ [\ D_2$, we mean the union

of D_1 and D_2 contains the edges of G as well as the vertices of G. Let c_i denote the center of D_i, and let C_i denote the circle bounding D_i, for $i = 1; 2$. We may assume that two centers, c_1 and c_2, are as close as possible. Then we can say that, for $i = 1; 2$, the circle C_i passes through at least one vertex of G that lies on the portion of C_i that appears on the boundary of the convex hull of $D_1 \cup D_2$. Such a vertex pair $(p; q)$ through which C_1 and C_2 can pass in order to be $G \subseteq D_1 \cup D_2$ is called a *candidate vertex pair* of G. We furthermore assume that G is not covered by a single disk of radius r; this is easily checked in $O(n)$ time [2]. From the above two assumptions, we can easily prove that $p \notin D_2$ and $q \notin D_1$ if C_1 passes through p and C_2 does through q for a candidate vertex pair $(p; q)$. Using the same approach to Lemma 3 in [10], we can bound the number of candidate vertex pairs of G by $O(n)$.

Lemma 3. [10] *Under the assumption that the vertices of G are in general circular positions, the number of candidate vertex pairs of G is $O(n)$, and the pairs can be found in $O(n)$ time.*

Suppose now that C_1 and C_2 pass through vertices p and q, respectively, where $(p; q)$ is a candidate vertex pair. We assume that p and q lie on x-axis, and p is to the left of q. The candidate pair $(p; q)$ divides ∂G, the boundary of G, into two chains; a lower chain $G_L = (u_1; \ldots; u_{n_1})$ and an upper chain $G_U = (w_1; \ldots; w_{n_2})$, where $u_1 = w_1 = p$ and $u_{n_1} = w_{n_2} = q$. We shall denote by P and Q the circles of radius r centered at vertices p and q, respectively. Then the center c_1 (resp., c_2) should be located on P (resp., Q). More precisely, c_1 is located on P_R and c_2 is located on Q_L. Here, P_R (resp., Q_L) denotes the semi-circle of the circle P (resp., Q) that lies on the right (resp., left) of the vertical line passing through p (resp., q). For each vertex v of G, draw a disk $D(v)$ and compute intersections of $\partial D(v)$ with P_R. Since two distinct circles intersect at most twice, the semi-circle P_R has at most $2n - 2$ intersections. Sort them along P_R in their y-coordinates. Then the intersections partition P_R into $O(n)$ circular-arcs A_j's.

Suppose again that the center c_1 of D_1 is on the j-th arc A_j. Following the edges of G_L from p to q, one meets an edge $(u_{k-1}; u_k)$ that C_1 firstly intersects. Then u_k is the first vertex that is not covered by D_1 when we follow the vertices of G_L from p to q. Similarly, we define the firstly intersected edge $(w_{l-1}; w_l)$ of G_U with C_1. Wherever c_1 is located on A_j, the set of the vertices covered by D_1 is the same, thus vertices u_k and w_l are uniquely determined with respect to A_j. Then the following lemma holds.

Lemma 4. *If the center c_1 is located on A_j and $G \subseteq D_1 \cup D_2$, then the vertex interval $\langle w_l; u_k \rangle$ is covered by D_2.*

Proof. For a contradiction, assume that there is a vertex $z \in \langle w_l; u_k \rangle$ not covered by D_2. Without loss of generality, assume that z is a vertex of G_U between u_k and q. Clearly, $z \neq u_k$ and $z \neq q$. Since $G \subseteq D_1 \cup D_2$, p and z must be covered only by D_1, and u_k and q must be covered only by D_2. This directly implies that C_1 and C_2 intersect at least three times, which is a contradiction. □

Let A_j^0 be an arc on Q_L where the center c_2 of D_2 must be located to cover $\langle w_l; u_k \rangle$. By the above lemma, the arc A_j^0 is defined as $A_j^0 = \bigcap_{v \in \langle w_l; u_k \rangle} (D(v) \setminus Q_L)$.

If $A_j^0 = \emptyset$, then $D_1 \cup D_2$ cannot cover G wherever c_1 is located on A_j. If $A_j^0 \neq \emptyset$ and the center c_2 is located on A_j^0, then $D_1 \cup D_2$ covers the vertices of G; however, we cannot say that every edge of G is covered by $D_1 \cup D_2$ because edges $(u_{k-1}; u_k)$ and $(w_{l-1}; w_l)$ may not be covered. Thus we have to check whether $D_1 \cup D_2$ covers those two edges. We will iterate this process over each arc A_j. If there are no arc pairs $(A_j; A_j^0)$ admitting $G \subseteq D_1 \cup D_2$, then G cannot be covered by any two disks of radius r. The decision algorithm is summarized below.

Algorithm Standard_2-Center_Decision $(G; r)$
(1) compute the candidate vertex pairs of G by Lemma 3.
(2) FOR each candidate vertex pair $(p; q)$ DO
(note that only this FOR loop is a parametric search part.)
(2.1) compute the arcs A_j's on P_R and sort them.
(2.2) construct complete binary trees T_L (resp., T_U) with vertices of G_L (resp., G_U) in its leaves as we did in the previous section.
(2.3) FOR each arc A_j DO
(suppose that c_1 is located in A_j.)
(2.3.1) find two vertices u_k, and w_l by searching T_L and T_U.
(2.3.2) compute A_j^0 on Q_L, corresponding to A_j.
(assume that $A_j^0 \neq \emptyset$ and c_2 is located in A_j^0.)
(2.3.3) IF $(u_{k-1}; u_k)$ and $(w_{l-1}; w_l)$ are covered by $D_1 \cup D_2$
THEN return "YES". (G is covered by $D_1 \cup D_2$.)
return "NO". (G is not covered by any two disks of radius r)
End-algorithm

We can perform Step (2.1) and Step (2.2) in $O(n \log n)$ time and in $O(\log n)$ time with $O(n)$ processors; we use Lemma 11 in Appendix A for Step (2.2). For a fixed arc A_j, if we can do Steps (2.3.1) – (2.3.3) in $O(\log n)$ time with a single processor, then we can perform Step (2.3) in $O(n \log n)$ sequential-time and in $O(\log n)$ parallel time with $O(n)$ processors.

Using the search procedure presented in the previous section, we can do Step (2.3.1) in $O(\log n)$ time by searching T_L and T_U. Let us now compute the arc A_j^0 for a fixed arc A_j in Step (2.3.2). Note that $A_j^0 = \bigcap_{v \in \langle w_l; u_k \rangle} (D(v) \setminus Q_L)$. To do it, we construct two segment trees K_L and K_U. The leaves of K_U store the vertices of G_U in x-increasing order and each node of K_U stores a circular arc $\bigcap_{v \in S(\cdot)} (D(v) \setminus Q_L)$ on Q_L, where $S(\cdot)$ denotes the vertices stored at the leaves of the subtree rooted at $\cdot$. The tree K_U can be easily constructed in a bottom-up fashion in $O(n \log n)$ sequential-time and $O(\log n)$ parallel-time with $O(n)$ processors. Similarly, construct another segment tree K_L for G_L. Using the standard range searching technique [14], we can compute an arc $\bigcap_{v \in \langle w_l; q \rangle} (D(v) \setminus Q_L)$ from K_U and an arc $\bigcap_{v \in \langle q; u_k \rangle} (D(v) \setminus Q_L)$ from K_L in $O(\log n)$ time. Since $\langle w_l; u_k \rangle = \langle w_l; q \rangle \cup \langle q; u_k \rangle$, the intersection of these two arcs is the interval A_j^0. Consequently, we can compute intervals A_j^0 for all intervals A_j within the desired bound.

What remains is to check whether two edges $(u_{k-1}; u_k)$ and $(w_{l-1}; w_l)$ of G are covered by $D_1 \cup D_2$ when c_1 is in A_j and c_2 is in A_j^0 $(\neq \emptyset)$, for each pair $(A_j; A_j^0)$. Although a proof is not given here due to the space limitation, we can show that

such a test for one arc pair can be done in $O(1)$ time under the extended real RAM model. Thus Steps (2.3.1) (2.3.3) can be performed in $O(\log n)$ time with a single processor, and Step (2.3) can be done in $O(n \log n)$ sequential-time and in $O(\log n)$ parallel-time with $O(n)$ processors.

Since there are $O(n)$ candidate vertex pairs, the decision algorithm can be solved in $O(n^2 \log n)$ sequential-time and in $O(\log n)$ parallel-time with $O(n^2)$ processors. Hence, we have the following result.

Theorem 3. *Given a convex polygon of n vertices in the plane, we can solve the standard 2-center problem for it in $O(n^2 \log^3 n)$ time.*

5 Open Problems

An immediate open question is to improve the time bound of our algorithms. In particular, we believe that the $O(n^2 \log^3 n)$ bound for the standard 2-center problem could be lowered sufficiently. It would be interesting to consider the standard 2-center problem for a set of the points in convex positions. The standard point-set 2-center problem can be solved in $O(n \log^4 n / \log\log n)$ worst-case time [9,7]. In spite of numerous efforts we have been unable to beat the bound. It would be also interesting to consider the k-center problems for the convex polygon for general $k > 2$.

Acknowledgment: The first author specially thanks Sang-Min Park for a careful reading of the first draft.

References

1. P. K. Agarwal, M. Sharir, and E. Welzl. The discrete 2-center problem. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997. 147–155.
2. A. Aggarwal, L. J. Guibas, J. Saxe, and P. W. Shor. A linear-time algorithm for computing the Voronoi diagram of a convex polygon. *Discrete Comput. Geom.*, 4(6):591–604, 1989.
3. A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.
4. M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput.*, 18:499–532, 1989.
5. M. J. Atallah and M. T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3:535–548, 1988.
6. L. Boxer and R. Miller. A parallel circle-cover minimization algorithm. *Inform. Process. Lett.*, 32(2):57–60, 1989.
7. T. M. Chan. Deterministic algorithms for 2-d convex programming and 3-d online linear programming. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 464–472, 1997.
8. B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
9. D. Eppstein. Faster construction of planar two-centers. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, 1997.
10. V. Estivill-Castro and J. Urrutia. Two-floodlight illumination of convex polygons. In *Proc. 4th Workshop Algorithms Data Struct.*, pages 62–73, 1995.

11. S. K. Kim. *Parallel algorithms for geometric intersection graphs*. Ph.D. thesis, CS Dept., U. of Washington, 1990.
12. C. C. Lee and D. T. Lee. On a circle-cover minimization problem. *Inform. Process. Lett.*, 18:109–115, 1984.
13. N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30:852–865, 1983.
14. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.
15. M. Sharir. A near-linear algorithm for the planar 2-center problem. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 106–112, 1996.
16. L. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4(3):348–355, 1975.

A Constructing the tree T

We here consider a problem of building the tree T that stores a vertex interval of G in its leaves. Notice that the vertices of the vertex interval may not form a sorted sequence according to their x-coordinates. Suppose that the vertices v_0; ; v_{m-1} of the vertex interval are stored in the leaves of T from the left to the right. The vertices appear consecutively along $@G$ in the clockwise direction from v_0 to v_{m-1}. Each node of T maintains $@I(\ )$ as two doubly linked lists; one for $@I^+(\ )$ and the other for $@I^-(\ )$. Note that the vertices stored in the leaves of the subtree rooted at also form some vertex interval of G. The goal is to build T in $O(m \log m)$ sequential-time and in $O(\log m)$ parallel-time with $O(m)$ processors.

Before proceeding further, we need to state some useful geometric properties on the intersection $I(L)$ for a vertex interval L. Suppose that each arc of $@I(L)$ is labeled with an integer j if the arc belongs to the disk $D(v_j)$. Due to the space limitation, we shall omit all the proofs of lemmas presented in this appendix.

Lemma 5. *Let L be a vertex interval of G. When we follow the arcs of $@I(L)$ in the clockwise direction, their labels form a circular sequence with no duplications, and the circular sequence is a subsequence of the clockwise-sequence of the vertices in L.*

Lemma 6. *Let L_1 and L_2 be vertex intervals of G such that they are disjoint and every vertex of L_2 appears after all vertices of L_1 in the clockwise direction on $@G$. Then the number of intersections between $@I(L_1)$ and $@I(L_2)$ is zero, one (if the intersection is tangential), or two (proper intersections).*

The tree T is constructed in a bottom-up fashion; compute $@I(\ )$ for each node at each level from the bottom level to the top level. If a node is the leaf storing the vertex v_i, then $@I(\ )$ is $@D(v_i)$ itself. If is an internal node with left child $!_l$ and right child $!_r$, then $@I(\ ) = @(I(!_l) \setminus I(!_r))$. To compute $@I(\ )$, we should compute the intersections between $@I(!_l)$ and $@I(!_r)$. By Lemma 6, the number of intersections between the chains is at most two.

Sequential algorithm of constructing T: Since the number of the intersections is at most two, we can easily compute $@I(\ )$ from $@I(!_l)$ and $@I(!_r)$ in

$O(j@I(!_l)j + j@I(!_r)j)$ time by traversing the arcs of $@I(!_l)$ and $@I(!_r)$, simultaneously. Thus T is constructed in $O(m \log m)$ sequential-time.

Parallel algorithm of constructing T: To compute the intersections between $@I(!_l)$ and $@I(!_r)$, we compute all the intersections between two chains of them; $@I^+(!_l)$ and $@I^+(!_r)$, $@I^+(!_l)$ and $@I^-(!_r)$, $@I^-(!_l)$ and $@I^+(!_r)$, and $@I^-(!_l)$ and $@I^-(!_r)$. If they can be computed in $O(1)$ time with $O(j@I(!_l)j + j@I(!_r)j)$ processors, then the whole tree T can be constructed in $O(\log m)$ time with $O(m)$ processors. In what follows, we will explain how to compute the intersections between two chains.

Let P be a chain ($@I^+(!_l)$ or $@I^-(!_l)$) of $@I(!_l)$ with m_1 vertices and let Q be a chain ($@I^+(!_r)$ or $@I^-(!_r)$) of $@I(!_r)$ with m_2 vertices. Assume that the vertices of P and Q are ordered from the left to the right, and assume, for convenience, that m_1 and m_2 are square numbers. Select every $\sqrt{m_1}$-th vertex of P (including the leftmost and rightmost vertices of P), i.e., 0-th (leftmost) vertex, $(\sqrt{m_1})$-th vertex, $(2\sqrt{m_1})$-th vertex, , and finally $(\sqrt{m_1}\sqrt{m_1})$-th (rightmost) vertex. Then, at most $\sqrt{m_1} + 1$ vertices are selected; particularly, we shall call such sampled vertices *s-vertices*. We shall denote the $(i\sqrt{m_1})$-th vertex by p_i for $0 \quad i \quad \sqrt{m_1}$; note that p_0 and $p_{\sqrt{m_1}}$ are the leftmost and rightmost vertices of P, respectively. Let P_i be a subchain consisting of the arcs of P between p_i and p_{i+1} for $0 \quad i < \sqrt{m_1}$. Similarly, we define each subchain Q_j as a collection of the arcs of Q between q_j and q_{j+1}, where q_j is the $(j\sqrt{m_2})$-th vertex of Q for $0 \quad j \quad \sqrt{m_2}$. Note that $jP_ij \quad \sqrt{m_1}$ and $jQ_jj \quad \sqrt{m_2}$ for any $0 \quad i < \sqrt{m_1}$, $0 \quad j < \sqrt{m_2}$. The following lemmas will be used as basic routines.

Lemma 7. *For some fixed i, we can compute the intersections (if exist) between P_i and Q in $O(1)$ time with $O(m_1 + m_2)$ processors. The same holds for P and Q_j for some fixed j.*

Lemma 8. *Suppose that P and Q intersect at most once and their intersection, if exists, is not tangential. Then, we can determine whether they intersect, in $O(1)$ time with $O(m_1 + m_2)$ processors. Moreover, if they intersect, then we can report the intersect within the same bound.*

We will first explain how to detect the intersections between P and Q when P is an upper chain $@I^+(!_l)$ and Q is a lower chain $@I^-(!_r)$. Let $P^?$ (resp., $Q^?$) be a closed, unbounded region consisting of all points that lie in or below P (resp., in or above Q); see Figure 1 (a). Since at least one of four end vertices of P and Q must lie outside $P^?$ or $Q^?$, we can assume, without loss of generality, that p_0 is outside $Q^?$. We consider two cases: (1) $p_{\sqrt{m_1}}$ is inside $Q^?$, and (2) $p_{\sqrt{m_1}}$ is outside $Q^?$. In case (1), P and Q intersect at most once. If the upward vertical ray emanating from q_0 does not meet P, then they must intersect at exactly one point, and moreover the intersection cannot be tangential. Hence, we can find the intersection, by Lemma 8, in $O(1)$ time with $O(m_1 + m_2)$ processors. If the vertical ray meets P, then it is easy to see that P and Q do not intersect each other. The remaining case is that $p_{\sqrt{m_1}}$ is outside $Q^?$. The possible configurations that could occur are shown in Figure 1; the number of the intersections is zero, one (the intersection is tangential), or two (proper

intersections). The idea to handle this case is to find a point f on $@(P^? \setminus Q^?)$. Let l denote a vertical line through f. If f is not tangential, then one of two intersections between P and Q lies to the left of l and the other does to the right of l. We have now two subproblems, each of which can be solved by Lemma 8. If f is tangential, then f itself is the unique intersection. If f does not exist, then we conclude that P and Q have no intersections.

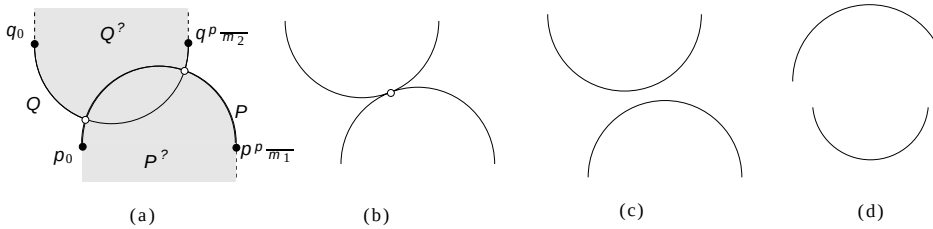


Fig. 1. Four possible configurations when p_0 and $p^p_{m_1}$ both are outside $Q^?$.

Let us now explain how to find a point f. Let P^0 be a polygonal chain that consists of line segments joining the s-vertices p_i and p_{i+1} for $0 \quad i \quad p_{m_1}$. First, compute the intersections between P^0 and Q. To do it, we use the result [5] that computes the intersection of a line (or a line segment) with a convex polygon in $O(1)$ time with the number of processors proportional to the square root of the size of the convex polygon. With a slight modification, we can apply the result to our case, so we can compute the intersections in $O(1)$ time with $O(p_{m_1 m_2})$ processors. If the intersection points between P^0 and Q are found, then one of them becomes f. Otherwise, we need some lemma to proceed further. Let $g \; 2 \; P^0$ and $h \; 2 \; Q$ be a closest pair of points (not necessarily vertices) that gives the minimum distance between P^0 and Q. Modifying the algorithm in [5] that computes the minimum distance between two disjoint convex polygons, the closest point pair g and h can be obtained in $O(1)$ time with $O(m_1 + m_2)$ processors. Let g_l and g_r be the vertices of P^0 immediately to the left and immediately to the right of g. We denote by $P(g_l; g_r)$ the subchain of the arcs of P between g_l and g_r. Clearly, the number of arcs of $P(g_l; g_r)$ is at most $2 p_{m_1}$; if g is an s-vertex p_i, then $g_l = p_{i-1}$ and $g_l = p_{i+1}$, so $P(g_l; g_r) = P_{i-1} [P_i$; otherwise (i.e., if g lies on a line segment $\overline{p_i p_{i+1}}$), $P(g_l; g_r) = P(p_i; p_{i+1}) = P_i$. Borrowing the idea from [5], we can prove the following lemma.

Lemma 9. *If f exists, then it must lie on $P(g_l; g_r)$.*

Finding the point f on $P(g_l; g_r)$ is equivalent to computing the intersections between $P(g_l; g_r)$ and Q. This is done by Lemma 7. As a consequence, we can find the intersections between the upper and lower chains in $O(1)$ time with $O(m_1 + m_2)$ processors

We next explain the method of finding the intersections of P and Q when both P and Q are lower chains. Let $P^?$ (resp., $Q^?$) be a closed region consisting of all points that lie in or above P (resp., Q). Without loss of generality, we assume that p_0 is

outside $Q^?$. As did previously, we consider two cases: (1) $p^p\overline{m_1}$ is inside $Q^?$, and (2) $p^p\overline{m_1}$ is outside $Q^?$. In case (1), P and Q intersect at most once, and the intersection $P \setminus Q$ can be found within the desired bound (refer to the case (1) when P and Q are the upper and lower chains, respectively). In the following, we consider the case (2), that is, p_0 and $p^p\overline{m_1}$ both are outside $Q^?$. Let t be a point on Q with minimum y-coordinate. The point t divides Q into the left subchain Q_L and the right subchain Q_R. Similarly, by the point s on P with minimum y-coordinate, P is divided into two subchains P_L and P_R. Note that each arc of P_R and Q_R (resp., P_L and Q_L) belongs to the fourth (resp., the third) quadrant[2]-arc of some circle of radius r. If t is outside $P^?$, then P and Q must intersect exactly twice (if we regard one tangential intersection as two intersections); one is $Q_L \setminus P_L$ and the other is $Q_R \setminus P_R$. Hence, by Lemma 8, we can compute the intersections within the desired bound. We now assume that t is inside $P^?$. Then one of the following situations occurs only: one is that $jP_L \setminus Q_Lj = 2$ but $jP_R \setminus Q_Rj = 0$, and the other is that $jP_L \setminus Q_Lj = 0$ but $jP_R \setminus Q_Rj = 2$. Suppose here that P_R and Q_R intersect at points z_1 and z_2, where z_1 is to the left of z_2. We shall find a point f on $@(P_R \setminus Q^?)$. Check if there exists an s-vertex p_i in P_R that is contained in $Q^?$; if exists, then f is set to the vertex p_i. If no s-vertices p_i's lie on $@(P_R \setminus Q^?)$, then $@(P_R \setminus Q^?)$ is completely contained in a subchain P_j for some $j < p\overline{m_1}$. This implies that if f exists, then it must lie on the subchain. To determine the subchain within constant time, we need the following lemma.

Lemma 10. *Let $d(p)$ be the vertical distance from a point p on P_R to its vertically projected point on Q. For any point p on $P_R(s; z_1)$, the vertical distance $d(p)$ is strictly increasing as p moves (from z_1) toward s. Also, for any point p on $P_R(z_2; p^p\overline{m_1})$, the vertical distance $d(p)$ is strictly increasing as p moves (from z_2) toward $p^p\overline{m_1}$.*

Note that the vertical distance $d(p_i)$ for each s-vertex p_i in P_R can be computed in $O(1)$ time with $O(m_1 + m_2)$ processors. Consider the s-vertices $p_l; p_{l+1};$ $; p^p\overline{m_1}$ in P_R, where p_l is the leftmost s-vertex among the s-vertices in P_R between s and z_1; if there are no s-vertices between them, then p_l is set to be s. The above lemma implies that the sequence of $d(p_l);$ $; d(p^p\overline{m_1})$ has at most two local minima. (In fact, if two local minima exist, then they occur at two consecutive s-vertices.) It is easy to show that the minima can be found in $O(1)$ time with $O(p\overline{m_1})$ processors. If the local minima occur at p_k and p_{k+1}, then the point f is contained in a subchain between p_k and p_{k+1}, i.e., P_k. If the local minima occur only at p_k, then f is contained either in P_k or in P_{k-1}; in this case, we can identify on which subchain of them f lies by computing both of $P_k \setminus Q$ and $P_{k-1} \setminus Q$. As a result, we have the following result.

Lemma 11. *The tree T for a vertex interval of G can be constructed in $O(m \log m)$ sequential-time and in $O(\log m)$ parallel-time with $O(m)$ processors. Here, m denotes the size of the vertex interval.*

[2] A point p in the plane partitions the plane into four quadrants. We refer to the quadrant $f(x; y)\ j\ x \quad x(p); y \quad y(p)g$ as the first quadrant. Here, $x(p)$ and $y(p)$ denote the x- and y-coordinates of p. The other quadrants are referred to in the counterclockwise direction as second, third, and fourth quadrants. Four quadrant-arcs of a circle is similarly defined.

Constructing Binary Space Partitions for Orthogonal Rectangles in Practice[?]

T. M. Murali[??], Pankaj K. Agarwal[???], and Jeffrey Scott Vitter[y]

Center for Geometric Computing
Department of Computer Science, Duke University
Box 90129, Durham, NC 27708-0129
{tmax,pankaj,jsv}@cs.duke.edu
WWW: http://www.cs.duke.edu/~{tmax,pankaj,jsv}

Abstract. In this paper, we develop a simple technique for constructing a Binary Space Partition (BSP) for a set of orthogonal rectangles in $\mathbb{R}^3$. Our algorithm has the novel feature that it tunes its performance to the geometric properties of the rectangles, e.g., their aspect ratios.
We have implemented our algorithm and tested its performance on real data sets. We have also systematically compared the performance of our algorithm with that of other techniques presented in the literature. Our studies show that our algorithm constructs BSPs of near-linear size and small height in practice, has fast running times, and answers queries efficiently. It is a method of choice for constructing BSPs for orthogonal rectangles.

1 Introduction

The Binary Space Partition (BSP) is a hierarchical partitioning of space that was originally proposed by Schumacker et al. [19] and was further refined by Fuchs et al. [10]. The BSP has been widely used in several areas, including computer graphics (global illumination [4], shadow generation [6,7], visibility determination [3,21], and ray tracing [15]), solid modeling [16,22], geometric data repair [12], network design [11], and surface simplification [2]. The BSP has been successful since it serves both as a model for an object (or a set of objects) and as a data structure for querying the object.

[?] A preliminary version of this paper appeared as a communication in the *Proceedings of the 13th Annual ACM Symposium on Computational Geometry, 1997*, pages 382–384.

[??] This author is affiliated with Brown University. Support was provided in part by National Science Foundation research grant CCR–9522047 and by Army Research Office MURI grant DAAH04–96–1–0013.

[???] Support was provided in part by National Science Foundation research grant CCR–93–01259, by Army Research Office MURI grant DAAH04–96–1–0013, by a Sloan fellowship, by a National Science Foundation NYI award and matching funds from Xerox Corp, and by a grant from the U.S.-Israeli Binational Science Foundation.

[y] Support was provided in part by National Science Foundation research grant CCR–9522047, by Army Research Office grant DAAH04–93–G–0076, and by Army Research Office MURI grant DAAH04–96–1–0013.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 211–222, 1998.
c Springer-Verlag Berlin Heidelberg 1998

Before proceeding further, we give a definition of the BSP. A *binary space partition* B for a set S of pairwise-disjoint triangles in $\mathbb{R}^3$ is a tree defined as follows: Each node v in B represents a convex polytope R_v and a set of triangles $S_v = \{s \setminus R_v \mid s \in S\}$ that intersect R_v. The polytope associated with the root is $\mathbb{R}^3$ itself. If S_v is empty, then node v is a leaf of B. Otherwise, we partition R_v into two convex polytopes by a *cutting plane* H_v. At v, we store the equation of H_v and the subset of triangles in S_v that lie in H_v. If we let H_v^- be the negative halfspace and H_v^+ be the positive halfspace bounded by H_v, the polytopes associated with the left and right children of v are $R_v \setminus H_v^-$ and $R_v \setminus H_v^+$, respectively. The left subtree of v is a BSP for the set of triangles $\{s \setminus H_v^- \mid s \in S_v\}$ and the right subtree of v is a BSP for the set of triangles $\{s \setminus H_v^+ \mid s \in S_v\}$. The size of B is the sum of the number of internal nodes in B and the total number of triangles stored at all the nodes in B.[1]

The efficiency of most BSP-based algorithms depends on the size and/or the height of the BSP. Therefore, several techniques to construct BSPs of small size and height have been developed [3,10,21,22]. These techniques may construct a BSP of size $\Omega(n^3)$ for some instances of n triangles. The first algorithms with non-trivial provable bounds on the size of a BSP were developed by Paterson and Yao. They show that a BSP of size $O(n^2)$ can be constructed for n disjoint triangles in $\mathbb{R}^3$ [17] and that a BSP of size $O(n\sqrt{n})$ can be constructed for n non-intersecting, orthogonal rectangles in $\mathbb{R}^3$ [18]. Agarwal et al. [1] consider the problem of constructing BSPs for fat rectangles. A rectangle is said to be *fat* if its aspect ratio is at most α, for some constant $\alpha \geq 1$; otherwise, it is said to be *thin*. If m rectangles are thin and the rest are fat, they present an algorithm that constructs a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$. A related result of de Berg shows that a BSP of linear size can be constructed for fat polyhedra in $\mathbb{R}^d$ [8].

In this paper, we consider the problem of constructing BSPs for orthogonal rectangles in $\mathbb{R}^3$. In many applications, common environments like buildings are composed largely of orthogonal rectangles. Further, it is a common practice (for example, in the BRL-CAD solid modeling system [13,20]) to approximate non-orthogonal objects by their orthogonal bounding boxes, since such approximations are simple, easy to manipulate, and often serve as very faithful representations of the original objects [9].

Our paper makes two important contributions. First, we develop and implement a simple technique for constructing a BSP for orthogonal rectangles in $\mathbb{R}^3$. Our algorithm has the useful property that it tunes its performance to the geometric structure present in the input, e.g., the aspect ratios of the input rectangles. While Agarwal et al. [1] use similar ideas, our algorithm is considerably simpler than theirs and is much more easy to implement. Moreover, our algorithm is "local" in the sense that in order to determine the cutting plane for a node v, it examines only the rectangles intersecting R_v. On the other hand,

[1] For each internal node v, we store the description of the polytope R_v. However, if v is a leaf, we do not store R_v, since it is completely defined by R_w and the cutting plane h_w, where w is the parent of v. Hence, we do not include the number of leaves while counting the size of B.

the algorithm of Agarwal et al. is more "global" in nature: to determine how to partition a node v, it uses splitting planes computed at ancestors of v in the BSP. Other "local" algorithms presented in the literature [3,21,22] can be easily incorporated into the framework of our algorithm but not into the Agarwal et al. algorithm. We also show that a slightly modified version of our algorithm constructs a BSP of size $n^{p}\overline{m}2^{O(\sqrt{\log n})}$ for a set of $n - m$ fat and n thin orthogonal rectangles in $\mathbb{R}^3$, achieving the same bound as the algorithm of Agarwal et al. [1].

We have implemented our algorithm to study its performance on "real" data sets. Our experiments show that our algorithm is practical: it constructs a BSP of near-linear size on real data sets (the size varies between 1.5 and 1.8 times the number of input rectangles).

The second contribution of our paper is a methodical study of the empirical performance of a variety of known algorithms for constructing BSPs. Our experiments show that our algorithm performs better than not only theoretical algorithms like that of Paterson and Yao [18] but also most other techniques described in the literature [3,10,22]. The only algorithm that performs better than our algorithm on some data sets is Teller's algorithm [21]; even in these cases, our algorithm has certain advantages in terms of the trade-off between the size of the BSP and query times (see Section 4).

To compare the different algorithms, we measure the size of the BSP each algorithm constructs and the time spent in answering various queries. The size measures the storage needed for the BSP. We use queries that are typically made in many BSP-based algorithms: *point location* (determine the leaf of the BSP that contains a query point) and *ray shooting* (determine the first rectangle intersected by a query ray).

2 Our Algorithm

In this section, we describe our algorithm New for constructing BSPs for orthogonal rectangles in $\mathbb{R}^3$. We first give some definitions, most of which are borrowed from Agarwal et al. [1].

We will often focus on a box B and construct a BSP for the rectangles intersecting it. We use S_B to denote the set $fs \setminus B \; j \; s \; 2 \; Sg$ of rectangles obtained by clipping the rectangles in S within B. We say that a rectangle in S_B is *free* if none of its edges lies in the interior of B; otherwise it is *non-free*. A *free cut* is a cutting plane that does not cross any rectangle in S and that either divides S into two non-empty sets or contains a rectangle in S. Note that the plane containing a free rectangle is a free cut.

A box B in $\mathbb{R}^3$ has six faces—*top, bottom, front, back, right,* and *left*. We say that a rectangle r in S_B is *long* with respect to a box B if none of the vertices of r lie in the interior of B. Otherwise, r is said to be *short*. We can partition long rectangles into three classes: a rectangle s that is long with respect to B belongs to the *top class* if two parallel edges of s are contained in the top and bottom faces of B. We similarly define the *front* and *right* classes. A long rectangle belongs to at least one of these three classes; a non-free rectangle belongs to a

unique class. Finally, for a set of points P, let P_B be the subset of P lying in the interior of B.

Our algorithm is recursive. At each step, we construct a BSP for the set S_B of rectangles intersecting a box B by partitioning B into two boxes B_1 and B_2 using an orthogonal plane, and recursively constructing a BSP for the sets of rectangles S_{B_1} and S_{B_2}. We start by applying the algorithm to a box that contains all the rectangles in S. We use F_B to denote the set of long rectangles in S_B. We define the measure (B) of B to be the quantity $jF_Bj + 2k_B$, where k_B is the number of vertices of rectangles in S_B that lie in the interior of B. We say that a class in F_B is *large* if the number of rectangles in that class is at least (B)=6. We split B using one of the following steps:

1. If S_B contains free rectangles, we use the free cut containing the median free rectangle to split B into two boxes. Note that such a free cut is not partitioned by any further cuts in B.
2. If all three classes in F_B are large, we split B as follows: Assume without loss of generality that among all the faces of B, the back face of B has the smallest area. Each long rectangle in the front class has an edge that is contained in the back face of B. We can find an orthogonal line $'$ in the back face that passes through an endpoint of one of these edges and does not intersect the interior of any other edge.[2] We split B using a plane that contains $'$ and is perpendicular to the back face of B.
   This step is most useful when all rectangles in F_B are fat (recall that a rectangle is fat if its aspect ratio is bounded by a constant). In such a case, we can prove that there are $O(\ )$ candidates for line $'$ and show that this step is used to split only $O(\ )$ boxes that B is recursively partitioned into before one of Steps 3, 4 or 5 is invoked. Thus, we "separate" the long rectangles into distinct boxes without increasing the total number of rectangles by more than a constant factor.
3. If only two classes in F_B are large, we make one cut that does not intersect any rectangle in the two large classes and partitions B into two boxes B_1 and B_2 such that
   (i) either (B_i) 2 (B)=3, for $i = 1; 2$, or
   (ii) there is an i 2 $f1; 2g$ such that (B_i) (B)=3.
4. If only one class in F_B is large, let g be the face of B that contains exactly one of the edges of each rectangle in B. We use a plane that is orthogonal to g to partition B into two boxes B_1 and B_2 so that after all free cuts in B_1 and B_2 are applied (by repeated invocation of Step 1), each of the resulting boxes has measure at most 2 (B)=3.
5. If no class in F_B is large, we split B into two boxes B_1 and B_2 such that (B_i) 2 (B)=3, for $i = 1; 2$.
   The intuition behind Steps 3–5 is that when B contains more short rectangles than long rectangles, we partition B into boxes that contain roughly half the

[2] If there is no such line $'$, we can prove that F_B contains at most two classes of rectangles [1].

number of short rectangles as B (but possibly as many long rectangles as B).

In Steps 2–5, if there are many planes that satisfy the conditions on the cuts, we use the plane that intersects the smallest number of rectangles in S_B. We recursively apply the above steps to the sub-boxes created by partitioning B. Due to lack of space, we defer an explanation of how we compute these cuts to the full version of the paper.

Remark: If all n rectangles in S are fat, we can prove that there are $O(\ )$ candidate lines to consider in Step 2. If we modify Step 2 to partition B using *all* the planes defined by these lines, we can prove that our algorithm constructs a BSP of size $n2^{O(\sqrt{\log n})}$. Furthermore, when m of the rectangles in S are thin, we can further modify our algorithm to construct a BSP of size $n\sqrt{m}2^{O(\sqrt{\log n})}$. The analysis is similar to that of Agarwal et al. [1]. We believe that the size of the BSP constructed by the simpler algorithm New is also $n\sqrt{m}2^{O(\sqrt{\log n})}$. However, we have been unable to prove this claim so far. The experiments we describe in Section 4 show that algorithm New constructs BSPs of linear size in practice.

3 Other Algorithms

In this section, we discuss our implementation of some other techniques presented in the literature for constructing BSPs. Note that some of the algorithms discussed below were originally developed to construct BSPs for arbitrarily-oriented polygons in $\mathbb{R}^3$. All the algorithms work on the same basic principle: To determine which plane to split a box B with, they examine each plane that supports a rectangle in S_B (recall that S_B is the set of rectangles in S clipped within B) and determine how "good" that plane is. They split B using the "best" plane and recurse. Our implementation refines the original descriptions of these algorithms in two respects: (i) At a node B, we first check whether S_B contains a free rectangle; if it does, we apply the free cut containing that rectangle.[3] (ii) If there is more than one "best" plane, we choose the medial plane.[4] To complete the description of each technique, it suffices to describe how it measures how "good" a candidate plane is.

For a plane , let f denote the number of rectangles in S_B intersected by , f^+ the number of rectangles in S_B completely lying in the positive halfspace defined by , and f^- the number of rectangles in S_B lying completely in the negative halfspace defined by . We define the *occlusion factor* to be the ratio of the total area of the rectangles in S_B lying in to the area of (when is clipped within B), the *balance* to be the ratio $\min\{f+f^+; f^-\}/\max\{f+f^+; f^-\}$

[3] Only Paterson and Yao's algorithm [18] originally incorporated the notion of free cuts.

[4] Only Teller's algorithm [21] picked the medial plane; the other algorithms do not specify how to deal with multiple "best" planes.

between the number of polygons that lie completely in each halfspace defined by , and to be the *split factor* of , which is the fraction of rectangles that intersects, i.e., $= f / |S_B|$. We now discuss how each algorithm measures how good a plane is.

ThibaultNaylor: We discuss two of the three heuristics that Thibault and Naylor [22] present (the third performed poorly in our experiments). Below, w is a positive weight that can be changed to tune the performance of the heuristics.

1. Pick a plane the minimizes the function $|f^+ - f^-| + wf$. This measure tries to balance the number of rectangles on each side of so that the height of the BSP is small and also tries to minimize the number of rectangles intersected by .
2. Maximize the measure $f^+ f^- - wf$. This measure is very similar to the previous one, except that it gives more weight to constructing a balanced BSP.

In our experiments, we use $w = 8$, as suggested by Thibault and Naylor [22].

Airey: Airey [3] proposes a measure function that is a linear combination of a plane's occlusion factor, its balance, and its split factor: 0.5 +0.3 +0.2 :

Teller: Let 0 1 be a real number. Teller [21] chooses the plane with the maximum occlusion factor , provided . If there is no such plane, he chooses the plane with the minimum value of f . We use the value $= 0.5$ in our implementation, as suggested by Teller. The intuition behind this algorithm is that planes that are "well-covered" are unlikely to intersect many rectangles and that data sets made up of orthogonal rectangles are likely to contain many coplanar rectangles.

PatersonYao: We have implemented a refined version of the algorithm of Paterson and Yao [18]. For a box B, let s_x (resp., s_y; s_z) denote the number of edges of the rectangles in S_B that lie in the interior of B and are parallel to the x-axis (resp., y-axis, z-axis). We define the measure of B to be $(B) = s_x s_y s_z$. We make a cut that is perpendicular to the smallest family of edges and divides B into two boxes, each with measure at most (B)=4. (Paterson and Yao prove that given any axis, we can find such a cut perpendicular to that axis.) We can show that this algorithm also constructs produces BSPs of size $O(n^{\log \overline{n}})$ for n rectangles, just like Paterson and Yao's original algorithm [18].

Rounds: We briefly describe the algorithm of Agarwal et al. [1]. Their algorithm proceeds in rounds. Each round partitions a box B using a sequence of cuts in two stages, the separating stage and the dividing stage. The separating stage partitions B into a set of boxes C such that for each box $C \in C$, F_C contains only two classes of long rectangles. To effect this partition, they use cuts similar to the cut we make in Step 2 of algorithm New. In the dividing stage, they refine each box $C \in C$ using cuts similar to those made in Steps 3 and 4 of algorithm New until the "weight" of each resulting box is less than the "weight" of B by a certain factor. A new round is executed recursively in each of these boxes. See Agarwal et al. [1] for more details. Below, we refer to their technique as algorithm Rounds.

Our implementations of these algorithms are efficient in terms of running time because we exploit the fact that we are constructing BSPs for orthogonal rectangles. After an initial sort, we determine the cut at any node in time linear in the number of the rectangles intersecting that node. If we were processing arbitrarily-oriented objects, computing a cut can take time quadratic in the number of objects.

We now briefly mention some other known techniques that we have not implemented, since we expect them to have performance similar to the algorithms we have implemented. Naylor has proposed a technique that controls the construction of the BSP by using estimates of the costs incurred when the BSP is used to answer standard queries [14]. While his idea is different from standard techniques used to construct BSPs, the measure functions he uses to choose cutting planes are very similar to the ones used in the algorithms we have implemented. Cassen et al. [5] use genetic algorithms to construct BSPs. We have not compared our algorithms to theirs since they report that their algorithm takes hours to run even for moderately-sized data sets. Note that de Berg's algorithm for constructing BSPs for fat polyhedra [8] cannot be used to solve our problem since rectangles in $\mathbb{R}^3$ are not fat in his model.

4 Experimental Results

We have implemented the above algorithms and run them on the following data sets containing orthogonal rectangles:[5]

1. the `Fifth` floor of Soda Hall containing 1677 rectangles,
2. the `Entire` Soda Hall model with 8690 rectangles,
3. the Orange United Methodist `Church` Fellowship Hall with 29988 rectangles,
4. the Sitterson Hall `Lobby` with 12207 rectangles, and
5. `Sitt`erson Hall containing 6002 rectangles.

We present three sets of results. For each set, we first discuss the experimental set-up and then present the performance of our algorithms. These experiments were run on a Sun SPARCstation 5 running SunOS 5.5.1 with 64MB of RAM.

4.1 Size of the BSP

Recall that we have defined the size of a BSP to be the sum of the number of interior nodes in the BSP and the total number of rectangles stored at all the nodes of the BSP. The total number of rectangles stored in the BSP is the sum of the number of input rectangles and the number of fragments created by the cutting planes in the BSP. The table below displays the size of the BSP and the total number of times the rectangles are fragmented by the cuts made by the BSP.

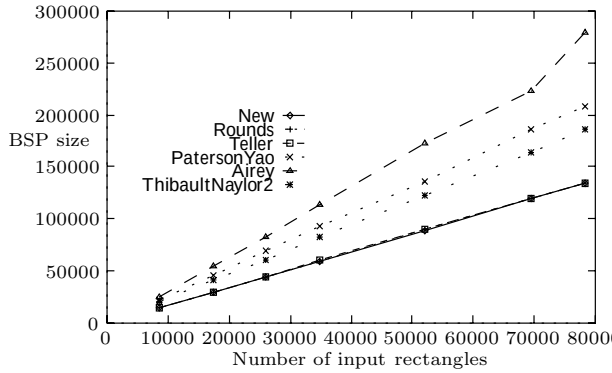
[5] We discarded all non-orthogonal polygons from these data sets. The number of such polygons was very small.

Number of Fragments						Size of the BSP				
Fifth	Entire	Church	Lobby	Sitt.	Datasets	Fifth	Entire	Church	Lobby	Sitt.
1677	8690	29988	12207	6002	#rectangles	1677	8690	29988	12207	6002
89	660	881	681	332	New	2715	14470	45528	22226	8983
113	741	838	475	312	Rounds	2744	14707	45427	22225	9060
301	1458	873	514	153	Teller	2931	14950	33518	13911	7340
449	5545	12517	9642	6428	PatersonYao	3310	22468	56868	30712	20600
675	7001	5494	5350	8307	Airey	3585	24683	41270	21753	19841
1868	10580	13797	3441	1324	ThibaultNaylor1	6092	32929	65313	25051	10836
262	2859	6905	1760	1601	ThibaultNaylor2	3235	20089	58175	23159	12192

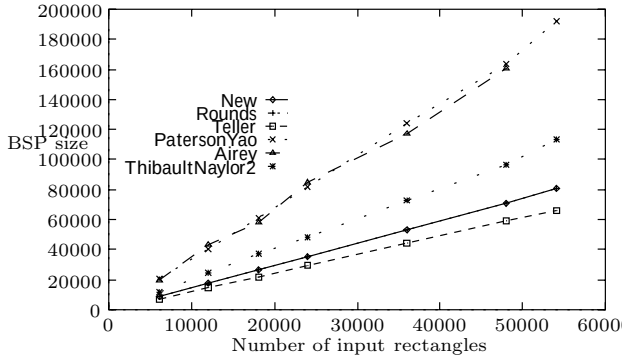
Examining this table, we note that, in general, the number of fragments and size of the BSP scale well with the size of the data set. For the Soda Hall data sets (`Fifth` and `Entire`), algorithm New creates the smallest number of fragments and constructs the smallest BSP. For the other three sets, algorithm Teller performs best in terms of BSP size. However, there are some peculiarities in the table. For example, for the `Church` data set, algorithm Rounds creates a smaller number of fragments than algorithm Teller but constructs a larger BSP. We believe that this difference is explained by the fact that the 29998 rectangles in the `Church` model lie in a total of only 859 distinct planes. Since algorithm Teller makes cuts based on how much of a plane's area is covered by rectangles, it is reasonable to expect that the algorithm will "place" a lot of rectangles in cuts made close to the root of the BSP, thus leading to a BSP with a small number of nodes.

We further examined the issue of how well the performance of the algorithms scaled with the size of the data by running the algorithms on data sets that we "created" by making translated and rotated copies of the original data sets. In Figure 1, we display the results of this experiment for the `Entire` Soda Hall and the `Sitterson` models. We have omitted graphs for the other data sets due to lack of space. In these graphs, we do not display the curve for algorithm ThibaultNaylor1 since its performance is always worse than the performance of algorithm ThibaultNaylor2. The graphs show that the size of the BSP constructed by most algorithms increases linearly with the size of the data. The performance of algorithms New, Rounds, and Teller is nearly identical for the `Entire` data set. However, algorithm Teller constructs a smaller BSP than algorithm New for the `Sitterson` data set. For this data set, note the performance of algorithm New is nearly identical to the performance of algorithm Rounds.

The time taken to construct the BSPs also scaled well with the size of the data sets. Algorithm New took 11 seconds to construct a BSP for the `Fifth` floor of Soda Hall and about 4:5 minutes for the `Church` data set. Typically, algorithm PatersonYao took about 15% less time than algorithm New while the other algorithms (Airey, ThibaultNaylor, and Teller) took 2–4 times as much time as algorithm New to construct a BSP. While the difference in time is negligible for small data sets, it can be considerable for large data sets. For example, for the data set obtained by placing 9 copies of the `Sitterson` model in a 3 3



(a) `Entire`: Algorithms New, Rounds, and Teller have nearly identical graphs.



(b) `Sitterson`: Algorithms New and Rounds have nearly identical graphs.

Fig. 1. Graphs displaying BSP size vs. the number of input rectangles.

array, algorithm New took 13 minutes to construct a BSP while algorithm Teller took 51 minutes.

4.2 Point Location

In the point location query, given a point, we want to locate the leaf of the BSP that contains the point. We answer a query by traversing the path from the root of the BSP that leads to the leaf that contains the query point. The cost of a query is the number of nodes in this path. In our experiments, we create the queries by generating 1000 random points from a uniform distribution over the box B containing all the rectangles in S.

Due to lack of space, we present a summary of the results for point location, concentrating on algorithms New, Rounds, and Teller These results were highly correlated to the height of the trees. Algorithms New and Rounds constructed BSPs of average height between 11 and 16 with the standard deviation of the height ranging from 2 to 2.5. The average cost of locating a point ranged between 10 and 15. The average height of the BSP constructed by algorithm Teller ranged between 15 and 20 with standard deviation ranging from 4 to 5, while the average cost of point location ranged between 7 and 15.

4.3 Ray Shooting

Given a ray , we want to determine the first rectangle in S that is intersected by or report that there is no such rectangle. To answer such a query, we trace through the leaves of the BSP that intersects. At each such leaf v, we check whether the first point where intersects the boundary of v is contained in a rectangle in S (such a rectangle must be stored with the bisecting plane of an ancestor of v or lie on the boundary of B). If so, we output the rectangle and stop the query. Otherwise, we continue tracing . There are two components to the cost of answering the query with : the number of nodes visited and the number of rectangles checked. We report the two factors separately below. The actual cost of a ray shooting query is a linear combination of these two components; its exact form depends on the implementation. In our experiment, we constructed 1000 rays for each data set by generating 1000 random (origin, direction) pairs, where the origin was picked from a uniform distribution over B and the direction was chosen from a uniform distribution over the sphere of directions.

# nodes visited						# rects. checked				
Fifth	Entire	Church	Lobby	Sitt.		Fifth	Entire	Church	Lobby	Sitt.
43.7	10.8	311.5	86.8	56.0	New	5.6	3.7	48.3	2.6	19.8
44.7	12.5	326.4	89.6	55.9	Rounds	5.7	3.0	49.6	2.0	19.2
17.1	13.7	96.6	13.0	37.3	Teller	12.0	11.0	4828.2	20.4	44.1
40.0	11.8	531.4	49.8	83.1	PatersonYao	4.0	5.7	5461.2	84.2	114.0
24.0	13.3	170.2	10.5	129.9	Airey	5.5	4.1	4757.9	11.9	27.5
44.1	31.3	256.8	102.6	69.1	ThibaultNaylor1	4.6	14.6	20.7	2.1	38.5
44.5	14.2	298.5	78.4	59.8	ThibaultNaylor2	5.1	7.5	28.5	2.6	7.3

There is an interesting tradeoff between these two costs, which is most sharply noticeable for the Church data set. Notice that the average number of nodes visited to answer ray shooting queries in the BSP constructed by algorithm Teller is about a third the number visited in the BSP built by algorithm New but the number of rectangles checked for algorithm Teller is about 10 times higher! This apparent discrepancy actually ties in with our earlier conclusion that algorithm Teller is able to construct a BSP with a small number of nodes for the Church model because the rectangles in this model lie in a small number of distinct planes. As a result, we do not visit too many nodes during a ray shooting query. However, when we check whether the intersection of a ray with a node is contained in a rectangle in S, we process a large number of rectangles since each cutting plane contains a large number of rectangles. This cost can be brought

down by using an efficient data structure for point location among rectangles. However, this change will increase the size of the BSP itself. Determining the right combination needs further investigation.

5 Conclusions

Our comparison indicates that algorithms New, Rounds and Teller construct the smallest BSPs for orthogonal rectangles in $\mathbb{R}^3$. Algorithms New and Rounds run 2-4 times faster and construct BSPs with smaller and more uniform height than algorithm Teller.

Algorithm Teller is best for applications like painter's algorithm [9] in which the entire BSP is traversed. On the other hand, for queries such as ray shooting, it might be advisable to use algorithm New or Rounds since they build BSPs whose sizes are not much more than algorithm Teller's BSPs but have better height and query costs. Note that we can prove that algorithms New and Rounds construct BSPs whose height is logarithmic in the number of rectangles in S [1]. Such guarantees are crucial to extending these BSP-construction algorithms to scenarios when the input rectangles move or are inserted into and deleted from the BSP.

Clearly, there is a tradeoff between the amount of time spent on constructing the BSP and the size of the resulting BSP. Our experience suggests that while algorithm Teller constructs the smallest BSPs, algorithms New and Rounds are likely to be fast in terms of execution, will build compact BSPs that answer queries efficiently, and can be efficiently extended to dynamic environments.

Acknowledgments We would like to thank Seth Teller for providing us with the Soda Hall data set created at the Department of Computer Science, University of California at Berkeley. We would also like to thank the Walkthrough Project, Department of Computer Science, University of North Carolina at Chapel Hill for providing us with the data sets for Sitterson Hall, the Orange United Methodist Church Fellowship Hall, and the Sitterson Hall Lobby.

References

1. P. K. Agarwal, E. F. Grove, T. M. Murali, and J. S. Vitter, Binary space partitions for fat rectangles, *Proc. 37th Annu. IEEE Sympos. Found. Comput. Sci.*, October 1996, pp. 482–491.
2. P. K. Agarwal and S. Suri, Surface approximation and geometric partitions, *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, 1994, pp. 24–33.
3. J. M. Airey, *Increasing Update Rates in the Building Walkthrough System with Automatic Model-space Subdivision and Potentially Visible Set Calculations*, Ph.D. Thesis, Dept. of Computer Science, University of North Carolina, Chapel Hill, 1990.
4. A. T. Campbell, *Modeling Global Diffuse Illumination for Image Synthesis*, Ph.D. Thesis, Dept. of Computer Sciences, University of Texas, Austin, 1991.
5. T. Cassen, K. R. Subramanian, and Z. Michalewicz, Near-optimal construction of partitioning trees by evolutionary techniques, *Proc. Graphics Interface '95*, 1995, pp. 263–271.

6. N. Chin and S. Feiner, Near real-time shadow generation using BSP trees, *Proc. SIGGRAPH 89*, *Comput. Graph.*, Vol. 23, ACM SIGGRAPH, 1989, pp. 99–106.
7. N. Chin and S. Feiner, Fast object-precision shadow generation for areal light sources using BSP trees, *Proc. 1992 Sympos. Interactive 3D Graphics*, 1992, pp. 21–30.
8. M. de Berg, Linear size binary space partitions for fat objects, *Proc. 3rd Annu. European Sympos. Algorithms*, *Lecture Notes Comput. Sci.*, Vol. 979, Springer-Verlag, 1995, pp. 252–263.
9. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, Reading, MA, 1990.
10. H. Fuchs, Z. M. Kedem, and B. Naylor, On visible surface generation by a priori tree structures, *Proc. SIGGRAPH 80*, *Comput. Graph.*, Vol. 14, ACM SIGGRAPH, 1980, pp. 124–133.
11. C. Mata and J. S. B. Mitchell, Approximation algorithms for geometric tour and network design problems, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 360–369.
12. T. M. Murali and T. A. Funkhouser, Consistent solid and boundary representations from arbitrary polygonal data, *Proc. 1997 Sympos. Interactive 3D Graphics*, 1997, pp. 155–162.
13. M. J. Muus, Understanding the preparation and analysis of solid models, in: *Techniques for Computer Graphics* (D. F. Rogers and R. A. Earnshaw, eds.), Springer-Verlag, 1987.
14. B. Naylor, Constructing good partitioning trees, *Proc. Graphics Interface '93*, 1993, pp. 181–191.
15. B. Naylor and W. Thibault, Application of BSP trees to ray-tracing and CSG evaluation, Technical Report GIT-ICS 86/03, Georgia Institute of Tech., School of Information and Computer Science, February 1986.
16. B. F. Naylor, J. Amanatides, and W. C. Thibault, Merging BSP trees yields polyhedral set operations, *Proc. SIGGRAPH 90*, *Comput. Graph.*, Vol. 24, ACM SIGGRAPH, 1990, pp. 115–124.
17. M. S. Paterson and F. F. Yao, Efficient binary space partitions for hidden-surface removal and solid modeling, *Discrete Comput. Geom.*, 5 (1990), 485–503.
18. M. S. Paterson and F. F. Yao, Optimal binary space partitions for orthogonal objects, *J. Algorithms*, 13 (1992), 99–113.
19. R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp, Study for applying computer-generated images to visual simulation, Tech. Rep. AFHRL–TR–69–14, U.S. Air Force Human Resources Laboratory, 1969.
20. P. J. Tanenbaum. Applications of computational geometry in army research and development. Invited talk, Second CGC Workshop on Computational Geometry, 1997.
21. S. J. Teller, *Visibility Computations in Densely Occluded Polyhedral Environments*, Ph.D. Thesis, Dept. of Computer Science, University of California, Berkeley, 1992.
22. W. C. Thibault and B. F. Naylor, Set operations on polyhedra using binary space partitioning trees, *Proc. SIGGRAPH 87*, *Comput. Graph.*, Vol. 21, ACM SIGGRAPH, 1987, pp. 153–162.

A Fast Random Greedy Algorithm for the Component Commonality Problem

Ravi Kannan[1] and Andreas Nolte[2?]

[1] Yale University, Department of Computer Science, 51 Prospect Street, New Haven, CT 06520-8285, kannan@cs.yale.edu
[2] Universität zu Köln, Institut für Informatik, Weyertal 80, 50931 Köln
anolte@informatik.uni-koeln.de

Abstract. This paper presents a fast, randomized greedy algorithm to solve the Component Commonality Problem approximately. It is based on a geometric random walk, that starts from a given initial solution and accepts only better points during the walk. We use a new type of analysis, that is not based on conductance, but makes use of structural geometric properties of the problem, namely the smoothness of the set of feasible points.

1 Introduction

The Component Commonality Problem is a stochastic optimization problem in discrete time, that arises as follows. There are m products (indexed by j) with correlated random demands $(d_1, \ldots, d_m)$ with distribution $H(.)$ and density $h(.)$ in any period, that require some or all of n components (indexed by i) in the ratio $u_{ij} \quad 0$. The key feature in this setting is that many of the n components are shared by many products, and as the assembly time is small relative to the procurement time of the components, the assembly is done after the demand comes in, thus taking advantage of the commonality of the components. The Component Commonality Problem, then, is to find the quantities x_i of each component (at unit cost $c_i > 0$) that should be on stock at the beginning of each period so as to minimize total costs of purchase while satisfying the demands with probability at least a given fraction γ. Formally, for a stock level $x = (x_1, \ldots, x_n)$, define $S_x = \{d : \sum_{j=1}^{m} u_{ij} d^j \quad x_j \text{ for all } i\}$ to be the set of demands that can be met with x on stock. Let $F(x) = \int_{S_x} h$ be the probability that we can meet the demands with x on stock. Then the set of feasible stock levels is:

$$P = \{x \in \mathbb{R}^n : \int_{S_x} h \quad \gamma\}$$

and the problem is:

$$\text{Minimize } \sum_i c_i x_i : x \in P$$

? The author was supported by a DAAD-Postdoctoral Fellowship during his visit at Yale University

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 223–234, 1998.
c Springer-Verlag Berlin Heidelberg 1998

We make the general assumption that the density is log-concave. It can be seen, that the feasible set of stock levels P is convex [7].

Our randomized algorithm is based on a very simple random walk. Starting from a current feasible solution x_f, a new point is generated uniformly at random in a ball with center x_f and accepted if it is feasible and has a better objective function value. This is a greedy algorithm in that we seek to improve the objective function at each step. We prove polynomial time bounds under certain mild assumptions on the density function (see next section). This seems to be one of the first provably fast random greedy algorithms to achieve near optimality with high probability in a non-trivial geometric setting.

This problem falls into a class of problems called probabilistic constrained programming, a type of a stochastic program. Traditional efforts to solve this type of problem have been by nonlinear programming methods that are gradient based. These algorithms need to address the following questions

1. check whether a given point is feasible (i.e. $x \in P$).
2. calculate the gradient of F.

While the first point, checking whether a given point is feasible, can be solved quite efficiently by sampling, the second point, the computation of the gradient, turns out to be more difficult. One has to compute each of the n components of the gradient very accurately, so that no provable polynomial time bounds are known for these algorithms. Only asymptotic convergence could be established ([7]). Our algorithm avoids the computation of a gradient by picking a random direction. Therefore, we only have to deal with membership queries, which are much easier to answer.

2 Idea of the Algorithm

As pointed out in the introduction, our algorithm is based on an extremely simple random greedy search . It proceeds as follows. Given an initial feasible solution $x_f \in P$ we generate uniformly at random a point x in a ball with a certain radius around x_f. If x has a better objective function value (i.e. $cx < cx_f$), then we move to x, otherwise we stay at x. The performance might speed up by doing a line search in the direction of the better point x.

If the generated ball with radius r lies completely in P, then we could expect an improvement of $\Omega(\|c\|r/\sqrt{n})$ at every step with constant probability. But the situation becomes worse, if the center of the ball is near the boundary, so that part of the ball is outside of P. In the following we will show that under certain mild assumptions on the density function P is smooth enough, so that not too much of the better part of the ball is outside P, if we are not already near the optimum. Thus, we have an improvement in each step with a certain probability. We will stop the algorithm after a certain number of steps. This number is large enough to ensure being near the optimum with high probability, but still polynomial in m and n.

3 The Algorithm and Time Bounds

The algorithm is described in detail in Figure 1. To prove the smoothness of the set P, we have to put some conditions on the instances.

Assumptions:

We assume that the angle between each pair of row vectors is bounded below by a constant (i.e. $U^{(i)}U^{(j)} \quad 1 - \delta$ for a constant $\delta \; 2 \; \mathbf{R}^+$). It is easy to see that P is not smooth, if this condition is not fulfilled. Furthermore we have to put some condition on the density function h, since obviously a sharply concentrated distribution would cause P to be nonsmooth. The exact conditions are:

- h is also a density function of every (n-1)- and (n-2)-dimensional subspace
- $9c_1, c_2 \; 2 \; \mathbf{R}^+ \quad 8x \; 2 \; \mathbf{R}^m \quad 8y \; 2 \; B(x, c_1) \quad \frac{h(x)}{h(y)} \quad 1 + c_2 k x - y k$ (this is a Lipschitz condition for $\log h$, which guarantees smoothness of h).

But these conditions do not put a severe condition on the density function, since all the most important distribution functions like the exponential and the normal distribution meet these requirements. We assume further, that the demand for each product is bounded by a constant, i.e. $h(y) = 0$, if there exists a i with $y_i \quad c_2$ with a constant c_2. This is reasonable, because the demand for each product will not depend on the number of products or the number of components in any practical applications. Our result can now be stated as follows.

Given an instance of the problem (a membership oracle for P and objective function c, $\epsilon > 0$ and an initial feasible point $x_f \; 2 \; P$) our algorithm succeeds with probability $1 - e^{-\Omega(n)}$ in finding a feasible point $x \; 2 \; P$ with

$$cx \quad (1+\epsilon) \min_{y 2 P} cy$$

(i.e. x is approximately optimal). The running time is bounded above by

$$\min \quad O \quad \frac{(\min_{y 2 P} cy) n^{3:5} m^{0:5}}{\epsilon^2} \quad , O((cx_f) n^{3:5} m^{0:5})$$

calls to the membership oracle.

While we assume that a membership oracle is available, we do not assume that a separation oracle is available. By a theorem of Yudin and Nemirovskii it is known that membership and separation oracles are polynomial time equivalent for convex programming problems like this one ([3]). But the computation of the separating hyperplanes needs a large running time and the approach here is more efficient, if the function value of the initial solution is not too large.

Kannan, Mount and Tayur have described a Simulated Annealing like algorithm for the Component Commonality Problem with polynomial running time, that is based on the ideas of volume computation ([2]). The analysis is based on the estimation of the conductance of a geometrical random walk.

Their algorithm is quite complicated due to the use of a gauge function and a biased Metropolis walk and the analysis is very involved. Our algorithm and the analysis are conceptually much simpler and yield competitive running time

results. But there is another point we would like to stress. In contrast to their algorithm we could use a line search in the direction of the accepted better point without affecting the analysis. This is very likely to speed up the algorithm in practice. There are quite a few references, which report on a very good empirical running times of these so called Hit and Run algorithms (in comparison to traditional gradient based methods) ([7]), but there are only empirical running time bounds for these algorithms for any optimization problem yet.

Our research was motivated by the following observation: Since this problem does not have any local minima by convexity, the acceptance of any worse points (as common in the Simulated Annealing type algorithms) seems not to be necessary. Therefore, the question, whether a simple hillclimbing algorithm would also solve the problem, arises naturally. Unfortunately, the well developed machinery of analyzing geometric random walks does not apply here, since we are not dealing with an ergodic Markov chain any more. Therefore, a new type of analysis had to be developed. The key idea is to make use of the smoothness of the set of feasible points, which enables us to get running time bounds for this hillclimbing algorithm that are competitive with those obtained by conductance arguments. Since these smooth sets also arise naturally in other areas this approach might also be of more general interest.

Procedure RandomGreedy(x_f; , membership oracle for P)
begin
  $x = x_f$;
  $D = \sqrt{2}(1x)$;
  $B_u = 1x$;
  $B_l = 0$;
  Repeat;
    Repeat $\frac{18D^3n^{4.5}m^{0.5}}{B_u - B_l}$ times;
      $r = \frac{B_u - B_l}{3Dn^{3.5}m^{0.5}}$;
      Repeat(basic step)
        Generate a random $x' \in B(x;r)$;
        check whether $x \in P$;
        check whether $cx' < cx$;
      Until $x \in P$ and $cx' < cx$;
      $x = x'$;
      $D = \sqrt{21}x$;
    End(repeat);
    If $1x \quad 2=3B_u + 1=3B_l$ then $B_u = 2=3B_u + 1=3B_l$;
    Else $B_l = 2=3B_l + 1=3B_u$
  Until $B_u - B_l \quad B_l$;
end;

Fig. 1. The algorithm Random Greedy

4 Analysis

The Component Commonality Problem was described in detail in the introduction. If U denotes the matrix of the u_{ij}'s here, let $y = y(d) = Ud$. Under the assumption that the distribution of d $h(.)$ is log-concave, it is easy to see that y has a log-concave density. Let $H(.)$ denote the density of y and let μ_H denote the corresponding measure (so for every measurable set S $\mu_H(S) = \int_S H$). Then, the set P of feasible points of stochastic levels can be expressed as $P = \{x \in \mathbf{R}^n | \mu_H(dom(x)) \geq \gamma\}$, where $dom(x) = \{y | y \geq x\}$. It is easy to show that P is convex ([7]).

Before we start to analyze the algorithm, we want to introduce some notations that we use throughout the analysis.

Due to the unboundedness of P in every positive direction we might assume that the cost vector c has all strictly positive components. After rescaling, if necessary, we may also assume that $c = \mathbf{1}$, the vector of 1's ([5]). To simplify the notation we may further assume that the row vectors of the matrix U are unit vectors.

Let $x \in P$ be a feasible point. Initially we can assume that x is identical to the given feasible point x_f. Let $D = \sqrt{n}(\mathbf{1}x_f)$. Using $P \subseteq \mathbf{R}^n_+$ it is easy to see that D is an upper bound on the L_2 distance between x_f and the optimal solution. Let $\Delta_x = \frac{\mathbf{1}x - \min_{y \in P} \mathbf{1}y}{\sqrt{n}}$ In the rest of the paper we are concerned with the proof of the following main theorem.

Theorem 1. *Let $x_f \in P$ be an initial feasible solution. The algorithm Random Greedy will need at most*

$$\min \quad O \quad \frac{(\min_{y \in P} cy)n^{3.5}m^{0.5}}{\epsilon^2} \quad , O((cx_f)n^{3.5}m^{0.5})$$

calls to the membership oracle of P to get an approximate solution $x \in P$ s.t. $cx \leq (1+\epsilon)\min_{y \in P} cy$ with probability at least $1 - e^{-\Omega(n)}$.

□

4.1 Analysis of one basic step

As described in Figure 1 the basic step of our greedy algorithm is to pick a random point in a ball $B(x, r)$ with center x and radius r. This section is concerned with the analysis of this basic step of the algorithm. The next section will give a bound on the overall performance of the algorithm by carefully choosing the radii of the balls according to the progress made in previous steps.

As a first step in the analysis of the algorithm we look for a lower bound on the angle between the tangent hyperplane and the isohyperplane (same values of the objective function) at an arbitrary point x. This angle gives us a rough estimate of the probability of picking a better point, if P is smooth enough and the radius r is not too large.

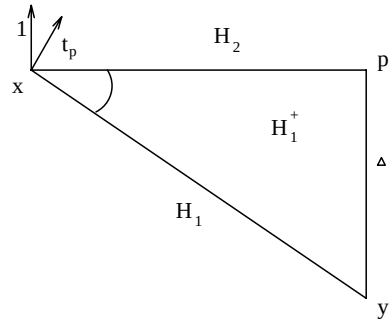


Fig. 2. The angle between the tangent and the isohyperplane

Lemma 1. *Let H_1 be the tangent hyperplane of P at x with normal vector t and H_2 be the hyperplane with normal vector $1/\sqrt{n}$ and $x \in H_2$. This implies that $\alpha \geq \cos \frac{\Delta}{D}$.*

Proof. According to the definition of Δ_x the following is true $\exists p \in H_2, H_1^+ \quad y = p - \Delta 1/\sqrt{n} \in H_1^+$, where H_1^+ denotes the halfspace that contains P. Let t_p be the projection of t in the 2 dimensional plane generated by x, y and p. Obviously $\|t_p\| \leq \|t\|_2$ and $t_p 1 = t1$. It follows $\frac{t_p 1}{\sqrt{n}\|t_p\|_2} \geq \frac{t_p 1}{\sqrt{n}\|t\|_2} = \frac{t1}{\sqrt{n}\|t\|_2} = \cos\alpha \quad (say)$. Therefore it is sufficient to look for an upper bound of $\cos\alpha' = \frac{t_p 1}{\sqrt{n}\|t_p\|_2}$. The Pythagorean theorem implies

$$\cos\alpha' \leq \sqrt{1 - \frac{\Delta^2}{D^2}} \leq 1 - \frac{\Delta^2}{2D^2}.$$

Because of the Taylor series expansion of the cos we get $\cos(\Delta/D) \geq 1 - \frac{\Delta^2}{2D^2}$. This implies $\cos\alpha' \geq \cos(\Delta/D)$ and $\alpha \geq \alpha' \geq \frac{\Delta}{D}$. □

Roughly speaking, Lemma 1 implies that, if we are quite far away from the optimum in terms of the objective function, but not too far away in terms of L_2 distance, we can expect a quite large angle between the tangent hyperplane and the isohyperplane. This will give us, as we will see later, a large probability of improving the objective function.

As a next step we are concerned with the fact that P is sufficiently smooth, i.e. the border of P is not too far away from the tangent hyperplane in a small ball with center x. We define for $x \in \mathrm{R}^n$: $F(x) = \int_{S_x} h$ with $S_x = \{y \in \mathrm{R}^n | Uy \leq x\}$ (set of feasible production vectors). This implies $P = \{x \in \mathrm{R}^n | F(x) \geq \gamma\}$.

Proposition 1. *Suppose $x \in P$ with $F(x) \geq \gamma + (1-\gamma)/2 < 1$ and r is a positive real number. Let $z \in \mathrm{R}^n$ be a point with $\|x - z\|_2 \leq r, r \in \mathrm{R}^+$ and $rF(x)(z - x) > 0$. Then there exist constants $c_1, c_2 \in \mathrm{R}^+$, s.t.*

$$F(z) \geq F(x) + \frac{c_1 \kappa}{mn} - c_2 r^2 n^2$$

with $\kappa = \frac{rF(x)}{k\,rF(x)k_2}(z - x)$.

□

This proposition states that if we go from the border of P a little bit inside P (along the gradient) then we have some freedom to go perpendicular to the gradient and stay still inside of P.

The proof of this proposition requires a sequence of lemmata. Crucially is the differentiability of F. But, using standard techniques from analysis ([7]), observing the fact that the density function h is differentiable and the normalvectors of each pair of hyperplanes are not identical, it is easy to see that $F(x) = \int_{S_x} h$ is differentiable. Next we prove a lower bound on the length of the gradient.

Lemma 2. $k\,rF(x)k_2 \quad \frac{c}{nm}$ *for a certain constant* $c \in \mathrm{R}^+$.

Proof. The log-concavity implies the log-concavity of F ([7]). Because of the assumption, that there is a constant bound for each product, it is easy to see that we have $F(x\ ) = 1$ for $x_i = \Omega(\frac{1}{n})$ for all i. Let $F(x) = \gamma^0 < 1$ and we get $x_i \quad c\frac{1}{m}$ and $kx - x\ k \quad c\frac{1}{nm}$. with a constant $c \in \mathrm{R}^+$. By the log-concavity we have that

$$\begin{aligned} \log F(x\ ) \quad & \log F(x) + r(\log F(x))(x - x\ ) \\ & \log F(x) + \frac{k\,rF(x)k}{F(x)}kx - x\ k. \end{aligned}$$

So $k\,rF(x)k \quad \frac{-\log F(x)F(x)}{\frac{1}{nm}}$ and the lemma follows. □

Now, we want to get an upper bound on the norm of the Hessian of F. F is almost everywhere twice differentiable and we want to consider $kHF(x)k_2 = \max_{kzk;kyk=1} kz^T HF(x)yk$ with $x \in P$.

Lemma 3. $kHF(x)k_2 = O(n^1)$

Proof. The proof is technical and deferred to the Appendix. □

With the help of the last two lemmata we can prove Proposition 1.

Proof of Proposition 1: Let x be in P with $F(x) = \gamma^0 < 1$, $z \in P \quad \mathrm{R}^n$ with $kx - zk_2 \quad r, r \in \mathrm{R}^+$ and $rF(x)(z - x) > 0$. We consider the function $G(\lambda) = F(x + \lambda(y - x))$ for $\lambda \in [0, 1]$. Obviously $G(0) = x$ and $G(1) = y$. Applying the second order Taylor theorem we get

$$G(1) = G(0) + G^0(0) + 1/2G^{00}(\zeta) \quad \text{for} \quad \zeta \in [0, 1]. \tag{1}$$

We are a little bit sloppy by ignoring the fact that F might not be twice differentiable everywhere. But this happens only at a set with measure 0 and we could use an arbitrary good approximate instead. We obtain with Lemma 2:

$$\begin{aligned} G^0(0) &= rF(x)(z - x) \\ &= \kappa k\,rF(x)k_2 \\ &\kappa\frac{c}{nm} \end{aligned}$$

with $\kappa = \frac{rF(x)}{k\,rF(x)k_2}(z - x) > 0$. Furthermore, with Lemma 3

$$jG^{00}(\zeta)j = j(z - x)HF(x + \zeta(z - x))(z - x)j \quad r^2 n.$$

By plugging this into (1) we get the result. ∎

Now we can proceed by analyzing the ball walk. First, we try to get a lower bound on the probability to get a feasible point in P by randomly picking a point in a ball $B(x, r)$ with center x and radius r.

Lemma 4. *Let α be as in Lemma 1, $r = \frac{}{n^2 p\overline{m}}$ and*

$$B_{nf} = fy \in \mathrm{R}^n jky - xk \quad r \quad \text{and} \quad F(y) \quad F(x)g.$$

Then vol $B_{nf}(x, r)$ $\frac{1}{4}\frac{}{2}$vol $B(x, r)$.

Proof. We can assume $F(x) \quad \gamma + (1 - \gamma)/2 < 1$, since otherwise $B(x, r) \quad P$, because $krF(x)k \quad n$. Let $\kappa = \frac{c_p r}{2\overline{n}}$ with a certain constant c to be determined later. The situation at point x can be described as follows (see figure 3).

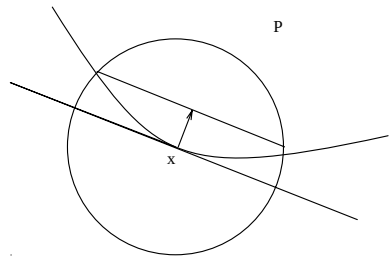


Fig. 3. $B_{nf}(x, r)$

The tangent hyperplane cuts the generated ball in half. According to Proposition 1 the volume of the set $B_{nf}(x, r)$ of points y that are on the right side of the hyperplane and $F(y) < F(x)$ (i.e. $y \notin P$) is bounded above by

$$\mathrm{vol}_{n-1}B(x, r)\kappa = \frac{r^{n-1}\tilde{c}^{n-1}c\alpha r}{(n-1)^{\frac{n-1}{2}}2\overline{n}} \quad \frac{1}{4}\frac{\alpha}{2\pi}\mathrm{vol}_n B(x, r)$$

for $c = \frac{2\tilde{c}e^{-\frac{1}{2}}}{2}$, where $\tilde{c}$ is a certain, ball specific constant. ∎

We consider now the tangent hyperplane at x and the isohyperplane $fyj1y = 1xg$. According to Proposition 1 we know that there is at least an angle of $\alpha \quad \frac{}{D}$ between the corresponding normal vectors. We need the following obvious lemma.

Lemma 5. *Let $B_b = fy \in \mathrm{R}^n j\, rF(x)y \quad rF(x)x$ and $1y \quad 1xg \setminus B(x, r)$ be the set of better points in the ball, that are on the right side of the tangent hyperplane. It follows vol B_b $\frac{}{2}$vol $B(x, r)$.*

∎

In the following we want to estimate how much progress we could expect in each step.

Lemma 6. *Let* $B_p = \{ y \in \mathbb{R}^n : 1x \quad 1y \quad 1x - \Theta \quad p\frac{r}{n} \quad p\frac{}{n} \}$. *Then we get vol* B_p $\frac{1}{4}\frac{}{2}$ *vol* $B(x,r)$.

Proof. Analogous to the proof of Lemma 4. □

As a corollary of the last lemma and Lemma 4 we get a lower bound on the expected step size.

Corollary 1. *The volume of the set* $B(x,r) \setminus P \setminus \{ 1y \quad 1x - \Theta \quad p\frac{r}{n} \quad p\frac{}{n} \}$ *is at least* $\Omega(\alpha)$ *vol* $B(x,r)$.

As a result we can summarize

Proposition 2. *Let* $x \in P$. *If* $D = p\frac{}{2}1x,\ 1x \quad \min_{y \in P} 1y + \Delta p\frac{}{n}, \Delta \in \mathbb{R}^+$ *and* $r = \frac{}{Dn^2 p\frac{}{m}}$, *then the process of choosing a random point in* $B(x,r)$ *hits a point* $y \in P$ *with* $1y \quad 1x - \Theta(\alpha r)$ *with probability of at least* $\Omega(\alpha) = \Omega(\frac{}{D})$. □

Thus, we have an analysis of one basic step of the algorithm.

4.2 Stages

We will perform the algorithm in stages, taking care of the dependence of the radius on α. To motivate the procedure we will describe the first step. Let x_f be the initially given feasible solution and $B_0^u = 1x_f$ be an upper and B_0^l be a lower bound on the optimal solution (e.g. $B_0^l = 0$). Define

$$B^l = 2/3B_0^l + 1/3B_0^u \quad \text{and} \quad B^u = 2/3B_0^u + 1/3B_0^l.$$

Let us assume for the moment $\min_{y \in P} 1y \quad B^l$. Then we obtain

Lemma 7. *By performing*

$$\frac{18D^3 n^{3.5} m^{0.5}}{(B_0^u - B_0^l)^2}$$

steps of the ball walk with radius $r = \frac{B_0^u - B_0^l}{3Dn^{2.5}m^{0.5}}$ *we will reach a* $y \in P$ *with* $1y \quad B^u$ *(i.e. we make a progress of* $1/3(B_0^u - B_0^l)$ *in terms of the objective function) with probability* $1 - e^{-\Omega(n)}$.

Proof. We know $\min_{y \in P} 1y \quad 2/3B_0^l + 1/3B_0^u$. For x with $1x \quad B^u$ we get:

$$1x \quad 2/3B_0^l + 1/3B_0^u + 1/3(B_0^u - B_0^l) \quad \min_{y \in P} 1y + \Delta p\frac{}{n}$$

with $\Delta = \frac{1=3(B_0^u - B_0^l)}{p\frac{}{n}}$. According to Proposition 2 we make a progress of $\Omega(\frac{}{D}r)$ with probability $\Omega(\alpha) = \Omega(\frac{}{D})$ for all steps between B_0^u and B^u by choosing $r = \frac{}{Dn^2 p\frac{}{m}}$. Applying Chernoff's inequality [6] we get an improvement of $\Omega(\frac{}{D}r)$

in at least $\frac{\sqrt{n}D}{r}$ steps with probability $1 - e^{-\Omega(n)}$, while performing $2^{\frac{\sqrt{n}D^2}{r}}$ steps. This implies that we need at most $2^{\frac{\sqrt{n}D^2}{r}} = 2^{\frac{D^3 n^{2.5} m^{0.5}}{2}}$ steps to get $\frac{\sqrt{n}D}{r}\frac{1}{D}r = \Delta\sqrt{n} = 1/3(B_0^u - B_0^l)$ improvement, i.e. to reach B^u with high probability. □

If we have reached B^u after the number of steps mentioned in the last lemma, we set $B_1^u = B^u$ and $B_1^l = B_0^l$, update $D = \sqrt{2}(1x)$ and iterate. We can be sure that we have reduced the gap $B_0^u - B_0^l$ by a factor of $2/3$.

If we have not reached B^u within these steps, we can assume with high probability that $\min_{y \in P} 1y \geq B^l$, and we can set $B_1^l = B^l$ and $B_1^u = B_0^u$, reducing the gap also by a factor of $2/3$.

As a first step for the final analysis we analyze the time necessary to get a solution x being within a constant factor of the optimal solution.

Lemma 8. *Let x_f be the initial given solution, $D_0 = \sqrt{2}(1x_f)$ and assume $D_0 > 2^{\sqrt{2}}\min_{y \in P} 1y$. Then the time necessary to reach a D_k with $D_k < 2^{\sqrt{2}}\min_{y \in P} 1y$ with high probability is bounded by $O(D_0 n^{3.5} m^{0.5})$.*

Proof. The result follows by an easy calculation from the last lemma. □

Now, we assume $D < 2^{\sqrt{2}}\min_{y \in P} 1y$ and try to analyze the time necessary to get a relative error of ϵ.

Lemma 9. *Let x be a feasible solution and $D = \sqrt{2}(1x) \geq 2^{\sqrt{2}}\min_{y \in P} 1y$. The time necessary to reach a $y \in P$ with*

$$1y \leq (1+\epsilon)\min_{y \in P} 1y$$

is bounded above by

$$O\left(\frac{(\min_{y \in P} 1y) n^{3.5} m^{0.5}}{\epsilon^2}\right).$$

Proof. The proof is analogous to the proof of the last lemmata and omitted from this abstract. □

This completes the running time analysis and the proof of Theorem 1.

References

1. Dyer, M.A.; Frieze, A.:*Computing the volume of convex bodies: A case where randomness provably helps*, Proceedings of the Symposium on Applied Math, 1991
2. Dyer, M.A.; Frieze, A.; Kannan, R.:*A random polynomial time algorithm for approximating the volume of convex bodies*, Journal of the ACM 38, 1991
3. Grötschel, M.; Lovasz, L.; Schrijver, A.:*Geometric Algorithms and Combinatorial Optimization*, Springer Verlag, New York, 1988
4. Jayaraman, J.;Srinivasan, J.; Roundy, R.; Tayur, S.:*Procurement of common components in a stochastic environment*, IBM Technical Report, 1992
5. Kannan, R.; Mount, J.; Tayur, S.: *A randomized Algorithm to optimize over certain convex sets*, Mathematics of Operations Research 20, 1995
6. Motwani, R., Raghavan, P.:*Randomized Algorithms*, Cambridge University Press, 1995
7. Wets, R.: *Stochastic Programming* in Handbooks of Operations Research and Management Science, Vol. 1, North Holland, 1989

5 Appendix

Proof of Lemma 3:

Let $i, 1 \quad i \quad n$ be fixed. We consider $\frac{@^2F}{@^2x_i}$ first. Let $FS_i = \{y \in S_x | U^{(i)}y = x_i\} \quad \mathbb{R}^n$. Suppose this hyperplane is a facet of S_x (otherwise the second partial derivative would vanish). Let $\delta \in \mathbb{R}^+$ and $x = x - \delta e_i$ (e_i is the i-th unit vector). It is easy to see $\frac{@^2F}{@^2x_i} = \lim_{\delta \to 0} \frac{h - h}{\delta}$ (see figure 5). Let us assume

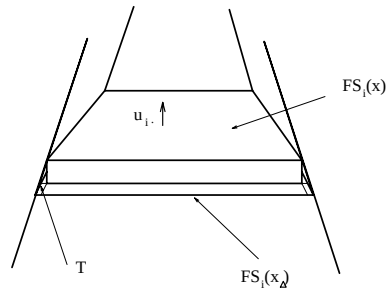


Fig. 4. The second partial derivative $\frac{@^2F}{@^2x_i}$

for the moment that $FS_i \setminus FS_j(x)$ for $i \neq j$ is either empty or a $n-2$ dimensional object. Consider $g(y) = y - \delta U^{(i)}$ for $y \in \mathbb{R}^n$. Obviously (all rows $U^{(j)}$ are non-negative) $g(FS_i(x)) \quad FS_i(x)$ (g is just a translation along the unit vector $U^{(i)}$). It follows by the transformation theorem $\int_{g(FS_i(x))} h = \int_{FS_i(x)} h(g)$. By the Lipschitz-assumption on the density function we get

$$\lim_{\delta \to 0} \frac{\left| \int_{FS_i(x)} h - \int_{g(FS_i(x))} h \right|}{\delta} \quad \lim_{\delta \to 0} \int_{FS_i(x)} c_2 h = c_2$$

because h is also a density function with constant integral over every (n-1)-dimensional subspace. Thus, we have dealt with the central part of $\int_{FS_i(x)} h$. Now we look at the border of $FS_i(x)$ (see Figure 5). Let f_b be one $(n-2)$-dimensional 'facet' of $FS_i(x)$, i.e. the intersection of $FS_i(x)$ and one hyperplane $\{y \in \mathbb{R}^n | U^{(j)}y = x_j\}$. Furthermore, let $\tilde{f}$ be $g(f_b)$, i.e. the projection of f_b onto $FS_i(x)$. Let $d \in [U^{(i)}, U^{(j)}]$ (linear hull), d perpendicular to $U^{(i)}$, $dU^{(j)} > 0$ and $\|d\| = 1$. Then we can define a set D by

$D = \{\tilde{f}_b + \mathbb{R}_0^+ d\} \setminus \{y | U^{(j)}y \quad x_j\}$, which is obviously part of $FS_i(x)$. Analyzing D more carefully we obtain $D = \{\tilde{f}_b + [0, \delta/tan\alpha]d\}$ with $U^{(i)}U^{(j)} = \cos\alpha$. Let $g_t(y) = y + td$ for $y \in \mathbb{R}^n$. We get with Fubini

$$\int_D h = \int_0^{=\tan} \int_{g_t(\tilde{f}_b)} h \quad dt \quad \int_0^{=\tan} c_2 = \frac{c_2\delta}{\tan\alpha}$$

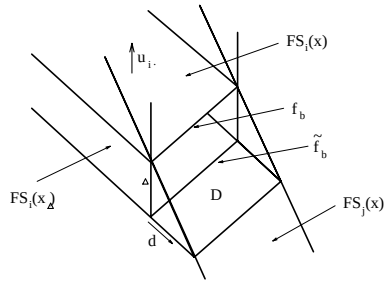


Fig. 5. The border case

for a constant $c_2 \in \mathbb{R}^+$, because h is a density function for every $(n-2)$-dimensional subspace. Since α is bounded below by a constant, the above calculation yields that $\lim_{\delta \to 0} \frac{\int_D h}{\delta}$ is also bounded by a constant. To get a bound on $\frac{\partial^2 F}{\partial^2 x_i}$, we observe, that there are at most n borders like f_b. It is easy to see that the integral of h over the 'corner squares' T (see Figure 5) is a second order effect vanishing by taking the limit $\delta \to 0$. Furthermore, we can drop the assumption that $FS_i \setminus FS_j(x)$ for $i \neq j$ is either empty or a $n-2$ dimensional object. If we had some additional hyperplane touching $FS_i(x)$, $FS_i(x)$ could only become smaller. This is not relevant for us, because we are only interested in an upper bound of the second derivative. Thus, we obtain

$$\frac{\partial^2 F}{\partial^2 x_i} = \lim_{\delta \to 0} \frac{\int_{FS_i(x)} h - \int_{FS_i(x)} h}{\delta} \qquad O(n) + c_2 = O(n).$$

One shows analogously for $j \neq i$: $\frac{\partial^2 F}{\partial x_i \partial x_j} \qquad c_3$ for a certain constant c_3, because in this case we only have to worry about one border of $FS_i(x)$ (the one belonging to the hyperplane $\{y \in \mathbb{R}^n \mid U^{(j)} y = x_j\}$) and the central part is missing here. We can summarize

$$\|HF(x)\|_2 \qquad \max_{\|z\|=1;\|y\|=1} \Big| \sum_i z_i HF_{ii} y_i \Big| + \sum_{i \neq j} |z_i HF(x)_{ij} y_j| = O(n).$$

□

Maximizing Job Completions Online

Bala Kalyanasundaram and Kirk Pruhs

Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA. 15260 USA
{kalyan, kirk}@cs.pitt.edu
http://www.cs.pitt.edu/~{kalyan,kirk}

Abstract. We consider the problem of maximizing the number of jobs completed by their deadline in an online single processor system where the jobs are preemptable and have release times. So in the standard three field scheduling notation, this is the online version of the problem $1 \mid r_i; pmtn \mid \sum (1 - U_i)$. We give a constant competitive randomized algorithm for this problem. It is known that no constant competitive deterministic algorithm exists for this problem. This is the first time that this phenomenon, the randomized competitive ratio is constant in spite of the fact that the deterministic competitive ratio is nonconstant, has been demonstrated to occur in a natural online problem. This result is also a first step toward determining how an online scheduler can use additional processors in a real-time setting to achieve competitiveness.

1 Introduction

We consider the problem of maximizing the number of jobs completed by their deadline in an online single processor system where the jobs are preemptable and have release times. So in the standard three field scheduling notation [3], this is the online version of the problem $1 \mid r_i; pmtn \mid \sum (1 - U_i)$. We present a deterministic algorithm LAX, and show that for every instance I, it is the case that either LAX, or the well-known deterministic algorithm SRPT (Shortest Remaining Processing Time), is constant competitive on I.

The first consequence of this result is that one can then easily obtain a constant competitive randomized online algorithm by running each of SRPT and LAX with equal probability. It is known that no constant competitive deterministic algorithm exists for this problem [1]. This is the first time that this phenomenon, the randomized competitive ratio is constant in spite of the fact that the deterministic competitive ratio is nonconstant, has been demonstrated to occur in an unarguably natural online problem. Previously, the most natural problems that demonstrably exhibited this phenomenon were various scheduling problems where the input was restricted in various ways, e.g. assuming that jobs have only two possible lengths.

The second consequence of this result is that it is possible for an online deterministic scheduler, equipped with two unit speed processors, to be constant competitive with an adversary equipped with one unit speed processor. For a

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 235–246, 1998.
© Springer-Verlag Berlin Heidelberg 1998

discussion of the usefulness of resource augmentation analysis in online scheduling problems see [6] and [8]. Broadly speaking, [6] and [8] show that an online scheduler, equipped with either faster processors or more processors, can be constant competitive with respect to flow time, and that an online scheduler, equipped with faster processors, can be constant competitive in various real-time scheduling problems. The obvious unanswered question inherent in these papers is, "How can an online scheduler make use of extra processors in real-time problems to achieve competitiveness?" The results in this paper are a first step toward answering this question.

1.1 Problem De nition

We consider the online version of the problem $1 \mid r_i; pmtn \mid \sum (1 - U_i)$. An instance I consists of a collection $J_1; : : : J_n$ of jobs. Each job J_i has a *release time* r_i, a *length* or execution time x_i, and a deadline d_i. The online scheduler is unaware of J_i until time r_i, at which time the scheduler additionally learns x_i, and d_i. Thus the scheduler may schedule J_i on a single processor any time after time r_i. The system is *preemptive*, that is, the processor may instantaneously abandon a job J_i, and later restart J_i from the point of suspension. (In a non-preemptive system, the scheduler must run a job to completion once it has started that job.) The objective function is the total number jobs completed before their deadlines, and our goal is to maximize this objective function.

In the context of this problem the competitive ratio of a randomized online algorithm A on an instance I is $\frac{jOPT(I)j}{E[jA(I)j]}$, where $OPT(I)$ is the optimal offline schedule, $A(I)$ is the schedule constructed by A on input I, and jSj is the number of jobs completed by their deadline in the schedule S. The *competitive ratio* of a randomized algorithm for a problem is is the supremum over all instances I of the competitive ratio for A on I. The competitive ratio can be viewed as the payoff to a game played between two players, the online player and the adversary. The adversary specifies the input and services that input optimally. We assume an *oblivious adversary*, that is, the adversary may not modify I in response to the outcome of a random event in A. For background information on online algorithms see [5].

1.2 Related Results

We start with results on the online version of $1 \mid r_i; pmtn \mid \sum (1 - U_i)$. Every deterministic algorithm for this problem has a competitive ratio of $\Omega(\frac{\log}{\log \log})$, where is the ratio of the length of the longest job to the length of the shortest job [1]. The algorithm SRPT is (log) competitive [4]. Constant competitive deterministic algorithms for special instances (e.g. equal job lengths or monotone deadlines) can also be found in [1]. If all the jobs can be completed by their deadline then the EDF (Earliest Deadline First) algorithm will produce an optimal schedule [3]. In [6] it is shown that if the online scheduler is given a faster processor than the adversary, then there is a relatively simple algorithm

that is constant competitive. Further this result holds even for the more general problem $1 \mid r_i; pmtn \mid \sum (1 - U_i)w_i$, where the the jobs have associated positive benefits, and the goal is to maximize the aggregate benefit of the jobs completed by their deadline. There is no constant competitive deterministic or randomized algorithm for $1 \mid r_i; pmtn \mid \sum (1 - U_i)w_i$ [7].

We now survey results on problems that are one "change" away from the problem that we consider. For a recent general survey of online scheduling see [9].

We first consider problems where one changes the objective function. If the objective function is to minimize the number of jobs that miss their deadline, then there is no constant competitive randomized online algorithm [4]. If the objective function is to maximize processor utilization (the fraction of time that the processor is working on a job that it will complete by its deadline), then there is a 4-competitive deterministic algorithm, and this is optimal for deterministic online algorithms [2,10]. It is well known that the algorithm SRPT minimizes the total flow time, which is the sum over all jobs of the completion time minus the release time of that job.

If one changes the job environment to disallow preemption then it is easy to see that no constant competitive randomized algorithm exists.

In [8] several results are presented in the case that the machine environment has multiple processors. In particular, they show that the algorithm that runs the job with least laxity first, and the algorithm that runs the job with the earliest deadline first, will complete all the jobs by their deadline, on instances where the adversary completes all the jobs, if they are equipped with a processor twice as fast as the adversary's processor.

The offline version of of $1 \mid r_i; pmtn \mid \sum (1 - U_i)$ can be solved in polynomial time using a dynamic programming algorithm [3].

2 Algorithm Descriptions

Among all jobs that can be completed by their deadline, SRPT is always running the job that it can complete first.

LAX maintains a stack H of jobs. At any particular time we will denote the jobs in H from bottom to top as $J_{h(1)}, \ldots, J_{h(k)}$. LAX is always running the job $J_{h(k)}$ that is at the top of H.

Definition 1. { *The* laxity ℓ_i *of a job* J_i *is* $d_i - r_i - x_i$.
{ *The* value v_i *of a job* J_i *is the minimum of* ℓ_i *and* x_i.
{ *Let* 16 *be some constant.*
{ *Let* $x_i(t)$ *be the length of* J_j *not executed by* LAX *before time* t.
{ *A job* J_i *is* viable *at time* t *if* $r_i \quad t$ *and* $d_i - t - x_i(t) \quad \ell_i = 2$.
{ *Let* $V(H)$ *be the collection of jobs* J_i
that are not yet completed by LAX,
that are currently viable,
that was not previously popped in line (4) of LAX,
that are not in H, *and*

that satisfy $x_i \quad v_{h(k)}$.

There are two type of events that will cause LAX to take action, the release of a job, and completion of the top job in H. The algorithms for both of these events use the following procedure FILL.

```
          While V(H) is not empty Do
(1)             Select the job J_j 2 V(t) with maximum value
(2)             Push J_j onto H // Note this updates V(H)
```

LAX responds to the release of a job J_i at time r_i in the following manner.

```
          If v_{h(k)}    x_i Then
(3)             Push J_i onto H
          Else If v_{h(k-1)}    x_i and v_i > v_{h(k)} Then
(4)             Pop H // Note this updates V(H)
(5)             Select the job J_j 2 V(t) with maximum value
(6)             Push J_j onto H // Note this updates V(H)
(7)             Call FILL
```

LAX responds to the completion of the top job $J_{h(k)}$ in H at time t in the following manner.

```
(8)       Pop H
          While t + x_{h(k)}(t) > d_{h(k)} Do
(9)             Pop H
(10)      Call FILL
```

Observation 1. *At all times it is the case that* $v_{h(i)} \quad x_{h(i+1)}$, *and hence,* $v_{h(i)} \quad v_{h(i+1)}$, *for* $1 \quad i \quad k-1$.

Observation 2. *Let* P *be a contiguous period of time during which* LAX *does not execute a push in either line (2) or line (3) of its code. Then the value of* $v_{h(k)}$ *is monotonically nondecreasing over* P.

Observation 3. *At no time does there exists a* $J_i \; 2 \; V(H)$ *such that* $x_i \; v_{h(k)}$.

3 Algorithm Analysis

We prove that for all instances I, $j\text{LAX}(I)j+j\text{SRPT}(I)j = \Omega(j\text{OPT}(I)j)$. Throughout this section, I will denote a generic instance of this problem.

3.1 Structure of the Optimal Schedule

To facilitate our later bounding of the competitive ratio, we show in this subsection how to transform an optimal schedule to a near optimal schedule with nice combinatorial features.

De nition 2. { *We say a schedule* S completes *a job* J_i *if* J_i *is run to completion before its deadline.*

{ *A job* J_i *completed in a schedule* S *is* idle *if the processor is idle for some non-zero amount of time between when* J_i *is first run and when* J_i *is completed.*

{ *We say a schedule is* efficient *if, for every job* J_i *run in* S, S *completes* J_i, *and* J_i *is not idle.*

{ *A schedule* S *is a* forest *if whenever a job* J_i *preempts another job* J_j *then* J_i *will be completed before* J_j *is run again.*

{ *Let* $\gamma = \frac{(\quad -3)}{(5\ ^2-5\ )}$, *and* $\quad = \frac{32}{\gamma}$.

{ *For each job* J_i, *let* $\mathrm{Push}(i)$ *be the time that* J_i *was pushed on* H, *and* $\mathrm{Pop}(i)$ *be the time that* J_i *was popped off* H.

{ *We define the* pseudo-release time, *denoted* pr_i, *of a job* J_i. *If* J_i *is in* H *at some time then then* $pr_i = \mathrm{Push}(i)$. *Else let* t *be the earliest time, after time* r_i, *such that there exists an* $\quad > 0$ *such* $\quad x_{h(k)} > v_i$ *during the interval* $(t; t+\ )$. *If* $t \quad r_i + \text{`}_i/2$ *then* $pr_i = t$, *else* $pr_i = +1$.

{ *Let* J *be the collection of jobs in* I *with finite pseudo-release times.*

If a schedule S is a forest, then we think of it as a forest, in the graph theoretic sense, where the descendents of a job $J_i \ 2\ S$ are those jobs that are run after J_i is first run for a non-zero amount of time, but before J_i is completed. We then adopt standard graph theoretic terminology, i.e. parent, child, etc. We number the children of a job in S chronologically by time of first execution, so for example, the first child of a job J_j in S is the first job to preempt J_j.

Lemma 1. *For every schedule* T *there is another schedule* S *such that* S *is an efficient forest, and* S *completes exactly the same jobs as* T.

Proof. This is a direct consequence of the optimality of EDF on instances where all jobs can be completed by their deadline. ∎

Lemma 2. *For every forest schedule* T, *there is another schedule* S *such that*

1. S *is an efficient forest,*
2. *for all first children* J_i *in* S *it is the case that* J_i *is first executed at time* r_i *in* S,
3. *if* J_i *is a descendant of* J_j *in* S *then* J_i *is a descendant of* J_j *in* T, *and*
4. *the jobs completed in* S *and* T *are the same.*

Proof. We first apply the construction in lemma 1 to make T an efficient forest. We then repeatedly modify T in the following manner. Consider an instance of a job J_i that is preempted by its first child J_j at a time t different from r_j. Let s be the time that J_i begins execution in T. Let $m = \max(r_j; s)$. Let P be the period of time during which either J_j or J_i is being run in T. We modify the schedule

to run J_j for the next x_j time units after m in P, and run J_i for the remaining x_i time units in P. If r_j s then J_j becomes a sibling of J_i. Otherwise, J_j stays a child of J_i. Furthermore, in some cases, some children of of J_j may become children of J_i.

Note that each such change either makes a first child start execution at its release time, or shortens the aggregate heights of the nodes in the forest. Hence, after at most a finite number of steps, this process will terminate with a schedule with the desired properties. ∎

Lemma 3. *For every schedule T, there is another schedule S such that*

1. *S is an efficient forest,*
2. *every job completed in S is completed in T,*
3. *for all J_i 2 S, and for all children J_j of J_i in S, x_j v_i,*
4. *for all first children J_i in S it is the case that J_i is first executed at time r_i in S,*
5. *jSj $\frac{\gamma}{4}$ jTj.*

Proof. We first apply the construction in lemma 2 to T. We now break the proof into two cases. A job J_i is *skinny* if x_i $'_i$, and otherwise J_i is *fat*.

In the first case assume that there are more skinny jobs in T than there are fat jobs. Let T^0 be T with the fat jobs deleted. Now apply the construction in lemma 2 to T^0. Process each tree in T^0 from the root to the leaves in a pre-order fashion. At each node J_i, delete the at most descendants J_j of J_i with x_j $'_i$. After this pre-order processing has been completed, the resulting schedule T^{00} must satisfy condition 3, and contain at least $\frac{jT^0j}{(+1)}$ jobs.

In the second case assume that there are more fat jobs in T than there are skinny jobs. Let T^0 be T with the skinny jobs deleted. In [6] it is shown how to modify a schedule T^0 of fat jobs to create a schedule T^{00} that satisfies condition 3, and contains at least $\frac{\gamma}{2}$ jT^0j jobs.

Note that for > 4, $\frac{\gamma}{2}$ $\frac{1}{(+1)}$. We then re-apply the construction in lemma 2 to T^{00} to get the schedule S. Note that condition 3 of the construction in lemma 2 guarantees that condition 3 of this construction remains valid. ∎

The lemma 4 follows immediately from lemma 3 by noting the symmetry of release times and deadlines.

Lemma 4. *For every schedule T, there is another schedule S such that*

1. *S is an efficient forest,*
2. *every job completed in S is completed in T,*
3. *for all J_i 2 S, and for all children J_j of J_i in S, x_j v_i,*
4. *for all last children J_i in S it is the case that J_i finished execution at time d_i in S,*
5. *jSj $\frac{\gamma}{4}$ jTj.*

Lemma 5. *In any forest schedule S at least* 1=2 *the nodes are either last children or leaves. In any forest schedule S at least* 1=2 *the nodes are either first children or leaves.*

Proof. This follows easily by induction on the the height of the tallest tree in the forest S. ∎

Lemma 6. *For every forest schedule T there is another forest schedule S such that*

1. *S completes the same jobs as T,*
2. *S is efficient,*
3. *for all J_i completed in T, the completion time of J_i in S is the same as the completion time of J_i in T, and*
4. *if J_i is a descendant of J_j in S then J_i is a descendant of J_j in T.*

Proof. We repeatedly apply the following construction. Let J_i be an idle job with no idle descendents. Let l be the total time that the processor is idle between when J_i is first executed and when J_i is completed. We then delay the initial execution of J_i by l time units. ∎

Definition 3. { *For a schedule S, let* TEMP(S) *be the schedule that you get from applying to S the construction in lemma 4, deleting all jobs that are not last children or leaves, and the construction in lemma 6, in that order.*
{ *We say a forest is is* leafy *if at least half of the nodes in the forest are leaves.*

Lemma 7. *Let T be an arbitrary schedule for the instance J with the property that* TEMP(T) *is not leafy. Then there is another schedule S for J such that*

1. *S is an efficient forest,*
2. *every job completed in S is completed in T,*
3. *for all $J_i \in S$, and for all children J_j of J_i in S, $x_j \leq v_i$,*
4. *for all first children J_i in S, J_i is not executed before time pr_i in S,*
5. *$|S| \geq |T|$.*

Proof. We first construct TEMP(T). Note that $|\text{TEMP}(T)| \geq \frac{7}{8}|T|$. We then delete all the leaves in TEMP(T), and apply the construction in lemma 6 to get a schedule T'. Hence, $|T'| \geq \frac{7}{16}|T|$. We then delete each job J_i completed in T' such that the total execution time of the descendents of J_i in T' exceeds $\ell_i/2$. Call the resulting forest T''.

We then claim that for every job J_i that was deleted in this transformation from T' to T'' there exists a $z \geq 1$ such that there exist 2^z descendents z hops down from J_i in the forest T', i.e. J_i has 2 children, or 4 grandchildren, or 8 great-grandchildren, etc. To see this note that if this was not the case then the total execution times of the descendents of J_i is at most $\sum_{z=1}^{\infty} \frac{2^z v_i}{z} = \frac{2\ell_i}{-2}$ $\frac{v_i}{2} \leq \frac{\ell_i}{2}$. The second to last inequality follows from the fact that ≥ 6. This is a contradiction.

We now claim that $2|T''| \geq |T'|$. To see this consider the following amortization scheme. Each node $J_i \in T''$ is initially given one credit. J_i then contributes $1/2^{y+1}$ credits to its ancestor y hops up in the tree containing J_i in T', for each $y \geq 0$. We now argue by contradiction that each job J_j completed in T' gets an aggregate credit of at least 1/2. To reach a contradiction let J_j be a job in T'

that receives less than 1=2 a credit, and all of the descendents of J_j received at least 1=2 a credit. Then there must exist a $z \quad 1$ such that J_j has a collection Z of 2^z descendents that each received at least 1=2 a credit. The nodes that contributed the at least 1=2 credits to each node $J_i \ 2 \ Z$, contribute at least $\frac{1}{2^{z+1}}$ credits to J_j. Hence, J_j received at least 1=2 a credit, contradiction.

We get the final schedule S for J by applying the construction in lemma 2 to T^{00}, with the release time of each job J_i completed in T^{00} modified to be pr_i. □

3.2 Counting Pushes

Our goal in this subsection is to use an amortization argument to show that the number of jobs completed by LAX and SRPT is at least a constant times the number of times that line (2) or line (3) of LAX was executed.

Definition 4. { *Let D be the collection of jobs J_i that were popped in line (9) of* LAX*. Note that jobs in D are not completed by* LAX*.*
{ *Let C be the number of times that line (2) or line (3) of* LAX *was executed.*

Lemma 8. *Let J_i be a job in D. There exists an integer $z > 0$, such that during the interval* [Push(i); Pop(i)]*,* LAX *was running jobs with lengths in the range $R = (\frac{v_i}{z+1}; \frac{v_i}{z}]$ for $\frac{\ell_i}{2^{z+1}}$ units of time.*

Proof. Between time Push(i) and time Pop(i) LAX will only run J_i, and jobs of length at most $v_i =$. Since J_i was viable when it was added to H, LAX ran these shorter jobs for at least ℓ_i=2 units of time. Assume to reach a contradiction that, for each $z > 0$, LAX ran jobs with lengths in the range R less than $\frac{\ell_i}{2^{z+1}}$ units of time between time Push(i) and time Pop(i). Then the total time that LAX wasn't running J_i between time Push(i) and Pop(i) would have to be less than $\sum_{z=1}^{1} \frac{\ell_i}{2^{z+1}} = \frac{\ell_i}{2}$. This contradicts the assumption that LAX did not complete J_i by time d_i. □

Lemma 9. $2 \ |\mathrm{SRPT}(I)| \quad |D|$.

Proof. We give an amortization scheme to pay for jobs in D. Let J_j be a job that SRPT completes at time t. Each $J_i \ 2 \ H$ at time t is given $\frac{2}{2^{f(i)}}$ credits, where $f(i)$ is the depth of J_i in H at this time. Let Q be a maximal contiguous period of time during which LAX is running J_j. During Q, each $J_i \ 2 \ H$ is given $\frac{|Q|}{x_j} \frac{2}{2^{f(i)}}$ credits. Note that the total time that LAX runs J_j is at most x_j. Hence, the maximum credits transfered from J_j is at most $2 \sum_{c=1}^{1} \frac{2}{2^c} \quad 1$.

Consider a job $J_i \ 2 \ D$. By lemma 8, there exists a $z > 0$, such that between time Push(i) and time Pop(i) LAX spent at least $\frac{v_i}{2^{z+1}}$ units of time running jobs with lengths in the range $R = (\frac{v_i}{z+1}; \frac{v_i}{z}]$. Let X be the collection of jobs with lengths in the range R that LAX was running between time Push(i) and time Pop(i). While LAX was running jobs in X between time Push(i) and time Pop(i) it is the case that $f(i) \quad z+1$. Call a job $J_j \ 2 \ X$ *fresh* if SRPT had not finished J_j at the time that LAX started running J_j.

Repeat the following iterative argument. Consider a fresh job $J_j \in X$ (not eliminated by an earlier iteration) that LAX first started to run at a time $t \in [\text{Push}(i); \text{Pop}(i)]$. Note that SRPT finishes a job between time t and time $t+x_j$. Remove from future consideration J_j, and all fresh jobs in X that LAX begins executing between time time t and time $t+x_j$. Note that the total time that LAX was executing the removed jobs during $[\text{Push}(i); \text{Pop}(i)]$ is at most $2\frac{v_i}{z}$. Hence, during the period of the removed jobs, J_i was earning credits at an amortized rate of at least $\frac{1}{2} \cdot \frac{z}{v_i} \cdot \frac{2}{2^{z+1}} = \frac{2^z}{v_i}$ credits per unit time.

Repeat the following iterative argument on non-fresh jobs in X. Let $[t; t+p]$ be the maximal contiguous period during which LAX ran a job $J_j \in X$ (that was not eliminated by an earlier iteration). During this period, J_i was earning credits at an amortized rate of at least $\frac{z}{v_i} \cdot \frac{2}{2^{z+1}} = \frac{2^{z+1}}{v_i}$ credits per unit time. Hence, J_i earns at least $\frac{v_i}{2^{z+1}} \cdot \frac{2^z}{v_i} = \frac{1}{2}$ credits. ∎

Lemma 10. $2 \cdot |\text{SRPT}(I)| + |\text{LAX}(I)| \geq C$.

Proof. A push in line (2) or line (3) of LAX that increments the size of H from k to $k+1$ is paid for by the first subsequent pop to reduce H to size k. If the pop occurred in line (8) of the code then it is paid for by LAX, which completed the job. If the pop occurred in line (9) of the code then it is paid for by SRPT. ∎

3.3 Optimality

Definition 5. *1. Let* $\text{AOPT}(J)$ *be the schedule that is created when the construction in lemma 7 is applied to* $\text{OPT}(J)$.
2. Let N *be the collection of jobs* J_i *with* $pr_i = +1$, *that is,* $N = I - J$.

Lemma 11. *Let* S *be an arbitrary forest schedule for an instance* I. *Then* $|\text{SRPT}(I)|$ *is at least half the number of leaves in* S.

Proof. Consider a job J_i that is a leaf in $\text{OPT}(I)$ that begins execution at time t in $\text{OPT}(I)$. If SRPT didn't completed J_i before time t, then it will complete a job by time $t + x_i$. ∎

Lemma 12. $8\,|\text{SRPT}(I)| \geq |\text{OPT}(N)|$.

Proof. Let T be the schedule that you get from applying to $\text{OPT}(N)$ the construction in lemma 3, and throwing away all jobs except leaves and first children. Note that $|T| \geq \frac{1}{8}|\text{OPT}(N)|$ by lemmas 3 and 5. If T is leafy (recall definition 3), $|\text{SRPT}(I)| \geq |\text{OPT}(N)|$. Now assume that T is not leafy. Let S be T with all the leaves of T removed. Note that now each job J_i completed in S begins execution at time r_i, and that S has at least $\frac{1}{16}|\text{OPT}(N)|$ jobs.

We give an amortization scheme to pay for jobs S. One can interpret a forest schedule as the output of a stack based algorithm. At any point in time the adversary's stack is the jobs that have been started but not finished in S,

ordered from bottom to top by the time of first execution. The adversary is always either idle or running the top job from the stack. Let J_j be a job that SRPT completes at time t. Each J_i that is in the adversary's stack at time t, and satisfies v_i x_j, is given $\lfloor 2^{\log \frac{v_i}{x_j}} \rfloor$ credits. Let Q be a maximal contiguous period of time during which LAX is running J_j, and the job being run in S does not change. During Q, each J_i in the adversary's stack, with v_i x_j, is given $\frac{jQj}{x_j} \lfloor 2^{\log \frac{v_i}{x_j}} \rfloor$ credits. Hence, the maximum credits transfered from J_j is $2 \sum_{c=1}^{1} \lfloor 2^{c} \rfloor$ 1.

Consider a job J_i 2 S. By the definition of N, between time r_i and time $r_i + v_i$=2, LAX was always running jobs J_j with x_j v_i. Hence, there exists a $z > 0$, such that between time r_i and time $r_i + v_i$=2, LAX spent at least $\frac{v_i}{2^{z+1}}$ units of time running jobs with lengths in the range $R = (\frac{v_i}{z+1}; \frac{v_i}{z}]$. Let X be the collection of jobs with lengths in the range R that LAX was running between time r_i and time $r_i + v_i$=2. Call a job J_j 2 X *fresh* if SRPT had not finished J_j at the time that LAX started running J_j.

Notice that there is at most one fresh job J_a 2 X that LAX is running during the interval $[r_i; r_i + v_i$=2] that LAX did not start in this interval. This is because LAX can not have two jobs, with lengths in R, in H at the same time. Note jXj 4 since 16.

Repeat the following iterative argument. Consider a fresh job J_j 2 X (not eliminated by an earlier iteration) that LAX first started to run at a time t 2 $[r_i; r_i + v_i$=2]. If J_j 6= J_a then SRPT finishes a job between time t and time $t + x_j$. Remove from future consideration J_j, and all fresh jobs in X that LAX begins executing between time t and time $t + x_j$. Note that the total time that LAX was executing the removed jobs during $[r_i; r_i + v_i$=2] is at most $2\frac{v_i}{z}$. Hence, during the period of the removed jobs, J_i was earning credits at an amortized rate of at least $\frac{1}{2} \frac{z}{v_i} \lfloor 2^{z} \rfloor = \frac{2^{z-1}}{v_i}$.

Repeat the following iterative argument on non-fresh jobs in X. Let $[t; t+p]$ be the maximal contiguous period during which LAX ran a job J_j 2 X (that was not eliminated by an earlier iteration). During this period, J_i was earning credits at an amortized rate of at least $\frac{z}{v_i} \lfloor 2^{z} \rfloor = \frac{2^z}{v_i}$ credits per unit time. Hence, J_i earns at least $\frac{1}{2} \lfloor \frac{v_i}{2^{z+1}} \rfloor \frac{2^{z-1}}{v_i} = \frac{1}{8}$ credits. ∎

Lemma 13. *If* TEMP(OPT(J)) *is leafy then* 2 jSRPT(I)j jOPT(J)j.

Lemma 14. *Let J_d be the first child of a job J_c, that is a first child of a job J_b, that is a first child of a job J_a, in* AOPT(J). *Then during the time interval $[pr_b; pr_d]$, it must be the case that* LAX *executed line (2) or line (3) of its code.*

Proof. Assume to reach a contradiction that neither line (2) or (3) of LAX was executed during $[pr_b; pr_d]$. By observation 2 the value of $v_{h(k)}$ is monotonically nondecreasing during the time interval $[pr_b; pr_c)$.

First assume that LAX pushed J_b on H at time pr_b. By assumption this push happened in line (6) of LAX. Throughout the interval $(pr_b; pr_c]$, $v_{h(k)}$ v_b.

By observation 1, $x_c \quad v_b$. Hence, by observation 3 some job must have been pushed on H in line (2) or (3) by time pr_c, contradiction.

Now assume that LAX didn't push J_b at time pr_b. Then by the definition of pr_b, it was the case that $x_{h(k)} > v_b$ at time pr_b. By observation 1, $x_h(k)$ $v_{h(k-1)}$. Hence, $v_b \quad v_{h(k-1)}$. By lemma 7, $x_c \quad v_b$. Hence, throughout the interval $(pr_b; pr_c)$, $x_c \quad v_{h(k-1)}$.

Consider the case that J_c was pushed on H at time pr_c by line (6) of LAX. By observation 1, $x_d \quad v_c$. Hence, by observation 3 some job must have been pushed on H in line (2) or (3) by time pr_d, contradiction.

Finally, consider the case that J_c was not pushed on H at time pr_c. Hence, at time pr_c either some job must have been popped from H, or J_c was released. First consider the case that a pop occurred at time pr_c. If this pop happened in line (8) or (9) then J_c or some other job will be pushed on H in line (2) of LAX at time pr_c, contradiction. If this pop happened in line (4) of LAX then J_c, or some job with larger value, must have been pushed on H in line (6) of LAX. Hence, during the interval $(pr_c; pr_d)$, $v_{h(k)} \quad v_c$. Now consider the case that $r_c = pr_c$. In this case, either LAX didn't execute the body of the else statement because $v_{h(k)} \quad v_c$, or LAX must have executed the push in (6) at time pr_c, and since $J_c \; 2 \; V(H)$ at that time, we again get that during the interval $(pr_c; pr_d)$, $v_{h(k)} \quad v_c$.

By lemma 7, $x_d \quad v_c$. Hence, by observation 3 some job must have been pushed on H in line (2) or (3) by time pr_d, contradiction. ∎

Lemma 15. *If* TEMP(OPT(J)) *is not leafy,* $3C + 6\,|\text{SRPT}(I)| \quad |\text{AOPT}(J)|$.

Proof. We use an amortization argument to pay for the jobs completed in AOPT(J). Let J_a be an arbitrary job in AOPT(J). We now break the proof into many cases.

If J_a is a leaf in AOPT(J), then it is paid for by SRPT. Otherwise, let J_b be the first job that preempts J_a in AOPT(J). By lemma 2, J_b begins execution in AOPT(J) at time pr_b.

If J_b is a leaf in AOPT(J), then it is paid for by SRPT. Otherwise, let J_c be the first job that preempts J_b in AOPT(J). By lemma 2, J_c begins execution in AOPT(J) at time pr_c.

If J_c is a leaf in AOPT(J), then it is paid for by SRPT. Otherwise, let J_d be the first job that preempts J_c in AOPT(J). By lemma 2, J_d begins execution in AOPT(J) at time pr_d.

Else we charge J_a to the push that LAX must have executed in line (2) or line (3) during $(pr_b; pr_d]$ by lemma 14. Note that no push is charged more than three times. ∎

Theorem 1. $(6 \quad + 6 \quad + 8)|\text{SRPT}(I)| + 3 \;\; |\text{LAX}(I)| \quad |\text{OPT}(I)|$.

Proof. First assume that TEMP(OPT(J)) is not leafy. Then

$$\begin{aligned} |\text{OPT}(I)| \quad & |\text{OPT}(N)| + |\text{OPT}(J)| \\ & |\text{OPT}(N)| + \quad |\text{AOPT}(J)| \end{aligned}$$

$$
\begin{aligned}
&8j\mathrm{SRPT}(I)j + \ jA\mathrm{OPT}(J)j \\
&(6 \ +8)j\mathrm{SRPT}(I)j + 3 \ C \\
&(6 \ \ +6 \ +8)j\mathrm{SRPT}(I)j + 3 \ jL\mathrm{AX}(I)j
\end{aligned}
$$

The first inequality holds since $I = J \ [\ N$. The second inequality holds by lemma 7. The third inequality holds by lemma 12. The fourth inequality holds by lemma 15. The fourth inequality holds by lemma 10.

Now assume that $\mathrm{TEMP}(\mathrm{OPT}(J))$ is leafy.

$$
\begin{aligned}
j\mathrm{OPT}(I)j \quad & j\mathrm{OPT}(N)j + j\mathrm{OPT}(J)j \\
& j\mathrm{OPT}(N)j + 2 \ j\mathrm{SRPT}(I)j \\
& (2 \ +8)j\mathrm{SRPT}(I)j
\end{aligned}
$$

The second inequality holds by lemma 13. ∎

References

1. S. Baruah, J. Harita, and N. Sharma, "On-line scheduling to maximize task completions", *IEEE Real-time Systems Symposium*, 1994.
2. S. Baruah, G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang, "On the competitiveness of on-line real-time task scheduling", *Journal of Real-Time Systems*, **4**, 124–144, 1992.
3. P. Brucker, *Scheduling Algorithms*, Springer-Verlag, 1995.
4. K. Christian, "Analyzing Real-Time Online Scheduling Algorithms with Respect to Completion Count", manuscript.
5. S. Irani, and A. Karlin, "Online computation", Chapter 13 of *Approximation Algorithms for NP-hard Problems*, ed. D. Hochbaum, PWS Publishing, 1997.
6. B. Kalyanasundaram and K. Pruhs "Speed is as powerful as clairvoyance", *IEEE Foundations of Computer Science*, 214–223, 1995.
7. G. Koren and D. Shasha, "MOCA: A multiprocessor on-line competitive algorithm for real-time systems scheduling", *Theoretical Computer Science*, **128**, 75–97, 1994.
8. C. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation", *ACM Symposium on Theory of Computation*, 140 – 149, 1997.
9. Jiri Sgall, "On-line scheduling - a survey", `http://www.math.cas.cz/~sgall/ps/schsurv.ps.gz`, To appear in the proceedings of the Dagstuhl workshop on *On-Line Algorithms*, eds. A. Fiat and G. Woeginger, Lecture Notes in Computer Science, Springer-Verlag.
10. G. Woeginger, "On-line scheduling of jobs with fixed start and end time", *Theoretical Computer Science*, **130**, 5–16, 1994..

A Randomized Algorithm for Two Servers on the Line (Extended Abstract)

Yair Bartal[1?], Marek Chrobak[2??], and Lawrence L. Larmore[3???]

[1] Bell-Labs, Lucent Technologies, 600 Mountain Avenue, Murray Hill, NJ 0797
[2] Department of Computer Science, University of California, Riverside, CA 92521
[3] Department of Computer Science, University of Nevada, Las Vegas, NV 89154-4019

Abstract. In the k-server problem we wish to minimize, in an online fashion, the movement cost of k servers in response to a sequence of requests. For 2 servers, it is known that the optimal deterministic algorithm has competitive ratio 2, and it has been a long-standing open problem whether it is possible to improve this ratio using randomization. We give a positive answer to this problem when the underlying metric space is a real line, by providing a randomized online algorithm for this case with competitive ratio at most $\frac{155}{78}$ 1:987. This is the first algorithm for 2 servers with competitive ratio smaller than 2 in a non-uniform metric space with more than three points.
We consider a more general problem called the $(k; l)$-*server problem*, in which a request is served using l out of k available servers. We show that the randomized 2-server problem can be reduced to the *deterministic* $(2l; l)$-server problem. We prove a lower bound of 2 on the competitive ratio of the $(4; 2)$-server problem. This implies that one unbiased random bit is not sufficient to improve the ratio of 2 for the 2-server problem. Then we give a $\frac{155}{78}$-competitive algorithm for the $(6; 3)$-server problem on the real line. Our algorithm is simple and memoryless. The solution has been obtained using linear programming techniques that may have applications for other online problems.

1 Introduction

In the *k-server problem* we are given k mobile servers that occupy a metric space M. A sequence of requests is issued, where each request is a point $r \in M$. The request is satisfied by moving one of the servers to r at a cost equal to the distance from its current location to r. Our goal is to design an algorithm A that serves the requests at a small cost and satisfies the following *online property:* the decision of which server to choose is made without knowledge of future requests.

An online algorithm A is said to be *C-competitive* if the cost incurred by A to service each request sequence σ is at most C times the optimal service cost

? This research was partially conducted while the author was at ICSI, Berkeley.
?? Research supported by NSF grant CCR-9503498. This research was partially conducted when the author was visiting ICSI, Berkeley.
??? Research supported by NSF grant CCR-9503441.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 247–258, 1998.
© Springer-Verlag Berlin Heidelberg 1998

for %, plus possibly an additive constant independent of %. The *competitive ratio of A* is the smallest C for which A is C-competitive. The competitive ratio is commonly used as a performance measure of online algorithms for the k-server problem, as well as for other online problems.

Manasse, McGeoch and Sleator [16] introduce the k-server problem and prove that k is a lower bound on the competitive ratio of any deterministic online algorithm in any metric space with at least $k+1$ points. They also present an algorithm for the 2-server problem which is 2-competitive in any metric space. In the general case, for $k \quad 3$, the best known upper bound on the competitive ratio is $2k - 1$, given by Koutsoupias and Papadimitriou [12,13]. The *k-Server Conjecture* is that there exists a k-competitive algorithm that works in all metric spaces, but so far this conjecture has been proven only in a number of special cases, including trees and spaces with at most $k+2$ points [6,7,14].

For many online problems it is possible to improve the competitive ratios using randomization. Very little is known, however, about randomized algorithms for k servers. When the underlying space is uniform (all distances are 1) the competitive ratio is $H_k \quad \ln k$, the k-th harmonic number [1,17]. Metric spaces with three points have been investigated by [11] and [15]. The results of Bartal *et al.* [3] imply that in a metric space with $k + o(\sqrt{k/\log k})$ points there is a randomized k-server algorithm whose competitive ratio is smaller than k. Except for these cases, no randomized online algorithm with competitive ratio better than k is known. This is true, in fact, even for $k = 2$ and 4-point spaces.

Some lower bounds on the competitive ratios of randomized k-server algorithms are known. Blum *et al.* [4] give an asymptotic $\Omega(\sqrt{\log k/\log\log k})$ lower bound for an arbitrary space with at least $k+1$ points, and a $\Omega(\log k/\log\log k)$ lower bound for $k+1$ equally spaced points on the line. For two servers, the best known lower bound is $1 + e^{-1=2} \quad 1{:}6065$ [9].

The main contribution of this paper is a randomized online algorithm for two servers on the line whose competitive ratio is at most $\frac{155}{78} \quad 1{:}987$. This is the first algorithm for 2 servers with competitive ratio smaller than 2 in a metric space with unbounded distances, or, in fact, in a non-uniform metric space with more than three points. The real-line case is of its own interest, since it corresponds to the problem of motion planning for 2-headed disks (see [5]).

Our method is to consider a more general problem called the $(k; l)$-*server problem*, in which a request must be served using l out of k available servers. We show that the randomized 2-server problem can be reduced to the *deterministic* $(2l; l)$-server problem. We examine first the case $l = 2$. Quite unexpectedly, it turns out that no deterministic online algorithm can be better than 2-competitive for the $(4; 2)$-server problem, even in metric spaces with only four points. This implies that each randomized algorithm for 2 servers defined by uniformly choosing between two deterministic algorithms has competitive ratio at least 2, and thus is no better than a deterministic algorithm. In other words, one unbiased random bit does not help. This result relates to recent work on *barely random* algorithms, that is, algorithms that use a constant number of random bits (see, for example, [2,10]). The proof of this lower bound is given in Section 3.

In Section 4 we consider the next value of l, that is the $(6; 3)$-server problem. We concentrate on the metric space consisting of points on the real line. For this case we present an algorithm called DC2, whose competitive ratio is at most $\frac{155}{78}$ 1:987. Algorithm DC2 is very simple and memoryless. The competitive analysis has been achieved using linear programming techniques that may have applications for other online problems. Our analysis is nearly tight, since we also show that the competitive ratio of DC2 is no better than $\frac{107}{54}$ 1:981.

2 Preliminaries

Let k l 1. In the $(k; l)$-*server problem* we are given k servers $s_1; : : : ; s_k$ that reside and move in a metric space M. (For simplicity, we also use s_i to denote the current location of the i^{th} server.) Initially all servers are on the same point. At each time step a request r 2 M is issued. In response to this request we must move l servers to r in order to "satisfy" the request. The k-server problem is a special case of the $(k; l)$-server problem where $l = 1$.

Formally, an *online* $(k; l)$-*server algorithm* is a function $A(\%)$ that, to a given request sequence % 2 M , assigns the k server locations after serving %, with at least l of these locations being on the last request of %. The sequence of configurations determined by A on a given request sequence % is referred to as a $(k; l)$-*service* of %, or simply a *service*. The cost of A on %, $cost_A(\%)$, is defined as the total distance traveled by its servers while serving %.

Denote by $opt(\%)$ the optimal cost of serving the request sequence %. Algorithm A is C-*competitive* if there is a constant b such that $cost_A(\%)$ C $opt(\%)+b$ for every request sequence % . The *competitive ratio* of A is defined to be the minimum C for which A is C-competitive.

A *randomized algorithm* can be defined as a probability distribution B on the set of all deterministic algorithms for a given problem. The definition of competitiveness extends naturally to randomized algorithms, by replacing $cost_A(\%)$ by $cost_B(\%)$, the *expected* cost of B on %. We will say that B is l-*uniform* if it is defined by a uniform probability distribution on l deterministic algorithms.

Theorem 1. *The following two statements are equivalent:* (a) *There is a deterministic C-competitive online algorithm A for the* $(2l; l)$-*server problem.* (b) *There is a C-competitive randomized l-uniform online algorithm B for* 2 *servers.*

The complete proof of Theorem 1 will be given in the journal version of this paper. The general idea is to establish the following relationship between A and B: A has i servers on a given point (without loss of generality i l) iff B has a server on this point with probability $i=l$.

Throughout the paper, for the purpose of competitive analysis, we will view the problem as a game between our algorithm and the adversary who serves the requests with his own servers. We can assume that the adversary serves each request optimally. Without loss of generality, we can also assume that the adversary has just two servers a_1, a_2, that he serves each request with one server, but we charge him the cost equal l times the distance traveled by his servers.

(It may not be quite obvious that we can make this assumption. The proof will be given in the final version of the paper.) At each step the adversary chooses a server a_i, moves a_i to a point r, announces that a request has been made on r, and then our algorithm serves the request with its servers. When proving an upper bound on the competitive ratio, our goal is to show that our algorithm's cost is no more than C times the adversary's cost, independent of the adversary strategy. To prove a lower bound, we must present an adversary strategy of arbitrarily large cost that forces our algorithm to pay no less than C times the adversary cost minus a constant.

In order to prove that a given algorithm A is C-competitive we will use a potential argument, which is to define a *potential function* that maps server configurations (ours and the adversary's) into nonnegative real numbers, and satisfies the following inequality at each move:

$$cost + \qquad C \quad opt; \tag{1}$$

where *cost*, , and *opt* are, respectively, A's cost, the potential change, and the adversary cost in a given move. Inequality (1) implies the C-competitiveness of A by simple summation over the whole request sequence.

3 The (4; 2)-Server Problem

Theorem 2. *There is no online deterministic algorithm for the* $(4; 2)$*-server problem that is better than 2-competitive.*

Proof. Let N be a large integer and $= 1=2N$. We use a 4-point metric space with points x, y, z, z^0, where the distances are: $xz = yz = 1$, $xy = 2$ and $zz^0 =$. The distances xz^0 and yz^0 are in the range $1 - \quad xz^0; yz^0 \quad 1 +$, but their exact values are not relevant.

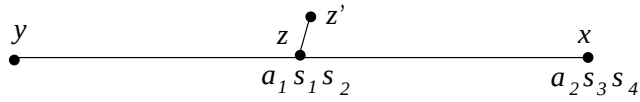


Fig. 1. The space used in the lower bound proof and the initial configuration.

Let A be an online algorithm. The adversary strategy is divided into phases. We start with $s_1 = s_2 = a_1 = z$ and $s_3 = s_4 = a_2 = x$. (see Fig. 1). The adversary now alternates requests on z^0 and z, stopping when one of the following two events occurs, whichever happens first:

(A) A moves one server from x to z (or z^0). Then the adversary requests x, completing the phase.
(B) The distance traveled by s_1 and s_2 to serve the requests on z and z^0 is 2, *i.e.*, the pair z^0z has been requested N times. We now proceed as follows:

(1) The adversary requests y. Without loss of generality, A moves s_1 and s_2 to y, and s_3, s_4 stay on x.
(2) Now the adversary requests z. We consider three subcases:
  (2.a) If A moves two servers from y then the adversary requests y, completing the phase.
  (2.b) If A moves one server from x and one server from y then the adversary requests y, completing the phase.
  (2.c) If A moves two servers from x then the adversary requests x. Now the adversary repeats Step (2) with x and y interchanged.

The request sequence generated by the adversary up to this point has the form: $z^0 z\, z^0 z \cdots z^0 z\, y z\, x z\, y z\, x z \cdots$. If the last request was on x (resp. y), the adversary alternates the request on x (resp. y) and z to make sure that A has two servers on x (resp. y) and two servers on z, before starting the next phase.

We now analyze the costs of A and the adversary, to show that, asymptotically, A's cost is at least twice the adversary cost. For clarity, we ignore low-order terms $O(\ )$ in our analysis, and when we say that "the cost of $\cdots$ is a" we mean that this cost is actually $a + O(\ )$.

In Case (A), suppose the cost of serving the sequence $zz^0 zz^0 \cdots$ is d. Since Case (A) occurred, we have $d \quad 2$. The adversary cost is d, and A's cost is $2 + d \quad 2d$, so the ratio in this phase is at least 2.

To analyze Case (B), let $j \quad 1$ be the number of repetitions of Step (2). A pays 2 to serve $zz^0 zz^0 \cdots$; then it pays 4 for each repetition of Step (2), and then it pays either 4 or 2 (or more), depending on whether case (2.a) or (2.b) occurred. So the total cost of A is at least $4 + 4j$.

We claim that the adversary cost is $2j + 2$. Initially, the adversary has a_1 on z and a_2 on x. If j is odd, move a_2 to z to serve $zz^0 zz^0 \cdots$ at no cost, and then to y when it is requested. In the remaining $j - 1$ iterations use a_1 to serve the requests on z and x. Then the cost is $4+4(j-1)=2 = 2j+2$. If j is even, a_2 stays on x and all requests on z and y are served by a_1 at the cost of $2+4j=2 = 2j+2$. In either case, the adversary returns to the initial configuration.

If a phase never ends, A pays 4 during each iteration, and the adversary pays 4 every second iteration, thus the ratio approaches 2. ∎

Note that the lower bound proof uses a metric space with only four points, and that these points could be located on a line, or they can form a star (a metric space corresponding to the weighted-cache problem). Theorems 1 and 2 imply the following result.

Corollary 1. *There is no randomized 2-uniform online algorithm for 2 servers with competitive ratio better than 2.*

4 The (6; 3)-Server Problem for the Line

4.1 Algorithm DC2.

We name our servers $s_1; :::; s_6$, ordered from left to right. We serve a request r in a sequence of as many as three "transitions," each of which moves one or more servers toward r. DC2 repeats this process until there are three servers at r.

Outward Transition: If $s_i > r$, for any $i = 1; 2; 3$, then move s_i to r. Similarly, if $s_i < r$, for any $i = 4; 5; 6$, then move s_i to r. We will sometimes refer to these transitions as *leftward* or *rightward*, depending on which of the two cases above occurred. (Clearly, only one of these cases can occur.)

Inward Transition: Suppose s_3 r s_4 and there are fewer than three servers on r. Each server has a *speed* assigned to it: s_1 and s_6 have speed 2, and the other four servers have speed 1. Pick largest i and smallest j such that $s_i < r < s_j$, and move s_i and s_j continuously towards r, at their assigned speeds, until one of them reaches r.

By definition, outward transitions will be executed first, if they apply. The algorithm is illustrated in Figure 2. The request on r is served in a sequence of three transitions: (i) first s_3 moves to r, then (ii) s_2 and s_4 move towards r with the same speed, and s_2 reaches r first, and then (iii) s_1 and s_4 move towards r, with s_2 moving twice as fast as s_4. Servers s_1 and s_4 arrive at r simultaneously.

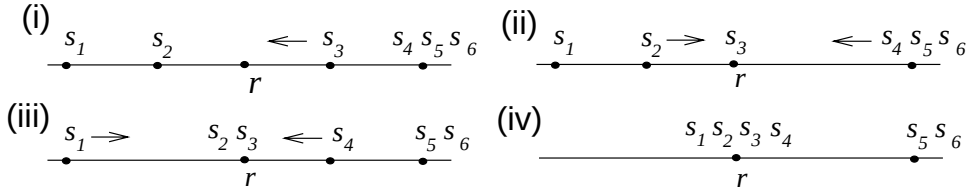


Fig. 2. An example of DC2 serving a request.

4.2 The Linear Programming Approach

In this section we outline the method we used to estimate the competitive ratio of algorithm DC2 and to compute the potential function. The general idea is to assume that the potential function can be expressed as a linear combination of the distances between the servers, with unknown coefficients $_i$. With each move we can associate an inequality (1), which now becomes a linear inequality involving the $_i$ and the competitive ratio C of DC2. This gives us a linear program whose objective function is to minimize C.

For better illustration, we restrict the movement of the adversary servers by assuming that he only moves his servers a_1 and a_2 leftward. We call this adversary *leftist*. We also assume that initially all servers are on the same point.

Without loss of generality we assume that the adversary moves each server a_i only when he is going to request it. We divide each step into *basic moves*. Each basic move involves either an adversary moving a server, or DC2 moving one or more servers, or possibly a combined move of the adversary and DC2. During a basic move, the set of servers that are in motion does not change.

Each move of a_1 (to the left) is a basic move. Any inward transition of DC2 is also a basic move. When the request is on a_1, we execute the leftward transitions in up to three basic moves: (i) move s_3, (ii) if s_3 reaches s_2 then we move s_2 and s_3 together, and (iii) if they both reach s_1, then all s_1, s_2 and s_3 move together. When the adversary intends to make a request with a_2 to the left of s_3, this adversary move and the resulting leftward transitions are replaced by the following basic "drag" moves: (i) when a_2 passes s_3 moving left, then s_3 moves to the left together with a_2, (ii) when a_2 passes s_2 moving left, then then s_2 and s_3 move to the left together with a_2, and (iii) when a_2 passes s_1 moving left, then s_1, s_2 and s_3 move to the left together with a_2. Further, we can also assume that a_2 never moves to the left of a_1, for whenever this happens, we can simply switch the labels of a_1 and a_2.

With these assumptions, all servers are ordered $a_1; s_1; s_2; s_3; a_2; s_4; s_5; s_6$ from left to right. This gives us a partition of the interval $[a_1; s_6]$ into seven intervals between consecutive servers. Let x_i be the length of the ith interval. We assume now that the potential function is a linear combination of the x_i:

$$= \sum_{i=1}^{\bar{X}} {}_i x_i \tag{2}$$

where the ${}_i$ are unknown non-negative coefficients.

Now we set up our linear program. Let *cost*, *opt* and denote our cost, the adversary cost, and the potential change in a basic move. Inequalities corresponding to the basic moves have the form *cost* + C *opt*. For example, when s_3 and s_6 move, the corresponding inequality is $3 + {}_3 - {}_4 - 2 {}_7$ 0. When a_2 moves left, the inequality is $- {}_4 + {}_5 \quad 3C$.

The objective is to minimize C. Unfortunately, the solution for this linear program is $C = 2$. Thus we cannot analyze DC2 accurately using a potential function in the form of (2). The main idea leading to more accurate analysis is to notice that it is not necessary to use the same weight on a given interval at all times. We can allow weights to change according to the following rule: a weight of x_i can be decreased at any time, but it can increase only when $x_i = 0$.

Divide the computation into *phases*, each phase starting when a_1 makes a request. In each given phase we distinguish two stages. Stage 1 starts when the phase starts. When s_3 and s_4 meet on a request for the first time in this phase, Stage 1 ends and Stage 2 begins, and it lasts till the end of the phase. If s_3 and s_4 do not meet, Stage 1 lasts throughout the phase and Stage 2 is empty. In Stage 1, x_2 and x_5 will have weights ${}_2$ and ${}_5$. In Stage 2, they will have weights ${}^0_2$ and ${}^0_5$, respectively. Since x_2 may be non-zero when Stage 1 ends, we need the inequality ${}_2 \quad {}^0_2$. Similarly, x_5 will be generally non-zero when Stage 2 ends (and the next phase begins), so we also need ${}^0_5 \quad {}_5$.

s_3	$1 - {}_3 + {}_4$	0		
s_2s_3	$2 - {}_2 + {}_4$	0	$2 - {}^0_2 + {}_4$	0
$s_1s_2s_3$	$3 - {}_1 + {}_4$	0		
s_3 ! s_4	$2 + {}_3 - {}_4 - {}_5 + {}_6$	0		
s_3 ! s_5	$2 + {}_3 - {}_4 - {}_6 + {}_7$	0		
s_3 ! s_6	$3 + {}_3 - {}_4 - 2\,{}_7$	0		
s_2 ! s_4	$2 + {}_2 - {}_3 - {}_5 + {}_6$	0	$2 + {}^0_2 - {}_3 - {}^0_5 + {}_6$	0
s_2 ! s_5	$2 + {}^0_2 - {}_3 - {}_6 + {}_7$	0		
s_1 ! s_4	$3 + 2\,{}_1 - 2\,{}_2 - {}_5 + {}_6$	0	$3 + 2\,{}_1 - 2\,{}^0_2 - {}^0_5 + {}_6$	0
a_1	${}_1$	$3C$		
a_2	$- {}_4 + {}_5$	$3C$		
s_3a_2	$1 - {}_3 + {}_5$	$3C$	$1 - {}_3 + {}^0_5$	$3C$
$s_2s_3a_2$	$2 - {}_2 + {}_5$	$3C$	$2 - {}^0_2 + {}^0_5$	$3C$
$s_1s_2s_3a_2$	$3 - {}_1 + {}_5$	$3C$		
weight change	${}^0_2$	${}_2$	${}_5$	${}^0_5$

Table 1. The inequalities of the linear program for the leftist adversary. The objective function is to minimize C. Some moves give rise to two inequalities, one for Stage 1, and one for Stage 2. All variables are nonnegative: ${}_1; \ldots; {}_7; {}^0_2; {}^0_5$ 0.

Table 1 shows the resulting linear program. The explanation of each inequality is given in the left column. The notation for moves in Table 1 is self-explanatory. For example, "s_3 ! s_5" denotes the inward transition involving s_3 and s_5, and " s_1s_2" denotes an outward basic move of s_1 and s_2.

The solution of the linear program in Table 1 is $C = \frac{155}{78}$, and the weights are: ${}_1$ = 155=26, ${}_2$ = 149=26, ${}^0_2$ = 145=26, ${}_3$ = 111=26, ${}_4$ = 77=26, ${}_5$ = 232=26, ${}^0_5$ = 240=26, ${}_6$ = 142=26, ${}_7$ = 56=26. This solution constitutes a proof that DC2 is $\frac{155}{78}$-competitive against the leftist adversary.

4.3 Competitive Analysis of DC2

We first define our potential function . Recall that our servers are ordered $s_1; \ldots; s_6$, from left to right, and, without loss of generality, a_1 is always to the left of a_2. It is convenient to express the potential as the sum of three quantities, Ψ, Γ and . We define Ψ and Γ first:

$$\Psi = \frac{1}{26}[\,33|a_1 - s_1| + 60|a_1 - s_2| + 62|a_1 - s_3| + 33|a_2 - s_6| + 60|a_2 - s_5| + 62|a_2 - s_4|\,]$$

$$\Gamma = \frac{1}{26}[\,23(s_6 - s_1) + 26(s_5 - s_2) + 28(s_4 - s_3)\,]$$

The formulas for Ψ and Γ are represented in Figure 3, which is a weighted graph whose vertices are the eight points a_i and s_j. Ψ is represented by dashed edges below the line, and Γ is represented by dashed edges above the line.

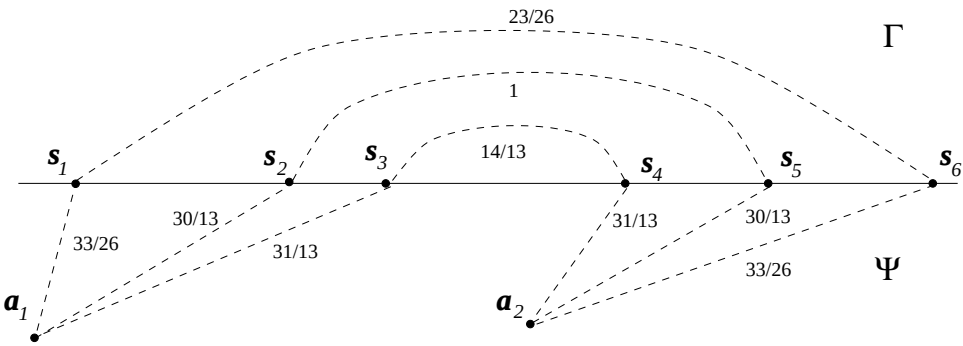


Fig. 3. The representation of Ψ and Γ.

The relationship between $\Psi + \Gamma$ and the potential function in the previous section can be seen by comparing the weights of the intervals between the servers. For example, if a_2 is to the left of s_4 then the weight in $\Psi + \Gamma$ of the interval between s_4 and s_5 is $23/26 + 1 + 30/13 + 33/26 = 142/26$, the same as $_6$ in the previous section. However, not all the weights are the same as before. We need to incorporate into the potential the idea of weights that can vary in different stages of the computation. Furthermore, since now we are not placing any restrictions on the adversary behavior, the servers may not necessarily be ordered as in the previous section. We solve these difficulties by introducing another, appropriately defined, term . Let $[x]^+ = \max\{x, 0\}$. Then is defined as follows:

$$_1 = \min \begin{matrix} [s_2 - a_1]^+ \\ s_2 - s_1 \end{matrix} \qquad _1 = \frac{2}{13} \min \begin{matrix} 2(s_4 - s_3) + [\ ^1_1 - s_6 + s_5]^+ \end{matrix}$$

$$_2 = \min \begin{matrix} [a_2 - s_5]^+ \\ s_6 - s_5 \end{matrix} \qquad _2 = \frac{2}{13} \min \begin{matrix} 2(s_4 - s_3) + [\ ^2_2 - s_2 + s_1]^+ \end{matrix}$$

and we take $= \max\{ _1; \ _2\}$. Now we are ready to prove our main theorem.

Theorem 3. *Algorithm* DC2 *is* $\frac{155}{78}$*-competitive.*

Proof. We use the potential function $= \Psi + \Gamma +$, where Ψ, Γ, are defined above. Note that is preserved under translation and inversion (flipping).

At each step the adversary moves one of its servers to the request point, announces the request, and then our servers serve the request. We divide each service into at most three *basic moves*. Each adversary move is a basic move. Also, each inward transition is a basic move. The leftward transitions are replaced by a sequence of three basic moves: (i) moving s_3 first till it reaches s_2, (ii) moving

s_2 and s_3 together till they reach s_1, and (iii) moving s_1, s_2 and s_3 together. The rightward transitions are replaced by rightward basic moves in a similar fashion. We view each basic move as a continuous process during which the servers move at their assigned speeds By we denote the length of the time interval needed to execute a given basic move. Our goal is to show that $\mathit{cost} +$ $\frac{155}{78}$ opt.

Adversary Moves. By symmetry, we may assume that the adversary moves a_1. The adversary cost is $\mathit{opt} = 3$. This move only affects Ψ and $_1$. If $a_1 < s_1$ or $a_1 > s_2$ then Ψ $\frac{155}{26}$ and $_1 = 0$. If $s_1 < a_1 < s_2$ then Ψ $\frac{89}{26}$ and $_1$ $\frac{2}{13}$. So $\frac{155}{78}$ opt in each case.

Outward Basic Moves. By symmetry, we only need to analyze leftward moves. We summarize the values of cost, Ψ, Γ and in the table below. The numbers represent the worst-case values of the corresponding quantities.

Move	*cost*	Ψ	Γ	
s_3		$-\frac{31}{13}$	$\frac{14}{13}$	$\frac{4}{13}$
$s_2 s_3$	2	$-\frac{61}{13}$	$\frac{27}{13}$	$\frac{6}{13}$
$s_1 s_2 s_3$	3	$-\frac{155}{26}$	$\frac{77}{26}$	0

The verification of the numbers in this table is quite straightforward. To illustrate, we explain the numbers in the last row, corresponding to the move when $s_1; s_2$ and s_3 move to the left. Since all three servers move, $\mathit{cost} = 3$. Since a_1 is to the left of $s_1; s_2; s_3$, the value of each absolute value $ja_1 - s_1j$, $ja_1 - s_2j$ and $ja_1 - s_3j$ in Ψ decreases by , and thus $\Psi = -\frac{155}{26}$. In Γ, all differences $s_6 - s_1$, $s_5 - s_2$ and $s_4 - s_3$ increase by , and thus $\Gamma = \frac{77}{26}$. The argument for is more subtle: During this move, $s_1 = s_2$. Thus we have $_1 = 0$, implying that $_1 = 0$. $_2 = 0$, and $2(s_4 - s_3) + [\ _2 - s_2 + s_1]^+ = 2(s_4 - s_3) + \ _2$, therefore $_2 = \frac{2}{13}\ _2$. We conclude that $_2 = 0$.

Inward Basic Moves. By symmetry we may assume a_2 is between s_3 and s_4. Therefore $_2 = 0$, and $= \Psi + \Gamma + \ _1$. We summarize the costs and potential changes in the table below.

	Move	*cost*	Ψ	Γ	$_1$
	s_3 ! s_4	2	0	$-\frac{28}{13}$	0
	s_3 ! s_5	2	$\frac{1}{13}$	$-\frac{27}{13}$	0
	s_3 ! s_6	3	$-\frac{2}{13}$	$-\frac{37}{13}$	0
	s_2 ! s_4	2	$-\frac{1}{13}$	$-\frac{27}{13}$	$\frac{2}{13}$
	s_2 ! s_5	2	0	-2	0
a_1	s_1 and s_1 ! s_4	3	$\frac{2}{13}$	$-\frac{37}{13}$	$-\frac{4}{13}$
$a_1 > s_1$	and s_1 ! s_4	3	$-\frac{64}{13}$	$-\frac{37}{13}$	0

To illustrate, we analyze the move "s_2 ! s_5", the inward transition involving s_2 and s_5. Since both servers move at speed 1, $\mathit{cost} = 2$. Since s_5 moves towards a_2, the expression $ja_2 - s_5j$ in Ψ decreases by , while $ja_1 - s_2j$ cannot

increase by more than , and the other terms in Ψ do not change. Therefore Ψ 0. In Γ, only the term $(s_5 - s_2)$ changes, decreasing by 2 , and thus $\Gamma = -2$. To determine the upper bound on $_1$, note that in this move $s_3 = s_4$, implying that $_1 = \frac{2}{13}[\ _1 - s_6 + s_5]^+$. Since s_5 decreases by , and $_1$ increases by at most , we obtain $_1$ 0.

Other moves can be analyzed in a similar fashion. Summarizing, we have $cost +$ $\frac{155}{78}$ opt for each move, and therefore DC2 is $\frac{155}{78}$-competitive. □

Our analysis is nearly tight, as shown in the following theorem. (The proof will be given in the journal version of the paper.)

Theorem 4. *The competitive ratio of* DC2 *is no less than* $\frac{107}{54}$.

5 Final Comments

We have presented a $\frac{155}{78}$-competitive randomized online algorithm for 2 servers on the line. This is the first online algorithm for 2 servers with competitive ratio smaller than 2 in a non-uniform metric space with more than 3 points.

One natural question is whether we can improve the competitive ratio by using other server speeds. We experimented with different speeds for s_1 and s_6, but the calculations analogous to the ones in the paper yield only C 1:986 for speeds approximately 1:75.

It still remains an open problem whether there is a less-than-2-competitive randomized algorithm for 2 servers that works in an arbitrary metric space. The general problem appears very hard, so it is reasonable to concentrate on other natural special cases: star-shaped spaces (or the weighted-cache problem), and trees. We believe that our technique can be extended to continuous trees. Any result that considerably improves the lower bound of $1 + e^{-1=2}$ 1:6065 from [9] would also be of interest.

In addition to being useful for studying randomized 2-server algorithms, the $(k; l)$-server problem is of its own interest. What is the best competitive ratio of online $(k; l)$-server algorithms? The problem is apparently very difficult, since even the case $l = 1$, that is, the k-server problem, is still open. As given in this paper, the competitive ratio of the $(4; 2)$-server problem is 2. It is possible that the work function techniques [8,12,13] can be extended to at least some cases, for example $k = 2l$ and $k = l - 1$.

References

1. Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. In *Proc. 4th European Symp. on Algorithms*, volume 1136 of *Lecture Notes in Computer Science*, pages 419–430. Springer, 1996.
2. Susanne Albers, Bernhard von Stengel, and Ralph Werchner. A combined BIT and TIMESTAMP algorithm for the list update problem. *Information Processing Letters*, 56:135–139, 1995.

3. Yair Bartal, Avrim Blum, Carl Burch, and Andrew Tomkins. A polylog(n)-competitive algorithm for metrical task systems. In *Proc. 29th Symp. Theory of Computing*, pages 711–719, 1997.
4. Avrim Blum, Howard Karloff, Yuval Rabani, and Michael Saks. A decomposition theorem and lower bounds for randomized server problems. In *Proc. 33rd Symp. Foundations of Computer Science*, pages 197–207, 1992.
5. A. R. Calderbank, Edward G. Coffman, and Leopold Flatto. Sequencing problems in two-server systems. *Mathematics of Operations Research*, 10:585–598, 1985.
6. Marek Chrobak, Howard Karloff, Tom H. Payne, and Sundar Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4:172–181, 1991.
7. Marek Chrobak and Lawrence L. Larmore. An optimal online algorithm for k servers on trees. *SIAM Journal on Computing*, 20:144–148, 1991.
8. Marek Chrobak and Lawrence L. Larmore. The server problem and on-line games. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 7, pages 11–64, 1992.
9. Marek Chrobak, Lawrence L. Larmore, Carsten Lund, and Nick Reingold. A better lower bound on the competitive ratio of the randomized 2-server problem. *Information Processing Letters*, 63(2):79–83, 1997.
10. Sandy Irani and Steve Seiden. Randomized algorithms for metrical task systems. In *Proc. 4th Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1995.
11. Anna Karlin, Mark Manasse, Lyle McGeoch, and Susan Owicki. Randomized competitive algorithms for non-uniform problems. In *Proc. 1st Symp. on Discrete Algorithms*, pages 301–309, 1990.
12. Elias Koutsoupias and Christos Papadimitriou. On the k-server conjecture. In *Proc. 26th Symp. Theory of Computing*, pages 507–511, 1994.
13. Elias Koutsoupias and Christos Papadimitriou. On the k-server conjecture. *Journal of the ACM*, 42:971–983, 1995.
14. Elias Koutsoupias and Christos Papadimitriou. The 2-evader problem. *Information Processing Letters*, 57:249–252, 1996.
15. Carsten Lund and Nick Reingold. Linear programs for randomized on-line algorithms. In *Proc. 5th Symp. on Discrete Algorithms*, pages 382–391, 1994.
16. Mark Manasse, Lyle A. McGeoch, and Daniel Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.
17. Lyle McGeoch and Daniel Sleator. A strongly competitive randomized paging algorithm. *Journal of Algorithms*, 6:816–825, 1991.

On Nonblocking Properties of the Benes Network

Petr Kolman*

Department of Applied Mathematics, Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic
kolman@kam.ms.mff.cuni.cz

Abstract. A network is called nonblocking if every unused input can be connected by a path through unused edges to any unused output, regardless of which inputs and outputs have been already connected. The Beneš network of dimension n is shown to be strictly nonblocking if only a suitable chosen fraction of $1/n$ of inputs and outputs is used. This has several consequences. First, there is a very simple strict sense nonblocking network with $N = 2^n$ inputs and outputs, namely a $(n + \log n + 1)$–dimensional Beneš network. Its depth is $O(\log N)$, it has $O(N \log^2 N)$ edges and it is not constructed of expanders. Secondly it leads to a $(3 \log N)$–competitive randomized algorithm for a $(\log N)$–dimensional Beneš network and a $O(\log^2 N)$–competitive randomized algorithm for a $(\log N)$–dimensional hypercube, for routing permanent calls.

1 Introduction

A network is a directed graph with two disjoint subsets of nodes called inputs and outputs. It is rearrangeable if, given any set of disjoint paths between some inputs and outputs, every unused input can be connected by a path through unused edges to any unused output with possible rearrangement of the previously established paths. If the rearrangement is prohibited and still it is possible to find the path from an unused input to an unused output the network is called strict sense nonblocking.

Searching for disjoint paths is closely related to circuit-switching routing technique. This technique has been used in telecommunication networks, in parallel computers and also the ATM standard is based on virtual circuit routing.

The Beneš network has been proposed for use in telephone networks [6]. It and other closely related networks were also used as interconnection networks for parallel computers (e.g. BBN Butterfly GP1000). More recently, it has been suggested for use in ATM switches [16]. Beneš [6] and Beizer [5] proved that the Beneš network is rearrangeable, and Waksman [17] gave an simple recursive algorithm for establishing the connections. A natural question is whether the network is also strict sense nonblocking? We show that if we use more then a

* This research has been supported by the Grant Agency of the Czech Republic under grant No. 102/97/1055 and by European Community under project ALTEC-KIT INCO-COP 96-0195.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 259{270, 1998.
c Springer-Verlag Berlin Heidelberg 1998

fraction of 3=4 of all inputs and outputs it is not. On the other hand if in n–dimensional Beneš network only a suitable chosen fraction of $1=n$ of inputs and outputs is used (e.g. the first $2^n=n$ inputs and outputs), then it is strict sense nonblocking. That is we have a very simple strict sense nonblocking network with $N = 2^n$ inputs and outputs, namely a $(n+\log n+1)$–dimensional Beneš network. It has $O(N \log^2 N)$ edges and depth $O(\log N)$. An almost identical network (the Cantor network, cf. [7]) was shown, by other method, to be strictly nonblocking by Melen and Turner [14].

There are asymptotically smaller strictly nonblocking networks. Bassalygo and Pinsker [4] proved existence of an asymptotically optimal network with $O(N \log N)$ edges (the lower bound was given by Shannon [15]). However, their result relies heavily on expander graphs. Also other constructions of nonblocking networks and their generalizations are based on expander graphs [1,8]. If we compare the constants hidden in the big O notation in the result of Bassalygo and Pinsker [4] and in our result we find that for less than about 2^{15} inputs and outputs the Beneš network is the smaller one. Thus, from practical point of view the Beneš network is at least comparable.

This property has an application in on–line algorithms for call admission and circuit routing. It leads to a $(3 \log N)$–competitive randomized algorithm for the $(\log N)$–dimensional Beneš network and a $O(\log^2 N)$–competitive randomized algorithm for the $(\log N)$–dimensional hypercube, for permanent calls problem. Although there are algorithms with better competitive ratios for other networks (e.g. randomized logarithmically competitive algorithms for line [9], trees [2,3] and meshes [11], deterministic $O(\log N)$–competitive algorithms for expander graphs [10]) we are not aware of any ones for Beneš network and the hypercube.

In the next section the crucial notion of i–similarity is introduced and some useful lemmas as well as necessary and sufficient condition for existence of a free path in a partially blocked Beneš network are given. This yields an efficient algorithm for searching for a path in a partially blocked network in Sect. 3. The partially nonblocking property of the Beneš network and its consequences for the on–line routing are given in Sect. 4. Section 5 states some stronger results about Beneš networks of very small dimensions. Finally, in Sect. 6 it is shown that the Beneš network is not so good in the stronger nonblocking properties as it is in rearrangeability.

2 De nitions and Tools

A *network* is a directed graph with two disjoint subsets of nodes called inputs and outputs. Its *depth* is the length of the maximal shortest path between inputs and outputs, and its *order* is the number of its inputs and outputs. There are two types of *requests* for a network: (i) to make a connection between given input and output, and (ii) to destroy a connection. If not stated otherwise, connection requests will be considered only between unused inputs and outputs.

Most of the time we will consider blocking of edges. Therefore the following definitions are given only for this case. However, when needed (Theorem 10) analogous definitions will be assumed also for a case of blocking nodes.

A network with N inputs and N outputs is called *strict sense edge nonblocking of order N* [1] if, no matter what paths are established in it, it is possible to connect every unused input to any unused output through unused edges. If there is an algorithm that is able to serve every sequence of requests in an edge disjoint fashion, starting from an empty network, the network is called *wide sense edge nonblocking* [8,1]. If there is an algorithm that is able to serve each sequence of connection requests only we call the network *weak sense edge nonblocking*. A network is *edge rearrangeable* [6,1] if for any set of requests given at once it is possible to serve all the requests in an edge disjoint fashion. Clearly, any strict sense nonblocking network is wide sense nonblocking, wide sense nonblocking is weak sense nonblocking, weak sense nonblocking is rearrangeable.

The n-dimensional *butterfly* network is defined as follows [12, p.454]. The nodes correspond to pairs $(w; i)$, where i is an integer, $0 \quad i \quad n$, and w is an n-bit binary number. Usually i is called the *level* of the node and w its *row*. There is an edge from $(w; i)$ to $(u; j)$ if and only if $j = i + 1$, and w and u are either identical, or they differ precisely in the jth bit. The n-dimensional *Benes* network B_n consists of two back-to-back butterflies. Overall it has $2n + 1$ levels. The 0–level nodes are the inputs, $2N$–level nodes are the outputs.

Numbers a and b are said to be i{*similar* if their binary representations agree on i least significant bits. Their *similarity* is 2^{i-1}. Two requests or paths from a to b and from c to d are i{*similar* if either a and c, or b and d are i–similar.

Lemma 1. *Suppose there are k paths established in the Benes network B_n. Then these paths are edge disjoint if and only if there is no Benes subnetwork B_i, $0 \quad i \quad n$, and no paths $p_1; p_2$ such that both p_1 and p_2 are passing through the B_i and they are $(i + 1)${similar.*

Proof.) By contradiction. Suppose the paths are edge disjoint and there is B_i with two paths p_1 and p_2 passing through it, with their *outputs*, without loss of generality, $(i + 1)$–similar. Then, after $n + i$ steps they have to reach a node on a line that is i–similar with their outputs and, since in each B_i there is only one such line, the paths reach the same node after $n + i$ steps. But because the outputs are $(i + 1)$–similar both paths have to continue via the same edge from the B_i, which is a contradiction.

(Analogically. □

Lemma 2. *Suppose there are k paths established in the Benes network B_n and there is a request for the next path from a to b. Then a path for this request cannot go, in an edge disjoint fashion with the previously established paths, through a vertex u on the medium level n if and only if there is an already established path p such that*

- *p is $(i + 1)${similar with the request $a \; ! \; b$, and*
- *p is passing through a medium level vertex v such that $bu{=}2^i c = bv{=}2^i c$.*

Proof.) Let us establish the path from a to b through u. Then it will use some edge already used by some path p. Let i be maximal number such that p and $a \to b$ are $(i+1)$–similar. Then both of them pass through the same B_i (Lemma 1), and thus $b\frac{u}{2^i}c = b\frac{v}{2^i}c$, i.e. we have the desired path.

(The path p disables all medium level nodes in the B_i it passes through from being a medium level nodes for path from a to b. Since $b\frac{u}{2^i}c = b\frac{v}{2^i}c$, we know u is one of these vertices. □

The lemma says that each path p, which is $(i+1)$–similar with the given request $a \to b$, disables the interval $I_p = \langle k2^i, \ldots, (k+1)2^i - 1\rangle$ of 2^i medium level nodes from being a medium level node of the path from a to b, where $k = b\frac{v}{2^i}c$ and v is the medium level node of p (Fig. 1). We have the following theorem:

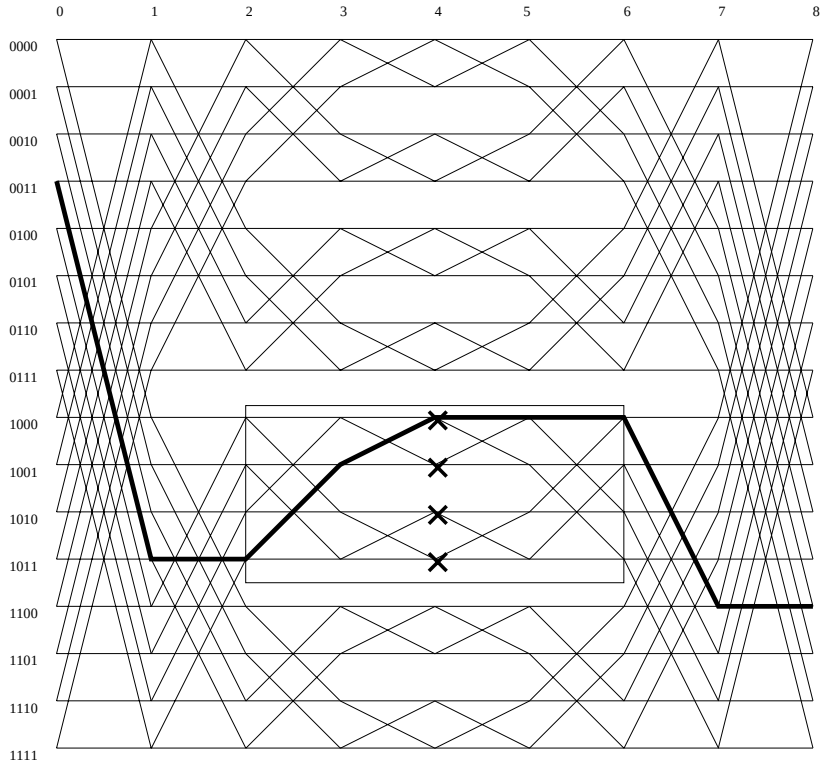


Fig. 1. The path p from input 3 to output 12 disables medium level nodes $8, 9, 10, 11$ from being a medium level node for a path from a ($0 \le a \le 15, a \ne 3$) to $b = 4$. The path p and the request $a \to b$ are (at least) 3–similar, their similarity is $2^{3-1} = 4$, $k = 8 \div 4 = 2$, $I_p = \langle 8, 9, 10, 11\rangle$.

Theorem 1 (Necessary and sufficient condition for existence of a free path). *Given a set P of paths and a request for a path from input a to output*

b, there is a free path from a to b if and only if

$$\bigcup_{p \in P} I_p \subsetneq \{0, 2^n - 1\} \ .$$

Let us notice also that for two intervals I_p and I_q, corresponding to paths p and q, such that $I_p \setminus I_q \neq \emptyset$ and $|I_p| \geq |I_q|$, it is always the case that $I_p \supseteq I_q$.

3 Searching for a Path

Theorem 2. *Given a set P of k paths in the Benes network and a request for a path from input a to output b, it is possible to decide in time $O(k)$ whether there exists a free path for this request, and if it exists to describe it.*

Proof. Due to Theorem 1 it suffices to decide whether the union of intervals I_p covers all nodes on the medium level or not. We shall use an array c of size 2^n to solve the problem. For simplicity we shall assume that this array is initialized to 0, later on we shall see how to avoid this requirement (we would like to avoid it since the initialization itself would take $\Omega(2^n)$ time). By taking any of the uncovered nodes in the medium level, if there is any, as the medium level node for the path from a to b we have also unique description of the new path.

Algorithm

```
for each path p ∈ P do
    i = max{similarity(a, in_p); similarity(b, out_p)}
    if i > 0 then
        k = ⌊medium_p / 2^i⌋
        b_p = k·2^i                    /* beginning of the interval I_p */
        e_p = (k + 1)2^i − 1           /* end of the interval I_p */
        c[b_p] = max{e_p, c[b_p]}
    endif
enddo
pointer = 0
while (pointer < 2^n) and (c[pointer] ≠ pointer) do
    pointer = c[pointer] + 1
enddo
if pointer < 2^n then "There is a free path via medium level node pointer."
    else "There is no free path from a to b."
endif
```

The values in the array c have the following properties at the end of each round of the first cycle: if $c[j] = 0$ then j is not a lower endpoint of interval I_p for any path p considered so far, and if $c[j] \neq 0$ then $c[j]$ is the upper endpoint of the largest interval with lower endpoint in j. It follows from this and from the fact that one of two overlapping intervals is always contained in the other that at the end of the algorithm *pointer* points to the minimal index unblocked medium level node, if there is any, or $pointer = 2^n$ if there is none.

It remains to show how to avoid the initialization. We will add some information to the array c: whenever we set the value of $c[j]$ we add information

about due to which path we set $c[j]$ in this way. Whenever reading $c[j]$ we verify whether the information read is really valid. The case $c[j]$ is not valid corresponds to $c[j] = 0$. This change causes only a constant slowdown.

Since both cycles run at most k times and each round takes $O(1)$ time, (in our model we count for an operation on two n–bits numbers a unit cost) the algorithm runs in time linear in the number of paths. □

For establishing and searching for paths it is also possible to use a parallel breadth first search with running time $O(n)$.

4 Nonblocking Property of the Benes Network and its Application for On{line Routing

Theorem 3. *The Benes network B_n is a strict sense edge nonblocking network of order $2^n/n$.*

Proof. Let us take as inputs only the 0–level nodes that agree on the $\lceil \log n \rceil - 1$ most significant bits (the leftmost) and similarly as outputs only the $2n$–level nodes that agree on the $\lceil \log n \rceil - 1$ most significant bits. The number of such inputs (and outputs) is $2^{n-(\lceil \log n \rceil - 1)} \geq 2^n/n$. It does not matter what is the actual setting of the $\lceil \log n \rceil - 1$ bits, especially the setting of the inputs need not be the same as of the outputs.

Suppose there are some paths established between the chosen inputs and outputs. Consider any request $a \rightarrow b$ from an unused input a to an unused output b. Our aim is to prove an existence of a free path for it. The proof will rely on the Theorem 1. We will show that the sum of similarities of a with all other input nodes is strictly less than 2^{n-1}, and the same for b and output nodes: The number of i–similar and not $(i+1)$–similar inputs is $2^{n-i-\lceil \log n \rceil}$, for $1 \leq i \leq n - \lceil \log n \rceil$. The similarity of each such input is 2^{i-1} and the sum of similarities of all i–similar inputs is $2^{n-\lceil \log n \rceil - 1}$. Since $(n - \lceil \log n \rceil)2^{n-\lceil \log n \rceil - 1} < 2^{n-1}$ and the same holds also for the outputs, we conclude that the sum of similarities of the current request with all already established paths is strictly less then 2^n. So the union $\bigcup_{p \in P} I_p$ of intervals is surely a proper subset of $\langle 0; 2^n - 1 \rangle$ and there is a free path from a to b. □

Consider the following modification of the $(n + \lceil \log n \rceil)$–dimensional Beneš network: replace each path of lenght $\lceil \log n \rceil$ starting in the first 2^n inputs by a single edge, and remove all other edges in the first $\lceil \log n \rceil$ levels. Do the same for outputs. You will get a Cantor network and it was also proved by other method to be strictly nonblocking [14].

From this point to the end of the section a slightly different kind of requests will be considered. There will be no destruction requests but only requests for connection. Each request will contain information about its starting and final points, which have to be 0–level and $2n$–level nodes, and about its duration. If the duration of all calls is infinite we speak about permanent calls problem, otherwise about known limited duration call problem. There is no restriction on the number of requests originating and terminating in a node. The property of the

Beneš network described in Theorem 3 leads to a randomized $O(n)$-competitive on-line algorithm for the n-dimensional Beneš network and similar algorithm for the hypercube. We analyze our algorithms against oblivious adversary. It's worth noting than even only weak sense nonblocking property will yield the same results.

Theorem 4. *There is a randomized $3n$-competitive algorithm for permanent calls problem on the n-dimensional Benes network.*

Proof. The immediate consequence of Theorem 3 is that there is a deterministic 3–competitive algorithm for establishing paths between only the first 2^n inputs and outputs on a $(n + \log n)$–dimensional Beneš network. Consider any such algorithm A. Choose uniformly at random an integer r, $0 \quad r < n$. Accept only those requests that are routed by A through any of medium level nodes $r2^n; r2^n + 1; \quad ; (r+1)2^n - 1$ and route them on B_n in the same way as they are routed by A through the $2n$ medium levels of the higher dimensional network. The expected number of calls accepted by the described algorithm is at least $1=(3n)$ of all calls and all of them are routed via edge disjoint paths. □

Because of similarities of hypercube and Beneš network the algorithm can be extended also for the hypercube, at the expense of a higher power of n.

Theorem 5. *There is a randomized $O(n^2)$-competitive algorithm for permanent calls problem on the n-dimensional hypercube.*

Proof. For simplicity a directed hypercube with *doubled* edges HD_n will be considered. However, this assumption will be used in such a way that it may be easily removed by increasing the dimension of the hypercube by one. Consider the following embedding of n-dimensional Beneš network into HD_n: all nodes $(w; i)$, $0 \quad i < 2n$, from raw w are mapped to a node w of the hypercube. Given the embedding, simulate the algorithm for Beneš network on the hypercube. The higher power of n is due to the fact that this algorithm is granted to accept only one call from and to a node, resp., whereas the optimal algorithm might accept up to n calls from and to the node, resp. □

The algorithms can be extended for known limited duration calls [2,13]. The competitive ratio is multiplied by a factor of $\log T$ ($\log T \log^{1+} \log T$, resp.) where T is the known (unknown, resp.) bound on the ration between the minimum and maximum duration of calls.

5 Small Dimension Benes Networks

In this section some new notation is used. Let $B_1(p)$ denotes the Beneš subnetwork B_1 that p is passing through, and $: B_1(p)$ the other B_1 belonging to the same B_2. We say that two paths $p; q$ such that $B_1(p) = B_1(q)$ are *parallel* if they do not have a common vertex in the $B_1(p)$. When it is allowed for two requests to start in the same node then we speak about $2 - 1$ requests. Moreover, when two requests may terminate in the same node they are called $2 - 2$ requests.

Theorem 6. *The Benes network B_2 is a wide sense edge nonblocking network for $2-1$ requests.*

Proof. Our algorithm makes routing decisions according to five rules:

1. Whenever routing a second request through a B_1, do it in a non-parallel way.
2. If there is no path in the B_2 then route the request through the upper B_1.
3. If there is only one path p in the B_2 then route the request through $B_1(p)$ if possible.
4. If there are two paths $p;q$ then route the request through $B_1(p)$ if possible.
5. Suppose there are three paths $p;q;s$ and it is possible to route the request both through the upper B_1 and the lower B_1. Without loss of generality we may assume that $B_1(p) = B_1(q)$. If by routing the request through $B_1(p)$ no new parallel paths would emerge then route the request through the $B_1(p)$, otherwise through $: B_1(p)$.

The rules 2-5 describe decisions about the first step, rule 1 about the second.

Consider a configuration of paths in B_2. If there exist paths $c_1;c_2;c_3$ such that $c_1;c_2$ are parallel, and c_3 could pass in a non-parallel way through $B_1(c_1) = B_1(c_2)$ but does not, we call it a *dead con guration*. Two lemmas will be proved that together show the correctness of the algorithm.

Lemma 3. *If the already established paths are not in a dead con guration, then the algorithm is able to route the next request.*

Lemma 4. *The algorithm never establishes a dead con guration.*

Proof. (Lemma 3) There is always enough space for routing a request if there are at most two established paths. Therefore assume there are three paths $p;q;s$ in the B_2 not in a dead configuration, and a request $a \; ! \; b$ is given. If the input a was not used up to this time, then there is a free path from a to b either through the upper or lower B_1 and the algorithm finds it. The situation is a bit more difficult if a is already used by a path, say by p, since then the request cannot go through $B_1(p)$. If at most one of $q;s$ is passing through $: B_1(p)$, or if both $q;s$ are passing through $: B_1(p)$ and they are not parallel, then there is a free path from a to b through $: B_1(p)$ and the algorithm finds it. The only remaining case is that q and s are parallel and $B_1(p) \neq B_1(q) = B_1(s)$. Assume for a while that q and s are the only paths in B_2 and consider an unused input u. Then u may be connected with exactly one of the two unused outputs by a path passing through $B_1(q)$. Now consider the path p again. From the observation above and from the fact that $p;q;s$ are not in a dead configuration it follows there is free path for the request through $: B_1(p)$ and the algorithm finds it. □

Proof. (Lemma 4) From the description of the algorithm it follows that the property is always satisfied whenever a third path is established. Also deleting any path will not create a dead configuration. Suppose now there are three paths not in a dead configuration and a request $a \; ! \; b$ is given. There are two ways how the "not dead" property may be broken: by establishing the path c_3 or by establishing c_1 or c_2 (i.e. one of the two parallel paths). By rules 1 and 5 we get that no parallel path will emerge if there is a choice for the first edge of a path

from a to b, or if there is no choice for it but there is at most one other path in the B_1 that the request is forced to go through. Neither it happens in these cases that a path c_3 emerges. The only left case is that the request is forced to pass through B_1 with two already established paths in it. Let p be the path that forces the request to do so, that is let p be the path that starts in a. The assumption that the configuration was not dead and the fact that the path p and the request have to use different B_1 yield that neither in this case a dead configuration is created. □

Theorem 7. *The Benes network B_3 is a wide sense edge nonoblocking network for $1-1$ requests.*

Proof. Consider this algorithm. If the endpoint of the request is in the lower half of B_3 then use for the path as the first edge the edge to the lower B_2, otherwise to the upper. By doing so we split the problem into two $2-1$ routing problems on B_2 which we will deal with according to the Theorem 6. □

Corollary 1. *There is a deterministic 3{competitive algorithm for permanent routing problem on a 3{dimensional Benes network.*

Theorem 8. *The Benes network B_3 is a weak sense edge nonoblocking network for $1-1$ requests.*

As we noticed at the beginning this is an immediate consequence of the previous theorem. We state it separately because of a very simple direct proof of it. Moreover, our conjecture is that the algorithm described in this proof works for the Beneš network B_4 as well.

Proof. Consider a greedy algorithm: route the given request through as low medium node as possible. We will show that this algorithm is always able to establish all 8 paths.

Assume there are some paths established and we are trying to establish the next one. There are always two reachable nodes in the first step. These two nodes are connected to four nodes in the next (i.e. second) level. Since we are dealing with B_3, at most one of these four nodes may be blocked by an already established path with a 2–similar input, and at most one of them may be blocked by an already established path with a 2–similar output. So it suffices to show the algorithm is able to do the third step as well. Suppose it is not able to do it. Let u be one of the reachable nodes in second level. Then it is a primarily dead node and one of the outgoing edges from u is blocked by an already established path p (i.e. in_p is 1–similar with a), and the other is going to a node with a path q going through it, such that out_q is 1–similar with b. Either p is going to the upper neighbor and q to the lower, or the other way round. But none of these possibilities may happen, since there would be three 1–similar paths in the same B_1, and thus two of them would be 2–similar, which is a contradiction. □

6 Negative Results

Theorem 9. *The Benes network B_n is not a wide sense edge nonblocking network of order 2^n for $2 - 2$ requests, for n 2.*

Proof. We have to show that there is no algorithm for establishing the paths when the requests are presented on-line. Suppose, by contradiction, there is such an algorithm. We will describe a sequence of requests such that it is unable to establish the edge disjoint paths for them. Let $a; c$ be two different inputs and $b; d; e$ three different outputs. Consider the following sequence of reguests:

1. a ! b, let suppose, w.l.o.g., that it passes through the lower B_{n-1},
2. a ! e, it has to pass through the upper B_{n-1}
3. c ! d, if it passes through the lower B_{n-1}, then there is no place left for request c ! e; if it passes through the upper B_{n-1}, then there is no place left for c ! b.

It is easy to see that on B_1 this argument cannot be used and that B_1 is a wide sense edge nonblocking network of order 2 for $2 - 2$ requests. □

Theorem 10. *The Benes network B_n is not a wide sense node nonblocking network of order 2^n for $1 - 1$ requests, for n 3.*

Proof. Again a sequence of requests will be provided that no deterministic algorithm is able to serve. Consider the following sequence of requests, for $N = 2^n$:

1. a ! b, a $5N/8$, b $3N/4$, let suppose, w.l.o.g., that it passes through the lower B_{n-1},
2. c ! d, $d = b - N/4$, if it would go through the upper B_{n-1} then there is no place left for x ! y, $x = a - N/2$, $y = d - N/2$, so let us suppose it passes through the lower B_{n-1},
3. e ! f, $e = a - N/8$, $f = b - N/2$, it has to go through the upper B_{n-1}, since both nodes in the level $2n - 2$ of the lower B_{n-1}, from which the output f is reachable, are already blocked,
4. g ! h, $g = a - 5N/8 = e - N/2$, $h = d - N/2$ – there is no place left for it.

□

Theorem 11. *The Benes network B_n is not a strict sense edge nonblocking network of order 2^n for $1 - 1$ requests, for n 3.*

Proof. For simplicity the proof is described only for B_3. Consider the following six paths (b_i says whether the i-th edge on the path leads to the upper ($b_i = 0$) or lower ($b_i = 1$) Beneš subnetwork B_{n-i}).

input	1	2	3	4	6	7
output	2	3	6	5	1	4
$b_1b_2b_3$	000	011	010	101	001	111

Then it is impossible to establish an edge disjoint path for request 0 ! 0. □

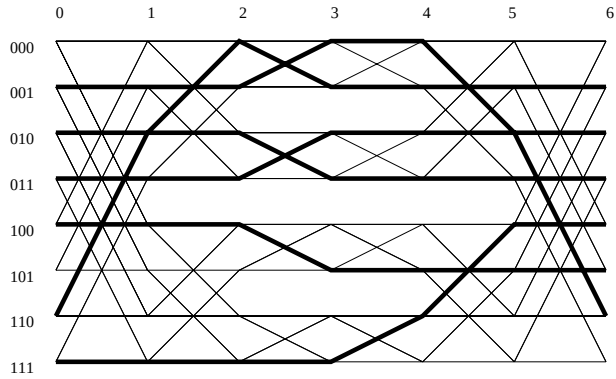


Fig. 2. The Beneš network B_n is *not* a strict sense edge nonblocking network of order 2^n for $1 - 1$ requests. The six described paths disable establishing a path from 0 to 0.

Corollary 2. *The Benes network B_n, $n \quad 3$, is not a strict sense edge nonblocking network of order s, for any $s > \frac{3}{4}\ 2^n$.*

Proof. If we allow more then 6 out of any 8 $(n - 3)$–similar 0–level nodes to be input nodes and more then 6 out of any 8 $(n - 3)$–similar $2n$–level nodes to be output nodes, then the construction from Theorem 11 (or a similar one) shows the network is not a strict sense nonblocking. ∎

7 Open Problems

There are still many open problems about the Beneš network and the hypercube. Let us mention some of them.

- We know B_n is not strict sense nonblocking network of order larger than $3 \quad 2^n/4$, but it is one of order $2^n/n$. Is B_n strict sense nonblocking network of larger order?
- Of what order is Beneš network wide sense nonblocking? Is it higher than for strict sense nonblocking?
- What are the optimal competitive ratios for randomized algorithms for permanent calls on Beneš and hypercube networks?

8 Acknowledgments

The author would like to thank Jiří Sgall for many valuable discussions, and also the anonymous reviewer for pointing to the paper of Melen and Turner [14].

References

1. S. Arora, F. T. Leighton, and B. M. Maggs. On-line algorithms for path selection in a nonblocking network. *SIAM Journal on Computing*, 25(3):600–625, June 1996.
2. B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosén. Competitive non-preemptive call control. In *Proceedings of SODA*, pages 312–320, 1994.
3. B. Awerbuch, R. Gawlick, T. Leighton, and Y. Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proceedings of FOCS*, pages 412–423, 1994.
4. L. A. Bassalygo and M. S. Pinsker. Complexity of an optimum nonblocking switching network without reconnections. *Problems of Information Transmission (translated from Problemy Peredachi Informatsii (Russian))*, 9:64–66, 1974.
5. B. Beizer. The analysis and synthesis of signal switching networks. In *Proceedings of the Symposium on Mathematical Theory of Automata*, pages 563–576. Brooklyn, NY, Brooklyn Polytechnic Institute, 1962.
6. V. E. Beneš. Permutation groups, complexes and rearrangeable connecting network. *The Bell System Technical Journal*, 43, 4:1619–1640, 1964.
7. D. G. Cantor. On non-blocking switching networks. *Networks*, 1:367–377, 1972.
8. P. Feldman, J. Friedman, and N. Pippenger. Wide-sense nonblocking networks. *SIAM Journal on Discrete Mathematics*, 1, 1988.
9. J. A. Garay, I. S. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient on-line call control algorithms. In *Proceedings of the 2nd Israeli Symposium on Theory of Computing and Systems*, pages 285–293, 1993.
10. J. Kleinberg and R. Rubinfeld. Short paths in expander graphs. In *Proceedings of FOCS*, pages 86–95, 1996.
11. J. Kleinberg and É. Tardos. Approximations for the disjoint paths problem in high-diameter planar networks. In *Proceedings of STOC*, pages 26–35, 1995.
12. F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays - Trees - Hypercubes*. Morgan Kaufmann, San Mateo, 1992.
13. R. J. Lipton and A. Tomkins. Online interval scheduling. In *Proceedings of SODA*, pages 302–311, 1994.
14. R. Melen and J. S. Turner. Nonblocking multirate networks. *SIAM Journal on Computing*, 18(2):301–313, 1989.
15. C. E. Shannon. Memory requirements in a telephone exchange. *Bell Syst. Tech. Journal*, 29:343–349, 1950.
16. J. Turner and N. Yamanaka. Architectural choices in large scale atm switches. Technical Report WUCS-97-21, Department of Computer Science, Washington University Saint Louis, 1997.
17. A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, January 1968.

Adaptability and the Usefulness of Hints

(Extended Abstract)

Piotr Berman[1] and Juan A. Garay[2?]

[1] The Pennsylvania State University, Computer Science Department
Pond Lab, University Park 16802, USA
berman@cse.psu.edu
[2] Information Sciences Research Center, Bell Laboratories
600 Mountain Ave, Murray Hill, NJ 07974, USA
garay@research.bell-labs.com

Abstract. In this paper we study the problem of designing algorithms in situations where there is some information concerning the typical conditions that are encountered when the respective problem is solved. The basic goal is to assure efficient performance in the typical case, while satisfying the correctness requirements in every case. We introduce *adaptability*, a new measure for the quality of an algorithm, which generalizes competitive analysis and related frameworks. This new notion applies to sequential, parallel and distributed algorithms alike.

In a nutshell, a "hint" function conveys certain information about the environment in which the algorithm operates. Adaptability compares the performance of the algorithm against the "specialist"—an algorithm specifically tuned to the particular hint value. From this perspective, finding that no single algorithm can adapt to all possible hint values is not necessarily a negative result, provided that a family of specialists can be constructed.

Our case studies provide examples of both kinds. We first consider the "ancient" problem of on-line scheduling of jobs in m identical processors, and show that no algorithm can fully adapt to the natural hint of largest job size. In particular, for the cases $m = 2; 3$, we present specialists that beat the general lower bound for on-line makespan.

In the domain of distributed computing, we analyze the Distributed Consensus problem under several hint functions. To fulfill the requirements of one of the cases we consider, we present the first consensus algorithm that is simultaneously optimal in number of processors, early-stopping property (that is, it runs in time proportional to the actual number of faults), and total number of communicated bits. In our new parlance, the algorithm adapts to the number of faults hint with both running time and communication.

? Work partly done while the author was visiting the Centrum voor Wiskunde en Informatica (CWI), Amsterdam, NL, and at the IBM T.J. Watson Research Center, Yorktown Heights.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 271–282, 1998.
c Springer-Verlag Berlin Heidelberg 1998

1 Introduction

In this paper we study the problem of designing algorithms in situations where there is some information concerning the typical conditions which are encountered when the respective problem is solved. The basic goal is to assure a correct behavior in *every* case, and to guarantee efficient performance in the *typical* case. We introduce *adaptability*, a new measure for the quality of an algorithm, which generalizes previous ones. In particular, the notion extends the "competitive throughput" formulation for distributed algorithms of Aspnes and Waarts [4]. As illustrated by our case studies, the notion applies to sequential algorithms as well, where it measures how well a given algorithm takes advantage of the special cases which a designer or user may regard as important.

While the formulation of adaptability is somewhat more complex, we feel that it provides a useful tool for algorithm design and evaluation in situations where the traditional worst case analysis is inadequate. In particular, we want to construct a measure of quality that would evaluate as optimal the algorithm which performs well in practice, while pin-pointing the situations where no generally optimal algorithm exists (and consequently that a *specialized* algorithm should be chosen).

The earliest rigorous study of this phenomenon is due to Sleator and Tarjan [21] for the case of search trees. If we build a search tree for n objects, we cannot guarantee to answer the queries in time better than $\log n$, while a perfectly balanced tree achieves this bound. However, if we knew the distribution of the queries in advance, we could build an optimum tree for that particular distribution, and in extremal cases achieve $O(1)$ average time per query. Therefore, a static search tree, even if optimal with respect to the worst-case performance, may be $\log n$ times worse than a specialized competitor. The question studied by Sleator and Tarjan was: Can a single algorithm compete with any specialized search tree without knowing the distribution of the queries? As we all know, they answered this question in the positive.

In the domain of distributed computing, a similar question of competitiveness arose for consensus protocols [20]. A protocol that tolerates the presence of t faults must, in the worst case, run for $t + 1$ rounds [14]. While in ordinary applications very few faults are encountered most of the time, the system designer may need to set the bound t on the maximum tolerated number of faults high as a form of assurance. Therefore, the question was whether we could have the best possible assurance against faults ($\lfloor (n-1)/3 \rfloor$ out of n), and yet have very fast runs in the average cases. Again, algorithms were found that have a maximum tolerance threshold and use $\min(f+2, t+1)$ rounds, where $f \leq t$ is the actual number of faults that occur in a run (e.g., [12,8]). Such algorithms have been called *early-stopping*.

Both examples above have the following common thread: the *usual* conditions experienced by the algorithm in a given application have some common characteristic. Given a *hint*, i.e., the value of this characteristic, one may provide an algorithm that performs optimally when the hint is true, and which under any conditions performs correctly. We may say that this algorithm is a *specialist*.

Given a problem, the question is whether we need a specialist, or whether there exists an optimal *generalist*, i.e., an algorithm that is able to match (up to a constant factor) the specialists' performance for every value of the hint. In the former case, we should investigate what the value of the hint is in order to choose the specialist properly; in the latter, we can rely on the generalist to *adapt* to the conditions that the hint would describe without any additional help.

Before we present our quality measure—the *adaptability* of an algorithm—more formally, we explain why we feel that the existing measures do not serve our stated goals. It is well known that worst-case performance does not capture the adaptive properties of algorithms at all. The "classical" *competitive analysis* [21,18] is essentially worst-case analysis applied to the case of on-line computations. On the other hand, average-case analysis requires to make a guess, sometimes very questionable, concerning the probability distribution of the request sequence.

This is particularly true in the area of distributed computing, where the environment in which an algorithm operates has numerous characteristics, such as the pattern of delays (in asynchronous settings), the pattern of faults, the schedule of the outside inputs, etc. Typically, the task of a distributed algorithm is conceptually simple, and the whole point is to be able to cope with the unpredictability of its environment. This unpredictability makes it difficult to define the notion of "average" in a satisfying manner. Therefore, rather than trying to characterize the average behavior, it is more natural to compare the algorithm being evaluated to possible competitors which somehow "know" certain characteristics of the environment. There were already several papers analyzing on-line and distributed algorithms in this fashion. In this paper we want to continue this approach and to offer a new concept that would make it easier to compare and to discuss distinct results.

The definition of *competitive latency* of a distributed algorithm of Ajtai *et al.* [2] looks rather close to our approach, but it is unclear whether it yields any more information than worst-case analysis. In particular, they found that their algorithm for the *cooperative collect* problem is better than previous ones by a factor $n^{1=2}=\log^2 n$ using the competitive latency measure, and by the same factor if they use the worst-case work measure. On the other hand, the later work on the same topic by Aspnes and Waarts [4] shows the advantages of the evaluated algorithm that would be missed by worst-case analysis. In terms of our framework, part of the reason is in the different choice of the hint function. One of our objections to their framework is that it delegates to fine print one of the most important aspects of the analysis, thus making the comparison between different results more difficult.

Our definition follows more closely the competitive analysis of paging done by Borodin *et al.* [9]. Their conceptual framework clearly helped in obtaining illuminating results about paging algorithms that could not be obtained through worst-case analysis. In our framework, we would say that they analyzed the role of the hint provided by the *access graph*, in which the pages that can be requested constitute the nodes and the pairs of pages that can be requested consecutively

are the edges. The access graph helps to capture the notion of locality of reference. Our objection to this framework is the choice of the optimality measure, which leads to a cumbersome formulation of the results. Borodin *et al.* clearly study the *ratio* of the performance of the evaluated algorithm to the performance of benchmark algorithms (which we call the "specialists"), and they express all their results and conjectures as ratios of two measures. Therefore, we feel that it would be more appropriate to use this ratio as the measure.

Another related work is that of Azar, Broder and Manasse [3], who investigated metrical tasks systems with the set of all possible inputs being partitioned into subsets, each with a specialized algorithm. They proposed a general method for switching between specialized algorithms on the fly, and used the specialized algorithms to define a "generic" algorithm with a competitive ratio that depends on the quality of the specialized algorithms and their number.

Our results. We now summarize the contributions of this paper.

- We introduce *adaptability*, a new notion for evaluating the quality of algorithms which generalizes previous ones. We present the framework formally in Section 2. We also show-case the notion in two different domains:
- We consider the "ancient" problem of scheduling jobs in m identical machines [15]. The hint function we consider is the largest size of a job that will be submitted, and we show that no generalist can *fully* adapt (i.e., match the performance of the specialists up to the constant factors). The proof is constructive, and consists of presenting two specialists for the cases $m = 2; 3$ that beat the general lower bound. These results are presented in Section 3.
- We analyze the Distributed Consensus problem [20] under several natural hint functions. In particular, for the function f—the *actual* number of faults that occur in a run of the algorithm—we provide the first algorithm (generalist) that adapts to f with both running time and communication. These results are presented in Section 4.

2 Adaptability

In general, we consider the problem of achieving some goal in environments from a set E.[1] An algorithm is *valid* if it achieves this goal in every environment $E \in \mathcal{E}$. A *hint* function h maps E into some arbitrary set H. For every valid algorithm A and environment $E \in \mathcal{E}$, a run of A in E has some cost, denoted $\mathrm{cost}(A; E)$. Our basic objective is to minimize this cost.[2]

[1] We do not make any distinctions between the input to a problem, or request sequence, and the remaining conditions that might affect a run of the algorithm (for example, the schedule of processes in an asynchronous distributed computing application—cf. [2,4]). An environment $E \in \mathcal{E}$ describes everything that affects the run.

[2] The notion of adaptability can be readily extended to cases where the objective is to maximize the profit accrued by the algorithm, rather than to minimize the cost it incurs. This is done by replacing the maxima by minima, and reversing the fraction. In this paper we focus on cost measures only.

Given a hint value v, we define the worst-case cost of the algorithm restricted to v as $\text{cost}_v(A) \stackrel{\text{def}}{=} \max_{E \in h^{-1}(v)} \text{cost}(A, E)$. Then the *adaptability of A to a hint value $h(E) = v$* is

$$a_A(h = v) \stackrel{\text{def}}{=} \max_{\text{valid algorithm } B} \frac{\text{cost}_v(A)}{\text{cost}_v(B)} .$$

A *specialist* for hint value v is an algorithm $S_{h=v}$ that is "uniquely adapted to v," i.e., $a_{S_v}(h = v) = O(1)$. The *adaptability of A to hint h* is

$$a_A(h) \stackrel{\text{def}}{=} \max_{v \in H} a_A(h = v) .$$

Intuitively, the hint h characterizes environments that are either typical, or critical (for example, we may need to minimize the response time of an algorithm only in critical cases). Knowing the hint v, we might be able to find a specialized algorithm S_v which minimizes the worst-case cost in the set of environments indicated by v. (Note that because the specialized algorithm must be correct even when $h(E) \neq v$, the hint is not an input to the algorithm in the conventional sense.) We evaluate how algorithm A compares against such specialists. If the adaptability of A with respect to h is $O(1)$, we say that *A adapts to h*. (Sometimes we will refer to such an algorithm as the *generalist*.)

In the full paper we express the examples mentioned in the Introduction in terms of adaptability, to show that they can be presented in a uniform manner. Adaptability can also be used in conjunction with the probabilistic analysis of algorithms (e.g., [19]).

3 Adaptability of an Ancient Scheduling Problem

One of the oldest problems studied in the area of scheduling is the case when a sequence of jobs $(1, \ldots, n)$ arrive to m identical machines, where job j has size s_j. As soon as a job arrives, the scheduler assigns it to a machine (later, a job is never reassigned to another machine). The goal is to minimize the *makespan*, i.e., the maximum sum of job sizes assigned to a single machine (we say that such a sum is the *load* of a machine). A very simple algorithm called *List*—always schedule on the currently least loaded machine—achieves a competitive ratio of $2 - 1/m$ [15]. Because Graham published this result in 1966, in recent papers the problem was dubbed "an ancient scheduling problem," which we will abbreviate as ASP. Although in recent years this algorithm was improved several times, [6,17,1], *List* is provably optimal for $m = 2, 3$[13].

The hint function that we will consider is the *largest size of a job* that will be submitted. In real-life situations we often have at least an approximated knowledge of this nature, which makes this hint quite natural. Because even the simple algorithm achieves a constant competitive ratio, the question of adaptability is trivial if we discuss it in the asymptotic sense. However, one may ask whether a generalist algorithm can *fully* adapt, i.e., match the performance of specialist algorithms up to the constant factors. This question is answered negatively by the following theorem.

Theorem 1. *Let the cost of a run of an ASP algorithm be the* makespan*, let M be the maximum job size that is submitted in a given run, and let* m *be the number of machines. Then for every ASP algorithm A*

$$a_A(M) \quad \frac{3=2}{4=3} \quad \textit{for } m = 2 \quad \textit{and} \quad a_A(M) \quad \frac{5=3}{1{:}475} \quad \textit{for } m = 3{:}$$

The M-hint (maximum job size) will be used by our specialized algorithms in the following manner. First, we can rescale the job sizes so that $M = 1$. Next, because we know that a job of size 1 will eventually arrive, we may schedule such a job as the first step of our algorithm. Later, when the first job of size 1 arrives, we can place it on the machine on which it is already scheduled. Thus we can analyze our algorithm under the assumption that the first job has size 1 and no larger job will ever arrive.

Before we proceed we will formulate a lemma which will be used several times and which holds for $m = 2; 3$.

Lemma 1. *Assume that at some point in time the scheduling algorithm has the maximum load of a machine equal to l 1. Assume also that from this point onwards the algorithm behaves exactly like algorithm List. Then starting from the rst time when the makespan increases again, the ratio of the obtained makespan to the optimum one is at most $m(l + 1)=(ml + 1)$.*

The proof of the lemma is entirely based on the observation that the optimum makespan is at least as large as the average of the machine loads. The details will be provided in the full version.

The algorithm for two machines is to simply run *List*. Right at the beginning the obtained makespan equals 1 and as long as it stays unchanged, it is optimal because it is equal to the largest job size. After it increases, by Lemma 1 the ratio of the obtained makespan to the optimal one is at most $2(1+1)/(2+1) = 4/3$. Therefore we can use the modified *List* as the optimum algorithm B in the definition of adaptability function $a_A(M)$. Note that the ratio of $4/3$ is optimum, because we can face the scheduler with the choice of the following two sequences of the job sizes: (1, 1/2, 1/2) and (1, 1/2, 1/2, 1). Either the scheduler will obtain the makespan of 3/2 when 1 is possible, or 2 when 3/2 is possible.

The same algorithm and argument can be applied to three machines where it yields the ratio of $3(1+1)/(3+1) = 3/2$. However, this ratio is not optimal (but already better than the ratio of 5/3 that is achievable without the hint). One can obtain a ratio of $r = (\sqrt{97} - 1)=6$ 1.475 using a somewhat more complicated algorithm and much more complicated argument. This algorithm has two phases. In the first, which we may call first fit, machines have fixed numbers 1, 2 and 3. When a new job of size s arrives, we schedule it on the first machine that had load l satisfying $l + s$ r. Once it is not possible anymore, we enter the second phase, when we simply run *List*.

To prove the ratio we start with two observations. In the first fit phase ratio r is assured, as the makespan is bounded by r and the largest job offers the lower bound of 1. Suppose that the ratio of obtained makespan to the optimum

is at most r after scheduling the first job in the *List* phase. Then we obtain the highest load of at least 1.47; by Lemma 1 this assures that after the change of the makespan the ratio will never exceed 1.24. Therefore it suffices to consider what happens when the first job is scheduled after the first fit phase.

For that part of the reasoning we define $l_0 = (\sqrt{97} + 3)/12$. We consider two cases according to the highest machine load machine that occurs during the first fit phase. If this load is higher (or equal) than l_0, the claim about ratio r will follow from Lemma 1, because $3(l_0 + 1)/(3l_0 + 1) = r$. If on the other hand this load is lower than l_0, than the first makespan in the *List* phase is less than $l_0 + 1$. On the other hand, all jobs on machines 2 and 3 exceed $r - l_0$; in an optimal schedule one of these jobs must be scheduled on the same machine as a job of size 1; this leads to a lower bound of $1 + r - l_0$ and the ratio estimate $(2 + l_0)/(1 + r - l_0) = r$. Our choice of the values of r and l_0 comes from solving the two equations that were used in this proof.

4 Adaptability of Distributed Consensus

As another case study of the notion of adaptability, we analyze the Distributed Consensus problem (a.k.a. Byzantine agreement) [20] under several natural hint functions. In particular, for the function f—the actual number of faults that occur in a run of the algorithm—we provide an algorithm that adapts with both running time and communication. We first define the problem formally. Given are n processors, at most t of which might be faulty. Each processor i has an initial value $v_i \in \{0, 1\}$. Required is an algorithm with the following properties:

Decision: Every non-faulty processor eventually decides on a value from $\{0, 1\}$.
Agreement: The non-faulty processors all decide on the same value.
Validity: If the initial values v_i of all nonfaulty processors are identical, then every nonfaulty processor decides on v_i.

We assume the standard model for this problem, in which processors may fail in arbitrarily malicious ways [20]. Each processor can communicate directly with every other processor via a reliable point-to-point channel. The processors are synchronous, and their communication proceeds in synchronous rounds: in a round, a processor can send one message to each of the other processors, and all messages are received in the round in which they are sent. Finally, it must be the case that $n > 3t$ for the problem to have a solution [20].

An environment in which a consensus protocol runs has several characteristics, including the sets of correct and faulty processors—P_C and P_F, respectively—and the sets of correct processors with initial values 0 and 1—P_0 and P_1, respectively. We consider the following hint functions for this problem:

- The actual number of faults that occur in a run of the algorithm, $f \stackrel{\text{def}}{=} \#P_F$;
- the *discrepancy* in the initial values $\stackrel{\text{def}}{=} \#P_1 - \#P_0$;
- the absolute value of the discrepancy, $|$ $|$;

- a subset of processors containing correct processors, i.e., $S \quad P$ such that $\#S = k$ and $S \setminus P_C \neq ;$.

We also consider two cost functions: the number of communication *rounds*, and the total number of *bits* transmitted. In Section 4.1 we present the first consensus algorithm for arbitrary failures that adapts to f with number of rounds and bits *simultaneously.* But before we provide a brief explanation of our results for the remaining hints; due to space limitations, a complete treatment is deferred to the full version of the paper.

Hint captures situations, common in practice, in which either one input value is distinctly more frequent, or when the performance is critical only if one of the two values predominates (typically, in such situations one decision value amounts to performing an action, such as "abort," and the other to refrain from it). We show that a consensus algorithm cannot adapt to with rounds, and that it suffices to choose one of just two specialized algorithms. Hint $j \ j$ captures situations when in a typical case correct processors start with almost unanimous preferences. We show that if the total number of processors exceeds $4t$, then adaptability with rounds is possible; in other cases, the problem remains open. Even then this hint is valuable, in the sense that certain design choices do not affect the worst case, but preclude adaptability to $j \ j$. The last hint corresponds to the cases where some processors are more reliable than others. We show that adaptability is not possible, but that one can construct a single family of specialists.

4.1 The actual number of faults hint

In this section we consider a hint function that specifies the actual number of faults f ($f \quad t$) that occur in a run of the consensus algorithm.

Recall that a lower bound of $t + 1$ rounds of communication has been established for this problem in the worst case [14]. Thus, overcoming potential faults is expensive, even though, in practice, they are rarely observed. This led researchers to explore ways in which the price of fault tolerance may be reduced. Dolev, Reischuk, and Strong [12] demonstrated that by permitting processors to stop at different times (instead of simultaneously), early-stopping consensus algorithms are possible that run in time proportional to f (specifically, $\min(f + 2; t + 1)$ rounds). That is, in our new parlance,

Theorem 2. [12,8] *Let the cost of a consensus algorithm be the* number of rounds*, and f the actual number of faulty processors. Then there exists an algorithm A such that $a_A(f) = O(1)$ (i.e., A adapts to f).*

Hadzilacos and Halpern [16] concentrated on runs that are failure-free, and presented consensus algorithms with optimal number of messages for these runs. Which brings us to the other cost measure: the communication costs of such protocols, given by the total number of transmitted bits. Dolev and Reischuk [11] showed a lower bound of $\Omega(nt)$ for this measure, a bound that holds for *every* run of a consensus algorithm. Moreover, algorithms have been developed that match this bound [7,10]. Intuitively, f is useless for this measure: any bit-optimal algorithm adapts to this hint—in the trivial sense. This immediately yields

Theorem 3. *Let the cost of a consensus algorithm be the* total number of transmitted bits*, and f be the actual number of faulty processors. Then there exists an algorithm A such that $a_A(f) = O(1)$.*

Given Theorems 2 and 3, it is natural to ask whether a consensus algorithm can adapt to f with both cost measures simultaneously. In the next section we answer this question in the positive.

Simultaneous adaptability. As mentioned above, existing early-stopping consensus algorithms adapt to f with rounds, but not with communication, as a total of $\Omega(n^2t)$ bits are transmitted. On the other hand, there exist algorithms with $O(nt)$ communication, which satisfy Theorem 3 [7,10]; the running time of such algorithms, however, is proportional to t, the maximum number of tolerable faults. We now present a new consensus algorithm that is able to adapt to f with both cost measures, i.e., it runs for $O(f)$ rounds while communicating a total of $O(nt)$ bits. We call this algorithm SA, for *Simultaneously-Adaptive*. Although the main point of this paper *is* analysis, due to space limitations we include in the paper only the description of the algorithm; its "code," analysis and proof of correctness are deferred to the full paper.

The new algorithm is based on the *phase king* paradigm for consensus of [8]. In such a paradigm, the computation consists of a series of phases, each consisting of possibly several rounds; in each phase, a distinguished processor(s) takes charge of the computation. SA is defined recursively. [3] For simplicity, we assume in this section that the total number of processors $n = 3t + 1$.

Recursive instance $\mathsf{SA}(k; P_k; t_k)$ is executed by a set P_k of at least $3t_k + 1$ processors, which learn of their initial "preferences" (initial values for this instance) before communication round k; the top instance is $\mathsf{SA}(1; P; t)$, where $P = P_1$ is the original set of processors. Assuming that at most t_k of these processors are faulty, the protocol assures that before round $k + r(t_k)$ all the correct processors will have the same final preference for this instance. Here r is a fixed function, $r(m) = O(m)$, that gives the (worst-case) number of rounds of the protocol. Additionally, if the correct processors are unanimous in their initial preferences, their final preferences will be the same.

Moreover, the protocol has the following early-stopping property: If the number of faulty processors in P_k is $f_k \quad t_k$, then the correct processors will know their final preferences by the end of round $k + r_{es}(f_k)$, where $r_{es}(m) = O(m)$ is a fixed function that estimates the number of rounds when an *early* decision (and *stopping*) can be guaranteed.

We now turn to a more detailed description of the algorithm.

Description of algorithm SA. The variables that the processors store and communicate are of the form $< v_p; \quad >$, where $v_p \ 2 \ f0; 1g$ is processor p's current preference (initially, v_p is p's initial value), and predicate $\quad 2 \ f0; 1; 2g$ represents

[3] Indeed, given the communication bound that we are trying to achieve, message exchanges must be restricted to hierarchically defined subsets of processors. However, we remark that special precaution must be taken to prevent a single faulty processor from delaying termination for a number of rounds equal to the depth of the recursion.

the strength of that preference (initially, $= 0$). In every case, to define protocol instance $SA(k; P_k; t_k)$ we select some p_k *2* P_k to be the *king of* P_k.

If $t_k = 0$, then we may assume that $P_k = fp_kg$. The conditional correctness assumes that p_k is correct. Obviously, a satisfactory protocol consists of p_k computing its final preference as equal to the initial one. Therefore $r(0) = 0$. If $t_k = 1$, then we may assume that P_k has four elements; in this case two rounds of the "Exponential Information Gathering" (EIG) algorithm of [5][4] define a satisfactory protocol. As a result, $r(1) = 2$ and $r_{es}(0)$ 2.

For t_k 2 we define the protocol recursively. Let a and b be equal or almost equal integers such that $t_k = a + b + 2$. Let P_{k+5} and $P_{k+r(a)+8}$ be two disjoint subsets of P_k − fp_kg with $3a + 1$ and $3b + 1$ members, respectively; we will refer to these sets as the *rst committee* and the *second committee.* The protocol consists of **three phases**: the king's phase and the phases of the two committees, and describes the actions in nine *logical* rounds. The actions of a single logical round would take place in a single communication round if the processors did not apply early stopping rules; because they do, different processors of P_k can execute them at different times. Indeed, the execution of the protocol effectively becomes asynchronous, as at a given point different processors may be executing different levels of the recursion. We show in the full paper that this can be handled without any message indexing, and thus asynchrony does not add to the communication overhead.

The first three logical rounds form the king's phase. In round 1, processors communicate their initial preference. It is possible, however, that a processor has announced its initial preference as part of some earlier message; this assumes that this processor concluded that the correct processors of P_k would be unanimous in their initial preferences. In any case, this action takes place in communication round k at the latest.

As soon as a processor knows that in round 1 $2t_k + 1$ processors of P_k have announced their preferences to be v, it sets its next preference to v and sends $(v; 1)$ as its message of round 2; the meaning of this message is "I **strongly** prefer v". Otherwise, if communication round k was completed and this condition is still not satisfied, the processor sends $(0; 0)$, meaning "I *weakly* prefer 0". At the latest, this action takes place in communication round $k + 1$.

As soon as a processor knows that in round 2 $2t_k + 1$ processors of P_k have announced that they strongly prefer v, it sets its next preference to v and sends $(v; 1)$ as its message of round 3. Otherwise, if communication round $k + 1$ is completed, the processor checks if it received at least $t_k + 1$ strong preferences for some value v. If yes, it sends $(v; 0)$ as a message or round 3, otherwise it sends $(0; 0)$. At the latest, this action takes place in communication round $k+2$.

As soon as a processor knows that in round 3 $2t_k + 1$ processors of P_k have announced that they strongly prefer v, it sends $(v; 2)$ as its message of round 4. This message means "I **very strongly** prefer v"; in other words: "I will prefer v, with the maximum strength possible, through the duration of this

[4] EIG is basically a full information protocol in which each processor broadcasts its value, and then every processor echoes the values they received in the previous round.

protocol and all its recursive calls; I will send no more messages in this protocol." Otherwise it computes its initial preference for the phase of the first committee when communication round $k+2$ is completed. If it learns that t_k+1 processors strongly prefer v, it will prefer v as well. If this holds neither for $v=0$ nor for $v=1$, then the processor takes its preference from the message received from the king, i.e., from p_k.

The phase of the first committee is similar in spirit, but there are several differences. The logical rounds 4, 5, and 6 take place at the latest in communication rounds $k+3$, $k+4$, and $k+5$, respectively. The role of the king is played by the first committee running instance $\mathsf{SA}(k+5; P_{k+5}; t_{k+5}=a)$. The initial preferences for this run are the preferences contained in the messages of round 5. The subsequent communications of this run take place within P_{k+5}, and as soon as a member of P_{k+5} knows its final preference for this run, it sends it to the members of $P_k - P_{k+5}$. Members of P_{k+5} know the preference of the first committee as soon as this run concludes, i.e., before communication round $k+5+r(t_k+5)$; members of $P_k - P_{k+5}$ know this preference when they receive $t_{k+5}+1$ announcements from P_{k+5} about the same value. This should happen at the latest when communication round $k+5+r(a)$ is completed.

The phase of the second committee is almost identical to that of the first committee. The logical rounds 7, 8, and 9 take place at the latest in communication rounds $k+6+r(a)$, $k+7+r(a)$ and $k+8+r(a)$, respectively. The final preferences are computed at the latest when communication round $k+8+r(a)+r(b)$ is completed. Because this is the last phase, processors do not send messages about their very strong preferences (these messages are used to execute an early stop, so they are useless at the very end).

The net result is

Theorem 4. *Consensus algorithm* SA *adapts to the actual number of faults* f *with number of communication rounds and total communicated bits, while tolerating the maximum possible number of faults* ($t < n/3$).

Roughly, a total of $O(nt)$ bits are communicated since the total number of bits sent in SA satisfies the recurrence $B(n) < O(n^2) + 2B(\frac{n}{2})$, and $n = O(t)$. The adaptability with runs follows from the fact that if in a run of $\mathsf{SA}(k; P_k; t_k)$ $f_k \quad t_k$ of the processors are faulty, then all the correct processors of P_k will reach the same final decision, equal to their initial prefence if they started unanimously, before communication round $k+r(t_k)+1$ and before round $k+6f_k+1$. Correctness follows from the fact that correct processors in an instance are unanimous after a king is correct, and this is guaranteed by the way the sets are partitioned.

Acknowledgments

We are thankful to Jim Aspnes and Amotz Bar-Noy for many useful insights and suggestions.

References

1. S. Albers, Better bounds for online scheduling. *Proc. 29th Annual ACM Symp. on the Theory of Computing*, pp. 130-139, El Paso, TX, May 1997.
2. M. Ajtai, J. Aspnes, C. Dwork, and O.Waarts. A Theory of Competitive Analysis for Distributed Algorithms. *Proc. 33rd IEEE Symp. on the Foundations of Computer Science,* pp. 401-411, November 1994.
3. Y. Azar, A. Broder and M. Manasse. On-line choice of on-line algorithms. *Proc. 4th Annual ACM/SIAM Symp. on Discrete Algorithms*, pp. 432-440, 1993.
4. J. Aspnes and O. Waarts. Modular Competitiveness for Distributed Algorithms. *Proc. 28th Annual ACM Symp. on the Theory of Computing,* pp. 237-246, Philadelphia, PA, May 1996.
5. A. Bar-Noy, D. Dolev, C. Dwork and H.R. Strong. Shifting gears: changing algorithms on the fly to expedite Byzantine Agreement. *Proc. 6th ACM Symposium on the Principles of Distributed Computing,*, pp. 42-51, 1987.
6. Y. Bartal, A. Fiat, H. Karloff and Y. Rabani, New algorithms for the ancient scheduling problem. *Journal of Computer and System Sciences,* 51:359-366, 1995.
7. P. Berman, J. Garay, and K. Perry. Bit Optimal Distributed Consensus. In *Computer Science Research* (ed. R. Yaeza-Bates and U. Manber), Plenum Publishing Corporation, NY, NY, pp. 313-322, 1992.
8. P. Berman, J. Garay, and K. Perry. Optimal Early Stopping in Distributed Consensus. *Proc. 6th International Workshop on Distributed Algorithms*, LNCS (647), Springer-Verlag, pp. 221-237, Haifa, Israel, November 1992.
9. A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive Paging with Locality of Reference. *Proc. 23rd Annual ACM Symp. on the Theory of Computing,* pp. 249-259, New Orleans, Louisiana, May 1991.
10. B. Coan and J. Welch, Modular Construction of an Efficient 1-Bit Byzantine Agreement Protocol. *Mathematical Systems Theory*, special issue on Fault-Tolerant Distributed Algorithms (ed. H.R. Strong), Vol. 26, No. 1 (1993).
11. D. Dolev and R. Reischuk, Bounds of Information Exchange for Byzantine Agreement. *JACM*, Vol. 32, No. 1, pp. 191-204, 1985.
12. D. Dolev, R. Reischuk and H.R. Strong. Early Stopping in Byzantine Agreement. *JACM*, Vol. 37, No. 4 (1990), pp. 720-741.
13. U. Faigle, W. Kern and G. Turan, On the performance of online algorithms for particular problems. *Acta Cybernetica*, 9:107-119, 1989.
14. M. J. Fischer and N. A. Lynch. A lower bound for the time to assure interactive consistency. *Inf. Proc. Letters*, 14 (1982), pp. 183–186.
15. R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal of Applied Mathematics*, 17:263–269, 1969.
16. V. Hadzilacos and J. Halpern. Message-Optimal Protocols for Byzantine Agreement. *Proc. 10th Annual ACM Symp. on the Principles of Distributed Computing*, pp. 309-324, Montreal, Canada, August 1991.
17. D.R. Karger, S.J. Phillips and E. Torng, A better algorithm for an ancient scheduling problem, *Journal of ALgorithms*, 20:400-430, 1996.
18. A. Karlin, M. Manasse, L. Rudolph, and D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 3(1):70-119, 1988.
19. A. Karlin, S. Phillips, and P. Raghavan. Markov Paging. *Proc. 33rd Annual IEEE Symp. on Foundations of Computer Science,* pp. 208-217, 1992.
20. L. Lamport, R.E. Shostak and M. Pease. The Byzantine generals problem. *ACM Trans. Prog. Lang. and Systems*, 4:3 (1982), pp. 382–401.
21. D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 32:652–686, 1985.

Fault-Tolerant Broadcasting in Radio Networks (Extended Abstract)

Evangelos Kranakis[1,3], Danny Krizanc[1,3], and Andrzej Pelc[2,3]

[1] School of Computer Science, Carleton University, Ottawa, Ontario, K1S 5B6, Canada f kranakis,krizanc g@scs.carleton.ca
[2] Département d'Informatique, Université du Québec à Hull, Hull, Québec J8X 3X7, Canada. Andrzej_P elc@uqah:uquebec:ca
[3] Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

Abstract. We consider broadcasting in radio networks that are subject to permanent node failures of unknown location. Nodes are spread in a region in some regular way. We consider two cases: nodes are either situated at integer points of a line or they are situated on the plane, at grid points of a square or hexagonal mesh. Nodes send messages in synchronous time slots. Each node v has a given transmission range of the same radius R. All nodes located within this range can receive messages from v. However, a node situated in the range of two or more nodes that send messages simultaneously, cannot receive these messages and hears only noise. Faulty nodes do not receive or send any messages.
We give broadcasting algorithms whose worst-case running time has optimal order of magnitude, and we prove corresponding lower bounds. In case of nonadaptive algorithms this order of magnitude is $(D + t)$, and for adaptive algorithms it is $(D + \log(\min(R; t)))$, where t is an upper bound on the number of faults in the network and D is the diameter of the fault-free part of the network that can be reached from the source as a result of those faults.

1 Introduction

Radio communication networks have recently received growing attention. This is due to the expanding applications of radio communication, such as cellular phones and wireless local area networks. The relatively low cost of infrastrucure and the flexibility of radio networks make them an attractive alternative to other types of communication media.

A radio network is a collection of transmitter-receiver devices (referred to as *nodes*), located in a geographical region. Nodes send messages in synchronous time slots. Each node v has a given transmission range. All nodes located within this range can receive messages from v. However, a node situated in the range of two or more nodes that send messages simultaneously, cannot receive these messages and hears only noise.

One of the fundamental tasks in network communication is *broadcasting*. One node of the network, called the *source*, has a piece of information which has to

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 283{294, 1998.
c Springer-Verlag Berlin Heidelberg 1998

be transmitted to all other nodes. Remote nodes get the source message via intermediate nodes, in several hops. One of the most important performance parameters of a broadcasting scheme is the total time it uses to inform all nodes of the network.

As the size of radio networks grows, they become increasingly vulnerable to component failures. Some nodes of the network may be faulty. Such nodes do not receive or send any messages. An important feature of a broadcasting algorithm is its capacity to inform all nodes reachable from the source, as fast as possible, in spite of other node failures and without knowing the location of faults.

1.1 Previous work

In most of the research on broadcasting in radio networks [1,3,4] the network is modeled as an undirected graph in which nodes are adjacent if they are in the range of each other. A lot of effort has been devoted to finding good upper and lower bounds on the broadcast time in (fault-free) radio networks represented as arbitrary graphs. In [1] the authors constructed a family of n-node networks of radius 2, for which any broadcast requires time $\Omega(\log^2 n)$, while in [3] it was proved that broadcasting can be done in time $O(D + \log^5 n)$ for any n-node network of diameter D. In [10] the authors restricted attention to communication graphs that can arise from actual geometric locations of nodes in the plane. They proved that scheduling optimal broadcasting is NP-hard even when restricted to such graphs and gave an $O(n \log n)$ algorithm to schedule an optimal broadcast when nodes are situated on a line. [9] is devoted to the study of asymptotically optimal information exchange when nodes are located randomly on a ring.

Very few results are known about broadcasting in radio networks in the presence of faults – in contrast to a plethora of papers on fault-tolerant communication in wired point-to-point networks (see [8] for a survey of this literature). In [6] the issue of reliable radio broadcasting was considered but only transient faults were dealt with. In [5] the authors studied the efficiency of overcoming noise in fully connected networks under the assumption that every transmitted bit can be flipped with some probability. To our knowledge, broadcasting in multihop radio networks with permanent node failures of unknown location has never been studied.

1.2 The model

The aim of this paper is to give fast broadcasting algorithms in radio networks subject to permanent node failures of unknown location. We consider radio networks whose nodes are spread in a region in some regular way. All nodes have distinct identities. Two scenarios are considered. In the first, nodes are situated at integer points of a line and in the second they are situated at grid points of a square or hexagonal mesh. The latter layout has particularly important practical applications as it forms a basis of cellular phone networks [7]. Every node v of the network has the same *range*. This is a segment of length $2R$ in case of the line, a square of side $2R$ in case of the square mesh and a regular hexagon of

side R in case of the hexagonal mesh, in all cases with center at node v. The assumption about receiving capacity of nodes is as described above for general radio networks.

We assume that at most t nodes are faulty. Faults are permanent (i.e., the fault status of a node does not change during the broadcast), and faulty nodes do not send or receive any messages. The location of faults is unknown and it is assumed to be worst-case. Moreover, we do not preclude the possibility that faults may disconnect the radio network, in which case the broadcast message can reach only the fault-free connected component of the source node, called the *domain*. For any configuration of faults, consider the graph G whose vertices are fault-free nodes, and adjacent vertices are those in each other's range. The domain is then the connected component of G containing the source and the *diameter* (denoted by D throughout the paper) is the maximum length of all shortest paths between the source and nodes of the domain. As usual in the case of communication in the presence of faults, it is natural to consider two types of broadcasting algorithms.

In *nonadaptive* algorithms, all transmissions have to be scheduled in advance. Thus every node is provided with a table specifying in which time slots it should transmit. If a given node is faulty, it never transmits, and if a fault-free node is scheduled to transmit before it receives the source message, it transmits a default message. In our algorithms the prescribed periodic behavior of a node does not depend on the length of the line or the size of the mesh, or on the configuration of faults, but only on the label of the node. We do not specify the number of transmissions for each node, treating the broadcasting process as repetitive, designed for possibly many source messages. The *time* of broadcasting a message in the presence of a given configuration of at most t faults is defined as the maximum number of time units between the transmission of this message by the source and the reception of it by a node in the domain.

In *adaptive* algorithms, nodes have the ability to schedule future transmissions on the basis of their communication history. In particular, this enables them to perform a preprocessing phase during which nodes learn the fault status of some other nodes from obtained messages and even from noise. Also in this case, the behavior of a node in our algorithms depends only on the label of the node. Since in adaptive algorithms nodes can send meaningful messages prior to the transmission from the source, the time of broadcasting a message in the presence of a given configuration of at most t faults is now defined as the maximum number of time units between the first transmission by any node and the reception of the source message by a node in the domain. Our adaptive algorithms consist of a preprocessing phase lasting a prescribed amount of time, and the proper broadcasting phase which is message driven: a node transmits only once, after the reception of the source message. The above definition of broadcasting time accounts for preprocessing as well as for proper broadcasting. If many messages are broadcast by the source, preprocessing can be done only once and the actual time for subsequent messages can be reduced to that of the proper broadcasting phase.

1.3 Our results

For all the above scenarios we give broadcasting algorithms whose worst-case time has optimal order of magnitude, and we prove corresponding lower bounds. In case of nonadaptive algorithms this order of magnitude is $(D + t)$, and for adaptive algorithms it is $(D + \log(\min(R; t)))$, where D is the diameter. More precisely, we give algorithms that, for any fault configuration of at most t faults, yielding diameter D, inform the entire domain in time corresponding to the respective upper bound ($O(D + t)$ for nonadaptive and $O(D + \log(\min(R; t)))$ for adaptive algorithms). Also, for any nonadaptive (resp. adaptive) algorithm we show configurations of at most t faults, yielding diameter D, for which this algorithm requires time $\Omega(D + t)$ (resp. $\Omega(D + \log(\min(R; t)))$) to inform the domain. In case of the line we show how the gap between the upper and lower bounds can be further tightened.

The difficulty of designing efficient fault-tolerant algorithms for radio communication lies in the need to overcome the contradictory impact of the power of radio broadcasting and of faults. On the one hand, scheduling simultaneous transmissions of nodes close to each other should be avoided because this will result in noise for many receiving nodes, in case both transmitting nodes are fault free. On the other hand, sparse transmission scheduling (few simultaneous transmissions from neighboring nodes) is dangerous as well: It causes communication delays in cases where nodes scheduled to transmit happen to be faulty.

2 The Line

In this section we consider the case when nodes of the radio network are situated at the points $0; 1; :::; n;$ on a line. We will use the notions "larger" and "smaller" with respect to this ordering. The range of every node v has the same radius R, i.e., it includes nodes $v; v + 1; :::; v + R; v - 1; :::; v - R$. We assume $n > R \quad 2$, since the other cases are trivial. Assume that the source is at node 0 and define the ith *segment* to be $f(i - 1)R + 1; :::; iRg$. m denotes the largest node of the domain and hence the diameter D is $dm=Re$. Clearly m and D depend on a particular fault configuration. Notice that in the special case when $t < R$, the domain consists of all fault-free nodes, regardless of the fault configuration. In this special case we can achieve tighter results than in general. For simplicity assume that R divides n. (Modifications in the general case are straightforward.)

2.1 The nonadaptive case

Number nodes in the ith segment $a_{i1}; :::; a_{iR}$ in decreasing order. Consider the following nonadaptive broadcasting algorithm.

Algorithm Line-NA

1. The source transmits in time 1.
2. Node a_{ij} transmits in time $i + j + c(R + 1)$, for $c = 0; 1; 2; :::$.

Lemma 1. *The algorithm Line-NA informs the domain in time at most* $D + t + dt=Re$.

Proof. First observe that no pair of nodes at distance at most R transmits in the same time slot and that if node i transmits in time x then node $i - 1$ transmits in time $x + 1$. In the absence of faults, R new nodes are informed in every time unit. Every fault can slow information progress in the following two ways: one time unit is lost to wait for $i - 1$ (instead of the faulty i) to transmit, and the largest fault-free informed node is by one closer to the source. This implies that the worst-case effect of t faults is the following. Time $D + t$ is spent to inform nodes in the domain with numbers at most $m - t$. The remaining segment of length t is informed in time dt/Re, for a total of $D + t + dt/Re$.

In order to prove the lower bound we need the following Lemmas. Lemma 2 is proved by induction on t. Lemma 3 follows from Lemma 2. Notice that for $t < R$, the upper bound given by Lemma 1 is $D + t + 1$, which means that algorithm Line-NA is almost the best possible in this case. For larger values of t we get an asymptotic lower bound from Lemma 4.

Lemma 2. *Suppose that the source message is known to a set A of nodes, where jAj t. Any nonadaptive broadcasting algorithm informing a node v outside of A must use time larger than t.*

Lemma 3. *If $t < R$, every nonadaptive broadcasting algorithm must use at least time $D + t$ to inform the domain in the worst case.*

Proof. Suppose that all segments except the pre-last are entirely fault free. Informing all segments except the last requires time $D - 1$. By Lemma 2 informing the last segment requires time at least $t + 1$, for a total of at least $D + t$.

Lemma 4. *Every nonadaptive broadcasting algorithm must use time $D + \Omega(t)$ to inform the domain, in the worst case.*

Proof. Fix a nonadaptive broadcasting algorithm. Let $t = a(R - 1) + b$, where $b < R - 1$. Let the even-numbered segments be fault free and allocate $R - 1$ faults to the first a odd-numbered segments, and b faults to the $(a + 1)$th odd-numbered segment. The adversary can distribute faults in these segments to make the algorithm spend at least time R in the first a odd-numbered segments and at least time $b + 1$ in the $(a + 1)$th odd-numbered segment (using Lemma 2). Notice that the presence of a fault-free node in each odd-numbered segment (guaranteed by the bound $R - 1$ on the number of allocated faults) implies that the domain consists of all fault-free nodes. The time spent to inform an odd-numbered segment, if the preceding (fault-free) segment is informed, is 1. There are at least $D/2$ odd-numbered segments. Hence the total time used by the algorithm is $D + t$, if $t \quad D(R - 1)/2$, and $D + D(R - 1)/2$, otherwise. Since $DR = n$ and $R \quad 2$, we have $D(R - 1)/2 \quad DR/4 = n/4 \quad t/4$.

Theorem 1. *The worst-case minimum time to inform the domain on the line by a nonadaptive broadcasting algorithm is $D +$ (t). Algorithm Line-NA achieves this performance.*

2.2 The adaptive case

The flexibility given by the adaptive scenario permits us to perform preprocessing after which specific fault-free nodes are elected as transmitters of information. This enables us to subsequently conduct the broadcasting itself much faster than in the nonadaptive case. We will show how to do the preprocessing in time logarithmic in the number of faults, thus considerably increasing the efficiency of the entire process.

We start with the description of a procedure that elects the largest fault-free node in a segment. First suppose that $t < R$ and consider a segment of length R of the line of nodes. Let A be the set consisting of $t+1$ largest nodes in this segment. Hence A contains at least one fault-free node.

The Procedure Binary Elect works as follows. All nodes in A are initialized to the status *active*. ($ACT := A$). After $\lceil \log(t+1) \rceil$ steps, exactly one node remains active (ACT has one element). This is the largest fault-free node in the segment. At any time unit $i \quad \lceil \log(t+1) \rceil$ the set ACT is divided into the larger half L and the smaller half S. All (fault-free) nodes in L transmit a message. If something is heard (a message or noise) then $ACT := L$ and nodes in S never transmit again. Otherwise (i.e., when all nodes in L are faulty), $ACT := S$.

Since all nodes are in the range of one another, this can be done distributively, and every node knows its status at every step and knows whether it should transmit or not. At the end of the procedure, when a single node remains active, this node broadcasts its identity and all other nodes in the segment learn it.

Procedure Binary Elect works in time $O(\log t)$. Clearly, an analogous procedure may be used to find the smallest fault-free node in a segment, and a multiple use of this procedure enables us to find the k largest fault-free nodes.

Note that in order to avoid conflicts, searching for the largest and for the smallest fault-free nodes in the same segment and searching for such nodes in adjacent segments must be performed at different times. This yields the following adaptive broadcasting algorithm. (We assume that each node sending a message appends its identity to the message, so all nodes in its range know who is the sender.)

Algorithm Line-ADA

1. **for all** even i **in parallel do**
   find the largest fault-free node l_i in the ith segment;
   find the smallest fault-free node s_i in the ith segment;
   for all odd i **in parallel do**
   find the largest fault-free node l_i in the ith segment;
   find the smallest fault-free node s_i in the ith segment;
2. the source tranmits the message;
   if the message arrived from the source in time r
   then l_1 transmits in time $r+1$.
   if the message arrived from l_i in time r
   then s_{i+1} transmits in time $r+1$.
   if the message arrived from s_i in time r
   then l_i transmits in time $r+1$.

Notice that after step 1, nodes s_i and l_i know each other, and l_i and s_{i+1} know each other, for any i, if they are in the domain. (If the distance between l_i and s_{i+1} exceeded R, they could not be both in the domain because these are consecutive fault-free nodes.) This guarantees that in step 2 exactly one node transmits in any time unit. Preprocessing (step 1) takes time $O(\log t)$. In step 2 two time units are spent in every segment. Hence we get the following lemma.

Lemma 5. *If $t < R$, algorithm Line-ADA informs the domain in time $2D + O(\log t)$.*

The lower bound for the adaptive case is based on Lemma 6 which also yields the lower bound of Lemma 7 in the case when $t < R$.

Lemma 6. *Suppose that the source message is known to a set A of nodes, where $|A| \geq t$. Any adaptive broadcasting algorithm informing a node v outside of A must use time larger than $\lfloor \log t \rfloor$ in the worst case.*

Proof. Let $k = \lfloor \log t \rfloor$ and consider any broadcasting algorithm working in worst-case time at most k. We describe an adversary strategy preventing message transmission to v. During the execution of the algorithm, the adversary constructs a set B of *black* nodes initialized as empty (in the beginning all nodes are *white*). If in time $i \leq k$ the set of nodes that transmit is A_i and $|A_i \cap B| \geq \lfloor t/2^i \rfloor$ then all nodes in $A_i \cap B$ are colored black (added to B). After k steps the adversary declares all black nodes faulty and all white nodes fault free.

We show that the number of fault-free nodes in every set A_i is different from 1. (This proves that v hears either silence or noise at all times.) Let B_i denote the value of B before the ith step. There are two cases. If $|A_i \cap B_i| \geq \lfloor t/2^i \rfloor$ then all nodes in A_i are faulty. If $|A_i \cap B_i| > \lfloor t/2^i \rfloor$ then faulty nodes in $A_i \cap B_i$ are those colored black in steps $i+1, \ldots, k$. The number of these nodes is at most $\lfloor t/2^{i+1} \rfloor + \ldots + \lfloor t/2^k \rfloor < \lfloor t/2^i \rfloor$. Hence, there are at least 2 fault-free nodes in A_i.

Lemma 7. *If $t < R$, every broadcasting algorithm must use at least time $D - 1 + \lfloor \log t \rfloor$ to inform the domain, in the worst case.*

Theorem 2. *If $t < R$, the worst-case minimum time to inform the domain by an adaptive broadcasting algorithm is $(D + \log t)$. Algorithm Line-ADA achieves this performance.*

In the case $t \geq R$, Procedure Binary Elect can be easily modified by taking the set A to be the entire segment of length R. A segment in the domain must contain at least one fault-free node and hence Procedure Binary Elect will find the largest such node in time $\lceil \log R \rceil$. Algorithm Line-ADA based on this modified procedure will work in time $2D + O(\log R)$. On the other hand, Lemma 6 modified by letting A to be the entire segment of length R yields the lower bound $D - 1 + \lfloor \log R \rfloor$, similarly as in Lemma 7. Hence we get the following generalization of Theorem 2.

Theorem 3. *The worst-case minimum time to inform the domain on the line by an adaptive broadcasting algorithm is $(D + \log(\min(R, t)))$. Algorithm Line-ADA achieves this performance.*

3 The Mesh

In this section we consider the case when nodes of the radio network are situated at grid points of a mesh. We make the presentation for the square mesh but all results remain valid for other similar planar subdivisions, e.g., for the hexagonal mesh, and the arguments are easy to modify.

Nodes are situated at points with coordinates $(i; j) : 1 \quad i; j \quad n$. We assume that the source is at point $(1; 1)$ (the general case is similar). For simplicity assume that R divides n and R is even. Every square $f(i - 1)R + 1; :::; iRg$ $f(j - 1)R + 1; :::; jRg$, for $1 \quad i; j < n=R$ is called a *cell*. Denote by C_0 the cell containing the source. Partition every cell into 4 squares of side $R=2$, called *tiles*. Two cells (resp. tiles) are called *neighboring* if they touch each other by a side or a corner. Hence every cell (tile) has 8 neighboring cells (tiles). The range of every node is a square of side $2R$ centered at this node. Hence every two nodes in neighboring tiles are in each other's range and only nodes in the same or neighboring cells are in each other's range.

3.1 The nonadaptive case

Color all tiles using 9 colors, assigning the same color to tiles separated by two other tiles vertically, horizontally or diagonally. Let $S = R^2=4$ and fix any ordering $a_{ki} : i = 1; :::; S$ of nodes in the kth tile, for all tiles. Let $c(k)$ denote the color number of the kth tile. Consider the following nonadaptive broadcasting algorithm.

Algorithm Mesh-NA

1. The source transmits in time 0.
2. Node a_{ki} transmits in time $9(i - 1) + c(k) + 9Sr$, for $r = 0; 1; 2; :::$

That is, in time units 1,...,9, first nodes of tiles colored (respectively) 1,...,9 transmit, then in time units 10,...,18, second nodes of tiles colored (respectively) 1,...,9 transmit, etc. Notice that nodes transmitting in the same time unit are not in each other's range, and in every tile all nodes are scheduled to transmit successively in time intervals of length 9.

Lemma 8. *Algorithm Mesh-NA informs the domain in time $O(D + t)$, for any con guration of at most t faults.*

Proof. Fix a configuration of at most t faults. Define the following graph H on the set of all cells: Two (neighboring) cells C and C^0 are adjacent if there exist fault-free nodes $v \; 2 \; C$ and $v^0 \; 2 \; C^0$ that are in each other's range. Clearly, the domain is included in the union of cells that form the connected component of C_0 (in H). Let L be the maximum length of the shortest path (in H) from C_0 to any vertex in this connected component. Clearly $L \quad D$. Hence it is enough to show that our algorithm informs the domain in time $O(L + t)$.

Suppose that, at some time , all fault-free nodes in a given cell C know the source message, and C^0 is a cell adjacent to C. Let q and q^0 denote the number of faults in cells C and C^0, respectively.

Claim. At time $+18(q+q^0+1)$ all fault-free nodes in C^0 know the source message.

First suppose that each of the 8 tiles in cells C and C^0 contains at least one fault-free node. After at most $9+9q$ time units, a fault-free node v in a tile T in C, touching the cell C^0, is scheduled to transmit. After at most another period of $9+9q^0$ time units, a fault-free node v^0 in a tile T^0 in C^0, neighboring T, is scheduled to transmit. Nodes v and v^0 are in each other's range, which proves the Claim in this case. Next suppose that one of the above mentioned 8 tiles consists entirely of faulty nodes. Thus $q+q^0$ S. Let w 2 C and w^0 2 C^0 be fault-free nodes in each other's range. After at most $9S$ time units node w is scheduled to transmit and after at most another period of $9S$ time units node w^0 is scheduled to transmit. This concludes the proof of the Claim.

The Claim shows that the source message uses time at most $18f+18$ to traverse any edge of the graph H, where f is the total number of faults in both cells incident to this edge. Hence 18 time units of delay can be charged to any edge and, additionally, 18 time units of delay can be charged to any fault. Since cells have degree 8, the same fault may be charged at most 8 times. It follows that the domain will be informed in time at most $18L+144t$.

Lemma 9. *Every nonadaptive broadcasting algorithm must use time $D+\Omega(t)$ to inform the domain, in the worst case.*

Proof. For any nonadaptive broadcasting algorithm and any t, we have to show a configuration of at most t faults, yielding domain diameter D, for which this algorithm uses time $D+\Omega(t)$ to inform the domain. Fix a nonadaptive broadcasting algorithm and consider two cases.

Case 1. $t < 6R^2$: We use $x = \min(t; R^2-2)$ faults, all located in cell C_0, as shown in the proof of Lemma 2. The domain consists of all fault-free nodes and $D = n/R$. By Lemma 2, more than x time units are needed to inform any node outside of C_0, and at least $D-1$ additional time units are needed to inform the entire domain. Hence the time used is $D+\Omega(t)$.

Case 2. cR^2 t $(c+1)R^2$, for c 6: We use approximately $t/3$ faults arranged in full cells to create a snake-shaped domain, as shown in Figure 1.

This yields $D > c/4$. In the domain we leave every second cell entirely fault free and in each of the remaining cells we use R^2-1 faults as indicated in the proof of Lemma 4 (fewer than $t/3$ faults are enough to do that). A similar argument shows that the time used to inform the domain is at least $D+\frac{D}{2}(R^2-1) = D+\Omega(t)$.

Theorem 4. *The worst-case minimum time to inform the domain on the mesh by a nonadaptive broadcasting algorithm is* $(D+t)$. *Algorithm Mesh-NA achieves this performance.*

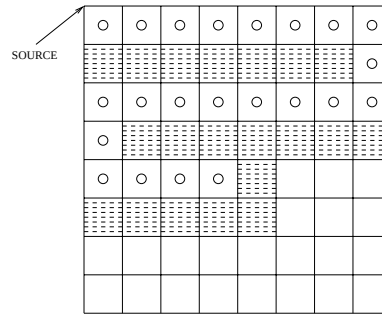


Fig. 1. Fault configuration for the lower bound. Shaded cells are entirely faulty. Cells with a circle are in the domain.

3.2 The adaptive case

As in the case of the line, our adaptive algorithm for the mesh consists of a preprocessing part and a proper broadcasting part. The aim of the preprocessing part is to elect representatives among all fault-free nodes in each cell. These representatives conduct message passing in the second phase. However, unlike in the case of the line, there is no natural choice of nodes in neighboring cells, analogous to the largest and smallest fault-free nodes in neighboring segments, guaranteed to be in each other's range. Hence, instead of Procedure Binary Elect, we use the following procedure independent of the topology of the radio network.

Let A and B be two sets of nodes of a radio network, such that every pair of nodes in A and every pair of nodes in B are in each other's range. We describe the Procedure Elect Couple that finds fault-free nodes $a \in A$ and $b \in B$, such that a and b are in each other's range, if such nodes exist. The procedure works in time $O(\log(|A| + |B|))$. First fix a binary partition of each of the sets A and B: Divide each of the sets into halves (or almost halves, in case of odd size), these halves into halves again, etc., down to singletons. For each member of the partition (except singletons), define the *first half* and the *second half*, in arbitrary order.

The basic step of Procedure Elect Couple (iterated $\lceil \log(\max(|A|, |B|)) \rceil$ times) is the following. Before this step a member set X of the binary partition of A and a member set Y of the binary partition of B are *promoted*. (They contain fault-free nodes in each other's range). This means that all fault-free nodes in A know that nodes in X have the "promoted" status; similarly for Y and B. After this step, one of the halves of X and one of the halves of Y (those containing fault-free nodes in each other's range) are promoted, the other halves are *defeated*. This basic step works in 8 time units as follows. Let X_1, X_2 and Y_1, Y_2 be the first and second halves of X and Y, respectively. We use the phrase "a nodes hears something" in the sense that a node hears either a message or noise.

1. All fault-free nodes in X_1 transmit.

2. All fault-free nodes in Y_1 that heard something in time 1, transmit. (If transmissions occur, Y_1 has been promoted).

3. All fault-free nodes in Y_2 that heard something in time 1 but nothing in time 2, transmit. (If transmissions occur, Y_2 has been promoted).

4. All fault-free nodes in X_1 that heard something in time 2 or 3, transmit. (If transmissions occur, X_1 has been promoted).

5. All fault-free nodes in X_2 that heard nothing in time 4, transmit.

6. All fault-free nodes in Y_1 that heard something in time 5, transmit. (If transmissions occur, Y_1 has been promoted).

7. All fault-free nodes in Y_2 that heard something in time 5 but nothing in time 6, transmit. (If transmissions occur, Y_2 has been promoted).

8. All fault-free nodes in X_2 that heard something in time 6 or 7, transmit. (If transmissions occur, X_2 has been promoted).

The first iteration of the above basic step is performed with $X = A$ and $Y = B$. Then, after $\lceil \log(\max(|A|, |B|)) \rceil$ iterations, exactly two fault-free nodes $a \in A$ and $b \in B$ are finally promoted, (if fault-free nodes in each other's range existed in sets A and B). They are said to be *elected* and are called *partners*. They are in each other's range and they know each other's identities. All other nodes have "defeated" status, they know it, and never transmit again.

In order to describe our adaptive broadcasting algorithm for the mesh, consider all pairs of neighboring cells. Partition these pairs into groups in such a way that distinct pairs in the same group do not contain neighboring cells. (36 groups are sufficient for the square mesh.) To each pair assign the number (0 to 35) of the group to which this pair belongs.

Now the algorithm can be described as follows.

Algorithm Mesh-ADA

1. Run Procedure Elect Couple for all pairs of neighboring cells, in 36 distinct phases, pairs of cells from the same group in the same phase, in parallel. (If $t < R^2$, start the procedure with subsets of cells of size $t + 1$, instead of the entire cells.) Partners elected for a pair of neighboring cells get the number 0 to 35 corresponding to this pair. (The same node may get more than one number.)
2. The source transmits the message.
   After an elected node heard the source message for the first time, it transmits it at the first time unit $u \equiv i \bmod 36$, where i is a number assigned to this node, and stops.

Lemma 10. *For any configuration of at most t faults, algorithm Mesh-ADA informs the domain in time $O(D + \log(\min(R, t)))$.*

Proof. As in the nonadaptive case, define the following graph H on the set of all cells: Two (neighboring) cells C and C' are adjacent if there exist fault-free nodes $v \in C$ and $v' \in C'$ that are in each other's range. In the preprocessing part of the algorithm (step 1) partners are elected for each pair of adjacent cells. This step takes time $O(\log(\min(R, t)))$. Consider two adjacent cells C and C' and suppose that at time all fault-free nodes in C know the message. After at most 36 time units, the partner in C, elected for the pair (C, C') transmits the message (if it had not done it before). After another period of 36 time units, its

partner in C^0 transmits and hence, in time at most $+72$, all fault-free nodes in C^0 know the source message. Hence traversing every edge of the graph H takes at most 72 time units. This proves that step 2 of the algorithm takes time $O(D)$.

The matching lower bound $D+\Omega(\log(\min(R;t)))$ can be derived from Lemma 6 similarly as in the case of the line. Hence we get the following result.

Theorem 5. *The worst-case minimum time to inform the domain on the mesh by an adaptive broadcasting algorithm is* $(D+\log(\min(R;t))$. *Algorithm Mesh-ADA achieves this performance.*

4 Conclusion

We presented asymptotically optimal fault-tolerant broadcasting algorithms for radio networks whose nodes are regularly situated on the line or on the mesh. The natural open problem is to generalize these results to the case of arbitrary graphs of reachability, as considered, e.g., in [1] and [3]. The lower bound from [1] shows that we cannot expect time $O(D+\log t)$ or $O(D+t)$ for arbitrary graphs. However, results from [1] and [3] leave a very small gap in the fault-free case. It would be interesting to get similarly close bounds for nonadaptive and adaptive algorithms in the presence of faults.

References

1. N. Alon, A. Bar-Noy, N. Linial and D. Peleg, A Lower Bound for Radio Broadcast, Journal of Computer and System Sciences 43 (1991), 290-298.
2. R. Bar-Yehuda, A. Israeli, and A. Itai, Multiple Communication in Multi-Hop Radio Networks, Proc. 8th ACM Symposium on Principles of Distributed Computing (1989), 329 - 338.
3. I. Gaber and Y. Mansour, Broadcast in Radio Networks, Proc. 6th Ann. ACM-SIAM Symp. on Discrete Algorithms, SODA'95, 577-585.
4. E. Kushilevitz and Y. Mansour, An $\Omega(D\log n)$ Lower Bound for Broadcast in Radio Networks, Proc. 12th Ann. ACM Symp. on Principles of Distributed Computing (1993), 65-73.
5. E. Kushilevitz and Y. Mansour, Computation in Noisy Radio Networks, Proc. 9th Ann. ACM-SIAM Symp. on Discrete Algorithms, SODA'98, 236-243.
6. E. Pagani and G. P. Rossi, Reliable Broadcast in Mobile Multihop Packet Networks, Proc. 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'97), (1997), 34 - 42.
7. K. Pahlavan and A. Levesque, Wireless Information Networks, Wiley-Interscience, New York, 1995.
8. A. Pelc, Fault-Tolerant Broadcasting and Gossiping in Communication Networks, Networks 28 (1996), 143-156.
9. K. Ravishankar and S. Singh, Asymptotically Optimal Gossiping in Radio Networks, Discrete Applied Mathematics 61 (1995), 61 - 82.
10. A. Sen and M. L. Huson, A New Model for Scheduling Packet Radio Networks, Proc. 15th Annual Joint Conference of the IEEE Computer and Communication Societies (IEEE INFOCOM'96) (1996), 1116 - 1124.

New Bounds for Oblivious Mesh Routing

Kazuo Iwama[1], Yahiko Kambayashi[1], and Eiji Miyano[2]

[1] Department of Information Science, Kyoto University, Kyoto 606-8501, Japan
{iwama,yahiko}@kuis.kyoto-u.ac.jp
[2] Kyushu Institute of Design, Fukuoka 815-8540, Japan
miyano@kyushu-id.ac.jp

Abstract. We give two, new upper bounds for oblivious permutation routing on the mesh network. One is an $O(N^{0.75})$ algorithm for the two-dimensional mesh with constant queue-size. This is the first algorithm which improves substantially the trivial $O(N)$ bound. The other is an $1.16\sqrt{N} + o(\sqrt{N})$ algorithm on the three-dimensional mesh with unlimited queue-size. This algorithm allows at most three bends in the path of each packet. If the number of bends is restricted to minimal, i.e., at most two, then the bound jumps to $\Omega(N^{2/3})$ as was shown in ESA'97.

1 Introduction

In the oblivious routing, the path of each packet is completely determined by its initial and final positions and is not affected by other packets. Hence, it is hard to avoid congestion in the worst case and it often takes much more time than it looks. Several lower-bounds which are rather surprising are known: For example: (i) An $\Omega(N)$ lower bound is known for any k-dimensional, constant queue-size mesh networks, where N is the total number of processors and k may be any constant [8]. (Note that an $O(N)$ upper bound can be achieved even on one-dimensional meshes, i.e., increasing dimensions in meshes does not work in the worst case.) (ii) An $\Omega(N^{2/3})$ lower bound is known for *three-dimensional*, unbounded queue-size meshes [5], which is much *worse* than the $O(\sqrt{N})$ bound for two-dimensional meshes. (iii) An $\Omega(\sqrt{N})$ lower bound for any constant-degree network [2,3,6]. It should be noted, however, that these lower bound proofs needed some supplementary conditions that might not seem so serious but are important for the proofs. In this paper, it is shown that the above lower bounds do not hold any more if those supplementary conditions are slightly relaxed.

More precisely, Krizanc needed the *pure condition* other than the oblivious condition to prove the $\Omega(N)$ lower bound in [8]. Roughly speaking, the pure condition requires that each packet must move if its next position is empty. Krizanc gave an open question, i.e., whether his linear bound can be improved by removing the pure condition. In this paper we give a positive answer to this question: it is shown that there are an $O(N^{0.75})$ algorithm on 2D meshes, an $O(N^{5/6})$ algorithm on 3D meshes and so on. The oblivious condition used in [8] is a little more stronger than the normal one, called the *source-oblivious condition*. That is also satisfied by our new algorithm, i.e., we remove only the pure condition in this paper. Note that this $\Omega(N)$ lower bound is quite tough; it still holds even without the oblivious condition; the *destination-exchangeable* strategy also implies the same lower bound [4]. Our new bound can be extended to the case of general

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 295–306, 1998.
© Springer-Verlag Berlin Heidelberg 1998

queue-size k, namely, it is shown that there are an $O(N^{0.75}/\sqrt{k})$ algorithm for 2D meshes of queue-size k, an $O(N^{5/6}/\sqrt{k})$ algorithm for 3D meshes and so on, while an $\Omega(N/k(8k)^{5k})$ lower bound was previously known for any constant degree, k-queue-size network under the pure condition [8]. For 2D meshes, if we set $k = \sqrt{N}$, then that is equivalent to unbounded queue-size. Our bound for this specific value of k is $O(N^{0.75}/N^{0.25}) = O(\sqrt{N})$, which matches the lower bound of [2,3,6].

Our second result concerns with 3D meshes: In [5] an important exception was proved against the well-known superiority of the 3D meshes over the 2D ones; oblivious permutation routing requires $\Omega(N^{2/3})$ steps over the 3D meshes under the following (not unusual, see the next paragraph) condition: The path must be shortest and be as straight as possible. In other words, each packet has to follow a path including at most two bends in the 3D case. [5] suggested that this lower bound may still hold even if the condition is removed; i.e., three-dimensional oblivious routing may be essentially inefficient. Fortunately this concern for 3D meshes was needless; we prove in this paper that any permutation can be routed over the 3D meshes in $1.16\sqrt{N} + o(\sqrt{N})$ steps by relaxing the condition a little bit: If we only allow the path of every packet to make one more bend, then the running time of the algorithm decreases from $N^{2/3}$ to $\sqrt{N}$. This upper bound is optimal within constant factor by [6] and does not change if we add the shortest-path condition.

For oblivious routing, there is a general lower bound, i.e., $\sqrt{N}/d$ for degree-d networks of any type [6]. This is tight for the hypercube, namely, $\sqrt{N} = \log N$ is both upper and lower bounds for oblivious routing over the hypercube [6]. This is also tight within constant factor for the 2D mesh, where $2\sqrt{N} - 2$ steps is an upper bound and also is a lower bound (without any supplementary condition) [1,9,10,11]. Thus, tight bounds are known for two extremes, for the 2D mesh and for the $\log N$-dimensional mesh (= the hypercube). Furthermore, both upper bounds can be achieved rather easily, i.e., by using the most rigid, *dimension-order path strategy* [12,2,6]. However, for the 3D meshes, even the substantially weaker condition, i.e., the minimum-bending condition, provides the much worse bound as mentioned before [5]. Our second result now extends the family of meshes for which optimal oblivious routing is known. If randomization is allowed, then the bound decreases to $O(N^{1/3})$ [7,13] for 3D meshes. A similar bound also holds for deterministic routing but for random permutations [9], including our new algorithm.

2 Models and Problems

Two-dimensional meshes are illustrated in Figure 1-(a). The following definitions on the two-dimensional mesh can be naturally extended to the three-dimensional mesh illustrated in Figure 3. A *position* is denoted by (i, j), $1 \le i, j \le \sqrt{N}$ and a processor whose position is (i, j) is denoted by $P_{i,j}$. A connection between the neighboring processors is called a (*communication*) *link*. A *packet* is denoted by $[i, j]$, which shows that the destination of the packet is (i, j). (A real packet includes more information besides its destination such as its original position and body data, but they are not important within this paper and are omitted.) So we have N different packets in total. An *instance* of (*permutation*) *routing* consists of a sequence $\pi_1 \pi_2 \cdots \pi_N$ of packets that is a permutation of the N

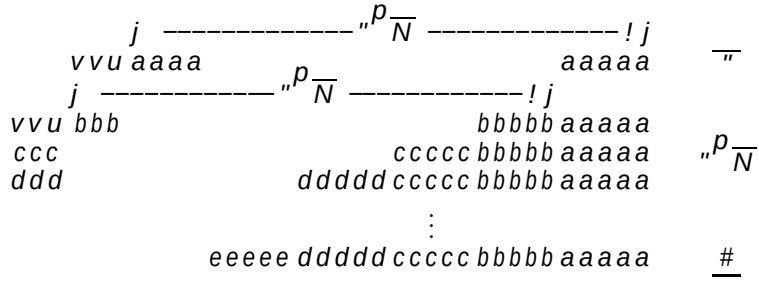
packets $[1;1]; [1;2]; \quad ; [\sqrt{N}; \sqrt{N}]$, where $\quad_1$ is originally placed in $P_{1;1}$, $\quad_2$ in $P_{1;2}$ and so on.

Each processor has four input and four output queues (see Figure 1-(b)). Each queue can hold up to k packets at the same time. The one-step computation consists of the following two steps: (i) Suppose that there remain l ($\quad 0$) packets, or there are $k - l$ spaces, in an output queue Q of processor P_i. Then P_i selects at most $k - l$ packets from its input queues, and moves them to Q. (ii) Let P_i and P_{i+1} be neighbouring processors (i.e., P_i's right output queue Q_i be connected to P_{i+1}'s left input queue Q_{i+1}). Then if the input queue Q_{i+1} has space, then P_i selects at most one packet (at most one packet can flow on each link in each time-step) from Q_i and send it to Q_{i+1}. Note that P_i makes several decisions due to a specific algorithm in both steps (i) and (ii). When making these decisions, P_i can use any information such as the information of the packets now held in its queues. Other kind of information, such as how many packets have moved horizontally in the recent t time-slots, can also be used.

If we fix an algorithm and an instance, then the path R of each packet is determined, which is a sequence of processors, $P_1(=$ source$); P_2; \quad ; P_j(=$ destination). R is said to be *b-bend* if R changes its direction at b positions. A routing algorithm, A , is said to be *b-bend* if the path of every packet is at most b-bend. A is said to be *oblivious* if the path of each packet is completely determined by its source and destination. Furthermore, A is said to be *source-oblivious* if the moving direction of each packet only depends on its current position and destination (regardless of its source position). A is said to be *minimal* if the path of every packet is the shortest one. A is said to be *pure* if a packet never stays at the current position when it is possible for the packet to advance.

Now here is an example of an oblivious routing algorithm, say, A_0, for $k = 1$. It turns out that A_0 is dimension-order, i.e., all packets move horizontally first and then make turns at most once at the crossings of source rows and destination columns. (i) Suppose that the top output queue of processor P_i is empty. Then P_i selects one packet whose destination is upward on this column. If there are more than one such packets, then the priority is given in the order of the left, right and bottom input queues. (Namely, if there is a packet that makes a turn in this processor, then it has a higher priority than a straight-moving packet.) Similarly for the bottom, left and right output queues, i.e., a turning packet has a priority if competition occurs. (ii) If an input queue is empty, then it is always filled by a packet from its neighboring output queue. Thus each queue of the processor never overflows under A_0. It is not hard to see that A_0 completes routing within roughly $c\sqrt{N}$ steps for many "usual" instances. Unfortunately, it is not true always.

Consider the following instance: Packets in the lower-left one-fourth plane are to move to the upper-right plane, and vice versa. The other packets in the lower-right and upper-left planes do not move at all. One can see that A_0 begins with moving (or shifting) packets in the lower-left plane to the right. Suppose that the flow of those packets looks like the following illustration: Here a shows a packet whose destination is on the rightmost column, b on the second rightmost column and so on. Note that the uppermost row includes a long sequence of a's. The second row includes five a's and a long b's, the third row includes five a's, five b 's and long c's and so on. We call such a sequence of packets which have the same destination column a *lump* of packets.



Now the lump of a's reach the rightmost column. One can see that the a's in the uppermost row can move into the vertical line smoothly and the the following packets can reach to their bending position smoothly also: Thus nothing happens against the uppermost row. However, the packet stream in the second row will encounter two different kinds of "blocks:" (1) The sequence of five a's is blocked at the upper-right corner since the $\sqrt{N}$ a's in the uppermost row have privileges. (2) One can verify that the last (leftmost) a of these five a's stops at the left queue of the second rightmost processor, which blocks the next sequence of b's, namely, they cannot enter the second rightmost column even if it is empty (see Figure 2). Thus, we need $\sqrt{N}$ steps before the long b's start moving. After they start moving, those b's in turn block the five b's on the third row and below. This argument can continue until the $\sqrt{N}$th row, which means we need at least $(\sqrt{N})^2$ steps only to move those packets.

One might think that this congestion is due to the rule of giving a higher priority to the packets that turn to the top from the left. This is not necessarily true. Although details are omitted, we can create "adversaries" that imply a similar congestion against other resolution rules such as giving a priority to straight-moving packets. In the next section we will see how we can avoid this kind of bad blocking.

3 2D Oblivious Routing

Theorem 1. There is an oblivious routing algorithm on 2D meshes of queue-size k ($0 \quad k \quad c\sqrt{N}$ for some constant c) which runs in $O(N^{0.75}/\sqrt{k})$ steps.

Proof. The whole plane is divided into 36 subplanes, $SP_{1,1}$ through $SP_{6,6}$ as shown in Figure 4. For simplicity, the total number of processors in 2D meshes is hereafter denoted by not N but $36n^2$, i.e., each subplane consists of $n \quad n$ processors.

Our basic idea is as follows: (1) The entire algorithm is divided into $36 \quad 36$ *phases*. In the first phase only packets whose sources and destinations are both in $SP_{1,1}$ move. In the second phase only packets from $SP_{1,1}$ to $SP_{1,2}$ move, and so on. (2) Suppose that it is now the phase where packets from $SP_{2,3}$ to $SP_{5,2}$ move. Then the paths of those packets are not shortest: They first move to $SP_{2,6}$, then to $SP_{5,6}$, and finally to $SP_{5,2}$ (see Figure 4). (3) This path consists of three different zones, the (*parallel*) *shifting zone*, the *sorting zone* and the *critical zone*. In the above example, the sorting zone is composed of three consecutive subplanes $SP_{2,3}$, $SP_{2,4}$ and $SP_{2,5}$, the critical zone is $SP_{2,6}$ and the parallel shifting zone includes all the remaining portions.

We first describe the purpose of three different zones: First of all, until packets reach the shifting zone, they move without changing their relative positions,

i.e., they move in parallel just like a formation flight. The sorting zone needs three consecutive subplanes where the flow of packets is changed, e.g., from an arbitrary order to the farthest-first order. The critical zone is the most important zone where each packet enters its correct column position (relatively within the subplane). Apparently no congestion as described before occurs in the shifting zone. Our goal is to reduce the congestion in the critical zone with a help of the sorting zone.

Now one can see why the path from $SP_{2;3}$ to $SP_{5;2}$ takes such a long way, namely, it is because we need the sorting zone before the critical zone. Note that the algorithm can determine the path independently for each of the 36^2 phases. If the starting subplane is in the left side of the whole plane (as $SP_{2;3}$), then the packets are once moved to the right end, which allows us to prepare the sorting zone. If the starting subplane is on the right side, then the packet are once moved to the left end. If the starting and ending subplanes are on the same row, then the path goes like the one given by the dotted line in Figure 4. It should be noted that this path design is obviously source-oblivious in each time-step, but is not in the whole time-steps. To make it source-oblivious in the whole steps is not hard, see the remark at the end of this section.

Recall that the algorithm A_0 given in the previous section takes $\Omega(n^2)$ steps, and one can see that this inefficiency comes from the co-existence of long and short lumps of packets. Actually, as shown in Lemma 2 later, the same algorithm runs quickly if there are only long lumps. On the other hand, if we have only short lumps, then we can also handle it efficiently (Lemma 3). So our basic strategy is quite simple: We first sort the packets so that packets having farther column destinations will go first. Then, we let only "long" lumps go to the critical zone. The remaining "short" lumps go afterwards but this time we adopt the so-called *shu ed order* as the sequence of packets.

Now we give more detailed description of a single phase of the algorithm Rout[2]. The size of queues is two ($k = 2$) for a while, but it is extended to Rout[k] later:

Algorithm: Rout[2]

A single step of Rout[2] is divided into the following seven stages:

Stage 1: The packets move in the shifting zone before enter the sorting zone.

Stage 2 (Sorting): The following algorithm appeared in [4]: Now a sequence of packets on some row go through the sorting zone, three consecutive subplanes, say, SP_1, SP_2 and SP_3. Look at the following example:

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
initial positions	a	c		b		c	b	a		a

which shows that the sequence of packets are now included in SP_1. Note that a should go to the farthest column, b the second farthest and so on as before. We wish those packet to go out of SP_1 in the reverse order, i.e., in the order of c, b and a. (This order is again reversed later.) At this moment, all the packets once stop moving and resume moving as follows: In the first step, the leftmost packet, a in this example, moves to the right and P_2 now holds a and c in its input queue (recall that $k = 2$). In the second step, c is selected among those a and c since c should go out earlier than a, and it goes to P_3. For example, after the eighth step, the positions change as follows:

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
after the 8th step							b	a		a
						a	b	c	c	

In the next step, P_{10}, P_9, P_8 and P_7 receive c, c, b and a, respectively. In the next step, the packet c first goes out of SP_1 and so on. Recall that only packets from one subplane to one subplane move in a single phase. So, the initial sequence of packets may include spaces or "null packets," but it does not cause any problem by regarding the null packets as the farthest (going out latest) packets.

Stage 3 (Counting): Our main purpose of this stage is to make processors know the size of each lump in order to divide lumps into two groups, long lumps and short ones. Now the (reversely) sorted sequence of packets are in SP_2:

$$\begin{array}{cccccccccccccccccccccccccccccc} P_{11} & P_{12} & P_{13} & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} & P_{19} & P_{20} & P_{21} & P_{22} & P_{23} & P_{24} & P_{25} & P_{26} & P_{27} & P_{28} & P_{29} & P_{30} \\ a_5 & a_4 & a_3 & a_2 & a_1 & b_2 & b_1 & c_3 & c_2 & c_1 \end{array}$$

Then we change the current position of each packet into the symmetrical position with respect to the boundary between SP_2 and SP_3, i.e., c_1 moves to P_{21}, c_2 to P_{22}, c_3 to P_{23} and so on: All we have to do is to shift all the packets one position to the right step by step. One can see that c_1 arrives at the temporal destination P_{21} in the first step, c_2 arrives at P_{22} in the third step, and so on. For example, after the sixth step, the positions change as follows:

$$\begin{array}{cccccccccccccccccccccccccccccc} P_{11} & P_{12} & P_{13} & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} & P_{19} & P_{20} & P_{21} & P_{22} & P_{23} & P_{24} & P_{25} & P_{26} & P_{27} & P_{28} & P_{29} & P_{30} \\ & & & & & & a_5 & a_4 & a_3 & a_2 & a_1 & b_2 & b_1 & & & & & & & \\ & & & & & & & & & & c_1 & c_2 & c_3 & & & & & & & \end{array}$$

The leftmost packet a_5 finally arrives at the rightmost processor P_{30} in the 19th step of this stage:

$$\begin{array}{cccccccccccccccccccccccccccccc} P_{11} & P_{12} & P_{13} & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} & P_{19} & P_{20} & P_{21} & P_{22} & P_{23} & P_{24} & P_{25} & P_{26} & P_{27} & P_{28} & P_{29} & P_{30} \\ & & & & & & & & & & c_1 & c_2 & c_3 & b_1 & b_2 & a_1 & a_2 & a_3 & a_4 & a_5 \end{array}$$

Note that after this stage, the stream of packets is changed into the farthest-first order. More importantly, every processor P_i now knows how many packets in the same lump exist on its right side by counting the number of the flowing packets. For example, P_{22} knows one more c exists in P_{23}.

Stage 4 (Moving Long Lumps in Farthest-First Order): Suppose for example that a lump is defined to be long if it includes four or more packets. Then the lump of a's is long in the above example:

$$\begin{array}{ccccc} P_{26} & P_{27} & P_{28} & P_{29} & P_{30} \\ a_1 & a_2 & a_3 & a_4 & a_5 \end{array}$$

Now, P_{26} and P_{27} know that this lump is long since they have already sent at least three packets to the right. Then they move their packets to the right in the next step. In the next step, P_{28} receives a_1, which makes P_{28} know that its lump is long and so on. Thus long lumps move forward. This action is initiated in all long lumps simultaneously. Note that P_{21} who knows that this lump is not long does not initiate the move of packets, and thus this short lump remains in SP_3.

Stage 5 (Moving Short Lumps in Shuffle Order): Now there remain only packets of short lumps in SP_3. For example, consider the following short lumps of packets:

$$d_2\ d_1 \qquad c_3\ c_2\ c_1 \qquad b_2\ b_1 \qquad a_3\ a_2\ a_1$$

In the first step, the four rightmost packets $a_1; b_1; c_1; d_1$ of each lump move horizontally to the critical zone. In the second step, the four second rightmost packets $a_2; b_2; c_2; d_2$ move horizontally, and so on. Namely, we change the above farthest-first sequence to the following *shuffle order* sequence (which may include spaces between some two packets):

$$c_3 \qquad a_3 \qquad d_2 \qquad c_2 \qquad b_2 \qquad a_2 \qquad d_1 \qquad c_1 \qquad b_1 \qquad a_1$$

To design this algorithm is not hard since each processor knows how many packets in its lump are on its right side and when it should start its packet moving.

Stage 6 (Critical Zone): Right after the sorting zone, the packets enter the critical zone. Recall that long lumps enter first in the farthest-first order and then short lumps in the shuffle order. Each packet changes the direction from row to column at the crossing of its correct destination column (relatively in the subplane). What Rout[2] does in this zone is exactly the same as A_0, i.e., it gives a higher priority to turning packets.

Stage 7 (Shifting Zone): Now packets move in the shifting zone towards their final goals.

Now we shall investigate the time complexity of Rout[2]. The following discussions are only about what happens in the critical zone. (The time needed for the sorting and shifting zones is apparently linear in n.) Suppose that a packet is now in the left input queue of a processor P_i, where is to enter the destination column by making a turn. Then it is said that is *ready to turn*. We first show a simple but important lemma which our following argument depends on.

Lemma 1. Suppose that the rightmost packet of a lump L of packets is now ready to turn into its destination column. Then all the packets in L go out of the critical zone at latest $3n$ steps from now regardless of the behavior of other packets.

Proof. Suppose that a packet which is in the same lump L cannot move on. Then there must be a packet, , which is ready to turn into the same column in some upper position than and which can move on in the next time-step. Let us call such a packet , *a blocking packet against* . Note that cannot be a blocking packet against any longer in the next step. Thus, in each step from now on, the following (i) or (ii) must occur: (i) The leftmost packet in L moves one position on. (ii) One blocking packet against this leftmost packet in L disappears. Since there are at most n packets that are to turn into a single column, (ii) can occur at most n times. (i) can occur at most $2n$ times also since we have at most $2n$ processors on the path of each packet in the critical zone. □

Suppose from now on that a lump is said to be long if it includes at least d packets for some positive constant d; otherwise, it is said to be short.

Lemma 2. All the long lumps can go through the critical zone within $O(n^2/d)$ steps.

Proof. The following argument holds for any row: Let L_i be a lump of packets in some row such that packets of L_i head for a farther column than packets of L_j for $i < j$. Since packets of long lumps move in the farthest-first order, they flow in the following form of lumps:

$$L_3 \quad L_2 \quad L_1$$

Since each long lump has at least d packets, there are at most n/d different lumps in each row. Now the rightmost packet of L_1 must be ready to turn within $2n$ steps. By Lemma 1, then, those packets in L_1 must go out of the critical zone within the next $2n$ steps, i.e., within $4n$ steps in total. After L_1 goes out of the critical zone, L_2 must go out within $4n$ steps and so on. Thus $4n \quad n/d = 4n^2/d$ is the maximum amount of steps all the long lumps need to go out. □

Lemma 3. All the short lumps can go through the critical zone within $O(dn)$ steps.

Proof. The flow of packets on each row looks as follows:

$$z_1 \quad z_{l-1}\ z_l\ y_1 \quad y_{j-1}\ y_j\ x_1 \quad x_{i-1}\ x_i$$

Here $x_1 \quad x_i$ is a sequence of different packets such that x_i is heading for a farther column than x_{i-1}. Let us call this sequence an *ordered string*. $y_1 \quad y_j$ is the next ordered string (i.e., y_j is heading for a farther column than x_1) and so on. One can see that each packet in the first ordered string becomes ready to turn within $2n$ steps and must go out of the critical zone within the next $2n$ steps by Lemma 1 (regardless of possible spaces between some two packets). After that, the second ordered string must be ready to turn within $2n$ steps and then goes out as before. Note that we have only d ordered strings since each short lump has less than d packets. Hence, $4n \quad d$ is the enough number of steps before all the short lumps go out. □

Now we are almost done. When moving packets from the sorting zone to the critical zone, we first move only lumps whose size is at least $\sqrt{n}$. Then by Lemma 2, the routing for these "long" lumps will finish in $O(n\sqrt{n})$ steps. After that we move "short" lumps, which will be also finished, by Lemma 3, in $O(n^2/\sqrt{n}) = O(n\sqrt{n})$ steps.

Algorithm Rout[2] can be extended to Rout[k] for a general queue-size k: We can obtain similar lemmas, say Lemmas 1', 2' and 3' to Lemmas 1, 2 and 3, respectively for the general queue-size. Lemma 1' is exactly the same as Lemma 1. As for Lemma 2', we replace d by kd. Then the time complexity for Rout[k] is $O(n^2/kd)$. As for Lemma 3', we replace d by kd and furthermore each ordered string $x_1 \quad x_{i-1}x_i$ is replaced by $X_1 \quad X_{i-1}X_i$. Here X_{i-m} consists of k packets of the same destination instead of only one packet for x_{i-m}. Since our queue-size is k this time, each X_{i-m} can fit into a single input queue, which means all the packets in X_{i-m} can get ready to turn within $2n$ steps. Also note that the number of the ordered strings is at most d. Hence the bound in Lemma 3, $O(nd)$, does not differ in Lemma 3', which is a benefit of the large queue-size.

If we set $d = \sqrt{n/k}$, then both $O(n^2/kd)$ and $O(nd)$ become $O(n^{1.5}/\sqrt{k})$. □

Remark. Recall that the current movement of packets is not source-oblivious in the whole time-steps, which can be modified as follows: Suppose that the goal subplane is in the left-lower of the whole plane. Then all the packets but in the rightmost subplanes move upward first. Then they move to the right in the uppermost subplanes, and then go downward in the rightmost subplanes (here we place the sorting zone), and finally go to the left. (Exceptionally, packets in the rightmost-lower subplanes move to the left first and then follow the same path as above.) Thus there is only one direction set in each subplane, i.e., we can design a source-oblivious algorithm. However, here is one important point: Packets originally placed near the destination in this path cannot go through the sorting zone. Hence, we allow them once to go through the destinations and then join the standard path after that.

Remark. Extension to 3D case. Now each packet x moves temporarily up to the ith horizontal yz-plane if the destination of x is the ith vertical xy-plane (see Figure 3). We need n steps to move each of the n^2 packets placed on some horizontal plane up to its correct horizontal one, and we need $O(n^{1.5}/\sqrt{k})$ steps to move the packets in one horizontal plane. It is not hard to move each packet from the horizontal to the vertical planes in $O(n)$ steps. Since there are n horizontal planes, the total routing time is $O(n \quad n^{1.5}/\sqrt{k}) = O(n^{2.5}/\sqrt{k})$.

4 3D Oblivious Routing

For oblivious routing on any d-degree network, Kaklamanis *et al.* [6] proved an $\Omega(\sqrt{N}/d)$ lower bound. It is known [6,9,12] that this bound is tight for the hypercube and the 2D meshes. However, in the case of 3D meshes, it was not known whether the lower bound is indeed tight. Here is our answer:

Theorem 2. There exists a deterministic, oblivious routing algorithm on the three-dimensional mesh that routes any permutation in $1.16\sqrt{N} + o(\sqrt{N})$ steps.

Remark. Theorem 2 holds for minimal and 3-bend routing. However, if we impose the 2-bend condition, then any oblivious routing algorithm requires $\Omega(N^{2-3})$ steps [5]. Recall that in the case of 2D meshes, dimension-order routing, which is even more rigid than the 1-bend routing, can achieve the tight $O(\sqrt{N})$ bound. Thus there is a big gap between 2D and 3D meshes in the difficulty of oblivious routing.

Proof. Let the total number of processors in 3D meshes be hereafter n^3, not N. We first present a deterministic, but non-minimal oblivious routing algorithm on the 3D mesh that routes any permutation in $O(n\sqrt{n})$ $(= O(\sqrt{N}))$ steps. At the end of this section, we will describe how to strengthen the algorithm to route every packet along its shortest path. See Figure 3 again. The three dimensions are denoted by *x-dimension*, *y-dimension* and *z-dimension*. *x_i-plane* $(1 \le i \le n)$ means the two-dimensional plane determined by $x = i$. Similarly for y_j-plane and z_k-plane. For example, the x_1-plane on the 3D mesh is the top, horizontal plane in Figure 3, which includes n^2 positions $(1,1,1), \ldots, (1,n,1), (1,1,2), \ldots, (1,n,n)$. Let *$x_iy_j$-segment* $(1 \le i,j \le n)$ denote the linear array determined by $x = i$ and $y = j$, which consists of n processors $(i,j,1)$ through (i,j,n). Similarly for y_jz_k-segment and z_kx_i-segment.

The whole $n \times n \times n$ mesh is partitioned into several submeshes. See Figure 5. *x-zone*[1] consists of the top $\sqrt{n}$ planes, the x_1-plane through the $x_{\sqrt{n}}$-plane. Similarly *x-zone*[2] consists of the next $\sqrt{n}$ planes. Each plane is divided into $\sqrt{n} \times \sqrt{n}$ groups, called *blocks*. *z_k-block*$[i,j]$ on the z_k-plane $(1 \le k \le n)$ is the ith $\sqrt{n}$ from the top and jth from the left. Then $\sqrt{n}$ processors on the jth row from the top of each z_k-block$[i,j]$ are called *critical positions*, which play on an important role in the following algorithms. Note that the number of critical positions on a single z_kx_i-segment is exactly $\sqrt{n}$.

The following observation makes clear the reason why the elementary-path or dimension-order routing does not work efficiently: Suppose that n^2 packets placed on the z_1-plane should move to the x_1-plane. Also suppose that all of those n^2 packets first go up to the n processors on a single segment, $P_{1,1,1}$ through $P_{1,n,1}$, and then move along y-dimension. Then most of the n^2 paths probably go through one of the $n - 1$ links between those n processors, which requires at least $\Omega(n^2)$ steps. Apparently we need to eliminate such heavy traffic on the single segment. The key strategy is as follows: The n^2 packets or paths (which go through a single segment in the worst case) are divided into $\sqrt{n}$ groups, $n\sqrt{n}$ paths each. Then the number of paths which a single segment has to handle can be reduced to those $n\sqrt{n}$ paths from the previous n^2 ones, where critical positions of each block play their roles. Here is our algorithm based on the dimension-order routing:

Algorithm: DO-3-bend

Stage 1: Every packet moves along x-dimension to a critical position which is located in the same x-zone$[i]$ as its final destination. Namely, all of the packets go up or down into their correct x-zones.

Stage 2: Every packet placed now on the critical position moves along y-dimension to its correct y-coordinate, i.e., moves horizontally into its correct y_j-plane.

Stage 3: Every packet moves again along x-dimension to its correct x-coordinate, i.e., into its correct $x_i y_j$-segment.

Stage 4: Every packet moves along z-dimension to its final position.

One can see that the path of each packet is completely determined by its initial position and destination and that the algorithm can deliver all the packets for any permutation.

Lemma 4. DO-3-bend can route any permutation in at most $2n\sqrt[p]{\overline{n}}+o(n\sqrt[p]{\overline{n}})$ steps.

Proof. (1) Stage 1 requires at most n steps since every packet can move without delays. (2) Stage 2 requires at most $n\sqrt[p]{\overline{n}}+n$ steps. The reason is as follows: Recall that the number of critical positions on a single segment is exactly $\sqrt[p]{\overline{n}}$. Since each critical position holds at most n packets after Stage 1, the total number of packets that go through each $z_k x_i$-segment is at most $n\sqrt[p]{\overline{n}}$. Thus, it takes at most $n\sqrt[p]{\overline{n}}$ steps until those $n\sqrt[p]{\overline{n}}$ packets move out of the current block and it furthermore takes at most n steps until the packet which finally starts from the block arrives at its next intermediate position. (3) After Stage 2 every packet arrives at its correct sub-y_j-plane of the $\sqrt[p]{\overline{n}}$ n positions. Thus, the total number of packets that go through each link along x-dimension is at most $n\sqrt[p]{\overline{n}}$. Thus it takes at most $n\sqrt[p]{\overline{n}}+\sqrt[p]{\overline{n}}$ steps until all the packets arrive at their next positions. (4) Since each processor temporally holds at most n packets, $2n$ steps are sufficient for all the packets to arrive at their final positions. As a result, the whole algorithm requires at most $2n\sqrt[p]{\overline{n}}+o(n\sqrt[p]{\overline{n}})$ steps. □

The above algorithm is based on the dimension-order routing, i.e., all the packets move in the same direction in each stage. However, by making the packets move in different directions we can route more efficiently. The whole mesh is partitioned into the three groups shown in Figure 6. Then, in the first stage, the algorithm NDO-3-bend moves all the packets which are initially placed on the groups marked "X", "Y" and "Z" along x-dimension, y-dimension and z-dimension, respectively. In the remaining stages, dimensions are switched adequately for each group. Note that each block can be viewed as $\sqrt[p]{\overline{n=3}}$ $\sqrt[p]{\overline{n=3}}$ mesh. Now the following lemma holds:

Lemma 5. NDO-3-bend can route any permutation in at most $1.16n\sqrt[p]{\overline{n}}+o(n\sqrt[p]{\overline{n}})$ steps.

Proof. It also takes at most n steps for each packet to arrive at its critical position in the first stage. Note that there are $3\sqrt[p]{\overline{n=3}}$ critical positions on a single $z_k x_i$-segment and each critical position holds at most $n/3$ packets after the first stage. Then it takes at most $n\sqrt[p]{\overline{n=3}}+n$ steps during the second stage. In the third and fourth stages, it takes at most $n\sqrt[p]{\overline{n=3}}+\sqrt[p]{\overline{n=3}}$ and $2n$ steps, respectively. Since $2+\sqrt[p]{3}\approx 1.16$, the lemma holds. □

Finally we add the condition that all of the packets should follow their shortest routes: Suppose that a packet which heads for the x_t-plane is originally placed on the x_s-plane for $s > t$. Then, in the first stage, should go up along

x-dimension to the closest but below critical position to the x_t-plane so as to follow its shortest path. Similarly for $s < t$. However, as a special case, if there is no critical position between 's original and destination planes, then does not move at all in the first stage. It is not hard to implement this idea in the same amount of time as the above algorithm:

Theorem 3. There exists a deterministic, oblivious, minimal, 3-bend routing algorithm on the three-dimensional mesh that can route any permutation in at most $1.16n\sqrt{n} + o(n\sqrt{n})$ steps. (The proof is omitted.)

Remark. Each algorithm can be modified so as to satisfy the source-oblivious condition by increasing the constant factor of its running time slightly.

Remark. It is not hard to show that our algorithm runs for random permutations in $3n + o(n)$ steps with high probability. Here is the reason: Since there are $\sqrt{n}$ critical positions on each vertical segment, each critical position receives $\sqrt{n}$ packets on average in the first stage. Then $\sqrt{n} \cdot \sqrt{n} = n$ packets temporally stay on each horizontal segment. In the second stage, those n packets move horizontally in n steps, and each processor receives one packet on average. The third stage requires $\sqrt{n}$ steps. Finally, all the packets arrive at their final positions in n steps.

References

1. A. Bar-Noy, P. Raghavan, B. Schieber and H. Tamaki, "Fast deflection routing for packets and worms," In *Proc. ACM Symposium on Principles of Distributed Computing* (1993) 75-86.
2. A. Borodin and J.E. Hopcroft, "Routing, merging, and sorting on parallel models of computation," *J. Computer and System Sciences* 30 (1985) 130-145.
3. A. Borodin, P. Raghavan, B. Schieber and E. Upfal, "How much can hardware help routing?," In *Proc. ACM Symposium on Theory of Computing* (1993) 573-582.
4. D.D. Chinn, T. Leighton and M. Tompa, "Minimal adaptive routing on the mesh with bounded queue size," *J. Parallel and Distributed Computing* 34 (1996) 154-170.
5. K. Iwama and E. Miyano, "Three-dimensional meshes are less powerful than two-dimensional ones in oblivious routing," In *Proc. 5th European Symposium on Algorithms* (1997) 284-295.
6. C. Kaklamanis, D. Krizanc and A. Tsantilas, "Tight bounds for oblivious routing in the hypercube," *Mathematical Systems Theory 24* (1991) 223-232.
7. C. Kaklamanis, D. Krizanc and S. Rao, "Simple path selection for optimal routing on processor arrays," In *Proc. 1992 ACM Symposium on Parallel Algorithms and Architectures* (1992) 23-30.
8. D. Krizanc, "Oblivious routing with limited buffer capacity," *J. Computer and System Sciences* 43 (1991) 317-327.
9. F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes,* Morgan Kaufmann (1992).
10. F.T. Leighton, F. Makedon and I. Tollis, "A $2n - 2$ step algorithm for routing in an $n \times n$ array with constant queue sizes," *Algorithmica* 14 (1995) 291-304.
11. J.F. Sibeyn, B.S. Chlebus and M. Kaufmann, "Deterministic permutation routing on meshes," *J. Algorithms* 22 (1997) 111-141.
12. M. Tompa, *Lecture notes on message routing in parallel machines,* Technical Report # 94-06-05, Dept of Computer Sci. & Eng., Univ of Washington (1994).
13. L.G. Valiant and G.J. Brebner, "Universal schemes for parallel communication," In *Proc. ACM Symposium on Theory of Computing* (1981) 263-277.

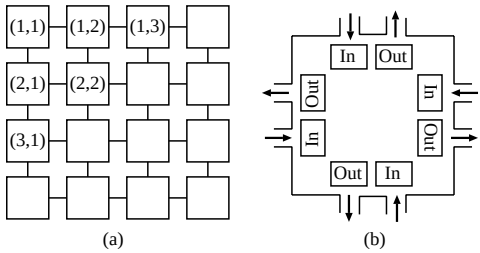


Figure 1: (a) 2D mesh (b) Processor

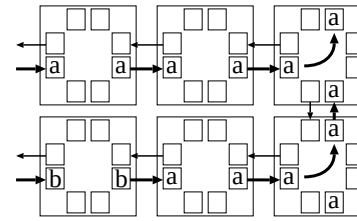


Figure 2: Block

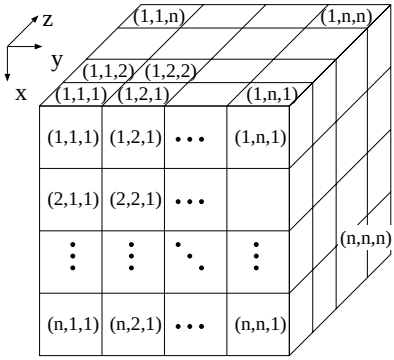


Figure 3: 3D mesh

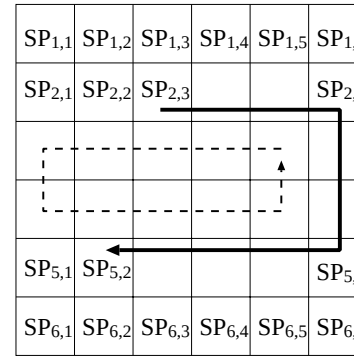


Figure 4: 36 subplanes

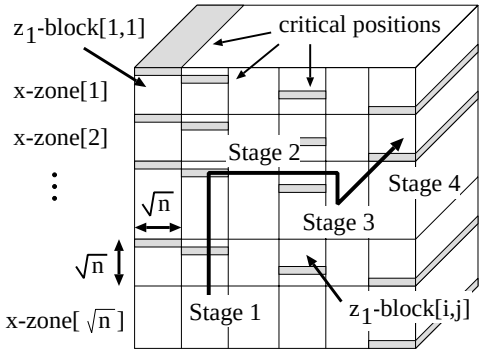


Figure 5: Zones, blocks, critical positions

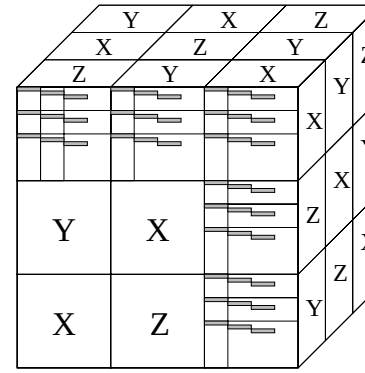


Figure 6: Three groups, X, Y, Z

Evaluating Server-Assisted Cache Replacement in the Web

Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford

AT&T Labs–Research, 180 Park Avenue, Florham Park, NJ 07932 USA
{edith,bala,jrex}@research.att.com
Web page: http://www.research.att.com/~{edith,bala,jrex}

Abstract. To reduce user-perceived latency in retrieving documents on the world wide web, a commonly used technique is caching both at the client's browser and more gainfully (due to sharing) at a proxy. The effectiveness of Web caching hinges on the replacement policy that determines the relative value of caching different objects. An important component of such policy is to predict next-request times. We propose a caching policy utilizing statistics on resource inter-request times. Such statistics can be collected either locally or at the server, and piggybacked to the proxy. Using various Web server logs, we compared existing cache replacement policies with our *server-assisted* schemes. The experiments show that utilizing the server knowledge of access patterns can greatly improve the effectiveness of proxy caches. Our experimental evaluation and proposed policies use a *price function* framework. The price function values the utility of a unit of cache storage as a function of time. Instead of the usual tradeoffs of *profit* (combined value of cache hits) and cache size we measure tradeoffs of profit and *caching cost* (average allocated cache portion). The price-function framework allows us to evaluate and compare different replacement policies by using *server logs*, without having to construct a full workload model for each client's cache.

1 Introduction

The popularity of the World Wide Web has imposed a significant burden on the communication network and web servers, leading to noticeable increases in user-perceived latency. Caching popular resources at proxies offers an effective way to reduce overhead and improve performance. Given that proxy caches are of a finite size, sound policies are needed for replacing resources in the cache. Although cache replacement has been studied extensively in the context of processor architecture and file systems, the Web introduces more complicated challenges, since resources vary substantially in their sizes, fetching costs, and access patterns. Indeed, new replacement policies have been developed to incorporate these parameters [1,2,3,4]. Although the Web exhibits high variability between the access patterns of different resources, individual resources have more regular access patterns. The most important factor for a good replacement policy remains predicting the next request time of a resource which depends on the ability to observe and interpret the access patterns.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 307{319, 1998.
c Springer-Verlag Berlin Heidelberg 1998

The natural place to collect statistical information on request patterns and interdependencies is the Web server. Any one proxy views a small number of requests to any one server, and hence, the data at its disposal may not be sufficient to model the access patterns. In addition, the proxy contacts a large number of different servers, and the required computational efforts to generate such models may not be justified. Proxy cache performance can be enhanced by having servers generate predictions of accesses to their resources and communicate this knowledge back to the proxies (as hinted in [5] and explored in [6]). Following the approach in [6,7,8], we assume that proxies and servers can piggyback information on regular request and response messages. The information exchange can be incorporated into the HTTP 1.1 protocol without disrupting the operation on non-participating proxies and servers [6]. For the sake of efficiency, the server does not maintain an online history of each proxy's past accesses and, hence, bases its response to the proxy only on the requested resource and piggyback content. The protocol does not involve any new messages between the proxy and server sites. Operating within these practical constraints, we propose policies for *server-assisted* cache replacement.

Performance of cache replacement policies hinges on the accuracy of the predictions of the next access time for each resource. The LRU policy, for example, utilizes the last request time of each resource as the only predictor for its next request. The LFU replacement policy utilizes the frequency with which a resource is accessed and the LRV policy [4] utilizes the number of previous requests to the resource. We propose to enhance replacement policies by providing the proxy with the distribution function of the next request time for each resource. The distribution is estimated by collecting per-resource statistics on inter-request times. In server-assisted replacement, the server generates a histogram of inter-request times by observing its request logs. The histogram is piggybacked to the requesting proxy and used to determine the duration for caching resources. In addition, the server may piggyback hints about related resources that are likely to be accessed in the future.

Our experimental evaluation compares server-assisted and *proxy-local* (i.e., without the help of server hints) replacement policies. The evaluation of server-assisted policies necessitated the use of *server logs* since proxy logs do not provide us the statistics available to the servers. We used three large Web server logs. The logs provide, for each client, the full request sequence of resources belonging to the particular server. We used *price functions* to evaluate the performance of different replacement policies using the information in the server log, without directly considering (or simulating) the full workload of the cache. The *price* is a function of time capturing the current value of a unit of cache storage. The *caching-cost* of a resource for a time period is the product of the resource size, the length of the time interval, and the average price during that period. In the experimental evaluation we used constant-valued price functions (i.e., an assumption of *steady-state*). Instead of considering the usual tradeoffs of cache size and *pro t* (sum of the fetching cost of objects across all cache hits), we measured tradeoffs of profit and *average cache portion allocated to server's re-*

sources.[1] The profit and caching costs are computed for each client separately and summed over all clients. Tradeoffs are obtained by varying the *threshold* price. When using price functions, cache replacement policies are viewed as generating a sequence of decisions on if and how long to cache each resource. In the evaluation, we cast traditional cache replacement policies (such as LRU) in this framework.

In Section 2 we formulate the optimal caching policy in terms of a price function where each resource has a distribution of inter-request times. We discuss a replacement policy when knowledge of this distribution is replaced by statistics on inter-request times. Section 3 provides an experimental evaluation of our approach. We conclude with a section on ongoing and future work.

2 Cache Replacement Model

Our caching framework models the value of caching a resource in terms of its fetching cost, size, next request time, and *cache prices* during the time period between requests. After deriving expressions for the profit and the cost of caching a resource, we describe how the Web server can aid the proxy by estimating the request interarrival distribution and the likelihood of accesses to related resources. Together, the proxy's price function and the inter-request statistics guide the cache replacement decision.

2.1 Cache Price Function and Resource Utility

As the basis of the cache replacement policy, we use the *price function* (t) of the cache and the *utility* of a resource to the proxy. For a resource r requested at time t, let s_r be the size, $t + \ _{r;t}$ be the time of its next request, and $f_{r;t}$ be its fetching cost at time $t + \ _{r;t}$. If $\ _{r;t}$ is known in advance, then the resource is worth caching for either 0 or $T = \ _{r;t}$ seconds. Intuitively, the proxy should favor the caching of resources with high fetching cost and/or small size. We define the *utility* of caching the resource for T seconds as

$$u_{r;t}(T) = \frac{f_{r;t}}{s_r \int_t^{t+T} (t)dt} :$$

When caching decisions are made with respect to a price-function and a threshold value, the resource is cached for a time period if its utility exceeds the threshold throughout the interval.

In reality, the proxy has only partial or statistical knowledge of fetching costs and next request time of cached objects. When making the caching decisions, we use *estimated utility* for caching a resource for a particular time period. The fetching cost $f_{r;t}$ can be based on the last fetching time, the network load on the path to the server, or the server load. The most challenging part of estimating resource utility is to obtain a good estimate of the next request time. The various

[1] Total caching cost under steady-state, divided by elapsed time.

cache replacement policies make implicit assumptions about this value, and their performance hinges on the quality of (absolute or relative) estimates for $_{r;t}$. For example, the LRU replacement policy assumes $f_{r;t}/s_r$ to be identical and fixed for all resources and utilizes an estimate on $_{r;t}$ that is decreasing with the last request time for r. The Greedy-Dual-Size algorithm [1] (generalization of LRU that incorporates fetching cost and sizes) can be viewed in the above framework as assuming a steady-state and substituting $t - t_r$ for $_{r;t}$, where t_r is the last request time of r.

Instead of assuming a particular expression for the next request time, we formulate an optimal request sequence when the time between requests is captured by a probability density function $_{r;t}(x)$ ($x \quad 0$) (probability density that the object is requested in time $t + x$) and let $L_{r;t}(x)$ be the corresponding distribution function. Let $f_r(x)$ be the expected fetching cost of r at time $t + x$. We compute the expected profit and cost of caching the resource for T time units (or up till its next request). The estimated utility of caching the object for T time units is the ratio of the expected profit and expected cost. The expected profit is

$$\text{profit}_{r;t}(T) = \int_0^T {}_{r;t}(x) f_r(x) dx$$

and the expected cost is

$$\text{cost}_{r;t}(T) = s_r \int_0^T (t+x)(1 - L_{r;t}(x)) dx :$$

The estimated utility of increasing caching time from T_1 to T_2 is:

$$\frac{\text{profit}_{r;t}(T_2) - \text{profit}_{r;t}(T_1)}{\text{cost}_{r;t}(T_2) - \text{cost}_{r;t}(T_1)} = \frac{\int_{T_1}^{T_2} {}_{r;t}(x) f_r(x) dx}{s_r \int_{T_1}^{T_2} (t+x)(1 - L_{r;t}(x)) dx} :$$

If V is the threshold then the caching time T of a resource can be predetermined as follows. Caching for T^0 time is *valid* if the estimated utility of increasing caching period from y to T^0 exceeds V for all $y \; 2 \; [0; T^0]$.

$$8y < T^0; \frac{\text{profit}_{r;t}(T^0) - \text{profit}_{r;t}(y)}{\text{cost}_{r;t}(T^0) - \text{cost}_{r;t}(y)} \quad V :$$

Among all valid T^0, we select T to maximize the expected profit, that is, we select the maximum valid T^0. Formally,

$$T = \arg\max_{T^0} \quad T^0 \text{ is valid} \tag{1}$$

The value of T serves as a expiration timeout for evicting resource r from the proxy cache.

In price-function-optimal cache replacement the goal is to maximize cache hits for a given caching cost. We notice tight correspondence between these trade-offs and optimal trade-offs of hits and cache size: (1) Consider an (offline)

instance of the Web caching problem where resource sizes are small relative to the cache size. There exists a price function such that optimal replacement with respect to it yields near-optimal hits-cache-size tradeoffs. These prices can be obtained by formulating (fractional) caching as a linear program, and setting the prices according to the dual optimal solution. (2) For paging with uniform fetching costs and resource sizes, the least-valuable resource is the one to be requested furthest in the future [9]. When varying fetching costs are introduced, there is no such total order on values of caching different resources. The price-function metric induces such an ordering. (3) Studies show that large proxy caches exhibit regular usage patterns and a definite diurnal cycle [10]. This suggests that the same price function is applicable on different days.

Related Work: Replacement policies were previously analyzed under some restriction or statistical assumptions on the data [11,12]. Lund et al. [13] and Keshav et al.[14] studied policies for virtual circuit holding time in IP over ATM networks, based on statistics of the packet inter-arrival sequence. In cache replacement context, these policies translate into considering each client-resource pair separately and utilizing the resource inter-request statistics to price the caching cost of a resource. Capturing and utilizing reference locality for replacement decisions was attempted in [15].

2.2 Utilizing Inter-Request Statistics

By receiving requests from a large number of clients, servers can estimate the distribution $_{r;t}(x)$ of the time between successive requests. In responding to a request for resource r, the server can piggyback the probability distribution (in a discretized form) to aid the proxy in its cache replacement decisions. The distribution is computed using a portion of the server logs. Consider discrete times $0 = t_0; t_1; \ldots; t_N$ and a proxy cache in steady-state ((t) constant) with fetching costs that do not change across time ($f_r(x)$ f_r). For a resource r let s_r be its size and f_r its fetching cost. Let n_r be the total number of requests for r under consideration in the server logs. Let $c_r(i)$ be the number of requests for r such that the previous request for r by same client preceded by t_i to t_{i-1} seconds. Let P_i be the price per unit-size of caching for t_i time. Then the estimated expected cost of caching for time-length t_i is approximately

$$\text{cost}_r(i) = \frac{s_r}{n_r}(P_i \sum_{j \quad i} c_r(j) + \sum_{j<i} c_r(j)P_j)$$

(in the experiments we compute the exact value, where the second term is replaced by the sum of prices per unit-size over all inter-request times smaller than t_i) and the estimated expected profit is

$$\text{profit}_r(i) = \frac{f_r}{n_r} \sum_{j \quad i} c_r(j) :$$

To compute the recommended time interval i we start with $i = 0$ and repeat: we look for the minimum $j > i$ such that

$$\frac{\text{profit}_r(j) - \text{profit}_r(i)}{\text{cost}_r(j) - \text{cost}_r(i)} \quad V$$

and set i j. If no such j exists, we stop.

The use of server-generated predictions implicitly assumes that the set of clients have similar access patterns. The server can increase the accuracy of the predictions by classifying various "client types" based on parameters such as their volume of traffic to the server and time-of-day at the proxy and the server. Statistics such as access frequency of resources can be collected for each class separately. The proxy identifies its "type" to the server (e.g., provides it with estimate on number of daily/weekly requests made to server resources) and server returns statistical information about the next request time, along with the request resource. The more accurately the server-generated distribution function fits the access patterns of a particular client/resource, and the less variance these distributions exhibit, the better are the replacement sequences generated. However, dividing the proxies into too many types also decreases the amount of data available for estimating distributions, which results in lower accuracy and higher computational complexity. (The extreme partition is assigning a type for each client. This is essentially a local policy that does not utilize the server's additional knowledge.)

The above formulation associates with each resource a distribution on its next request time. In reality, different resources are dependent, and conditional probabilities capturing these interdependencies may yield more accurate predictions. In [6], we proposed heuristics for constructing *volumes* that capture the pairwise dependencies between resources by observing the stream of requests at a server; measuring such dependencies was also suggested by [16,17] and further developed in [6]. Let p_{sjr} be the proportion of requests for resource r that are followed by a request for resource s by the same client within T seconds. Resource s is included in r's volume if p_{sjr} is greater than or equal to a threshold probability p_t. When a proxy requests resource r, the server constructs a piggyback message from the set of resources s with p_{sjr} p_t. Based on these predictions, the proxy can extend the expiration time (or adjust the priority) of resources that appear in the piggyback message and are stored in its cache. After estimating the implication probabilities p_{sjr}, the server can evaluate the potential effectiveness of piggyback information by measuring how often a piggyback message generates a *new* prediction for resource s (particularly important when an access to s is often preceded by a *sequence* of requests by the same proxy).

3 Performance Evaluation

To compare the *proxy-local* and *server-assisted* cache replacement policies, we evaluated the various schemes on three large server logs. The server logs can be viewed as providing a sequence of triples, with the requesting client, requested

resource, and request time. For this initial study, we assumed constant price functions and that all resources have the same size and fetching cost; hence, the profit is simply the number of cache hits and the cost is the total time resources spend in cache. We evaluated the cache hit ratio versus the mean cost per hit, averaged across all requests. These measures are computed by partitioning the server logs by clients, and computing the number of hits and the total time-in-cache for each resource and for each client. The latter quantities are then summed over clients and resources. The total hit ratio is obtained by dividing the total number of cache hits by the total number of requests. The cost per hit is obtained by dividing the total cost (object-seconds) by the number of hits.

3.1 Server Logs

The experiments use the access logs of three Web servers, from Amnesty International USA, Apache Group, and Sun Microsystems. The server logs represent a range of Web sites in terms of the number of resources and accesses, as shown in Table 3.1. Though the servers do not necessarily see the requests that were satisfied directly at the client or proxy caches, the logs do include all if-modified-since requests to validated cached copies of resources. Many clients have very short interactions with a server, resulting in a small number of requests. These clients experience very low cache hit rates, even under an optimal replacement policy. In the AIUSA log, only 11% of requests were for resources already requested by same client. The corresponding figures are 36% and 38% for the Apache and Sun logs, respectively. These numbers provide an upper bound on the cache hit ratio for all of the cache replacement policies.

Log (days)	Number of Requests	Number of Clients	Requests per Source	Unique Resources Considered
AIUSA (28)	180,324	7,627	23.64	1,102
Apache (49)	2,916,549	271,687	10.73	788
Sun (9)	13,037,895	218,518	59.66	29,436

Table 1. Server log characteristics

3.2 Cache Replacement Policies

We examined a variety of replacement policies, including an optimal omniscient policy, three proposed proxy-local policies, and our server-assisted policies. Each policy is described in terms of our framework, assuming constant price functions:

Optimal (Opt): The optimal replacement sequence caches all resources with a next request time within some fixed threshold value. Cost-performance trade-offs

are obtained across a range of threshold values. Opt provides a good yardstick for gauging other policies.

Fixed (LRU): LRU predicts a resource's next request time based only on the last request time. This results in a proxy-local policy that keeps all resources in the cache for the same, fixed, period of time. Cost-performance trade-offs are measured by varying the resource expiration times across a range of values from 40 seconds to 24 hours.

Exponentially averaged mean and variance (EMV): This proxy-local policy is based on the premise that current inter-request time interval of a resource is more correlated with recent inter-request times. This policy was evaluated by Keshav et al. [14] on circuit holding times. For a parameter $0 < \quad < 1$ (we used $\quad = 0{:}3$), the policy maintains exponentially averaged estimates on the mean and variance of the inter-request times: initially $\quad_0 = 0$ and $\quad_0 = 0$. After observing the kth inter-request time, t_k, we compute

$$\quad_{k+1} = \quad t_k + (1 - \quad) \quad_k$$
$$\quad_{k+1} = \quad j \quad_k - t_k j + (1 - \quad) \quad_k$$

A resource is cached for $\quad_k + 2 \quad_k$ time units, when this is below the threshold.

Local inter-request statistics (LIS): This is the policy outlined in Section 2.2, where the inter-request times histogram is constructed using locally-available (at the client) data. If fetching costs and resource sizes are uniform, this local policy reduces to the *adaptive policy* proposed in [13,14] for circuit holding times. Typically, resources are requested only a few times by any one client in the duration of the log. When predicting an inter-request interval for resources requested 15 or fewer times, we used a histogram containing all *other* inter-request time intervals. Otherwise, the histogram contained all inter-request times. Future inter-request times were included in order to account for the initial lack of history. Note that LIS does not cache resources requested two or fewer times. The histogram bin partition used to evaluate LIS, SIS and SIS-c policies (see below) consisted of 21 time intervals varying from 20 seconds to 24 hours.

Server inter-request statistics (SIS): The server-assisted policy outlined in Section 2.2. The server generates inter-request distributions for each resource r using statistics from all its clients. For each resource and threshold value U, there is a "recommended caching period" $t_r(U)$. Trade-offs are obtained by sweeping U.

Server inter-request statistics, filtered by client type (SIS-c): This policy is a refinement of SIS, where the server generates separate inter-request distributions and caching periods for different types of clients, based on how often they issue requests to the server. The statistics obtained for each type are applied to clients of that type. The experiments consider three classes for the AIUSA logs, four for the Apache logs, and five for the Sun logs.

Volume enhanced (VOL): This enhancement captures interdependencies between server resources, and can be combined with any of the other cache replacement policies. We evaluate the LRU+VOL policy, which is the LRU policy (each object is kept for the same amount of time) with the following enhancement: When a resource *r* appears in a piggybacked volume, and is currently cached, its expiration time is extended by its stated caching time. We also evaluate SIS+VOL, a similar enhancement of the SIS policy.

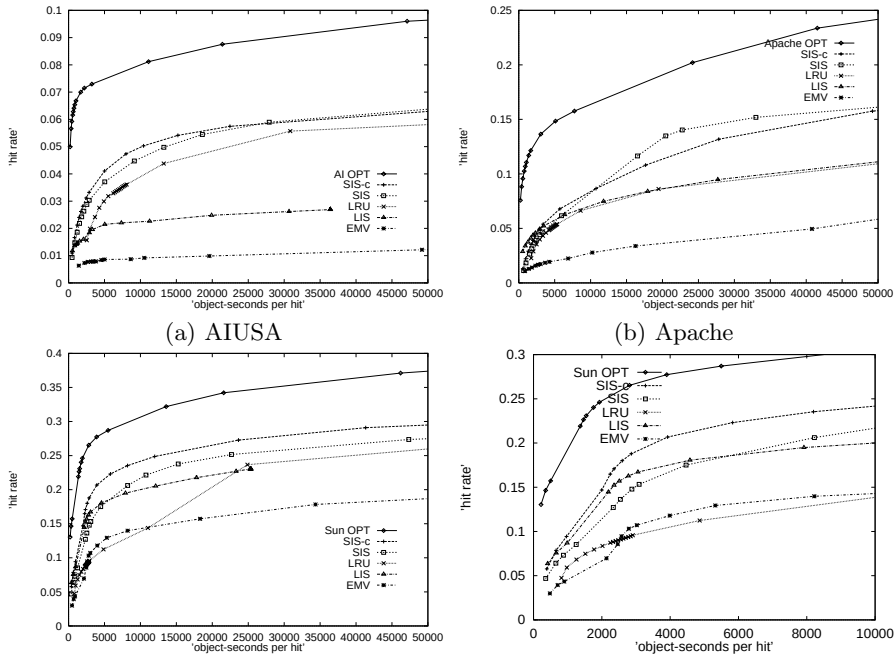
3.3 Cost-Performance Trade-O s

The graphs in Figure 1 compare the Opt, LRU, EMV, LIS, SIS, and SIS-c algorithms on all three server logs, across a range of thresholds. Each threshold value results in a single measure of performance (the hit rate on the y-axis) and overhead (object-seconds-per-hit on the x-axis). The cost per hit increases with the hit ratio for all of the replacement policies, since small improvements in the hit ratio incur progressively larger costs. As a result, even the OPT algorithm incurs a significant cost per hit as the hit ratio grows closer to the upper bound (11%, 36%, and 38%, respectively, for the AIUSA, Apache, and Sun logs). The graphs also show that the server-assisted SIS and SIS-c policies typically outperform the three proxy-local policies (LRU, EMV, and LIS). Consequently, server-assisted policies achieve a higher hit ratio for the same cache size, or the same hit ratio for a lower cache size, compared to proxy-local schemes. Even SIS, where the server supplies the same hints per-resource for all clients, outperforms the LRU scheme that treats all resources uniformly.

Among local policies, LRU outperforms the more involved LIS and EMV when the cost-per-hit is high. This seemingly counter-intuitive behavior is due to the large number of resource-client pairs with small number of requests. LIS does not cache a resource requested by a client twice or less, and EMV does not cache a resource on its first request. Hence, LIS and EMV are subject to a lower performance ceiling than LRU. Furthermore, when there are only a few requests, the selection of a caching period is based on very limited statistics and is less accurate. The above suggests that a local policy which combines LIS or EMV with a default non-zero caching period would enhance its performance.

The policies SIS, SIS-c, and LIS differ in the partition of the clients used to collect the inter-request intervals statistics. The granularity of the client partition corresponds to a tradeoff between the amount of statistics available and the applicability of the statistics to a particular client. The two extremes are LIS, which collects statistics locally at each client, and SIS which collects the same statistics for all clients. SIS-c differentiates between different types of clients, based on total number of requests.

On very low ranges of object-seconds-per hit (see the zoomed in portion of Figure 1), the local policy LIS outperforms other policies. This is more pronounced for the Apache and in particular the Sun logs, where client-resource interactions are on average longer (and LIS indeed exhibits better relative performance). The explanation is that resources that have smaller inter-request times, and hence, lower expected cost-per-hit, tend to be requested many times



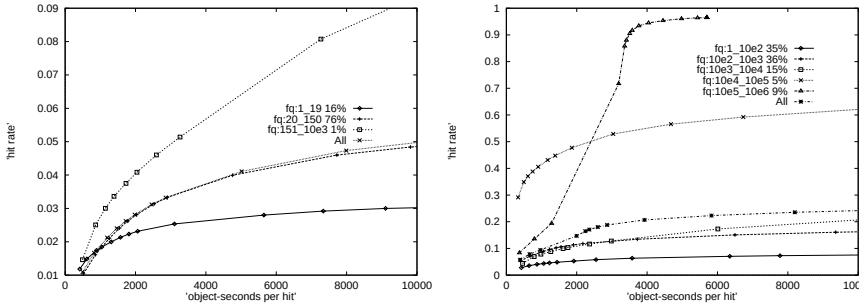
(a) AIUSA (b) Apache

(c) Sun (c1) Sun (zoomed in at low-cost hits)

Fig. 1. Hit rate vs. cost-per-hit for all six replacement policies

by clients and hence, there is a good amount of inter-request statistics available locally. As the cost-per-hit increases, SIS and SIS-c significantly outperform LIS, indicating that locally-available statistics are not sufficient. SIS-c typically outperforms SIS, substantiating the value of differentiating between client types. The above results suggest a combined policy likely to dominate all three: For each client, each resource is considered separately; if there is "enough" local statistics, we use LIS; otherwise, the client utilizes the statistics of its client-type (SIS-c) or even the statistics of all clients (SIS). The downside of the combined policy (and of LIS) is that each entity in the hierarchy needs to collect inter-request statistics.

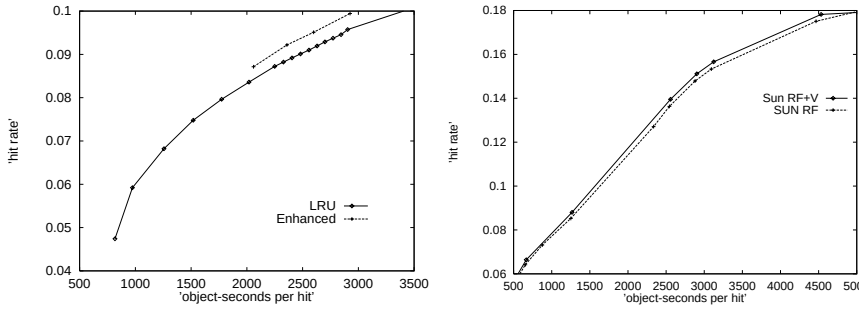
Figure 2 shows the performance of the SIS-c policy on various client classes. For example, the top curve in Figure 2(a) shows the cost-performance trade-offs for the top clients which accessed the AIUSA server 151–1000 times during the 28-day period. These clients also exhibit much higher hit rates than the other clients. This is also true under other replacement policies, such as OPT, since these clients are much more likely to access the same resource multiple times. These accesses stem from high-end users or, perhaps, from proxy sites that relay requests for multiple clients. It is precisely these clients or proxies that would most benefit from participating in an enhanced information exchange with the server sites.



(a) AIUSA (b) Sun

Fig. 2. Hit rate vs. cost-per-hit for SIS-c client classes

In Figure 3, we investigate the benefits of using server-generated volumes to extend the expiration time of cached resources. The piggyback-enhanced replacement policies utilize volumes obtained with time interval $T = 300$, probability threshold $p_t = 0.25$, and effective probability 0.2 [6]. Incorporating the piggyback information improves the performance of the LRU policy, as shown in Figure 3(a). The volume information augments the proxy-local policy with information about resource access patterns. This extra knowledge is less useful for the SIS policy, as shown in Figure 3(b), since the basic SIS scheme already gleans useful information from the server estimates of the inter-request times for individual resources. In fact, the basic SIS scheme even outperforms the volume-enhanced LRU policy, suggesting that accurate estimates of inter-request times may be more useful to the proxy than predictions about future accesses to other resources. Further experiments with other server logs and volume parameters (e.g., larger p_t) should lend greater insight into the cost-performance trade-offs of volume-enhanced cache replacement policies.



(a) LRU vs. LRU+VOL (b) RF vs. RF+VOL

Fig. 3. Volume-enhanced policies on the Sun log

4 Summary and Future Work

Our experiments implicitly assume that the servers see all or most of the client requests. This becomes less likely as proxy caching becomes more common, particularly when the proxy employs cache coherency policies that avoid generating If-Modified-Since requests to servers. By hiding client accesses from the server, these trends potentially limit the server's ability to generate accurate predictions of inter-request distributions and dependencies between resources. One approach is for the server to collect statistics based on the fraction of traffic due to low-volume clients. These clients are more likely to be individual users with only browser caching. As an extension to our server-assisted cache replacement model, we are also considering ways for participating proxies to summarize information about client requests that are satisfied in the cache. The server can accumulate these additional statistics across multiple proxy sites to generate better estimates of the inter-request distributions and resource dependencies. the piggyback response messages.

References

1. P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997. `http://www.cs.wisc.edu/~cao/papers/gd-size.html`.
2. S. Irani, "Page replacement with multi-size pages and applications to web caching," in *Proc. 29th Annual ACM Symposium on Theory of Computing*, 1997.
3. N. Young, "On line file caching," in *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, 1998.
4. L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," tech. rep., University of Pisa, January 1998. `http://www.iet.unipi.it/~luigi/lrv98.ps.gz`.
5. J. C. Mogul, "Hinted caching in the web," in *Proceedings of the 1996 SIGOPS European Workshop*, 1996. `http://mosquitonet.stanford.edu/sigops96/papers/mogul.ps`.
6. E. Cohen, B. Krishnamurthy, and J. Rexford, "Improving end-to-end performance of the Web using server volumes and proxy filters," in *Proceedings of ACM SIGCOMM*, September 1998. `http://www.research.att.com/~bala/papers/sigcomm98.ps.gz`.
7. B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the world wide web," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997. `http://www.research.att.com/~bala/papers/pcv-usits97.ps.gz`.
8. B. Krishnamurthy and C. E. Wills, "Piggyback server invalidation for proxy cache coherency," in *Proceedings of the World Wide Web-7 Conference*, April 1998. `http://www.research.att.com/~bala/papers/psi-www7.ps.gz`.
9. L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Systems Journal*, vol. 5, pp. 78–101, 1966.
10. S. D. Gribble and E. A. Brewer, "System design issues for Internet middleware services: Deductions from a large client trace," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997. `http://www.usenix.org/events/usits97`.

11. A. Borodin, S. Irani, P. Raghavan, and B. Schieber, "Competitive paging with locality of reference," in *Proc. 23rd Annual ACM Symposium on Theory of Computing*, 1991.
12. A. Karlin, S. Phillips, and P. Raghavan, "Markov paging," in *Proc. 33rd IEEE Annual Symposium on Foundations of Computer Science*, IEEE, 1992.
13. C. Lund, N. Reingold, and S. Phillips, "IP over connection oriented networks and distributional paging," in *Proc. 35th IEEE Annual Symposium on Foundations of Computer Science*, 1994.
14. S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran, "An empirical evaluation of virtual circuit holding time policies in IP-over-ATM networks," *Journal on Selected Areas in Communications*, vol. 13, October 1995.
`http://www.cs.cornell.edu/skeshav/doc/94/2-16.ps`.
15. A. Fiat and Z. Rosen, "Experimental studies of access graph based heuristics: Beating the LRU standard?," in *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, 1997.
16. A. Bestavros, "Using speculation to reduce server load and service time on the WWW," in *Proceedings of the ACM 4th International Conference on Information and Knowledge Management*, 1995.
`http://www.cs.bu.edu/faculty/best/res/papers/Home.html`.
17. V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," *Computer Communication Review*, vol. 26, no. 3, pp. 22–36, 1996.
`http://daedalus.cs.berkeley.edu/publications/ccr-july96.ps.gz`.

Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights*

D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni

[1] Max Planck Institut für Informatik, IM Stadtwald, 66123, Saarbrücken, Germany
[2] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, I-00198 Roma, Italy, falberto, frigioni, nannig@dis.uniroma1.it

Abstract. We study the problem of maintaining the distances and the shortest paths from a source node in a directed graph with arbitrary arc weights, when weight updates of arcs are performed. We propose algorithms that work for any graph and require linear space and optimal query time. If a negative–length cycle is added during *weight-decrease* operations it is detected by the algorithms. The algorithms explicitly deal with zero–length cycles. We show that, if the graph has a k-bounded accounting function (as in the case of graphs with genus, arboricity, degree, treewidth or pagenumber bounded by k, and k-inductive graphs) the algorithms require $O(k \ n \ \log n)$ worst case time. In the case of graphs with n nodes and m arcs $k = O(\sqrt{m})$; this gives $O(\sqrt{m} \ n \log n)$ worst case time per operation, which is better for a factor of $O(\sqrt{m} \log n)$ than recomputing everything from scratch after each update.
If we perform also insertions and deletions of arcs, then the above bounds become amortized.

1 Introduction

We study the dynamic single source shortest paths problem in a directed graph $G = (N; A)$ with real arc weights, n nodes and m arcs. The best known static algorithm for this problem takes $O(mn)$ time; it either detects a negative-length cycle, if one exists, or solves the shortest paths problem (see e.g. [1]).

The dynamic version of the problem consists of maintaining shortest paths while the graph is changing, without recomputing them from scratch. The most general repertoire of changes to the graph includes insertions and deletions of arcs, and update operations on the weights of arcs. When arbitrary sequences of the above operations are allowed we refer to the *fully dynamic problem*; if we consider only insertions (deletions) of arcs then we refer to the *incremental* (*decremental*) problem.

* Work partially supported by the ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244, and by *Progetto Finalizzato Trasporti 2* of the Italian National Research Council (CNR). The work of the first author was supported by the NATO – CNR Advanced Fellowships Program n. 215.29 of the CNR.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 320{331, 1998.
c Springer-Verlag Berlin Heidelberg 1998

In the case of positive arc weights several solutions have been proposed to deal with dynamic shortest paths [2,3,6,7,8,9,10,15,16]. However, neither a fully dynamic solution nor a decremental solution for the single source shortest paths problem on general graphs is known in the literature that is asymptotically better than recomputing everything from scratch. Also in the case of general graphs with arbitrary arc weights we are not aware of any solution that is provably better than recomputing everything from scratch. In this paper we propose a solution with a worst case bound of $O(\sqrt{m}\, n \log n)$ when the weight of an arc is changed. This is better for a factor of $O(\sqrt{m}/\log n)$ than recomputing everything from scratch after each update using the best known static algorithm.

The above bounds hold in the general case; they are even better if one of (or both) the following conditions holds: A) the graph satisfies some structural property; B) the update operation introduces a "small" change in the shortest paths tree. We now consider the above conditions in more detail.

A) The structural constraints that we consider have been introduced in the framework of dynamic algorithms in [8]. An *accounting function* F for G is a function that for each arc (x, y) determines either x or y as the *owner* of the arc; F is k-bounded if k is the maximum over all nodes x of the size of the set of arcs owned by x. An analogous notion, the *orientation* of arcs in an undirected graph with low out-degree, has been considered in [4,13].

The value of k for any graph G can be bounded in different ways (see e.g. [8,9]). For example, k is immediately bounded by the maximum degree of G. If G is d-inductive (see e.g. [11]), then k is bounded by d. Furthermore, k is bounded by the arboricity, the pagenumber and the treewidth of G. Moreover, it is possible to show that $k = O(1 + \sqrt{\gamma})$, where γ is the *genus* of G, by observing that the *pagenumber* of a genus γ graph is $O(\sqrt{\gamma})$ [12]; this implies that a graph with m arcs has a $O(\sqrt{m})$-bounded accounting function.

If we assume that the graph has a k-bounded accounting function (k-b.a.f. from now on), then the running time of the algorithms proposed in this paper become $O(n\ k \log n)$. We remark that the notion of k-b.a.f. is useful only to bound the running times, but does not affect the behavior of our algorithms.

In the case of bounded treewidth graphs in [3] a solution is given for the all-pairs shortest paths problem when weight updates of arcs are allowed. Their running times are better than ours on graphs with *small* treewidth; however their solution cannot deal with insertions and deletions of arcs.

B) The analysis of dynamic algorithms using the *output complexity* model has been introduced by Ramalingam and Reps in [14,15] and it has been subsequently modified by the authors of this paper in [8]. In [14,15] Ramalingam and Reps propose also the only fully dynamic solution for shortest paths on digraphs with arbitrary arc weights known in the literature. In this solution they assume the digraph has no negative-length cycles before and after any input change. In addition, they do not deal with zero-length cycles.

Following the model of [8], in the case of the single source shortest paths problem for a digraph G with source s, the *output information* consists of: *(i)* for any node x, the value of the distance of x from s; *(ii)* a shortest paths tree

rooted in s. Let be an arc operation to be performed on G (insertion, deletion, or weight update), and G^0 be the new graph after that has been performed. The *set of output updates* $U(G;$) on the solution of the problem is given by the set of nodes that either change their distance from s in G^0, or change their parent in the shortest paths tree, due to . The *number of output updates* caused by is the cardinality of $U(G;$). This notion of output complexity can be extended to compute *amortized* costs [8].

If the digraph has a k-b.a.f., then our algorithms require $O(k \log n)$ time per output update in the case of *weight-decrease* operations. When the weight of an arc $(x; y)$ is increased we observe that if $(x; y)$ is an arc belonging to the shortest paths tree, then the set of output updates depends on the existence of alternative paths; in fact it does not necessarily include all nodes belonging to the subtree of the shortest paths tree rooted at y. In this case the running time of the update procedure is $O(k \log n)$ per output update plus time linear in the size of the subtree of the shortest paths tree rooted at y.

The cost of computing the minimum k such that G admits a k-b.a.f. can be hard; however, a simple greedy algorithm allows to compute a 2-approximation in $O(m + n \log n)$ time. Using this 2-approximation the cost of our algorithms increases at most of a factor of 2.

In this paper we use the following techniques introduced in previous papers [8,9]: the idea of k-b.a.f. and the one of associating a potential function to arcs. Roughly speaking the potential function that is associated to each arc allows to run a Dijkstra-like algorithm [5] only on the subgraph that is affected by an input change. However, the potential function used in [8,9] does not allow to deal with negative weights and negative-length cycles. The original contribution of this paper is the following:

1. We explicitly deal with negative-length cycles that might be introduced by an operation that decreases the weight of an arc or that adds an arc to the graph.
2. In the case of deletions or of weight increase operations the main difficulty is given by the presence of zero-length cycles. Our algorithm computes the set of nodes affected by the operation and detects the zero-length cycles including affected nodes. Then the new shortest paths tree is recomputed only on the subgraph induced by the affected nodes. Since zero-length cycles have never been handled before, we believe that this part is interesting on its own.
3. We introduce a new potential function that allows us to compute the new distances after an input change to the graph without using a label correcting algorithm, but running a Dijkstra-like algorithm [5] after that the set of nodes affected by the input change has been determined.

In the sequel we present only the operations that decrease or increase the weight of an arc. The extension to the general case in which also insertions and deletions of arcs are allowed is based on techniques developed in [9], and will be presented in the full version of the paper. This is done without recomputing the k-b.a.f. from scratch every time the graph is modified, but using a simple scheme for resetting ownerships; in this way the proposed bounds become amortized. Furthermore, for lack of space all the proofs will be given in the full paper.

2 Preliminaries

Let $G = (N; A)$ be a weighted directed graph (digraph) with n nodes and m arcs, and let $s \in N$ be a fixed *source* node. For each $z \in N$, we denote as $\text{IN}(z)$ and $\text{OUT}(z)$, the arcs of A incoming and outgoing z, respectively. To each $(x; y) \in A$, a real weight $w_{x;y}$ is associated. A *path* in G is a sequence of nodes $\langle x_1; x_2; \ldots; x_r \rangle$ such that $(x_i; x_{i+1}) \in A, \forall i = 1; 2; \ldots; r-1$. The length of a path is the sum of the weights of the arcs in the path. A cycle is a path $\langle x_1; x_2; \ldots; x_r \rangle$ such that $(x_i; x_{i+1}) \in A, \forall i = 1; 2; \ldots; r-1$ and $(x_r; x_1) \in A$. A negative–length (zero–length) cycle is a cycle C such that the sum of the weights of the arcs in C is negative (zero).

If the graph does not contain negative–length cycles then we denote as $d : N \to <$ the distance function that gives, for each $x \in N$, the length of the shortest path from s to x in G. Given two nodes x and y we denote as $l(x; y)$ the length of the shortest path between x and y. $T(s)$ denotes a shortest paths tree rooted at s; for any $x \in N$, $T(x)$ is the subtree of $T(s)$ rooted at x. We now recall the well known condition that states the optimality of a distance function d on $G = (N; A)$. For each $(z; q) \in A$, the following *optimality condition* holds: $d(q) \leq d(z) + w_{z;q}$.

For each $z \in N$, $d(z)$ and $d'(z)$ denote the distance of z before and after an arc modification, respectively. The new shortest paths tree in the graph G' obtained from G after an arc operation, is denoted as $T'(s)$.

Definition 1. *Given $G = (N; A)$ and $z \in N$, the* backward_level *$b\_level_z(q)$ of arc $(z; q) \in \text{OUT}(z)$ is given by $d(q) - w_{z;q}$; the* forward_level *$f\_level_z(v)$ of arc $(v; z) \in \text{IN}(z)$ is given by $d(v) + w_{v;z}$.*
After an arc update on G, the variation *of distance of $z \in N$ from s is $\Delta(z) = d'(z) - d(z)$.*

In order to bound the number of arcs scanned by our algorithms, for each node z, we partition each of the sets $\text{IN}(z)$ and $\text{OUT}(z)$ in two subsets as follows. Any $(x; y) \in A$ has an *owner* that is either x or y. For each $x \in N$, $\text{IN-OWN}(x)$ denotes the subset of $\text{IN}(x)$ containing the arcs owned by x, and $\overline{\text{IN-OWN}}(x)$ denotes the set of arcs in $\text{IN}(x)$ not owned by x. Analogously, $\text{OUT-OWN}(x)$ and $\overline{\text{OUT-OWN}}(x)$ represent the arcs in $\text{OUT}(x)$ owned and not owned by x, respectively. We say that G has a k-b.a.f. if both $\text{IN-OWN}(x)$ and $\text{OUT-OWN}(x)$ contain at most k arcs.

We use the following additional data structures. For each node x, $D(x)$ and $P(x)$ store the distance and the parent of x in the shortest paths tree, respectively. $D(x)$ satisfies the following property: $D(x) = d(x)$ ($D(x) = d'(x)$) before (after) the execution of the proposed algorithms. Furthermore, $\Delta(x)$ stores the computed value of $\Delta(x)$; before and after the execution of the algorithms $\Delta(x) = 0$. The arcs in $\text{IN-OWN}(x)$ and in $\text{OUT-OWN}(x)$ are stored in two linked lists, each containing at most k arcs. The arcs in $\overline{\text{IN-OWN}}(x)$ and in $\overline{\text{OUT-OWN}}(x)$ are stored in two priority queues as follows:

1. $\overline{\text{IN-OWN}}(x)$ is a min-based priority queue where the priority of arc $(y; x)$ (of node y), denoted as $f_x(y)$, is the computed value of $f\_level_x(y)$.

2. $\overline{\text{OUT-OWN}}(x)$ is a max-based priority queue where the priority of arc $(x;y)$ (of node y), denoted as $b_x(y)$, is the computed value of $b\_level_x(y)$;

3 Decreasing the Weight of an Arc

In this section we show how to maintain a shortest paths tree of a digraph $G = (N;A)$ with arbitrary arc weights, after decreasing the weight of an arc. We assume that the graph before the *weight-decrease* operation does not contain negative–length cycles; if the *weight-decrease* operation does not introduce a negative–length cycle, Procedure `Decrease`, shown in Fig. 1, properly updates the current shortest paths tree, otherwise it detects the negative–length cycle introduced and halts.

```
     procedure Decrease(x;y : node;   : positive_real)
1.   begin
2.   w_{x;y}    - w_{x;y} -
3.   if D(x) + w_{x;y} - D(y) < 0 then
4.      begin
5.         Q    - ; finitialize an empty heap Qg
6.           (y)    - D(x) + w_{x;y} - D(y)
7.         Enqueue(Q;hy;   (y);xi)
8.         while Non_Empty(Q) do
9.            begin
10.              hz;   (z);qi    - Extract_Min(Q)
11.              D(z)    - D(z) +    (z)
12.              P(z)    - q
13.              for each arc (v;z) 2 IN-OWN(z) do b_v(z)    - D(z) - w_{v;z}
14.              for each arc (z;v) 2 OUT-OWN(z) do f_v(z)    - D(z) + w_{z;v}
15.              for each (z;h) 2 OUT-OWN(z) and
                    for each (z;h) 2 OUT-OWN(z) s.t. b_z(h) > D(z) do
16.                    if D(z) + w_{z;h} - D(h) <    (h) then
17.                       begin
18.                          if h = x
19.                             then EXIT fa negative cycle has been detectedg
20.                             else begin
21.                                        (h)    - D(z) + w_{z;h} - D(h)
22.                                   Heap_Insert_or_Improve(Q;hh;   (h);zi)
23.                                end
24.                       end
25.           end
26.     end
27.  end
```

Fig. 1. Decrease by positive quantity the weight of arc $(x;y)$

Assume that the weight of arc $(x;y)$ is decreased by a positive quantity . It is easy to see that, if $d(x) + w_{x;y} - d(y)$, then no node changes its distance from

s. On the other hand, if $d(x)+w_{x;y} - \ < d(y)$ then all the nodes in $T(y)$ decrease their distance of the same quantity of y. In addition, $T^0(y)$ may include other nodes not contained in $T(y)$. Each of these nodes decreases its distance from s of a quantity which is at most the reduction of y's distance. We denote as *red* the nodes that decrease their distance from s after a *weight-decrease* operation, and as n_R the number of *red* nodes. The following facts can be easily proved:

F1) If node y decreases its distance from s after decreasing the weight of $(x;y)$, then the new shortest paths from s to the *red* nodes will contain arc $(x;y)$; if y does not decrease its distance from s then no negative–length cycle is added to G, and all the nodes preserve their shortest distance from s.

F2) Node x reduces its shortest distance from s after decreasing the weight of arc $(x;y)$ if and only if the *weight-decrease* operation introduces a negative–length cycle; in this case $(x;y)$ belongs to C.

Let us consider the nontrivial case where $d^0(y) < d(y)$. In order to update the distances of *red* nodes we adopt a strategy similar to that of Dijkstra's algorithm on the subgraph induced by the *red* nodes. In particular, the *red* nodes are inserted in a heap Q. The presence of arcs with negative weights implies that, if we want to use a Dikstra-like algorithm, the priority of a node in Q cannot be the lenght of the path found by the procedure. However we will see that if the priority of z in Q is (z), that is an estimate of the (negative) variation $(z) = d^0(z) - d(z)$, then a Dijkstra-like procedure is sufficient to update the shortest paths tree.

Namely, at the beginning of procedure `Decrease` (see Fig. 1), node y is inserted in Q with priority $(y) = D(x)+w_{x;y} - \ - D(y)$. Then, in the main **while** loop a Dijkstra-like algorithm is performed starting from y. This is motivated by Fact *F1*. In particular, the node z with minimum priority (z) is dequeued from Q, and the new distance from s is computed as $d^0(z) = D(z) + \ (z)$, and stored in $D(z)$. At this point, both for the arcs $(z;h)$ in OUT-OWN(z), and for the arcs $(z;h)$ in $\overline{\text{OUT-OWN}}(z)$ such that $b_z(h) > d^0(z)$, the priority of h and v in Q is possibly updated (i.e., if $d^0(z) + w_{z;h} - D(h) < \ (h)$), as well as the current parent. If any improvement is determined for node x, then by Fact *F2*, a negative-length cycle is detected.

Using (z) as the priority of node z in Q, we are able to show that: *i*) $(z) \ (z)$, i.e., (z) is an upper bound on the actual variation of the distance of z from s; *ii*) nodes are extracted from Q in nondecreasing order of priority; *iii*) each node is inserted in (extracted from) Q exactly once. The correctness of Procedure `Decrease` is based on points *i*), *ii*) and *iii*) above, and on the following lemma.

Lemma 1. *Let z be any node of G such that $(z) = d^0(z) - d(z) < 0$ after a weight decrease operation on $(x;y)$. If $hy = z_0;z_1;:::;z_p = zi$ is the portion of any shortest path from s to z in G^0 starting from y, then for $i = 1;2;:::;p$, $(z_{i-1}) \ (z_i)$:*

The following theorem gives the complexity bounds of Procedure `Decrease`.

Theorem 1. *Let $G = (N; A)$ be a digraph with arbitrary arc weights. If G has a k-b.a.f., then it is possible to update $T(s)$ and the distances of nodes from s, or to detect the introduction of a negative{length cycle in G after a weight-decrease operation, in $O(n_R \ k \ \log n)$ worst case time.*

4 Increasing the Weight of an Arc

In the following we assume that before the *weight-increase* operation the data structures store the correct values, i.e., the array P induces a shortest paths tree rooted in s and, for each $z \ 2 \ N$, $D(z) = d(z)$. If the weight of an arc $(x; y)$ is increased, no negative–length cycle can be introduced in G as a consequence of that operation. On the other hand we allow the presence of zero–length cycles and deal explicitly with them. Furthermore, it is easy to see that: i) for each node $z \ 2 \ T(y)$, $d^0(z) = d(z)$; ii) there exists a new shortest path tree $T^0(s)$ such that, for each $z \ 2 \ T(y)$ the old parent in $T(s)$ is preserved.

We define a coloring of the nodes of G, depending on the algorithm, in order to distinguish how nodes are affected by a *weight-increase* operation, as follows:

$q \ 2 \ N$ is *white* if and only if q changes neither the distance from s nor the parent in $T(s)$;

$q \ 2 \ N$ is *red* if and only if q increases the distance from s, i.e., $d^0(q) > d(q)$;

$q \ 2 \ N$ is *pink* if and only if q maintains its distance from s, but changes the parent in $T(s)$.

$q \ 2 \ N$ is *blue* if q belongs to a zero–length cycle that is detected by the algorithm.

It is easy to verify that, if q is *red* then all the children of q in $T(s)$ must be updated and will be either *pink* or *red*; on the contrary, if q is *pink* or *white* then all nodes in $T(q)$ are *white*. Observe that a node is colored *pink* depending on the possibility of finding alternative paths with the same length of the shortest path before the *weight-increase* operation; it follows that a node q can be colored *white* even if the shortest path from s to q in G is not a shortest path in G^0; it is sufficient that there exists one shortest path from s to q in G^0 of the same length of the shortest path from s to q in G, where node q has the same parent as before. We also observe that a node colored *blue* will be later colored again with a different color; on the other side when a node is colored *white*, *pink* or *red* the color will not be changed anymore.

The set of output updates $U(G; \ )$ determined by a *weight-increase* operation is given by the *red* and *pink* nodes. We remark that the size of $U(G; \ )$ depends on the solution found by the algorithm. In fact, a nonred node might be colored either pink or white depending on the specific shortest path found by the algorithm. Hence, it is possible to have different sizes for the set of output updates, due to the input modification .

Algorithm `Increase` (Fig. 4), works in two phases. In the first phase it uses Procedure `Color`, shown in Fig. 2, that colors the nodes in $T(y)$ after increasing the weight of arc $(x; y)$, according to the above described rules, and gives a new parent in the shortest paths tree to each node which is colored *pink*. This is done

by inserting the nodes in a heap M, extracting them in non-decreasing order of their old distance from the source, and searching an alternative shortest path from the source. In the second phase Procedure `Increase` properly updates the information on the shortest paths from s for all the nodes that have been colored *red*, by performing a computation analogous to Dijkstra's algorithm.

Definition 2. *Let $G = (N, A)$ be a digraph in which the weight of arc (x, y) has been increased. A node q is a* candidate parent *of a node p if arc (q, p) belongs to a shortest path from s to p in G (i.e., $d(p) = d(q) + w_{q,p}$). Node q is an* equivalent parent *for p if it is a candidate parent of p and $d^0(q) = d(q)$.*

```
    procedure Color(y : node)
1.  begin
2.      M := ∅; {M is an heap}
3.      Heap_Insert(M, y, D(y));
4.      while Non_Empty(M) do
5.          begin
6.              z := Extract_Min(M);
7.              if z is uncolored then
8.                  Search_Equivalent_Path(z, y)
9.          end
10.     color red each arc with both endpoints red
11. end
```

Fig. 2. Color the nodes in $T(y)$ after increasing the weight of arc (x, y)

Procedure `Color` uses Procedure `Search_Equivalent_Path`, shown in Fig. 3, which searches for a path from s to a node z in G^0 whose length is equal to $D(z)$. In order to achieve this goal Procedure `Search_Equivalent_Path` uses a stack Q which is initialized with node z. During the execution of the procedure Q contains a set of nodes $q_1, q_2, \ldots, q_k$ ($q_k = Top(Q)$) such that q_i is a candidate parent of q_{i-1}, for $i = 2, 3, \ldots, k$. The nodes in Q are either uncolored or *blue*, and hence they represent nodes that are waiting that their current candidate parent is correctly colored.

Let us define the *best nonred neighbor* of a node z as the node q such that $(q, z) \in \text{IN}(z)$, q is nonred, and the shortest path from s to z passing through q is the minimum among those passing through the nonred neighbors of z. After initializing Q with z, `Search_Equivalent_Path` searches the best nonred neighbor of z in IN-OWN(z) and in $\overline{\text{IN-OWN}}(z)$, respectively; finally, it chooses q as the best between the two neighbors found. If q belongs to $T(z)$ or to the subtree of $T(y)$ rooted at some node currently in the stack, then a zero–length cycle has been detected; then the procedure colors *blue* the nodes in that cycle (this is done in order to avoid multiple visits of the same node) and considers another candidate parent of z.

A number of cases may arise once we have found the best nonred neighbor q of z. If q is not a candidate parent of z, and z is not *blue*, then there is no

```
    procedure Search_Equivalent_Path(z; y : node)
1.  begin
2.     Q  – ; {Q is a stack}
3.     Push(Q; z)
4.     repeat
5.        p  – Top(Q)
6.        let q be the next best nonred neighbor of p
7.        if q does not exist or q is not a candidate parent for p
8.           then begin
9.                   Pop(Q)
10.                  if p is not blue then
11.                     begin
12.                        color(p)  – red
13.                        for each v 2 children(p) do Heap_Insert(M; hv; D(v)i)
14.                     end
15.               end
16.          else if q 62 T(y) or q is pink or q is white
17.                  then
18.                     begin
19.                        w  – q
20.                        repeat
21.                           v  – Top(Q)
22.                           Pop(Q)
23.                           if (w; v) 2 T(s) then color(v)  – white
24.                              else begin
25.                                      color(v)  – pink
26.                                      P(v)  – w
27.                                   end
28.                           color white all nodes in T(v)
29.                           w  – v
30.                        until Q becomes empty
31.                        color white or pink all remaining blue nodes
32.                     end
33.                  else if q 2 Q or q is blue {a zero–length cycle is detected}
34.                           then color blue all nodes in Q
35.                           else Push(Q; q) {q 2 T(y) and q is uncolored}
36.    until Q becomes empty
37.    color red all blue nodes
38. end
```

Fig. 3. Search an alternative parent for node z in T(s)

alternative path from s to z in G^0 with length $D(z)$ and, therefore, z is colored *red*. Then all the children of z in $T(y)$ are inserted in M and will be colored later either *red* or *pink*. On the other hand, if q is not a candidate parent of z and z is *blue*, then z belongs to a zero–length cycle. In this case we cannot give a final color to z; hence z is deleted from Q, and it will be given a final color later.

```
    procedure Increase(x; y : node;   : positive_real)
1.  begin
2.     w_{x;y}   - w_{x;y} +
3.     if (x; y) 2 T(s)
4.       then update either b_x(y) or f_y(x) (depending on owner(x; y)) and EXIT
5.     Color(y)
6.     H   - ; finitialize an empty heap Hg
7.     for each red node z do
8.       begin
9.          let p be the next best nonred neighbor of z
10.         if p 6= Null
11.           then begin
12.                      (z)   - D(p) + w_{p;z} - D(z)
13.                   Enqueue(H; hz;   (z); pi)
14.                end
15.           else begin
16.                      (z)   - + 1
17.                   Enqueue(H; hz;   (z); Nulli)
18.               end
19.      end
20.    while Non_Empty(H) do
21.      begin
22.         hz;   (z); qi   - Extract_Min(H)
23.         D(z)   - D(z) +   (z)
24.         P(z)   - q
25.         for each arc (v; z) 2 IN-OWN(z) do b_v(z)   - D(z) - w_{v;z}
26.         for each arc (z; v) 2 OUT-OWN(z) do f_v(z)   - D(z) + w_{z;v}
27.         for each red arc (z; h) leaving z do
28.           if D(z) + w_{z;h} - D(h) <   (h)
29.                then begin
30.                           (h)   - D(z) + w_{z;h} - D(h)
31.                        Heap_Improve(H; hh;   (h); zi)
32.                end
33.         uncolor all the red arcs (q; z) entering z
34.      end
35. end
```

Fig. 4. Increase by positive quantity the weight of arc $(x; y)$

If q is a candidate parent of z and q does not belong to $T(y)$, or it has been already colored either *pink* or *white* then it is an equivalent parent for z. In fact, in this case we have found a shortest path from s to z passing through q whose

length is $D(z)$: z must be colored either *white* or *pink*, depending on whether or not node q was the parent of z in $T(s)$ before the *weight-increase* operation.

If q is *blue* or $q \in Q$ then a zero–length cycle has been detected, and therefore all nodes currently in Q are colored *blue*. Finally, if q is uncolored, then q represents a candidate parent of z, but the algorithm is not able to determine whether q is an equivalent parent of z, because it could change its distance later. In this case the search continues, node q is pushed in Q, and the algorithm looks for an equivalent parent of q, that now is the top node in the stack.

Observe that a *blue* node will be colored again; in fact, when the node on the top of Q is colored either *pink* or *white*, then all the nodes in the stack and all the *blue* nodes have found a path from s of the same length of the previous one, and hence they are colored either *pink* or *white*, depending on whether or not they change their parent in $T(s)$. Moreover the last step of the procedure colors *red* all remaining *blue* nodes. The correctness of Procedure `Color` is based on the following lemma.

Lemma 2. *Let $G = (N; A)$ be a digraph with arbitrary arc weights, and assume that a weight increase operation is performed on arc $(x; y)$.*

1. *If p and q are two nodes in $T(y)$ such that p and q belong to a zero{length cycle, then p and q are colored either both red or both nonred by Procedure* `Search_Equivalent_Path`.
2. *Procedure* `Color` *colors a node $z \in T(y)$ red if and only if $d'(z) > d(z)$.*

We now present procedure `Increase`, that, roughly speaking is Dijkstra's algorithm applied to the subgraph of G' induced by *red* arcs. Initially, the *red* nodes are inserted in a heap H. The main difference with respect to the standard Dijkstra's algorithm is the priority given to each *red* node z in H, which is (z) instead of $D(z)$. Then the procedure repeatedly extracts node z with minimum priority from H and updates its distance label. In this case, for each red arc $(z; h) \in \text{OUT}(z)$, Procedure `Heap_Improve` updates the priority associated to h in H (if required), in order to restore the optimality condition.

The correctness of procedure `Increase` is based on the following lemma, analogous to the one we have given for a *weight-decrease* operation.

Lemma 3. *Let z be any node of G with variation $(z) = d'(z) - d(z) > 0$ after that a weight-increase operation is performed on G. If $hs = z_0; z_1; \ldots; z_p = zi$ is a shortest path from s to z in G', then for $i = 1; 2; \ldots; p$, we have: $(z_{i-1}) \quad (z_i)$:*

Let n_R, n_P and n_W be the number of nodes that have been colored *red*, *pink*, and *white*, respectively, at the end of Procedure `Color`, and let m_R, $m_R \quad k \quad n_R$, be the number of *red* arcs, i.e., arcs whose both endpoints are *red*.

Theorem 2. *Let $G = (N; A)$ be a digraph with arbitrary arc weights. If G has a k-b.a.f., then it is possible to update $T(s)$ and the distances of nodes from s after a weight-increase operation, in $O((n_R + n_P)k \log n + n_W)$ worst case time.*

Combining Theorems 1 and 2 and the fact that a digraph with m arcs has a $O(\sqrt{m})$-bounded accounting function, we obtain the following corollary.

Corollary 1. *It is possible to update $T(s)$ and the distances of nodes from s in a general digraph after a weight-decrease or a weight-increase operation in $O(n\sqrt{m}\log n)$ worst case time.*

References

1. R. K. Ahuia, T. L. Magnanti and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ (1993).
2. G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, **12**, 4 (1991), 615–638.
3. S. Chaudhuri and C. D. Zaroliagis. Shortest path queries in digraphs of small treewidth. *Proc. Int. Coll. on Automata Languages and Programming. Lecture Notes in Computer Science* 944, 244–255, 1995.
4. M. Chrobak and D. Eppstein, Planar orientations with low out-degree and compaction of adjacency matrices, *Theoretical Computer Science*, **86** (1991), 243–266.
5. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, **1** (1959), 269–271.
6. S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, **49** (1985), 371–387.
7. P. G. Franciosa, D. Frigioni, R. Giaccio. Semi dynamic shortest paths and breadth-first search on digraphs. *Proc. Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science* 1200, 26–40, 1997.
8. D. Frigioni, A. Marchetti-Spaccamela and U. Nanni. Semi dynamic algorithms for maintaining single source shortest path trees. *Algorithmica* - Special Issue on Dynamic Graph Algorithms, to appear.
9. D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. Fully dynamic output bounded single source shortest path problem. *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 212–221, 1996.
10. P. N. Klein, S. Rao, M. Rauch and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Proc. ACM Symposium on Theory of Computing*, 27–37, 1994.
11. S. Irani. Coloring inductive graphs on-line. *Proc. IEEE Symposium on Foundations of Computer Science*, 470–479, 1990.
12. S. M. Malitz. Genus g graphs have pagenumber $O(\sqrt{g})$. *Journal of Algorithms*, **17** (1994), 85–109.
13. C. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, **36** (1961), 445–450.
14. G. Ramalingam. Bounded incremental computation. *Lecture Notes in Computer Science* 1089, 1996.
15. G. Ramalingam and T. Reps, On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, **158**, (1996), 233–277.
16. H. Rohnert. A dynamization of the all-pairs least cost path problem. *Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science* 182, 279–286.

A Functional Approach to External Graph Algorithms

James Abello, Adam L. Buchsbaum, and Jeffery R. Westbrook

AT&T Labs, 180 Park Ave., Florham Park, NJ 07932, USA
fabello,alb,jeffwg@research.att.com,
http://www.research.att.com/info/fabello,alb,jeffwg

Abstract. We present a new approach for designing external graph algorithms and use it to design simple external algorithms for computing connected components, minimum spanning trees, bottleneck minimum spanning trees, and maximal matchings in undirected graphs and multi-graphs. Our I/O bounds compete with those of previous approaches. Unlike previous approaches, ours is purely functional—without side effects—and is thus amenable to standard checkpointing and programming language optimization techniques. This is an important practical consideration for applications that may take hours to run.

1 Introduction

We present a divide-and-conquer approach for designing external graph algorithms, i.e., algorithms on graphs that are too large to fit in main memory. Our approach is simple to describe and implement: it builds a succession of graph transformations that reduce to sorting, selection, and a recursive bucketing technique. No sophisticated data structures are needed. We apply our techniques to devise external algorithms for computing connected components, minimum spanning trees (MSTs), bottleneck minimum spanning trees (BMSTs), and maximal matchings in undirected graphs and multi-graphs.

We focus on producing algorithms that are purely functional. That is, each algorithm is specified as a sequence of functions applied to input data and producing output data, with the property that information, once written, remains unchanged. The function is then said to have no "side effects." A functional approach has several benefits. External memory algorithms may run for hours or days in practice. The lack of side effects on the external data allows standard checkpointing techniques to be applied [16,19], increasing the reliability of any real application. A functional approach is also amenable to general purpose programming language transformations that can reduce running time. (See, e.g., Wadler [22].) We formally define the *functional I/O model* in Section 1.1.

The key measure of external memory graph algorithms is the disk I/O complexity. For the problems mentioned above, our algorithms perform $O(\frac{E}{B} \log_{M/B} \frac{E}{B} \log_2 \frac{V}{M})$ I/Os, where E is the number of edges, V the number of vertices, M the size of main memory, and B the disk block size. The BMST and maximal matching results are new. The asymptotic I/O complexities of our connected components and MST algorithms match those of Chiang et al. [8]. Kumar and Schwabe [15] give algorithms for breadth-first search (which can compute connected components) and MSTs that perform $O(V + \frac{E}{B} \log_2 \frac{E}{B})$ and $O(\frac{E}{B} \log_{M/B} \frac{E}{B} \log_2 B + \frac{E}{B} \log_2 V)$ I/Os, respectively. Our connected components algorithm is asymptotically better when $V < M^2/B$, and our

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 332–343, 1998.
c Springer-Verlag Berlin Heidelberg 1998

MST algorithm is asymptotically better when $V < MB$. While the above algorithms of Chiang et al. [8] are functional, those of Kumar and Schwabe [15] are not. Compared to either previous approach, our algorithms are simpler to describe and implement.

We also consider a *semi-external* model for graph problems, in which the vertices but not the edges fit in memory. This is not uncommon in practice, and when vertices can be kept in memory, significantly more efficient algorithms are possible. We design new algorithms for external grouping and sorting with duplicates and apply them to produce better I/O bounds for the semi-external case of connected components.

We begin below by describing the I/O model. In Section 2, we sketch two previous approaches for designing external graph algorithms. In Section 3, we describe our functional approach and detail a suite of simple graph transformations. In Section 4, we apply our approach to design new, simple algorithms for computing connected components, MSTs, BMSTs, and maximal matchings. In Section 5, we consider semi-external graph problems and give improved I/O bounds for the semi-external case of connected components. We conclude in Section 6.

1.1 The Functional I/O Model

We adapt the I/O model of complexity as defined by Aggarwal and Vitter [1]. For some problem instance, we define N to be the number of items in the instance, M to be the number of items that can fit in main memory, B to be the number of items per disk block, and $b = \lfloor M/B \rfloor$. A typical compute server might have $M \approx 10^9$ and $B \approx 10^3$.

We assume that the input graph is presented as an unordered list of edges, each edge a pair of endpoints plus possibly a weight. We define V to be the number of vertices, E to be the number of edges, and $N = V + E$. (We abuse notation and also use V and E to be the actual vertex and edge sets; the context will clarify any ambiguity.) In general, $1 < B \ll M < N$. Our algorithms work in the single-disk model; we discuss parallel disks in Section 6. For the connected components problem, the output is a *delineated list* of component edges, $\{C_1, C_2, \ldots, C_k\}$, where k is the number of components: each C_i is the list of edges in component i, and the output is the file of C_is catenated together, with a separator record between adjacent components. For the MST and BMST problems the output is a delineated list of edges in each tree in the spanning forest. For matching, the output is the list of edges in the matching.

Following Chiang et al. [1] we define $\mathit{scan}(N) = \lceil N/B \rceil$ to be the number of disk I/Os required to transfer N contiguous items between disk and memory, and we define $\mathit{sort}(N) = \Theta(\mathit{scan}(N) \log_b \frac{N}{B})$ to be the number of I/Os required to sort N items. The I/O model stresses the importance of disk accesses over computation for large problem instances. In particular, time spent computing in main memory is not counted.

The *functional I/O (FIO) model* is as above, but operations can make only functional transformations to data, which do not change the input. Once a disk cell, representing some piece of state, is allocated and written, its contents cannot be changed. This imposes a sequential, write-once discipline on disk writes, which allows the use of standard checkpointing techniques [16,19], increasing the reliability of our algorithms. When results of intermediate computations are no longer needed, space is reclaimed, e.g., through garbage collection. The maximum disk space active at any one time is used to measure the space complexity. All of our algorithms use only linear space.

2 Previous Approaches

2.1 PRAM Simulation

Chiang et al. [8] show how to simulate a CRCW PRAM algorithm using one processor and an external disk, thus giving a general method for constructing external graph algorithms from PRAM graph algorithms. Given a PRAM algorithm, the simulation maintains on disk arrays A, which contains the contents of main memory, and T, which contains the current state for each processor. Each step of the algorithm is simulated by constructing an array, D, which contains for each processor the memory address to be read. Sorting A and D by memory address and scanning them in tandem suffices to update T. A similar procedure is used to write updated values back to memory.

Each step thus requires a constant number of scans and sorts of arrays of size $|T|$ and a few scans of A. Typically, therefore, a PRAM algorithm using N processors and N space to solve a problem of size N in time t can be simulated in external memory by one processor using $O(t \ \mathit{sort}(N))$ I/Os. Better bounds are possible if the number of active processors and memory cells decreases linearly over the course of the PRAM algorithm. These techniques can, for example, simulate the PRAM maximal matching algorithm of Kelsen [14] in $O(\mathit{sort}(E) \log_2^4 V)$ I/Os and the PRAM connected components and MST algorithms of Chin, Lam, and Chen [9] in $O(\mathit{sort}(E) \log_2 \frac{V}{M})$ I/Os.

The PRAM simulation works in the FIO model, if each step writes new copies of T and A. To the best of our knowledge, however, no algorithm based on the simulation has been implemented. Such an implementation would require not only a practical PRAM algorithm but also either a meticulous direct implementation of the corresponding external memory simulation or a suitable low-level machine description of the PRAM algorithm together with a general simulation tool. In contrast, a major goal of our work is to provide simple, direct, and implementable algorithms.

2.2 Buffering Data Structures

Another recent approach is based on external variants of classical internal data structures. Arge [3] introduces *buffer trees*, which support sequences of insert, delete, and deletemin operations on N elements in $O(\frac{1}{B} \log_b \frac{N}{B})$ amortized I/Os each. Kumar and Schwabe [15] introduce a variant of the buffer tree, achieving the same heap bounds as Arge. These bounds are optimal, since the heaps can be used to sort externally. Kumar and Schwabe [15] also introduce external *tournament trees*. The tournament tree maintains the elements 1 to N, each with a key, subject to the operations delete, deletemin, and update: deletemin returns the element of minimum key, and update reduces the key of a given element. Each tournament tree operation takes $O(\frac{1}{B} \log_2 \frac{N}{B})$ amortized I/Os.

These data structures work by buffering operations in nodes and performing updates in batches. The maintenance procedures on the data structures are intuitively simple but involve many implementation details. The data structures also are not functional. The node-copying techniques of Driscoll et al. [10] could be used to make them functional, but at the cost of significant extra I/O overhead.

Finally, while such data structures excel in computational geometry applications, they are hard to apply to external graph algorithms. Consider computing an MST. The

classical greedy algorithm repeatedly performs deletemins on a heap to find the next vertex v to attach to the tree, T, and then performs updates for each neighbor w of v that becomes closer to T by way of v. In the external version of the algorithm, however, finding the neighbors of v is non-trivial, requiring a separate adjacency list. Furthermore, determining the current key of a given neighbor w (distance of w to T) is problematic, yet the key is required to decide whether to perform the update operation.

Intuitively, while the update operations on the external data structures can be buffered, yielding efficient amortized I/O complexity, standard applications of these data structures in graph algorithms require certain queries (key finding, e.g.) to be performed on-line. Current applications of these data structures thus require ancillary data structures to obviate this problem, increasing the I/O and implementation complexity.

3 Functional Graph Transformations

In this paper, we utilize a divide-and-conquer paradigm based on a few graph transformations that, for many problems, preserve certain critical properties. We implement each stage in our approach using simple and efficient techniques: sorting, selection, and bucketing. We illustrate our approach with connected components. Let $G = (V, E)$ be a graph. Let $f(G)$ V V be a forest of rooted stars (trees of height one) representing the connected components of G. That is, if $r_G(v)$ is the root of the star containing v in $f(G)$, then $r_G(v) = r_G(u)$ if and only if v and u are in the same connected component in G; $f(G)$ is presented as a delineated edge list.

Consider E^0 V V, and let $G^0 = G/E^0$ denote the result of contracting all vertex pairs in E^0. (This generalizes the usual notion of contraction, by allowing the contraction of vertices that are not adjacent in G.) For any x 2 V, let $s(x)$ be the supervertex in G^0 into which x is contracted. It is easy to prove that if each pair in E^0 contains two vertices in the same connected component, then $r_{G^0}(s(v)) = r_{G^0}(s(u))$ if and only if $r_G(v) = r_G(u)$; i.e., contraction preserves connected components. Thus, given procedures to contract a list of components of a graph and to re-expand the result, we derive the following simple algorithm to compute $f(G)$.

1. Let E_1 be any half of the edges of G; let $G_1 = (V, E_1)$.
2. Compute $f(G_1)$ recursively.
3. Let $G^0 = G/f(G_1)$.
4. Compute $f(G^0)$ recursively.
5. $f(G) = f(G^0)$ [$R(f(G^0), f(G_1))$, where $R(X, Y)$ relabels edge list Y by forest X: each vertex u occurring in edges in Y is replaced by its parent in X if it exists.

Our approach is functional if selection, relabeling, and contraction can be implemented without side effects on their arguments. We show how to do this below. In the following section, we use these tools to design functional external algorithms for computing connected components, MSTs, BMSTs, and maximal matchings.

3.1 Transformations

Selection. Let I be a list of items with totally ordered keys. `Select`(I, k) returns the kth biggest element from I; i.e., $\{x \in I \mid x < \texttt{Select}(I, k)\}| = k - 1$. We adapt the classical algorithm for `Select`(I, k) [2].

1. Partition I into j-element subsets, for some $j \quad M$.
2. Sort each subset in main memory. Let S be the set of medians of the subsets.
3. $m \quad$ Select$(S, dS/2e)$.
4. Let I_1, I_2, I_3 be those elements less than, equal to, and greater than m, respectively.
5. If $jI_1j \quad k$, then return Select(I_1, k).
6. Else if $jI_1j + jI_2j \quad k$, then return m.
7. Else return Select$(I_3, k - jI_1j - jI_2j)$.

Lemma 1. Select(I, k) *can be performed in* $O(scan(jIj))$ *I/Os in the FIO model.*

Relabeling. Given a forest F as an unordered sequence of tree edges $f(p(v), v), \ldots g$, and an edge set I, *relabeling* produces a new edge set $I^0 = ffr(u), r(v)gj fu, vg 2 Ig$, where $r(x) = p(x)$ if $(p(x), x) 2 F$, and $r(x) = x$ otherwise. That is, for each edge $fu, vg 2 I$, each of u and v is replaced by its respective parent, if it exists, in F. We implement relabeling as follows.

1. Sort F by source vertex, v.
2. Sort I by second component.
3. Process F and I in tandem.
   (a) Let $fs, hg 2 I$ be the current edge to be relabeled.
   (b) Scan F starting from the current edge until finding $(p(v), v)$ such that $v \quad h$.
   (c) If $v = h$, then add $fs, p(v)g$ to I^{00}; otherwise, add fs, hg to I^{00}.
4. Repeat Steps 2 and 3, relabeling first components of edges in I^{00} to construct I^0.

Relabeling is related to pointer jumping, a technique widely applied in parallel graph algorithms [11]. Given a forest $F = f(p(v), v), \ldots g$, *pointer jumping* produces a new forest $F^0 = f(p(p(v)), v) j (p(v), v) 2 Fg$; i.e., each v of depth two or greater in F points in F^0 to its grandparent in F. (Define $p(v) = v$ if v is a root in F.) Our implementation of relabeling is similar to Chiang's [7] implementation of pointer jumping.

Lemma 2. *Relabeling an edge list* I *by a forest* F *can be performed in* $O(sort(jIj) + sort(jFj))$ *I/Os in the FIO model.*

Contraction. Given an edge list I and a list $C = fC_1, C_2, \ldots g$ of delineated components, we can *contract* each component C_i in I into a supervertex by constructing and applying an appropriate relabeling to I.

1. For each $C_i = ffu_1, v_1g, \ldots g$:
   (a) $R_i \quad ;$.
   (b) Pick u_1 to be the canonical vertex.
   (c) For each $fx, yg 2 C_i$, add (u_1, x) and (u_1, y) to relabeling R_i.
2. Apply relabeling $\bigcup_i R_i$ to I, yielding the contracted edge list I^0.

Lemma 3. *Contracting an edge list* I *by a list of delineated components* $C = fC_1, C_2, \ldots g$ *can be performed in* $O(sort(jIj) + sort(\sum_i jC_ij))$ *I/Os in the FIO model.*

Deletion. Given edge lists I and D, it is straightforward to construct $I' = I \setminus D$: simply sort I and D lexicographically, and process them in tandem to construct I' from the edges in I but not D. If D is a vertex list, we can similarly construct $I'' = \{\{u, v\} \in I \mid u, v \notin D\}$, which we also denote by $I \setminus D$.

Lemma 4. *Deleting a vertex or edge set D from an edge set I can be performed in $O(sort(|I|) + sort(|D|))$ I/Os in the FIO model.*

4 Applying the Techniques

In this section, we devise efficient, functional external algorithms for computing connected components, MSTs, BMSTs, and maximal matchings of undirected graphs. Each of our algorithms needs $O(sort(E) \log_2 \frac{V}{M})$ I/Os. Our algorithms extend to multigraphs, by sorting the edges and removing duplicates in a preprocessing pass.

4.1 Deterministic Algorithms

Connected Components.

Theorem 1. *The delineated edge list of components of a graph can be computed in the FIO model in $O(sort(E) \log_2 \frac{V}{M})$ I/Os.*

Proof (Sketch). Let $T(E)$ be the number of I/Os required to compute $f(G)$, the forest of rooted stars corresponding to the connected components of G, presented as a delineated edge list. Recall from Section 3 the algorithm for computing $f(G)$. We can easily select half the edges in E in $scan(E)$ I/Os. Contraction takes $O(sort(E))$ I/Os, by Lemma 3. We use the forest $f(G')$ to relabel the forest $f(G_1)$. Combining the two forests and sorting the result by target vertex then yields the desired result. Thus, $T(E) \leq O(sort(E)) + 2T(E/2)$. We stop the recursion when a subproblem fits in internal memory, so $T(E) = O(sort(E) \log_2 \frac{V}{M})$.

Given $f(G)$, we can label each edge in E with its component in $O(sort(E))$ I/Os, by sorting E (by, say, first vertex) and $f(G)$ (by source vertex) and processing them in tandem to assign component labels to the edges. This creates a new, labeled edge list, E''. We then sort E'' by label, creating the desired output E'.

Minimum Spanning Trees. We use our approach to design a top-down variant of Boruvka's MST algorithm [4]. Let $G = (V, E)$ be a weighted, undirected graph, and let $f(G)$ be the delineated list of edges in a minimum spanning forest (MSF) of G.

1. Let $m = \texttt{Select}(E, |E|/2)$ be the edge of median weight.
2. Let $S(G) \subseteq E$ be the edges of weight less than m and half the edges of weight m.
3. Let G_2 be the contraction $G/f(S(G))$.
4. Then $f(G) = f(S(G)) \cup R_2(f(G_2))$, presented as a delineated edge list, where $R_2(\,)$ re-expands edges in G_2 that are incident on supervertices created by the contraction $G/f(S(G))$ to be incident on their original endpoints in $S(G)$.

Theorem 2. *A minimum spanning forest of a graph can be computed in the FIO model in* $O(sort(E)\log_2 \frac{V}{M})$ *I/Os.*

Proof (Sketch). Correctness follows from the analysis of the standard greedy approach. The I/O analysis uses the same recursion as in the proof of Theorem 1. Selection incurs $O(scan(E))$ I/Os, by Lemma 1. The input to $f(\ )$ is an annotated edge list: each edge has a weight and a label. Contraction incurs $O(sort(E))$ I/Os, by Lemma 3, and installs the corresponding original edge as the label of the contracted edge. The labels allow us to "re-expand" contracted edges by a reverse relabeling. To produce the final edge list, we apply a procedure similar to that used to delineate connected components.

Because our contraction procedure requires a list of delineated components, we cannot implement the classical, bottom-up Boruvka MST algorithm [4], in which each vertex selects the incident edge of minimum weight, the selected edges are contracted, and the process repeats until one supervertex remains. An efficient procedure to contract an arbitrary edge list could thus be used to construct a faster external MST algorithm.

Bottleneck Minimum Spanning Trees. Given a graph $G = (V, E)$, a *bottleneck minimum spanning tree* (*BMST*, or *forest*, *BMSF*) is a spanning tree (or forest) of G that minimizes the maximum weight of an edge. Camerini [5] shows how to compute a BMST of an undirected graph in $O(E)$ time, using a recursive procedure similar to that for MSTs.

1. Let $S(G)$ be the lower-weighted half of the edges of E.
2. If $S(G)$ spans G, then compute a BMST of $S(G)$.
3. Otherwise, contract $S(G)$, and compute a BMST of the remaining graph.

We design a functional external variant of Camerini's algorithm analogously to the MST algorithm above. In the BMST algorithm, $f(G)$ returns a BMSF of G and a bit indicating whether or not it is connected (a BMST). If $f(S(G))$ is a tree, then $f(G) = f(S(G))$; otherwise, we contract $f(S(G))$ and recurse on the upper-weighted half.

Theorem 3. *A bottleneck minimum spanning tree can be computed in the FIO model in* $O(sort(E)\log_2 \frac{V}{M})$ *I/Os.*

Whether BMSTs can be computed externally more efficiently than MSTs is an open problem. If we could determine whether or not a subset $E' \subseteq E$ spans a graph in $g(E')$ I/Os, then we can use that procedure to limit the recursion to one half of the edges of E, as in the classical BMST algorithm. This would reduce the I/O complexity of finding a BMST to $O(g(E) + sort(E))$ ($sort(E)$ to perform the contraction).

Maximal Matching. Let $f(G)$ be a maximal matching of a graph $G = (V, E)$, and let $V(f(G))$ be the vertices matched by $f(G)$. We find $f(G)$ functionally as follows.

1. Let $S(G)$ be any half of the edges of E.
2. Let $G_2 = E \cap V(f(S(G)))$.
3. Then $f(G) = f(S(G)) \cup f(G_2)$.

Theorem 4. *A maximal matching of a graph can be computed in the FIO model in* $O(sort(E)\log_2 \frac{V}{M})$ *I/Os.*

Proof (Sketch). Selecting half the edges takes $scan(E)$ I/Os. Deletion takes $O(sort(E))$ I/Os, by Lemma 4. Deleting all edges incident on matched vertices must remove all edges in E_1 from G by the assumption of maximality. Hence $jE(G_2)j \quad jEj/2$.

Finding a maximum matching externally (efficiently) remains an open problem.

4.2 Randomized Algorithms

Using the random vertex selection technique of Karger, Klein, and Tarjan [12], we can reduce by a constant fraction the number of vertices as well as edges at each step. This leads to randomized functional algorithms for connected components, MSTs, and BMSTs that incur $O(sort(E))$ I/Os with high probability. We can also implement a functional external version of Luby's randomized maximal independent set algorithm [17] that uses $O(sort(E))$ I/Os with high probability, yielding the same bounds for maximal matching.

Similar approaches were suggested by Chiang et al. [8] and Mehlhorn [18]. We leave details of these randomized algorithms to the full paper.

5 Semi-External Problems

We now consider *semi-external* graph problems, when $V \quad M$ but $E > M$. These cases often have practical applications, e.g., in graphs induced by monitoring long-term traffic patterns among relatively few nodes in a network. The ability to maintain information about the vertices in memory often simplifies the problems.

For example, the semi-external MST problem can be solved with $O(sort(E))$ I/Os: we simply scan the sorted edge list, using a disjoint set union (DSU) data structure [21] in memory to maintain the forest. We can even solve the problem in $scan(E)$ I/Os, using dynamic trees [20] to maintain the forest. For each edge, we delete the maximum weight edge on any cycle created. The total internal computation time becomes $O(E \log V)$.

Semi-external BMSTs are similarly simplified, because we can check internally if an edge subset spans a graph. Semi-external maximal matching is possible in one edge-list scan, simply by maintaining a matching internally.

If $V \quad M$, we can compute the forest of rooted stars corresponding to the connected components of a graph in one scan, using DSU to maintain a forest internally. We can label the edges by their components in another scan, and we can then sort the edge list to arrange edges contiguously by component. The sorting bound is pessimistic for computing connected components, however, if the number of components is small. Below we give an algorithm to group N records with K distinct keys, so that records with distinct keys appear contiguously, in $O(scan(N)\log_b K)$ I/Os. Therefore, if $V \quad M$ we can compute connected components on a graph $G = (V, E)$ in $O(scan(E)\log_b C(G))$ I/Os, where $C(G) \quad V$ is the number of components of G. In many applications, $C(G) \quad V$, so this approach significantly improves upon sorting.

5.1 Grouping

Let $I = (x_1, \ldots, x_N)$ be a list of N records. We use x_i to mean the key of record x_i as well as the record itself. The *grouping problem* is to permute I into a new list $I' = (x_{\pi_1}, \ldots, x_{\pi_N})$, such that $x_{\pi_i} = x_{\pi_j}$, $i < j$ $\Rightarrow$ $x_{\pi_i} = x_{\pi_{i+1}}$; i.e., all records with equal keys appear contiguously in I'. Grouping differs from sorting in that the order among elements of different keys is arbitrary.

We first assume that the keys of the elements of I are integers in the range $[1, G]$; we can scan I once to find G, if necessary. Our grouping algorithm, which we call *quickscan*, recursively re-partitions the elements in I as follows.

The first pass permutes I so that elements in the first G/b groups appear contiguously, elements in the second G/b groups appear contiguously, etc. The second pass refines the permutation so that elements in the first G/b^2 groups appear contiguously, elements in the second G/b^2 groups appear contiguously, etc. In general, the output of the kth pass is a permutation $I_k = (x_{\pi^k_1}, \ldots, x_{\pi^k_N})$, such that

$$\frac{x_{\pi^k_i}}{\lceil G/b^k \rceil} = \frac{x_{\pi^k_j}}{\lceil G/b^k \rceil}, \; i < j \Rightarrow \frac{x_{\pi^k_i}}{\lceil G/b^k \rceil} = \frac{x_{\pi^k_{i+1}}}{\lceil G/b^k \rceil}.$$

Let $\Delta_k = \lceil G/b^k \rceil$. Then keys in the range $[1 + j\Delta_k, 1 + (j+1)\Delta_k - 1]$ appear contiguously in I_k, for all k and $0 \le j < b^k$. $I_0 = I$ is the initial input, and $I' = I_k$ is the desired final output when $k = \lceil \log_b G \rceil$, bounding the number of passes.

Refining a Partition. We first describe a procedure that permutes a list L of records with keys in a given integral range $[K, K']$. Let $\delta = \frac{K' - K + 1}{b}$. The procedure produces a permutation L' such that records with keys in the range $[K + j\delta, K + (j+1)\delta - 1]$ occur contiguously, for $0 \le j < b$. Initially, each memory block is empty, and a pointer P points to the first disk block available for the output. As blocks of L are scanned, each record x is assigned to memory block $m = \frac{x - K}{\delta}$. Memory block m will thus be assigned keys in the range $[K + m\delta, K + (m+1)\delta - 1]$. When block m becomes full, it is output to disk block P, and P is updated to point to the next empty disk block.

Since we do not know where the boundaries will be between groups of contiguous records, we must construct a singly linked list of disk blocks to contain the output. We assume that we can reference a disk block with $O(1)$ memory cells; each disk block can thus store a pointer to its successor. Additionally, each memory block m will store pointers to the first and last disk blocks to which it has been written. An output to disk block P thus requires one disk read and two disk writes: to find and update the pointers in the disk block preceding P in L' and to write P itself.

After processing the last record from L, each of the b memory blocks is empty or partially full. Let M_1 be the subset of memory blocks, each of which was never filled and written to disk; let M_2 be the remaining memory blocks. We compact all the records in blocks in M_1 into full memory blocks, leaving at most one memory block, m_0, partially filled. We then write these compacted blocks (including m_0) to L'.

Let $\{m_1, \ldots, m_g\} = M_2$. We wish to write the remaining records in M_2 to disk so that at completion, there will be at most one partial block in L'. Let L_i be the last disk block written from m_i; among all the L_is we will maintain the invariant that there

is at most one partial block. At the beginning, L_0 can be the only partial block. When considering each m_i in turn, for $1 \quad i \quad \ell$, only L_{i-1} may be partial.

We combine in turn the records in L_{i-1} with those in m_i and possibly some from L_i to *percolate* the partial block from L_{i-1} to L_i. Let $|X|$ be the number of records stored in a block X. If $n_i = |L_{i-1}| + |m_i| \quad B$, we add $B - |L_{i-1}|$ records from m_i to L_{i-1}, and we store the remaining $|m_i| - B + |L_{i-1}|$ records from m_i in the new partial block, L_i^0, which we insert into L^0 after L_i; we then set $L_i \quad L_i^0$. Otherwise, $n_i < B$, and we add all the records in m_i as well as move $B - n_i$ records from L_i to L_{i-1}; this again causes L_i to become the partial block.

Quickscan. We now describe phase i of quickscan, given input I_{i-1}. $\Delta_{i-1} = \lfloor G/b^{i-1} \rfloor$, and keys in the range $[1 + j\Delta_{i-1}, 1 + (j+1)\Delta_{i-1} - 1]$ appear contiguously in I_{i-1}, for $0 \quad j < b^{i-1}$. While scanning I_{i-1}, therefore, we always know the range of keys in the current region, so we can iteratively apply the refinement procedure.

We maintain a value S, which denotes the lowest key in the current range. Initially, $S = 1$; the first record scanned will assign a proper value to S. We scan I_{i-1}, considering each record x in each block in turn. If $x \in [S, S + \Delta_{i-1} - 1]$, then x is the first record in a new group of keys, each of which is in the integral range $[S^0, S^0 + \Delta_{i-1} - 1]$, for $S^0 = 1 + \left\lfloor \frac{x-1}{\Delta_{i-1}} \right\rfloor \Delta_{i-1}$. Furthermore, all keys within this range appear contiguously in I_{i-1}. The record read previously to x was therefore the last record in the old range, and so we finish the refinement procedure underway and start a new one to refine the records in the new range, $[S^0, S^0 + \Delta_{i-1} - 1]$, starting with x.

We use percolation to combine partial blocks remaining from successive refinements. The space usage of the algorithm is thus optimal: quickscan can be implemented using $2 \quad scan(N)$ extra blocks, to store the input and output of each phase in an alternating fashion. A non-functional quickscan can be implemented in place, because the ith block output is not written to disk until after the ith block in the input is scanned.

In either case, grouping N items with keys in the integral range $[1, G]$ takes $O(scan(N) \log_b G)$ I/Os. Note that grouping solves the *proximate neighbors* problem [8]: given N records on $N/2$ distinct keys, such that each key is assigned to exactly two records, permute the records so that records with identical keys reside in the same disk block. Chiang et al. [8] show a lower bound of $\Omega(\min\{N, sort(N)\})$ I/Os to solve proximate neighbors in the single-disk I/O model. Our grouping result does not violate this bound, since in the proximate neighbors problem, $G = N/2$.

5.2 Sorting with Duplicates

Because of the compaction of partially filled blocks at the end of a refinement, quickscan cannot sort the output. By using a constant factor extra space, however, quickscan can produce sorted output. This is *sorting with duplicates*.

Consider the refinement procedure applied to a list L. When the last block of L has been processed, the b memory blocks are partitioned into sets M_1 and M_2. The procedure optimizes space by compacting and outputting the records in blocks in M_1 and then writing out blocks in M_2 in turn, percolating the (at most) one partially filled block on disk. Call the blocks in M_1 *short blocks* and the blocks in M_2 *long blocks*. The global compaction of the short blocks results in the output being grouped but not sorted.

The short blocks can be partitioned into subsets of M_1 that occur contiguously in memory, corresponding to contiguous ranges of keys in the input. We restrict compaction to contiguous subsets of short blocks, which we sort internally. Then we write these subsets into place in the output L^0. L^0 is thus partitioned into ranges that alternatively correspond to long blocks and contiguous subsets of short blocks. Since a long block and the following contiguous subset of short blocks together produce at most two partially filled blocks in L^0, the number of partially filled blocks in L^0 is bounded by twice the number of ranges produced by long blocks. Each range produced by long blocks contains at least one full block, so the number of blocks required to store L^0 is at most 3 $scan(|L|)$. The output L^0, however, is sorted, not just grouped.

We apply this modified refinement procedure iteratively, as above, to sort the input list I. Each sorted, contiguous region of short blocks produced in phase i, however, is skipped in succeeding phases: we need not refine it further. Linking regions of short blocks produced at the boundaries of the outputs of various phases is straightforward, so in general the blocks in any phase alternate between long and short. Therefore, skipping the regions of short blocks does not affect the I/O complexity.

Sorting N records with G distinct keys in the integral range $[1, G]$ thus takes $\Theta(scan(N) \log_b G)$ I/Os. The algorithm is stable, assuming a stable internal sort.

5.3 Grouping with Arbitrary Keys

If the G distinct keys in I span an arbitrary range, we can implement a randomized quickscan to group them. Let H be a family of universal hash functions [6] that map I to the integral range $[1, b]$. During each phase i in randomized quickscan, we pick an h_i uniformly at random from H and consider $h_i(x)$ to be the key of $x \in I$.

Each bucket of records output in phase i is thus refined in phase $i + 1$ by hashing its records into one of b new buckets, using function h_{i+1}. Let η be the number of distinct keys in some bucket. If $\eta \leq b$, we can group the records in one additional scan. Linking the records in the buckets output in the first phase T in which each bucket has no more than b distinct keys therefore produces the desired grouped output.

Let η' be the number of distinct keys hashed into some new bucket from a bucket with η distinct keys. The properties of universal hashing [6] show that $E[\eta'] = \eta/b$. Theorem 1.1 of Karp [13] thus shows that $\Pr[T \geq b\log_b G\rceil + c + 1] \leq G/b^{b\log_b G\rceil + c}$ for any positive integer c. Therefore, with high probability, $O(scan(N) \log_b G)$ I/Os suffice to group I. We use global compaction and percolation to optimize space usage.

6 Conclusion

Our functional approach produces external graph algorithms that compete with the I/O performance of the best previous algorithms but that are simpler to describe and implement. Our algorithms are conducive to standard checkpointing and programming language optimization tools. An interesting open question is to devise incremental and dynamic algorithms for external graph problems. The data-structural approach of Arge [3] and Kumar and Schwabe [15] holds promise for this area. Designing external graph algorithms that exploit parallel disks also remains open. Treating P disks as one with

a block size of PB extends standard algorithms only when $P = O((M/B)^{\alpha})$ for $0 \leq \alpha < 1$. In this case, the $\log_{M/B}$ terms degrade by only a constant factor.

Acknowledgements. We thank Ken Church, Kathleen Fisher, David Johnson, Haim Kaplan, David Karger, Kurt Mehlhorn, and Anne Rogers for useful discussions.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *C. ACM*, 31(8):1116–27, 1988.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
3. L. Arge. The buffer tree: A new technique for optimal I/O algorithms. In *Proc. 4th WADS*, volume 955 of *LNCS*, pages 334–45. Springer-Verlag, 1995.
4. O. Boruvka. O jistém problému miniméln´m. *Práce Mor. Pr´rodoved. Spol. v Brne*, 3:37–58, 1926.
5. P. M. Camerini. The min-max spanning tree problem and some extensions. *IPL*, 7:10–4, 1978.
6. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *JCSS*, 18:143–54, 1979.
7. Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results*. PhD thesis, Dept. of Comp. Sci., Brown Univ., 1995.
8. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM SODA*, pages 139–49, 1995.
9. F. Y. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *C. ACM*, 25(9):659–65, 1982.
10. J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *JCSS*, 38(1):86–124, February 1989.
11. J. Ja' Ja. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
12. D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–28, 1995.
13. R. M. Karp. Probabilistic recurrence relations. *J. ACM*, 41(6):1136–50, 1994.
14. P. Kelsen. An optimal parallel algorithm for maximal matching. *IPL*, 52(4):223–8, 1994.
15. V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE SPDP*, pages 169–76, 1996.
16. M. J. Litzkow and M. Livny. Making workstations a friendly environment for batch jobs. In *Proc. 3rd Wks. on Work. Oper. Sys.*, April 1992.
17. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comp.*, 15:1036–53, 1986.
18. K. Mehlhorn. Personal communication. `http://www.mpi-sb.mpg.de/ crauser-/courses.html`, 1998.
19. J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under UNIX. In *Proc. Usenix Winter 1995 Tech. Conf.*, pages 213–23, 1995.
20. D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26(3):362–91, 1983.
21. R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–81, 1984.
22. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comp. Sci.*, 73:231–48, 1990.

Minimal Triangulations for Graphs with "Few" Minimal Separators

Vincent Bouchitté and Ioan Todinca

LIP-Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
{Vincent.Bouchitte, Ioan.Todinca}@ens-lyon.fr

Abstract. We give a characterization of minimal triangulation of graphs using the notion of "maximal set of neighbor separators". We prove that if all the maximal sets of neighbor separators of some graphs can be computed in polynomial time, the treewidth of those graphs can be computed in polynomial time. This notion also unifies the already known algorithms computing the treewidth of several classes of graphs.

1 Introduction

The *treewidth* of graphs, introduced by Robertson and Seymour [12], has been intensively studied in the last years, mainly because many NP-hard problems become solvable in polynomial and even linear time when restricted to graphs with small treewidth. These algorithms use a tree-decomposition of small width of the graph. A tree-decomposition or a *triangulation* of a graph is a chordal supergraph (i.e. all the cycles of the supergraph of length more than three have a chord). Computing the treewidth of a graph corresponds to finding a triangulation with the smallest cliquesize. In particular, we can restrict ourselves to triangulations minimal by inclusion, that we call *minimal triangulations*.

Computing the treewidth of arbitrary graphs is NP-hard. Nevertheless, the treewidth can be computed in polynomial time for several well-known classes of graphs, for example the chordal bipartite graphs [9], the circle and circular-arc graphs [5] [14] and permutation graphs [2]. All these algorithms use the *minimal separators* of the graph and the fact that these classes of graphs have "few" minimal separators, in the sense that the number of the separators is polynomially bounded in the size of the graph. All of them compute minimal triangulations. Different algorithms and different proofs are given for each of these classes of graphs. This paper gives a unified version of the cited algorithms.

Characterizations of the minimal triangulations of a graph by the minimal separators have already been given in [2], [11]. Their approach is a global vision of all the minimal separators of a graph and therefore they do not yield an algorithmic construction of minimal triangulations with small cliquesize. A more local view of the minimal triangulations has been tried in [6], but it was unfortunately not correct [7]. We will give a new local characterization of minimal triangulations by mean of "maximal sets of neighbor separators", that we

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 344{355, 1998.
c Springer-Verlag Berlin Heidelberg 1998

will use to compute the treewidth for particular classes of graphs, including those previously mentioned. Notice that the same technique can also be used for computing the minimum fill-in of these classes.

We could also use this characterization for the first algorithm computing the treewidth of weakly triangulated graphs.

2 Chordal Graphs and Minimal Separators

Throughout this paper we consider connected, simple, finite, undirected graphs.

A graph H is *chordal* (or *triangulated*) if every cycle of length at least four has a chord. A *triangulation* of a graph $G = (V; E)$ is a chordal graph $H = (V; E^0)$ such that $E \quad E^0$. H is a *minimal triangulation* if for any intermediate set E^{00} with $E \quad E^{00} \quad E^0$, the graph $(V; E^{00})$ is not triangulated.

We will use the representation of the chordal graphs provided by *clique trees*. For an extensive survey of these notions, see [1], [3]. A *clique* is a complete subgraph of G. Consider now the set $K_G = f\Omega_1; : : : ; \Omega_p g$ of maximal cliques of G. Let T be a tree on K_G, i.e. every maximal clique $\Omega \; 2 \; K_G$ corresponds to exactly one node of T (we also say that the nodes of T are *labeled* by the cliques of K_G). We say that T is a *clique tree* of G if it satisfies the *clique-intersection property*: for every pair of distinct cliques $\Omega; \Omega^0 \; 2 \; K$, the set $\Omega \setminus \Omega^0$ is contained in every clique on the path connecting Ω and Ω^0 in the tree T. It is well known (see [1] for a proof) that a graph G is chordal if and only if it has a clique tree.

A subset $S \quad V$ is an $a; b$-*separator* for two nonadjacent vertices $a; b \; 2 \; V$ if the removal of S from the graph separates a and b in different connected components. S is a *minimal* $a; b$-*separator* if no proper subset of S separates a and b. We say that S is a *minimal separator* of G if there are two vertices a and b such that S is a minimal $a; b$ separator (notice that a minimal separator can be strictly included in another). We denote by $\quad G$ is the set of all minimal separators of G.

The following crucial property is proved in [1] :

Proposition 1. *Let H be a chordal graph and T be any clique tree of H. A set S is a minimal separator if and only if $S = \Omega \setminus \Omega^0$ for some maximal cliques Ω and Ω^0 of H adjacent in the clique tree T.*

Let G be a graph and S a minimal separator of G. We note $C_G(S)$ the set of connected components of $G - S$. A component $C \; 2 \; C_G(S)$ is *full* if every vertex of S is adjacent to some vertex of C. We denote by $C_G(S)$ the set of all full components of $G - S$. We remark that every minimal separator has at least two full components and it is a minimal $a; b$-separator for every couple of vertices a and b which are in different full components of S. If $C \; 2 \; C(S)$, we say that $(S; C) = S \; [\; C$ is a *block* of S. A block $(S; C)$ is called *full* if C is a full component of S.

De nition 1. *Two separators S and T* cross*, denoted by $S]T$, if there are some distinct components C and D of $G - T$ such that S intersects both of them. If S and T do not cross, they are called* parallel*, denoted by SkT.*

It is easy prove that these relations are symmetric (see [10]). Remark that for any couple of parallel separators S and T, T is contained in some block $(S; C)$ of S. The parallelism of the separators has a certain "pseudo-transitivity". Indeed, if $T \text{k} S$, $S \text{k} U$ and T and U are contained in different blocks of S, then then $T \text{k} U$. In particular, S separates any vertex of $T - S$ from any vertex of $U - S$. In this case we say that S *separates* T and U.

Using the fact that a separator cannot separate two adjacent vertices and the proposition 1, we deduce the following lemma.

Lemma 1. *Let G be a graph, S a minimal separator and Ω a clique of G. Then Ω is included in some block of S. In particular, the minimal separators of a chordal graph are pairwise parallel.*

De nition 2. *Let G be a graph. The* treewidth *of G is the minimum, over all triangulations H of G, of $!(H) - 1$, where $!(H)$ is the the maximum cliquesize of H.*

In other words, computing the treewidth of G means finding a triangulation with smallest cliquesize, so we can restrict our work to minimal triangulations.

Let $S \; 2 \;$ $_G$ be a minimal separator. We denote by G_S the graph obtained from G by completing S, i.e. by adding an edge between every pair of vertices $x; y \; 2 \; S$. If Γ $_G$ is a set of separators of G, G_Γ is the graph obtained by completing all the separators of Γ. The results of [2], concluded in [11], establish a strong relation between the minimal triangulations of a graph and its separator graph.

Theorem 1.

- *Let $\Gamma \; 2$ $_G$ be a maximal set of pairwise parallel separators of G. Then $H = G_\Gamma$ is a minimal triangulation of G and $_H = \Gamma$.*
- *Let H be a minimal triangulation of a graph G. Then $_H$ is a maximal set of pairwise parallel separators of G and $H = G$ $_H$.*

In other terms, every minimal triangulation of a graph G is obtained by considering a maximal set Γ of pairwise parallel separators of G and completing the separators of Γ. The minimal separators of the triangulation are exactly the elements of Γ.

It is important to know that the elements of Γ, who become the separators of H, have strictly the same behavior in H as in G. Indeed, the connected components of $H - S$ are exactly the same in $G - S$, for every $S \; 2 \; \Gamma$. Moreover, the full components are the same in the two graph: $C_H(S) = C_G(S)$.

3 Maximal Sets of Neighbor Separators

The previous theorem gives a characterization of the minimal triangulations of a graph by means of minimal separators, but it needs a global look over the set of minimal separators. Therefore, it gives no algorithmic information about how

we should construct a minimal triangulation in order to optimize its cliquesize. In this section we give a local characterization of minimal triangulations. We introduce the notion of "maximal set of neighbor separators" and we show how these sets are related to the maximal cliques of any triangulation of a graph. We also give a tool for recognizing the maximal sets of neighbor separators.

Definition 3. *Let H be a triangulated graph. We say that S $_H$ is a* maximal set of neighbor separators *if there is a maximal clique Ω of H such that $S = \{S \in {}_H|S \subset \Omega\}$. We also say that S* borders Ω *in H.*

Definition 4. *Let G be a graph and S $_G$ a set of pairwise parallel separators such that for any $S \in S$, there is a block $(S; C(S))$ containing all the separators of S.*

We define the piece between *the elements of S, denoted by $P(S)$, as*

$$\begin{cases} \bigcup, & \text{if there is some } U \in S \text{ that contains every } S \in S. \\ {}_{S \in S}(S; C(S)) & \text{otherwise.} \end{cases}$$

Notice that if we are in the second case, for any $S \in S$ the block of S containing all the separators of S is unique : if $T \in S$ is not included in S, there is a unique connected component of S containing $T - S$.

Lemma 2. *Let S $_H$ be a maximal set of neighbor separators. Then $P(S)$ exists.*

Proof. Let Ω be a clique bordered by S. For any $S \in S$, the clique Ω is contained in some block $(S; C)$ (lemma 1). So all the elements of S are contained in a same block of S. □

We are now able to establish the relation between a maximal set S of neighbor separators of a chordal graph H and a maximal clique of H bordered by S.

Theorem 2. *Let S be the maximal set of neighbor separators bordering a maximal clique Ω of a chordal graph H. If all the separators of S are included in some $S \in S$, then Ω is a block of S. Otherwise, $\Omega = P(S)$.*

Proof. Suppose that $\bigcup_{T \in S} T = S$ with $S \in S$. Ω is included in some block $(S; C)$ of S. If $x \in C$ is not in Ω, we take in a clique tree of H the shortest path $\Omega; \Omega^0; :::; \Omega^{(i)}$ of adjacent cliques such that $x \in \Omega^{(i)}$. Then $T = \Omega \setminus \Omega^0$ is a minimal separator of H included in Ω, so it must be an element of S. In particular, T is included in S. T separates x and any vertex of $\Omega - T$. Since $T \subset S$, clearly S separates x and any vertex of $\Omega - S$, contradicting the fact that x and Ω are in the same block of S. It remains that $(S; C) = \Omega$.

Suppose now that no separator $S \in S$ contains all the others. According to the lemma 2, $P(S)$ exists. On one hand $\Omega \subset P(S)$. Indeed, for any $S \in S$, Ω is in the block $(S; C(S))$ containing all the separators of S. On the other hand, consider $y \in P(S) - \Omega$. In a clique tree of H, we consider the shortest path $\Omega; \Omega^0; :::; \Omega^{(i)}$ of adjacent cliques such that $y \in \Omega^{(i)}$. Then $S = \Omega \setminus \Omega^0$

is a minimal separator that belongs to S and S separates y from every vertex of $\Omega - S$. Let T be a separator of S, not included in S. Then S separates y and a vertex of $T - S$, so $y \in (S; C_S(S))$, contradicting our choice. So we have $\Omega = P(S)$. □

Definition 5. *For an arbitrary graph G, S G is a* maximal set of neighbor separators *if there is a minimal triangulation H of G such that S is a maximal set of neighbor separators in H.*

We now establish the "reverse" of the theorem 2, which will allow us to "recognize" a set of neighbor separators. For this we need some easy lemmas, that are given without proofs.

Lemma 3. *Let H be a chordal graph, Ω a maximal clique and S a minimal separator of H intersecting Ω. Then $S \setminus \Omega$ is included in some minimal separator T with T Ω.*

Lemma 4. *Let H be a minimal triangulation of a graph G and T be a minimal separator of G such that $H[T]$ is a clique. Then T is also a minimal separator of H.*

Corollary 1. *In particular, lemma 4 asserts that if T is a minimal separator of G contained in S with $S \in$ H, then $T \in$ H.*

Lemma 5. *Let $(S; C)$ be a block in G such that $G_S[S \cup C]$ is a clique. Then every minimal separator T $(S; C)$ is included in S.*

Theorem 3. *Let G be a graph and S G a set of separators having a maximum element S (S contains every $T \in S$). Then S is a maximal set of neighbor separators of G if and only if:*

1. *there is a block $(S; C)$ such that $G_S[S \cup C]$ is a clique.*
2. *$\forall T \in$ G, if T S, then $T \in S$.*

Proof. "$\Rightarrow$" Let $H = G_\Gamma$ be a minimal triangulation such that S borders a clique Ω. According to theorem 2, there is a block $(S; C)$ such that $H[S \cup C] = \Omega$. We prove that $S \cup C$ is a clique in G_S. Indeed, every edge of $H[S \cup C]$ is either an edge of G, or its endpoints x and y are on a same separator $T \in \Gamma$. In the last case, by lemma 3 we have a separator T' of H such that T' Ω and T' contains x and y. Necessarily $T' \in S$ (cf. the definition of maximal set of neighbor separators), so T' S. It remains that x and y are adjacent in $G_S[S \cup C]$. We conclude that $G_S[S \cup C]$ is a clique.

The second statement is a direct consequence of corollary 1.

"$\Leftarrow$" Notice that the separators of S are pairwise parallel. Let $H = G_\Gamma$ be a minimal triangulation extending S, i.e. with S Γ. We prove that $S \cup C$ is a

maximal clique of H, bordered by S. The clique $S \cup C$ of H is maximal. Indeed, if Ω is a clique of H including $S \cup C$, Ω must be in some block of S (lemma 1) and the only choice is $\Omega = (S; C)$. We denote by S^0 the maximal set of neighbor separators bordering Ω. We have $S \subseteq S^0$. Moreover, every element of S^0 is a separator of H, and also a separator of G. By the lemma 5 such a separator is included in S, and by the second statement it is an element of S. It remains that $S = S^0$. □

Theorem 4. *Let G be a graph and $S \subseteq \Delta_G$ a set of separators not having a greatest element. Then S is a maximal set of neighbor separators of G if and only if:*

1. *$P(S)$ is non empty and $G_S[P(S)]$ is a clique.*
2. *$\forall S' \in \Delta_G$, if $S \cap P(S) \neq \emptyset$ then $S' \in S$.*

Proof. The proof is almost the same as the one of the previous theorem.

"$\Rightarrow$" We consider a minimal triangulation H of G such that S is a maximal set of neighbor separators in H, bordering a clique Ω. According to theorem 2, $\Omega = P(S)$. We show now that $H[P(S)]$ is exactly $G_S[P(S)]$. For every x and $y \in \Omega$, either x and y are adjacent in G or they belong to a same separator of H. The later implies that x and y belong to a same separator T included in Ω by lemma 3, and by definition of a maximal set of neighbor separators we get $T \in S$. In both cases x and y are adjacent in $G_S[P(S)]$. This proves that $G_S[P(S)]$ is a clique.

Let T be a separator of G included in $P(S)$. $H[T]$ is a clique, so $T \in \Gamma$ by lemma 4 and necessarily $T \in S$, which proves the second statement.

"$\Leftarrow$" As in the previous theorem, we prove that S borders the clique $G_S[P(S)]$ in any triangulation extending S.

The first statement implies that the separators of S are pairwise parallel. We consider a maximal set of pairwise parallel separators $\Gamma \supseteq S$ and we take $H = G_\Gamma$. We consider a maximal clique Ω of H containing $P(S)$. We want to prove that $\Omega = P(S)$. Let x be a vertex of Ω. For any $S \in S$, x must be in the block $(S; C_\Omega(S))$ containing Ω. So $x \in P(S)$, which proves that $\Omega = P(S)$. Let now S^0 be the maximal set of neighbor separators bordering Ω. Clearly $S \subseteq S^0$ and by the second statement S^0 is a subset of S. It remains that $S = S^0$ is a maximal set of neighbor separators of G. □

So, given a set of separators S, one can easily verify if it is a maximal set of neighbor separators.

For a maximal set of neighbor separators S, we can define a clique Ω_S bordered by S in G_S as a clique $(S; C)$ of G_S if S is maximum in S, $P(S)$ otherwise (respectively the cliques that appear in the two previous theorems). Remark that if we are under the condition of the theorem 4 the clique Ω_S is unique, but that might not be the case if we are under the condition of theorem 3. By the proofs of the two previous theorems we deduce that:

Corollary 2. *Let S be a maximal set of neighbor separators of G and $\Gamma \quad S$ a set of pairwise parallel separators. Then a clique bordered by S in G_S is a maximal clique in G_Γ.*

From now on, when speaking about a clique bordered by a maximal set of neighbor separators S we will mean a clique of G_S.

Let S be a minimal separator and let $(S; C)$ be a full block of S. A maximal set of neighbor separators S is said to be *associated* to the block $(S; C)$ if $S \ 2 \ S$ and some clique Ω_S bordered by S is contained in $(S; C)$.

4 Computing the Treewidth

In this section we show how to compute in polynomial time the treewidth of a graph if we have all its maximal sets of neighbor separators.

Let $F = fS_1; S_2; : : : ; S_p g$ be a family of maximal sets of neighbor separators and $\Gamma_F = fSj9S \ 2 \ F$ s.t. $S \ 2 \ Sg$. We say that the family F is a *complete family of separators* if for any separator $S \ 2 \ \Gamma_F$ and for any full block $(S; C)$ of S, there is a maximal set of neighbor separators $S \ 2 \ F$ associated to the block $(S; C)$.

For example, if H is a triangulation of a graph G, the family $F = fSjS \quad _H$ is a maximal set of neighbor separators of Hg is a complete family of separators of G. Remark that if H has treewidth at most k, for any maximal set of neighbor separators S of H, all the cliques bordered by S have at most $k + 1$ vertices.

We will first compute the biggest complete family of separators of G, denoted by $F = fS_1; S_2; : : : ; S_p g$ such that all the maximal cliques bordered by any $S \ 2 \ F$ have at most $k+1$ elements. This family is empty if and only if G has treewidth greater than k or G is a clique.

Elimination algorithm

1. Compute all the maximal sets S of neighbor separators such that any clique bordered by S has at most $k + 1$ vertices. Let F be the family of these separators and Γ_F the union of the sets S.
2. Extract from F the maximum complete family of separators :
   `do`
   { choose if possible an $S \ 2 \ \Gamma_F$ such that for some full bloc $(S; C)$, there is no maximal set S of neighbor separators associated to $(S; C)$ with $S \ 2 \ F$.
   { $F = F_S - fSjS \ 2 \ Sg$;
   $\Gamma_F = \quad _{S2F} S$.
   `while` such a separator S exists.
3. `return` F.

Clearly, F is a complete family of separators.

Lemma 6. *Let G be a graph and H be any minimal triangulation of treewidth at most k. Then, at the end of the previous algorithm, any maximal set $S \quad _H$ of neighbor separators of H is in F.*

Proof. Let $F^0 = \{S_1^0, S_2^0, \ldots, S_q^0\}$ be the maximum sets of neighbor separators of H. For any $S \in F^0$, any clique Ω_S bordered by S has at most $k+1$ vertices, so S will be in F after the first step of the algorithm.

Moreover, let $S \in \Delta_H$ be a minimal separator of H and G. For any full block (S, C) of S, consider a maximal clique Ω of H included in (S, C) such that $S \subset \Omega$. Then the maximal set of neighbor separators S^0 that borders Ω is an element of F^0, so an element of F. It remains that no $S^0 \in F^0$ will be eliminated from F during the second step. □

The next step is to obtain from any non-empty complete family of separators F a minimal triangulation H of G such that the maximal sets of neighbor separators of H are elements of F.

We will construct a tree $T = (I, E_T)$ and we label each of its nodes by a subset of $V(G)$. The nodes will be of two types :

- "treated", in which case the label will be some maximal clique Ω bordered by some set of separators $S \in F$.
- "untreated", in which case the node is a leaf of the tree and the label of the node is a full block (S, C).

T will be a clique tree for a minimal triangulation of G.

Extraction algorithm

1. `Initialization step`
   - Chose a maximal set of neighbor separators $S \in F$. Chose a clique Ω bordered by S.
   - Create the root of T, labeled with Ω. Mark the root as "treated".
   - for all full blocks (S, C) such that $S \in S$ and $\Omega \not\supset (S, C)$, create a leaf as son of the root and label it with (S, C). Mark the leaf as "untreated".
   - put $F^0 = \{S\}$.
2. `Expansion step`
   - Choose an untreated leaf N, labeled (S, C). Choose a maximal set of neighbor separators $S \in F$ associated to (S, C). Let Ω be the clique bordered by S included in (S, C).
   - Label the node with Ω and mark it as treated.
   - for all the full blocks (S^0, C^0) such that $S^0 \in S$, $S^0 \not\subset S$ and $\Omega \not\supset (S^0, C^0)$, create a leaf node as son of N, mark it as "untreated" and label it by (S^0, C^0).
   - put $F^0 = F^0 \cup \{S\}$.
3. `repeat` the expansion step `until` all the nodes of the tree become "treated".
4. `return` $H = G_{\Gamma_{F^0}}$.

Notice first that when we expand a node labeled (S, C), the labels (S_i, C_i) of its sons will be strictly included in (S, C), so the algorithm terminates.

Lemma 7. *At any moment of the execution, all the vertices of G are in some label of T.*

Proof. Remark that if S is a minimal separator of a graph G and $C \in \mathcal{C}(S)$ is a non-full connected component of $G - S$, then the set $S' = \{x \in S | x$ is adjacent to some vertex of $C\}$ is a minimal separator of G and C is full component of S'. Indeed, consider a vertex $a \in C$ and a vertex b in a full component D of S. Clearly S' separates a and b. Moreover, for any $x \in S'$, there is a path from a to x included in $C \cup \{x\}$ and a path from x to b in $D \cup \{x\}$. This proves that $S' - \{x\}$ is not an a,b separator, so we conclude that S' is a minimal a,b separator.

Consider now a maximal set of neighbor separators S of G. If (S, C) is a non-full block of some $S \in \mathcal{S}$, we have proved that there is a minimal separator $S' \subset S$ such that (S', C) is a full block of S'. But we have $S' \in \mathcal{S}$ by theorems 3 and 4. Consequently, if $\mathcal{S}$ is a maximal set of separators of G, any vertex x is in some full block (S, C) with $S \in \mathcal{S}$. We deduce that at any moment of the execution of the extraction algorithm, any vertex of G is in some label of the tree T. □

Lemma 8. *At any time of the algorithm, for any untreated leaf labeled* (S, C), C *does not intersect any other label of the tree* T.

Proof. Let us prove the property after the initialization step. Let (S_1, C_1) be the label of a leaf. Let Ω be the label of the root. Clearly $C_1 \setminus \Omega = \emptyset$ by construction. Now let (S_2, C_2) be the label of another leaf and suppose that C_1 intersects (S_2, C_2). Since $S_2 \subset \Omega$, we must have $C_1 \setminus C_2 \neq \emptyset$. Notice that C_1 and C_2 are connected components of $G - \Omega$. Consequently, $C_1 = C_2$ contradicting the fact that the labels (S_1, C_1) and (S_2, C_2) are different.

Consider now an expansion step. Let N be an untreated node labeled (S, C); C does not intersect any other label of T. Let (S_1, C_1) be the label of a new leaf, obtained by the expansion of N. Like for the initialization step, C_1 does not intersect the new label Ω of N or the label (S_2, C_2) of a new created leaf. Moreover, $C_1 \setminus S = \emptyset$ and we have $C_1 \subset C$. Since C does not intersect any other label of the tree, C_1 cannot intersect any other label. □

At any moment of the execution, let T' the subtree of T formed by the treated nodes. We take $\Gamma = \{S \in \mathcal{S} | S \subset_G jS$ is in the label of some node of $T'\}$, $V' = \{x \in V | x$ is in the label of some node of $T'\}$ and $H = G_\Gamma[V']$.

Lemma 9. *At any moment of the execution,* Γ *is a set of pairwise parallel separators and* T' *is a clique tree of* $H = G_\Gamma[V']$.

Proof. The property is obvious after the initialization step.

Suppose that the property is true at a certain moment of the processing and we expand a node N. Let (S, C) be the label of N. Let $S \in \mathcal{F}$ be the maximal set of neighbor separators associated to (S, C) by the algorithm and Ω the clique bordered by S in (S, C). We denote by Γ_e, V'_e, T'_e, H_e the new parameters, after the expansion step.

First, we prove that the separators of Γ_e are pairwise parallel. Clearly we have $\Gamma_e = \Gamma \cup S$. It is sufficient to show that if $T \in \Gamma$ and $U \in S$ we have $T \# U$.

Because we have $T \quad V^0$ and $U \quad (S; C)$ and $C \setminus V^0 = ;$ (lemma 8), S separates T and U so $T k U$.

Now, we prove that T_e^0 is a clique tree of $H_e = G_{\Gamma_e}[V_e^0]$. Notice that $V_e^0 = V^0 \cup \Omega$. By corollary 2, Ω is a maximal clique of H_e. Moreover, H_e has exactly one more maximal clique than H, which is Ω, since the edges in H_e are either in H or in $H_e[\Omega]$. So the labels of T_e^0 are exactly the maximal cliques of H_e. It remains to show that for any label $\Omega^0 \neq \Omega$ of T_e^0, the set $\Omega \setminus \Omega^0$ is contained in every clique on the path connecting Ω and Ω^0 in T_e^0. Let Ω_0 be the neighbor of Ω in T_e^0. All we have to prove is that $\Omega \setminus \Omega^0 \quad \Omega_0$. By lemma 8, $\Omega \setminus V^0 = S$. But clearly $\Omega^0 \quad V^0$ and $S \quad \Omega_0$, so $\Omega \setminus \Omega^0 = S \setminus \Omega^0 \quad \Omega_0$. □

We deduce directly from lemmas 7 and 9:

Theorem 5. *Let G be a graph, F be a complete family of minimal separators of G. If F is not empty, we can construct a minimal triangulation H of G such that any maximal set of neighbor separators of H is an element of F.*

Corollary 3. *Let $\mathcal{G}$ be a class of graphs having a polynomial time algorithm for computing the family of all maximal sets of neighbor separators for every $G \in \mathcal{G}$. Then for any graph in $\mathcal{G}$ the treewidth is polynomially tractable.*

Proof. To decide if the treewidth of any $G \in \mathcal{G}$ is at most a fixed k, it is sufficient to :

1. apply the elimination algorithm to obtain a complete set of separators F.
2. `if` F is empty `then` the treewidth of G is greater then k, unless G is a clique with at most $k+1$ vertices.
3. `else` apply the extraction algorithm on F to get a triangulation H of G with treewidth at most k.

By lemma 6, if the treewidth of G is at most k the elimination algorithm will produce a non-empty family of separators F. Conversely, if the family F returned by the elimination algorithm is non-empty, by theorem 5 the extraction algorithm will produce a triangulation of treewidth at most k.

Concerning the complexity of this algorithm, remark that if the calculus of all the maximal sets of neighbor separators is done in polynomial time, both the elimination algorithm and the extraction algorithm work in polynomial time. □

5 Application to Some Classes of Graphs

Several classes of graphs have "few" minimal separators, in the sense that the number of minimal separators of such a graph is polynomially bounded in the size of the graph. Moreover, an algorithm given in [8] computes all the minimal separators of these graphs.

For these classes of graphs we also have algorithms computing the treewidth in polynomial time, using the minimal separators (cf. [9], [2], [5], [14], [10]). Different proofs have been given for each of these algorithms. We have remarked

that, for computing the treewith of a graph, all these algorithms compute all the maximal sets of neighbor separators of the graph. Therefore, our approach unifies the cited algorithms.

AT-free graphs are graphs with no asteroidal triple where three vertices constitute an asteroidal triple if between every two of them there exists a path avoiding the neighborhood of the third. One can prove (see for example [10]) that a graph G is AT-free if and only if for any three pairwise separators $S; T; U$ 2 $_G$, one of them separates the two others. It follows that if S is a maximal set of neighbor separators of an AT-free graph, S ordered by inclusion has at most two maximal elements. Therefore, it is easy to compute the maximal sets of neighbor separators for any AT-free graph with "few" separators. The class of AT-free graphs includes the *cocomparability graphs* and in particular the *permutation graphs* (which have $O(n^2)$ minimal separators for graphs with n vertices [2]) or the *d-trapezoid graphs*, which also have a polynomial number of separators for any fixed d.

Circle and circular arc graphs are obtained from an intersection model. A graph is a *circle graph* if we can associate every vertex of the graph to a chord of a circle such that two vertices are adjacent in the graph if and only if the corresponding chords cross. The circle and its chords are the *circle model* of the graph. In the same manner, a graph is a *circular arc graph* if the vertices can be associated to some arcs of a circle and two vertices cross if and only if the corresponding arcs overlap. For these "geometrical" classes of graphs, the minimal separators can be modeled by *scan-lines*. In the circle model of the circle graphs, we can add a *scan-point* between every two consecutive endpoints of the chords. For the circular arc graphs, the scan-points will be the middle of every circular arc. The scan-lines are the straight line segments between two scan-points. Using the results of [5] and [14], we deduce that the maximal sets of neighbor separators of these graphs correspond to some triangles of scanlines. Therefore, all the maximal sets of neighbor separators of circle and circular arc graphs can be computed in polynomial time.

Chordal bipartite graphs are bipartite graphs for which every cycle of length at least 6 has a chord. It was shown in [9] that for any bipartite chordal graph G, any maximal set of neighbor separators is characterized by two maximal bipartite subgraphs. Since any chordal bipartite graph G with e edges has $O(e)$ complete bipartite subgraphs, one can compute in polynomial time all the maximal sets of neighbor separators of G.

Weakly triangulated graphs are graphs G such that neither G, nor its complement $\overline{G}$ have induced cycle of more than four vertices. They were introduced in [4] and they contain the chordal bipartite graphs, the chordal graphs and the distance hereditary graphs. No algorithm computing the treewidth of these graphs was known up to now, although a polynomial bound for the number of the minimal separators was given in [6]. Using the recognition algorithm of weakly triangulated graphs given in [13], we could prove that a weakly triangulated graph G has at most $\overline{e}$ minimal separators, where $\overline{e}$ is the number of edges of $\overline{G}$. Moreover, we proved that G has at most $2\overline{e}$ maximal sets of neighbor sepa-

rators and they are easily tractable in polynomial time. We obtain a polynomial algorithm computing the treewidth of weakly triangulated graphs. The proofs of these facts will be given in the full version of the paper.

References

1. J. R. S. Blair and B. Peyton. An introduction to chordal graphs and clique trees. In A. George, J. R. Gilbert, and J. H. U. Liu, editors, *Graph Theory and Sparse Matrix Computations*, pages 1–29. Springer, 1993.
2. H.L. Bodlaender, T. Kloks, and D. Kratsch. Treewidth and pathwidth of permutation graphs. In *Proceedings of the 20th International Colloquium on Automata, Languages and Programming (ICALP'93)*, volume 700 of *Lecture Notes in Computer Science*, pages 114–125. Springer-Verlag, 1993.
3. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.
4. R. Hayward. Weakly triangulated graphs. *J. Combin. Theory ser. B*, 39:200–208, 1985.
5. T. Kloks. Treewidth of circle graphs. In *Proceedings 4th Annual International Symposium on Algorithms and Computation (ISAAC'93)*, volume 762 of *Lecture Notes in Computer Science*, pages 108–117. Springer-Verlag, 1993.
6. T. Kloks, H.L. Bodlaender, H. Mller, and D. Kratsch. Computing treewidth and minimum fill-in: All you need are the minimal separators. In *Proceedings First Annual European Symposium on Algorithms (ESA'93)*, volume 726 of *Lecture Notes in Computer Science*, pages 260–271. Springer-Verlag, 1993.
7. T. Kloks, H.L. Bodlaender, H. Mller, and D. Kratsch. Erratum to the ESA'93 proceedings. In *Proceedings Second Annual European Symposium on Algorithms (ESA'94)*, volume 855 of *Lecture Notes in Computer Science*, page 508. Springer-Verlag, 1994.
8. T. Kloks and D. Kratsch. Finding all minimal separators of a graph. In *Proceedings 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS'94)*, volume 775 of *Lecture Notes in Computer Science*, pages 759–768. Springer-Verlag, 1994.
9. T. Kloks and D. Kratsch. Treewidth of chordal bipartite graphs. In *Proceedings 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS'94)*, volume 775 of *Lecture Notes in Computer Science*, pages 759–768. Springer-Verlag, 1994.
10. A. Parra. *Structural and Algorithmic Aspects of Chordal Graph Embeddings.* PhD thesis, Technische Universität Berlin, 1996.
11. A. Parra and P. Scheffler. How to use the minimal separators of a graph for its chordal triangulation. In *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming (ICALP'95)*, volume 994 of *Lecture Notes in Computer Science*, pages 123–134. Springer-Verlag, 1995.
12. N. Robertson and P. Seymour. Graphs minors. II. Algorithmic aspects of treewidth. *J. of Algorithms*, 7:309–322, 1986.
13. J. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Discrete Applied Mathematics*, 59:181–191, 1995.
14. R. Sundaram, K. Sher Singh, and C. Pandu Rangan. Treewidth of circular-arc graphs. *SIAM J. Discrete Math.*, 7:647–655, 1994.

Finding an Optimal Path without Growing the Tree*

Danny Z. Chen, Ovidiu Daescu, Xiaobo (Sharon) Hu, and Jinhui Xu

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
{chen,odaescu,shu,jxu}@cse.nd.edu

Abstract. In this paper, we study a class of optimal path problems with the following phenomenon: The *space* complexity of the algorithms for reporting the *lengths* of single-source optimal paths for these problems is asymptotically smaller than the space complexity of the "standard" tree-growing algorithms for finding actual optimal paths. We present a general and efficient algorithmic paradigm for finding an actual optimal path for such problems without having to grow a single-source optimal path tree. Our paradigm is based on the "marriage-before-conquer" strategy, the prune-and-search technique, and a data structure called *clipped trees*. The paradigm enables us to compute an actual path for a number of optimal path problems and dynamic programming problems in computational geometry, graph theory, and combinatorial optimization. Our algorithmic solutions improve the space bounds (in certain cases, the time bounds as well) of the previously best known algorithms, and settle some open problems. Our techniques are likely to be applicable to other problems.

1 Introduction

For combinatorial problems on computing an optimal path as well as its length, the "standard" approach for finding an actual optimal path is by building (or "growing") a single-source optimal path tree. This tree-growing approach is effective for finding actual single-source optimal paths, especially as the *time* complexity is concerned. In fact, it is well-known that no general algorithms are known that compute an optimal path between *one pair* of locations with a faster *time* complexity than that for computing single-source optimal paths. In this paper, we study a class of optimal path problems with the following interesting yet less-exploited phenomenon: The *space* complexity of the algorithms for reporting the *lengths* of single-source optimal paths for these problems is asymptotically smaller than the space complexity of the "standard" tree-growing algorithms for finding actual optimal paths. Our goal is to show that for such problems, it is possible to find an actual optimal path without having to grow a single-source

* The work of the first, second, and fourth authors was supported in part by the National Science Foundation under Grant CCR-9623585. The work of the third author was supported in part by the National Science Foundation under Grant MIP-9701416 and by HP Labs, Bristol, England under an external research program grant.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 356–367, 1998.
© Springer-Verlag Berlin Heidelberg 1998

optimal path tree, thus achieving asymptotically better space bounds for finding one actual optimal path than those for single-source optimal paths.

It should be mentioned that the phenomenon that the space bound for finding an actual optimal path can be smaller than that for single-source optimal paths has been observed and exploited in some scattered situations. For example, Edelsbrunner and Guibas [9] showed that for computing a longest monotone path or a longest monotone concave path on the arrangement of size $O(n^2)$ formed by n lines on the plane, it is possible to report the *length* of such a path in $O(n^2)$ time and $O(n)$ space. To output an actual longest monotone path, they used $O(n^2 \log n)$ time and $O(n \log n)$ space, and to output an actual longest monotone concave path, they used $O(n^2 \log n)$ time and $O(n \log n)$ space (or alternatively, $O(n^3)$ time and $O(n)$ space). It was posed as open problems in [9] whether these extra time and space bounds for reporting an actual longest monotone path or longest monotone concave path could be partially or completely avoided. Another example is the problem of computing a longest common subsequence of two strings of size n [6,14,19] (this problem can be reduced to an optimal path problem). Hirschberg [14] used dynamic programming to find an actual longest common subsequence and its length in $O(n^2)$ time and $O(n)$ space without growing a single-source tree. The actual optimal path algorithms in [9] use a recursive back-up method, and the one in [14] is based on a special divide-and-conquer strategy called "marriage-before-conquer".

We study in a systematic manner the phenomenon that the space bound for finding an actual optimal path can be smaller than that for single-source optimal paths. We develop a general algorithmic paradigm for reporting an actual optimal path without using the tree-growing approach, and characterize a class of optimal path and dynamic programming problems to which our paradigm is applicable. This paradigm not only considerably generalizes the marriage-before-conquer strategy used in [14], but also brings forward additional interesting techniques such as prune-and-search and a new data structure called *clipped trees*. Furthermore, the paradigm makes it possible to exploit useful structures of some of the problems we consider. Our techniques enable us to compute efficiently an actual optimal solution for a number of optimal path and dynamic programming problems in computational geometry, graph theory, and combinatorial optimization, improving the space bounds (in certain cases, the time bounds as well) of the previously best known algorithms. Below is a summary of our main results on computing an actual optimal solution.

Computing a shortest path in the arrangement of n lines on the plane. As mentioned in [4,12], it is easy to reduce this problem to a shortest path problem on a planar graph of size $O(n^2)$ that represents the arrangement, and then solve it in $O(n^2)$ time and space by using the optimal shortest path algorithm for planar graphs [15]. We present an $O(n^2)$ time, $O(n)$ space algorithm.

Computing a longest monotone concave path in the arrangement of n lines on the plane. An $O(n^2 \log n)$ time, $O(n \log n)$ space algorithm and an $O(n^3)$ time, $O(n)$ space algorithm were given by Edelsbrunner and Guibas [9]. We present

an $O(n^2)$ time, $O(n)$ space algorithm. Our solution is an improvement on those of [9], and settles the corresponding open problem in [9].

Computing a longest monotone path in the arrangement of n lines on the plane. An $O(n^2 \log n)$ time, $O(n \log n)$ space algorithm and an $O(\frac{n^2}{\quad})$ time, $O(\frac{n^{1+}}{\quad})$ space algorithm were given by Edelsbrunner and Guibas [9]. We present an $O(\frac{n^2 \log n}{\log(h+1)})$ time, $O(nh)$ space algorithm, where h is any integer such that $1 \quad h \quad n$ for any constant with $0 < < 1$. Note that for $h = O(1)$, our algorithm uses $O(n^2 \log n)$ time and $O(n)$ space, and for $h = n$, our algorithm uses $O(\frac{n^2}{\quad})$ time and $O(n^{1+})$ space (unlike [9], our space bound does not depend on the $\frac{1}{\quad}$ factor). Our solution is an improvement on those of [9], and provides an answer to the corresponding open problem in [9].

Computing a longest monotone path in the arrangement of n planes in the 3-D space. An $O(n^3)$ time, $O(n^2)$ space algorithm was given by Anagnostou, Guibas, and Polimenis [1] for computing the *length* of such a path. If the techniques in [9] are used, then an actual path would be computed in $O(n^3 \log n)$ time and $O(n^2 \log n)$ space. We present an $O(\frac{n^3 \log n}{\log(h+1)})$ time, $O(n^2 h)$ space algorithm, where h is any integer such that $1 \quad h \quad n$ for any positive constant < 1.

Computing a minimum-weight, k-link path in a graph. A standard tree-growing approach uses $O(k(n+m))$ time and $O(kn)$ working space to compute a minimum-weight, k-link path in an edge-weighted graph of n vertices and m edges. We present an $O(\frac{k(n+m)\log k}{\log(h+1)})$ time, $O(nh)$ working space algorithm, where h is any integer such that $1 \quad h \quad k$ for any constant with $0 < < 1$. Note that for $h = O(1)$, our algorithm uses $O(k(n+m)\log k)$ time and $O(n)$ working space, and for $h = k$, our algorithm uses $O(\frac{1}{\quad}k(n+m))$ time and $O(nk)$ working space (the constant of the working space bound does not depend on $\frac{1}{\quad}$). Furthermore, if G is a directed acyclic graph, then our algorithm uses $O(k(n+m))$ time and $O(n)$ working space.

0–1 knapsack with integer item sizes. The 0–1 knapsack problem is NP-complete and has often been solved by dynamic programming [17] or by reducing the problem to computing an optimal path in a directed acyclic graph of $O(nB)$ vertices and edges. If a standard tree-growing approach is used for computing an actual solution, then it would use $O(nB)$ time and space [17] (it was also shown in [17] how to use a bit representation to reduce the space bound to $O(\frac{nB}{\log(n+B)})$). We present an $O(nB)$ time, $O(n+B)$ space algorithm.

Single-vehicle scheduling. The general problem is to schedule a route for a vehicle to visit n given sites each of which has a time window during which the vehicle is allowed to visit that site. The goal is to minimize a certain objective function of the route (e.g., time or distance), if such a route is possible. This problem is clearly a generalization of the Traveling Salesperson Problem and is NP-hard even for some very special cases. For example, it is NP-hard for the case in which a vehicle is to visit n sites on a straight line (equivalently, a ship is to visit n harbors on a convex shoreline) with time windows whose start times and end times (i.e., *deadlines*) are arbitrary [5]. Psaraftis *et al.* [18] gave an $O(n^2)$ time and space dynamic programming algorithm for the case with n sites on a

straight line whose time windows have only (possibly different) start times. Chan and Young [5] gave an $O(n^2)$ time and space dynamic programming algorithm for the case with n sites on a straight line whose time windows have the same start time but various deadlines. We present $O(n^2)$ time, $O(n)$ space algorithms for both these cases.

Due to the space limit, quite a few of our algorithms must be left to the full paper. Also, the proofs of our lemmas are left to the full paper.

2 Clipped Trees

A key ingredient of our general paradigm is the data structure called *clipped trees* that we introduce in this section. Clipped trees are important to our paradigm because they contain information needed for carrying out techniques such as marriage-before-conquer and prune-and-search.

In a nutshell, a clipped tree T is a "compressed" version of a corresponding single-source optimal path tree SST, such that T consists of a (usually sparse) sample set of the nodes of SST and maintains certain topological structures of SST. The sample nodes are selected from SST based on a certain criterion (e.g., geometric or graphical) that depends on the specific problem.

Let T^0 be a rooted tree with root node r. Let S be a set of sample nodes from T^0 with $r \in S$. A clipped tree T of T^0 based on the sample set S is defined as follows:

- The nodes of the clipped tree T are precisely those in S.
- For every node $v \in S - \{r\}$, the parent of v in T is the nearest proper ancestor w of v in T^0 such that $w \in S$.

Clearly, the size of T is $O(|S|)$. If S consists of all the nodes of T^0, then T is simply T^0 itself. The clipped tree T of T^0 can be obtained by the following simple procedure:

- Make the root r of T^0 the root of T, and pass down to all children of r in T^0 a pointer to r.
- For every node v of T^0 that receives from its parent in T^0 a pointer to a proper ancestor node w of v in T^0 (inductively, w is already a node of T), do the following: If $v \in S$, then add v to T, make w the parent of v in T, and pass down to all children of v in T^0 (if any) a pointer to v; otherwise, pass down to all children of v in T^0 (if any) the pointer to w.

It is easy to see that it takes $O(|T^0|)$ time to construct the clipped tree T from T^0 and the sample set S, and $O(|S|)$ space to store T. Also, observe that the above procedure need not have the tree T^0 explicitly stored. In fact, as long as the nodes of T^0 are produced in any parent-to-children order, T can be constructed. Note that this is precisely the order in which a single-source optimal path tree grows, and this growing process takes place in the same time as the lengths of optimal paths are being computed. Further, observe that one need not have the sample set S explicitly available in order to construct T. As long as a criterion

is available for deciding (preferably in $O(1)$ time) whether any given node v of T^0 belongs to the sample set S, the above procedure is applicable.

Consequently, one can use an algorithm for computing the lengths of single-source optimal paths and a criterion for determining the membership for a sample set S of the nodes of the single-source optimal path tree SST to construct a clipped tree T based on SST and S, without having to store SST. Actually, when T is being constructed, it is beneficial to associate with the nodes of T some information about the corresponding optimal paths to which these nodes belong. Once the process of computing the lengths of single-source optimal paths terminates, the clipped tree T, together with useful optimal path information stored in its nodes, is obtained.

Perhaps we should point out a seemingly minor but probably subtle aspect: The above procedure for building a clipped tree depends only on the ability to generate a single-source optimal path tree in a parent-to-children (or source-to-destination) order. This is crucial for the applicability of our general paradigm. In contrast, the marriage-before-conquer algorithm in [14] computes an actual optimal path using both the source-to-destination and destination-to-source orders. Although the problem in [14] is symmetric with respect to these two orders, it need not be the case with many other optimal path problems. For example, for some dynamic programming problems that are solvable by following a source-to-destination order (e.g., [5,18]), it may be quite difficult or even impossible to use the destination-to-source order. This aspect of clipped trees also enables us to avoid using the recursive back-up method of [9], since it may be difficult to use this back-up method to significantly reduce the sizes of the subproblems in a marriage-before-conquer algorithm.

3 Shortest Paths in an Arrangement

In this section, we illustrate our algorithmic paradigm with algorithms for finding a shortest path and its length between two points in the arrangement of lines on the plane. The problem can be stated as follows: Given a set H of n planar lines and two points s and t on some lines of H, find an s-to-t path of the shortest Euclidean distance that is restricted to lie on the lines of H. As mentioned in [4,12], to solve this geometric shortest path problem, one can construct a planar graph of size $O(n^2)$ that represents the arrangement of H and then apply the optimal algorithm for computing a shortest path in a planar graph [15]. Such an algorithm (even for the path *length*) uses $O(n^2)$ time and space, and it has been an open problem to improve these bounds. Although we are not yet able to improve the asymptotic time bound, we show how to reduce the space bound by a factor of n. Our first algorithm reports the length of the shortest s-to-t path in $O(n)$ space and $O(n \log n + K)$ time. Our second algorithm finds an actual shortest path in $O(n)$ space and $O(n \log^2 n \log(K/n) + \min\{n^2, K \log n\})$ time. Here, K is the size of the part of the arrangement of H that is inside a special convex polygonal region R (R will be defined below), and $K = O(n^2)$ in the worst case. Hence both our algorithms in the worst case take $O(n)$ space and $O(n^2)$ time.

Our path length algorithm is based on the topological sweep [9] and topological walk [2,3] techniques on planar arrangements. Our actual path algorithm makes use of additional techniques such as marriage-before-conquer, prune-and-search, and the clipped tree data structure. Our solutions also exploit a number of interesting observations on this particular problem.

3.1 Topological Sweep and Topological Walk

Arrangements are a fundamental structure in combinatorial and computational geometry [8], and a great deal of work has been devoted to studying various arrangements and their properties. Topological sweep [1,9] and topological walk [2,3] are two powerful space-efficient techniques for computing and traversing arrangements in a 2-D or 3-D space.

Let $H = \{l_1, l_2, \ldots, l_n\}$ be a set of n straight lines on a plane. The lines in H partition the plane into a subdivision called the *arrangement* $A(H)$ of H. $A(H)$ consists of a set of convex regions (*cells*), each bounded by some edges (i.e., segments of the lines in H) and vertices (i.e., intersection points between the lines). In general, $A(H)$ consists of $O(n^2)$ cells, edges, and vertices. Without loss of generality (WLOG), we assume that the lines in H are in general position, i.e., no three lines meet at the same point (the general case can be handled by using the techniques in [10]).

If one is interested only in constructing and reporting (but not storing) $A(H)$, then this can be done by a relatively easy algorithm that sweeps the plane with a vertical line, in $O(n^2 \log n)$ time and $O(n)$ space [11]. Edelsbrunner and Guibas [9] discovered the novel *topological sweep* approach for constructing and reporting $A(H)$ in $O(n^2)$ time and $O(n)$ space. The topological sweep approach sweeps the plane with an unbounded simple curve that is monotone to the y-axis and that intersects each line of H exactly once. Asano, Guibas, and Tokuyama [2] developed another approach, called *topological walk*, for constructing and reporting $A(H)$ in $O(n^2)$ time and $O(n)$ space. Essentially, a topological walk traverses $A(H)$ in a depth-first search fashion [2,3]. This approach can be extended to traversing a portion of $A(H)$ that is inside a convex polygonal region R on the plane in $O(K + (n + |R|) \log(n + |R|))$ time and $O(n + |R|)$ space, where K is the size of the portion of $A(H)$ in R and $|R|$ is the number of vertices of R.

3.2 Computing Shortest Path Lengths

We begin with some preliminaries. Let s and t be the source and destination points on the arrangement $A(H)$ for the sought shortest path. Let $\overline{st}$ be the line segment connecting s and t. WLOG, assume $\overline{st}$ is horizontal with s as the left end vertex. Let $H_c(\overline{st})$ be the set of lines in H that intersect the interior of $\overline{st}$, called the *crossing lines* of $\overline{st}$. Let $HP(H - H_c(\overline{st}))$ be the set of half-planes each of which is bounded by a line in $H - H_c(\overline{st})$ and contains $\overline{st}$. As observed in [4], since no shortest path in $A(H)$ can cross a line in H twice, one can restrict the search of a shortest s-to-t path to the (possibly unbounded) convex polygonal region R that is the common intersection of the half-planes in $HP(H - H_c(\overline{st}))$.

Hence, the problem of finding a shortest s-to-t path in $A(H)$ can be reduced in $O(n \log n)$ time to that of finding a shortest s-to-t path in the portion of $A(H)$ contained in R (by computing the common intersection of the half-planes in $HP(H - H_c(\overline{st}))$ and identifying the crossing lines of $\overline{st}$). Henceforth, we let n denote the number of lines of H intersecting the convex region R and let A_R denote the portion $A(H) \setminus R$ of $A(H)$.

We use topological walk, starting at s, to report the length of the shortest s-to-t path in A_R. In fact, our algorithm correctly reports the length of the shortest path from s to every vertex in A_R. The following is a simple yet useful lemma to our algorithm.

Lemma 1. *For any line $l \in H$ and any vertex v of $A(H)$ on l, no shortest s-to-v path in $A(H)$ can cross l (i.e., intersecting the interior of both the half-planes bounded by l).*

We use Lemma 1 in conjunction with the fact that, for a line $l \in H_c(\overline{st})$ and a vertex v of A_R on l, the topological walk visits v by following paths in A_R that stay on the left half-plane of l before visiting v on any path that crosses l (this follows from the definition of the topological walk [2,3]). By incorporating the computation of shortest path lengths with the construction and traversing of A_R by the topological walk, we obtain an algorithm for computing the lengths of the single-source shortest paths in A_R from the source s. Furthermore, the shortest path lengths are computed in the parent-to-children order in the single-source shortest path tree rooted at s (the details of the algorithm are a little tedious and left to the full paper). Hence we have the following lemma.

Lemma 2. *The length of the shortest path in A_R from s to every vertex of A_R can be computed in $O(n \log n + K)$ time and $O(n)$ space, where K is the number of vertices of A_R.*

3.3 Computing an Actual Shortest Path

In this subsection, we present our $O(n \log^2 n \log(K/n) + \min\{n^2, K \log n\})$ time, $O(n)$ space algorithm for reporting an actual shortest s-to-t path in A_R, where K is the size of A_R. In contrast, this algorithm is more interesting yet more sophisticated than the shortest path length algorithm.

Let v be a vertex on a shortest s-to-t path in A_R such that v is the intersection of two lines l_i and l_j of H and such that at least one of l_i and l_j is a crossing line of $\overline{st}$. Let $SP(s,t)$ denote the shortest s-to-t path in A_R. Then $SP(s,t) = SP(s,v) \cup SP(v,t)$. The following lemmas are a key to our algorithm.

Lemma 3. *The two lines l_i and l_j of H define two interior-disjoint convex subregions R_1 and R_2 in R such that $SP(s,v)$ stays within R_1 and $SP(v,t)$ stays within R_2. Further, at most four lines of H (two of them are l_i and l_j) can appear on the boundaries of both R_1 and R_2.*

Lemma 4. *The lines in H that $SP(s,t)$ crosses are exactly the crossing lines of $\overline{st}$ (i.e., $H_c(\overline{st})$).*

Lemma 5. *Let l_i and l_j be defined as in Lemma 3. The crossing lines of $\overline{st}$ in $H_c(\overline{st}) - \{l_i, l_j\}$ can be partitioned into two subsets H_1 and H_2, such that no line in H_1 (resp., H_2) intersects $SP(v;t)$ (resp., $SP(s;v)$). Moreover, H_1 (resp., H_2) consists of all the lines in $H_c(\overline{st}) - \{l_i, l_j\}$ that intersect the interior of the line segment $\overline{sv}$ (resp., $\overline{vt}$), i.e., $H_1 = H_c(\overline{sv})$ (resp., $H_2 = H_c(\overline{vt})$).*

Lemma 5 implies that if we are to compute $SP(s;v)$ (resp., $SP(v;t)$), the lines in $H_c(\overline{vt})$ (resp., $H_c(\overline{sv})$) need not be considered. Let v_c denote the number of lines in H crossed by $SP(s;v)$ (i.e., $v_c = |H_c(\overline{sv})|$), called the *crossing number* of v. If we could somehow find a vertex v on $SP(s;t)$ such that its crossing number v_c is (roughly) half the crossing number t_c of t, then we would have an efficient marriage-before-conquer algorithm for reporting $SP(s;t)$. This is because we would be able to recursively report the subpaths $SP(s;v)$ and $SP(v;t)$ in R_1 and R_2, respectively (by Lemma 3). Moreover, when computing $SP(s;v)$ and $SP(v;t)$, we would not have to consider the intersections between lines from the two line sets $H_c(\overline{sv})$ and $H_c(\overline{vt})$, thus eliminating from further consideration a constant fraction of the total $O(n^2)$ intersections of $A(H)$ among the n lines.

The next lemma makes it possible for an incremental method to compute the crossing numbers.

Lemma 6. *Let u and w be two neighboring vertices of $A(H)$ on a line $l \not\supseteq H$ (i.e., $\overline{uw}$ is an edge of $A(H)$ on l). Let the line $l(u)$ (resp., $l(w)$) of H intersect l at u (resp., w). Then $H_c(\overline{su})$ differs from $H_c(\overline{sw})$ on at most two elements. Furthermore, these different elements are in $\{l(u), l(w)\}$.*

Based on Lemma 6, if the crossing number u_c of a vertex u of $A(H)$ is already known, then it is easy to compute w_c for a neighboring vertex w of u in $A(H)$. This immediately implies that the crossing numbers of the vertices of A_R can be computed by a topological walk starting from the source vertex s (with $s_c = 0$). In particular, our shortest path length algorithm can be easily modified to report (but not store) the crossing numbers of the vertices of A_R.

At this point, it might be tempting to try to compute an actual path $SP(s;t)$ with the algorithm below. Let k be half the crossing number t_c of t ($k = \frac{t_c}{2}$ can be obtained by running the shortest path length algorithm on A_R once, as a preprocessing step). Then do the following.

1. If $k = O(\sqrt{n})$, then report $SP(s;t)$ by a tree-growing approach in A_R. Otherwise, continue.
2. Run the path length algorithm on A_R, and build a clipped tree T with sample nodes s, t, and all vertices u of A_R such that u is on a crossing line of $\overline{st}$ (by Lemma 3) and $u_c = k$.
3. From the clipped tree T, find a vertex v on $SP(s;t)$ such that $v_c = k$ (the parent node of t in T is such a vertex).
4. Using the vertex v, recursively report the subpaths $SP(s;v)$ and $SP(v;t)$ in R_1 and R_2.

The above algorithm, however, does not work well due to one difficulty: The size of the sample node set S for T is *super-linear*! The astute reader may have

observed that the size of the sample set S is closely related to the well-known problem on the combinatorial complexity of the k-th level of the arrangement of n planar lines. The best known lower bound for the k-th level size of such an arrangement is $O(n \log(k+1))$ [13,16], and the best known upper bound is $O(nk^{1=3})$ [7]. Hence the clipped tree T based on such a sample set S would use super-linear space, not the desired $O(n)$ space. To resolve this difficulty, we avoid using these vertices u as sample nodes for T such that u is on a crossing line of $\overline{st}$ and $u_c = k$. Instead, we use a prune-and-search approach to locate a vertex v on $SP(s;t)$ such that v is on a crossing line of $\overline{st}$ and $v_c = k$. Our prune-and-search procedure is based on some additional observations and (again) on the clipped tree data structure.

Lemma 7. *For any two vertices u and w of $A(H)$ such that u is on $SP(s;w)$, $w_c \quad u_c$.*

Lemma 8. *It is possible to find, in $O(K + n\log n)$ time and $O(n)$ space, a vertical line L such that L partitions the K vertices of A_R into two subsets of sizes c_1K and c_2K, where c_1 and c_2 are both positive constants and $c_1 + c_2 = 1$.*

Lemma 7 provides a structure on $SP(s;t)$ for searching, and Lemma 8 provides a means for pruning. The procedure below finds such a desired vertex v on $SP(s\ ;t\ )$ in a convex subregion $R\ $ of R that (possibly) is between two vertical lines (initially, $R\ = R$, $s\ = s$, and $t\ = t$).

1. Let $K\ $ be the number of vertices of $A_R\ = A_R \setminus R\ $. If $K\ = O(n)$, then find the desired vertex v on $SP(s\ ;t\ )$ in $A_R\ $ by a tree-growing approach. Otherwise, continue.
2. Compute a vertical line L as specified in Lemma 8. Let L partition the region $R\ $ into two convex subregions R^0 and R^{00}. Let S be the set of sample nodes that includes $s\ , t\ $, and all vertices u and w of $A_R\ $ such that $\overline{uw}$ is an edge of $A_R\ $ that intersects L. Note that $jSj = O(n)$ since L intersects each line of H once.
3. Run the path length algorithm on $A_R\ $, and build a clipped tree T based on the sample node set S. Associate with each node of S its crossing number.
4. Find all proper ancestors $s\ , u_1, u_2, :::; u_r$ of $t\ $ in T. If T contains no such nodes u_i, then $SP(s\ ;t\ )$ does not touch the vertical line L and hence the search for v is reduced to the subregion (say) R^0 containing $s\ $ and $t\ $; go to Step 6. Otherwise, go to Step 5.
5. T contains such nodes u_i, and hence $SP(s\ ;t\ )$ touches L (possibly multiple times). Let $u_1, u_2, :::; u_r$ appear along $SP(s\ ;t\ )$ in the $s\ $-to-$t\ $ order. Then, either the desired vertex $v\ 2\ fu_1; u_2; :::; u_rg$, or v is an interior vertex on exactly one path $SP(u_i; u_{i+1})$, $i = 0; 1; :::; r$ (with $u_0 = s\ $ and $u_{r+1} = t\ $). Note that based on the definition of the sample set S, such a path $SP(u_i; u_{i+1})$ stays completely inside one of the subregions R^0 and R^{00}.
6. Let R^0 be the subregion containing v. Search for v on $SP(u_i; u_{i+1})$ recursively in R^0.

It is not hard to show that the above procedure takes $O(K + n\log n \log(K/n))$ time and $O(n)$ space, where $K = O(|A_R|)$. Making use of this procedure, we are able to report an actual path $SP(s;t)$ in $O(n\log^2 n \log(K/n) + \min\{n^2; K\log n\})$ time and $O(n)$ space (the complete details of the $SP(s;t)$ algorithm and its analysis are left to the full paper).

Remark: By using a prune-and-search procedure and a marriage-before-conquer algorithm of a similar nature as those above, we are able to find an actual longest monotone concave path in $A(H)$ on the plane in $O(n\log n \log(K/n) + \min\{n^2; K\log n\})$ time and $O(n)$ space, improving the $O(n^2\log n)$ time, $O(n\log n)$ space and $O(n^3)$ time, $O(n)$ space solutions in [9].

4 Longest Monotone Paths in an Arrangement

In this section, we illustrate our algorithmic paradigm with an algorithm for computing a longest monotone path in the planar arrangement $A(H)$. This algorithm makes use of topological sweep [9] and topological walk [2,3] on arrangements, and of the clipped tree data structure. It takes $O(\frac{n^2 \log n}{\log(h+1)})$ time and $O(nh)$ space, where h is any integer such that $1 \quad h \quad n$ for any positive constant with < 1. This is an improvement over the $O(n^2\log n)$ time, $O(n\log n)$ space solution in [9].

A monotone path in $A(H)$ is a continuous curve consisting of edges and vertices of $A(H)$, such that every vertical line intersects in exactly one point. A vertex of is a *turn* if the two incident edges are not collinear. The length of is defined as the number of its turns plus one. The longest monotone path in $A(H)$ is denoted by $LMP(A(H))$.

Edelsbrunner and Guibas [9] used topological sweep to compute the length of $LMP(A(H))$, in $O(n^2)$ time and $O(n)$ space. To find the actual path $LMP(A(H))$, they used a recursive back-up method that maintains some "snapshots" which are states of their sweeping process. Storing each snapshot uses $O(n)$ space, which enables them to resume the sweeping process of their algorithm at the corresponding state, without having to start from the scratch. As it turns out, the algorithm for reporting $LMP(A(H))$ in [9] needs to maintain simultaneously $O(\log n)$ snapshots. Altogether, it takes $O(n^2\log n)$ time and $O(n\log n)$ space.

Our techniques are different from [9]. We use a marriage-before-conquer approach and a clipped tree whose sample node set is determined by h y-monotone curves $C_1; C_2; \ldots; C_h$, where each $C_i; i = 1; \ldots; h$, is a snapshot of the y-monotone curve used in the topological sweep [9]. Those curves partition the $O(n^2)$ vertices of $A(H)$ into $h+1$ subsets of (roughly) equal sizes of $O(n^2/(h+1))$. With the clipped tree, we can identify for each C_i a vertex v_i on the path $LMP(A(H))$, such that v_i is adjacent to an edge $e(v_i)$ of $LMP(A(H))$ that intersects C_i. After the h vertices $v_1; v_2; \ldots; v_h$ are identified, the problem is reduced to $h+1$ subproblems. The i-th subproblem is to find a longest monotone subpath between v_i and v_{i+1} in the region delimited by C_i and C_{i+1} (initially, with v_0 and v_{h+1} being on the vertical lines $x = -1$ and $x = +1$, respectively). We then solve the subproblem on each such region recursively, until the region

for each subproblem contains only $O(nh)$ vertices of $A(H)$ (at that point, we simply use a tree-growing approach to report the portion of $LMP(A(H))$ in that region). In the above algorithm, once $v_1; v_2; :::; v_h$ are identified, we associate v_i with the number of vertices of $A(H)$, denoted by num_i, between C_i and C_{i+1}. Therefore we can release the space occupied by the snapshots for $C_2; C_3; :::; C_h$. When the first subproblem is solved, we do a sweeping, starting from the last snapshot made in the first subproblem, in order to restore the snapshot of C_2. This is done by counting the number of $A(H)$ vertices until the num_1-th vertex is met by the sweeping. This computation continues with the other subproblems.

Note that, unlike our algorithm for the actual *shortest* path problem on $A(H)$, it is not clear to us how the sizes of the arrangement portions for the subproblems can be significantly reduced. One reason for this is that a longest monotone path in $A(H)$ can cross a line in H multiple times.

Our algorithm takes $O(\frac{n^2 \log n}{\log(h+1)})$ time and $O(nh)$ space. We leave the details of this algorithm, the correctness proof, and analysis to the full paper.

5 Dynamic Programming Problems

We briefly characterize the class of dynamic programming problems to which our general paradigm is applicable. Generally speaking, our paradigm applies to problems of the following nature:

- The problem seeks an optimal solution that consists of a value (e.g., an optimal path length) and a structure formed by a set of actual elements (e.g., an actual optimal path).
- The optimal value can be obtained by a dynamic programming approach by building a table M, such that each row of M is computed from $O(1)$ immediately preceding rows.

Let the table M have n columns and k rows (where n is the size of the input). Using our clipped tree based paradigm, we can report an actual solution by first finding an element of the actual solution at row k=2 (if row k=2 contains such an element), and then recursively solving the subproblems on the two subtables of M (one above and the other below row k=2).

Depending on the particular structures of the problems, one of two possibilities may occur. One possible situation is that the original problem of size n can be reduced to solving two independent subproblems of size r and size $n - r$, resulting in that a constant fraction of the entries of M is eliminated from consideration when solving these subproblems. Our algorithms for finding an actual solution for problems of this type have the same time and space bounds as those for computing the optimal value. Another possible situation is that it is not clear how to reduce the original problem of size n to two independent subproblems of sizes r and $n - r$ (i.e., each subproblem is still of size n), forcing one to use virtually the whole table M when solving the subproblems. Our algorithms for finding an actual solution for problems of this type have the same space bound as that for computing the optimal value, and a time bound with an extra $\log k$ factor.

References

1. E.G. Anagnostou, L.J. Guibas, and V.G. Polimenis, "Topological sweeping in three dimensions," *Lecture Notes in Computer Science, Vol. 450, Proc. SIGAL International Symp. on Algorithms*, Springer-Verlag, 1990, pp. 310–317.
2. T. Asano, L.J. Guibas, and T. Tokuyama, "Walking in an arrangement topologically," *Int. J. of Computational Geometry & Applications*, Vol. 4, No. 2, 1994, pp. 123–151.
3. T. Asano and T. Tokuyama, "Topological walk revisited," *Proc. 6th Canadian Conf. on Computational Geometry*, 1994, pp. 1–6.
4. P. Bose, W. Evans, D. Kirkpatrick, M. McAllister, and J. Snoeyink, "Approximating shortest paths in arrangements of lines," *Proc. 8th Canadian Conf. on Computational Geometry*, 1996, pp. 143–148.
5. C. Chan and G.H. Young, "Single-vehicle scheduling problem on a straight line with time window constraints," *Proc. 1st Annual International Conf. on Computing and Combinatorics*, 1995, pp. 617–626.
6. V. Chvatal, D.A. Klarner, and D.E. Knuth, "Selected combinatorial research problems," STAN-CS-72-292, 26, Stanford University, June 1972.
7. T. Dey, "Improved bounds for k-sets and k-th levels," *Proc. 38th Annual IEEE Symp. on Foundations of Computer Science*, 1997.
8. H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, 1987.
9. H. Edelsbrunner and L.J. Guibas, "Topologically sweeping an arrangement," *J. of Computer and System Sciences*, Vol. 38, 1989, pp. 165–194.
10. H. Edelsbrunner and E.P. Mücke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *ACM Trans. on Graphics*, Vol. 9, 1990, pp. 66–104.
11. H. Edelsbrunner and E. Welzl, "Constructing belts in two-dimensional arrangements with applications," *SIAM J. Comput.*, Vol. 15, 1986, pp. 271–284.
12. D. Eppstein and D. Hart, "An efficient algorithm for shortest paths in vertical and horizontal segments," *Lecture Notes in Computer Science, Vol. 1272, Proc. 5th International Workshop on Algorithms and Data Structures*, Springer-Verlag, 1997, pp. 234-247.
13. P. Erdös, L. Lovász, A. Simmons, and E.G. Straus, "Dissection graphs of planar point sets," In J.N. Srivastava *et al.* (Eds.), *A Survey of Combinatorial Theory*, North Holland, 1973, pp. 139–149.
14. D.S. Hirschberg, "A linear-space algorithm for computing maximal common subsequences," *Comm. ACM*, Vol. 18, No. 6, 1975, pp. 341–343.
15. P.N. Klein, S. Rao, M.H. Rauch, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Proc. 26th Annual ACM Symp. Theory of Computing*, 1994, pp. 27–37.
16. L. Lovász, "On the number of halving lines," *Ann. Univ. Sci. Budapest, Eötvös, Sec. Math.*, Vol. 14, 1971, pp. 107–108.
17. S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, Wiley, New York, 1990.
18. H.N. Psaraftis, M.M. Solomon, T.L. Magnanti and T.-U. Kim, "Routing and scheduling on a shoreline with release times," *Management Science*, Vol. 36, No. 2, 1990, pp. 212–223.
19. R.A. Wagner and M.J. Fischer, "The string-to-string correction problem," *J. ACM*, Vol. 21, No. 1, 1974, pp. 168–173.

An Experimental Study of Dynamic Algorithms for Directed Graphs *

Daniele Frigioni[1,2], Tobias Miller[1], Umberto Nanni[2], Giulio Pasqualone[2], Guido Schaefer[1], and Christos Zaroliagis[1,3]

[1] Max-Planck-Institut für Informatik, Im Stadtwald, 66123, Saarbrücken, Germany
{miller,shaefer,zarog}@mpi-sb.mpg.de
[2] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, I-00198 Roma, Italy
{pasqualo, frigioni, nanni}@dis.uniroma1.it
[3] Department of Computer Science, King's College, University of London Strand, London WC2R 2LS, UK

1 Introduction

Dynamic graph algorithms maintain a certain property (e.g., connectivity) of a graph that changes dynamically over time. Typical changes include insertion of a new edge and deletion of an existing edge. The challenge for a dynamic algorithm is to maintain, in an environment of dynamic changes, the desired graph property efficiently, i.e., without recomputing everything from scratch after each change. A problem is *fully dynamic* if both edge insertions and deletions are allowed, and it is *partially dynamic* if either edge insertions or edge deletions (but not both) are allowed. In the case of edge insertions (resp. deletions), the partially dynamic algorithm is called *incremental* (resp. *decremental*).

Dynamic graph algorithms have been an active and blossoming research field over the last years, due to their applications in a variety of contexts including operating systems, information systems, and network management. Many important theoretical results have been obtained for both fully and partially dynamic maintenance of several properties on undirected graphs. Recently, an equally important effort has started on implementing these techniques and showing their practical merits [1,2]. These were the first implementations concerning fully dynamic maintenance of certain properties (connectivity, minimum spanning tree) in undirected graphs.

On the other hand, the development of fully dynamic algorithms for directed graphs (digraphs) turned out to be a much harder problem and much of the research so far was concentrated on the design of partially dynamic algorithms. Actually, only fully dynamic *output bounded* algorithms for shortest paths [6,12], and a fully dynamic *randomized* solution for transitive closure [7] are currently known on directed graphs. However, despite the number of interesting theoretical results achieved (see e.g., [5,8,9,10,13]), very little has been done so far w.r.t. implementations even for the most fundamental problems.

* Work partially supported by the ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244. The work of the first author is supported by CNR–NATO Adv. Fellowships Program n. 215.29 of the Italian National Research Council (CNR).

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 368{380, 1998.
c Springer-Verlag Berlin Heidelberg 1998

In this paper, we are making a step forward in bridging the gap between theoretical results for directed graphs and their implementation by studying the practical properties of several dynamic algorithms for transitive closure on digraphs and depth first search and topological sorting on directed acyclic graphs (DAGs). The main goal of the paper is to confirm in practice the theoretical evidence that dynamic algorithms for digraphs are more efficient than their static counterparts, and characterize their behaviour in a given range of parameters. Due to space limitations, we report and discuss only on some of our experiments for the transitive closure problem. More experiments concerning this problem as well as depth first search and topological sorting will be given in the full paper.

The reported experiments concern incremental algorithms on digraphs, decremental algorithms on DAGs and fully dynamic algorithms on digraphs and DAGs. We have implemented the algorithms proposed in [5,7,8,9,13], plus several variants of them, and several simple-minded algorithms that are easy to implement and likely to be fast in practice. In addition, we have developed a new algorithm which is a variant of Italiano's algorithms [8,9] and whose decremental part applies to any digraph, not only to DAGs.

All the implementations have been written in `C++` using the LEDA library for combinatorial and geometric computing [11]. The source codes of our implementations along with the experimental data (input graphs and sequences of operations), as well as demo programs, are available via anonymous ftp from `ftp.dis.uniroma1.it` under `/pub/pasqualo` in the file `esa98exp.tgz`.

Our experiments have been performed by generating several kinds of random inputs (that are better specified in the following) on random graphs, and on a real world graph: the graph describing the connections among the autonomous systems of the Internet, as registered at the *RIPE* net server [4]. In these experiments we considered two types of queries: `Boolean` queries (which simply return yes or no), and `Path` queries (which return an actual path, if exists).

All the experiments performed have substantially confirmed our expectation about the practical behaviour of the implemented algorithms. In the incremental case with `Path` queries, Italiano's algorithm [8,9] was almost always the fastest among the dynamic algorithms. For dense graphs the simple-minded algorithms were better. In the case of `Boolean` queries the fine-tuning to the transitive closure problem of the general technique proposed in [5] was always the fastest. This is due to its very simple data structures. For the decremental case, Italiano's algorithm was faster in dense graphs for `Path` queries, while for sparse graphs the simple-minded algorithms were better. For `Boolean` queries the algorithm of [5] and the simple-minded algorithms were always the best, depending on the various cases considered. In fully dynamic settings on digraphs the simple-minded algorithms were always extremely faster than the Henzinger-King algorithm [7] and our variant of Italiano's algorithm. For DAGs and `Boolean` queries the algorithm of [5] was always the fastest except for very sparse graphs where the simple-minded algorithms competed with it.

2 Description of Implemented Algorithms

Given a digraph $G = (V; E)$, the *transitive closure* (or *reachability*) problem consists in finding if there is a directed path between any two given vertices in G. We say that a vertex v is *reachable* by vertex u iff there is a (directed) path from u to v in G. The digraph $G = (V; E)$ that has the same vertex set with G but has an edge $(u; v)$ 2 E iff v is reachable by u in G is called the *transitive closure* of G; we shall denote $jE\ j$ by m . If v is reachable from u in G, then we call v a *descendant* of u, and u an *ancestor* of v.

Our starting point was the implementation of three partially dynamic algorithms (Italiano's [8,9], Yellin's [13], and Cicerone et.al. [5]), as well as of several variants of them. The incremental version of these algorithms applies to any digraph, while the decremental one applies only to DAGs. We also developed a new algorithm (variant of Italiano's) whose decremental part applies to any digraph. Italiano's and Yellin's data structures support both `Path` and `Boolean` queries, while the data structure of Cicerone et.al. [5] supports only `Boolean` queries. Concerning fully dynamic algorithms, we have implemented one of the two randomized algorithms proposed by Henzinger and King [7]. This algorithm is based on a new partially dynamic (decremental) randomized algorithm given in the same paper [7] which we also implemented. This algorithm supports both `Boolean` queries and `Path` queries.

2.1 Italiano's algorithm and its variants

The main idea of the data structure proposed in [8,9] is to associate (and maintain) with every vertex u 2 V a set $DESC[u]$ containing all descendants of u in G. Each such set is organized as a spanning tree rooted at u. In addition, an n n matrix of pointers, called $INDEX$, is maintained which allows fast access to vertices in these trees. More precisely, $INDEX[i; j]$ points to vertex j in $DESC[i]$, if j 2 $DESC[i]$, and it is *Null* otherwise. If G_0 is the initial digraph having n vertices and m_0 edges, the data structure requires $O(n^2)$ space, and is initialized in $O(n^2 + nm_0)$ time.

A `Boolean` query for vertices i and j is carried out in $O(1)$ time, by simply checking $INDEX[i; j]$. A `Path` query for i and j is carried out in $O(')$ time, where $'$ is the number of edges of the reported path, by making a bottom-up traversal from j to i in $DESC[i]$. The incremental part of the data structure requires $O(n(m_0+m))$ time to process a sequence of m edge insertions, while the decremental one requires $O(nm_0)$ time to process any number (m) of deletions; hence, if $m = \Omega(m_0)$, then an update operation takes $O(n)$ amortized time.

The insertion of an edge $(i; j)$ is done as follows. The data structure is updated only if there is no i-j path in G. The insertion of edge $(i; j)$ may create new paths from any ancestor u of i to any descendant of j only if there was no previous u-j path in G. In such a case the tree $DESC[u]$ is updated using the information in $DESC[j]$ (deleting duplicate vertices) and accordingly the u-th row of $INDEX$.

The deletion of an edge $(i; j)$ on a DAG G is done as follows. If $(i; j)$ does not belong to any $DESC$ tree, then the data structure is not updated. Otherwise,

$(i;j)$ should be deleted from all $DESC$ trees to which it belongs. Assume that $(i;j)$ belongs to $DESC[u]$. The deletion of $(i;j)$ from $DESC[u]$ splits it into two subtrees, and a new tree should be reconstructed. This is accomplished as follows. Check whether there exists a u-j path that avoids $(i;j)$; this is done by checking if there is an edge $(v;j)$ in G such that the u-v path in $DESC[u]$ avoids $(i;j)$. If such an edge exists, then swap $(i;j)$ with $(v;j)$ in $DESC[u]$, and join the two subtrees using $(v;j)$. In such a case, $(v;j)$ is called a *valid replacement* for $(i;j)$, and v is called a *hook* for j. If such an edge does not exist, then there is no u-v path in G and consequently j cannot be a descendant of u anymore: delete j from $DESC[u]$ and proceed recursively by deleting the outgoing edges of j in $DESC[u]$. We shall refer to the implementation of the above described algorithm as `Ital`.

Finally, we have extended the above idea and developed a new algorithm whose decremental part applies to any digraph, and not only to DAGs. In this case a sequence of m edge insertions requires $O(n(m_0+m))$ time; any sequence of edge deletions requires either $O(m_0^2)$ worst-case time, or $O(n^2 + nm_0)$ expected time. We shall refer to this algorithm as `Ital-Gen`.

This algorithm is based on the fact that if we shrink every strongly connected component to a vertex, then the resulting graph is a DAG. Note that a similar idea was used in [10]; however, the data structures and techniques used in that paper can answer only `Boolean` queries. A major difference of `Ital-Gen` with `Ital` is that we don't keep explicitly the $DESC$ trees as graphs; instead, we maintain them implicitly. We also maintain information regarding the strongly connected components (SCCs) which is crucial for the decremental part of the algorithm. Our data structure consists of: (a) an $n \quad n$ Boolean matrix $INDEX$, where $INDEX[i;j]$ = true iff j is a descendant of i in G; (b) an array SCC of length n, where $SCC[i]$ points to the SCC containing vertex i; and (c) the SCCs of G as graphs. Furthermore, with each k-vertex SCC we maintain two additional data structures: an array $HOOK$ of length n, where $HOOK[i]$ points to the incoming edge of the SCC used in the (implicitly represented) descendant tree rooted at i; and a sparse representative of the SCC consisting of k vertices and $2k - 2$ edges. The sparse representative is computed as follows. Take any vertex r of the SCC and perform a DFS rooted at r. Then, reverse the directions of all edges in the SCC and perform a second DFS rooted at r. Restore the original direction of the edges in the second DFS tree. The union of the two resulted trees is the required sparse representative. By replacing every SCC in G with a single vertex, we obtain a graph which is a DAG. We shall refer to this graph as the *DAG-part* of G and the vertices replacing SCCs as *supervertices*.

The initialization requires $O(n^2 + nm_0)$ time and $O(n^2)$ space, and involves computation of the above data structures where the computation of the SCC and their sparse representatives is performed only for the decremental part of the algorithm, i.e., before any sequence of edge deletions. (If there is no such sequence, then every vertex is taken as a SCC by itself.)

`Boolean` and `Path` queries can be answered in the same time bounds of `Ital`, by first looking at the $INDEX$ matrix to check whether there exists a path; if

yes, then the path can be found using the *HOOK* arrays (which provide the path in the DAG-part of G) and the sparse representative of each SCC (which gives the parts of the required path represented by supervertices in the DAG-part).

The insertion of an edge $(i; j)$ is done similarly to `Ital`.

Deleting an edge $(i; j)$ is done as follows. If $(i; j)$ does not belong to any SCC, then we use `Ital` to delete $(i; j)$ from the DAG-part of G. Otherwise, we check if $(i; j)$ belongs to the sparse representative of the SCC or not. In the latter case, we do nothing. In the former case, we check if the deletion of $(i; j)$ breaks the SCC. If the SCC doesn't break, we simply recompute the sparse representative. If the SCC breaks, then we compute the new SCC, update properly the *HOOK* arrays so that the information concerning hooks in the new DAG-part is preserved, and finally we call `Ital` to delete $(i; j)$ from the new DAG-part.

2.2 Yellin's algorithm

We first describe the `Boolean` version of Yellin's algorithm (`Yellin`), and then we show how to extend this data structure in order to handle `Path` queries.

Yellin's data structure associates with every vertex v the doubly linked list *Adjacent*(v) of the heads of its outgoing edges, and the doubly linked list *Reaches*(v) of the tails of its incoming edges. In addition, an $n \quad n$ array *INDEX* is maintained. Each entry *INDEX*$[v; w]$ has (among others) a field called *refcount* that stores the number of v-w paths in G, defined as *refcount*$(v; w)=$ $jref(v; w)j + 1$, if $(v; w)$ *2* E, and *refcount*$(v; w)=$ $jref(v; w)j$, otherwise, where $ref(v; w) = f(v; z; w) : z \ 2 \ V \ \wedge (v; z) \ 2 \ E \quad \wedge (z; w) \ 2 \ Eg$. If G_0 is the initial digraph having n vertices and m_0 edges, the data structure requires $O(n^2)$ space, and is initialized in $O(n^2 + nm_0)$ time.

The incremental version of `Yellin` requires $O(d(m_0 + m) \ )$ time to process a sequence of m edge insertions starting from G_0 and resulting in a digraph G; d is the maximum outdegree of G and $(m_0 + m) \ $ is the number of edges in G . The decremental version requires $O(dm_0)$ time to process any sequence of m edge deletions; d is the maximum outdegree of G_0.

The main idea for updating the data structure after the insertion of the edge $(a; b)$ is to find, for all $x; z \ 2 \ V$, the new triples $(x; y; z)$ that should be added to $ref(x; z)$ and update *INDEX*$[v; w]$.*refcount*. The insertion algorithm finds first all vertices x such that $(x; a)$ was an edge of G_{old} (the transitive closure graph before the insertion of $(a; b)$). In this case, $(x; a; b)$ is a new triple in $ref(x; b)$, and *refcount*$(x; b)$ has to be incremented. Then, the insertion algorithm considers each new edge $(x; y)$ in G_{new} (the transitive closure graph after the insertion of $(a; b)$) and each edge $(y; z)$ of G; $(x; y)$ is a new transitive closure edge if its *refcount* increased from 0 to 1. Now, $(x; y; z)$ is a new triple for $ref(x; z)$ and *refcount*$(x; z)$ is incremented. The edge deletion algorithm is the "dual" of the edge insertion algorithm described above.

The data structure described so far cannot return an arbitrary x-y path in G, if such a path exists. To support the `Path` query, the data structure is augmented with the so called *support graph*. The support graph is a digraph consisting of two kinds of vertices: closure vertices, that have a label $(x; z)$ corresponding to

an edge in G , and join vertices, that have a label $(x;y;z)$ corresponding to the triple $(x;y;z)$ in $ref(x;z)$. The support graph can be updated after an edge insertion or edge deletion within the same resource bounds. The `Path` version of `Yellin`, denoted as `Yellin-SG`, requires $O(n^2 + dm_0)$ time and space to be initialized, where d is the maximum outdegree of G_0.

2.3 The algorithm of Cicerone et al.

The algorithms in [5] provide a uniform approach for maintaining several binary relationships (e.g., transitive closure, dominance, transitive reduction) incrementally on digraphs and decrementally on DAGs. The implementation does not depend on the particular problem; i.e., the same procedures can be used for different problems by simply setting appropriate boundary conditions. The approach allows to define a *propagation property* (PP), based on a binary relationship $R \quad V \quad V$, that describes how R "propagates" along the edges of G; R satisfies PP over G with boundary condition $R_0 \quad R$ if, for any $hx;yi \ 2 \ V \quad V$, $hx;yi \ 2 \ R$ iff either $hx;yi \ 2 \ R_0$, or $x \ 6= \ y$ and there exists a vertex $z \ 6= \ y$ such that $hx;zi \ 2 \ R$ and $(z;y) \ 2 \ E$. R_0 defines the elements of R that cannot be deduced using PP. For example, if R is the transitive closure, then $R_0 = f(x;x) : x \ 2 \ V g$. Actually, if we consider the transitive closure alone, the algorithms of [5] collapse to the solution of La Poutré and van Leeuwen [10].

The idea is to maintain a matrix containing, for each pair $hx;yi \ 2 \ V \quad V$, the number $U_R[x;y]$ of edges useful to that pair. An edge $(z;y)$ is *useful* to pair $hx;yi$ if $z \ 6= \ y$ and $hx;zi \ 2 \ R$. It is easy to see that, for any pair $hx;yi \ 2 \ V \quad V$, $hx;yi \ 2 \ R$ iff either $hx;yi \ 2 \ R_0$ or $U_R[x;y] > 0$. For each vertex k, two data structures are maintained: (a) a set $\text{OUT}[k]$ that contains all edges outgoing k; and (b) a queue Q_k to handle edges $(h;y)$ useful to pair $hk;yi$. If G_0 is the initial digraph having n vertices and m_0 edges, the data structure requires $O(n^2)$ space, and is initialized in $O(n^2+nm_0)$ time. The incremental algorithm requires $O(n(m_0 + m))$ to process a sequence of m insertions, while the decremental one requires $O(nm_0)$ time to process any number (m) of deletions; hence, if $m = \Omega(m_0)$, then an update operation takes $O(n)$ amortized time.

After the insertion of edge $(i;j)$ the number of edges useful to any pair $hk;yi$ can only increase. An edge insertion is performed as follows: first of all, for each vertex k, the new edge $(i;j)$ is inserted into the empty queue Q_k if and only if $hk;ii \ 2 \ R$, and hence it is useful to pair $hk;ji$; then, edges $(t;h)$ are extracted from queues Q_k, and the values $U_R[k;h]$ are incremented by one, because these edges are useful to pair $(k;h)$. Now, edges $(h;y) \ 2 \ \text{OUT}[h]$ are inserted in Q_k if and only if the pair $hk;hi$ has been added for the first time to R due to an edge insertion (i.e., $U_R[k;h] = 1$). This implies that, during a sequence of edge insertions, the edge $(h;y)$ can be inserted in Q_k at most once. An edge deletion is handled similarly. In fact, some edges could no longer be useful to a pair $hk;yi$, and then the corresponding value $U_R[k;y]$ has to be properly decreased. In this case the queue Q_k is used to handle edges that are no longer useful to $hk;yi$.

We implemented the general technique described above, denoted as `CFNP`, and a fine-tuned version of it, denoted as `CFNP-TC`. The main difference is that

in the case of edge insertions, CFNP performs at least a computation of $O(n)$ in order to update the counters modified by the insertion, while CFNP-TC starts only when the inserted edge $(i; j)$ introduces a path between i and j and no such path existed before. This implies that, differently from CFNP, CFNP-TC cannot be used in a fully-dynamic environment on DAGs, because instead of the matrix of counters it simply maintains a binary matrix representing the transitive closure.

2.4 The Henzinger-King algorithms

The algorithms in [7] are based on the maintainance of BFS trees of vertices reachable from (or which reach) a specific distinguished vertex, and the fact that with very high probability every vertex in the graph reaches (or is reachable by) a distinguished vertex by a path of small distance (counted in number of edges). Let $out(x; k)$ (resp. $in(x; k)$) denote the set of vertices reachable from (resp. which reach) vertex x by a path of distance at most k. The decremental algorithm, denoted as HK-1, selects at random sets of distinguished vertices S_i, for $i = 1; : : : ; \log n$, where $jS_ij = \min fO(2^i \log n); ng$. For every $x \in S_i$ the algorithm maintains (a) $out(x; n=2^i)$ and $in(x; n=2^i)$; and (b) $Out(x) = [_{i:x \in S_i} out(x; n=2^i)$ and $In(x) = [_{i:x \in S_i} in(x; n=2^i)$. In addition, for each $u \in V$ the sets $out(u; \log^2 n)$ and $in(u; \log^2 n)$ are maintained.

In our implementation we have associated with every $out(x; k)$ (resp. $in(x; k)$) tree an array whose u-th entry equals 1 if there is a x-u (resp. u-x) path, and 0 otherwise. Moreover, if in $Out(x)$ (or in $In(x)$) there are more than one trees having as root the same vertex, we keep only the tree with the largest depth.

Any sequence of edge deletions requires $O(m_0 n \log^2 n)$ time. A query for vertices u and v is carried out as follows. Check if v is in $out(u; \log^2 n)$. If not, then check for any vertex x if $u \in In(x)$ and $v \in Out(x)$. If such an x exists, then there is a u-v path; otherwise, such a path does not exist with high probability. A Boolean-query is answered in $O(n= \log n)$ time and a find-path query in additional $O(`)$ time.

The fully dynamic algorithm, denoted as HK-2, keeps the above decremental data structure to give answers if there is an "old" path between two vertices (i.e., a path that does not use any of the newly inserted edges). Updates are carried out as follows. After the insertion of an edge $(i; j)$, compute $in(i; n)$ and $out(i; n)$. After the deletion of an edge, recompute $in(i; n)$ and $out(i; n)$ for all inserted edges $(i; j)$, and update the decremental data structure for old paths. Rebuild the decremental data structure after $p\overline{n}$ updates. Let m_0 be either the initial number of edges or the number of edges in G at the time of the last rebuild. The total time for a sequence of no more that $p\overline{n}$ insertions and $p\overline{n}$ deletions is $O(m_0 n \log^2 n + n^p\overline{n})$. Let $\hat{m}$ be the average number of edges in G during the sequence of updates. Since $m_0 \quad \hat{m} + p\overline{n}$, we have an $O(\hat{m}^p\overline{n} \log^2 n + n)$ amortized bound per update. Both algorithms are initialized in $O(nm_0)$ time.

To answer a query for u and v, check first if there is an old path between them. If not, then check if $u \in in(i; n)$ and $v \in out(i; n)$ for all i which are tails of the newly inserted edges $(i; j)$. The query bounds are the same of HK-1.

3 Experimental Results

All of our implementations follow closely the dynamic algorithms in [5,7,8,9,13], as well as their variants described above. They have been implemented as `C++` classes using several advanced data types of LEDA [11]. Each class is installed in an interface program that allows to read graphs and sequences from files, to perform operations, to store the results again on files and to show them on graphics. It is worth noting that our implementation of dynamic algorithms has been done using the new base class of LEDA for dynamic graphs [3].

We have also augmented the implementation of the original algorithms in [8,9,13] with additional procedures that help us to easily verify the correctness of the implementation. For example, a `Path`$(u; v)$ operation has been implemented either to return the u-v path (if it exists), or to exhibit a cut in the graph separating the vertices reachable by u from the rest of the graph. In the former case, the correctness check is trivial. In the latter case, the heads of all edges in the cut should belong to the part containing u.

All our experiments were run on a SUN Ultra SPARC-2 with one 170 MHz processor and 64 MB of main memory. The given time bounds were measured using the Unix function `getrusage`.

We performed our tests on random digraphs and random DAGs, and on random sequences of operations (edge insertions intermixed with queries, or edge deletions intermixed with queries, or fully-dynamic sequences). To ensure correctness, we generated for every dynamic algorithm a file containing the final graph (i.e., after all updates) and compared it with a similar file produced by a static algorithm executed after every update operation. Correctness implies that these files should be identical (a fact that was verified by our experiments).

We compared our implementations of dynamic algorithms with a static algorithm for transitive closure and simple heuristic algorithms that were easy to implement and likely to be fast in practice. The static algorithm used was the `TRANSITIVE_CLOSURE` algorithm provided by LEDA and requires $O(nm)$ time. This algorithm was called only in the case where a `Boolean` query was issued that was preceded by a dynamic change (edge insertion/deletion) on G. Despite this fact, it was much slower than any of the simple heuristic approaches described below and for this reason we don't report on experimental results regarding comparison with `TRANSITIVE_CLOSURE`.

The simple-minded algorithms that we implemented are based on the following method: in the case of an edge insertion (resp. deletion), the new (resp. existing) edge is simply added to (resp. removed from) the graph and nothing else is computed. In the case of a query, a search from the source vertex s is performed until the target vertex t is reached (if an s-t path exists) or until all vertices reachable from s are exhausted. Depending on the search method used (DFS, BFS, and a combination of them), we have made three different implementations that require no initialization time, $O(1)$ time per edge insertion or deletion, and $O(n + m)$ time per query operation. Our implementation of the simple-minded algorithms include: `DFS`, `BFS`, and `DBFS` which is a combination of `DFS` and `BFS` that works as follows. Vertices are visited in DFS order. Every time a vertex is

visited we first check whether any of its adjacent vertices is the target vertex. If yes, then we stop; otherwise, we continue the visit in a DFS manner.

We performed our tests on random digraphs and on random sequences of operations. We generated a large collection of data sets, each consisting of 5 samples. A data set corresponded to a fixed value of graph parameters. Each sample consisted of a random digraph initialized to contain n vertices and m_0 edges, and a random sequence of update operations (insertions/deletions) intermixed with `Boolean` or `Path` queries. Each query and update were uniformly mixed, i.e., occurred with probability 1=2 (we considered – as it is also assumed in [5,7,8,9,13] – an on-line environment with no prediction of the future, and where queries and updates are equally likely). We ran our algorithms on each sample and retained the average CPU time in seconds over the 5 samples for each program. The choice of 5 samples was motivated by the fact that in preliminary experiments we observed a very small variance.

For each experiment performed, among the implementations of the simple-minded algorithms (`BFS`, `DFS` and `DBFS`) we selected the fastest and reported comparisons only with it. Most of the times `BFS` and `DBFS` were the fastest. Furthermore, the performances of `Yellin-SG` were so slow and space consuming in our experiments, due to the maintenance of the support graph (occupying $O(n^3)$ space), that we do not report results regarding comparison with it. Finally, since all the algorithms require at least quadratic space, it was not easy to deal with dense instances of big graphs due to memory limitations; therefore, we report experiments on graphs of at most 300 vertices. All figures below show the time (in secs) to process a sequence of operations.

3.1 Edge insertions

We start with the case of `Path` queries. Figure 1 illustrates our experiments for random digraphs with 300 vertices and different values of m_0, and for evenly mixed random sequences of 10.000 operations (edge insertions and `Path` queries). Almost always the fastest algorithms were `Ital` and `DBFS`.

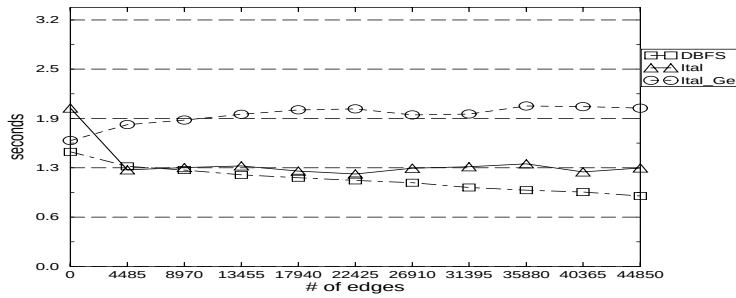


Fig. 1. 10.000 operations (`Path` queries and edge insertions) on digraphs with 300 vertices and different values of m_0.

`Ital` seems to be more robust, as it has the same behaviour regardless of graph density. `DBFS` is slower than `Ital` for sparse digraphs, and becomes faster

as the digraph density increases. Very similar results hold for different values of n, m_0 and number of operations.

For `Boolean` queries Figure 2 illustrates our experiments for random digraphs with 300 vertices and different (initial) edge densities, and for evenly mixed random sequences of 10.000 operations (edge insertions and `Boolean` queries). Very similar results hold for different values of n, m_0 and number of operations. All the algorithms have a stable behaviour, regardless of the initial graph density. This can be explained as follows: the graph becomes more dense and an edge insertion does not add many information to the transitive closure of the graph.

As expected, the fastest algorithm is always `CFNP-TC`, while the slowest one is always `Yellin`. An interesting observation is that `Ital-Gen` is faster than `Ital`. This is due to the fact that `Ital-Gen` maintains the descendant trees of the vertices implicitly, and not explicitly as `Ital`. In this case `CFNP-TC` is faster than `DBFS` because, for each query, `CFNP-TC` performs only a table lookup, while `DBFS` needs $\Omega(n)$ time.

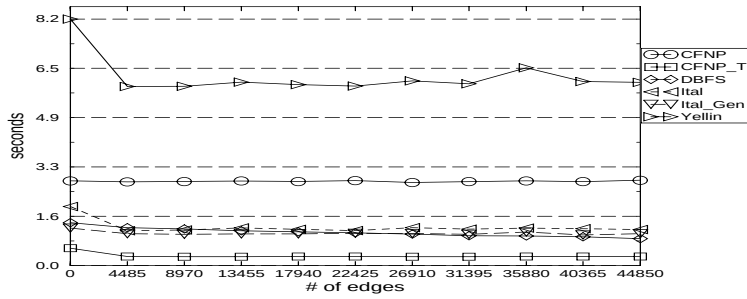


Fig. 2. 10.000 operations (`Boolean` queries and edge insertions) on digraphs with 300 vertices and different values of m_0.

3.2 Edge deletions

We start with the case of `Path` queries. Figure 3 illustrates our experiments for random DAGs with 300 vertices and different values of m_0, and for evenly mixed random sequences of 5000 edge deletions and `Path` queries. Similar results hold for different values of n, m_0 and number of operations.

For dense DAGs `Ital` is better than the other algorithms. This is due to the fact that the probability, for `Ital`, to find a hook quickly in such graphs is large. For sparser DAGs, however, the other algorithms (i.e., `BFS`, `DFS`, `DBFS` and `Ital-Gen`) are faster than `Ital`.

In the case of sequences of edge deletions and `Path` queries (or sequences of edge deletions and `Boolean` queries) on digraphs our experiments have shown that the simple-minded algorithms are always extremely faster than `HK-1`. Experiments with `HK-1` were always very time consuming, due to its heavy data structure, even for very small graphs.

In the case of `Boolean` queries, we have a similar situation. Figure 4 illustrates our experiments for random DAGs with 300 vertices and different values of m_0,

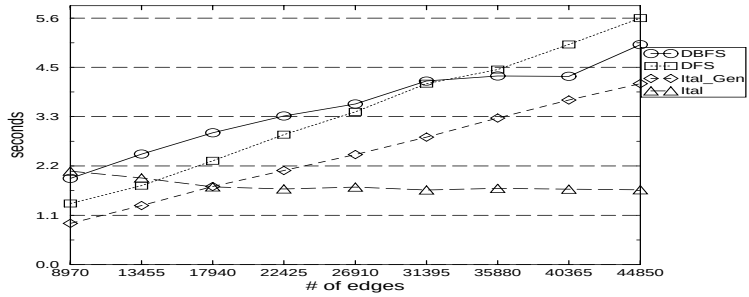


Fig. 3. 5.000 operations (`Path` queries and edge deletions) on DAGs with 300 vertices and different values of m_0.

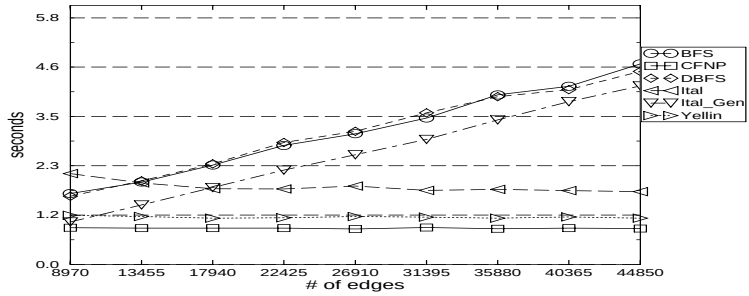


Fig. 4. 5.000 operations (`Boolean` queries and edge deletions) on DAGs with 300 vertices and different values of m_0.

and for evenly mixed random sequences of 5000 operations consisted of edge deletions and `Boolean` queries. `CFNP` is always the best and its behaviour is stable while the number of edges of the DAG increases, as well as the behaviour of `Ital` and `Yellin`. On the other hand (as expected) the performances of the simple-minded algorithms increase on denser DAGs and the dynamic algorithms become significantly better. For very sparse DAGs `CFNP` is better than `Yell`, and the simple-minded algorithms are better than `Ital`, because `Ital` cannot find the hook quickly.

3.3 Fully dynamic environment

`CFNP` and `Yellin` are suitable to be used in a fully dynamic setting, although in this case their theoretical bounds do not hold. On the other hand, `Ital` and `Ital-Gen` need a `Reset` operation to reset all hook entries before any sequence of edge insertions that is preceded by a sequence of edge deletions. This affects the amortized bounds per operation, since reseting all hooks takes $O(n^2)$ time.

In the experiments that we performed on digraphs the performances of `HK-2` and `Ital-Gen` were extremely slower than the simple-minded algorithms for long sequences of updates, and hence we do not report on this comparisons. On the other hand, `HK-2` is expected to be more efficient for short sequences. This fact was not confirmed by our experiments as it is shown for example in Figure 5.

In the case of `Boolean` queries (Figure 6) `CFNP` is always the fastest, and its behaviour is very stable (as the one of `Ital` and `Yellin`). In both cases

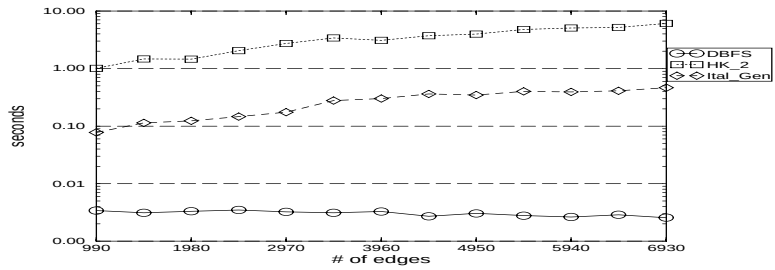


Fig. 5. 60 operations (`Boolean` queries, edge insertions and edge deletions) on digraphs with 100 vertices and different values of m_0.

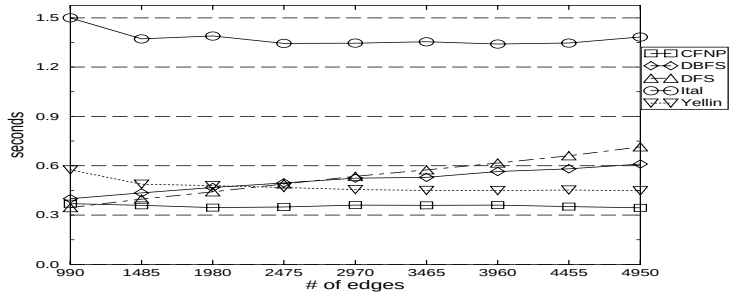


Fig. 6. 5.000 operations (`Boolean` queries, edge insertions and edge deletions) on DAGs with 100 vertices and different values of m_0.

`Ital` is the worst due to the `Reset` operation. In the case of `Path` queries, the simple-minded algorithms are always faster than `Ital` (by a factor of 10).

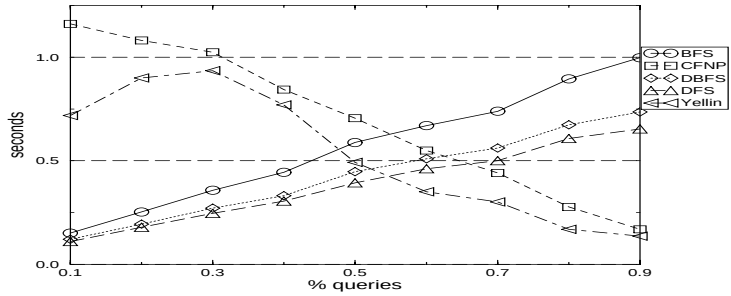


Fig. 7. 3.000 operations (`Boolean` queries, edge deletions and re-insertions) on the Internet graph.

Besides using random instances, we have carried out experiments on the graph modeling the Internet network. We downloaded the fragment of the network visible from one of the main European servers: *Reseaux IP Europeene* (*RIPE*), at the site `dbase.ripe.net`, whose interpretation is provided in [4]. We obtained a directed graph G with 1259 vertices (the *autonomous systems*), and 5104 edges. Since it was not possible to run `HK-2` and `Ital-Gen` on this big graph, and the dynamic algorithms `Ital`, `Yellin`, `CFNP` can be used in a fully dynamic setting only on DAGs, we performed the following simple mod-

ification to G: we inverted the direction of some edges of G obtaining a DAG G^0. The updates that we considered on G^0 were sequences of edge deletions and re-insertions, i.e., they don't change the topology of G^0. These experiments can be viewed as simulations of failures and recoveries of links, without perturbing the topology of G^0.

We compared the performances of `Ital`, `Yellin`, `CFNP` and the simple-minded algorithms on G^0. We observed no substantial difference in their behaviour by changing the length of the sequence, and hence we performed experiments with different percentages of queries in the sequences of updates (from 10% to 90%). These experiments may give useful suggestions on how to proceed in the presence of various patterns of link failures/recoveries. The results (see Figure 7) show the expected behaviour, but also provide a quantitative idea of what is the break point, i.e., after what percentage of queries the dynamic algorithms overcome the simple-minded algorithms which are better in the case of frequent updates. In this graphic we do not report on `Ital` because it is much slower than the other two dynamic algorithms (due to the `Reset` operation).

References

1. D. Alberts, G. Cattaneo, and G. F. Italiano. An empirical study of dynamic graph algorithms. In *ACM-SIAM Symposium on Discrete Algorithms*, pp. 192–201, 1996.
2. G. Amato, G. Cattaneo, and G. F. Italiano. Experimental analysis of dynamic minimum spanning tree algorithms. In *ACM-SIAM Symp. on Discrete Alg.*, 1997.
3. D. Alberts, G. Cattaneo, G.F. Italiano, U. Nanni, and C. Zaroliagis. A Software Library of Dynamic Graph Algorithms. In *Proc. Workshop on Algorithms and Experiments* – ALEX'98, 1998, pp.129-136.
4. T. Bates, E. Gerich, L. Joncheray, J-M. Jouanigot, D. Karrenberg, M. Terpstra, and J. Yu. Representation of IP routing policies in a routing registry. Technical report, RIPE-181, October 1994.
5. S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi dynamic problems in digraphs. *Theoretical Computer Science*, to appear.
6. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem. In *ACM-SIAM Symposium on Discrete Algorithms*, 1996, pp. 212–221.
7. M. R. Henzinger, and V. King. Fully dynamic biconnectivity and transitive closure. In *IEEE Symposium on Foundations of Computer Science*, 1995.
8. G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Th. Comp. Sc.*, 48:273–281, 1986.
9. G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *IPL*, 28:5–11, 1988.
10. J. A. La Poutré, and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Int. Work. on Graph-Theoretic Concepts in Comp. Sci.*, pages 106–120. Lect. Notes in Comp. Sci., 314, 1988.
11. K. Mehlhorn and S. Naher. LEDA, a platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.
12. G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.
13. D. M. Yellin. Speeding up dynamic transitive closure for bounded degree graphs. *Acta Informatica*, 30:369–384, 1993.

Matching Medical Students to Pairs of Hospitals: A New Variation on a Well-known Theme

Robert W. Irving

Computing Science Department, University of Glasgow, Glasgow, G12 8QQ, Scotland
rwi@dcs.gla.ac.uk

Abstract. The National Resident Matching Program in the U.S. provides one of the most successful applications of an algorithmic matching process. Under this scheme, graduating medical students are allocated to hospitals, taking into account the ordered preferences of both students and hospitals, so that the resulting matching is *stable* in a precise technical sense. Variants of this problem arise elsewhere. For example, in the United Kingdom, the students/hospitals problem is more complicated, primarily because students must take on two positions in their pre-registration year, namely a medical post and a surgical post. Hence there is a need for a matching process that assigns each student to one medical and one surgical unit, taking into account varying numbers of posts in the two half years, and students' seasonal preferences, as well as the basic preferences of students and hospital units for each other.
This paper describes an algorithmic solution to this new variation on an old theme that uses network flow and graph theoretic methods as well as the traditional stable matching algorithm. The approach described has been implemented and will soon come into use as a centralised matching scheme in Scotland.

1 Introduction

1.1 The background

The annual allocation of graduating medical students to hospitals, taking into account the preferences of both, is one of the best known practical applications of a matching algorithm. The most renowned scheme of this kind, the National Resident Matching Program (NRMP) provides a centrally administered national system responsible for such an allocation in the United States, and, since 1951, this organisation has employed an algorithm that is guaranteed to produce a *stable* matching (see below). It has been convincingly argued by Roth [8] that it is this stability property that has ensured the success of the NRMP scheme and others like it (such as its Canadian equivalent, CaRMS). In [8], Roth gives a graphic account of how lack of stability in such a matching is likely to lead to breakdown in the system.

In schemes such as the NRMP, each student (or resident) provides a list of hospitals, typically of some specified fixed length, in strict preference order, and

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 381{392, 1998.
c Springer-Verlag Berlin Heidelberg 1998

each hospital, which will have a specified number of available posts, likewise ranks the students to whom it would like to offer posts. A *matching* is an allocation of each student to at most one hospital so that no hospital has more students allocated than it has posts. A student S and hospital H are said to *block* the matching, or form a *blocking pair*, if the following three conditions are met:

- H appears on S's list, and S on that of H
- S is either unmatched or prefers H to his/her allocated hospital
- H either has an unfilled post or prefers S to at least one of its allocated students.

The existence of a blocking pair $(S; H)$ means either that the matching can be simply extended (by allocating S to H) or that S and H can undermine the matching to their mutual benefit. Hence the existence of a blocking pair implies instability, and a matching is therefore defined to be *stable* if and only if it admits no blocking pair.

The algorithm employed by the NRMP is essentially an extended version of what, (for the case of one-to-one matching known as the *Stable Marriage problem*), has come to be known as the Gale-Shapley algorithm [2] . This latter problem and its many variants and ramifications have been the subject of much study, see for example [4], [5], [11]. One of the key contributions of [2] was to show that, for both the one-to-one and many-to-one stable matching problems, every instance is bound to admit at least one stable matching. Of course, in the student/hospital context, where preference lists are incomplete, and the total number of students is unlikely to equal the total number of posts, such matchings are almost certain to be *partial*, in the sense that not all students will be matched or not all posts filled, or, most likely, both of these. As a result, in reality, some more or less ad hoc arrangements, or possibly a second phase of the whole matching process, may be necessary to complete the allocation. As an indication of what happens in practice, however, the main NRMP algorithm consistently matches over 95% of the students taking part.

1.2 Hospital and student optimal solutions

One feature of stable matching problems such as these is that the set of all stable solutions forms a distributive lattice under a natural dominance partial order relation (see [4] for details). One extreme, say the top of the lattice, is a stable matching that is simultaneously optimal for all of the members of one side of the market, say the hospitals, while the other extreme is a stable matching that is simultaneously optimal for all of the students. These are referred to as the *hospital-optimal* and *student-optimal* matchings, respectively. By optimal, we mean that each student or hospital has the most favourable allocation that is possible in any stable matching. However, it turns out that what is optimal for one side is worst possible for the other.

The extended version of the Gale/Shapley algorithm may be applied in one of two ways — from the hospitals' side or from the students' side. In the former

case, it yields the hospital optimal solution, and in the latter case the student optimal solution. A controversial feature of the NRMP scheme is that, from the outset, it used the hospital optimal solution as the basis for the allocation. As the implications of this policy for the students gained increasing recognition, pressure gradually built up for a change in policy [12] that has finally recently been agreed [7]. So in future, the algorithm will be applied from the students' side, thereby ensuring that the student optimal solution will be found and used as the basis of the allocation.

2 A New Variation on the Theme

2.1 The UK market

In the UK, the market for graduating medical students (known as pre-house registration officers, or PRHOs) is organised on a regional basis, and Roth [9], [10] reports on some of the experiences with automated matching processes. These have met with only limited success, partly because of their failure, in some cases, to generate stable matchings, and partly because of the different constraints that apply in the British context.

One key difference in the current UK system is that each student must undertake two appointments in the year, one medical and one surgical, either one of which may precede the other. So the hospitals can be viewed as two disjoint sets, the medical units and the surgical units, each of which has a specified number of posts for each half year. A given unit need not have the same number of posts for the two half years, though in practice this is often the case.

Current practice in the West of Scotland region is somewhat typical of that in many other regions. It is the responsibility of each individual student to negotiate directly with units in order to obtain offers of appointment — a situation recognised by many of those participating as unsatisfactory, and just the sort of market that had led to such chaos in the US before the introduction of the NRMP [8]. Hence, recognition has grown over many years of the need to devise an effective centralised matching scheme to replace the existing 'free-for-all'. The following further points complicate the allocation problem.

- A (typically small) number of applicants seek only one appointment — either the medical or the surgical — within the region, and the other appointment elsewhere; these are referred to as single-post students.
- There is strong pressure to allow students to express a preference as to which of their appointments they should take up first, medical before surgical or surgical before medical; this will be referred to as a *seasonal preference*. In the case of single-post students, the seasonal preference indicates in which half year the student seeks a post, and because such students generally seek the other post in a different region, it may be appropriate to view this as a requirement rather than a preference, or at least to give seasonal preferences for these students the highest priority. We will return to this issue later.

So, in this context, the requirement is for a matching scheme specified as follows:

The inputs are:

- The list of students, and for each one,
  whether s/he is seeking a medical post, a surgical post, or both;
  a preference list of medical units, or a preference list of surgical units, or both of these, as appropriate;
  an optional seasonal preference.
- The lists of medical units and surgical units, and for each one,
  the number of posts available in each half year;
  a single preference list of the students to which it wishes to offer an appointment.

The outputs are:

- A stable matching of medical candidates to medical units, based on the number of posts in each unit summed over the two half years.
- A stable matching of surgical candidates to surgical units, again based on the total number of posts in each unit.
- An allocation of each matched (student, unit) pair to a half year period so that
  1. each student who is matched to two posts has them scheduled in different half year periods;
  2. for each unit, and for each half year, the number of allocated students does not exceed the number of posts for that half year;
  3. the number of satisfied seasonal preferences is as large as possible.

An allocation satisfying conditions (1) and (2) will be called *valid*. Note that, in general, there is no guarantee that a valid solution exists; for example, suppose that a medical unit M and a surgical unit S each have one post in the first half year and no posts in the second half year, and that a particular student is matched with the sole post at S and the sole post at M.

As regards condition (3), it is easy to see that, in general, it will not be possible to satisfy all seasonal preferences; as an extreme example, imagine that every student wished to take up the medical post before the surgical one.

Note that, in practice, the length of a student's preference list would be specified by the scheme's administrators; this value does not affect the algorithmic issues (though it may well affect the sizes of the matchings).

2.2 The contribution of this paper

This paper describes the algorithm that is in process of implementation [6] to provide a centralised matching system for students and hospitals in Scotland. The starting point was the overriding consideration that the matching of students to medical units and the (separate) matching of students to surgical units must be stable, and to conform with current practice elsewhere, it was decided as a

matter of policy that the student-optimal stable solution should be used in each case (though this does not affect the rest of the allocation process described here). These matchings are found by an application of the (suitably extended) Gale-Shapley algorithm — see [4] for details. The matching produced for medical units (and likewise the matching for surgical units) then has the following properties:

- each student seeking a medical appointment is assigned to at most one medical unit
- each medical unit is assigned a number of students not exceeding the combined number of posts that it has available over the two half year periods
- the matching is stable.

The task that remains is to allocate each (student, unit) pairing to a half year period so as to produce a valid allocation (if one exists), and furthermore one that maximises the number of satisfied seasonal preferences.

The first question that arises is whether a valid allocation is possible in any particular case. It is not hard to show that a sufficient condition for the existence of a valid allocation is that each unit has an equal number of posts across the two half year periods (a condition that is likely to be close to being true in practice). The network-flow algorithm described in Section 3 will determine whether a valid allocation is possible, and when it is, it will find one such allocation. An iterative process, which uses a variant of the classical Bellman-Ford algorithm (see, for example, [1]) to detect negative-weight cycles in a weighted graph, is then applied to generate a sequence of valid allocations, culminating in one that is optimal in the sense that it maximises the number of satisfied seasonal preferences. Section 4 describes this phase of the algorithm. Finally, Section 5 contains some concluding remarks and mentions some open problems.

3 A Network Flow Approach

3.1 Terminology and notation

A hospital unit is either *medical* or *surgical*. Let the set of medical units be $M = \{M_1, M_2, \ldots, M_m\}$ and the set of surgical units $S = \{S_1, S_2, \ldots, S_s\}$.

Suppose that the medical unit M_i has x_i^1 posts in the first half year, and x_i^2 posts in the second half year ($1 \leq i \leq m$). Likewise, suppose that the surgical unit S_j has y_j^1 posts in the first half year and y_j^2 posts in the second half year ($1 \leq j \leq s$).

Denote the set of students (also known as PRHO's) by $P = \{P_1, \ldots, P_n\}$. For the moment, we ignore seasonal preferences, but we note that, in general, P contains two-post students, one-post students seeking only a medical post, and one-post students seeking only a surgical post.

We now describe how network flow methods may be used to find a valid allocation (or to determine that no such allocation exists). The algorithm constructs a network N to represent the assignment of students to units, finds a particular maximum flow from source to sink in this network, and interprets the flow in such a way as to decide the allocation of each student in P.

Input to the algorithm

(i) The sets M and S, for each $M_i \in M$ the integers x_i^1 and x_i^2, and for each $S_j \in S$ the integers y_j^1 and y_j^2.
(ii) The set P of students.
(iii) A stable matching of a subset P_M of the students in P to medical units, with the number of students matched to each unit bounded by the total number of available posts in that unit. P_M will exclude any surgical-only students (and perhaps some others who failed to be matched to a medical unit).
(iv) A stable matching of a subset P_S of the students in P to surgical units, again with the number of students matched to each unit bounded by the total number of available posts in that unit. P_S will exclude any medical-only students (and perhaps some others who failed to be matched to a surgical unit).

Note that any student in $P \setminus (P_M \cup P_S)$, i.e., a student who is matched in neither of the stable matchings, plays no part in the allocation algorithm, so may be omitted from further consideration; we assume that P is redefined to exclude such students. (In practice, such students, together with two-post candidates matched to just one post, must be handled by the 'second phase' of the matching scheme alluded to in Section 1.1.)

Output from the algorithm

For each student in P who is assigned to medical unit M_i, the value 1 or 2, indicating that the post is to be taken up in the first or second half year, respectively. Likewise, for each student in P who is assigned to surgical unit S_j, the value 1 or 2, with the same interpretation.

The allocation will be valid, i.e., it will be such that
(i) for each i $(1 \leq i \leq m)$, the total number of students assigned to unit M_i for the two half years is at most x_i^1 and x_i^2 respectively;
(ii) for each j $(1 \leq j \leq s)$, the total number of students assigned to unit S_j for the two half years is at most y_j^1 and y_j^2 respectively; and
(iii) a student assigned to both a medical unit and a surgical unit will have the two posts allocated to different half year periods.

In the case where no valid allocation exists, the sole output will be a message to this effect.

3.2 The Algorithm

For convenience, we introduce special 'dummy' medical and surgical units, denoted by M_0 and S_0 respectively. In order that all students can be treated in a uniform way, the algorithm will assume that a student assigned to just one post is assigned to the dummy unit of the other speciality. Throughout, all sums over i are from 0 to m, and all sums over j are from 0 to s.

Step 1: Create a source node A, a sink node Z, and nodes M_i and S_j for each medical and surgical unit $(0 \leq i \leq m;\ 0 \leq j \leq s)$.

Step 2: For each student $P_k \in P$, assigned to medical unit M_i and surgical unit S_j, include an edge $(M_i; S_j)$ of capacity 1. Note that there may be more than one student assigned to the same medical and surgical units, so the network is, in general, a multigraph.

At this point, denote by x_i the out-degree of node M_i $(0 \le i \le m)$, and by y_j the in-degree of node S_j $(0 \le j \le s)$. So x_i is the total number of students assigned to M_i, y_j is the total number assigned to S_j, and $\sum x_i = \sum y_j = n = |P|$. Furthermore, let $x_0^1 = x_0^2 = x_0$ and $y_0^1 = y_0^2 = y_0$.

Step 3: Include an edge $(A; M_i)$ of capacity x_i^1 $(0 \le i \le m)$, and an edge $(S_j; Z)$ of capacity y_j^2 $(0 \le j \le s)$.

Step 4: Add a special node M, and include an edge $(M; S_j)$ of capacity $y_j^1 + y_j^2 - y_j$ for each j $(0 \le j \le s)$. Add a second special node S, and include an edge $(M_i; S)$ of capacity $x_i^1 + x_i^2 - x_i$ for each i $(0 \le i \le m)$. Include an edge $(A; M)$ of capacity c_M, an edge $(S; Z)$ of capacity c_S, and an edge $(M; S)$ of capacity $c_{M;S}$, where

$$c_M = \sum y_j^2 + \min(\sum x_i^2; \sum y_j^1) - n;$$
$$c_S = \sum x_i^1 + \min(\sum x_i^2; \sum y_j^1) - n;$$

and

$$c_{M;S} = \min(\sum x_i^2; \sum y_j^1) + \min(\sum x_i^1; \sum y_j^2) - n.$$

It is easy to verify that all of the capacities in the resulting network are non-negative integers.

Theorem 1 below justifies the construction of this network.

Theorem 1 *Let F be a maximum (integer valued) flow in the network N constructed by the above algorithm.*
(i) If the value of F is

$$f = \sum x_i^1 + \sum y_j^2 + \min(\sum x_i^2; \sum y_j^1) - n$$

then assigning students represented by an edge with flow 1 to be MS and those represented by an edge with flow 0 to be SM gives a valid assignment.
(ii) If the value of F is $< f$ then no valid assignment exists.

Proof (i) With such a flow, every edge incident from A and every edge incident to Z is saturated. Consider a fixed node M_i $(1 \le i \le m)$. The flow into M_i is x_i^1, so the flow out of that node is also x_i^1. Hence at most x_i^1 edges $(M_i; S_j)$ $(0 \le j \le s)$ have a flow of 1, and as a consequence, at most x_i^1 of the students assigned to M_i are MS-students. Hence the number of students assigned to M_i for the first half year does not exceed the number of posts. Similarly, the flow out of any fixed node S_j $(1 \le j \le s)$ is y_j^2, so the flow into that node is also y_j^2. Hence at most y_j^2 edges $(M_i; S_j)$ $(0 \le i \le m)$ have a flow of 1, and as a consequence, at most y_j^2 of the PRHOs assigned to S_j are MS-students. Hence

the number of PRHOs assigned to S_j for the second half year does not exceed the number of posts.

The flow in the edge $(M_i; S)$ cannot exceed $x_i^1 + x_i^2 - x_i$. Hence the flow in the edges $(M_i; S_j)$ $(0 \quad j \quad s)$ must be at least $x_i^1 - x_i^1 - x_i^2 + x_i = x_i - x_i^2$. So the number of such edges $(M_i; S_j)$ in which the flow is zero is at most $x_i - (x_i - x_i^2) = x_i^2$, and as a consequence the number of students allocated to M_i for the second half year cannot exceed the number of available posts. Likewise, the flow in the edge $(M; S_j)$ cannot exceed $y_j^1 + y_j^2 - y_i$. Hence the flow in the edges $(M_i; S_j)$ $(1 \quad i \quad m)$ must be at least $y_j^2 - y_j^2 - y_j^1 + y_j = y_j - y_j^1$. So the number of such edges $(M_i; S_j)$ in which the flow is zero is at most $y_j - (y_j - y_j^1) = y_j^1$, and as a consequence the number of students allocated to S_j for the first half year cannot exceed the number of available posts.

(ii) We show this by demonstrating that, from a valid assignment of students, a flow of value F from A to Z in the network can be constructed. We first assign flows to saturate all of the edges incident from A and all of the edges incident to Z. If we can now assign flows to the edges $(M_i; S_j)$, $(M_i; S)$, $(M; S_j)$, and $(M; S)$ so that Kirchhoff's Law is satisfied at every vertex then we will have a flow of value F as required.

For each given student, represented by edge $(M_i; S_j)$, assign a flow of 1 or 0 to that edge according as the student is SM or MS in the valid assignment. Suppose that the numbers of MS and SM students allocated to M_i in this assignment are a_i and b_i respectively, so that $a_i \quad x_i^1$, $b_i \quad x_i^2$, and $a_i + b_i = x_i$. The flow into node M_i has value x_i^1, so to satisfy Kirchhoff's Law we need an outward flow of x_i^1 from that node. The edges $(M_i; S_j)$ $(0 \quad j \quad s)$ contribute a_i, so a further $x_i^1 - a_i$ is needed. But the capacity of edge $(M_i; S)$ is $x_i^1 + x_i^2 - x_i \quad x_i^1 + b_i - x_i = x_i^1 - a_i$, so we can allocate the required flow to that edge. This can be repeated for all the nodes M_i, and in a wholly analogous way for all the node S_j in order to satisfy Kirchhoff's Law at those nodes also. So it remains to consider nodes M and S — we deal explicitly with node S, node M being analogous.

The flow into node S along edges $(M_i; S)$ is $\sum (x_i^1 - a_i)$. The flow out along edges $(S; Z)$ is $\sum x_i^1 + \min(\sum x_i^2; \sum y_j^1) - n$. Now $\sum a_i = \sum x_i - \sum b_i = n - \sum b_i$, and $\sum b_i$ is the total number of SM-students in the valid assignment, so

$$\sum b_i \quad \min(\sum x_i^2; \sum y_j^1):$$

Hence

$$\sum a_i \quad n - \min(\sum x_i^2; \sum y_j^1);$$

and so

$$\sum (x_i^1 - a_i) = \sum x_i^1 - \sum a_i \quad \sum x_i^1 + \min(\sum x_i^2; \sum y_j^1) - n:$$

It follows that the flow into node S along edges $(M_i; S)$ cannot exceed the flow out of node S, so, provided the capacity of edge $(M; S)$ is sufficient, we can satisfy Kirchhoff's Law at node S by assigning a suitable flow to edge $(M; S)$. The required value is

$$\min(\sum x_i^2; \sum y_j^1) - n + \sum a_i:$$

(The required value arises from consideration of the flow through node M.) So provided the capacity of edge $(M; S)$ is sufficient to cope with this flow when a_i takes its maximum value, the proof is complete. But

$$\sum_{a_i} \min(\sum x_i^1; \sum y_j^2)$$

so the result follows from the choice of value of $c_{M;S}$. ■

Hence we have an efficient algorithm to find a valid allocation, or to establish that none exists. For a precise measure of efficiency, the Goldberg/Tarjan network flow algorithm [3], for example, has complexity $O(jVjjEj \log(jVj^2/jEj))$, where V is the vertex-set and E the edge-set of the graph. Here, $jVj = O(m+s)$ and $jEj = O(n)$, so that our algorithm is $O((m+s)n \log((m+s)^2/n))$ in the worst case.

4 Maximising the Number of Satisfied Seasonal Preferences

4.1 The problem

A solution arising from an arbitrary maximum flow in the network N takes no account of any expressed seasonal preferences. If we wished to ensure that all such preferences were met, some of the flows would have to be pre-assigned. To be precise, the flow in an edge $(M_i; S_j)$ representing student P_k must have value 1 if P_k has an MS-preference and value 0 if P_k has an SM-preference. (Note that a medical-post only student has an MS-preference if s/he wishes to take up the post in the first half year, etc.) This can be ensured by removing all edges corresponding to students with seasonal preferences from the network and adjusting certain other edge capacities accordingly. Alternatively, make the necessary pre-assignments, calculate afresh the appropriate values of x_i^1, x_i^2, y_j^1, and y_j^2, and start the construction of the network from Step 1.

However, this may result in edges with negative capacity, and even if this does not happen, there may well be no maximum flow with the required value, In other words, it may be impossible to find a valid assignment in which the seasonal preference of every student is satisfied. Nonetheless, seasonal preferences have a special significance for medical-only or surgical-only students, so it may be appropriate to pre-assign the flow in the edges incident to M_0 and S_0. In practice, the number of such students is very small, and it is unlikely that this small number of pre-assignments will affect the existence of a valid assignment.

Whether or not such pre-assignments are made for this small group of students, the second phase of our allocation process will repeatedly amend the initial maximum flow in order to increase the number of seasonal preferences that are satisfied, until we reach a maximum flow that corresponds to a valid assignment that is best possible in this respect

4.2 A solution based on negative length cycles in a graph

A valid allocation corresponds to a maximum flow in the network N, so suppose we have found such a maximum flow F. We focus on the edges from M_i to S_j ($0 \quad i \quad m; 0 \quad j \quad s$), and the edges incident from the special node M and to the special node S. So that we can treat these edges in a uniform way, suppose that each edge incident from M, of capacity c say, is replaced by c (parallel) edges of capacity 1. A flow of g in such an edge is mirrored by a flow of 1 in g of the parallel edges and a flow of 0 in the remainder. Likewise, replace all the edges incident to S by parallel edges of capacity 1.

As observed earlier, a flow of 1 in the edge $(M_i; S_j)$ corresponds to an MS-assignment for the student represented by that edge, and a flow of 0 represents an SM-assignment. If a student expressed an MS-preference then we say that the *target* for the corresponding edge is 1, an SM-preference corresponds to a target of 0, and if no preference was expressed then the target is undefined. We specify that the target is also undefined for all of the edges incident from M and all of the edges incident to S.

We are going to use shortest path methods to solve this problem, so we now define the length of each edge in this model. An edge has length 0 if the target for that edge is undefined, length $+1$ if the target is equal to the flow, and length -1 if the target is defined but not equal to the flow. In what follows, our attention is restricted to the subgraph of N that excludes nodes A and Z and their incident edges.

We redirect the edges so that an edge with flow 1 is directed from M or M_i to S or S_j, and an edge of flow 0 is directed from S or S_j to M or M_i. So any path in this newly directed graph has edges that alternate in flow between 1 and 0. Call this new network N_F (reflecting the fact that it depends on the flow F).

Theorem 2 *If C is a negative length cycle in N_F then switching the flow along each edge of C (0 to 1 and vice-versa) gives another maximum flow in the original network N, and in the resulting assignment, the number of students who have their seasonal preference satis ed is increased.*

Proof The length of a cycle is the number of edges in which flow = target minus the number in which flow $\neq$ target. Hence switching the flow in each edge increases the number in which flow = target. Also, in N, Kirchhoff's laws continue to hold, and so the result is still a valid flow with the same value as F. ∎

Define the *penalty* of a flow to be the number of contravened seasonal preferences in the corresponding assignment. In terms of N_F, the penalty is the number of edges in which flow $\neq$ target.

Theorem 3 *A necessary and su cient condition that a particular maximum flow F has minimum penalty is that the corresponding network N_F should contain no negative length cycle.*

Proof The necessity of the condition follows at once from the previous theorem.

For sufficiency, we show that if F does not have minimum penalty then there must exist a cycle in N_F of negative length. Let G be a maximum flow (in N) of minimum penalty, and consider the graph H comprising the edges that have different flows in F and G together with the incident vertices . Each vertex in H must be balanced, in the sense that the number of incident edges with a flow of 1 in F and 0 in G is equal to the number of incident edges with a flow of 0 in F and a flow of 1 in G. So H can be partitioned into cycles whose edges alternate between these two types — and these are just cycles in N_F. In order that the penalty of G should be less than the penalty of F, it is essential that the length of at least one of these cycles be negative, since the sum of their lengths is equal to the penalty of G minus the penalty of F. ■

It follows that a maximum flow of minimum penalty, corresponding to an assignment with the maximum number of satisfied seasonal preferences, can be found from an initial maximum flow F by a sequence of iterations. In each iteration, we seek a negative length cycle in N_F (using, say a variant of the Bellman-Ford algorithm [1]); if no such cycle exists we are done, otherwise we switch the flows on the edges of the cycle to get a new better maximum flow, which becomes F for the next iteration.

As far as the complexity of this part of the algorithm is concerned, a naive application of the extended Bellman-Ford algorithm leads to $O(|V||E|) = O((m + s)n)$ complexity for each iteration. Each iteration increases the number of satisfied seasonal preferences, so that the number of iterations cannot exceed n in the worst case. So the overall worst case is certainly $O((m + s)n^2)$. It may well be possible to choose negative length cycles in such a way as to improve this.

5 Conclusion

We have described an efficient and practical algorithm for the allocation of graduating medical students to pairs of posts corresponding to the medical education requirements in the UK. The resulting allocations to medical and surgical units are separately stable, the bounds on numbers of posts in each half year are satisfied for all units, and, subject to these constraints, a maximum number of students have their seasonal preference satisfied.

Preliminary computational experience using both random and realistic test data is impressive. The programs have been written in C++ and implemented on a PC, and allocations involving several hundred students are completed in a few seconds. At the time of writing, it is planned that a centralised matching scheme using the algorithm described will be operational in Scotland from the year 1999-2000.

Some open issues remain. For example, seasonal preference is a more important consideration for single-post students, and it would be useful to find a variant of the Bellman-Ford approach that would give priority to such students

when seasonal preferences are considered. The alternative that is currently implemented pre-allocates such students to the appropriate half year before the network-flow phase of the algorithm, and because their number is small in practice, it is very unlikely that this pre-allocation, in itself, will bar the existence of a valid allocation.

A second open question concerns the situation where no valid allocation exists. Although this is unlikely to arise in practice, the algorithm can easily be adapted to find a partial allocation that respects the bounds on numbers of posts, and leaves a minimum number of matched students unallocated. An alternative would be to investigate whether the use of stable matchings other than the student-optimal in each case might permit a valid allocation to be found.

References

1. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.
2. D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.
3. A. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. In *Proceedings of the 18th ACM Symposium on the Theory of Computing*, pages 136–146, 1986.
4. D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, Boston, MA, 1989.
5. D.E. Knuth. *Stable Marriage and its Relation to Other Combinatorial Problems*. American Mathematical Society, Providence, RI, 1996.
6. G.M. Low. Pre-register post matching in the West of Scotland. Master's thesis, University of Glasgow, 1997.
7. M. Mukerjee. Medical mismatch. *Scienti c American*, 276(6):40–41, June 1997.
8. A.E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92:991–1016, 1984.
9. A.E. Roth. New physicians: a natural experiment in market organization. *Science*, 250:1524–1528, 1990.
10. A.E. Roth. A natural experiment in the organization of entry-level labor markets: regional markets for new physicians and surgeons in the United Kingdom. *The American Economic Review*, 81:415–440, 1991.
11. A.E. Roth and M. Sotomayor. *Two Sided Matching: A Study in Game-Theoretic Modelling and Analysis*. Cambridge University Press, 1991.
12. K.J. Williams. A reexamination of the NRMP matching algorithm. *Academic Medicine*, 70:470–476, 1995.

-Stepping : A Parallel Single Source Shortest Path Algorithm

Ulrich Meyer and Peter Sanders

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

Abstract. In spite of intensive research, little progress has been made towards fast and work-efficient parallel algorithms for the single source shortest path problem. Our -*stepping algorithm*, a generalization of Dial's algorithm and the Bellman-Ford algorithm, improves this situation at least in the following "average-case" sense: For random directed graphs with edge probability $\frac{d}{n}$ and uniformly distributed edge weights a PRAM version works in expected time $O(\log^3 n / \log\log n)$ using linear work. The algorithm also allows for efficient adaptation to distributed memory machines. Implementations show that our approach works on real machines. As a side effect, we get a simple linear time sequential algorithm for a large class of not necessarily random directed graphs with random edge weights.

1 Introduction

The shortest path problem is a fundamental and well-studied combinatorial optimization problem with many practical and theoretical applications [2]. Let $G = (V; E)$ be a directed graph, $jEj = m$, $jV j = n$, let s be a distinguished vertex of the graph, and c be a function assigning a non-negative real-valued *weight* to each edge of G. The *single source shortest path problem* (SSSP) is that of computing, for each vertex v reachable from s, the weight of a minimum-weight path from s to v; the weight of a path is the sum of the weights of its edges.

The theoretically most efficient sequential algorithm on directed graphs with non-negative edge weights is Dijkstra's algorithm [11]. Using Fibonacci heaps its running time is bounded by $O(n \log n + m)$ [1]. Dijkstra's algorithm is inherently sequential since its work efficiency depends on the fact that nodes are considered in a fixed priority order. On the other hand, the Bellmann-Ford algorithm allows to consider all nodes in parallel but for that very reason it is not work efficient.

Therefore, we propose the following generalization named -stepping: Nodes are ordered using *buckets* representing priority ranges of size . Each bucket may

[1] There is also an $O(n + m)$ time RAM algorithm for undirected graphs [25] which requires $n > 2^{12^{20}}$ due to the usage of atomic heaps however.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 393{404, 1998.
c Springer-Verlag Berlin Heidelberg 1998

be processed in parallel. This can be implemented using only linear work if is not too small. Refer to Sect. 2 for details.

From now on, with *random edge weights* we mean uniformly distributed weights in $[0;1]$ and random graphs are chosen from the set $G(n; \frac{d}{n})$, i.e. have n nodes and edge probability $\frac{d}{n}$. In Sect. 3 we show that choosing $= \frac{1}{d}$ will not significantly increase the work performed and that only $O(\log^2 n / \log\log n)$ buckets have to be emptied to solve the SSSP on random graphs. Then, in sections 4 and 5, -stepping is adapted to the arbitrary CRCW-PRAM[2] model and to distributed memory machines respectively such that the work remains linear and time $O(\log^3 n / \log\log n)$ suffices for random graphs. Sect. 6 complements the theoretical analysis with simulation and implementation results. Finally, Sect. 7 summarizes the most important aspects and sketches some possible future improvements.

Previous Work

The SSSP problem has so far resisted fast efficient parallel solutions: Most previous work was done in the PRAM model. The *work* of a parallel algorithm is given by the product of its running time and the number of processing units (PUs) p. There is no parallel $O(n \log n + m)$ work PRAM algorithm with sublinear running time for general digraphs with non-negative edge weights. The best $O(n \log n + m)$ work solution (refining [22] with [13]) has running time $O(n \log n)$. All known algorithms with polylogarithmic execution time are work-inefficient. (The algorithm in [16] uses $O(\log^2 n)$ time and $O(n^3 (\log\log n / \log n)^{1/3})$ work.) An $O(n)$ time algorithm requiring $O((n+m)\log n)$ work was presented in [4].

For special classes of graphs, like planar digraphs [27] or graphs with separator decomposition [8], some less inefficient algorithms are known. Randomization was used in order to find approximate solutions with small error probability [19,7]. For random graphs only unit weight edges have been considered so far [6]: This solution is restricted to constant edge probabilities or edge probability $(\log^k n / n)$ $(k > 1)$. In the latter case $O(n \log^{k+1} n)$ work is needed.

Experimental studies for distributed computation of SSSP with multiple queues and some forms of oblivious parallel expansion strategies are provided in [5,1,26]. No profound theoretical analysis beyond correctness proofs are given, the experimental results yield only quite limited speedup.

2 The -Stepping Algorithm

Our algorithm can be viewed as a variant of Dijkstra's algorithm. Dijkstra's algorithm maintains a partition of V into *settled*, *queued* and *unreached* nodes and for each node v a *tentative distance* tent(v); tent(v) is always the weight of some path from s to v and hence an upper bound on dist(v), the weight of the shortest path from s to v. For unreached nodes, tent(v) $= \infty$. Initially, s is

[2] Concurrent read concurrent write parallel random access machine [18].

queued, tent(s) = 0, and all other nodes are unreached. In each iteration the queued node v with smallest tentative distance is selected and declared settled and all edges (v,w) are *relaxed*, i.e., tent(w) is set to $\min\{\text{tent}(w), \text{tent}(v) + c(v,w)\}$. If w was unreached, it is now queued. It is well known that tent(v) = dist(v) when v is selected from the queue.

```
foreach v ∈ V do                                          { { Initialize node data structures
    heavy(v) := {(v,w) ∈ E : c(v,w) > Δ}                  { { Find heavy edges
    light(v) := {(v,w) ∈ E : c(v,w) ≤ Δ}                  { { Find light edges
    tent(v) := ∞                                          { { Unreached
relax(s, 0); i := 0                                       { { Source node at distance 0
while ∃ nonempty bucket do                                { { Some queued nodes left
    S := ∅                                                { { No nodes deleted for this bucket yet
    while B[i] ≠ ∅ do                                     { { New phase
        Req := {(w, tent(v) + c(v,w)) : v ∈ B[i] ∧ (v,w) ∈ light(v)}
        S := S ∪ B[i]; B[i] := ∅                          { { Remember deleted nodes
        foreach (v,x) ∈ Req do relax(v, x)                { { This may reinsert nodes
    od
    Req := {(w, tent(v) + c(v,w)) : v ∈ S ∧ (v,w) ∈ heavy(v)}
    foreach (v,x) ∈ Req do relax(v, x)                    { { Relax previously deferred edges
    i := i + 1                                            { { Next bucket

Procedure relax(v, x)                                     { { Shorter path to v?
    if x < tent(v) then                                   { { Yes: decrease-key respectively insert
        B[⌊tent(v)/Δ⌋] := B[⌊tent(v)/Δ⌋] \ {v}             { { Remove if present
        B[⌊x/Δ⌋] := B[⌊x/Δ⌋] ∪ {v}                         { { Insert into new bucket
        tent(v) := x
```

Fig. 1. High level -stepping SSSP algorithm.

In the -stepping algorithm shown in Figure 1 we weaken the total ordering of the queue and only maintain an array B of *buckets* such that $B[i]$ stores $\{v \in V : v$ is queued and tent(v) $\in [i\Delta, (i+1)\Delta)\}$. In each *phase*, i.e., each iteration of the inner while-loop, we remove all nodes from the first nonempty bucket and relax all *light* edges ($c(e)$) which might lead to new nodes for the current bucket. For the remaining *heavy* edges it is sufficient to relax them once and for all when a bucket finally remains empty. Deletion and edge relaxation for an entire bucket can be done in parallel and in arbitrary order as long as an individual relaxation is atomic. If we do not want to sequentially execute the requests for a given node we can additionally exploit that requests which won't change the tent() array can be ignored.

For integer weights and = 1, -stepping coincides with Dial's implementation of Dijkstra's algorithm (e.g. [2, Sect. 4.6]). We can reuse the data structure used there. Buckets are implemented as doubly linked lists. Inserting or deleting a node, finding a bucket for a given tentative distance and skipping an empty bucket can be done in constant time. By cyclically reusing empty buckets we only need space for $\max_{e \in E} c(e)/$ buckets.

Our algorithm may remove nodes from the queue for which dist(v) < tent(v) and hence may have to reinsert those nodes until they are finally settled (dist(v) = tent(v)). In fact, for $\Delta = 1$ we get the Bellman-Ford algorithm. It has high parallelism since all edges can be relaxed in parallel but it may be quite inefficient compared to Dijkstra's algorithm. The idea behind Δ-stepping is to find an easily computable Δ which yields a good compromise between these two extremes. There are graphs for which there is no good compromise. But at least for random edge weights and in particular for random graphs with random edge weights we identify such a compromise in Sect. 3. In Sect. 7 we give additional evidence that similar algorithms can also yield useful parallelism for real world problems.

3 Analysis

Our analysis of the Δ-stepping algorithms proceeds in three stages in order to make the results adaptable to different graph classes. In Sect. 3.1 we analyze the number of phases needed and the number of reinsertions in terms of logical properties like the maximum path weight $d_c := \max\{\text{dist}(v) : \text{dist}(v) < 1\}$ which make no assumptions on the class of graphs investigated. Sect. 3.2 analyzes these conditions for the case of random edge weights. Finally, Sect. 3.3 completes the analysis by additionally assuming random graphs from $G(n, d/n)$.

3.1 Reinsertions and Progress

Let P denote the set of paths with weight at most Δ. P plays a key role in analyzing both the overhead and the progress of Δ-stepping. The overhead compared to Dijkstra's algorithm is due to *reinsertions* and *rerelaxations*, i.e., insertions of nodes which have previously been deleted and relaxation of their outgoing edges.

Lemma 1. *The total number of reinsertions is bounded by $|P|$ and the total number of rerelaxations is bounded by $|P_2|$.*

Proof. We give an injective mapping from the set of reinsertions into P. Consider a node v which is reinserted in phase i. There must be a most recent phase $i' < i$ when v was deleted. Consider a shortest path $(s, \ldots, v', \ldots, v)$ where v' is the first unsettled node on this path immediately before phase i'. In contrast to v, v' is settled by phase i', i.e., $v' \neq v$. Furthermore, $(v', \ldots, v) \in P$ since both v and v' are deleted by phase i'. Since v' is settled, the reinsertion of v in phase i can be uniquely mapped to $(v', \ldots, v)$.

Similarly, each rerelaxation can be uniquely identified by a reinsertion plus a light edge e, i.e., the path in P_2 defined by appending e to the path in P identifying the reinsertion can be injectively mapped to the rerelaxation. ∎

The number of phases needed can be bounded based on the step width Δ, the maximum distance d_c and a parameter $l_{\max}$ which must exceed the maximum number of edges in any path in P.

Lemma 2. *For any step width , the number of phases is bounded by* $\frac{d_c}{} l_{\max}$.

Proof. It suffices to show that no bucket is expanded more than $l_{\max}$ times. So assume the contrary, i.e., there is a bucket $B[i]$ which is expanded at time steps $t - l_{\max}, \ldots, t$. Let $v = \min\{\text{tent}(w) : w \in B[i](t)\}$. v is removed for the last time in phase t because otherwise it could not be the minimum queued element. Consider the path $P = (s, \ldots, v_{l_{\max}}, \ldots, v_1, v)$ in the shortest path tree leading from s to v. v_1 must be settled in phase $t - 1$. It cannot be settled later because then v would not have its final distance yet. v_1 cannot be settled earlier because else v would have received its final distance earlier and would also be settled by now since it is already in $B[i]$ for the last $l_{\max}$ phases.

Similarly, we can show by induction that $v_1, \ldots, v_{l_{\max}}$, have been settled one by one in the last $l_{\max}$ phases. So, $v_{l_{\max}}$ has been settled in phase $t - l_{\max}$ and this is only possible if its final distance puts it in bucket $B[i]$ and therefore $(v_{l_{\max}}, \ldots, v)$ is in P and has $l_{\max}$ edges. This is a contradiction. ∎

For a parallel algorithm the goal is to find a which is small enough to keep $|P|$ in $O(n)$ yet large enough to yield sufficient parallelism. For example, it is easy to see that there are no reinsertions if we choose the step-width to be the minimum edge weight and $r = \frac{d_c}{}$ phases suffice in this case [12]. We can even achieve the same result for larger by adding a "shortcut" edge (v, w) for each $(v, \ldots, w) \in P$. However, our experimental results on random graphs indicate that the basic algorithm also completes in $O(\frac{d_c}{})$ phases in practice so that we continue with the analysis of the simpler algorithm.

3.2 Random Edge Weights

This section is devoted to proving that $= (1/d)$ is a good compromise between work efficiency and parallelism for random edge weights:

Theorem 1. *Given a graph with maximum degree d or a random graph from $G(n, d/n)$. For random edge weights, a $(1/d)$-stepping scheme performs $O(dn)$ expected sequential work divided between $O(\frac{d_c}{} \frac{\log n}{\log\log n})$ phases whp*[3].

Our main tool is the fact that long paths with small weight are unlikely.

Lemma 3. *Given a path of length l. The probability that its total weight is bounded by 1 is $^{l}/l!$.*

Proof. By induction over l. ∎

This is the only part in our analysis which would have to be adapted for other than uniform weight distribution. Now we can bound $|P|$ and the $l_{\max}$ from Lemma 2.

Lemma 4. *For $= (1/d)$, $\mathrm{E}[|P|] = O(n)$, $\mathrm{E}[|P_2|] = O(n)$ and $l_{\max} = O(\log n / \log\log n)$ whp.*

[3] Throughout this paper "whp" stands for "with high probability" in the sense that the probability for some event is at least $1 - n^{-}$ for a constant > 0.

Proof. There can be at most d^l paths of length l leading into a given node v or nd^l such paths overall.(For random graphs there are n^l possible paths per node with a probability of $(d/n)^l$ each.) Using Lemma 3 we can conclude that the expected number of light paths is bounded by $\sum_{l=1} nd^l (d/n)^l/l! = n(e^d - 1) = O(n)$ and analogously for P_2.

Similarly, $\mathbf{P}[\exists P_2 : |P_2| = l] \leq n(d/n)^l/l!$. Therefore

$$\begin{aligned}\mathbf{P}[\exists P_2 : |P_2| \geq l_{\max}] &\leq \sum_{l \geq l_{\max}} n(d/n)^l/l! \leq n(d/n)^{l_{\max}}/l_{\max}! \sum_{l \geq 0} (d/n)^l/l! \\ &= n(d/n)^{l_{\max}} e^d/l_{\max}! = O(n)/l_{\max}! \text{ for } 1/d \\ &\leq (e/l_{\max})^{l_{\max}} \leq O(n) \text{ since } k! \geq (k/e)^k.\end{aligned}$$

Now it is easy to see that we can choose an $l_{\max} = O(\log n/\log\log n)$ such that the above probability is polynomially small. ∎

Theorem 1 is now an immediate consequence of lemmata 1, 2 and 4.

3.3 Random Graphs

So far, the analysis treated the maximum path weight, d_c, as a parameter and it is clear that there are graphs – even with random edge weights – where $d_c = \Omega(n)$ so that it makes no sense to expand nodes in parallel. But this is quite untypical. In particular, for random graphs d_c is rather small:

Theorem 2. *For random graphs from $G(n, d/n)$, $d_c = O\left(\frac{\log n}{d}\right)$ whp.*

For large degree – $d \geq a \log n$ for some constant a – this is a well known result [17,14]. We now outline a proof for the remaining case $d = O(\log n)$ and refer to [10] for more details.

From random graph theory we know that if s is not in the giant component it is in a very small component of size $O(\log n)$ and the SSSP is very simple. Otherwise d_c can be bounded by the diameter of the giant component.[4]

Lemma 5. *For $d > 1$ the diameter of the giant component is $O(\log n)$ whp.*

Proof. (Outline) The node exploration procedure used in [3, Sect. 10] can be adapted to a breadth first traversal. Using this approach it can be shown that all but $O\left(n^{1/2+\varepsilon}\right)$ nodes in the giant component are reached after $O(\log n)$ phases whp. Later, the expected number of newly reached nodes in the breadth first traversal decreases geometrically so that after $O(\log n)$ more steps no more nodes are reached whp. ∎

[4] Note that this diameter is *not* the diameter of the graph (which may be infinite in this "sparse" case). So, the fact that the diameter of random graphs is well studied in the literature is not directly helpful.

This yields a good upper bound for constant d. For larger degree, the basic idea is to first consider only those edges with weight at most $2/d$ (any a/n with $a > 1$ will do). Those form a "backbone" of the giant component for which $d_c = O(\frac{\log n}{d})$. Finally, the exploration procedure from [3, Sect. 10] can be adapted once more to show that every node in the giant component connects to the backbone by a path of length $O(d\frac{\log n}{d}e)$ whp. ∎

Substituting this result into Theorem 1 we see that $r = O(\log^2 n/\log\log n)$ phases of a $(1/d)$-stepping algorithm suffice to solve the SSSP. If we have introduced shortcut edges this reduces to $(\log n)$ phases.

4 Adaptation to PRAM

We now explain how the abstract $(1/d)$-stepping algorithm from Fig. 1 can be efficiently implemented on an arbitrary-write CRCW PRAM for random graphs from $G(n; \frac{d}{n})$ with random edge weights. The actual number of edges is $m = (dn)$ whp. We concentrate on the most interesting case $d = O(\log n)$. (Note that whp all but the $c\log n$ smallest edges per node can be ignored without changing the shortest paths for some constant c [17,14].) A direct way for handling larger m is discussed in [10].

Preparations: The nodes are assigned to random PUs by generating an array "ind" of random PU indices. The adjacency lists are reorganized into arrays of heavy and light edges for each node. Each of the p PUs maintains its own bucket structure and stores there the queued nodes it is responsible for. These operations can be done in time $O(dn/p + \log(dn))$ and take $O(dn)$ space.

Loop Control: Detecting when one or all buckets are globally empty and advancing i is easy to do in time $O(\log n)$ (or even $O(1)$) per iteration.

Maintaining S: S can be represented as a simple list per PU. Inserting a node several times can be avoided by storing a flag with each node which is set when it is inserted for the first time.

Generating Requests: Since the nodes have been randomly assigned, no PU has to delete more than $O(|B[i]|/p + \log n)$ nodes from its local part of $B[i]$ whp. Using prefix sums, the light edges to be scanned can be evenly distributed between the PUs in time $O(\log n)$. The request set generated is represented as a global array of target-distance pairs. Analogously, requests for heavy edges can be generated in a load balanced way with a control overhead of $O(\log n)$ time steps whp.

Assigning Heavy Requests to PUs: PU i maintains a request buffer which must be a constant factor larger than needed to accommodate the requests $(w; x)$ with $\text{ind}(w) = i$. Since heavy edges are relaxed only once, for random graphs their targets are independently distributed and therefore by Chernoff bounds, a buffer area of size $O(|S|/p + \log n)$ sufficies whp. The requests can be placed into the appropriate buffers using randomized dart throwing in time $O(|B[i]|/p + \log n)$ whp [21]. (For the unlikely case that a buffer is to small correctness can be preserved by checking periodically whether the dart throwing has terminated and increasing the buffer sizes if necessary.)

Assigning Light Requests to PUs: Assigning light request works as for heavy requests. The targets of rerelaxed edges are no longer independent. However, targets are still independent when edges are relaxed for the first time. Let $K_i := |\{f(v,w) \in E : \text{dist}(v) \in [i\Delta,(i+1)\Delta) \wedge c((v,w)) \le \Delta\}|$, i.e., the number of light edges ever relaxed in bucket i not counting rerelaxations. Then, by Chernoff bounds, no node receives more than $O(dK_i \log n/n e)$ requests in any phase for bucket i. Let K^0_{ij} denote the number of requests sent in the j-th phase for bucket i. Since nodes are placed independent of the computation, we can use the weighted Chernoff bound from [23], to see that no PU receives more than $O\left(K^0_{ij}/p + \log n + \sqrt{dK_i \log n/n e}\right)$ requests in phase j for bucket i whp. By Lemma 4 we have $\mathbf{E}[\sum_{ij} K^0_{ij}] = O(n)$. Furthermore, no bucket is emptied more than $l_{\max} = O(\log n/\log\log n)$ times so that $\sum_{ij} K_i = O(n l_{\max})$ whp. Therefore, the expected request contention summed over all phases is bounded by $O\left(n/p + l_{\max}\log^2 n\right)$.

Performing Relaxations: Each PUs scans its request buffer and sequentially performs the relaxations assigned to it. Since no other PUs work on its nodes the relaxations will be atomic. (This is the only place where significant modifications for handling the case $d \gg \log n$ would be needed if one chooses not to filter out too heavy edges in a preprocessing: p/n PUs work together on a single node and first find the minimum distance request and only perform a relaxation for this request.)

To summarize, control overhead accounts for at most $O(\log n)$ time per phase; the expected load imbalance accounts for at most $O(\log n)$ time per phase plus $O\left(n/p + l_{\max}\log^2 n\right)$ overall for light requests; work done in a load balanced way cannot take more than $O(dn/p)$ expected time.[5]

Theorem 3. *The SSSP on random graphs from $G(n, \frac{d}{n})$ with random edge weights can be solved in expected time $O(\log^3 n/\log\log n)$ and $O(dn)$ work using $\frac{dn\log\log n}{\log^3 n}$ processors on a CRCW PRAM.*

5 Adaptation to Distributed Memory

Let $T_{\texttt{routing}}(k)$ denote the time required to route k constant size messages per PU to random destinations. Let $T_{\texttt{coll}}(k)$ bound the time to perform a (possibly segmented) reduction or broadcast involving a message of length k. The analysis can focus on finding the number of necessary basic operations. The execution time for a particular network or abstract model is then easy to determine. For example, in the BSP model [20] we simply substitute $T_{\texttt{routing}}(k) = O(l + (k + \log p)g)$ and $T_{\texttt{coll}}(k) = O((l + gk)\log p)$.

Let $r = O\left(\log^2 n/\log\log n\right)$ denote the number of phases required whp. Our PRAM algorithm is already almost a distributed memory algorithm if we restrict ourselves to the practically most interesting case $p \le \frac{n}{r\log n}$. Each PU stores the

[5] All results also hold with high probability if such a bound for the number of rerelaxations is available.

adjacency lists of the nodes assigned to it using a hash function ind(w) we assume to be computable in constant time. (Essentially the same assumptions are made for efficient PRAM simulation algorithms [28, Section 4.3] and this is certainly warranted for the simple hash functions used in practice.) Load balancing for generating requests is already achieved if each PU simply scans the adjacency lists of its local part of $B[i]$ (for random graphs).

The dart throwing process for assigning requests can be replaced by simply routing a request $(w; x)$ to PU ind(w). An analysis similar to the PRAM case yields an expected execution time in $O(dn/p + r(T_{\text{coll}}(1) + T_{\text{routing}}(dn/(pr))))$.

Many edges Using some additional measures we can employ more PUs ($p = \frac{m}{r \log n}$ as in the PRAM case). Again we concentrate on the case $d = O(\log n)$. Now d PUs work together in *p/d list groups*. As before, nodes are hashed to individual PUs. But the heavy parts of the adjacency lists are independently assigned to a random list group where they are stored in a global round robin fashion: looking at a particular list of length l, each PU in the corresponding list group will either store $\lfloor l/d \rfloor$ or $\lceil l/d \rceil$ entries of that list. When heavy edges are relaxed, a PU sends the distances of the nodes it has deleted to the first PU in the list group responsible for this node. The first PU in a list group transmits all the distance node pairs it has received to the other group members using a pipelined broadcast operation. After a detailed analysis and similar measures as in the PRAM case for $d \geq \log n$ we get:

Theorem 4. *If the number of delete-phases is bounded by r then the SSSP on random graphs with random edge weights can be solved on a distributed memory machine with $p \leq \frac{dn}{r \log n}$ processors in expected time $O(\frac{dn}{p} + r(T_{\text{coll}}(\frac{dn}{pr}) + T_{\text{routing}}(\frac{dn}{pr})))$ and $O(dn)$ work.*

Note, that on powerful interconnection networks like multiported hypercubes we can achieve a time $O(\log p + k)$ whp for $T_{\text{routing}}(k)$ and $T_{\text{coll}}(k)$ so that we get the same asymptotic performance as our CRCW-PRAM algorithm.

6 Simulation and Implementation

Simulations of different algorithm variants played a key role in designing the algorithm and are still interesting as a means to estimate the constant factors involved. $\Delta = 4/d$ proves to be a good choice for the bucket range: For all tested values for $d \geq 2$ the number of phases were bounded by $5 \ln n$ and less than $0.25n$ reinsertions occured. Reachable nodes could be accessed by paths of weight $2.15\frac{\ln n}{d}$ or shorter. For sparse graphs, the number of phases can be further reduced at a low price in terms of reinsertions by emptying all buckets at once when they hold only few nodes.

Actually implementing a linear work algorithm which requires a linear number of tiny messages with irregular communication pattern is not easy. However, for small p and large n, a machine with high bandwidth interconnection network and an efficient library routine for personalized all-to-all communication can do

the job. We implemented a simple version of the algorithm for distributed memory machines and random d-regular graphs using the library MPI [24]. Tests were run on an INTEL Paragon with 16 processors. For $n = 2^{19}$ nodes and $d = 3$ speedup 9.2 was obtained against the sequential -stepping approach. The latter in turn is 3.1 times faster than an optimized sequential implementation of Dijkstra's algorithm. Due to the increased communication costs, our results on dense graphs are slightly worse: for $n = 2^{16}$ and $d = 32$ the speedup of parallel -stepping compared to its sequential counterpart was 7.5 [6], sequential -stepping was 1.8 times faster than Dijkstra's algorithm.

7 Conclusions and Future Work

A $(1/d)$-stepping scheme solves the SSSP for random graphs from $G(n, d/n)$ with random edge weights in $O(\log^3 n / \log\log n)$ time and $O(dn)$ expected work on PRAMs and many distributed memory machines. If one views random graphs with random edge weights as a model for "average" graphs this implies a striking difference between the average case and the worst case for which no sublinear time work efficient solution is known.

The number of phases can be reduced by a factor $(\log n / \log\log n)$ if shortcut edges for paths in $P_{(1/d)}$ are inserted. Following our simulations we conjecture that even without shortcuts $O(\log n)$ phases suffice. It might be possible to further speed up the CRCW-PRAM algorithm by replacing our $O(\log n)$ load balancing and dart throwing routines by "almost constant time algorithms" [15]. For $d = \Omega(\log n)$ this might even be possible in a work efficient way.

Our SSSP solution immediately yields an improved algorithm for the all-pairs shortest path problem on random graphs with random edge weights.

For random graphs with random edge weights the random assignment of nodes to PEs should be dispensable. Since this is the only source of randomness needed in the distributed memory algorithm, we get a deterministic algorithm. From a practical point of view it is more interesting however to lift the randomness assumptions on the graph and the edge weights.

For arbitrary directed graphs with random edge weights and maximum degree d sequential $(1/d)$-stepping works in expected time $O(d_c d + m + n)$ where d_c is the maximum shortest path weight. This is linear for all but quite degenerate cases. If $d_c d$ m there is also considerable parallelism which can be exploited without affecting work efficiency. However, there is some work to be done regarding an efficient parallel implementation for graphs where load balancing is more difficult than for random graphs without shortcuts.

Interesting research can be done on parallel shortest path for non-random edge weights. One approach could be to start with $= \min_{e \in E} c(e)$ and then double until $|P|$ n. $|P|$ can for example be determined using parallel

[6] Our current implementation does not distinguish between heavy and light edges which increases the communication overhead. Therefore, we expect somewhat higher speedups for the full version of the paper.

depth first traversal of light paths starting from each node. However, this is not work efficient for small d.

One can also look for other ways of determining the set R of nodes to be deleted in a phase. We have made experiments where $|R|$ is some fraction of the total priority queue size $|Q|$. In our simulations this works as well as $(1/d)$-stepping for random graphs with $|R| = \Theta(|Q|/\log|Q|)$, for random planar graphs we could even use $|R| = |Q|/2$. We also tested this approach on real world graphs and edge weights: starting with a road-map of a town ($n = 10{,}000$) the tested graphs successively grew up to a large road-map of Southern Germany ($n = 157{,}457$). Good performance was found for $|R| = \Theta(|Q|^{3/4})$. While repeatedly doubling the number of nodes, the average number of phases (for different starting points) only increased by a factor of about 1.5; for $n = 157{,}457$ the simulation needed $1{,}178$ phases, the number of reinserts was bounded by $0.2n$. In [10,9] we develop an algorithm which needs no reexpansions for arbitrary edge weights. However, even for random edge weights it needs $\Omega(n^{1/3})$ phases even for random edge weights.

Acknowledgements

We would like to thank in particular Kurt Mehlhorn and Volker Priebe for many fruitful discussions and suggestions.

References

1. P. Adamson and E. Tick. Greedy partitioned algorithms for the shortest path problem. *International Journal of Parallel Programming*, 20(4):271–298, 1991.
2. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows : Theory, Algorithms and Applications*. Prentice Hall, 1993.
3. N. Alon, J. H. Spencer, and P. Erdős. *The Probabilistic Method*. Wiley, 1992.
4. G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operation. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 689–693. IEEE, 1997.
5. K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–837, 1982.
6. A. Clementi, J. Rolim, and E. Urland. Randomized parallel algorithms. In A. Ferreira and P. Pardalos, editors, *Solving Combinatorial Optimization Problem in Parallel*, volume 1054 of *LNCS*, pages 25–50. Springer, 1996.
7. E. Cohen. Polylog-time and near-linear work approximation scheme for undirected shortest paths. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, pages 16–26, 1994.
8. E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.
9. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *23rd Symposium on Mathematical Foundations of Computer Science*, LNCS, Brno, Czech Republic, 1998. Springer.
10. A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. Parallelizing Dijkstra's shortest path algorithm. Technical report, MPI-Informatik, 1998. in preparation.

11. E. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.
12. E. A. Dinic. Economical algorithms for finding shortest paths in a network. In *Transportation Modeling Systems*, pages 36–44, 1978.
13. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.
14. A. Frieze and G. Grimmett. The shortest-path problem for graphs with random arc-lengths. *Discrete Appl. Math.*, 10:57–77, 1985.
15. T. Hagerup. The log-star revolution. In A. Finkel and M. Jantzen, editors, *Proceedings of Symposion on Theoretical Aspects of Computer Science (STACS '92)*, volume 577 of *LNCS*, pages 259–280. Springer, Feb. 1992.
16. Y. Han, V. Pan, and J. Reif. Efficient parallel algorithms for computing all pairs shortest paths in directed graphs. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 353–362. ACM Press, 1992.
17. R. Hassin and E. Zemel. On shortest paths in graphs with random weights. *Math. Oper. Res.*, 10(4):557–564, 1985.
18. J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
19. P. N. Klein and S. Sairam. A parallel randomized approximation scheme for shortest paths. In *Proc. 24th Ann. ACM Symp. on Theory of Computing*, pages 750–758, Victoria, B.C., Canada, 1992.
20. W. F. McColl. Universal computing. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. Euro-Par '96 Parallel Processing*, volume 1123 of *LNCS*, pages 25–36. Springer, 1996.
21. G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Symposium on Foundations of Computer Science*, pages 478–489. IEEE, 1985.
22. R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *International Conference on Parallel Processing*, pages 14–20. IEEE, 1985.
23. P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.
24. M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI – the Complete Reference*. MIT Press, 1996.
25. M. Thorup. Undirected single source shortest paths in linear time. In *38th Annual Symposium on Foundations of Computer Science*, pages 12–21. IEEE, 1997.
26. J. L. Träff. An experimental comparison of two distributed single-source shortest path algorithms. *Parallel Computing*, 21:1505–1532, 1995.
27. J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Irregular' 96*, volume 1117 of *LNCS*, pages 183–194. Springer, 1996.
28. L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, pages 943–971. Elsevier, 1990.

Improved Deterministic Parallel Padded Sorting

Ka Wong Chong and Edgar A. Ramos

Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

Abstract. Given an input array of n real numbers, sorting with padding consists in writing those numbers in order in an array of size $(1+\ )n$, thus leaving n entries *empty*. Only comparisons are allowed between the numbers to be sorted. We describe an algorithm that, with nk processors in the CRCW PRAM model, sorts the input array with padding $o(1)$ using time $O(\log_k n \log\ (\log_k n) + \text{loglog}\, k)$. This improves a previous algorithm with time bound $O(\log_k n\ (\text{loglog}\, k)^3\ \ 2^{C(\log\ n - \log\ k)})$. The best known lower bound is $\Omega(\log_k n)$.

1 Introduction

Given an input array of n real numbers, sorting with padding consists in writing those numbers in order in an array of size $(1+\ )n$, thus leaving n entries *empty*. Comparisons are the only operations allowed on the numbers to be sorted. Although in the CRCW PRAM model there is a lower bound $\Omega(\log n/\text{loglog}\, n)$ for *standard* sorting (i.e, without padding) using a polynomial number of processors [9], it is possible to *pad-sort* faster [31,25,26]. In particular, one is interested in studying the speed-up achievable when nk processors are available. The parameter k is called the *processor advantage*. Using randomization, the known lower bound $\Omega(\log_k n)$[1] [6,3,11] is matched by an algorithm by Hagerup and Raman [25]. They also gave a deterministic algorithm with running time $O(\log_k n(\text{loglog}\, k)^5\ \ 2^{C(\log\ n - \log\ k)})$ [26]. Goldberg and Zwick [20] reduced the exponent in the $\text{loglog}\, k$ term to 3 by showing that approximate counting is possible in constant time with a polynomial number of processors in the CRCW PRAM model (a translation of the work of Ajtai [2] for bounded depth circuits).

We describe a deterministic algorithm for the CRCW PRAM model that can pad-sort n numbers in time $O(\log_k n \log\ (\log_k n) + \text{loglog}\, k)$ using nk processors with padding $= 1/\log^c(k + \log n)$. If $k = \log\ n$, for any constant > 0, our algorithm provides a nearly optimal $o(\log n)$ time standard sorting algorithm: it sorts in time $O(\log n \log\ n/\text{loglog}\, n)$ using $n \log\ n$ processors (this time is a factor $\log\ n$ greater than the lower bound also matched by a randomized algorithm of Rajasekaran and Reif [36]). Alternatively, the optimal time can be achieved with a processor advantage of $(\log n)^{\log\ n}$.

Our algorithm is a refinement of that in [26], which uses a natural divide-and-conquer approach. Specifically, it computes a subset of the input, called *splitter*,

[1] As usual $\log_k n$ denotes logarithm base k of n, that is $\log n/\log k$, and $\log\ n = \min\{j : \log^{(j)} n \quad 1\}$ where $\log^{(1)} n = \log n$, $\log^{(i)} n = \log(\log^{(i-1)} n)$.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 405{416, 1998.
c Springer-Verlag Berlin Heidelberg 1998

that partitions the input into "nearly equal" *intervals* and then pad-sorts those intervals recursively. Unlike their algorithm, we make the problem of computing a splitter independent of that of sorting and integrate the computation of a splitter with the computation of the corresponding intervals induced by the splitter. We essentially reduce the sorting problem to the problem of computing a splitter. As a result, we obtain an additive term $\log\log k$ in the running time rather than a multiplicative one, and a multiplicative term $\log(\log_k n)$ rather than $2^{C(\log n - \log k)}$. At the base of the splitter computation, there is an algorithm that computes a splitter in time $O(1)$ using a polynomial number of processors; this is obtained by derandomizing a probabilistic argument through the method of limited independence. Then a slower but more processor efficient algorithm is obtained by taking advantage of the properties of *samples*, a concept stronger than splitters. Finally, the general algorithm works by successive refinement using the previous algorithm. Our algorithms rely essentially on the CRCW PRAM implementation by Goldberg and Zwick [20] of the approximate counting circuits of Ajtai [2]. Also, we take advantage of previous work on random sampling in the context of computational geometry [38,15,32,33,21,22,24], like the concept of *semi-splitters* and constant time computation of good samples and good splitters.

In Section 2 we review results concerning approximate counting and its applications which are needed in our algorithms. In Section 3 we describe our pad-sorting algorithm. The remaining sections deal with computing splitters. All the results in this paper are for the Arbitrary CRCW PRAM model. Due to lack of space, most of the proofs are omitted in this extended abstract.

2 Approximate Counting and Applications

In this section, we review approximate versions of several basic operations needed in our sorting algorithm. Though exact counting in the CRCW PRAM model requires $\Omega(\log n/\log\log n)$ time (by a reduction to the parity problem [9]), in many applications approximate counting is sufficient and can be performed faster. A number x' is a $\quad$-*approximation* of x if $x/(1+\quad) \quad x' \quad (1+\quad)x$: After some previous work in [23], Goldberg and Zwick [20] pointed out that Ajtai's bounded depth circuits for approximate counting [2] can be translated directly into an algorithm in the CRCW PRAM model. Their result is the following.

Fact 1 *For any constant $c > 0$, there is a constant $a > 0$ such that given an array of n integers, each with $O(\log n)$ bits, their sum can be $(\log n)^{-c}$-approximated in time $O(1)$ using n^a processors.*

Approximate Prefix Sums. Correspondingly, the definition of the *prefix sums* problem, which are the basis of many parallel algorithms, has a relaxed version: A sequence $0 = b_0, b_1, \ldots, b_n$ is said to be a $\quad$-approximate prefix sums of a sequence $a_1, a_2, \ldots, a_n$ if for every i, $1 \quad i \quad n$, we have $\sum_{j=1}^{i} a_j \quad b_j \quad (1+\quad)\sum_{j=1}^{i} a_j$ (approximation) and $b_i - b_{i-1} \quad a_i$ (consistency). The following is an application of Fact 1 as shown in [20].

Fact 2 *For any constant $c > 0$, there is a constant $a > 0$ such that given an array of n integers, each with $O(\log n)$ bits, the $(\log n)^{-c}$-approximate prefix sums can be computed in time $O(1)$ using n^a processors.*

Goldberg and Zwick show how to reduce the number of processors so that the amount of work performed is linear (optimal). In that case, the time increases to $O(\text{loglog}\, n)$ and the approximation factor to $1 + 1/(\text{loglog}\, n)^c$. Following the same argument, one can show the following result for nk processors.

Lemma 1. *Given an array of n integers, each with $O(\log n)$ bits, the $(\log(k + \log n))^{-c}$-approximate prefix sums can be computed in time $O(\log \log_k n)$ using nk processors.*

Applications. The result on approximate prefix sums leads directly to the approximate solution of some basic operations in parallel computation [20]:

(i) *Approximate Compaction:* Given a boolean array of size n, compact the indices of the places that contain 1's into an array of size at most $(1 + \ )b$, where b is the number of 1's in the array.
(ii) *Interval Allocation:* Given a sequence of n integers, $a_1, a_2, \ldots, a_n$, each with $O(\log n)$-bit, allocate n non-overlapping intervals of lengths $a_1, a_2, \ldots, a_n$ within an interval of length $(1 + \ ) \sum_{i=1}^{n} a_i$.

Lemma 2. *The approximate compaction and the interval allocation problems of size n can be solved in $O(\log \log_k n)$ time using nk processors and with $= O(1/\log^c(k + \log n))$.*

3 Pad-Sorting Algorithm

Let X be a set of real numbers. For $a, b \in \ <$, the *interval* $X(a, b)$ is $X \setminus (a, b)$ where (a, b) is the usual real interval. The set $I(S, X)$ of *basic* intervals of X induced by S consists of the intervals $X(a, b)$ where $a, b \in S \cup \{-1, +1\}$ and $S \setminus (a, b) = \ $ (i.e., a and b are *consecutive* in S). Our pad-sorting algorithm essentially reduces the problem to that of computing a *splitter*: $S \subseteq X$ is an $\ $-splitter for X, with $0 < \ < 1$, if for each $I \in I(S, X)$, $|I| \leq \ |X|$. For conciseness, an $\ $-splitter of size at most $C/$ is called a $(C, \ )$-splitter. We say that $\ $ is the *coarseness* and C the *size factor* of the splitter. In Section 7, we describe an algorithm, called `Splitter`(X, r), [2], which returns **(i)** a $(1/r)$-splitter S of size $O(n \ r)$ for X, for some constant $0 < \ < 1/2$, and **(ii)** the set of basic interval $P = I(S, X)$. `Splitter`(X, r) runs in time $O(\log_k r \log\ (\log_k n) + \log \log_k n)$ using nk processors. The concept of splitter leads naturally to a divide-and-conquer sorting algorithm as in [26]: If X has size $O(1)$ then it can be sorted in

[2] Though it is more natural to use the parameter $\ $ in the definition and also more convenient to express some properties, we also let $\ = 1/r$ and use the parameter r in procedures and their analysis (running times, processor bounds).

$O(1)$ time; otherwise, we compute a splitter S for X, and then in parallel, S and each interval determined by S in X are pad-sorted recursively. Two important points of our approach are that a splitter is not required to be sorted, and the computation of a splitter and finding its intervals are integrated.

```
    Pad-Sort(X)
1.    If jXj = O(1), then sort X in O(1) time and return it
2.    Let (S; P)    Splitter(X; jXj^{1=2})
3.    In parallel:
3.1.    S^0    Pad-Sort(S)
3.2.    For each I 2 P, let I^0    Pad-Sort(I)
4.    For each x 2 X compute its approximate rank r and write into X^0[r]
5.    Return X^0 compressed into size of at most (1 + )jXj
```

Since `Pad-Sort` works recursively, we allow the input itself to be a padded array. We can view `Pad-Sort`(X) as computing a *consistent approximate rank* for each element in X: an approximation to the actual rank which is consistent with the ordering. In Step 4, consistent approximate ranks are computed using the approximate ranks in S^0 and in the I^0. The allocation needed in Step 3, the rank computation in Step 4, and the compaction in Step 5 are performed using approximate counting. This takes $O(\log\log_k n)$ time. To achieve the desired time bound, it is sufficient that the $(1=n^{1=2})$-splitter have size $O(n^{1=2+\ })$, where $0 < \ < 1=2$ is a small constant.

The running time $T(n)$ of `Pad-Sort` satisfies the recurrence relation:[3]

$$T(n;k) \quad \begin{array}{ll} A & \text{for } n \quad n_0 \\ T(n^{1=2+\ };k) + B\log_k n \log\ (\log_k n) & \text{otherwise} \end{array}$$

The solution is $O(\log_k n \log\ (\log_k n) + \log\log k)$. Thus, we obtain the following.

Theorem 1. *For any integer* $k \quad 1$, *an array of* n *numbers can be sorted with padding* $1=\log^c(k+\log n)$ *in* $O(\log_k n \log\ (\log_k n) + \log\log k)$ *time using* nk *processors.*

Remark. The algorithm `Pad-Sort` actually shows that $T(n;k)$ is bounded by $O(T_s(n)) + T(k;k)$, where $T_s(n)$ is the time to compute the splitter. This is made clear by modifying Step 1 to invoke `PolySort`(X), a procedure to pad-sort using n^a processors, when $jXj \quad k$. We observe that the discrepancy between our upper bound and the $\Omega(\log_k n)$ lower bound comes from two sources: computing the splitter produces the $\log\ (\log_k n)$ term, and pad-sorting with a polynomial number of processors produces the $\log\log k$ term. It appears that the first discrepancy can be improved; in fact, Cole's k-way merge sort [17] achieves a time bound $O(\log n=\log\log k)$ which is better than our algorithm for small k. However, it is not clear whether that factor can be removed altogether. As for the second discrepancy, it remains an open problem whether $T(k;k)$ is $\Omega(\log\log k)$.

Using exact compression on the output of the pad-sort algorithm, we can improve previous bounds for standard sorting in $o(\log n)$. Note that there is a time lower bound $\Omega(\log n=\log\log n)$ when $n\log\ n$ processors is used [3,6,11].

[3] We assume $\log x = 1$ for $x < 2$.

Corollary 1. *An array of n numbers can be sorted (without padding) (i) in time $O(\log n \log n/\log\log n)$ using $n \log n$ processors or (ii) in time $O(\log n \log\log n)$ using $n(\log n)^{\log^* n}$ processors.*

In the remaining sections we concentrate on the computation of splitters. First, we show that a $(1/r)$-splitter can be computed in $O(1)$ time using n^a processors. Second, using the stronger concept of $(\ ;\ )$-samples, a $(1/r)$-splitter can be computed in $O(\log\log_r n)$ time using nr^a processors. Finally, an $O(\log_k r \log(\log_k n) + \log\log_k n)$ time algorithm using nk processors is given.

4 Sampling in Constant Time

The final splitter algorithm is built upon basic algorithms that use $O(1)$ time but require a polynomial number of processors. Two other *sampling* concepts turn out to be relevant in the computation of splitters. They have been used in previous works on selection and sorting [18,12,25,26], and also in computational geometry [38,27,15,32,21]

4.1 Two Other Sampling Concepts

Samples. To achieve better processors bound, we consider splitters that satisfy a stronger requirement. A $(\ ;\ )$-*sample* for X is a subset $S \subseteq X$ such that for any interval $I = X(a;b)$ (not just the basic ones), the following holds:

$$\frac{|X \setminus I|}{|X|} \qquad \frac{|S \setminus I|}{|S|} + \ :$$

This gives an upper bound for the *density* of I in X in terms of the density of I in S. Observe that a $(\ ;\ )$-sample is automatically an $\ $-splitter. In Section 6, we see the advantage by strengthening the concept of splitter to that of sample.

Semi-Splitters. An $(a;C;\ )$-*semi-splitter* for X is a subset $S \subseteq X$ such that for any $c \geq a$, the following *moment* bound holds:

$$\sum_{I \supseteq I(S;X)} |I|^c \leq C(\ n)^c \frac{1}{-}:$$

Note that an $(a;C;\ )$-semi-splitter has size at most $C/\ $. Although a semi-splitter may fail to have all basic intervals of size at most $\ n$, still there is some control on these sizes in the form of the moment bounds. As a result, a semi-splitter can be refined into a good splitter using a weaker splitter algorithm as follows [14,15,32,7,19]. Assume that on input $(X;\ )$, algorithm **A** computes an $(a;C;\ )$-semi-splitter for X, and that on input $(X;\ )$, algorithm **B** computes an $\ $-splitter of size $D(1/\ )^c$ (i.e., too large) for X with $c \geq a$. A better $\ $-splitter is computed as follows: First, compute an $(a;C;\ )$-semi-splitter S using algorithm **A**, and then for each $I \supseteq I(S;X)$ with $|I| \geq |X|/|S|$, compute an $\ _I$-splitter S_I for I of size $D(1/\ _I)^c$ where $\ _I = |X||I|/|S|$, using algorithm **B**. The union T of S and all the S_I is clearly an $\ $-splitter for X, and its size is at most $CD/\ $:

$$\sum_I D\left(\frac{1}{\ _I}\right)^c \leq D\left(\frac{1}{|X|}\right)^c \sum_I |I|^c \leq D\left(\frac{1}{|X|}\right)^c C(\ |X|)^c \frac{1}{-} \leq \frac{CD}{\ }:$$

4.2 Probability Spaces with Limited Independence

To compute good samples in parallel fast and deterministically, we follow the approach of sampling using probability spaces whose *support* has polynomial size. More specifically, we use t*-wise independent* probability spaces [28,30,4].[4] The following fact is essential for our algorithms [28,29].

Fact 3 *For any* s, $1 \quad s < n$ *and* $p = s/n$, *there exists a probability space for* n *indicator r.v.'s so that (i) it is* t*-wise independent, (ii) the size of its support is* $O(n^t)$, *and (iii) the value of each r.v. at each point of the probability space can be evaluated in constant time (which depends on* t*).*

4.3 Main Sampling Results

For s with $1 \quad s < n$, a $(t; s)$*-random-sample* is a subset $S \quad X$ obtained by choosing each element $x \; 2 \; X$ into S with probability $p = s/n$ using a t-wise independent probability distribution. The following lemma is our basic tool.

Lemma 3. *Let* $S \quad X$ *be a* $(t; s)$*-random-sample. Then, for constants* $D; D^0; D^{00}$ *depending on* t, *with probability at least 1/2 the following holds:*

(i) If $s = Dr^{1+2/t}$ *then* S *is a* $(Dr^{2/t}; r)$*-splitter for* X.
(ii) If $s = D^0 r^{2+4/t}$ *then* S *is a* $(1; r)$*-sample for* X.
(iii) If $s = r$ *then* S *is a* $(t/2 - 4; D^{00}; r)$*-semi-splitter for* X.

Remark. The more general results in [21,22,24] only claim sizes rn and $r^2 n$ in (i) and (ii) of the theorem. For our purposes, in (i) and (ii), it will be sufficient to use $t = 2$; but in (iii) we need at least $t = 12$ to obtain a $(2; \; ; \;)$-semi-splitter (and we actually need an even higher exponent in Lemma 5).

4.4 Coping with Approximate Counting

Our goal is to derive from Lemma 3 constant time algorithms via derandomization. An obstacle to this, however, is that exact counting to verify the property of a sample is not available. Fortunately, approximate verification suffices. Following [22,24], for a set jXj, let jXj^0 be a corresponding -approximate size. We say that S is -*verified* to be a $(\; ; \;)$-sample if $jX \setminus Ij^0/jXj^0 \quad jS \setminus Ij^0/jSj^0 + \; .$

Observation 1 *If* S *is a* $(\; ; \;)$*-sample, then* S *is* -*verified to be a* $(\; (1 + \;)^4; \; (1 + \;)^2)$*-sample. If* S *is* -*verified to be a* $(\; ; \;)$*-sample, then* S *is a* $(\; (1 + \;)^4; \; (1 + \;)^2)$*-sample.*

Since the probabilistic argument guarantees the existence of a $(1; \;)$-sample, it follows that we can find a $((1 + \;)^8; \; (1 + \;)^4)$-sample, a loss in the quality but sufficient in our applications.

[4] In our algorithm, there is no advantage in using other more efficient constructions of probability spaces, like spaces with *almost k-wise independence* [35,5].

4.5 Algorithms

By Lemma 3, Fact 3, and Observation 1, we can obtain $O(1)$ time algorithms for computing splitters and samples using a polynomial number of processors.

Lemma 4. *For any constant $c > 0$, there is a constant $a > 0$ such that a $(1 + 1/\log^c n; 1/r)$-sample for X of size $O(r^4)$ can be computed in $O(1)$ time using n^a processors.*

Lemma 5. *For some constants $C_1; a > 0$, a $(C_1; 1/r)$-splitter for X can be computed in $O(1)$ time using n^a processors.*

For later use, we call these algorithms `PolySample` and `PolySplitter` respectively. The latter one returns the splitter S and $P = I(S; X)$.

5 Splitters via Re ning

The computation of a splitter can be reduced to that of coarser splitters by *refining*. Suppose `BaseSplitter`$(X; r)$ computes a $(C; 1/r)$-splitter for X, for $r \quad r_0$. An algorithm `RefineSplitter`$(X; r)$ for computing a $(1/r)$-splitter for X with unrestricted r can be obtained by iterating algorithm `BaseSplitter` in the natural manner:

`RefineSplitter`$(X; r)$
1. If $r \quad r_0$ then return `BaseSplitter`$(X; r)$
2. Else
2.1 Let $(S; P) \quad$ `BaseSplitter`$(X; r_0)$
2.2. For each $I \ 2 \ S$, $(S_I; P_I) \quad$ `RefineSplitter`$(I; rjIj/jXj)$
2.3. Return $(S [\ _I S_I; \ _I P_I)$

Lemma 6. *Let $r_0 \quad 2$. Assuming that* `BaseSplitter`$(X; r)$ *returns a $(C; 1/r)$-splitter for $r \quad r_0$ then* `RefineSplitter`$(X; r)$ *returns a $(3C; 1/r)$-splitter.*

6 Splitters via Resampling

A processor efficient algorithm for computing a splitter is obtained by reduction to computation of splitters of smaller sets. For this, one actually needs the stronger concept of $(\ ; \ )$-sample. Specifically, samples have the following properties [32,16,21,18,26].

Observation 2 *(i)* Merging: *Let $X_1; : : : ; X_m$ be disjoint sets with $X = \ _{i=1}^{m} X_i$ and $jX_ij = jXj/m$. For $i = 1; : : : ; m$, let S_i be a $(\ ; \ )$-sample for X_i and they all have the same size. Then $S = \ _{i=1}^{m} S_i$ is a $(\ ; \ )$-sample for X. (ii)* Resampling: *Let S be a $(\ ; \ )$-sample for X and T a $(\ ^0; \ ^0)$-sample for S. Then T is a $(\quad ^0; \quad ^0 + \ )$-sample for X.*

These properties lead naturally to a divide-and-conquer algorithm to compute a sample.[5] Let $b = 8a$ where a is the maximum of the a's in Lemmas 4 and 5. Let $= 1=2a$ and $\gamma = 1=8a(1 -)$. We assume the processor advantage is r^b.

```
Resample(X; r)
1.    If jXj   r^8 then return PolySample(X; r)
2.    Else
2.1.    Split X into l = n  groups X_1; : : : ; X_l of equal size
2.2.    For each i, let S_i   Resample(X_i; r)
2.3.    Return PolySample( S_i; n^γ r^{1-8γ})
```

Remark. For simplicity, in Step 2.1 we ignore that the exact splitting is not possible in general. We can take care of this by allowing the last group to be larger than the others, and adjusting the parameter r for that group in Step 2.2 accordingly. Also, we ignore that the S_i's do not have all exactly the same size, so Observation 2 does not apply as stated. We can also take care of this: the approximations are of size Cr^4 $O(r^3)$, so the relative error is only $O(1=r)$ and can be absorbed by the error resulting from the approximate counting.

Lemma 7. *For some constants $C_2; C_3 > 0$,* `Resample`$(X; r)$ *returns a $(C_2; C_3=r)$-sample of size $O(n^{4\gamma} r^{4(1-8\gamma)})$ in $O(\log\log_r n)$ time using nr^b processors.*

The resulting sample is too large, but that can be easily corrected with a further sampling step. The following simple observation allows us to mix the splitter and sample computations.

Observation 3 *(i) A $(C;)$-splitter for X is a $(C;)$-sample for X. (ii) Let Y be a $(;)$-sample for X and let Z be an ^{0}-splitter for Y. Then Z is a $(^0 +)$-splitter for X.*

Thus, using `Resample` we can now write `ResampleSplitter` which assumes the processor advantage is r^b. Here, C_2 and C_3 are the constants in Lemma 7.

```
ResampleSplitter(X; r)
1.    Let S^0   Resample(X; 2C_3 r)
2.    Let S   PolySplitter(S^0; 2C_2 r)
3.    P   Locate(S; X)
4.    Return (S; P)
```

By Lemma 7 and Observation 3, the set S in Step 2 is a $(2C_1C_2; r)$-splitter where C_1 is the constant of Lemma 5. In Step 3, procedure `Locate`$(S; X)$ determines the basic intervals induced by S on X. Since $jSj = O(r)$ and there is a processor advantage of r^b, `Locate`$(S; X)$ can be performed as follows: in $O(1)$ time, find the interval for each element, and in additional $O(\log\log_r n)$ time, compress the elements of each interval. This shows the following lemma.

Lemma 8. `ResampleSplitter`$(X; r)$ *returns a $(C_4; 1=r)$-splitter in time $O(\log\log_r n)$ using nr^b processors where $C_4 = 2C_1C_2$.*

[5] This recursive presentation follows the work in [32,21,22]. In [26], the corresponding computation follows a bottom up approach that they call *resampling*.

7 Splitter Algorithm

¿From the previous sections, we obtain a first complete algorithm for computing a splitter using nk processors. Setting $r_0 = k^{1=b}$ and using `ResampleSplitter` as `BaseSplitter` in `RefineSplitter` we obtain the following.

Lemma 9. *A* $(C_5; 1=r)$*-splitter can be found in time* $O(\log_k r \log\log_k n)$ *using* nk *processors, where* $C_5 = 3C_4$.

As in [26], `RefineSplitter` can be improved by selecting a larger r_0. Consider the computation tree of `Resample`. Each level, including the bottom one, uses $O(1)$ time, for a total of $O(\log\log_k n)$ time. The idea is to increase r_0 and replace `PolySplitter` by another procedure so that the bottom level of the computation also uses $O(\log\log_k n)$ time, while the other levels still use $O(1)$ time. The choice is $r_0 = k^0(n)$, where $k^0(n) = k^{\log\log_k n}$ (note that $\log_k k^0(n) = \log\log_k n$). Accordingly, we replace `PolySplitter` by a recursive call to `RefineSplitter`; thus, unlike the algorithm in [26], we avoid the use of sorting. The modified procedures are combined into procedure `Splitter` (C_6 is a constant to be determined):

`Splitter`$(X; r)$
1. If $r \quad k^0(jXj)$ then
1.1. If $k \quad r^b$ then return `ResampleSplitter`$(X; r)$
1.2. Let $\hat{r} \quad rC_6^{\log\ (\log_k n)-1}$
1.3. Divide X into blocks $Q_1; : : : ; Q_l$ of size $\hat{r}^{b+1}$
1.4. For each i, let $(S_i; P_S) \quad$ `Splitter`$(Q_i; 2r)$
1.5. Let $S^0 \quad$ `Resample`$(\ _i S_i; 4C_3\hat{r})$
1.6. Let $(S; P) \quad$ `PolySplitter`$(S^0; 4C_2\hat{r})$
1.7. $P \quad$ `Locate`$(S; (S_1; P_1); : : : ; (S_l; P_l))$
1.8. Return $(S; P)$
2. Else
2.1. Let $(S; P) \quad$ `Splitter`$(X; k^0(jXj))$
2.2. For each $l \ 2 \ S$, $(S_l; P_l) \quad$ `Splitter`$(l; rjlj=jXj)$
2.3. Return $(S \ [\ _l S_l; \ _l P_l)$

`Splitter` works as follows. If $r > k^0(n)$ then Steps 2.1 to 2.3 perform a refinement. If k is already sufficient processor advantage then `ResampleSplitter` is used in Step 1.1. Otherwise, Steps 1.3 and 1.4 achieve the processor advantage of $\hat{r}^b$ needed in Steps 1.5 and 1.6 to obtain the desired splitter; and finally Step 1.7 uses the procedure `Locate` to compute the basic intervals. The value of $\hat{r}$ set in Step 1.2 is used in Steps 1.5 and 1.6 to compensate for the size factor of the splitter obtained in Step 1.4 recursively. Note that `Locate` is no longer trivial because the processor advantage is only k while S has size $O(rC_6^{\log\ (\log_k n)})$. This procedure runs in time $O(\log\log_k n)$ and is described below. Assuming this, we can evaluate the running time of the algorithm, and the size and coarseness of the splitter returned.

Lemma 10. `Splitter`$(X; r)$ *runs in* $O(\log_k r \log\ (\log_k n) + \log\log_k n)$ *time using* nk *processors.*

Proof. The time $T(n; r)$ required by `Splitter`$(X; r)$, with $n = |X|$, satisfies the following recurrence relation:

$$T(n; r) \quad \begin{cases} A \log\log_k n & \text{if } k \quad r^b \\ T(n; k^0(n)) + T(n/k^0(n); r/k^0(n)) + B \log\log_k n & \text{if } k^0(n) < r \\ T(r^{b+1}; r) + D \log\log_k n & \text{otherwise} \end{cases}$$

Using this recurrence, one can verify that $T(n; r)$ is bounded as claimed. ∎

Lemma 11. *For some constant* $C_6 > 0$, `Splitter`$(X; r)$ *returns a* $(C_6^{\log\ (\log_k n)}; 1/r)$*-splitter for* X*.*

Procedure `Locate`**.** Let $(n) = C_6^{\log\ (\log_k n)}$. We assume that $k \quad C_7^{\log\ n}$ where $C_7 > 0$ is a constant so that $(n) \quad k^{1/2}$. This assumption can be removed by slightly complicating the algorithm[6] and it is not a major limitation since in any case for $k < C_7^{\log\ n}$ the time bound for our algorithm is outperformed by that of Cole's k-way merge-sort. For a set X, let $I(X) = I(X; <)$, the basic intervals induced by X in $<$.

`Locate`$(S; (S_1; P_1); : : : ; (S_l; P_l))$

1. For each i, locate each $x \in S$ in $I(S_i)$ and each $x \in S_i$ in $I(S)$
2. For each i and each $x \in P_i$, locate x in $I(S)$
3. For each i collect those $x \in P_i$ lying in the same $I \in I(S)$
4. Collect those $x \in \bigcup_i P_i$ lying in the same $I \in I(S)$

We elaborate on how to perform Steps 1 to 4:

Step 1. Since the size of S_i is $r \ (k''(n))$ and the size of S is $r \ (n)$, the number of processors available is sufficient to allocate one to each pair $x \in S$, $I \in I(S_i)$ and each pair $x \in S_i$ and $I \in I(S)$ so that this step can be performed in $O(1)$ time. More precisely, the number of processors available for each S_i is $|P_i|k = r^{b+1}k$ and this is greater than $r^2 \ (n) \ (k''(n))$.
Step 2. Let $I_{i;j} \in I(S_i)$ and let $_{i;j} = |S \setminus I_{i;j}|$. Since S_i is a $(1/r)$-splitter for P_i then $|I_{i;j}| \quad |P_i|/r$. We can allocate $_{i;j}$ processors to each $x \in I_{i;j}$ because

$$\sum_j |I_{i;j}| \ _{i;j} \quad \frac{|P_i|}{r} \sum_j \ _{i;j} \quad \frac{|P_i|}{r}|S| \quad |P_i| \ (n) \quad |P_i|k^{1/2}:$$

The allocation can be performed in time $O(\log\log_k r)$, and then each x can locate itself in $I(S)$ using $O(1)$ time by checking all the possibilities in parallel. Note that there is still a processor advantage of $k^{1/2}$.
Step 3. First, in each $I_{i;j}$, using the $_{i;j}$ processors assigned to each $x \in I_{i;j}$ in the previous step, collect those x lying in the same intervals of S; this takes $O(\log\log_k r)$ time. Then, for each i, collect those $x \in P_i$ in the same intervals of S, using the fact that each interval of S knows from Step 1 which intervals of S_i intersect it; this also takes $O(\log\log_k r)$ time.

[6] Specifically, since the time available for `Locate` is $O(\log\log_k n)$, it is actually sufficient to allocate $_{i;j} = \ (n)$ processors in Steps 2 and 3.

Step 4. Allocate jSj processors to each P_i. Then for each interval of S, using approximate prefix sums, determine the compressed positions of all elements in that interval over all P_i. Then move them to the their final positions. Since $l = n=r^{b+1}$, the processor allocation is possible. This takes $O(\log\log_r n)$ time (there is a processor advantage of r).

Thus, the total time required for `Locate` is $O(\log\log_k n)$. Therefore, we can conclude the following.

Theorem 2. *For some constant* $C > 0$, *a* $(C^{\log\ (\log_k n)}; 1=r)$*-splitter can be computed in time* $O(\log_k r \log\ (\log_k n) + \log\log_k n)$ *using* nk *processors.*

References

1. P. K. Agarwal. Geometric partitioning and its applications. In J. E. Goodman, R. Pollack, and W. Steiger, Eds., *Computational Geometry: Papers from the DIMACS Special Year*. Amer. Math. Soc., 1991.
2. M. Ajtai. Approximate counting with uniform constant depth circuits. In J.-Y. Cai, Ed., *Advances in Computational Complexity Theory,* DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1–20. Amer. Math. Soc., 1993.
3. N. Alon and Y. Azar. The average complexity of deterministic and randomized parallel comparison sorting algorithms. *SIAM. J. Comput.,* 17 (1988) 1178–1192.
4. N. Alon, L. Babai and A. Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *J. Algorithms,* 7 (1986) 567–583.
5. N. Alon, O. Goldreich, J. Hastad and R. Peralta. Simple Constructions of Almost k-wise Independent Random Variables. *Rand. Struct. and Alg.,* 3(1992) 289-304.
6. Y. Azar and U. Vishkin. Tight comparison bounds on the complexity of parallel sorting. *SIAM. J. Comput.,* 16 (1987) 458–464.
7. N.M. Amato, M.T. Goodrich, and E.A. Ramos. Parallel algorithms for higher-dimensional convex hulls. FOCS'94, 683–694.
8. N. M. Amato, M. T. Goodrich and E. A. Ramos. Computing faces in segment and simplex arrangements. STOC'85, 672–682.
9. P. Beame and J. Hastad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM,* 36 (1989) 643–670.
10. M. Bellare and J. Rompel. Randomness-efficient oblivious sampling. FOCS'94, 276-287.
11. R.B. Boppana. The average-case parallel complexity of sorting. *IPL* 33(1989) 145-146.
12. S. Chaudhuri, T. Hagerup and R. Raman. Approximate and exact deterministic parallel selection. Technical Report MPII, MPI-I-93-118, 1993.
13. B. Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete Comput. Geom.*, 9 (1993) 145–158.
14. B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discrete Comput. Geom.*, 10 (1993) 377–409.
15. B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica,* 10 (1990) 229–249.
16. B. Chazelle and J. Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. SODA'93, 281–290.
17. R. Cole. Parallel merge sort. *SIAM J. Comput.,* 17 (1988) 770–785.

18. R. Cole. An optimally efficient selection algorithm. *IPL,* 26 (1988) 295–299.
19. M. L. Fredman, J. Komlos and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM,* 31 (1984) 538–544.
20. T. Goldberg and U. Zwick. Optimal deterministic approximate parallel prefix sums and their applications. In *Proc. 4th IEEE Israel Symp. on Theory of Comput. and Sys*, 220-228, 1995.
21. M.T. Goodrich. Geometric partitioning made easier, even in parallel. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, 73–82, 1993.
22. M.T. Goodrich. Fixed-dimensional parallel linear programming via relative - approximations. SODA'96, 132–141.
23. M. T. Goodrich, Y. Matias and U. Vishkin. Approximate parallel prefix computation and its applications. IPPS'93, 318–325.
24. M. T. Goodrich and E. A. Ramos. Bounded independence derandomization of geometric partitioning with applications to parallel fixed-dimensional linear programming. *Discrete Comput. Geom.*, 18 (1997) 397–420.
25. T. Hagerup and R. Raman. Waste makes haste: tight bounds for loose parallel sorting. FOCS'92, 628–637.
26. T. Hagerup and R. Raman. Fast deterministic approximate and exact parallel sorting. SPAA'93, 346–355.
27. D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete Comput. Geom.,* 2 (1987) 127–151.
28. A. Joffe. On a set of almost deterministic k-independent random variables. *Annals of Probability*, 2 (1974) 161–162.
29. H. Karloff and Y. Mansour. On construction of k–wise independent random variables. STOC'94, 564–573.
30. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15 (1986) 1036–1053.
31. P. MacKenzie and Q. Stout. Ultrafast expected time parallel algorithms. SODA'91, 414–423.
32. J. Matoušek. Approximations and optimal geometric divide-and-conquer. STOC'91, 505-511.
33. J. Matoušek. Cutting hyperplane arrangements. *Discrete Comput. Geom.,* 6(1991) 385-406.
34. K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1993.
35. J. Naor and M. Naor. Small-bias probability spaces: efficient constructions and applications. *SIAM J. Comput.,* 22 (1993) 838–856.
36. S. Rajasekaran and J. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.,* 18 (1989) 594–607.
37. J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. SODA'93, 331–340.
38. V.N. Vapnik and A.Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory Probab. Appl.,* 16(1971) 264-280.

Analyzing an Infinite Parallel Job Allocation Process

Micah Adler[1], Petra Berenbrink[2], and Klaus Schröder[3]

[1] Department of Computer Science, University of Toronto, Canada
micah@cs.toronto.edu [y]
[2] Department of Mathematics and Computer Science, Paderborn University, Germany
pebe@uni-paderborn.de [z]
[3] Heinz Nixdorf Institute and Department of Mathematics and Computer Science, Paderborn University, Germany
ellern@hni.uni-paderborn.de [x]

Abstract. In recent years the task of allocating jobs to servers has been studied with the "balls and bins" abstraction. Results in this area exploit the large decrease in maximum load that can be achieved by allowing each job (ball) a very small amount of choice in choosing its destination server (bin). The scenarios considered can be divided into two categories: *sequential*, where each job can be placed at a server before the next job arrives, and *parallel*, where the jobs arrive in large batches that must be dealt with simultaneously. Another, orthogonal, classification of load balancing scenarios is into *fixed time* and *infinite*. Fixed time processes are only analyzed for an interval of time that is known in advance, and for all such results thus far either the number of rounds or the total expected number of arrivals at each server is a constant. In the infinite case, there is an arrival process and a deletion process that are both defined over an infinite time line.

In this paper, we present an algorithm for allocating jobs arriving in parallel over an infinite time line. While there have been several results for the infinite sequential case, no analogous results exist for the infinite parallel case. We consider the process where m jobs arrive in each round to n servers, and each server is allowed to remove one job per round. We introduce a simple algorithm, where it is sufficient for each job to choose between 2 random servers, that obtains the following result: if $m \quad \frac{n}{6e}$, then for any given round, the probability that any job has to wait more than $O(\log\log n)$ rounds before being processed is at most $1=n$ for any constant . Furthermore, we analyze the distribution on waiting times: with the same probability, the number of jobs of any given round that have to wait $t + c$ rounds to be processed is at most $O(\frac{n}{2^{(2^t)}})$ for a small constant c. These results are comparable with existing results for the infinite sequential case.

[y] Supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada, and by ITRC, an Ontario Centre of Excellence. This research was conducted in part while he was at the Heinz Nixdorf Institute Graduate College, Paderborn, Germany.

[z] Supported by DFG-SFB 376 "Massive Parallelität", and by EU ESPRIT Long Term Reseach Project 20244 (ALCOM-IT).

[x] Supported by the DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik", by DFG-SFB 376 "Massive Parallelität", and by EU ESPRIT Long Term Reseach Project 20244 (ALCOM-IT).

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 417–428, 1998.
c Springer-Verlag Berlin Heidelberg 1998

1 Introduction

Dynamic resource allocation problems have seen much attention in recent years. In these problems, the objective is to distribute a set of jobs across a set of servers as evenly as possible, with a minimum of coordination between jobs and servers. Often, this problem is stated in terms of balls and bins; such occupancy results have several applications in load balancing, hashing, and PRAM simulation [KLM92], [ABKU94], [ACMR95]. A simple distributed algorithm for allocating jobs to servers is to place each job at a random server. This requires no coordination, but it is well known that if there are n jobs and n servers, it is likely that one of the servers receives $O(\frac{\log n}{\log \log n})$ jobs. However, [ABKU94] demonstrated that a small amount of choice in possible servers for each job can lead to as much as an exponential decrease in the maximum imbalance, without prohibitively increasing coordination.

Some of the work in this area [ABKU94,CS97] can be described as *sequential*. In such a scenario, jobs arrive one at a time, and each is placed at a server prior to the arrival of the subsequent job. Results for such a scenario are rather limited in their applicability to parallel and distributed settings, and thus much of the work on allocation processes has concentrated on the *parallel* scenario ([ACMR95,Mit96,St96,BMS97], [Mit97]), where the jobs arrive in large batches that must be processed simultaneously.

The seminal paper on the sequential scenario [ABKU94] considered both a *fixed time* (which they call finite) process, and an *infinite* process. Fixed time processes are only analyzed for a fixed interval of time that is known in advance, where infinite processes consist of a job arrival process and a deletion process that are both defined over an infinite time line. Despite a large subsequent body of work on the parallel scenario as well as further work on infinite processes in the sequential scenario [CS97], up to now there has been no analysis of an infinite process for the parallel scenario. In fact, for all previous parallel results, either the number of rounds or the total expected number of arrivals at each server is a constant. However, most long running systems do in fact process a large number of jobs at each server.

In this paper, we present the first results for jobs arriving in parallel over an infinite time line. We consider a model where m jobs arrive in each synchronized round to n servers, and each server is allowed to complete one job per round. The *waiting time* of a job is the number of rounds the job stays in the system. Our goal is to assign the jobs to the servers in such a way that we minimize both the expected waiting time, as well as the maximum waiting time of the jobs that arrive during any given round.

1.1 New Results

In this paper we introduce and analyze a simple infinite process allocating jobs to servers. We assume that the system starts empty. During each round, every job arriving to the system sends requests to $d \quad 2$ independently and uniformly at random (i.u.r.) chosen servers. Each of the servers stores all of its requesting jobs in a First-in-First-out (FIFO) queue. During each round, the server performs the first job of its queue and deletes the other requests of that job.

We provide bounds on the performance of this process. It is not hard to show that, when $m < \frac{n}{2de}$, the expected waiting time of any job is a constant number of rounds.

Here e is the base of the natural logarithm. It is much more difficult to analyze the entire distribution on the waiting time. The main results of this paper are the following two theorems:

Theorem 1. *Let* c *be* $\frac{2\ +1}{\log(6em/n)}\ +\ 1$ *and* $m\ <\ \frac{n}{3de}$. *For any constant* $\ >\ 0$*:*
Each of the jobs generated in any fixed round T_j *has waiting time smaller than* $\frac{\log\log n}{\log d}\ +$ $2\ \ +\ c$ *with probability at least* $1\ -\ \frac{1}{n}$.

Note that this result implies the stability of the considered process: the number of jobs in the system does not grow to infinity.

Theorem 2. *For* $c\ \ \ 6\ =\log\ \frac{n}{6e\ m}\ +\ 1, \max f5; \log(12\ \ ); cg\ \ \ t^0\ \ \log\log n\ -\ 3$,
and $m\ \ \ \frac{n}{6de^{1\ +\ 1=e}}$ *it holds for any constant* $\ >\ 0$*:*
At most $\frac{n}{2^{(d\ t^0)}}$ *of the* m *jobs generated in any fixed round* T_j *have waiting time* $t\ \ \ t^0 + c$
with probability at least $1\ -\ \frac{1}{n}$.

Here, and throughout the paper, we say *with high probability* (w.h.p.) to denote with probability at least $1\ -\ \frac{1}{n}$ for any constant $\ >\ 0$. It is easy to show that the same result holds if the jobs are generated by n generators which can be arbitrarily distributed over the processors (see [SV96]). Each generator is allowed to produce a job with a probability smaller than $\frac{1}{2de}$ per round. It is also possible to use generators with different generation probabilities if the expected overall generated load is smaller than $\frac{1}{2de}$. Our results also hold if the jobs are generated asynchronously, for instance by a Poisson Process.

The analysis of the process is performed using a type of delay sequence argument, using a structure known as a witness tree [MSS95]. To use this type of argument, we first show that every time the process fails a witness tree must exist. We then show that that it is very unlikely for a witness tree to occur. Usually the latter involves enumerating the set of possible witness trees and then bounding the probability that a given witness tree exists. In the infinite case analyzed here, this simple enumeration technique is impossible, since an unbounded number of jobs may appear, and the size of the witness trees that can exist cannot be bounded. We develop a method for using a witness tree argument in the analysis of an infinite process; this is the main technical contribution of our paper.

1.2 Previous Work

Azar et al. [ABKU94] examine a sequential protocol called *greedy process* to place n balls into n bins. For each ball they choose d bins i.u.r. and put the ball into the bin with minimum load at the time of placement. They show that after placing n balls the maximum load is $(\log\log(n)/\log(d)\ +\ 1)$, w.h.p. Furthermore, they provide a result for an infinite version of their sequential process: in the stationary distribution, the fullest bin contains less than $\log\log(n)/\log(d)\ +\ O(1)$ balls, w.h.p., where n is both the number of balls and bins in the system. The simple sequential game has many applications and is also used as an online algorithm for competitive Load Balancing (see [ABK94], [AKP$^+$93], and [PW93]). Recently, Czumaj et al. ([CS97]) extended

the results in several directions. They present an adaptive process where the number of choices made in order to place a ball depends on the load of the previously chosen bins, and an off-line allocation process knowing the random choices in advance. Then, they consider a scenario allowing reassignments: During the allocation of a new ball one may ask for some number of possible locations and then arbitrarily distribute all balls (together with the new one) among the chosen bins. They show that a process with reassignments yields a maximum load that is never smaller by more than an constant factor than the maximum load of the process from [ABKU94].

Adler et al. [ACMR95] explore the problem in parallel and distributed settings for the case of placing n balls into n bins. They provide a lower bound for *non-adaptive* (possible destinations are chosen before any communication takes place) and *symmetric* algorithms (all balls and bins perform the same underlying algorithm, and all destinations are chosen i.u.r.). For any constant number $r \in \mathbb{N}$ of communication rounds, the maximum load is shown to be at least $\Omega\left(\sqrt[r]{\frac{\log n}{\log\log n}}\right)$. Additionally, they present parallelizations of the sequential strategy found in [ABKU94]. They give a two-round parallelization of the greedy process, matching the lower bound. Furthermore, they introduce a multiple-round strategy requiring $\log\log n + O(1)$ rounds of communication and achieving maximum load $\log\log n + O(1)$, w.h.p. Finally, they examine a strategy using a threshold T: In each of r communication rounds each non-allocated ball tries to access two bins chosen i.u.r. and each bin accepts up to T balls during each round. They show that with $T = \sqrt[r]{\frac{(2r+o(1))\log n}{\log\log n}}$ this algorithm terminates after r rounds with maximum load $r \cdot T$, w.h.p.

Stemann [St96] extends the results for the case where the number of balls m is larger than the number n of bins. For $m = n$, he analyzes a very simple class of algorithms achieving maximum load $O\left(\sqrt[r]{\frac{\log n}{\log\log n}}\right)$ if r rounds of communication are allowed. This matches the lower bound presented in [ACMR95]. He generalizes the algorithm for $m > n$ balls and achieves the optimal load of $O(\frac{m}{n})$ using $\frac{\log\log n}{\log(m/n)}$ rounds of communication, w.h.p., or load $\max\left\{\sqrt[r]{\log n}, O\left(\frac{m}{n}\right)\right\}$ using r rounds of communication, w.h.p.

In [BMS97] the authors extend the lower bound of [ACMR95] to arbitrary $r \leq \log\log n$, implying that the result of Stemanns protocol is optimal for all r. Their main result is a generalization of Stemanns upper bound to weighted jobs: Let W^A (W^M) denote the average (maximum) weight of the balls and $\alpha = W^A/W^M$. The authors present a protocol which achieves maximum load of $\gamma\left(\frac{m}{n} W^A + W^M\right)$ using $O\left(\frac{\log\log n}{\log(\gamma((m/n)\alpha+1))}\right)$ rounds of communication. In particular, for $\log\log n$ rounds the optimal load of $O\left(\frac{m}{n} W^A + W^M\right)$ is achieved.

Mitzenmacher [Mit96] analyzes a dynamic but fixed time allocation strategy: Customers (balls) arrive as a Poisson stream of rate $n\lambda$, $\lambda < 1$, at a collection of n servers (bins). Each customer chooses d servers i.u.r. and joins the server with the fewest customers. Customers are served according to the First-Come-First-Serve protocol, and the service time is exponentially distributed with mean one. He calls his model the supermarket model. For a time interval of fixed and constant length T, he shows the expected waiting time to be $O(1)$ for $N \to \infty$, and the maximum queue length to be

$O(\log\log n + o(1))$, w.h.p. His analysis makes use of deep results of Kurtz ([Kur81]) on so called density dependent Markov Chains. In [Mit97] the author extends his results to several different load generation and consumption schemes. For example, he analyzes the same process with constant service times, the customers having a different number of choices, and bounded queue lengths. However, the results of Mitzenmacher are only valid in the case of time intervals of constant length.

2 The Infinite Allocation Process

During each round, m jobs enter the system; the running time of the jobs is equal to the duration of one round. Each job J sends a request to d i.u.r. chosen servers $B_1(J), \ldots, B_d(J)$. We say these d servers *hold a request* of J, and $B_i(J)$ is called the i-th request of J. Each request of job J is marked with a unique job identifier, the number of the round J enters the system (this round is called *entry round* of J in the following), and a list of all d servers holding a request of J.

The server stores all incoming requests in a First-in-First-out (FIFO) queue, jobs belonging to the same round are stored in arbitrary order. During each round, each non-empty server deletes the first request of its FIFO-queue and performs the work for that job. Additionally, each server sends a *deletion message* to the other servers holding a request of the finished job, and each server deletes the requests corresponding to a deletion message from its queue. The infinite allocation process is given in Figure 1.

Infinite Allocation Process

For all jobs J_i generated during round t **do in parallel**
-Choose i.u.r. d servers $B_1(J_i), \ldots, B_d(J_i)$.
-Send a request $(t, J_i, B_1(J_i), \ldots, B_d(J_i))$ to each chosen server.
For all servers B_j **do in parallel**
-Insert incoming requests in an arbitrary order into the FIFO queue.
-Delete the first job J_{B_j} from the queue and send a deletion message to each server holding a request of J_{B_j}.
-Finish the work on the job.

Fig. 1. The Infinite Allocation Process

3 Bounding the Maximum Waiting Time

In this section we prove Theorem 1 estimating the waiting time of a job entering the system in any fixed round T_j. For ease of presentation we state the proof only for the case $d = 2$, the proof for $d \geq 2$ appears in a full version of this paper.

Job J^0 *weakly-blocks* job J in round T, if J and J^0 send requests to a server B, and B performs J^0 during round T. If J^0 weakly-blocks J and J is performed in round $T+1$ (maybe by another server than B), J^0 is said to *block* J. We make use of a *delay tree* built of several *delay sequences*. A delay sequence consists of a sequence of jobs, each job (weakly-) blocks his predecessor in the sequence. A delay tree is a (not necessarily

complete) 2-ary tree whose nodes represent jobs. The jobs on any branch from the root to a leaf form a delay sequence. We show that a delay tree with depth $t + 1$ has to exist if there is a job J with waiting time at least t. The root of the tree represents J and the branches of the tree represent delay sequences, i.e. a sequence of jobs which have blocked their predecessors respectively. We show that it is very unlikely that such a delay tree exists.

Let J be a job generated during round T_j which is not performed after round $T = T_j + t$ with $t = \log\log n + 2 \quad + c$. In round T, J is weakly-blocked by two jobs J_1 and J_2. J_1 and J_2 are generated not later than round T_j and wait at least $t - 1$ steps to be performed (according to the FIFO-rule used by the servers to choose one of their jobs). In turn, J_1 (J_2) is blocked by two jobs J_1^1 and J_1^2 (J_2^1 and J_2^2) during round $T - 1$. (Since J_1 and J_2 are performed in round T they are *blocked* rather than being only *weakly-blocked*.) Clearly, J_1^1, J_1^2, J_2^1 and J_2^2 wait at least $t - 2$ steps to be performed. Thus, the construction can be carried on until jobs without waiting time are reached, these jobs are performed in the same round as they have been generated. Note that a blocked job J performed at server B is in general blocked by two jobs $J_1; J_2$ (J_1 at server $B_1(J)$ and J_2 at $B_2(J)$). Clearly, we have $B_1(J) = B$ or $B_2(J) = B$. To simplify our further discussion we assume in the following that the left child of v represents $B_1(J)$ if $B_1(J) = B$, and $B_2(J)$, otherwise.

The construction above yields a delay tree D_t of height at least $t + 1$, the root V of the tree represents the job J, the two children of the root represent J_1 and J_2. The children of J_1 (J_2) represent J_1^1 and J_1^2 (J_2^1 and J_2^2), and so on. Thus, the nodes of D_t represent jobs blocking the job represented by their predecessor. Each branch of a delay tree D_t starting at the root has length at least t. This holds as each server with a non-empty queue performs a request and J has waiting time at least t.

As a consequence of the construction above, we state Lemma 1.

Lemma 1. *If a job J has waiting time at least t rounds, there exists a delay tree D_t with root J.*

In the sequel we use a recursive decomposition of D_t into delay sequences different from the delay sequences used to define the delay tree. The first delay sequence S is defined by the branch P from the root to the leftmost leaf of D_t. Remove P's edges from D_t. Then D_t becomes a forest of trees, cf. Figure 2. The root of each resulting tree has degree one and is a node of P. We apply the construction recursively to these trees: The branch P^0 from the root of a new tree to the leftmost leaf of that tree defines a delay sequence (see dotted lines in Figure 2). The edges of P^0 are removed generating a new forest to which the construction is applied again. This decomposition of the delay tree into delay sequences is called *regular decomposition*. The delay sequences of a regular decomposition are called *regular delay sequences*. In each regular delay sequence except the one starting at the root, the first edge is a right edge of D_t, and the other ones are left edges. Thus, the jobs of one regular delay sequence are all blocked in the same server.

Note that the regular delay sequences share some nodes as a result of the construction. Let the up-most node v of the branch defining a regular delay sequence S be called *top node* of S, and let the first job of S be the *top job* of S.

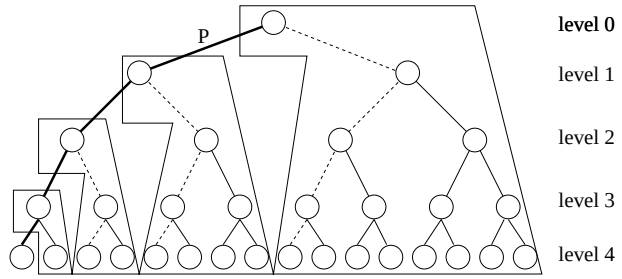


Fig. 2. recursive decomposition into delay trees and sequences

Let the length of a delay sequence S be the number of jobs in S minus one – i.e. we do not count the top job of S. Consider the set V of level $\log\log n + 2$ nodes of D_t. Each node $v \in V$ is on a path defining a regular delay sequence S_v of length at least c (recall that D_t has at least $\log\log n + 2 + c$ complete levels). We call the regular delay sequences S_v with $v \in V$ *long delay sequences*. In the following we only examine the tree consisting of these long delay sequences.

Observation *There are* $2^{\log\log n + 2} = 2^2 \log n$ *long delay sequences.*

The next Lemma summarizes some results concerning the delay trees and sequences built by the decomposition of a delay tree.

Lemma 2.

1. *A job occurring twice in a delay tree is represented by nodes of the same level of the tree, and jobs represented by nodes of a single delay sequence are distinct from each other.*
2. *The entry rounds of the jobs represented by a delay sequence are non-increasing, and the last job of a long delay sequence is performed in the same round it enters the system.*

Proof. 1. The nodes of level $i > 0$ of the delay tree represent jobs which are performed by a server during round $T - i + 1$, $1 \quad i \quad T$. If a job is performed by a server, its other requests are deleted in the same round. Thus, it is not possible that a job is performed twice during distinct rounds. This entails the fact that jobs belonging to a delay sequence are distinct from each other.

2. The entry rounds of jobs represented by a delay sequence are non-increasing because each job blocks the job represented by its predecessor, and the jobs are performed following the FIFO-rule. The construction of a branch in the delay tree ends reaching a job J^0 which is not blocked by another job. Thus, J^0 enters an empty FIFO queue and is performed in the same round as it enters the system.

We call two delay sequences S_1 and S_2 *pairwise distinct*, if every job of S_1 is either the top job of S_1 or S_2, or does not appear in S_2. The rest of the proof is divided into several Lemmas. We know that there exists a delay tree D_t with root J if a job J has waiting time larger than t rounds. Then, either job J is the root of a delay tree with

$\log n$ pairwise distinct long delay sequences, or it is the root of a delay tree with less than $\log n$ pairwise distinct long delay sequences. We show that both events are very unlikely. In order to estimate the probability that D_t consists of less than $\log n$ pairwise distinct long delay sequences, we have to upper bound the number of nodes of D_t. The next lemma upper bounds the probability that there exists a delay sequence S of length l with a fixed top job. The fixed top job will help us to estimate an upper bound for the range of the possible entry rounds of the jobs involved in S.

Lemma 3. *Let J_0 be the top job of a delay sequence S and T_0 be the entry round of J_0. Then it holds: The expected number of delay sequences with top job J_0 and length l is at most $\left\lceil \frac{6em}{n} \right\rceil^l$.*

Proof. Let $J_0; J_1; J_2; \ldots; J_l$ be the sequence of jobs of S, i.e. J_i is blocked by job J_{i+1}. Let T_i be the entry round of the i-th job, let $T_i \quad T_i$ be the round where J_i is performed, and let $= T_0 - T_l$. There are $\binom{+l-1}{l-1}$ possible ways to choose the $l-1$ entry rounds of the jobs in a way that they are not increasing. If the entry rounds of the jobs are fixed, there are at most m^l ways to choose l jobs of the delay sequence. Thus, the number of possible delay sequences is bounded by

$$\binom{+l-1}{l-1} m^l \quad \binom{+l}{l} m^l \quad \frac{(\quad +l)\ e}{l}^l m^l:$$

Next, we bound , the range of the entry rounds of the considered jobs. J_i is performed in round $T_0 - i$, and thus $T_l = T_l = T_0 - l$. Hence $= T_0 - T_l \quad T_0 - T_l = l$, and it follows that

$$\frac{(\quad +l)\ e}{l}^l m^l \quad \frac{2e\ l}{l}^l m^l = (2e\ m)^l:$$

To bound the probability that there exists such a delay sequence we first choose whether the first or the second request of a job leads to the blocking. This yields 2^l possibilities. The probability that the chosen requests of J_i and J_{i+1}, $1 \quad i < l$ go to the same server is bounded by $\frac{1}{n - r\ log \log n}^l$ for a constant r. This holds because the number of servers involved in a delay sequence can be bounded by $r \quad \log\log n$ (note: the servers which are involved in the delay sequences can only change in the first $\log\log n + 2$ levels of the tree). Thus, the probability that there exists a delay sequence of length l can be bounded by

$$(2e\ m)^l \quad \frac{2}{n - r\ \log\log n}^l = \frac{6e\ m}{n}^l:$$

The following Corollary is a consequence of Lemma 3.

Corollary 1. *Let $\left\lceil \frac{6em}{n} \right\rceil < 1$; let J_0 be a job. Then the expected number of delay sequences S with top job J_0 and length at least l is at most $\left\lceil \frac{6em}{n} \right\rceil^{l-1}$.*

Note that in Lemma 3 and Corollary 1 the expected number of delay sequences is also an upper bound for the probability that such a sequence does exist. In the next lemma we prove that the number of nodes in the long delay sequences is bounded by $O(\log^2 n)$, w.h.p. This result will be used to bound the number of jobs occurring several times in the tree.

Lemma 4. *Let $r \in \mathbb{N}$ be large enough. The total number of jobs contained in long delay sequences of D_t is at most $r \log^2 n$, w.h.p.*

Proof. First we show that each long delay sequence of D_t has length at most $r' \log n$ for a constant r', w.h.p. (r and r' are dependent on). If one of the long delay sequences S of the delay tree has length at least $r' \log n$, there exists a lengthened delay sequence consisting of S and the path from the top node of S to the root. This delay sequence S' also has length at least $r' \log n$.

By using Corollary 1 we show that the probability that S' has length at least $r' \log n$ is bounded by $\left(\frac{6em}{n}\right)^{r' \log n - 1}$. Thus, the probability that any of the lengthened delay sequence and, consequently, any of the long delay sequences has length larger than $r' \log n$ can be bounded by

$$2^{\log\log n + 2} \left(\frac{6em}{n}\right)^{r' \log n - 1} \leq n^2 \left(\frac{6em}{n}\right)^{r' \log n - 1} \leq \left(\frac{1}{n}\right)^{\cdot},$$

for suitably chosen but constant r'. Since there are $2^{\log\log n + 2}$ long delay sequences, the total number of jobs contained in long delay sequences is at most $2^2 \log n \, r' \log n \leq r \log^2 n$, for a constant $r \in \mathbb{N}$.

Now we estimate the probability that the long delay sequences of D_t are not pairwise distinct.

Lemma 5. *The delay tree D_t that consists of at most $r \log^2 n$ nodes contains at least $\log n$ pairwise distinct long delay sequences, w.h.p.*

Proof. Consider a long delay sequence S whose top node is not the root of D_t. Let S be a subset of the long delay sequences of D_t, with $S \neq S$. Let J be the set of jobs contained in a long delay sequence $S' \in S$, let B be the set of servers defined by the long delay sequences S' – i.e. if $J'_i \in S'$ is blocked by $J'_{i+1} \in S'$ at server B'_{i+1}, then $B'_{i+1} \in B$. Let v be the top node of S. Node v represents a job J_0, and we assume that v is the only node of D_t representing J_0 served at a server B_0. The right child of v represents job J_1 blocking J_0 at a server B_1. Due to the construction of the delay tree, B_1 is a server chosen i.u.r. Thus the probability that $B_1 \in B$ is at most $\frac{r \log^2 n}{n}$. If $B_1 \notin B$, the probability that any $J \in J$ sends a request to B_1 is at most $\frac{2r \log^2 n}{n}$. On account of the construction, S contains only requests sent to B_1. Thus, the total probability that $J_i \in J$ for at least one $J_i \in S$, $J_i \neq J_0$ is at most $\frac{3r \log^2 n}{n}$.

For each node v of level $0, \ldots, \log n + 2$ of D_t let S_v be defined as the long delay sequence with top node v. If P_v is the path from v to the root of D_t, then the delay sequences S_w with $w \in P_v$, are called the *ancestors* of S_v. Now consider a breadth first ordering $\{v_1, v_2, \ldots\}$ of the nodes of D_t. This ordering defines an ordering $S_1 = S_{v_1}, S_2 = S_{v_2}, \ldots$ of the long delay sequences. Let $S_1 = \{S_1\}$, and let $S_i = S_{i-1} \cup \{S_i\}$, if each delay sequence in S_{i-1} is pairwise distinct of S_i, and all ancestors of S_i are in S_{i-1}. Otherwise let $S_i = S_{i-1}$. If S_i is *not* pairwise distinct of all delay sequences in S_{i-1} but all ancestors of S_i are in S_{i-1}, we call S_i a *first order delete*. As stated above, the probability for S_i being a first order delete is at most $\frac{3r \log^2 n}{n}$. There

are $k = 2^2 \log n$ long delay sequences in D_t. Thus, for n large enough, the probability that at least 2 long delay sequences S_i are first order deletes is at most

$$\binom{k}{2}\left(\frac{3r\log^2 n}{n}\right)^2 \le 2^{2\log n}\left(\frac{3r\log^2 n}{n}\right)^2 \le \frac{1}{n}.$$

Let each level of D_t containing a top node of a first order delete be called *faulty level*. As we have already shown, with high probability there are at most 2 faulty levels in D_t. In each faulty level, there is at least one node who is top node of a first order delete. In each non-faulty level, the number of delay sequences in S_k doubles. In the sequel we assume that S_k does not contain delay sequences with top nodes in a faulty level, or heaving an ancestor with a top node in a faulty level. Since D_t contains at least $\log\log n$ non-faulty levels, w.h.p., there are at least $\log n$ long delay sequences in S_k and thus, there are at least $\log n$ pairwise distinct long delay sequences.

Lemma 6. *Let J be a job entering during round T_j. There exists no delay tree D_t with root representing job J containing at least $\log n$ pairwise distinct long delay sequences, w.h.p.*

Proof. We bound the expected number of delay trees. According to Lemma 1 there exists a delay tree D_t if job J entering in round T_j has waiting time at least $t = \log\log n + 2 + c$ rounds. Lemma 4 ensures that D_t contains at least $\log n$ pairwise different delay sequences of length at least c, w.h.p. Furthermore, the number of servers involved in the whole tree can be bounded by $r \log^2 n$. The expected number of delay sequences with fixed top jobs and length at least c can be bounded by

$$\left(\frac{2}{n - r\log^2 n}\right)^{c-1} \left(\frac{6em}{n}\right)^{c-1}.$$

Thus we first have to "fix" the top job of each delay sequence.

If v is the root of D_t, the top job of S_v is J, which is already fixed. If v' is a node of the branch from the root of D_t to the leftmost leaf of D_t, then the top job of $S_{v'}$ is fixed once S_v has been fixed. In general, the top job of S_i (cf. proof of Lemma 5) is fixed, when the jobs of the delay sequences in S_{i-1} are fixed, or: the expected number of ways to fill the delay sequences in S_{i-1} upper bounds the expected number of possible top jobs in S_i. Since S_k does not contain all long delay sequences, we first have to choose S_k among the set of all long delay sequences. There are at most $\log\log n + 2$ levels in D_t possibly containing the top node of a first order delete. Thus, there are at most $\binom{\log\log n + 2}{2}$ ways to choose the faulty levels. The expected number of delay trees containing at least $\log n$ pairwise distinct long delay sequences is at most

$$\binom{\log\log n + 2}{2}\left(\frac{6em}{n}\right)^{(c-1)\log n}(2\log\log n)^2 \le n^{-2} \le n^{-1}$$

if $c \ge \frac{2}{\log\left(\frac{6em}{n}\right)} + 1$.

Since $m \ge n$ jobs enter the system per round, with probability at least $\frac{1}{n}$ there is no delay tree D_t whose root represents a job entering the system in a given round. This ends the proof of Theorem 1.

4 Distribution of the Waiting Time

In this section, we sketch the proof of Theorem 2 specifying the distribution of the waiting time of the jobs generated in any fixed round T_j.

We assume the worst case $m = \frac{n}{4e}$. As in Section 3, we can show that a set $D_t = \{D_t^1 ::: D_t^k\}$ of $k = \frac{n}{2^{2^{t^0}}}$ delay trees of depth at least $t = t^0 + c$ per branch, and roots representing jobs generated in round T_j have to exist if k of the jobs generated in round T_j have a waiting time of at least t. We can easily show a Lemma that is related to Lemma 1. We state it without any explicit proof:

Lemma 7. *If $n=2^{2^{t^0}}$ jobs generated in round T_j have a waiting time of a length greater than $t = t^0 + c$ rounds, there exists $n=2^{2^{t^0}}$ delay trees with roots representing jobs generated during round T_j.*

As already described in Section 3, every tree $D_t^i \in D_t$ can be regularly decomposed into regular delay sequences. Furthermore, each node of level t^0 defines a regular delay sequence of length at least c. There are 2^{t^0} of these long delay sequences per tree. In the following we call the first t^0 levels of the delay trees their *upper parts*. The next Lemma shows that at most half of the upper parts of the delay trees share a job with one of the other upper parts of the trees, i.e. at least half of the upper parts of the trees represent completely different jobs. We say a job J^0 *occurs only once* in D_t, if there is at most one node in one of the trees $\{D_t^1 ::: D_t^k\}$ representing J^0. The proof can be done according to the proof of Lemma 5.

Lemma 8. *At least $n=2 \; 2^{2^{t^0}}$ of the upper parts of the delay trees in $D^0{}_t$ represent jobs occurring only once in $D^0{}_t$, w.h.p.*

It remains to show that $n=2^{2^{t^0}}$ delay trees of height $t^0 + c$ do not occur, w.h.p. Each of the above delay trees induces 2^{t^0} long delay sequences with fixed top jobs as defined in Section 3. Furthermore, the servers belonging to the Jobs of level t^0 of the trees are pairwise different from each other. The next Lemma bounds the expected number of occurrences of $n \; 2^{t^0}=2 \; 2^{2^{t^0}}$ long delay sequences with fixed top jobs. Due to space limitations we present it without proof.

Lemma 9. *The expected number of occurrences of $n \; 2^{t^0}=(2 \; 2^{2^{t^0}})$ long delay sequences with fixed top jobs is at most $\left(\frac{4e\,m}{n}\right)^{c-1} + \frac{2^{t^0+1}}{2^{2^{t^0}}}$ $n\,2^{t^0}=2\,2^{2^{t^0}}$.*

The next Lemma bounds the the probability that $n=2^{2^{t^0}}$ Delay trees of height $t^0 + c$ occur. The proof can be done according to Lemma 6.

Lemma 10. *There exists no set $D_t = \{$ of $\frac{n}{2^{2^{t^0}}}$ delay trees with roots representing jobs generated during round T_j, w.h.p.*

References

ABK94. Yossi Azar, Andrei Z. Broder, and Anna R. Karlin. On-line load balancing. *Theoretical Computer Science*, 130:73–84, 1994. A preliminary version appeared in FOCS 92.

ABKU94. Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 593–602, New York, 1994. ACM, ACM Press.

ACMR95. Micah Adler, Soumen Chakrabarti, Michael Mitzenmacher, and Lars Rasmussen. Parallel randomized load balancing. In *Proceedings of the 27th ACM Symposium on Theory of Computing*, pages 238–247, New York, USA, 1995. ACM, ACM Press.

AKP+93. Yossi Azar, Bala Kakyanasundaram, Serge Plotkin, Kirk R. Pruhs, and Orli Waarts. On-line load balancing of temporary tasks. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, pages 119–130, 1993.

BMS97. Petra Berenbrink, Friedhelm Meyer auf der Heide, and Klaus Schröder. Allocating weighted jobs in parallel. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 302–310, 1997.

CS97. Artur Czumaj and Volker Stemann. Randomized allocation processes. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 194–203, Miami Beach, FL, 1997. IEEE Computer Society Press, Los Alamitos.

KLM92. Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, 1992.

Kur81. Thomas G. Kurtz. *Approximation of Population Processes*. Regional Conference Series in Applied Mathematics. CMBS-NSF, 1981.

Mit96. Michael Mitzenmacher. Density dependent jump markov processes and applications to load balancing. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, pages 213–223, 1996.

Mit97. Michael Mitzenmacher. On the analysis of randomized load balancing schemes. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 292–301, Newport, Rhode Island, 1997.

MSS95. Friedhelm Meyer auf der Heide, Christian Scheideler, and Volker Stemann. Exploiting storage redundancy to speed up randomized shared memory simulations. In *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science*, 1995.

PW93. S. Phillips and J. Westbrook. Online load balancing and network flow. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 402–411. ACM, 1993.

SV96. Christian Scheideler and Berthold Voecking. Continuous Routing Strategies. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1996.

St96. Volker Stemann. Parallel balanced allocations. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–269, New York, USA, 1996. ACM.

Nearest Neighbor Load Balancing on Graphs ?

Ralf Diekmann[1], Andreas Frommer[2], and Burkhard Monien[1]

[1] Department of Computer Science, University of Paderborn,
Fürstenallee 11, D-33102 Paderborn, Germany
fdiek, bmg@uni-paderborn.de
[2] Department of Mathematics and Institute for Applied Computer Science
University of Wuppertal, D-42097 Wuppertal, Germany
frommer@math.uni-wuppertal.de

Abstract. We design a general mathematical framework to analyze the properties of nearest neighbor balancing algorithms of the diffusion type. Within this framework we develop a new *optimal polynomial scheme* (OPS) which we show to terminate within a finite number m of steps, where m only depends on the graph and not on the initial load distribution.
We show that all existing diffusion load balancing algorithms, including OPS determine a flow of load on the edges of the graph which is uniquely defined, independent of the method and minimal in the l_2-norm. This result can also be extended to edge weighted graphs.
The l_2-minimality is achieved only if a diffusion algorithm is used as preprocessing and the real movement of load is performed in a second step. Thus, it is advisable to split the balancing process into the two steps of first determining a balancing flow and afterwards moving the load. We introduce the problem of scheduling a flow and present some first results on the approximation quality of local greedy heuristics.

1 Introduction

We consider the following abstract distributed load balancing problem. We are given an arbitrary, undirected, connected graph $G = (V, E)$ in which node v_i 2 V contains a number w_i of unit-sized tokens. Our goal is to determine a schedule to move tokens across edges so that finally, the weight on each node is (approximately) equal. In each step we are allowed to move any number of tokens from a node to each of its neighbors in G. Communication between non-adjacent nodes is not allowed. We assume that the situation is fixed, i.e. no load is generated or consumed during the balancing process, and the graph G does not change.

This problem describes load balancing in synchronous distributed processor networks and parallel machines when we associate a node with a processor, an edge with a communication link of unbounded capacity between two processors, and the tokens with identical, independent tasks [4]. It also models load balancing in parallel adaptive finite element simulations where a geometric space,

? Partly supported by the DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen" and the EC ESPRIT Long Term Research Project 20244 (ALCOM-IT).

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 429{440, 1998.
c Springer-Verlag Berlin Heidelberg 1998

discretized using a mesh, is partitioned into sub–regions and the computation proceeds on mesh elements in each sub–region independently [7,9]; here we associate a node with a mesh region, an edge with the geometric adjacency between two regions, and tokens with mesh elements in each region. As the computation proceeds, the mesh refines/coarsens depending on problem characteristics such as turbulence or shocks (in the case of fluid dynamics simulations, for example) and the size of the sub–regions (in terms of numbers of elements) has to be balanced. Because elements have to reside in their geometric adjacency, they can only be moved between adjacent mesh regions, i.e. via edges of the graph [7]. The problem of parallel finite element simulation has been extensively studied – see the book [9] for a selection of applications, case studies and references.

Scalable algorithms for our load balancing problem operate locally on the nodes of the graph. They iteratively balance the load of a node with its neighbors until the whole network is globally balanced. The class of local iterative load balancing algorithms distinguishes between *diffusion* [2,4] and *dimension exchange* [4,17] iterations which mainly differ in the model of communication they are based on. Diffusion algorithms assume that a node of the graph is able to send and receive messages to/from all its neighbors simultaneously, whereas dimension exchange uses only pairwise communication, iteratively balancing with one neighbor after the other. Throughout this work we focus on diffusive schemes, i.e. we assume that nodes are able to communicate via all their edges simultaneously.

The *quality* of a balancing algorithm can be measured in terms of numbers of steps it requires to reach a balanced state and in terms of the amount of load moved over the edges of the graph. Recently, diffusive algorithms gained some new attention [6,7,10,13,14,16,17]. The original algorithm described by Cybenko [4] and, independently, by Boillat [2] lacks in performance because of its very slow convergence to the balanced state. Ghosh et al. use the idea of over-relaxation – a standard technique in numerical linear algebra – to speed up the iteration process by an order of magnitude [10]. We will see in the following that other, more advanced techniques from numerical linear algebra can be used to develop local iterative methods showing an improved – and in a sense even optimal – convergence behavior.

Hu and Blake investigate the flow of load via the edges of the graph and propose a non-local method to determine a *balancing flow* which is minimal in the l_2-norm [13]. From experimental observations they conjecture that the local diffusion iteration of Cybenko also ends up with an l_2-minimal flow. We will see in the following that this is indeed the case, i.e. we give a mathematical proof that all local iterative diffusion algorithms including our new optimal OPS scheme determine a balancing flow which is l_2-minimal, uniquely defined and independent of the method and the parameters used.

Some of the more theoretical papers dealing with diffusion algorithms suggest to move the load directly as the iteration proceeds [4,10]. This one-phase approach usually moves load items back and forth over the edges as the iteration proceeds. Thus, the resulting flow of load is by far not l_2-optimal. In

practise therefore, the diffusion iteration is used as preprocessing just to determine the balancing flow. The real movement of load is performed in a second phase [7,13,14,16]. In addition, this two-phase approach has the advantage of avoiding any problems with adopting the local iterative algorithms to integral values (like it is, for example, done in [10]).

The movement of load has to be scheduled in such a way that each node does not send more load than it possesses in a certain step. Using experiments, we will see that simple greedy heuristics like they are used in practical applications like e.g. [16] allow to finish the load movement in much less steps than taken by the fastest diffusion algorithm. Interestingly, this *flow scheduling problem* appears to be un-studied up to now, so we introduce it here together with some first theoretical results.

The main contributions of this paper are summarized as follows:

- Based on matrix polynomials we develop a general mathematical framework to analyze the convergence behavior of existing diffusion type methods. Within this framework, we develop an *Optimal Polynomial Scheme* (OPS) which determines a balancing flow within m steps if m is the number of distinct eigenvalues of the graph. The OPS algorithm makes use of the full set of eigenvalues which can be computed in a preprocessing step. Information about the initial load distribution is not necessary.
- We consider the quality of the balancing flows determined by local iterative diffusion algorithms. We show that all algorithms end up with the same flow of load which is optimal in the l_2-norm, provided the diffusion matrix is a scaled and shifted version of the Laplacian.
- We introduce the *Flow Scheduling Problem* and show that certain local greedy heuristics for this problem are $\Theta(\sqrt{n})$-optimal.

Map: The next section gives some basic definitions and notations. Section 3 develops the general framework for analyzing nearest neighbor schemes and presents the new optimal method. Section 4 shows that the methods considered here all find l_2-minimal flows of load. Section 5 deals with the flow scheduling problem and, finally, Section 6 shows results of some simulations.

2 Basic Definitions and Notations

Let $G = (V, E)$ be a connected, undirected graph with $|V| = n$ nodes and $|E| = N$ edges. Let $w_i \in \mathbb{R}$ be the *load* of node $v_i \in V$ and $w \in \mathbb{R}^n$ be the vector of load values. The vector $\overline{w} := \frac{\lambda}{n}(1, \ldots, 1)$ with $\lambda = \sum_{i=1}^{n} w_i$ denotes the corresponding vector of average load.

Define $A \in \{-1, 0, 1\}^{n \times N}$ to be the node-edge *incidence matrix* of G. A contains a row for each node and a column for each edge. Each column has exactly two non-zero entries – a "1" and a "-1" – according to the two nodes incident to the corresponding edge. The signs of these non-zeros (implicitly) define directions for the edges of G. These directions will later on be used to express the direction of the flow.

Let $B \in \{0,1\}^{n \times n}$ be the *adjacency matrix* of G. For some of our constructions we need the *Laplacian* $L \in \mathbb{Z}^{n \times n}$ of G defined as $L := F - B$, where $F \in \mathbb{N}^{n \times n}$ contains the node degrees as diagonal entries and 0 elsewhere. It is not difficult to see that $L = AA^T$.

Let $x \in \mathbb{R}^N$ be a flow on the edges of G. The direction of the flow is given by the signs of the entries of x in conjunction with the directions in A. x is called a *balancing flow* on G iff

$$Ax = w - \overline{w}. \tag{1}$$

Equation (1) expresses the fact that the flow balance at each node corresponds to the difference between its initial load and the mean load value, i.e. after shipping exactly x_e tokens via each edge $e \in E$, the load is globally balanced.

Among the set of possible flows which fulfill (1) we are interested in such x achieving certain quality criterions. We especially look at balancing flows x with minimal l_2-norm $\|x\|_2$ defined as $\|x\|_2^2 = \sum_{i=1}^{N} x_i^2$.

By *local iterative balancing algorithms* we denote a class of methods performing iterations on the nodes of G which require communication with adjacent nodes only. The simplest of these methods performs on each node $v_i \in V$ the iteration

$$\begin{aligned} &\forall e = \{v_i, v_j\} \in E: \; y_e^{k-1} = \alpha_e (w_i^{k-1} - w_j^{k-1}); \quad x_e^k = x_e^{k-1} + y_e^{k-1}; \\ &\text{and} \quad w_i^k = w_i^{k-1} - \sum_{e = \{v_i, v_j\} \in E} y_e^{k-1} \end{aligned} \tag{2}$$

Here, y_e^k is the amount of load sent via edge e in step k. This scheme is known as the *diffusion algorithm* and has been described by Cybenko [4] and, independently, by Boillat [2]. Denoting by $a = (\alpha_1, \ldots, \alpha_N)^T$ the vector of edge weights and by $D = \mathrm{diag}(a) \in \mathbb{R}^{N \times N}$ the $N \times N$ diagonal matrix containing the edge weights on its diagonal, (2) can be written in matrix notation as $w^k = Mw^{k-1}$ with $M = I - ADA^T \in \mathbb{R}^{n \times n}$. The non-zero structure of M is equivalent to the adjacency matrix B. M contains α_e at position (i,j) of edge $e = (v_i, v_j)$, $1 - \sum_{e = \{v_i, v_j\} \in E} \alpha_e$ at diagonal entry i, and 0 elsewhere. The edge weights α_e have to be chosen in such a way that M is nonnegative. Hence, M is nonnegative, symmetric and doubly stochastic, i.e. its rows and columns sum up to 1. We call such a matrix M a *diffusion matrix*, if it has the additional property that in the case that the graph G is bipartite, at least one diagonal entry is positive. Then $+1$ is a simple eigenvalue of M and all other eigenvalues are smaller in modulus [4]. Consequently, the iteration (2) converges to the average load $\overline{w}$ [4]. If $\alpha_e = \alpha$ for all edges $e \in E$, then M is of the special form $M = I - \alpha L$. We will see in Section 4 that in this case the iteration (2) converges to an l_2-minimal flow x, which is independent of the value of α. If M is of the more general form $M = I - ADA^T$, iteration (2) determines a flow which is minimal in some weighted Euclidean norm.

After a balancing flow has been computed, a schedule of load movements has to be found obeying the flow demands. This is particularly easy if initially each

node has sufficiently many tokens to fulfill the demands on its outgoing edges. In this case the load can be balanced in one step. In the general case, a valid schedule has to be found which decomposes the flow in such a way that in each step a node moves not more tokens than it possesses at the beginning of the step, i.e. tokens received in step i can not be sent before step $i+1$. The task here is to find a schedule of minimal length.

More formally, let $\tilde{A} \in \{-1,0,1\}^{n \times N}$ be the incidence matrix A of G where the implicit edge directions express the directions of the flow, i.e. $\tilde{x}_i = |x_i|$ for all i and $\tilde{A}\tilde{x} = w - \overline{w}$. Let $\tilde{A} = \tilde{A}^+ + \tilde{A}^-$ be a decomposition of $\tilde{A}$ into its positive and negative part. With $\tilde{A}^- \in \{-1,0\}^{n \times N}$ ($\tilde{A}^+ \in \{0,1\}^{n \times N}$) we denote the $n \times N$ matrix derived from $\tilde{A}$ by setting all the $+1$-entries (-1-entries) to 0. The *flow scheduling problem* is defined as follows:

Definition 1. (The Flow Scheduling Problem)
Input: *A graph G, node weights $w^0 := w$, a flow $\tilde{x}$ and a number $k \geq 0$.*
Question: *Is there a decomposition $S(\tilde{x}) = (\tilde{x}^0, \ldots, \tilde{x}^{k-1})$ of the flow $\tilde{x}$ with*

$$\tilde{x} = \sum_{j=0}^{k-1} \tilde{x}^j \qquad \text{and} \qquad w^{j+1} = w^j + \tilde{A}\tilde{x}^j = w^j + \underbrace{\tilde{A}^-\tilde{x}^j}_{\text{send}} + \underbrace{\tilde{A}^+\tilde{x}^j}_{\text{receive}}$$

such that

$$w^j + \tilde{A}^- x^j \geq 0 \quad \forall\, j = 0, \ldots, k-1 \tag{3}$$

A schedule $S(\tilde{x})$ satisfying (3) *is called* valid. *A valid schedule with minimal k among all possible valid schedules is called* time-optimal.

Note that for this type of scheduling problem the weight $\tilde{x}_e$ of an edge e determines how many tokens have to be sent via this edge. However, the destination of a token is not known in advance and has to be determined by the schedule. This problem is of interest in a much broader context than load balancing. It appears whenever a flow has to be scheduled satisfying certain constraints.

For the rest of the paper we separately deal with the two problems of finding a balancing flow and finding a schedule for the flow. We consider local iterative algorithms finding l_2-optimal flows in the next two sections, and the scheduling problem in Section 5. Note that because we propose to use the iterative algorithms just to determine the flow, it is possible for them to operate on real values.

3 Local and Optimal Local Algorithms

In this section we present a general framework for nearest neighbor load balancing schemes in which the FOS and SOS (*second order scheme*; the over-relaxed diffusion by Ghosh et al. [10]) methods appear as special cases as well as a scheme based on the Chebyshev polynomials [5,12,15]. We will further present new 'optimal polynomial' schemes (OPS) which determine the average load after a finite number of iterative steps. The numerical experiments of Section 6 show that OPS can significantly improve over the SOS method.

3.1 General Framework

Due to our general assumption from Section 2, M has m distinct eigenvalues $1 = \mu_1 > \mu_2 > \quad > \mu_m > -1$, $\mu_1 = 1$ is a simple eigenvalue and $(1, 1, \ldots, 1)$ is an eigenvector of M to eigenvalue μ_1. We denote $\gamma = \max\{|\mu_2|, |\mu_m|\} < 1$.

Definition 2. *A* polynomial based *load balancing scheme is any scheme for which the work load w^k in step k can be expressed in the form*

$$w^k = p_k(M)w^0 \quad \textit{where} \quad p_k \in \Pi_k \tag{4}$$

and Π_k denotes the set of polynomials p of degree $\deg(p) \quad k$ satisfying $p(1) = 1$.

Note that the condition $p_k(1) = 1$ implies that all row sums in the matrix $p_k(M)$ are equal to 1. This in turn means that the total work load is conserved, i.e. $\sum_{i=1}^{n} w_i^k = \sum_{i=1}^{n} w_i^0$. Moreover, $p_k(1) = 1$ also implies that $p_k(M)\overline{w} = \overline{w}$, which yields the fundamental relation

$$e^k = p_k(M)e^0, \quad k = 0, 1, 2, \ldots . \tag{5}$$

for the errors $e^k = w^k - \overline{w}$.

Let us also note that for (4) defining an algorithmically feasible nearest neighbor scheme, it must be possible to rewrite it as an update process where w^k is computed from w^{k-1} (and maybe some more previous iterates) involving *one* multiplication with M only. This means that the polynomials p_k have to satisfy some kind of short recurrence relation. Within the polynomial representation the FOS method turns out to be $p_k(t) = t^k$ which results in the simple short recurrence $p_k(t) = t \quad p_{k-1}(t), k = 1, 2, \ldots$. The SOS can be written as $p_k(t) = \beta t p_{k-1}(t) + (1 - \beta)p_{k-2}(t)$. The Chebyshev method differs from SOS just in the fact that β is now dependent on the iteration step k. A more detailed analysis can be found in [5].

3.2 Optimal Polynomial Methods

Choose eigenvectors z_i of M to eigenvalues μ_i in such a way that $w^0 = \sum_{i=1}^{m} z_i$. Then, $z_1 = \overline{w}$, $e^0 = \sum_{i=2}^{m} z_i$ and (5) yields $e^k = \sum_{i=2}^{m} p_k(\mu_i) z_i$. This implies

$$\|e^k\|_2^2 \quad \sum_{i=2} p_k(\mu_i)^2 \quad \max_{i=2}^{m} \|z_i\|_2^2 \; . \tag{6}$$

As we will see now, it is possible to construct algorithmically feasible nearest neighbor schemes which minimize the factor $\sum_{i=2}^{m} p_k(\mu_i)^2$ in (6). We need some additional terminology. For any two polynomials p, q we define the (indefinite) inner product $\langle \cdot , \cdot \rangle$ as

$$\langle p, q \rangle := \sum_{j=2} \omega_j p(\mu_j) q(\mu_j) \; ,$$

where $\omega_2, \ldots, \omega_m$ are *a priori* given positive weights. We are interested in polynomials p_k which minimize $\langle p, p \rangle$ over Π_k. The following theorem gives a rather complete answer.

Theorem 1. *For $k = 0, \ldots, m-1$ define the polynomials $p_k \in \Pi_k$ as follows:*

$$p_0(t) \equiv 1, \; p_1(t) = \frac{1}{\gamma_1}[(\alpha_1 - t)p_0(t)],$$
$$p_k(t) = \frac{1}{\gamma_k}[(\alpha_k - t)p_{k-1}(t) - \beta_k p_{k-2}(t)], \; k = 2, \ldots, m-1, \tag{7}$$

where

$$\left.\begin{aligned} \alpha_k &= \langle tp_{k-1}, p_{k-1}\rangle / \langle p_{k-1}, p_{k-1}\rangle, \; k = 1, \ldots, m-1; \\ \beta_k &= \gamma_{k-1}\langle p_{k-1}, p_{k-1}\rangle / \langle p_{k-2}, p_{k-2}\rangle, \; k = 2, \ldots, m-1; \\ \gamma_1 &= \alpha_1 - 1, \; \gamma_k = \alpha_k - 1 - \beta_k, \; k = 2, \ldots, m-1. \end{aligned}\right\} \tag{8}$$

Then we have

$$\langle p_k, p_j\rangle = 0 \quad \text{for} \quad k, j = 0, \ldots, m-1, \; k \neq j \tag{9}$$

and

$$\sum_{j=2}^{m} \frac{\omega_j}{1-\mu_j} \, p_k(\mu_j)^2 = \min_{p \in \Pi_k} \sum_{j=2}^{m} \frac{\omega_j}{1-\mu_j} \, p(\mu_j)^2, \; k = 0, \ldots, m-1 . \tag{10}$$

Proof: The whole theorem is well known in numerical analysis, although it is usually stated in terms of the 'residual polynomials' $r(t) = p(1-t)$. We refer the reader to [5,8]. □

Taking $\omega_j = 1 - \mu_j$, Theorem 1 shows how to construct a sequence of polynomials p_k for which the bound $\sum_{i=2}^{m} p_k(\mu_i)^2$ from (6) is smallest possible. Turning this into a computational algorithm, we realize that we first have to precompute *all* eigenvalues of the matrix M and then precompute the scalars $\alpha_i, \beta_i, \gamma_i$ from Theorem 1. Once this is done, and the scalars $\alpha_i, \beta_i, \gamma_i$ are made available to all processors, we get the OPS nearest neighbor load balancing scheme

$$w^1 = \frac{1}{\gamma_1}[\alpha_1 w^0 - Mw^0] \; ,$$
$$w^k = \frac{1}{\gamma_k}[\alpha_k w^{k-1} - Mw^{k-1} - \beta_k w^{k-2}], \; k = 2, \ldots, m-1 \; .$$

Note that $p(t) = \prod_{i=2}^{m}(1 - \frac{t}{\mu_i})$ is the only polynomial from Π_{m-1} which achieves $\langle p, p\rangle = 0$, i.e. $p_{m-1}(t) = p(t)$. Thus, (6) gives $e^{m-1} = 0$, which shows that the above method is a finite method in the sense that it arrives at $w^k = \overline{w}$ in at most $k = m-1$ steps. Let us note that the standard CG method [11] shares this finite termination property. However, the CG method requires the computation of two inner products within each iterative step, so it is not a local method.

4 Solution Quality

The purpose of this section is to show that the load balancing algorithms of the previous section can easily be modified in such a manner that, in addition to the

iterative work loads w^k, they also compute an l_2-minimal flow from w^0 to w^k. The essential assumption is that the diffusion matrix M in the load balancing scheme is of the form $M = I - \alpha L$, where L is the Laplacian of the processor graph and α is a fixed weight for all edges $e \in E$.

4.1 Basic Results

According to [13], l_2-minimal flow solutions for given work loads w and $v \in \mathbb{R}^n$, i.e. vectors $x \in \mathbb{R}^N$ which have minimal norm $\|x\|_2$ under all those satisfying $Ax = b$ where $b = w - v$, can be characterized as follows.

Lemma 1. *Consider the l_2 minimization problem*

$$\text{minimize } \|x\|_2 \text{ over all } x \text{ with } Ax = b.$$

Provided that $b \in \overline{w}^?$, the unique solution to this problem is given by

$$x = A^T z, \text{ where } Lz = b. \tag{11}$$

Our next lemma shows that if we have a sequence of work loads converging to the average load, and if we have an l_2-minimal flow for each such load, then these minimal flows converge to the minimal flow for the average load.

Lemma 2. *Let w^k be a (finite or infinite) sequence of work loads which converges to the average load $\overline{w}$. Moreover, let $w^k = w^0 + Ax^k$ be such that $\|x^k\|_2$ is minimal, i.e. (by Lemma 1) $x^k = A^T z^k$, where $Lz^k = w^k - w^0$. Then, $\lim_{k \to \infty} x^k = \bar{x}$ exists, $\overline{w} = w^0 + A\bar{x}$, and $\|\bar{x}\|_2$ is minimal.*

Proof: Let $\lambda = \sum_{i=1}^{n} w_i^0$ so that $\overline{w}_i = \lambda/n$, $i = 1, \ldots, n$. Since there are several z^k satisfying $Lz^k = w^k - w^0$, let us take the (Moore-Penrose [11]) pseudoinverse solution for all k, i.e.

$$z^k = L^{\dagger}(w^k - w^0) \; .$$

This immediately implies that $\lim_{k \to \infty} z^k = \bar{z}$ exists, satisfying $\bar{z} = L^{\dagger}(\overline{w} - w^0)$. Consequently, $\lim_{k \to \infty} x^k = \bar{x}$ exists, too, and it satisfies $\bar{x} = A^T \bar{z}$ as well as $\overline{w} = w^0 + A\bar{x}$. So, by Lemma 1, $\bar{x}$ is the l_2-minimal flow. □

Hu and Blake suggest to solve $Lz = \overline{w} - w$ directly using for example the conjugate gradient iteration [13]. The flow is then given as $x = A^T z$. We show in the following how to iteratively update x within any of the nearest neighbor schemes considered so far such that x^k converges to the l_2-minimal flow $\bar{x}$. In this manner we get a true nearest neighbor scheme for computing the minimal flow as well.

4.2 Computing Work Loads and Minimal Flows

We assume that the diffusion matrix M is of the particular form

$$M = I - \alpha L, \quad \alpha \in \mathbb{R} \; . \tag{12}$$

We start with a general observation which holds for any polynomial based method with diffusion matrix M, i.e. for methods where we have $w^k = p_k(M)w^0$ with $p_k \in \Pi_k$. Since $p_k(1) = 1$, the polynomial $p_k(1 - \alpha t)$ has value 1 for $t = 0$, so that we get the representation

$$p_k(1 - \alpha t) = 1 + tq_{k-1}(t), \quad \deg(q_{k-1}) \quad k - 1 .$$

Because of (12) this shows that $w^k = p_k(M)w^0 = w^0 + Lq_{k-1}(L)w^0$, so that z^k from Lemma 2 is given by $q_{k-1}(L)w^0$. Thus, the z^k and, consequently, the x^k are related in quite a straightforward manner to the polynomials defining the load balancing method. However, for practical algorithmic formulations we have to turn this relation into a cheap update process for the x^k. Thus, our goal is to find a vector d^{k-1} which is easy to update and which can be used to calulate a flow increment y^{k-1}. The next theorem describes the update process.

Theorem 2. *Let $p \in \Pi_k$ be a polynomial satisfying the 3-term recurrence*

$$p_k(t) = (\sigma_k t - \tau_k)p_{k-1}(t) + \rho_k p_{k-2}(t) , \tag{13}$$

$$\text{with} \qquad \sigma_k - \tau_k + \rho_k = 1 \quad \forall\, k = 1, 2, \ldots \tag{14}$$

Let $d^0 = -\alpha\sigma_1 w^0$, $x^1 = y^0 = A^T d^0$, $w^1 = w^0 + Ay^0$, and for $k = 2, 3, \ldots$

$$\begin{array}{ll} d^{k-1} = -\alpha\sigma_k w^{k-1} - \rho_k d^{k-2}; & y^{k-1} = A^T d^{k-1}; \\ x^k \quad = x^{k-1} + y^{k-1}; & w^k \quad = w^{k-1} + Ay^{k-1}; \end{array} \tag{15}$$

be the update process for x^k and w^k. Then, $\lim_{k\to\infty} x^k = \bar{x}$ and $\lim_{k\to\infty} w^k = \overline{w}$ exist, $\overline{w} = w^0 + A\bar{x}$ and $\|x - x_2\|$ is minimal.

Proof: With $p_k(1 - \alpha t) = 1 + tq_{k-1}(t)$ equation (13) becomes

$$1 + tq_{k-1}(t) = (\sigma_k(1 - \alpha t) - \tau_k)(1 + tq_{k-2}(t)) + \rho_k(1 + tq_{k-3}(t))$$

which, after some algebraic manipulations, yields

$$\begin{aligned} q_{k-1}(t) &= -\alpha\sigma_k(1 + tq_{k-2}(t)) + (\sigma_k - \tau_k)q_{k-2}(t) + \rho_k q_{k-3}(t) \\ &= -\alpha\sigma_k p_{k-1}(1 - \alpha t) + (\sigma_k - \tau_k)q_{k-2}(t) + \rho_k q_{k-3}(t). \end{aligned}$$

This shows that z^k from Lemma 2 is given by

$$z^k = -\alpha\sigma_k w^{k-1} + (\sigma_k - \tau_k)z^{k-1} + \rho_k z^{k-2}.$$

Substituting $z^k = z^{k-1} + d^{k-1}$ and $z^{k-2} = z^{k-1} - d^{k-2}$ and using (14) we finally arrive at the update formula $d^{k-1} = -\alpha\sigma_k w^{k-1} - \rho_k d^{k-2}$ for d^{k-1}. Theorem 2 now follows with Lemma 2. ∎

Looking at the schemes discussed so far, we have for the FOS $\sigma_k = 1$, $\tau_k = \rho_k = 0$ so that $d^{k-1} = -\alpha w^{k-1}$. In the Chebyshev scheme, for each but the first step, we have $\sigma_k = \beta_k$, $\tau_k = 0$, $\rho_k = (1 - \beta_k)$ which yields $d^{k-1} = -\alpha\beta_k w^{k-1} - (1 - \beta_k)d^{k-2}$. The first step is identical to FOS. The SOS scheme differs from

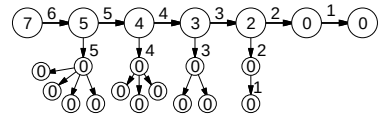


Fig. 1. A bad example for the local scheduling algorithm.

Chebyshev only by the fact that $\sigma_k = \beta_{opt}$. Finally, for all but the first step in the OPS scheme we have $\sigma_k = -1/\gamma_k$, $\tau_k = -\alpha_k/\gamma_k$ and $\rho_k = -\beta_k/\gamma_k$ so that $d^{k-1} = (\alpha w^{k-1} + \beta_k d^{k-2})/\gamma_k$. The first step can be formulated in a similar way. A precise pseudocode describing the iteration on processor level is given in [5].

As a final comment to this section, let us just mention that the above discussion carries over to the case where we associate different costs with each of the edges and M is of the more general form $M = I - ADA^T$, where the diagonal matrix D is related to the cost vector. Details are in [5].

5 Flow Scheduling

Consider the simple example of a chain of three nodes u, v, w where the left and the right node hold $3r$ tokens each, and the middle is empty. FOS with parameter $\alpha = \frac{1}{2}$ shifts a total amount of $3r$ tokens via each of the two edges whereas an optimal flow moves only r from u to v and r from w to v. So the two-phase approach moves much less.

Considering the execution time, experimental observations show that usually the load can be moved in a small number of steps after the balancing flow has been found. Figure 2 (Sec. 6) shows impressive examples for this fact. All nearest neighbor schemes take a rather large number of iterative steps (even the optimal one), whereas the load movement using a simple greedy strategy is finished after at most 3 steps. However, it remains open up to now how to justify this obervation analytically.

It is interesting to notice that the diameter of the graph is not an upper bound on the number of steps a schedule can take. More specifically, we show in [5] that there exist graphs with $n^2 + 1$ nodes and diameter 4 where any scheduling of an l_2-optimal balancing flow has to take at least $\frac{3}{4}n - 1$ steps.

A local greedy flow scheduling algorithm determines for each node and each step how many of the available tokens to send to which of the outgoing edges. Local greedy heuristics can be characterized by the following points: (i) Their scheduling decision depends only on local information about the flow demands x and the available load w. (ii) If in a certain step a node contains enough tokens to fulfill all is outflow-demands, it immediately saturates all its outgoing edges. (iii) If a node does not contain enough load, it distributes the available tokens to its outgoing edges according to some tie-breaking. In the experiments reported in Section 6 such a local scheduling heuristic balances the load in only a small number of steps. We show that this is not always the case.

Consider the class of memory-less greedy algorithms where a decision depends only on the current situation and not on the history. By *Round-Robin Greedy*

(RRG), we denote the local greedy scheduling algorithm which fills up one edge after the other. Per step, RRG still moves as much load as possible, i.e. it sends all its available tokens, but chooses a subset of edges which are filled up to saturation (where the last edge in the subset might not be saturated completely).

Lemma 3. *For every balancing flow the Round-Robin local greedy scheduling algorithm is $\Theta(\sqrt{n})$-optimal.*

Proof: We only show the lower bound. The complete proof can be found in [5]. Consider the construction of Figure 1. Let p be the length of the upper line of nodes (the "backbone"). The graph contains $n = \frac{1}{2}p(p-1)$ nodes. In an optimal schedule, the nodes on the backbone send their load to their right neighbor first. Such a schedule needs 3 steps. For a local greedy scheduling algorithm filling up one edge after the other we may assume that it sends the available load downwards first. In this case, p steps are needed. ∎

6 Experimental Results

Here, we shortly report some numerical results which demonstrate the advantages of the OPS method and compare it with the FOS and SOS schemes. We also include experimental results for the RRG greedy schedule.

We consider two different processor graphs, each with 64 nodes: The 8 8-torus and a quotient graph arising from a partition of the finite element mesh "airfoil1" into 64 sub-domains [6]. The initial work load is identically generated as a uniformly random distribution. All computations are performed with real numbers according to the algorithms given in the previous section. After the flow is determined, we round it up to integral values and schedule it using the RRG local greedy heuristic. For the flow calculation, the diffusion matrix M is initially taken to be of the form $M = I - \alpha L$ with α the inverse of the maximum node degree ($\alpha = 1/4$ and $1/10$ for the 2D Torus and the FE-quotient graph, resp.). For the FOS and SOS scheme we apply an additional spectral shift of the form $M \quad (1-\delta)I + \delta M$ as described in [4] in order to minimize γ and therefore maximize the speed of convergence (note that this shift also ensures $\gamma < 1$).

Figure 2 shows that on average OPS requires only half as many iterations as SOS, with FOS being by far the slowest scheme (see also [10]). It is also apparent that in early iterations the SOS and the OPS schemes can behave quite similarly. Figure 2 very clearly illustrates the fact that OPS achieves the solution after $m - 1$ steps (m: number of different eigenvalues of M). Very interestingly, this convergence takes place quite 'brutally' with the immediately preceeding iterates still being relatively far from the solution.

Figure 2 also shows that the time needed to actually balance the load if a valid balancing flow is known is much less than the number of iterations performed by any of the nearest neighbor schemes. On both examples shown here, the simple greedy schedule requires only 2 steps, and such a low number appears to be typical for many other examples we tested.

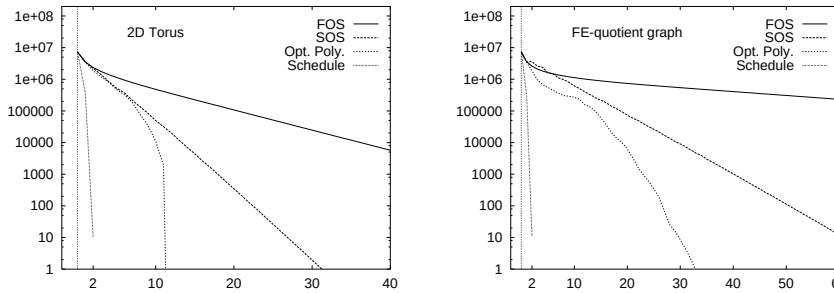


Fig. 2. Performance of different balancing schemes ($\|w^k - \overline{w}\|_2$ vs. iteration step k).

References

1. A. Berman, R.J. Plemmons. *Nonnegative Matrices in the Mathematical Sciences.* Academic Press, 1979.
2. J.E. Boillat. Load Balancing and Poisson Equation in a Graph. *Concurrency: Prac. & Exp.*, 2(4):289-313, 1990.
3. D.M. Cvetković, M. Doob, H. Sachs. *Spectra of Graphs.* Barth, Heidelberg, 1995.
4. G. Cybenko. Load Balancing for Distributed Memory Multiprocessors. *J. Par. Distr. Comp.*, 7:279-301, 1989.
5. R. Diekmann, A. Frommer, B. Monien. Efficient Schemes for Nearest Neighbor Load Balancing. Techn. Rep., Univ. of Paderborn, 1998.
6. R. Diekmann, S. Muthukrishnan, M.V. Nayakkankuppam. *Engineering Diffusive Load Balancing Algorithms Using Experiments.* Proceedings IRREGULAR, Springer LNCS 1253, 111-122, 1997.
7. R. Diekmann, F. Schlimbach, C. Walshaw. *Quality Balancing for Parallel Adaptive FEM.* Proceedings IRREGULAR, Springer LNCS, 1998.
8. B. Fischer. *Polynomial Based Iteration Methods for Symmetric Linear Systems.* Wiley, 1996.
9. G. Fox, R. Williams, P. Messina. *Parallel Computing Works!* Morgan Kaufmann, 1994.
10. B. Ghosh, S. Muthukrishnan, M.H. Schultz. *First and Second Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing.* ACM-SPAA, 72-81, 1996.
11. G. Golub, Ch. van Loan. *Matrix Computations.* Johns Hopkins, Baltimore, 1989.
12. G. Golub, R. Varga. Chebyshev semi-iterative methods, successive overrelaxation iterative methods, and second order Richardson iterative methods. *Numer. Math.*, 3:147–156, 1961.
13. Y.F. Hu, R.J. Blake. *An optimal dynamic load balancing algorithm.* Techn. Rep. DL-P-95-011, Daresbury Lab., UK, 1995 (to appear in *Concur.: Pract. & Exp.*).
14. K. Schloegel, G. Karypis, V. Kumar. *Parallel Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes.* Proceedings EuroPar '97, Springer LNCS, 1997.
15. R. Varga. *Matrix Iterative Analysis.* Prentice-Hall, 1962.
16. C. Walshaw, M. Cross, M. Everett. *Dynamic load-balancing for parallel adaptive unstructured meshes.* 8th SIAM Conf. Par. Proc. for Sc. Computing, 1997.
17. C. Xu, F. Lau. *Load balancing in Parallel Computers: Theory & Practice.* Kluwer Academic Publishers, 1997.

2-Approximation Algorithm for Finding a Spanning Tree with Maximum Number of Leaves

Roberto Solis-Oba

Max Planck Institut für Informatik
Im Stadtwald, 66123 Saarbrücken, Germany
solis@mpi-sb.mpg.de

Abstract. We study the problem of finding a spanning tree with maximum number of leaves. We present a simple 2-approximation algorithm for the problem, improving on the approximation ratio of 3 achieved by the best previous algorithms. We also study the variant in which a given set of vertices must be leaves of the spanning tree, and we present a 5/2-approximation algorithm for this version of the problem.

1 Introduction

In this paper we study the problem of finding in a given graph a spanning tree with maximum number of leaves. This problem has applications in the design of communication networks [5], circuit layouts [11], and in distributed systems [10]. Galbiati et. al [3] have proven that the problem is MAX SNP-complete, and hence that there is no polynomial time approximation scheme for the problem unless P=NP. In this paper we present a 2-approximation algorithm for the problem, improving on the previous best performance ratio of 3 achieved by algorithms of Ravi and Lu [8,9].

We briefly review previous and related work to this problem. The problem of finding a spanning tree with maximum number of leaves is, from the point of view of optimization, equivalent to the problem of finding a minimum connected dominating set. But, the problems are very different when considering how well their solutions can be approximated. Khuller and Guha [5] gave an approximation preserving reduction from the set cover problem to the minimum connected dominating set. This result implies that the set cover problem cannot be approximated within ratio $(1 - o(1)) \ln n$, where n is the number of elements in the ground set, unless NP DTIME($n^{\log \log n}$) [2]. However the solution to the problem of finding a spanning tree with maximum number of leaves is known to be approximable within a constant of the optimum value [8,9].

There are several papers that deal with the question of determining the largest value $'_k$ such that every connected graph with minimum degree k has a spanning tree with at least $'_k$ leaves [1,4,7,11]. Kleitman and West [7], Storer [11], and Griggs et al. [4] show that every connected graph with n vertices and minimum degree $k = 3$ has a spanning tree with at least $n=4 + 2$ leaves. For $k = 4$ Kleitman and West [7] give a bound of $(2n + 8)=5$ for the value of $'_k$,

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 441{452, 1998.
c Springer-Verlag Berlin Heidelberg 1998

and for arbitrary k they give a lower bound of $(1 - \Omega(\ln k/k))n$ for the number of leaves. This bound was improved by Duckworth et al. [1] to $\frac{k-5}{k+1}2^k + 2$ for the special case of a hypercube of dimension k. All these algorithms can be used to approximate the solution to the problem of finding a spanning tree with maximum number of leaves, but only for graphs with minimum degree k, $k \geq 3$.

Ravi and Lu [8] presented the first constant-factor approximation algorithm for the problem on arbitrary graphs. Using local-improvement techniques they designed approximation algorithms with performance ratios of 5 and 3. Later [9] they introduced the concept of *leafy forest* that allowed them to design a more efficient 3-approximation algorithm for the problem. A leafy forest has two nice properties: (1) it can be completed into a spanning tree by converting a small number of leaves of the forest into internal vertices of the tree, and (2) the number of leaves in an optimal tree can be upper bounded in terms of the number of vertices in the forest.

We improve on the algorithms by Ravi and Lu by providing a linear time algorithm that finds a spanning tree with at least half of the number of leaves in any spanning tree of a given undirected graph. Our algorithm uses *expansion rules*, to be defined later, similar to those in [7]. However we assign priorities to the rules and use them to build a forest instead of a tree as in [7]. Incidentally, the forest F that our rules build is a leafy forest, so we can take advantage of its structure to build a spanning tree with a number of leaves close to that in the forest.

Informally, the priority of a rule reflects the number of leaves that the rule adds to the forest F. Hence it is desirable to use only "high" priority rules to build the forest. The "low" priority rules are needed, though, to ensure that only a bounded number of leaves become internal vertices when connecting the trees in F to form a spanning tree.

The key idea that allows us to prove the approximation ratio of 2 for the algorithm is an upper bound for the number of leaves in any spanning tree that takes into account the number of times that "low" priority rules need to be used to build the forest F.

We also consider the variant of the problem in which a given set S of vertices must be leaves and a spanning tree T_S with maximum number of leaves subject to this constraint is sought. By using the above algorithm we reduce this problem to a variant of the set covering problem in which instead of minimizing the size of a cover, we want to maximize the number of sets which do not belong to the cover. We present a simple heuristic for this latter problem which yields a (5=2)-approximation algorithm for finding the spanning tree T_S.

The rest of the paper is organized in the following way. In Sect. 2 we present our approximation algorithm. In Sect. 3 we prove a weaker bound of 3 for the performance ratio of the algorithm, and in Sect. 4 we strengthen the analysis to show the ratio of 2. Finally, in Sect. 5 we present a (5=2)-approximation algorithm for the version of the problem in which a given set of vertices must be leaves of the tree.

2 The Algorithm

Let $G = (V; E)$ be an undirected connected graph. We denote by m the number of edges and by n the number of vertices in G. Let T be a spanning tree of G with maximum number of leaves. In this section we present an algorithm that finds a spanning tree T of G with at least half of the number of leaves in T .

The algorithm first builds a forest F by using a sequence of *expansion rules*, to be defined shortly. Then the trees in F are linked together to form a spanning tree T. The expansion rules used to build F are designed so that a "large" fraction of the vertices in F are leaves and when T is formed, the smallest possible number of leaves from F are transformed into internal vertices. Hence the resulting spanning tree has "many" leaves. When forming the spanning tree T, we say that a leaf of F is *killed* when it becomes an internal vertex of T.

We now elaborate on how the forest F is constructed. Every tree T_i of F is built by first choosing a vertex of degree at least 3 as its root. Then the expansion rules described in Fig. 1 are used to grow the tree. These rules are applied to the leaves of the tree as follows. If a leaf x has at least two neighbors not in T_i then the rule shown in Fig. 1(b) is used which places all neighbors of x not belonging to T_i as its children. On the other hand, if x has only one neighbor y that does not belong to T_i and at least two neighbors of y are not in T_i, then the rule shown in Fig. 1(a) is used. This rule puts y as the only child of x and all the neighbors of y not in T_i are made children of y.

When a rule is applied to a vertex x we say that x is *expanded* by the rule. A tree T_i is grown until none of its leaves can be expanded.

We assign priorities to the expansion rules as follows. The rule shown in Fig. 2, namely a leaf x has a single neighbor y not in F and y has exactly two neighbors outside F, has priority 1. All other expansion rules have priority 2. In the rest of the paper we will refer to the rule of priority 1 simply as rule 1. The internal vertex y added with rule 1 (see Fig. 2) is called a *black vertex*.

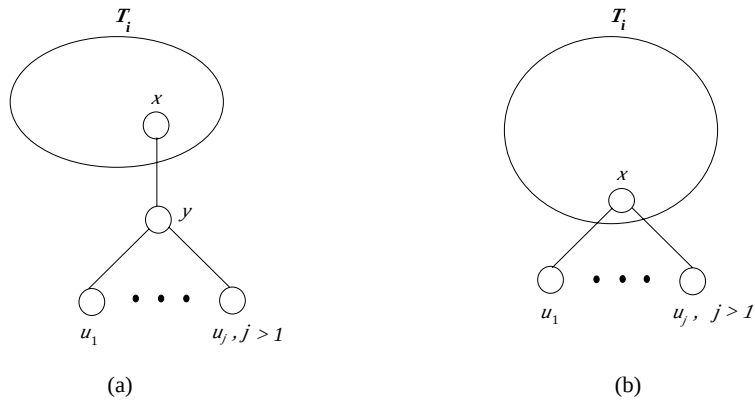


Fig. 1. Expansion rules.

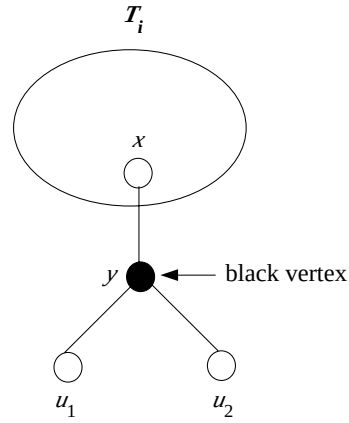


Fig. 2. Rule 1.

When building a tree $T_i \in F$, if two different leaves of T_i can be expanded, the leaf that can be expanded with the highest priority rule is expanded first. If two leaves can be expanded with rules of the same priority, then one is arbitrarily chosen for expansion. Our algorithm for finding a spanning tree T with many leaves is the following.

Algorithm *tree*(G)
$F \leftarrow \emptyset$;
while there is a vertex v of degree at least 3 **do**
  Build a tree T_i with root v and leaves the neighbors of v.
  while at least one leaf of T_i can be expanded **do**
    Find a leaf of T_i that can be expanded with
    a rule of largest priority, and expand it.
  end while
  $F \leftarrow F \cup T_i$
  Remove from G all vertices in T_i and all edges incident to them.
end while
Connect the trees in F and all vertices not in F to form a spanning tree T.

3 Analysis of the Algorithm

In this section we show that the performance ratio of algorithm *tree* is at most 3, and in the next section we tighten the analysis to prove the performance ratio of 2. Given a graph $G = (V, E)$, let $F = \{T_0, T_1, \ldots, T_k\}$ be the forest built by our algorithm. For a tree $T_i \in F$, let $V(T_i)$ be the set of vertices spanned by it, and let $B(T_i)$ be the set of black vertices in T_i. The set of vertices spanned by F is $V(F)$ and the set of black vertices in F is denoted as $B(F)$. The set of leaves in a tree T is denoted as $\ell(T)$.

Let $X = V - V(F)$, be the set of vertices not spanned by F. We call X the set of *exterior* vertices. An exterior vertex cannot be adjacent to an internal vertex of some tree T_i because when a vertex x is expanded all its neighbors not in F are placed as its children.

We show that forest F has the property that any way of interconnecting the trees T_i to form a spanning tree T of G kills exactly $2k$ leaves. Moreover, exactly one leaf of F must be killed to attach every exterior vertex to T. This two facts allow us to bound the number of leaves in any spanning tree of G in terms of the number of vertices in F.

Lemma 1. *Let $G' = (V'; E')$ be the graph formed by contracting every tree $T_i \in F$ to a single vertex and then removing multiple edges between pairs of vertices. Every exterior vertex has degree at most 2 in G'.*

Proof. The proof is by contradiction. Assume that there is an exterior vertex v that has degree at least 3 in G'. Consider 3 of the neighbors of v. Note that these 3 vertices cannot be exterior vertices because then algorithm *tree* would have chosen v as the root of a new tree. Hence, at least one neighbor of v is in F. Let T_i be the first tree built by our algorithm that contains one of the neighbors u of v. Since v is adjacent to two vertices not in T_i then algorithm *tree* would have expanded u using a rule of the form shown in Fig. 1(a), placing v as its child. □

Lemma 2. *Let u be a leaf of some tree $T_i \in F$. If u is adjacent to two vertices $v, w \notin T_i$, then v and w are leaves of the same tree $T_j \in F$.*

Proof. Clearly, v and w cannot be both exterior vertices. Also neither v nor w can be internal vertices of some tree T_j because if, say, v is an internal vertex of T_j then

1. if the algorithm builds tree T_j before T_i, then u would be placed as child of v in T_j, and
2. if T_i is built first, then v would have been placed as child of u. To see this observe that every internal vertex of a tree $T_j \in F$ has at least 3 neighbors in T_j. Thus, vertex u would have been expanded with a rule of the form shown in Figure 1(a) while building tree T_i.

Hence at least one of v, w must be a leaf in F. Let v be a leaf of some tree T_j and p be the parent of u in T_i. If w is not a leaf of T_j, then we can assume without loss of generality that when algorithm *tree* adds vertex v to T_j vertex w still does not belong to F. Note that then tree T_i cannot be built before T_j because our algorithm would have placed v and w as children of u. So let T_j be built before T_i.

Since vertices u, p, and w do not belong to F when T_j is built, then algorithm *tree* would expand v and place u as its child in T_j, which is a contradiction. □

Corollary 1. *Any spanning tree of G has at most $|V(F)| - 2k$ leaves.*

Proof. Let T^0 be a spanning tree of G and let F^0 be the forest induced by T^0 on $V(F)$. By Lemmas 1 and 2, any way of interconnecting the trees of F^0 to form a spanning tree must kill at least $2k$ leaves from F^0. Also, to form a spanning tree of G, every exterior vertex not used to interconnect the trees in F^0 must be attached to a different leaf of F^0. □

We now give a bound for the number of leaves in forest F and prove a performance ratio of 3 for the algorithm.

Lemma 3. *For any tree T_i 2 F, $j`(T_i)j \quad 3+jB(T_i)j+(jV(T_i)j-3jB(T_i)j-4)=2$.*

Proof. The root of T_i is a vertex of degree at least 3, so it has at least 3 children. Also, every application of rule 1 adds two new leaves to T_i while killing one. Hence, by using $jB(T_i)j$ times rule 1 the number of leaves in T_i is increased by $jB(T_i)j$. All other vertices in T_i are added by rules of priority 2. It is not difficult to check that a rule of priority 2 increases the number of leaves in T_i by at least half of the number of vertices added to T_i by that rule. □

Lemma 4. *The performance ratio of algorithm tree is smaller than* 3.

Proof. Let T be the spanning tree built by our algorithm and let T be a spanning tree with maximum number of leaves. By Corollary 1 and Lemma 3,

$$\frac{`(T\ )}{`(T)} \quad \frac{jV(F)j-2k}{\sum_{i=0}^{k}(3+jB(T_i)j+(jV(T_i)j-3jB(T_i)j-4)=2)-2k} \quad \frac{2(jV(F)j-2k)}{jV(F)j-jB(F)j-2k+2} \quad 2+\frac{2jB(F)j}{jV(F)j-jB(F)j-2k} \ :$$

Observe that $jV(F)j = \sum_{i=0}^{k} jV(T_i)j \quad \sum_{i=0}^{k}(4+3jB(T_i)j)$ because the root of a tree T_i has degree at least 3 and each application of rule 1 adds 3 vertices to the tree. So $jV(F)j > 4k + 3jB(F)j$, and therefore, $`(T\ )=`(T) < 2+2jB(F)j=(2jB(F)j+2k) \quad 3$. □

Note that if $jB(F)j = 0$ then the proof of Lemma 3 would give a bound of 2 for the performance ratio of the algorithm. However, if $jB(F)j > 0$ then our analysis yields a bound of only 3. Intuitively this is because rule 1 adds three vertices to a tree, but it increases the number of leaves of the tree by only 1. So, only one third of the vertices added by rule 1 are leaves. To prove the bound of 2 for the performance ratio of our algorithm we will show that there must be at least one internal vertex in T for every black vertex in F.

4 Top, Exterior, and Black Vertices

Let $T_0; T_1; :::; T_k$ be the trees in the forest F, indexed in the order in which they are built by algorithm *tree*. The set of vertices $V(T_i)$ spanned by tree T_i is called a *cluster*. Fix a spanning tree T of G with maximum number of leaves.

We choose one of the internal vertices r of T as its root. For the rest of this section we will assume that T is a rooted tree.

To simplify the analysis we modify the optimal tree T and the tree T built by our algorithm as follows. Let $v_0; v_1; \ldots; v_i$ be a path in T such that $v_1; \ldots; v_i$ are exterior vertices, v_i is a leaf of T, and $v_1; \ldots; v_{i-1}$ have degree 2 in T. We remove vertices $v_1; \ldots; v_{i-1}$ and add edge $(v_0; v_i)$. By doing this every exterior vertex which is a leaf in T is directly connected to a non-exterior vertex. Note that this change does not modify the number of leaves of T or T. Moreover, the forest F is not affected by this change. We call the resulting trees T and T.

We prove below a tighter upper bound than that given in Corollary 1 for the maximum number of leaves in a spanning tree of G, by showing that some vertices in $V(F)$ must be internal vertices in every spanning tree.

Consider a vertex x of some cluster $V(T_j)$ which does not contain the root r. Let p_{rx} be the path in T from r to x. Let x be the only vertex from $V(T_j)$ in p_{rx} and let y be the closest, non-exterior vertex, to x in p_{rx}. We say that y is a *top vertex*. The set of top vertices is a subset of the leaves of F which are killed when the trees T_i are interconnected to form T.

Given a vertex x of T, let T_x be the subtree of T rooted at x. We denote the set of top vertices in T_x as $P(T_x)$. Let $B_r(T_x)$ be the set formed by the black vertices in T_x and the vertices in T_x which are roots of trees in F. For every exterior vertex v which is a leaf in T_x, let $a(v)$ be the parent of v in T_x. Let $A(T_x)$ be the set formed by the parents $a(v)$ of all exterior vertices v which are leaves in T_x. Note that every vertex in $A(T_x)$ is a leaf of some tree in F.

Lemma 5. *In every subtree T_x of T, the sets $P(T_x); A(T_x)$, and $B_r(T_x)$ are disjoint.*

Proof. Here we only prove that $B_r(T_x) \setminus A(T_x) = ;$. The other proofs are similar. Let v be a vertex in $B_r(T_x)$. Either v is a black vertex or the root of some tree $T_i \; 2 \; F$. Observe that when algorithm *tree* adds vertex v to T_i, all the neighbors of v which do not belong already to F are placed as children of v in T_i. Thus, v cannot be adjacent to an exterior vertex and so $v \; \not\in \; A(T)$. □

We define the *de cit* of a subtree T_x, and denote it as *de cit*(T_x), as $jP(T_x) [A(T_x) [B_r(T_x)j - jI(T_x)j$, where $I(T_x)$ is the set of internal vertices in T_x. We prove below that the deficit of T is at most 1. This together with Lemma 5 shows that the number of leaves in T is at most $jVj - jA(T)j - jP(T)j - jB_r(T)j + 1 = jV(F)j - jP(T)j - jB_r(T)j + 1 < jV(F)j - jB(F)j - k - k$ because there are at least k top vertices in T and $jB_r(T)j = jB(F)j + k + 1$. This will immediately prove the following theorem.

Theorem 1. *Algorithm tree nds a spanning tree with at least half of the number of leaves in an optimal tree.*

Proof. By the proof of Lemma 4, the tree T built by our algorithm has $j'(T)j$ $(jV(F)j - jB(F)j - 2k)=2$. Hence by the above discussion,

$$\frac{'(T)}{'(T)} < \frac{jV(F)j - jB(F)j - 2k}{(jV(F)j - jB(F)j - 2k)=2} \quad 2 : \qquad \square$$

4.1 Bounding the Deficit of T^*

In this section we prove that the deficit of T is at most 1, this will complete the proof of Theorem 1. We say that a subtree T_x has *maximum deficit one* if $deficit(T_x) = 1$ and for every vertex v of T_x the deficit of the subtree T_v is at most one.

Lemma 6. *Let T_x be a subtree of T of maximum deficit one. Then at least one leaf of T_x belongs to $B_r(T)$.*

Proof. Since all vertices in $P(T_x)$ and $A(T_x)$ are internal vertices of T_x, then the only way in which T_x can have deficit 1 is if one of its leaves belongs to $B_r(T_x)$. □

Lemma 7. *No edge in G connects two vertices from $B_r(T)$.*

Proof. When algorithm *tree* adds a black vertex $u \in B_r(T)$ to some tree of F, all neighbors of u not in F are placed as its children. Since by definition the children of a vertex $u \in B_r(T)$ cannot belong to $B_r(T)$ the claim follows. □

Lemma 8. *If T_x is a subtree of maximum deficit one, then its root x is either a leaf or it has at least 2 children.*

Proof. The claim follows trivially if $x \in B_r(T)$ is a leaf of T. So we assume that x is an internal vertex of T. We consider 3 cases. (1) If $x \in A(T_x)$, then x is the parent of an exterior leaf u, and so the subtree T_u has deficit zero. Thus the only way in which T_x can have deficit 1 is if x has another child v and $deficit(T_v) = 1$. (2) If $x \notin B_r(T_x) \cup P(T_x) \cup A(T_x)$, then x must be the parent of at least 2 subtrees of deficit 1. (3) For the case when $x \in B_r(T_x)$ or $x \in P(T_x)$, we prove the lemma by showing that

(a) if $x \in B_r(T_x)$, then $deficit(T_x) < 1$, and
(b) if $x \in P(T_x)$, then x has at least two children u and v such that u is in the same cluster as x and $deficit(T_u) = 1$, and v is not in the same cluster as x and $deficit(T_v) < 1$.

We can prove claims (a) and (b) by induction on the number of vertices in T_x. Here we sketch the proof for claim (a) only. The basis of the induction is trivial. For the induction step, let us assume that $deficit(T_x) \geq 1$ and derive a contradiction from such assumption. Let u be a child of x and $deficit(T_u) = 1$. Note that u cannot be a leaf of T because then the only way in which $deficit(T_u)$ can be 1 is if $u \in B_r(T)$, but by Lemma 7 this cannot happen. Hence u is an internal vertex of T and by induction hypothesis all internal vertices v of T_u for which $deficit(T_v) = 1$ have at least two children.

To simplify the proof we modify the tree T_x as follows. For each internal vertex v of T_u such that $deficit(T_v) = 1$, remove all its children except two of them chosen as follows. Note that by induction hypothesis $v \notin B_r(T_x)$.

1. If $v \in V(T_x) - A(T_x) - P(T_x)$, keep 2 children that are roots of subtrees of deficit 1.
2. If $v \in A(T_x)$, keep the exterior vertex adjacent to v and one of its children that is the root of a subtree of deficit 1.
3. If $v \in P(T_x)$, then keep a vertex v_1 from the same cluster as v such that $\mathit{deficit}(T_{v_1}) = 1$ and a vertex v_2 not in the same cluster as v such that $\mathit{deficit}(T_{v_2}) < 1$. By induction hypothesis these vertices must exist. Also, remove all children of vertex v_2.

We denote the resulting tree as T_x. Note that these changes do not affect the value of $\mathit{deficit}(T_x)$.Every internal vertex of T_x, with the possible exception of x, has degree 3. Also, by Lemma 6, at least one leaf of T_x belongs to $B_r(T_x)$ and since $x \in B_r(T_x)$ then $|B_r(T_x)|$ 2. Consider the forest F built by our algorithm. Let w_1 and w_2 be, respectively, the first and second vertices from $B_r(T_x)$ that are added to forest F by algorithm *tree*. Note that w_1 and w_2 are not added at the same time to F.

Let S be the set of vertices from T_x which have been added to F by algorithm *tree* just before w_2 is included in some tree of F. Since w_1 is either a black vertex or the root of some tree $T_i \in F$, then all its neighbors must belong to S. Consider a longest path L in T_x starting at w_1 and going only through internal vertices of T_x that belong to S. Let y be the last vertex in L. Note that $y \neq x$ because otherwise $w_1 = x$ and so y would not be the last vertex in L since then (the internal vertex) u would also belong to S. Similarly, if w_1 is a leaf then its parent in T_x belongs to S and so $y \neq w_1$.

At least two of the neighbors of y in T_x must belong to S because otherwise y could be expanded by algorithm *tree* using a rule of priority 2 before w_2 is added to F, which by definition of S cannot happen. Let y_1 and y_2 be two neighbors of y in S. Clearly, both y_1 and y_2 cannot be internal vertices because otherwise L would not be a longest path as described above. Thus, let y_1 be a leaf of T_x. There are four cases that need to be considered.

1. $y \in V(T_x) - A(T_x) - P(T_x)$. Then $\mathit{deficit}(T_{y_1}) = 1$ and so $y_1 \in B_r(T_x)$ by Lemma 6. Since w_1 is the only vertex from $B_r(T_x)$ in S then $y_1 = w_1$. But by definition of y, vertex y_2 must also be a leaf and $\mathit{deficit}(T_{y_2}) = 1$. By Lemma 6, y_2 must belong to $B_r(T_x)$ which contradicts our assumption that w_1 was the only vertex from $B_r(T_x)$ in S.
2. $y \in B_r(T_x)$. This cannot happen since $y \neq w_1$ and we assumed that there is only one vertex form $B_r(T_x)$ in S.
3. $y \in A(T_x)$. Then either y_1 is an exterior vertex or $y_1 \in B_r(T_x)$. But y_1 cannot be an exterior vertex since $y_1 \in S$. Also y_1 cannot belong to $B_r(T_x)$ because if it does then $y_1 = w_1$ and y_2 would be an internal vertex of T contradicting our assumption for L.
4. $y \in P(T_x)$. We can derive a contradiction in this case also, but we omit the proof here since it is more complex than for the other cases.

The above arguments show that y cannot exist, and therefore if $x \in B_r(T\ )$ is an internal vertex of T then $\mathit{deficit}(T_x) < 1$. □

Lemma 9. *For all vertices x in T, $deficit(T_x) \leq 1$.*

Proof. The proof is by contradiction. Let x be an internal vertex of T such that T_x is a minimal tree of deficit larger than one, i.e., $deficit(T_x) > 1$ and for all vertices $v \neq x$ in T_x, $deficit(T_v) \leq 1$. By the proof of Lemma 8 we know that $x \notin B_r(T_x)$. Also by the same proof, if $x \in P(T_x)$ then x must have at least three children: two in the same cluster as x and one in a different cluster. We trim the tree T_x as described in the proof of Lemma 8 with the only exception that for vertex x we keep three children u, v, and w as follows.

1. If $x \in A(T_x)$, u is an exterior vertex, v and w are roots of trees of maximum deficit 1.
2. If $x \in P(T_x)$, u is a child in a cluster different from x, and v and w are children in the same cluster as x and which are roots of trees of maximum deficit one.
3. If $x \in V(T_x) - A(T_x) - P(T_x)$, u, v, and w are roots of trees of maximum deficit one.

Since x has at least two children which are roots of trees of deficit 1, then at least two leaves of T_x belong to $B_r(T_x)$. Let w_1, w_2, S, and L be as in the proof of Lemma 8. By using arguments similar to those used to prove Lemma 8 we can derive a contradiction, thus showing that $deficit(T_x) \leq 1$ for all vertices x. □

5 Fixing a Set of Leaves

Consider now that the vertices in some set $S \subseteq V$ are required to be leaves in a spanning tree of G, and the problem is to find a tree with maximum number of leaves subject to this constraint. In this section we present a 5/2-approximation algorithm for the problem.

It is easy to check if a graph G has a spanning tree in which a given set S of vertices are leaves. Two conditions are needed for such a spanning tree to exist. First, the graph obtained by removing from G all vertices in S and all edges incident to them must be connected. And second, every vertex in S must have at least one neighbor in $V - S$. For the rest of this section we will assume that there is at least one spanning tree having the vertices in S as leaves.

Without loss of generality we can assume that S forms an independent set of G, i.e. there are no edges having both endpoints in S. We can make this assumption since we are interested only in spanning trees in which the vertices of S are leaves and none of these trees includes an edge connecting two vertices from S.

Let $S_1 \subseteq S$ be the set formed by the vertices of degree 1 in S. Let G' be the graph obtained by removing from G the vertices in $S - S_1$ and all edges incident to them. Run algorithm *tree* on graph G' and let T' be the tree that it finds. Note that all vertices of S_1 are leaves in T'. If any vertex $v \in S - S_1$ is adjacent to an internal vertex u of T' then v is placed as child of u in T'. Let T' be the

resulting tree. Let S^0 S be the set formed by the vertices in S which do not belong to T^0. Note that the neighbors of vertices in S^0 are all leaves of T^0.

We say that a subset C of leaves of T^0 *covers* the vertices in S^0 if every vertex in S^0 is adjacent to at least one vertex in C. Let C be a minimal subset of leaves of T^0 that covers S^0, i.e., for every vertex u 2 C there is at least one vertex v 2 S^0 such that u is the only neighbor of S^0 in C. To build a spanning tree for G, we place arbitrarily the vertices of S^0 as children of C.

Let C_1 C be the set of vertices in C with only one child and let S_1^0 be the children of C_1. Since C is a minimal cover for S^0 then every vertex in S_1^0 is adjacent to only one vertex in C. To see this assume that a vertex u 2 S_1^0 is adjacent to at least 2 vertices $v; w$ 2 C, where v 2 C_1. But then, $C - fvg$ would also cover S, which cannot happen since C is a minimal cover. By the same argument, every vertex in S_1^0 is adjacent to at least one vertex in $'(T^0) - C$.

Find a minimal set C_1^0 $'(T^0) - C$ that covers S_1^0. Note that $C - C_1 [C_1^0$ is a minimal cover for S^0. If $jC_1^0j < jC_1j$ then place the vertices in S_1^0 as children of C^0 instead of as children of C_1. We let T be the spanning tree formed by this algorithm.

Lemma 10. $'(T)$ $\max fjS^0j; '(T^0); \frac{2}{3}(j'(T^0)j + jS^0j)g.$

Proof. Trivially $'(T)$ jS^0j since all vertices in S^0 are leaves of T. Let C be the minimal cover for S^0 selected by our algorithm to attach the vertices of S^0 to the tree. Then jCj jS^0j, and so $'(T) = '(T^0) - jCj + jSj$ $'(T^0)$.

Let C_1 be the set formed by all vertices in C that have only one child, and S_1^0 be the set of children of C_1. Note that $jS^0 - S_1^0j$ $2jC - C_1j$ since every vertex in $C - C_1$ has at least two children from S^0. Also, since every vertex in S_1^0 is adjacent to one vertex in C_1 and to at least one vertex in $'(T^0) - C$, then by of the way in which C was chosen, $j'(T^0) - Cj$ jC_1j. Hence,

$$\begin{aligned} 3(jS^0 - S_1^0j + jC_1j + j'(T^0) - Cj) \quad & 2(jS^0 - S_1^0j + jC_1j + j'(T^0) - Cj) + \\ & 2jC - C_1j + jC_1j + jC_1j \\ & = 2(jS^0j - jS_1^0j + jC_1j + j'(T^0)j) \\ & = 2(jS^0j + j'(T^0)j); \text{ because } jS_1^0j = jC_1j : \end{aligned}$$

Since $'(T) = jS^0 - S_1^0j + jC_1j + j'(T^0) - Cj$, then $'(T)$ $\frac{2}{3}(jSj + j'(T^0)j)$. ∎

Given a graph G and a subset of vertices S let T be a spanning tree of G with maximum number of leaves and in which all vertices from S are leaves. Our algorithm finds a tree T with at least (2=5)-times the number of leaves in T .

Theorem 2. $'(T)='(T)$ 5=2.

Proof. Let S^0 S be as defined above and let G^0 be the graph obtained by removing from G all vertices in S^0 and all edges incident to them. Let T^+ be a spanning tree of G^0 with maximum number of leaves and such that all vertices in $S - S^0$ are leaves. Let T^0 be as defined above, by Theorem 1, $'(T^+)='(T^0)$ 2. Note that $'(T)$ $'(T^+) + jS^0j$, hence by Lemma 10:

1. if $|S'| \le \ell(T')/2$, then $\ell(T^*)/\ell(T) \le (\ell(T^+) + |S'|)/\ell(T') \le 2 + \frac{1}{2} = \frac{5}{2}$;
2. if $|S'| \ge 2\ell(T')$, then $\ell(T^*)/\ell(T) \le (\ell(T^+) + |S'|)/|S'| \le \ell(T^+)/(2\ell(T')) + 1 = 2$;
3. if $\ell(T')/2 < |S'| < 2\ell(T')$, then $\ell(T^*)/\ell(T) \le \frac{3}{2}(\ell(T^+)+|S'|)/(\ell(T')+|S'|) \le \frac{3}{2} + \frac{3}{2}\ell(T')/(\ell(T') + |S'|) \le \frac{3}{2} + 1 = \frac{5}{2}$. ∎

References

1. Duckworth W., Dunne P.E., Gibbons A.M., and Zito M.: Leafy spanning trees in hypercubes. Technical Report CTAG-97008 (1997) University of Liverpool.
2. Feige U.: A threshold of $\ln n$ for approximating set-cover. Twenty Eighth ACM Symposium on Theory of Computing (1996) 314–318.
3. Galbiati G., Maffiol F., and Morzenti A.: A short note on the approximability of the maximum leaves spanning tree problem. Information Processing Letters **52** (1994) 45–49.
4. Griggs J.R., Kleitman D.J., and Shastri A.: Spanning trees with many leaves in cubic graphs. Journal of Graph Theory **13** (1989) 669–695.
5. Guha S. and Khuller S.: Approximation algorithms for connected dominating sets. Proceedings of the Fourth Annual European Symposium on Algorithms (1996) 179–193.
6. Guha S. and Khuller S.: Improved methods for approximating node weight Steiner trees and connected dominating sets. Technical Report CS-TR-3849 (1997) University of Maryland.
7. Kleitman D.J. and West D.B.: Spanning trees with many leaves. SIAM Journal on Discrete Mathematics **4** (1991) 99–106.
8. Lu H. and Ravi R.: The power of local optimization: approximation algorithms for maximum-leaf spanning tree. Proceedings of the Thirtieth Annual Allerton Conference on Communication, Control, and Computing (1992) 533–542.
9. Lu H. and Ravi R.: A near-linear time approximation algorithm for maximum-leaf spanning tree. Technical report CS-96-06 (1996) Brown University.
10. Payan C., Tchuente M., and Xuong N.H.: Arbres avec un nombre de maximum de sommets pendants. Discrete Mathematics **49** (1984) 267–273.
11. Storer J.A.: Constructing full spanning trees for cubic graphs. Information Processing Letters **13** (1981) 8–11.

Moving-Target TSP and Related Problems*

C. S. Helvig[1], Gabriel Robins[1], and Alex Zelikovsky[1,2]

[1] Department of Computer Science, University of Virginia, Charlottesville, VA 22903
[2] Department of Computer Science, UCLA, Los Angeles, CA 90095

Abstract. Previous literature on the Traveling Salesman Problem (TSP) assumed that the sites to be visited are stationary. Motivated by practical applications, we introduce a time-dependent generalization of TSP which we call Moving-Target TSP, where a pursuer must intercept in minimum time a set of targets which move with constant velocities. We propose approximate and exact algorithms for several natural variants of Moving-Target TSP. Our implementation is available on the Web.

1 Introduction

The classical Traveling Salesman Problem (TSP) has been studied extensively, and many TSP heuristics have been proposed over the years (see surveys such as [4] [7]). Previous works on TSP have assumed that the cities/targets to be visited are stationary. However, several practical scenarios give rise to TSP instances where the targets to be visited are themselves in motion (e.g., when a supply ship resupplies patrolling boats, or when an aircraft must intercept a number of mobile ground units). In this paper, we introduce a generalization of the Traveling Salesman Problem where targets can move with constant velocities, formulated as follows:

> **The Moving-Target Traveling Salesman Problem**: Given a set $S = fs_1; : : : ; s_n g$ of *targets*, each s_i moving at constant velocity v_i from an initial position $p_i \ 2 \ <^n$, and given a *pursuer* starting at the origin and having maximum speed $v \ > \ jv_i j$ for $i = 1; : : : ; n$, find the fastest tour starting (and ending) at the origin, which intercepts all targets.

Related formulations consider *time-dependent* versions of the Vehicle Routing Problem, where one or more trucks whose speeds vary with time-of-day (due to traffic congestion) serve customers at fixed locations [5]. These works give exponential-time algorithms based on a mixed integer linear programming formulation and dynamic programming [5] [6] [9]. One major drawback of such general formulations is that they do not yield efficient bounded-cost heuristics, since they generalize TSP without the triangle inequality, which admits no efficient approximation bounds unless $P = NP$ [1].

* Professor Robins is supported by a Packard Foundation Fellowship and by National Science Foundation Young Investigator Award MIP-9457412. The corresponding author is Gabriel Robins, (804) 982-2207, robins@cs.virginia.edu. Additional related papers and our code may be found at `http://www.cs.virginia.edu/~robins`

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 453–464, 1998.
c Springer-Verlag Berlin Heidelberg 1998

In this paper, we address several natural variants of Moving-Target TSP. In Section 2, we show that unlike the classical TSP, the restriction of Moving-Target TSP to one dimension is not trivial, and we give an exact $O(n^2)$-time algorithm. For Moving-Target TSP instances where the number of moving targets is sufficiently small, we develop a $(1 + \ )$-approximation algorithm, where denotes the approximation ratio of the best classical TSP heuristic.

Next, in Section 3, we shift our attention to selected variants of Moving-Target TSP with Resupply[1], where the pursuer must return to the origin for resupply after intercepting each target, as shown in Figure 1(c). For these variants, we assume that targets are moving strictly away from (or else towards) the origin. We present a surprisingly simple exact algorithm for Moving-Target TSP when the targets approaching the origin are either far away or slow. We also consider the case when all targets are moving towards the origin, but with the additional requirement that the pursuer must intercept all of the targets before they reach the origin. We show that such a tour always exists and that no tour which satisfies the constraints is more than twice as long as the optimal tour.

Finally, in Section 4, we generalize Moving-Target TSP with Resupply to allow multiple pursuers which are constrained to all move with the same maximum speed. This problem also can be viewed as a dynamic generalization of multiprocessor scheduling where the processing time depends on the starting time of processing a job. We show that the problem is NP-hard, which is a nontrivial result because our formulation has a total time objective which is different from the standard "makespan" and average completion time objectives. We also develop an approximation algorithm for the case when, if projecting back in time, all targets would have left the origin simultaneously. Finally, we present an exact algorithm for the case when all targets have the same speed, and conclude in Section 5 with some future research directions. Some proofs are condensed or altogether omitted due to page limitations. See [2] or our Web site http://www.cs.virginia.edu/-robins/ for a more complete version of this paper.

2 Special Instances of Moving-Target TSP

Since unrestricted Moving-Target TSP is NP-hard[2], and because non-optimal tours can have unbounded error, we consider special variants where Moving-Target TSP is solvable either exactly or to within a reasonable approximation ratio. In this section, we consider two variants: (1) when targets are confined to

[1] Our formulation corresponds to a dynamic version of the Vehicle Routing Problem, defined as follows. Given a set of n *targets*, each with *demand* c_i moving at a constant velocity v_i from an initial position p_i 2 $<^n$ for $i = 1;:::;n$, and a set of k *pursuers* initially located at the origin and having maximum speed V_j and *supply* C_j for $j = 1;:::;k$, find a tour for each pursuer such that the demand of each target is satisfied and the total time of vehicles in operation is minimized.

[2] Note that classical TSP is a special case of Moving-Target TSP where all the velocities are zero.

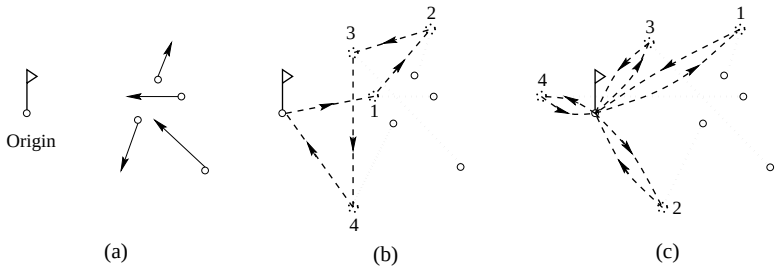


Fig. 1. (a) An instance of Moving-Target TSP. (b) A shortest-time tour (dashed line) which begins at the origin (flag) and intercepts all of the targets. (c) In Moving-Target TSP with Resupply, the pursuer must return to the origin after intercepting each target (the optimal interception schedule is in the order shown).

a single line[3], and (2) when the number of moving targets is small. The following lemma is essential for our analyses of Moving-Target TSP and its variants:

Lemma 1. *The pursuer must always move at maximum speed in any optimal Moving-Target TSP tour.*

2.1 Moving-Target TSP in One Dimension

In this subsection, we consider Moving-Target TSP where the pursuer and all targets are confined to a single line, and we develop an $O(n^2)$ exact algorithm based on dynamic programming. To see that this one-dimensional variant is not trivial, consider the following generalization of the exact algorithm when all targets are stationary. First, compute the cost of the tour which intercepts all targets to the left of the origin and then intercepts all targets to the right of the origin. Next, compute the cost of the tour which intercepts all targets to the right of the origin and then intercepts all targets to the left of the origin. Finally, from this pair of possible tours, choose the one with the least cost.

Unfortunately, this simple heuristic has unbounded error. Consider the case when there are four targets, two of them on either side of the origin, extremely close but moving away very quickly, while the other two targets are on either side of the origin, but much further from it, and are so slow as to be almost stationary. If we intercept the two fast targets immediately, then we spend almost no time in chasing them. However, if we first intercept all of the targets on one side of the origin, then we will later need to spend much more time chasing the fast target on the other side.

[3] Historical note: after we submitted this paper to ESA'98, we received an unpublished technical report [8] that claims to solve the one-dimensional variant in time $O(n^3)$; our algorithm for the same problem runs in time $O(n^2)$, based on a different approach.

Lemma 2. *In an optimal tour for one-dimensional Moving-Target TSP, the pursuer cannot change direction until it intercepts the fastest target ahead of it.*

Proof: Suppose towards contradiction that in an optimal tour T, the pursuer changes direction at time t before intercepting the fastest target s ahead of it. There exists some sufficiently small > 0 such that, in the time period between $t -$ and $t +$, the pursuer changes direction only at time t. Consider an alternative tour where the pursuer stops at time $t -$, waits without moving until time $t +$, and then continues along the original tour. All the targets that the pursuer would intercept in the time period between $t -$ and $t +$ are moving slower than s, and therefore cannot pass s. These targets will remain "sandwiched" between the pursuer and target s, and thus will automatically be intercepted later by the pursuer on its way to the faster target s. By Lemma 3, the original tour is not optimal, because it is equivalent to a tour with a waiting period. □

Thus, we may view a tour as a sequence of snapshots at the moments when the pursuer intercepts the fastest target which was ahead of it. We will later show that all of the relevant information in any such snapshot can be represented as a *state* $(s_k; s_f)$, where s_k is the target just intercepted, and s_f is the fastest target on the other side. All tours have the same initial state A_0 and the same final state A_{final}. Note that neither A_0 nor A_{final} have corresponding targets s_k or s_f. States have a time function associated with them, denoted $t(A)$, representing the shortest time in which this state can be achieved. Naturally, we assign $t(A_0) = 0$ for the initial state.

Note that Lemma 2.1 implies that there are at most two possible transitions from any state $A = (s_k; s_f)$ at assigned time $t(A)$. These two transitions represent the two possible choices of the next target to be intercepted, either the fastest to the left of the pursuer or the fastest to the right of the pursuer. The time of each transition , denoted $t(\ )$, is the time necessary for the pursuer to intercept the corresponding target (the fastest on the left or on the right) from the position of target s_k at time $t(A)$.

Our algorithm works as follows. In the preprocessing step of our algorithm, we partition the targets into two lists, *Left* and *Right* (according to whether targets are on the left or the right side of the origin, respectively). Then, we sort the lists according to non-increasing order of their speeds. We traverse both sorted lists and delete any target from the list which is closer to the origin than its predecessor. The targets which remain in these two lists are the only targets for which the pursuer may change direction after intercepting them.

We sort states in ascending order of the sum of the indices of the targets s_k and s_f (for each state $A = (s_k; s_f)$) in the *Left* and *Right* lists. Our algorithm iteratively traverses each state in this sorted list. Note that transitions cannot go backwards with respect to this order.

When we traverse a given state A in the algorithm, we do one of three things: (1) if the state has no transitions to it, then we proceed to the next state in the list; (2) if the pursuer has intercepted all of the targets on one side, we make a transition to the final state A_{final}; or (3) make two transitions

which correspond to sending the pursuer after either the fastest target on the left or the fastest target on the right. We define the time of a state B such that $t(B) = minft(A) + t(\ )\ j\quad : A\ !\ Bg$. In other words, $t(B)$ is the time required for the shortest sequence of transitions from A_0 to B. In this extended abstract, we omit the details for traversing each of the $O(n^2)$ states in constant (amortized) time.

Once we have visited each of the states, we can traverse the transitions backwards from the final state A_{final} back to the initial state A_0. This gives us the list of states which describes the turning points for the pursuer in the optimal tour. For each pair of turning points, we find the subset of targets which are intercepted between them. Once we have partitioned the targets into subsets, we sort the targets inside each subset by their interception times. Finally, we merge the sorted subsets into a combined interception order, yielding the solution.

Theorem 1. *The above algorithm for One-Dimensional Moving-Target TSP nds an optimal tour in $O(n^2)$ time.*

We have implemented this algorithm using the C++ programming language. Computational benchmarks against an exhaustive search algorithm empirically confirm its correctness (our implementation is available on the Web).

2.2 Heuristics when the Number of Moving Targets is Small

In this subsection, we consider Moving-Target TSP when most of the targets are stationary (while a few targets may be moving). From among the many existing approximation algorithms for classical (stationary) TSP [4], choose one such heuristic, having performance bound . Using this algorithm for stationary targets, we can construct an efficient algorithm (see Figure 2) with performance bound of $1 +$ when the number of moving targets is sufficiently small.

Theorem 2. *Moving-Target TSP where at most $O(\frac{\log n}{\log \log n})$ of the targets are moving can be approximated in polynomial time with performance bound $1 +$, where is a performance bound of an arbitrary classical TSP heuristic.*

3 Moving-Target TSP with Resupply After Intercepts

In this section, we consider Moving-Target TSP where a single pursuer must return to the origin after intercepting each target. We call this problem Moving-Target TSP with Resupply. We assume that targets all move either directly towards or away from the origin. These assumptions are natural because the projections of target velocities onto radial lines through the origin are approximately constant when targets are either (1) slow, (2) far from the origin, or (3) already moving along a radial line.

We define a *valid* tour to be a tour where no target passes near (or through) the origin without first being intercepted by the pursuer, and thus the velocities

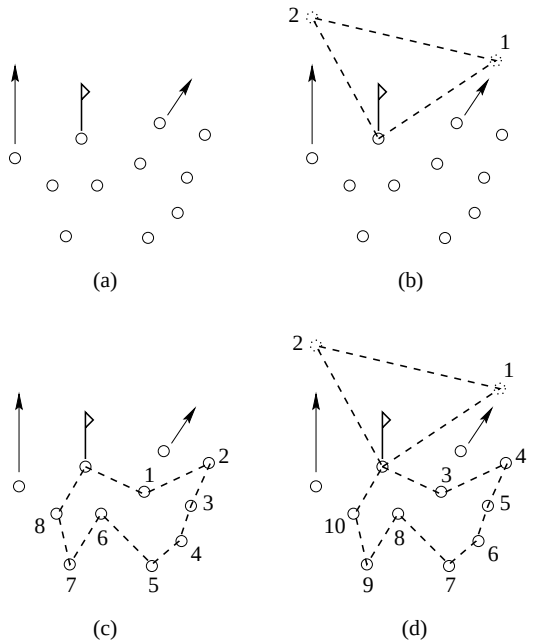


Fig. 2. (a) In this example, we have two moving targets and the rest are stationary. (b) First, we find the optimal tour for the moving points by trying all permutations. (c) Then, we use a classical TSP algorithm with performance bound for intercepting stationary targets. (d) Finally, we combine both tours into a single tour with performance bound $1 +$.

of all targets may be considered to be fixed with respect to the origin. In Subsection 3.1, we restrict the problem by considering the scenario when the shortest tour is valid. In other words, no target will pass the origin for a "sufficiently" long time (we will later elaborate on how long that is). We show that there is a simple way to determine the optimal target interception order for this scenario.

We consider a similar scenario in Subsection 3.2, except that we introduce a new constraint, namely that the pursuer must intercept each target before it reaches the origin (i.e., the tour *must* be valid). We formulate the problem in terms of "defending" the origin from the targets, and show that there is always a valid tour regardless of the number of targets. Then, we prove that the longest tour can be only twice the length of the shortest valid tour.

3.1 The Case when Targets Never Reach the Origin

Let d_i be the initial distance between the target s_i and the origin, i.e., $d_i = |p_i|$ where p_i is the position of s_i in the plane. If the target is moving towards the origin, then we define v_i to be negative, otherwise v_i is positive (see Figure 3).

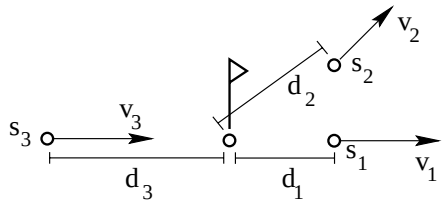


Fig. 3. Two targets, s_1 and s_2, are moving directly away from the origin at positive velocities v_1 and v_2, starting at distances d_1 and d_2. A third target s_3 approaches the origin, and thus its velocity v_3 is negative. The pursuer returns to the origin after intercepting each target.

First, we will determine the optimal order to intercept the receding targets, if all of the targets move away from the origin. Next, we will determine the order for intercepting targets when all targets move towards the origin. Finally, under certain conditions, we can combine the two orderings to obtain a single optimal ordering for a mix of approaching and receding targets.

Our solution for the case when targets move away from the origin is to intercept targets in increasing order of their ratios $\frac{d_i}{v_i}$. Note, that projecting backward in time, $\frac{d_i}{v_i}$ is the amount of time since the target intercepted the origin. Thus, the ratio $\frac{d_i}{v_i}$ corresponds to what we denote as the *age* of a target. The following theorem proves that the optimal algorithm must intercept the targets in non-decreasing order of their ages (i.e., younger targets are intercepted first).

Theorem 3. *In Moving-Target TSP with Resupply when all targets move directly away from the origin, the optimal tour intercepts the targets in non-decreasing order of their respective ratios* $\frac{d_i}{v_i}$.

Proof: First, we show that the theorem is true when an instance of Moving-Target TSP with Resupply contains only two targets. Let $t_{1;2}$ be the time required to intercept s_1 and then intercept s_2. Similarly, let $t_{2;1}$ be the time required to intercept s_2 and then intercept s_1. We assume that s_1 is younger (the other case is symmetrical). We would like to show that $t_{1;2}$ $t_{2;1}$. The time required for the pursuer to intercept s_1 is $\frac{2\,d_1}{v - v_1}$. Afterwards, intercepting s_2 will take time $2\ (d_2 + \frac{2\,d_1}{v - v_1}\ v_2)/(v - v_2)$. Algebraic manipulation yields $t_{1;2} - t_{2;1} = \frac{4\,d_1\,v_2 - 4\,d_2\,v_1}{(v - v_1)\,(v - v_2)}$. If s_1 is younger than s_2, then $\frac{d_1}{v_1}$ $\frac{d_2}{v_2}$ and $d_1\ v_2$ $d_2\ v_1$. This proves that $t_{1;2}$ $t_{2;1}$.

Given that Theorem 3 is true for two targets, we show that it is also true for any number of targets. Assume towards the contradiction that in the optimal tour, the pursuer intercepts two consecutive targets, first s_i and then s_j, in non-increasing order of their ages, i.e., $\frac{d_i}{v_i}$ $\frac{d_j}{v_j}$. First intercepting s_i and then intercepting s_j will require no less time than intercepting them in the reverse order, namely s_j first and then s_i. Thus, if the pursuer would alternatively intercept these two targets s_i and s_j in the reverse order, it would have time to

wait at the origin right after intercepting the second target, and then continue the rest of the original tour using the original interception order. But by Lemma 3, this means that the original (presumably optimal) tour is not optimal, since it can be improved. Thus, all pairs of consecutive targets in an optimal tour must be intercepted in nondecreasing order of their $\frac{d_i}{v_i}$ ratios. □

Next, we analyze an analogous variant where all targets are approaching the origin. This variant is essentially the time reversal of the previous variant where all targets are receding. The concept of the "age" of a target, however, is replaced with the concept of the "dangerousness" of a target. The problem of intercepting targets moving towards the origin can thus be reformulated as requiring a pursuer to *defend* the origin (against, e.g., incoming missiles).

The difference between the time reversal of the resupply variant where all targets move away from the origin and the case when all targets move towards the origin, is that a target may pass through the origin and then move away from it. This possibility makes the scenario quite complicated, because it causes an implicit change in direction which is absent in the first variant. Therefore, we consider only valid tours where no targets pass through the origin before the pursuer intercepts them.

Lemma 3. *Let all targets move towards the origin, and let T be the tour which intercepts targets in non-decreasing order of their respective ratios $\frac{d_i}{v_i}$. If T is a valid tour, then it is an optimal tour for Moving-Target TSP with Resupply.*

Note that for a mixture of approaching and receding targets, we should intercept the receding targets first. The longer we wait to intercept these targets, the further away they will be when we do intercept them. Targets that move towards the origin should be allowed as much time as possible to come close to the origin. Therefore, if we assume that no targets will pass through the origin while we are pursuing the targets that move away from the origin, then we should first intercept targets that are moving away from the origin, and then intercept targets that are moving towards the origin. Further, if we can intercept the receding targets and still intercept the approaching targets in increasing order of their dangerousness before any target crosses the origin, then the optimal tour for intercepting all of the targets is to first intercept the receding targets in order of increasing age and then to intercept the approaching targets in order of increasing dangerousness. This is formalized in the following theorem.

Theorem 4. *Let T be the tour which first intercepts targets moving away from the origin in non-decreasing order of their ratios $\frac{d_i}{v_i}$ and then intercepts the targets moving towards the origin in non-decreasing order of their ratios $\frac{d_i}{v_i}$. If T is a valid tour, then it is an optimal tour for Moving-Target TSP with Resupply.*

3.2 "Defending" the Origin Against Approaching Targets

In this subsection, we consider Moving-Target TSP when all targets approach the origin. We first show that if we intercept targets in order of most dangerous

to least dangerous, we will intercept all of the targets before any of them reach the origin. Next, we observe that from among all tours which intercept all targets before they reach the origin, the tour that intercepts targets in order of most dangerous to least dangerous is the longest. Finally, we can show that even this longest tour is never longer than twice the optimal tour which intercepts all targets before they reach the origin.

Although we can prove that the strategy of intercepting targets in order of least dangerous to most dangerous is optimal when no targets intercept the origin, it is still open whether there is an efficient algorithm for determining the optimal intercept order when some targets may pass through the origin before being intercepted. We can prove, however, that there is always a tour that intercepts all targets before they reach the origin.

Theorem 5. *The tour that intercepts all targets in non-increasing order of* $\frac{d_i}{v_i}$ *is the slowest valid tour.*

Lemma 3.1 and Theorem 5 lead to a natural question: what is the shortest valid tour? (i.e., the tour which intercepts all targets, yet prevents any targets from passing the origin). A natural strategy would be to always intercept the least dangerous target unless intercepting that target would allow the most dangerous target to reach the origin. In this case, we would intercept the least dangerous target that we can intercept and still obtain a valid tour. Unfortunately, this simple algorithm does not always return an optimal tour.

There are instances of the Moving-Target TSP for which the slowest tour may be twice as long as optimal. We can also prove that this bound is tight, i.e., no valid tour takes time more than twice the optimal valid tour.

Theorem 6. *For Moving-Target TSP with Resupply, when all targets move towards the origin, no valid tour is more than twice the optimal valid tour.*

Proof: We enumerate the targets in order of least dangerous to most dangerous. Let T be an optimal valid tour. The slowest valid tour intercepts targets in order of most to least dangerous (i.e., $s_n; \ldots; s_1$). We will show that the slowest tour can be no more than twice the length of the optimal valid tour T by iteratively transforming T into the slowest tour. Note that this transformation is equivalent to sorting, since the optimal tour can intercept targets in any order, and in the slowest order, the targets are sorted in decreasing order of their indices.

We write the tour T as a list of targets where the left-most targets will be intercepted first and the right-most target will be intercepted last. Our transformation starts with the right-most target in the original optimal tour T and gradually moves to the left, sorting all targets in decreasing order of their indices. In other words, at each step of our transformation, the current target, say s_i, and all targets to the left of s_i hold the same positions as in the tour T, while all of the targets to the right of s_i are already sorted in decreasing order of their indices. The step consists of removing target s_i from the current tour and inserting it into its proper position in the sorted list to the right.

Let t_i be the time required to intercept the target s_i in the original optimal tour T. Now, we show that each step of the transformation increases the total time of the tour by at most t_i. Note first that removing target s_i from the current tour cannot increase the total time of the tour. Indeed, the pursuer may wait for time t_i instead of intercepting the target s_i. Inserting the target s_i into its proper place in the sorted list decreases the time for intercepting targets to the right of its new location, because they will be intercepted later, i.e., when they will be closer to the origin. Similarly, the time to intercept the target s_i in its new position is at most t_i. Thus, inserting can increase the total time of the tour by at most t_i. Since each step of the transformation increases the cost of the tour by t_i for all s_i, the final tour may be at most twice the original cost. □

4 Multi-Pursuer Moving-Target TSP with Resupply

In this section, we address a generalization of Moving-Target TSP with Resupply when there are multiple pursuers. This generalization can also be considered as a dynamic version of the well-known Vehicle Routing and Multiprocessor Scheduling Problems. We restrict ourselves to the case when targets move strictly away from the origin and all k pursuers have equal speed (normalized to 1).

In the presence of multiple pursuers, Moving-Target TSP may have different time objectives. In the Vehicle Routing Problem, the typical objective has been to minimize the *total tour time*, i.e., to minimize the sum over all pursuers of all periods of time in which a pursuer is in operation[4]. In multiprocessor scheduling, a common objective has been to minimize the *makespan*, or in other words, to minimize the time when the last pursuer returns to the origin.

Note that achieving the makespan objective may be more computationally difficult than the total time objective, since for stationary targets, minimizing the makespan is equivalent to the NP-hard Multiprocessor Scheduling Problem, whereas the total time objective is invariant over all schedules. In the presence of moving targets, the problem of minimizing the total time also becomes NP-hard. To show this, we need the following lemma.

Lemma 4. *Let $\frac{d_i}{v_i} = t$ be the same for all targets s_i; where $i = 1;:::;n$. For each target s_i, let $u_i = \frac{1+v_i}{1-v_i}$. The time required to intercept targets $s_1;:::;s_p$ with one pursuer is equal to $t \cdot (\prod_{i=1}^{p} u_i - 1)$.*

Lemma 4 implies that the problem of distributing n targets between two pursuers includes as a special case the well-known NP-hard problem of partitioning of a set of numbers into two subsets with each having the same sum.

Theorem 7. *Moving-Target TSP with Two Pursuers and Resupply is NP-hard when the objective is to minimize the total time.*

[4] We assume that each pursuer is in operation starting from the time $t = 0$ until its final return to the origin.

4.1 Targets with the Same Age

Lemma 4 yields a reduction of Moving-Target TSP with k pursuers (in the case when all targets have the same age) to the standard Multiprocessor Scheduling Problem: given a set of n jobs with processing times t_i and k equivalent processors, find a schedule having the minimum makespan.

There are several different heuristics for the Multiprocessor Scheduling Problem: list scheduling, longest processing, and many others, including a polynomial-time approximation scheme [3]. Unfortunately, the error estimates for these heuristics cannot be transformed into bounds for Moving-Target TSP with Resupply and multiple pursuers because a multiplicative factor corresponds to the exponent in such transformations. A tour in which the next available pursuer is assigned to the next target (to which no pursuer has yet been assigned) from the list will be called a *list tour*.

Theorem 8. *Let $\frac{d_i}{v_i} = t$ be the same for all targets s_i, for $i = 1;:::;n$. Then the list tour takes total time at most $\max_{i=1;:::;n} \frac{1+v_i}{1-v_i}$ times optimal.*

This theorem implies that if the speed of any target is at most half the speed of pursuer, then the list tour interception order has an approximation ratio of 3.

4.2 Targets Moving with Equal Speed

In the case when there are multiple pursuers and all targets have the same speed v, we can efficiently compute an optimal solution. Similarly to the method outlined above, we order the targets by increasing value of their $\frac{d_i}{v_i}$, but since $v_i = v$ is the same for all targets, this reduces to ordering the targets by increasing initial distance from the origin. We have found that a very natural strategy suffices: send each pursuer after the next closest target to the origin at the time when pursuer resupplies at the origin. We call the resulting tour "CLOSEST", and we can prove that it is optimal.

Theorem 9. *The tour CLOSEST is the optimal tour for Moving-Target TSP with multiple pursuers when targets have equal speeds.*

Proof sketch: It can be shown that the cost of a tour for a single pursuer to intercept n targets having equal speeds $v_i = v$ is $T_n = \sum_{i=0}^{n} t_i \; c^{n-i+1}$ where $c = 1 + 2 \; \frac{v}{1-v}$ and $t_i = 2 \; \frac{d_i}{1-v}$ (in other words, t_i is the time for a pursuer to intercept a target if it intercepted that target first). In the tour CLOSEST, each of the k pursuers will intercept targets $s_i; s_{k+i};:::; s_{k[n=k]+i}$, where $i = 1;:::;k$. Given a pair of targets which are intercepted by different pursuers such that each pursuer has an equal number of targets left to pursue after intercepting them, the targets may be swapped between the two pursuers (i.e., each pursuer may intercept its counterpart's target instead of its own), while keeping the total time required the same. This allows us to transform any optimal tour into the tour CLOSEST, by using a sequence of target swappings between different pursuers, without increasing the total tour time. Therefore, the tour CLOSEST requires the optimal total time. □

5 Conclusion and Future Research

We formulated a Moving-Target version of the classical Traveling Salesman Problem, and provided the first heuristics and performance bounds for this problem and for other time-dependent variants. Topics for future research include providing approximation algorithms for more general variants of Moving-Target TSP (e.g., where targets are moving with non-zero accelerations, and/or along non-linear paths). Also, it would be interesting to generalize our results for Moving-Target TSP with Resupply to cases when each pursuer may service multiple targets before requiring resupply. Alternatively, any non-approximability results for such cases would be of interest as well.

6 Acknowledgements

We thank Piotr Berman for his help in developing the $O(n^2)$ dynamic programming algorithm for one-dimensional Moving-Target TSP. We also thank Mike Nahas for his implementation and for finding areas where additional clarification was needed. We appreciate the participation of Bhupinder Sethi and Doug Bateman in numerous fruitful discussions. Finally, we thank John Karro, Tongtong Zhang, Doug Blair, and Anish Singh for their help with proofreading.

References

1. T. H. Cormen, C. E. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
2. C. H. Helvig, G. Robins, and A. Zelikovsky. Moving target tsp and related problems. Technical Report CS-98-07, Department of Computer Science, University of Virginia, December 1997.
3. D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1995.
4. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy, and D. B. Shmoys. *The Traveling Salesman Problem: a Guided Tour of Combinatorial Optimization*. John Wiley and Sons, Chichester, New York, 1985.
5. C. Malandraki and M. S. Daskin. Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Journal of Transportation Science*, 26:185–200, August 1992.
6. C. Malandraki and R. B. Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal of Operations Research*, 90:45–55, 1996.
7. G. Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*. Springer Verlag, Berlin, Germany, 1994.
8. G. Rote. The traveling salesman problem for moving points on a line. Technical Report 232, Technische Universitat Graz, 1992.
9. R. J. Vander Wiel and N. V. Sahinidis. Heuristic bounds and test problem generation for the time-dependent traveling salesman problem. *Journal of Transportation Science*, 29:167–183, May 1995.

Fitting Points on the Real Line and Its Application to RH Mapping

Johan Håstad, Lars Ivansson, and Jens Lagergren

Department of Numerical Analysis and Computing Science
Royal Institute of Technology
SE-100 44 Stockholm
SWEDEN
fjohanh, ivan, jenslg@nada.kth.se

Abstract. The MATRIX-TO-LINE problem is that of, given an $n \times n$ symmetric matrix D, finding an arrangement of n points on the real line such that the so obtained distances agree as well as possible with the by D specified distances, *w.r.t.* the max-norm. The MATRIX-TO-LINE problem has previously been shown to be NP-complete [11]. We show that it can be approximated within 2, but not within 4=3 unless P=NP. We also show tight bounds under a stronger assumption. We show that the MATRIX-TO-LINE problem cannot be approximated within $2 - \epsilon$ unless 3-colorable graphs can be colored with $\lceil 4=\epsilon \rceil$ colors in polynomial time. Currently, the best polynomial time algorithm colors a 3-colorable graph with $\tilde{O}(n^{3=14})$ colors [4].
We apply our MATRIX-TO-LINE algorithm to a problem in computational biology, namely, the Radiation Hybrid (RH) problem, i.e., the algorithmic part of a physical mapping method called RH mapping. This gives us the first algorithm with a guaranteed convergence for the general RH problem.

1 Introduction

We study the MATRIX-TO-LINE problem, that is, the problem of, given a set of n points $\{p_i\}_{i=1}^n$ and an $n \times n$ distance matrix (i.e., positive, symmetric, and with an all zero diagonal) D, finding an arrangement $A : \{p_i\}_{i=1}^n \to \mathbb{R}^+$ which minimizes

$$\max_{i,j \in [n]} |D[i,j] - |A(p_i) - A(p_j)|| \tag{1}$$

over all such functions. This problem has previously been shown to be NP-complete [11]. We give an algorithm that approximates it within a factor 2. In contrast to this, we show that the MATRIX-TO-LINE problem cannot be approximated within a factor $4/3$ unless P=NP. We also show that the MATRIX-TO-LINE problem cannot be approximated within $2 - \delta$ in polynomial time, unless 3-colorable graphs can be colored with $\lceil 4/\delta \rceil$ colors in polynomial time.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 465–476, 1998.
© Springer-Verlag Berlin Heidelberg 1998

It is NP-hard to find a 4-coloring of a 3-colorable graph [9]. The problem of k-coloring a 3-colorable graph is not known to be NP-hard for k 5. However, it is a very well studied problem and despite this there is currently no polynomial time algorithm that colors a 3-colorable graph with less than $\tilde{O}(n^{3=14})$ colors [4].

Sufficient conditions and non-polynomial time algorithms for a more general form of the Matrix-To-Line problem have been given earlier [5]. The Matrix-To-Line problem is an example of a general type of problems, where a distance matrix D for n points is given, and the points should be embedded in some metric space. The goal is to embed the points so that the obtained distances are as close as possible to the distances specified by D (with respect to some norm). Polynomial time exact algorithms and approximation algorithms have previously been given for other examples of this general type of problems [1,7].

We apply our Matrix-To-Line algorithm to a physical mapping problem. Physical mapping is an important problem used in large-scale sequencing of DNA as well as for locating genes. Using RH mapping (which is described in Section 4) one can construct a physical map of, for instance, a human chromosome with respect to n markers, which can be genes or arbitrary DNA sequences; that is, one can find the order between these markers and the distance between them on the chromosome by performing a series of experiments and then performing an algorithmic analysis of the outcomes. However, experiments are costly and for this reason one naturally strives to perform as few as possible.

By applying our Matrix-To-Line algorithm, we obtain the first algorithm with a guaranteed convergence rate for the case of the RH problem where no prior lower bound is known on the minimum distance between two markers. Most previous algorithms, see for instance [3,10,13], are heuristics that do not guarantee convergence. In [2], Ben-Dor and Chor showed that after approximately $\delta_{\min}^{-2} \log n$ experiments, where $\delta_{\min}$ is a lower bound on the minimum marker distance, the laboratory data is with high probability, what they call, consistent. They also show that a number of rather straightforward algorithms always find the correct marker order when given consistent laboratory data; and so they obtain an algorithm that given a prior lower bound on $\delta_{\min}$ with high probability finds the correct marker order. We can prove that given as many samples as they need to obtain consistency, our algorithm finds the correct marker order as well, with high probability.. In Ben-Dor's and Chor's (and also our) model $\delta_{\min}$ $1/(n - 1)$. If $B(n)$ is the number of experiments needed to get a guaranteed consistency by [2] and $I(n)$ is the number of experiments needed to make our algorithm converge, then $B(n)$ $I(n)\Omega(n^2$), for $\delta_{\min}$ $1/n^{1+}$.

The remainder of this paper is organized as follows. In Section 2, the 2-approximation algorithm for the Matrix-To-Line problem is presented. In Section 3 the lower bound 4/3 on the approximability of the Matrix-To-Line problem is proven. It is also shown that Matrix-To-Line cannot be approximated within $2 - \delta$ in polynomial time, unless 3-colorable graphs can be colored with $d4/\delta e$ colors in polynomial time. In Section 4, a probabilistic model of an RH experiment is given. Finally, in Section 5, we show how our Matrix-To-Line algorithm can be applied to yield an algorithm for the RH problem.

2 Matrix-To-Line

In this section, we give an approximation algorithm for the MATRIX-TO-LINE problem.

Definition 1. *For two $n \times n$ matrices D and D', we define*

$$\|D, D'\|_1 = \max_{i,j \in [n]} |D[i,j] - D'[i,j]|. \tag{2}$$

An arrangement A is a mapping from a set of points $\{p_i\}_{i=1}^n$ to $\mathbb{R}^+$. To each arrangement we can associate a distance matrix D_A defined by $D_A[i,j] = |A(p_i) - A(p_j)|$. To avoid multiple subscripts we will abuse the notation above and write $\|D, A\|_1$ for $\|D, D_A\|_1$, and $\|A, A'\|_1$ for $\|D_A, D_{A'}\|_1$.

Let D be a given $n \times n$ distance matrix and A an optimal solution to the MATRIX-TO-LINE instance given by D. Let ϵ be defined by $\epsilon = \|D, A\|_1$. Furthermore, assume that we know that p_1 is the leftmost point in A and that $A(p_1) = 0$.

Let A^1 be the arrangement we get if we arrange all points according to the leftmost point p_1, i.e., $A^1(p_1) = 0$ and $A^1(p_i) = D[1,i]$ for $i = 2, 3, \ldots, n$.

Lemma 1. *$|A^1(p_i) - A(p_i)| \leq \epsilon$ for all points p_i.*

Proof. Follows from the fact that $\|A(p_1) - A(p_i)| - D[1,i]| \leq \epsilon$.

A corollary to Lemma 1 is that A^1 approximates the optimal arrangement within a factor 3.

Corollary 1. $\|A^1, D\|_1 \leq 3\epsilon$.

Proof. $|A^1(p_i) - A^1(p_j)| - D[i,j]| \leq |A(p_i) - A(p_j)| - D[i,j]| + 2\epsilon \leq 3\epsilon$.

Our algorithm is based on the observation that if we can modify the arrangement A^1 so that each point p_i, $i > 1$, is moved a distance $\epsilon/2$ to the side of $A^1(p_i)$ where $A(p_i)$ is located, the new arrangement will have the error 2ϵ. To each point p_i, for $i > 1$, we therefore associate a 0/1-variable x_i; where $x_i = 1$ corresponds to p_i being moved to the right of $A^1(p_i)$ and $x_i = 0$ corresponds to p_i being moved to the left. Given a truth assignment $S : x_i \to \{0,1\}$ for the set of variables $x_2, \ldots, x_n$, and an $\epsilon > 0$ we define the following arrangement.

Definition 2.

$$A^S(p_i) = \begin{cases} 0 & \text{if } i = 0, \\ A^1(p_i) - \epsilon/2 + S(x_i)\epsilon & \text{otherwise.} \end{cases} \tag{3}$$

Let S be the truth assignment defined by $S(x_i) = 1$ if $A(p_i) > A^1(p_i)$ and $S(x_i) = 0$ otherwise ($i > 1$).

Lemma 2. $\|A^S, D\|_1 \leq 2\epsilon$

Proof. For an arbitrary pair of points p_i, p_j we have that

$$\begin{aligned} &|A^S(p_i) - A^S(p_j)| \\ &\quad \le |A^S(p_i) - A(p_i)| + |A^S(p_j) - A(p_j)| + |A(p_i) - A(p_j)| \\ &\quad \le |A(p_i) - A(p_j)| + \epsilon \ . \end{aligned}$$

This implies that

$$|\,|A^S(p_i) - A^S(p_j)| - D[i,j]\,| \le |\,|A(p_i) - A(p_j)| - D[i,j]| + \epsilon \le 2\epsilon \ ,$$

which means that $|\,|A^S, D|| \le 2\epsilon$.

It is not necessary for us to find the truth assignment S . It will do with any assignment S for which $|\,|A^S, D|| \le 2\epsilon$, for some $\epsilon \le \epsilon$. For each pair of points p_i and p_j there are four possible ways to assign values to the variables x_i and x_j.

Definition 3. *An assignment S is ϵ-allowed for the pair of variables x_i and x_j if $|\,|A^S(p_i) - A^S(p_j)| - D[i,j]| \le 2\epsilon$.*

We thus need an assignment that is ϵ-allowed for all pairs of variables.

Lemma 3. *Let S be a truth assignment for the variables x_i, $i = 2, \ldots, n$, such that S is ϵ-allowed for all pairs x_i, x_j. Then $|\,|A^S, D|| \le 2\epsilon$.*

Proof. Follows immediately from Definition 3.

It is easy to construct a 2-SAT-clause that forbids a certain assignment for a pair x_i, x_j. For instance, the clause $(\overline{x_i} \vee x_j)$ forbids the assignment $x_i = 1$, $x_j = 0$. If we create a 2-SAT-formula forbidding all non-ϵ-allowed assignments, any satisfying assignment to that formula will have the property we are looking for.

Theorem 1. *For each pair x_i, x_j, let $\varphi_{i;j}$ be the conjunction of the at most four clauses forbidding all non-ϵ-allowed assignments for the pair, and let $\Phi = \bigwedge_{i<j} \varphi_{i;j}$. Every satisfying assignment S to Φ satisfies $|\,|A^S, D|| \le 2\epsilon$. Furthermore S is a satisfying assignment for Φ .*

Proof. Let S be any satisfying assignment for Φ . By construction, S is ϵ-allowed for each pair of variables x_i, x_j. From Lemma 3 follows immediately that $|\,|A^S, D|| \le 2\epsilon$.

From Lemma 2 we have that $|\,|A^S, D|| \le 2\epsilon$ so S is ϵ -allowed for all pairs of points and thus a satisfying assignment for Φ .

The following lemma, for which we only sketch the proof, reveals a useful property of the formulas Φ .

Lemma 4. *If c is a clause in Φ , then c is a clause in Φ' for all $0 \le \epsilon' \le \epsilon$.*

Proof. We only give an outline of the proof. It suffices to show that the property of an assignment being ϵ-allowed for a pair of variables is monotone in ϵ. If $|A^1(p_i) - A^1(p_j)| = D[i,j]$, every truth assignment for x_i and x_j is ϵ-allowed for all $\epsilon > 0$. If $|A^1(p_i) - A^1(p_j)| \neq D[i,j]$ every assignment is non-allowed for sufficiently small values of $\epsilon > 0$. As the value of ϵ increases each assignment will eventually be ϵ-allowed, and remain in this state as ϵ increases further.

We say that $\epsilon' \in \mathbb{R}^+$ is a *breakpoint* if $\Phi_{\epsilon'} \neq \Phi$ for all $\epsilon < \epsilon'$. From Lemma 4 follows that if S is a satisfying assignment to Φ, then S also satisfies $\Phi_{\epsilon'}$ for all $\epsilon' > \epsilon$. This means that if there is only a small number of breakpoints we could use binary search over the breakpoints to find an ϵ ϵ for which Φ has a satisfying assignment. (Note that S is a satisfying assignment for the 2-SAT-formula corresponding to the greatest breakpoint ϵ.)

Theorem 2. *There can be at most* $3\binom{n}{2}$ *breakpoints.*

Proof. There are $\binom{n}{2}$ pairs of variables x_i, x_j. There are at most 4 possible clauses for each pair x_i, x_j. From Lemma 4 follows that the status of each clause changes at most once. However, the clauses forbidding assignments where $S(x_i) = S(x_j)$ behave in exactly the same way as we change ϵ. This makes the total number of breakpoints at most $3\binom{n}{2}$.

We are now ready to formulate an algorithm

Algorithm 1. *(1) Construct the set of breakpoints. (2) Use binary search over the breakpoints to find the smallest breakpoint ϵ for which Φ has a satisfying assignment S. (3) Return A^S.*

Theorem 3. *Algorithm 1 approximates* MATRIX-TO-LINE *within 2 in time* $O(n^2 \log(n))$*, if the leftmost point in an optimal arrangement is known.*

Proof. The correctness of the algorithm follows from the derivations above. What we need to show is the time bound. Given the leftmost point in the optimal arrangement, we can compute the $O(n^2)$ breakpoints in time $O(n^2)$ and sort them in time $O(n^2 \log(n))$. In each step of the binary search we construct and solve a 2-SAT-formula with $O(n^2)$ clauses. 2-SAT can be solved in linear time, so the total time for that part of the algorithm is $O(n^2 \log(n))$. Hence the total running time of Algorithm 1 is $O(n^2 \log(n))$.

If the leftmost point in an optimal arrangement is unknown we can try all possible choices to find the correct one. The number of choices is of course always n so the running time will never be worse than $O(n^3 \log(n))$. However, simple heuristics should limit the number of choices considerably in most cases.

3 Lower Bounds

In this section, we first prove a lower bound of 4/3 on the approximability of MATRIX-TO-LINE under the assumption P $\neq$ NP; second, we show that

if MATRIX-TO-LINE is approximable within $2 - \delta$ in polynomial time, then every 3-colorable graph can be $\lceil 4/\delta \rceil$-colored in polynomial time. The problem of k-coloring a 3-colorable graph is a well studied problem. The problem is not known to be NP-hard. In fact, the best result so far is from [9] where they show that it is NP-hard to find a 4-coloring of a 3-colorable graph. However, the best approximation algorithm known for the corresponding optimization problem MINIMUM GRAPH COLORING for 3-colorable graphs, has performance ratio $\tilde{O}(n^{3/14})$ [4].

Our lower bound of 4/3 is obtained by a reduction from NOT-ALL-EQUAL-3-SAT, which is defined as follows. Given a set X of variables and a collection C of clauses over X such that each clause $c \in C$ has three literals. Then (X, C) belongs to NOT-ALL-EQUAL-3-SAT if and only if there is a truth assignment that for each clause c of C assigns at least one literal of c the value true and at least one literal of c the value false.

Deciding NOT-ALL-EQUAL-3-SAT was shown to be NP-complete by Schaefer [12].

Let $C = \{c_1, \ldots, c_m\}$ be a set of clauses over a set $X = \{x_1, \ldots, x_n\}$ of variables, such that c_i has three literals for each i. To each variable x and its complement $\bar{x}$ we associate two points p_x and $p_{\bar{x}}$. To each clause c we associate three points c_1, c_2 and c_3. We also introduce an additional point b. The intuition behind the construction is that a point being to the right of b corresponds to the associated literal being true, and analogously a point being to the left of b corresponds to the associated literal being false.

Note that by definition, if (X, C) does not belong to NOT-ALL-EQUAL-3-SAT, then for every truth assignment there is at least one clause such that all literals are true or all literals are false. Let $c = (u \_ v \_ w)$ be a clause in C. We choose the distance matrix D in such a way that if we arrange p_u, p_v and p_w so that at least one is to the right and at least one is to the left of b, it is possible to find locations for c_1, c_2 and c_3 so that the error is exactly 3; but if p_u, p_v and p_w are on the same side of b, the error will be at least 4 no matter how we arrange c_1, c_2 and c_3. For any point p, let $D[b, p] = 0$; for all $x \in X$, let $D[p_x, p_{\bar{x}}] = 9$; and for each clause $(u \_ v \_ w) \in C$, let $D[c_1, p_u] = 6$, $D[c_1, p_v] = 0$, $D[c_2, p_v] = 6$, $D[c_2, p_w] = 0$, $D[c_3, p_w] = 6$, $D[c_3, p_u] = 0$, $D[c_1, c_2] = 6$, $D[c_1, c_3] = 6$, $D[c_2, c_3] = 6$. For all other pairs of points p and q, let $D[p, q] = 3$.

Theorem 4. *An instance (C, X) belongs to* NOT-ALL-EQUAL-3-SAT *if and only if the associated* MATRIX-TO-LINE *problem has optimal value 3. Furthermore, if (C, X) does not belong to* NOT-ALL-EQUAL-3-SAT*, then the optimal value is* 4.

Proof. We only give an outline of the proof. Let $c = (u \_ v \_ w)$ be a clause such that u is true, and v, w are false. Figure 1 shows an arrangement of the associated points where the error is 3. The other cases with not all literals equal are similar.

It is obvious that any configuration with p_x and $p_{\bar{x}}$ on the same side of b has error > 4. Using linear programming on all permutations of the points p_u, p_v,

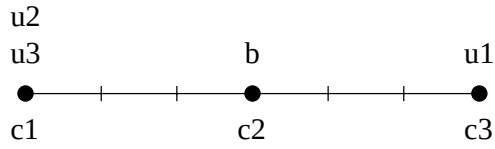


Fig. 1. The arrangement of a clause $c = (u \_ v \_ w)$, where u is true and v, w are false.

p_w, $p_{\bar{u}}$, $p_{\bar{v}}$, $p_{\bar{w}}$, c_1, c_2, c_3 and b, such that p_u, p_v, p_w are on one side of b and $p_{\bar{u}}$, $p_{\bar{v}}$, $p_{\bar{w}}$ on the other, we observe that the error is 4 in every case. Assume that $(X, C) \notin$ NOT-ALL-EQUAL-3-SAT. For every truth assignment, there is at least one clause such that all literals are true or all literals are false. It follows that any arrangement of the points will have an error 4. We conclude that the theorem holds.

Corollary 2. *It is NP-hard to approximate* MATRIX-TO-LINE *within* 4/3.

Now we show that it is at least as hard to approximate MATRIX-TO-LINE within $2 - \delta$ as it is to $\lceil 4/\delta \rceil$-color a 3-colorable graph. Let $G = (V, E)$ be any 3-colorable graph. For each vertex $v_i \in V$ we introduce a point p_i. For each pair of points we specify a distance $D[i, j]$ by $D[i, j] = 2$ if $(v_i, v_j) \in E$, and $D[i, j] = 1$ otherwise.

Let $c : V \to \{0, 1, 2\}$ be a 3-coloring of G. It is easy to check that for the arrangement A_c, defined by $A_c(p_i) = c(v_i)$, holds that $\|A_c, D\|_1$ 1. Assume that we can approximate MATRIX-TO-LINE within $2 - \delta$. This means that we can find an arrangement A such that $\|A, D\|_1$ $2 - \delta$.

W.l.o.g. assume that p_1 is the leftmost point in the arrangement A and that $A(p_1) = 0$. Since $D[1, i]$ 2 for all i, we conclude that $A(p_i)$ $4 - \delta$ for all i. Subdivide the interval $[0, 4 - \delta/2)$ into $\lceil 4/\delta \rceil$ intervals of equal length $d < \delta$. Consider the $\lceil 4/\delta \rceil$-coloring c_A of the vertex set V of G defined by $c_A(v_i) = j$ if $A(p_i) \in [(j - 1)d, jd)$.

Lemma 5. *c_A is a proper $\lceil 4/\delta \rceil$-coloring of G.*

Proof. We need to show that $c_A(p_i) \neq c_A(p_j)$ whenever $(v_i, v_j) \in E$. Assume that $(v_i, v_j) \in E$. This implies that $D[i, j] = 2$. Now, $\|A, D\|_1$ $2 - \delta$ so in this case $|A(p_i) - A(p_j)|$ $\delta > d$, which means that $c_A(p_i) \neq c_A(p_j)$.

We have thus proven the following theorem.

Theorem 5. *If* MATRIX-TO-LINE *is approximable within $2 - \delta$ in polynomial time, then we can $\lceil 4/\delta \rceil$-color any 3-colorable graph in polynomial time.*

4 The RH Model

A marker is a gene or an arbitrary DNA sequence for which there is an "easy" laboratory test for its presence in any fragment of DNA. Suppose that we want to construct a physical map of a human chromosome with respect to n markers; that is, we want to find the order and the distance between the markers on the chromosome.

Since there is no direct procedure giving the order between a pair of markers on a fragment of DNA, an RH-experiment is performed. The chromosome is exposed to gamma radiation which shatters it into fragments. Some of the fragments are incorporated into a hamster cell, which is grown to yield a hybrid cell line. Each marker is then tested for presence in cells from this cell line.

The outcome of one experiment is represented by a vector in $f0, 1g^n$ where 1 corresponds to presence; a number of experiments are in the natural way represented by a 0/1-matrix. Such a 0/1-matrix is the laboratory data which is the input to our algorithmic problem. That is, the RH problem is that of, given a 0/1-matrix, finding the order of the markers and the distance between them.

We use the following model of an RH experiment, which is basically the same model as in [2,10], but without the assumption that the markers are uniformly distributed. A genome is modeled by the unit interval $[0, 1]$. A set of n markers is modeled by a function $A : [n] \; ! \; [0, 1]$; that is, each marker is a point in $[0, 1]$. (The former is just a question of scaling. The latter is motivated by the fact that compared to the genome the markers are very short.) An *experiment for* $A : [n] \; ! \; [0, 1]$ is the following probabilistic procedure in which a vector $v \; 2 \; f0, 1g^n$ is produced.

1. Breaks are distributed in the unit interval $[0, 1]$ according to a Poisson process with rate λ. This induces a division of $[0, 1]$ into maximal subintervals without breaks, denoted $I_1, \ldots, I_l$. (This models how radiation breaks the genome.)
2. For each subinterval I_i, let I_i belong to the set S with probability p. (This models how some fragments are incorporated into the hamster genome.) Let $I = [_{I_i 2 S} I_i$.
3. For each $i \; 2 \; [n]$, if $A(i) \; 2 \; I$ let $v(i) = 1$ with probability $1 - \beta$ and otherwise let $v(i) = 1$ with probability α. (This models the negative and positive errors that can occur when a hamster genome is tested for presence of a marker.)

In this way each $A : [n] \; ! \; [0, 1]$ induces a probability distribution P_A on $f0, 1g^n$; that is, for each $x \; 2 \; f0, 1g^n$, $P_A(x)$ is the probability that an experiment for A produces x.

Since everything to the left of the leftmost marker in a genome will be unknown to us, we will assume that this marker is located in $x = 0$.

De nition 4. *A* marker function *is a function* $A : [n] \; ! \; [0, 1]$ *such that* $A(m) = 0$ *for some* $m \; 2 \; [n]$*, and the leftmost marker has lower index than the rightmost marker.*

The last condition is needed to assure that there is a unique marker function corresponding to a certain probability distribution on $f0, 1g^n$.

5 Finding the Genome Map

In this section we give an algorithm for the RH problem using the algorithm for MATRIX-TO-LINE from Section 2.

A marker function A induces a probability distribution P_A on the set $\{0,1\}^n$. We will use the L^1-norm for these distributions as a measure of the distance between marker functions.

Definition 5. *Let A and B be two marker functions. Define*

$$L^1(P_A, P_B) = \sum_{x \in \{0,1\}^n} |P_A(x) - P_B(x)|. \tag{4}$$

This is the same distance as the *variational distance* used for instance for Cavender-Farris trees in [6]. Following [6] it is possible to show that this is a metric for the marker functions.

Let A be the unknown marker function representing the genome we want to study, and let D be the distance matrix defined by $D[i,j] = |A(i) - A(j)|$. Given m experiments for A, we show how to construct a marker function $\hat{A}$ such that $L^1(P_A, P_{\hat{A}}) \leq O(n\sqrt{\log(n)/m})$, using the 2-approximation algorithm from Section 2. In fact, any approximation algorithm with constant performance ratio will give this bound, but the constant hidden in the O notation will be proportional to $1+\tau$, where τ is the performance ratio of the MATRIX-TO-LINE algorithm.

Definition 6. *Two markers i and j are* separated *by an experiment if $v(i) \neq v(j)$, where $v \in \{0,1\}^n$ is the outcome of the experiment.*

In [2], an expression for the probability of two markers being separated was derived for the case when $\alpha = \beta$. For the more general case we get the expression

$$\varphi_{i,j} = 2pq(1 - e^{-D[i,j]})(1 - (\alpha+\beta))^2 + g(\alpha,\beta,p), \tag{5}$$

where $q = 1-p$ and $g(\alpha,\beta,p)$ is some function. By calculating the frequency $\hat{\varphi}_{i,j}$ of separation between the markers i and j we get an estimate of $\varphi_{i,j}$. Solving for $D[i,j]$ in (5), we get an expression for the distances between markers as a function of $\varphi_{i,j}$. Using $\hat{\varphi}_{i,j}$ instead of $\varphi_{i,j}$ in this expression immediately gives us an estimate $\hat{D}[i,j]$ of $D[i,j]$.

Straightforward calculations including the use of Hoeffding's inequality [8] show that, with error probability σ,

$$\|D - \hat{D}\|_1 < \frac{\sqrt{\ln(n^2/\sigma)}}{2pq\lambda\sqrt{2m}(1-(\alpha+\beta))^2} e^{D_{\max}} \tag{6}$$

where $D_{\max} = \max_{i,j} D[i,j]$.

The idea is to use the 2-approximation algorithm for MATRIX-TO-LINE on the estimated distances $\hat{D}[i,j]$ to find a marker function $\hat{A}$ close to the true marker function A.

Lemma 6. *Let $\hat{A}$ be the marker function we obtain if we use the 2-approximation algorithm for* Matrix-To-Line *on the matrix $\hat{D}$.*

$$jj\hat{A}, Djj_1 \quad 3jjD, \hat{D}jj_1 \tag{7}$$

Proof. Since the approximation algorithm has performance ratio 2 we know that $jj\hat{A}, \hat{D}jj_1 \quad 2jjD, \hat{D}jj_1$; and the lemma follows immediately from the triangle inequality $jj\hat{A}, Djj_1 \quad jj\hat{A}, \hat{D}jj_1 + jj\hat{D}, Djj_1$..

Pairs of marker functions satisfying an additional condition have a nice property.

Lemma 7. *Let $f, g : [n] \; ! \; [0,1]$ be two functions such that $f(p) = g(q) = 0$, $f(q) \quad f(r)$, and $g(p) \quad g(r)$, for some $p, q, r \; 2 \; [n]$. If*

$$\max_{i;j \; 2[n]} jjf(i) - f(j)j - jg(i) - g(j)jj \quad \epsilon, \tag{8}$$

then $\sum_{i=1}^{n} jf(i) - g(i)j \quad 2n\epsilon$.

Proof. Omitted

Lemma 7 enables us to get an upper bound on the L^1-norm of the difference in probability distribution for two marker functions, for which the differences in distance between markers are bounded.

Lemma 8. *If A and B are two marker functions such that*

$$\max_{i;j \; 2[n]} jjA(i) - A(j)j - jB(i) - B(j)jj = \epsilon, \tag{9}$$

then $L^1(P_A, P_B) \quad 4\lambda n\epsilon$.

Proof. Let A be a marker function such that $A(p) = 0$. Let B be another marker function for which q is the leftmost marker and r is the rightmost. Let $\bar{B}$ be the function defined by $\bar{B}(i) = B(r) - B(i)$, i.e., the "flip" of B. It is easy to check that Lemma 7 is applicable to either A and B or A and $\bar{B}$. Furthermore, from the definition of an experiment it is clear that $P_A = P_{\bar{A}}$ for every marker function A. Therefore, we will w.l.o.g. assume that there exist $p, q, r \; 2 \; [n]$ such that $A(p) = B(q) = 0$, $A(q) \quad A(r)$, and $B(p) \quad B(r)$.

Assume that one experiment is performed for both A and B simultaneously. If the set of markers is partitioned differently for A and B, then the probability for a certain outcome of the experiment may differ, but otherwise it will not. We call such a break a *dangerous break*. Each marker i induces a subinterval $[A(i), B(i)]$ (or $[B(i), A(i)]$ if $A(i) > B(i)$) of $[0,1]$, within which each break is dangerous. If $\max_{i;j \; 2[n]} jjA(i) - A(j)j - jB(i) - B(j)jj = \epsilon$, the total length of the union of all these intervals will be at most $2n\epsilon$, according to Lemma 7.

Since the breaks are distributed in the interval $[0,1]$ according to a Poisson process with rate λ, the probability that at least one dangerous break occurs is at most $1 - e^{-2 \; n}$. Let F be the event that at least one dangerous break occurs.

Then $\Pr_A[x|\bar{F}] = \Pr_B[x|\bar{F}]$ for all x, since the probability of incorporation of fragments in hamster cells and false answers in the test of occurrences of markers are independent of the size of a fragment.

$$\begin{aligned} L^1(P_A, P_B) &= \sum_{x \in \{0,1\}^n} |P_A(x) - P_B(x)| \\ &= \Pr[F] \sum_{x \in \{0,1\}^n} |\Pr_A[x \mid F] - \Pr_B[x \mid F]| \\ &+ \Pr[\bar{F}] \sum_{x \in \{0,1\}^n} |\Pr_A[x \mid \bar{F}] - \Pr_B[x \mid \bar{F}]| \\ &2\Pr[F] = 2(1 - e^{-2\lambda n}) \quad 4\lambda n\epsilon \end{aligned}$$

where we have used the inequality $1 - x \quad e^{-x}$.

Combining Lemma 6 and Lemma 8, we obtain the following bound on the difference in distribution between $\hat{A}$ and A.

Theorem 6. *With probability* $1 - \sigma$,

$$L^1(P_A, P_{\hat{A}}) \quad \frac{3\sqrt{2n}\sqrt{\ln(n^2/\sigma)}}{pq\sqrt{m}(1 - (\alpha + \beta))^2} e^{D_{\max}} \tag{10}$$

where $D_{\max} = \max_{i,j} D[i,j]$.

The next theorem follows immediately from a lemma in [6], and gives a lower bound on the convergence rate for any algorithm.

Theorem 7. *Let A be a marker function and M an algorithm that given the output of m experiments for A returns an approximation $\hat{A}$ of A. Then with high probability* $L^1(P_A, P_{\hat{A}}) \quad \Omega(1/m)$.

Proof. The theorem follows from Lemma 1 in [6].

6 Acknowledgments

We thank Timothy F. Havel for kindly answering questions concerning the complexity of the Matrix-To-Line problem and pointing us to [11]. We also like to thank Gunnar Andersson for useful remarks.

References

1. Richa Agarwala, Vineet Bafna, Martin Farach, Babu Narayanan, Mike Paterson, and Mikkel Thorup. On the approximability of numerical taxonomy (fitting distances by tree metrics). In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 365–372, 1996.
2. Amir Ben-Dor and Benny Chor. On constructing radiation hybrid maps. In *Proceedings of the First International Conference on Computational Molecular Biology*, pages 17–26, 1997.
3. D. Timothy Bishop and Gillian P. Crockford. Comparison of radiation hybrid mapping and linkage mapping. *Cytogenet Cell Genet*, 59:93–95, 1992.
4. Avrim Blum and David Karger. An $\tilde{O}(n^{3=14})$-coloring algorithm for 3-colorable graphs. *Information Processing Letters*, 61(1):49–53, 1997.
5. Andreas W. M. Dress and Timothy F. Havel. Bound smoothing under chirality constraints. *SIAM J. Disc. Math.*, 4:535–549, 1991.
6. Martin Farach and Sampath Kannan. Efficient algorithms for inverting evolution. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 230–236, 1996.
7. Martin Farach, Sampath Kannan, and Tandy Warnow. A robust model for finding optimal evolutionary trees (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 137–145, 1993.
8. Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.
9. Sanjeev Khanna, Nathan Linial, and Shmuel Safra. On the hardness of approximating the chromatic number. In *ISTCS93*, 1993.
10. Kenneth Lange, Michael Boehnke, David R. Cox, and Kathryn L. Lunetta. Statistical methods for polyploid radiation hybrid mapping. *Genome Research*, 5:136–150, 1995.
11. James B. Saxe. Embeddability of graphs in k-space is strongly NP-hard. In *17th Allerton Conference in Communication, Control, and Computing*, pages 480–489, 1979.
12. Thomas J. Schaefer. The complexity of satisfiability problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 216–226, 1978.
13. Donna Slonim, Leonid Kruglyak, Lincoln Stein, and Eric Lander. Building human genome maps with radiation hybrids. In *Proceedings of the First International Conference on Computational Molecular Biology*, pages 277–286, 1997.

Approximate Coloring of Uniform Hypergraphs (Extended Abstract)

Michael Krivelevich[1] and Benny Sudakov[2]

[1] School of Mathematics, Institute for Advanced Study, Princeton, NJ 08540. mkrivel@math.ias.edu Research supported by an IAS/DIMACS Postdoctoral Fellowship.

[2] Department of Mathematics, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv, Israel. sudakov@math.tau.ac.il

Abstract. We consider an algorithmic problem of coloring r-uniform hypergraphs. The problem of finding the exact value of the chromatic number of a hypergraph is known to be NP-hard, so we discuss approximate solutions to it. Using a simple construction and known results on hardness of graph coloring, we show that for any $r \geq 3$ it is impossible to approximate in polynomial time the chromatic number of r-uniform hypergraphs on n vertices within a factor $n^{1-\varepsilon}$ for any $\varepsilon > 0$, unless $NP = ZPP$. On the positive side, we present an approximation algorithm for coloring r-uniform hypergraphs on n vertices, whose performance ratio is $O(n(\log\log n)^2/(\log n)^2)$. We also describe an algorithm for coloring 3-uniform 2-colorable hypergraphs on n vertices in $\tilde{O}(n^{9/41})$ colors, thus improving previous results of Chen and Frieze and of Kelsen, Mahajan and Ramesh.

1 Introduction

A *hypergraph* H is an ordered pair $H = (V, E)$, where V is a finite nonempty set (the set of *vertices*) and E is a collection of distinct nonempty subsets of V (the set of *edges*). H has *dimension* r if $\forall e \in E(H), |e| \leq r$. If all edges have size exactly r, H is called *r-uniform*. Thus, a 2-uniform hypergraph is just a graph. A set $I \subseteq V(H)$ is called *independent* if U spans no edges of H. The maximal size of an independent set in H is called the *independence number* of H and is denoted by $\alpha(H)$. A *k-coloring* of H is a mapping $f : V(H) \to \{1, \ldots, k\}$ such that no edge of H (besides singletons) has all vertices of the same color. Equivalently, a k-coloring of H is a partition of the vertex set $V(H)$ into k independent sets. The *chromatic number* of H, denoted by $\chi(H)$ is the minimal k, for which H admits a k-coloring.

In this paper we consider an algorithmic problem of coloring r-uniform hypergraphs, for given and fixed value of $r \geq 2$. The special case $r = 2$ (i.e. the case of graphs) is relatively well studied and many results have been obtained in both positive (that is, good approximation algorithms, see e.g. [13], [19], [3], [11], [4], [14], [5]) and negative (that is, by showing the hardness of approximating the chromatic number under some natural complexity assumptions, see

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 477–489, 1998.
© Springer-Verlag Berlin Heidelberg 1998

e.g. [16], [10]) directions. We will briefly survey these developments in the subsequent sections of the paper. However, much less is known about the general case. Lovász [17] showed that it is NP-hard to determine whether a 3-uniform hypergraph is 2-colorable. Additional results on complexity of hypergraph coloring were obtained in [18], [8], [7]. These hardness results give rise to attempts of developing algorithms for *approximate* uniform hypergraph coloring. The first non-trivial case of approximately coloring 2-colorable hypergraphs has recently been considered in papers of Chen and Frieze [9] and of Kelsen, Mahajan and Ramesh ([15], a journal version appeared in [1]). Both papers arrived independently to practically identical results. They presented an algorithm for coloring a 2-colorable r-uniform hypergraph in $O(n^{1-1=r})$ colors, using an idea closely related to the basic idea of Wigderson's coloring algorithm [19]. Another result of the above mentioned two papers is an algorithm for coloring 3-uniform 2-colorable hypergraphs in $\tilde{O}(n^{2=9})$ colors. The latter algorithm exploits the semidefinite programming approach, much in the spirit of the Karger-Motwani-Sudan coloring algorithm [14]. Nothing seems to have been known about general approximate coloring algorithms (that is, when the chromatic number of a hypergraph is not give in advance) and also about the hardness of this approximation problem.

This paper is aimed to (try to) fill a gap between the special case of graphs ($r = 2$) and the case of a general r. We present results in both negative and positive directions. In Section 2 we prove, using corresponding graph results, that unless NP ZPP, for any fixed r 3, it is impossible to approximate the chromatic number of r-uniform hypergraphs on n vertices in polynomial time within a factor of n^{1-} , for any $\epsilon > 0$.

In Section 3 we present an approximation algorithm for coloring r-uniform hypergraphs on n vertices, whose performance guarantee is $O(n(\log\log n)^2/(\log n)^2)$, thus matching the approximation ratio of Wigderson's algorithm [19]. This algorithm is quite similar in a spirit to the algorithm of Wigderson, though technically somewhat more complicated.

In Section 4 we consider the special case of 3-uniform 2-colorable hypergraphs. We develop an algorithm whose performance guarantee for hypergraphs on n vertices is $\tilde{O}(n^{9=41})$, thus slightly improving over the result of Chen and Frieze [9] and of Kelsen, Mahajan and Ramesh [15]. Similarly to the improvement of Blum and Karger [5] of the Karger-Motwani-Sudan algorithm for coloring 3-colorable graphs, an improvement here is achieved by gaining some advantage in the case of dense hypergraphs. Final Section 5 is devoted to concluding remarks.

All logarithms are natural unless written explicitly otherwise.

2 Hardness of Approximation

Several results on hardness of calculating exactly the chromatic number of r-uniform hypergraphs have been known previously. Lovász [17] showed that it is NP-complete to decide whether a given 3-uniform hypergraph H is 2-colorable. Phelps and Rödl proved in [18] that it is NP-complete to check k-colorability

of r-uniform hypergraphs for all $k, r \quad 3$, even when restricted to linear hypergraphs. Brown and Corneil [8] presented a polynomial transformation from k-chromatic graphs to k-chromatic r-uniform hypergraphs. Finally, Brown showed in [7] that, unless $P = NP$, it is impossible to check in polynomial time 2-colorability of r-uniform hypergraphs for any $r \quad 3$.

However, we are not aware about any result showing that it is also hard to *approximate* the chromatic number of r-uniform hypergraphs, where $r \quad 3$. For the graph case ($r = 2$), Feige and Kilian showed in [10], using the result of Håstad [12], that if NP does not have efficient randomized algorithms, then there is no polynomial time algorithm for approximating the chromatic number of an n vertex graph within a factor of n^{1-} , for any fixed $\epsilon > 0$.

In this section we present a construction for reducing the approximate graph coloring problem to approximate coloring of r-uniform hypergraphs, for any $r \quad 3$. Using this construction and the above mentioned result by Feige and Kilian we will be able to deduce hardness results in the hypergraph case.

Let $r \quad 3$ be a fixed uniformity number. Suppose we are given a graph $G = (V, E)$ on $jVj = n \quad r$ vertices with chromatic number $\chi(G) = k$. Define an r-uniform hypergraph $H = (V, F)$ in the following way. The vertex set of H is identical to that of G. For every edge $e \; 2 \; E$ and for every $(r - 2)$-subset $V_0 \quad V \; n \; e$ we include the edge $e \; [\; V_0$ in the edge set F of H. If $F(H)$ contains multiple edges, we leave only one copy of each edge. The obtained hypergraph H is r-uniform on n vertices. Now we claim that $k/(r - 1) \quad \chi(H) \quad k$. Indeed, a k-coloring of G is also a k-coloring of H, implying the upper bound on $\chi(H)$. To prove the lower bound, let $f : V \; ! \; f1, \ldots, k^0g$ be a k^0-coloring of H. Let G_0 be a subgraph of G, whose vertex set is V and whose edge set is composed of all these edges of G that are monochromatic under f. It is easy to see that the degree of every vertex $v \; 2 \; V$ in G_0 is at most $r - 2$ (otherwise the union of the edges of G_0 incident with v would form a monochromatic edge in H). Thus G_0 is $(r - 1)$-colorable. We infer that the edge set $E(G)$ of G can be partitioned into two subsets $E(G) \; n \; E(G_0)$ and $E(G_0)$ such that the first subset forms a k^0-colorable graph, while the second one is $(r - 1)$-colorable. Then G is $k^0(r - 1)$-colorable, as we can label each vertex by a pair whose first coordinate is its color in a k^0-coloration of the first subgraph, and the second coordinate comes from an $(r - 1)$-coloration of the second subgraph. Therefore G and H as defined above have the same number of vertices, and their chromatic numbers have the same order. Applying now the result of Feige and Kilian [10], we get the following theorem.

Theorem 1. *Let $r \quad 3$ be xed. If $NP \; 6 \; ZPP$, it is impossible to approximate the chromatic number of r-uniform hypergraphs on n vertices within a factor of n^{1-} for any xed $\epsilon > 0$ in time polynomial in n.*

3 A General Approximation Algorithm

In this section we present an approximation algorithm for the problem of coloring r-uniform hypergraphs, for a general fixed $r \quad 3$.

Let us start with describing briefly the history and state of the art of the corresponding problem of approximate graph coloring ($r = 2$). The first result on approximate graph coloring belongs to Johnson [13], who in 1974 proposed an algorithm with approximation ratio of order $n/\log n$, where $n = jV(G)j$. The next step was taken by Wigderson [19], whose algorithm achieves approximation ratio $O(n(\log\log n)^2/(\log n)^2)$. The main idea of Wigderson's algorithm was quite simple: if a graph is k-colorable then the neighborhood $N(v)$ of any vertex v 2 $V(G)$ is $(k-1)$-colorable, thus opening a way for recursion. Berger and Rompel [3] further improved Wigderson's result by a factor of $\log n/\log\log n$. They utilized the fact that if G is k-colorable then one can find efficiently a subset S of a largest color class which has size jSj $\log_k n$ and neighborhood $N(S)$ of size at most $n(1 - 1/k)$. Repeatedly finding such S and deleting it and its neighborhood leads to finding an independent set of size $(\log_k n)^2$. Finally, Halldórsson [11] came up with an approximation algorithm that uses at most $\chi(G)n(\log\log n)^2/(\log n)^3$ colors, currently best known result. His contribution is based on Ramsey-type arguments for finding a large independent set from his paper with Boppana [6]. Both papers [3] and [11] proceed by repeatedly finding a large independent set, coloring it by a fresh color and discharging it - quite a common approach in graph coloring algorithms. We will also adopt this strategy. It is worth noting here that one cannot hope for a major breakthrough in this question due to the hardness results mentioned in Section 2.

Unfortunately, most of the ideas of the above discussed papers do not seem to be applicable to the hypergraph case (i.e., when r 3). It is not clear how to define a notion of the neighborhood of a subset in order to apply the Berger-Rompel approach. Also, bounds on the hypergraph Ramsey numbers are too weak to lead to algorithmic applications in the spirit of [6], [11]. However, something from the graph case can still be rescued. Both papers [9] and [15], dealing with the case of 2-colorable hypergraphs, noticed that the main idea behind Wigderson's algorithm is still usable for the hypergraph case. Let us describe now the main instrument of these papers, playing a key role in our arguments as well. For a hypergraph $H = (V, E)$ and a subset of vertices S V, let $N(S) = fv$ 2 $V : S$ [fvg 2 Eg. The following procedure is used in both papers [9] and [15].

Procedure *Reduce*$(H; S)$
Input: A hypergraph $H = (V; E)$ and a vertex subset S V.
Output: A hypergraph $H^0 = (V; E^0)$.
1. Delete from E the set of edges fS [$fvg : v$ 2 $N(S)g$;
2. Add to E an edge S, denote the resulting hypergraph by H^0.

It is easy to see that this procedure has the following properties.

Proposition 1. *Let* $H^0 = Reduce(H, S)$.

1. *if* U *is an independent set in* H^0*, then* U *is independent in* H*;*
2. *If* H *is* k*-colorable and the induced subhypergraph* $H[N(S)]$ *is not* $(k - 1)$*-colorable, then* H^0 *is* k*-colorable.*

Note that the above proposition replaces edges of H by an edge of smaller size. Therefore, in order to apply it we need to widen our initial task and instead of developing an algorithm for coloring r-uniform hypergraphs to present an algorithm for hypergraphs of dimension r. Based on Prop. 1, we can use a recursion on k for coloring k-colorable hypergraphs of dimension r. Indeed, if for some S the subset $N(S)$ is relatively large and $H[N(S)]$ is $(k-1)$-colorable, then applying recursion we can find a relatively large independent subset of $N(S)$. If $H[N(S)]$ is not $(k-1)$-colorable, we can use procedure $Reduce(H,S)$ in order to reduce the total number of edges. Finally, when the hypergraph is relatively sparse, a large independent set can be found based on the following proposition.

Proposition 2. *Let $H = (V,E)$ be a hypergraph of dimension $r \geq 2$ on n vertices without singletons. If every subset $S \subseteq V$ of size $1 \leq |S| \leq r-1$ has a neighborhood $N(S)$ of size $|N(S)| \geq t$, then H contains an independent set U of size $|U| \geq \frac{1}{4}(n/t)^{1/(r-1)}$, which can be found in time polynomial in n.*

Proof. For every $2 \leq i \leq r$, Let E_i be the set of all edges of size i in H. Then $E = \bigcup_{i=2}^{r} E_i$. By the assumptions of the proposition we have

$$|E_i| \leq \binom{n}{i-1} t \leq n^{i-1} t \ .$$

Choose a random subset V_0 of V by taking each $v \in V$ into V_0 independently and with probability $p_0 \leq 1/n$, where the exact value of p_0 will be chosen later. Define random variables X, Y by letting X be the number of vertices in V_0 and letting Y be the number of edges spanned by V_0. Then

$$E[X] = np_0, \qquad E[Y] = \sum_{i=2}^{r} |E_i| p_0^i \leq \sum_{i=2}^{r} n^{i-1} t p_0^i \leq (r-1) n^{r-1} p_0^r t \ .$$

Now we choose p_0 so that $E[X] - E[Y] \geq E[Y]/2$. For example, we can take $p_0 = \frac{1}{2}(n^{r-2}t)^{-1/(r-1)}$. Then by linearity of expectation there exists a set V_0, for which $X - Y \geq \frac{1}{4}(n/t)^{1/(r-1)}$. Fix such a set V_0 and for every edge e spanned by V_0 delete from V_0 an arbitrary vertex of e. We get an independent set U of size $|U| \geq X - Y \geq \frac{1}{4}(n/t)^{1/(r-1)}$.

The above described randomized algorithm can be easily derandomized using standard derandomization techniques (see, e.g., [2], Ch. 15). □

We denote the algorithm described in Proposition 2 by $I(H,t)$. Here are its formal specifications.

Algorithm $I(H,t)$
Input: An integer t and a hypergraph $H = (V,E)$ of dimension r on n vertices, in which every $S \subseteq V$ of size $1 \leq |S| \leq r-1$ satisfies $|N(S)| \geq t$.
Output: An independent set U of H of size $|U| \geq \frac{1}{4}(n/t)^{1/(r-1)}$.

Now we are ready to give a formal description of a recursive algorithm for finding a large independent set in k-colorable hypergraphs of dimension r. Define two functions:

$$g_k(n) = \frac{1}{4} n^{\frac{1}{(r-1)(k-1)+1}}, \qquad f_k(n) = n^{1 - \frac{r-1}{(r-1)(k-1)+1}} \ .$$

One can easily check that g and f satisfy

$$g_{k-1}(f_k(n)) = g_k(n), \quad \frac{1}{4}\left(\frac{n}{f_k(n)}\right)^{\frac{1}{r-1}} = g_k(n) .$$

Algorithm $A(H;k)$
Input: An integer $k \geq 1$ and a hypergraph $H = (V;E)$ of dimension r.
Output: A subset U of V.
1. $n = |V(H)|$;
2. **if** $k = 1$ take U to be an arbitrary subset of V of size $|U| = g_k(n)$ and **return**(U);
3. **if** $k \geq 2$ **then**
4. **while** there exists a subset $S \subseteq V$, $1 \leq |S| \leq r-1$, such that $|N(S)| \geq f_k(n)$
5. Fix one such S and fix $T \subseteq N(S)$, $|T| = f_k(n)$;
6. $U = A(H[T];k-1)$;
7. **if** U is independent in H **return**(U);
8. **else** $H = Reduce(H;S)$;
9. **endwhile**;
10. **return**$(I(H;f_k(n)))$;

We claim that, given a k-colorable hypergraph H as an input, the above presented algorithm finds a large independent set. This follows from the next two propositions, which can be proved by induction on k. We omit the details.

Proposition 3. *Algorithm $A(H,k)$ returns a subset of size $g_k(|V(H)|)$.*

Proposition 4. *If H is k-colorable then $A(H,k)$ outputs an independent set in H.*

Algorithm A is relatively effective for small values of the chromatic number k. Similarly to Wigderson's paper, when k is large we will switch to the following algorithm for finding an independent set. It is worth noting that the idea of partitioning the vertex set of a k-colorable hypergraph H on n into bins of size $k \log_k n$ and performing an exhaustive search for an independent set of size $\log_k n$ in each bin is due to Berger and Rompel [3].

Let

$$h_k(n) = \log_k n = \log n / \log k .$$

Algorithm $B(H;k)$
Input: An integer $k \geq 2$ and a hypergraph $H = (V;E)$.
Output: A subset U of $V(H)$ of size $|U| = h_k(|V(H)|)$.
1. $n = |V(H)|$; $h = h_k(n)$;
2. $l = \lfloor n/hk \rfloor$;
3. Partition $V(H)$ into sets $V_1;\ldots;V_l$ where $|V_1| = \ldots = |V_{l-1}| = hk$ and $hk \leq |V_l| < 2hk$;
4. **for** $i = 1$ **to** l
5. **for** each subset U of V_i of size $|U| = h$
6. **if** U is independent in H **then return**(U);
7. **return** an arbitrary subset of $V(H)$ of size h;

Proposition 5. *Algorithm B is correct, i.e., for a k-colorable hypergraph H on n vertices it outputs an independent set of size $h_k(n)$ in time polynomial in n.*

Proof. If H is k-colorable it contains an independent set I of size $|I| \geq n/k$. Then for some $1 \leq i \leq k$ we have $|I \setminus V_i| \geq |I|/l \geq n/kl \geq h_k(n)$. Checking all subsets of V_i of size $h_k(n)$ will reveal an independent set of size $h_k(n)$. The number of subsets of size $h_k(n)$ to be checked by the algorithm does not exceed $l\binom{2h_k(n)k}{h_k(n)} = (O(1)k)^{h_k(n)} = n^{O(1)}$. □

As we have already mentioned above, an algorithm for finding independent sets can be easily converted to an algorithm for coloring. The idea is very simple – as long as there are some uncolored vertices (we denote their union by W), call an algorithm for finding an independent set in the spanned subhypergraph $H[W]$, color its output by a fresh color and update W. As we have two different algorithms A and B for finding independent sets, we present two coloring algorithms C_1 and C_2, using A and B, respectively, as subroutines. Since the only difference between these two algorithms is in calling A or B, we present them jointly.

Algorithms $C_1(H,k)$ and $C_2(H,k)$
Input: An integer $k \geq 2$ and a hypergraph $H = (V, E)$ of dimension r.
Output: A coloring of H or a message "H is not k-colorable".
1. $i = 1$; $W = V$;
2. **while** $W \neq \emptyset$ **do**
3. for $C_1(H,k)$: $U = A(H[W],k)$; for $C_2(H,k)$: $U = B(H[W],k)$;
4. **if** U is not independent in H **output** ("H is not k-colorable") and **halt**;
5. color U by color i; $i = i + 1$;
6. $W = W \setminus U$;
7. **endwhile**;
8. **return** a coloring of H;

The correctness of both algorithms C_1 and C_2 follows immediately from that of A and B, respectively. As for the performance guarantee, it can be derived from the following easy proposition, proven, for example, implicitly in the paper of Halldórsson [11].

Proposition 6. *An iterative application of an algorithm that guarantees finding an independent set of size $f(n) = O(n^{1-\epsilon})$ in a hypergraph H on n vertices for some fixed $\epsilon > 0$, produces a coloring of H with $O(n/f(n))$ colors.*

Corollary 1. *1. Algorithm $C_1(H,k)$ colors a k-colorable hypergraph H of dimension r on n vertices in at most $2n/g_k(n) = 8n^{1-\frac{1}{(r-1)(k-1)+1}}$ colors;*
2. Algorithm $C_2(H,k)$ colors a k-colorable hypergraph H on n vertices in at most $2n/h_k(n) = 2n \log k / \log n$ colors.

Proof. Follows immediately from Propositions 3, 4, 5 and 6. □

Now, given a k-colorable hypergraph H of dimension r, we can run both algorithms C_1 and C_2 and then choose the best result from their outputs. This is given by Algorithm D below.

Algorithm $D(H,k)$
Input: An integer $k \geq 2$ and a hypergraph $H = (V, E)$ of dimension r.
Output: A coloring of H or a message "H is not k-colorable".
1. Color H by Algorithm $C_1(H,k)$;
1. Color H by Algorithm $C_2(H,k)$;
3. **if** C_1 or C_2 output "H is not k-colorable", **output**("H is not k-colorable");
4. **else return** a coloring which uses fewer colors;

Until now we assumed that the chromatic number of the input hypergraph H is given in advance. Though this is not the case for general approximation algorithms, we can easily overcome this problem, for example, by trying all possible values of k from 1 to $n = |V(H)|$ and choosing a positive output of $D(H,k)$ which uses a minimal number of colors. Denote this algorithm by $E(H)$. In particular,

for $k = \chi(H)$, Algorithm C_1 produces a coloring with at most $8n^{1-\frac{1}{(r-1)(k-1)+1}}$ colors, while Algorithm C_2 gives a coloring with at most $2n \log n / \log k$ colors. Hence, the approximation ratio of Algorithm E is at most

$$\min \quad 8n^{1-\frac{1}{(r-1)(k-1)+1}}/k, \frac{2n \log k}{k \log n} \quad .$$

The first argument of the above min function is an increasing function of k, while the second one is decreasing. For $k = (1/(r-1)) \log n / \log \log n$ both expressions have order $O(n(\log \log n)^2/(\log n)^2)$. Therefore we get the following result.

Theorem 2. *For every xed r 3, coloring of hypergraphs of dimension r on n vertices is $O(n(\log \log n)^2/(\log n)^2))$ approximable.*

4 Coloring 3-Uniform 2-Colorable Hypergraphs

Let us start this section by defining terminology and notation to be used later. Given a 3-uniform hypergraph $H = (V, E)$, for a pair of vertices v, u 2 V we denote $N(u, v) = fw$ 2 $V : (u, v, w)$ 2 Eg. Let also $d(u, v) = jN(u, v)j$. H is *linear* if every pair of edges of H has at most one vertex in common, that is, $d(u, v)$ 1 for every u, v 2 V. Also, the *neighborhood* of v 2 V is defined as $N(v) = fu$ 2 V n $fvg : 9w$ 2 $V, (u, v, w)$ 2 $E(H)g$. The *independence ratio* $ir(H)$ of H is $ir(H) = \alpha(H)/jVj$.

Both papers [9] and [15] noticed that the special case of 3-uniform 2-colorable hypergraphs is different from other cases. The reason for this difference stems from the fact that for this particular case the semidefinite programming approach, pioneered for coloring problems by Karger, Motwani and Sudan [14], can be applied here as well. We refer the reader to [15] for a relevant discussion. In our algorithm we will use the semidefinite programming subroutine of [9], [15] to find a large independent set. Their analysis yields the existence of the following procedure.

Procedure *Semidef*(H)
Input: A 3-uniform 2-colorable hypergraph $H = (V; E)$ on n vertices with m n edges.
Output: An independent set I of size $jIj = \Omega(n^{9=8}=(m^{1=8}(\log n)^{9=8}))$.

Chen and Frieze and also Kelsen, Mahajan and Ramesh use essentially the above procedure to color a 3-uniform 2-colorable hypergraph H on n vertices in $\tilde{O}(n^{2=9})$ colors. We improve their result, again relying on the same procedure. Our main aim is to design an algorithm for finding an independent set I of size $jIj = \Omega(n^{32=41}/(\log n)^{81=82})$. As we have seen before (Prop. 6), such an algorithm can be then used to color H in $O(n^{9=41}(\log n)^{81=82})$ colors, thus giving a (slight) improvement over the result of [9], [15] in the exponent of n ($2/9 = 0.222\ldots$, $9/41 = 0.219\ldots$).

The main idea of the algorithm from [9], [15] is quite simple (given the existence of procedure *Semidef*): if there is a pair of vertices u, v 2 H such that $d(u, v) = \tilde{\Omega}(n^{7=9})$, then either $N(u, v)$ is independent, thus providing an independent set of a desired size, or one can apply procedure *Reduce* (see Section

3) to reduce the number of edges in H. Note that though *Reduce* creates edges of size 2, this does not cause any problem, for example, we can ignore all edges of size 2 that appear in the course of the algorithm execution and then, after having colored the hypergraph, take care of these edges. At this point we can use the fact that the graph formed by edges of size two is also 2-colorable and therefore we need at most two new colors for each color class in the coloring of the 3-uniform hypergraph. When $jE(H)j = \tilde{O}(n^{25=9})$, one can use *Semidef* to find a large independent set. We will achieve an improvement by using a more sophisticated idea in the case of dense hypergraphs. This approach is somewhat similar to that of the paper of Blum and Karger [5], where they improve the bound of Karger, Motwani and Sudan for coloring 3-colorable graphs by using more elaborate techniques for the dense case. Here is a brief outline of our algorithm. First we check whether there exists a pair of vertices $u, v \in V$ such that $d(u, v)$ is large, namely, $d_1(u, v) = \tilde{\Omega}(n^{32=41})$. If such a pair indeed exists then either $N(u, v)$ is independent or we can apply *Reduce*. Now, after all these reductions have been done, we check the number of edges of the resulting hypergraph H_1. If $jE(H_1)j = \tilde{O}(n^{113=41})$, we use *Semidef* to find an independent set of the desired size. It is intuitively clear that the more edges the hypergraph has the easier is the task of recovering its structure. We use this paradigm in the following way. If $jE(H_1)j$ is large, we find a linear subhypergraph H_2 of H_1 with all degrees at least $\tilde{\Omega}(n^{40=41})$. Now, it is rather easy to prove that there exists a vertex $v_0 \in V$ such that at least two third of the neighbors of v_0 in H_2 are colored in a color distinct from that of v in some 2-coloration of H (H_2). This implies that the neighborhood $N_{H_2}(v_0)$ has size $\tilde{\Omega}(n^{40=41})$ and spans a 2-colorable hypergraph H_3 whose vertex set contains an independent set of H of size at least $\frac{2}{3}jV(H_3)j$. Finally, we find an independent set of size $\tilde{\Omega}(jV(H_3)j^{4=5}) = \tilde{\Omega}(n^{32=41})$ in H_3 using a reminiscent of the subgraph exclusion technique applied by Boppana and Halldórsson [6].

Now we give a detailed description of the algorithm. To simplify the presentation we try to avoid the use of specific constants in some of the expressions below, hiding them in the standard O, Ω-notation, the exact values of the corresponding constants can easily be calculated. Suppose we are given a 3-uniform 2-colorable hypergraph $H = (V, E)$ on $jVj = n$ vertices. Recall that our aim is to find an independent set of size $\Omega(n^{32=41}/(\log n)^{81=82})$ in H. Let n be the desired size of the independent set, then

$$n = c\frac{n^{32=41}}{(\log n)^{81=82}}$$

for some specific constant $c > 0$. In the first phase of the algorithm we repeatedly check whether H contains a pair of vertices $u, v \in H$ such that $d(u, v) \quad n$. If such a pair exists, we check whether the subset $N(u, v)$ is independent in H. If it is, we return $N(u, v)$. If not, we apply procedure *Reduce* to replace all edges of size 3 passing through u, v by a single edge (u, v). In the end of this phase we get a hypergraph H_1 on n vertices in which $d(u, v) \quad n$ for every pair $u, v \in V$. By Prop. 1, an independent set in H_1 is also independent in H.

Now, if $jE(H_1)j = O(n^{113=41}/(\log n)^{45=41})$, we call procedure *Semidef* to find an independent set of the desired size. Otherwise, H_1 is a hypergraph with $\Omega(n^{113=41}/(\log n)^{45=41})$ edges in which there are at most n edges through any pair of vertices. Then H_1 contains a linear subhypergraph H_1^0 with the same vertex set and $\Omega(n^{81=41}/(\log n)^{9=82})$ edges. This subhypergraph can be found by a simple greedy procedure: start with $E(H_1^0) = ;$ and as long as there exists an edge $e \in E(H_1) \cap E(H_1^0)$ such that $je \setminus fj \leq 1$ for any edge $f \in E(H_1^0)$, add e to $E(H_1^0)$. If this procedure stops with $jE(H_1^0)j = t$ edges, then, as for every edge in $E(H_1^0)$ there are at most $3n$ other edges intersecting it in two vertices, we get $3tn \geq jE(H_1)j - t$, implying $t = \Omega(n^{81=41}/(\log n)^{9=82})$. Now, repeatedly delete from H_1^0 vertices with degree at most $t/2n$, stop when such vertices do not exist and denote the resulting subhypergraph by H_2. The total number of deleted edges does not exceed $(t/2n)n = t/2$ and therefore $E(H_2) \neq ;$. Also, every vertex of H_2 has degree at least $t/2n = \Omega(n^{40=41}/(\log n)^{9=82})$. Of course, H_2 is a linear hypergraph as well.

Let us fix some 2-coloration $V(H) = C_1 \cup C_2$ of H. Also, for a vertex $v \in V(H_2)$, let $C(v) \in \{C_1, C_2\}$ denote the color class of v. Consider the sum

$$\sum_{v \in V(H_2)} (jN_{H_2}(v) \cap C(v)j - 2jN_{H_2}(v) \setminus C(v)j) =$$

$$\sum_{v \in V(H_2)} jN_{H_2} \cap C(v)j - \sum_{v \in V(H_2)} 2jN_{H_2}(v) \setminus C(v)j .$$

Note that every edge $e \in E(H_2)$ has exactly two vertices of one color and exactly one vertex of the opposite color. Also, since H_2 is linear, for every $v \in V(H_2)$ every vertex $u \in N(v)$ belongs to exactly one edge incident with v. Therefore every edge e contributes 4 to the first sum of the right hand side of the above equality and the same amount to the second sum. This observation shows that the sum above is equal to zero, implying in turn that at least one of the summands is non-negative. We infer that there exists a vertex $v_0 \in V(H_2)$ such that $N_{H_2}(v_0) = \Omega(n^{40=41}/(\log n)^{9=82})$ and $jN_{H_2}(v_0) \cap C(v_0)j \geq (2/3)jN_{H_2}(v_0)j$. Let $H_3 = H[N_{H_2}(v_0)]$, $n_3 = jV(H_3)j = \Omega(n^{40=41}/(\log n)^{9=82})$. Then H_3 is 3-uniform, 2-colorable and satisfies $\alpha(H_3) \geq (2/3)n_3$. Suppose we have this vertex v_0 at hand (we can check all the vertices in polynomial time). Our aim is to find a large independent set in a hypergraph with the above properties.

At this point we apply subhypergraph exclusion techniques, similarly to the algorithm of Boppana and Halldórsson for finding a large independent set in graphs [6]. Our main tool is the following proposition.

Proposition 7. *Let $0 < a < b \leq 1$ be fixed numbers. Let G be a fixed r-uniform hypergraph with independence ratio $ir(G) = a$. Given an r-uniform hypergraph $H = (V, E)$ on n vertices with $ir(H) \geq b$, one can find in time polynomial in n a subset V_1 of V of size $jV_1j \geq (b - a)n$ such that the induced subhypergraph $H_1 = H[V_1]$ is G-free and $ir(H_1) \geq b$.*

Proof. Let $|V(G_0)| = n_0$. We start with $H_0 = H$ and as long as H_0 contains a copy of G we delete its vertex set from $V[H_0]$. The procedure stops when H_0 is G-free.

Denote by W the union of the vertex sets of deleted copies of G and let s be the number of deleted copies. Clearly, $s \leq n/n_0$. Let $V_1 = V \setminus W$. We claim that V_1 is the desired set. In order to prove it, let I be an independent set in H of size $|I| = bn$. As $ir(G) = a$, each deleted copy of G has at most an_0 vertices in common with I. Then $|I \cap W| \geq bn - san_0 \geq (b-a)n$, implying that $|V_1| \leq (b-a)n$. The subhypergraph $H_1 = H[V_1]$ contains the independent set $I \cap W$, therefore

$$ir(H_1) \geq \frac{|I \cap W|}{|V_1|} \geq \frac{bn - asn_0}{n - sn_0} \geq b$$

(recall that $a < b$). ∎

We will apply the above proposition several times. The first application is the following. Let G_1 be a hypergraph with vertex set $V(G_1) = \{v_1, \ldots, v_5\}$ and edge set composed of the following four edges: $\{(v_1, v_2, v_i) : 3 \leq i \leq 5\}; (v_3, v_4, v_5)\}$. It is easy to see that $\alpha(G_1) = 3$, implying $ir(G_1) = 3/5$. Now substituting $H = H_3$, $G = G_1$, $a = ir(G_1) = 3/5$ and $b = ir(H_3) \geq 2/3$ in Prop. 7, we find a hypergraph H_4, which is 3-uniform, 2-colorable, has $|V(H_4)| = n_4 \geq n_3/15 = \Omega(n^{40=41}/(\log n)^{9=82})$ vertices. Moreover, H_4 is G_1-free and $ir(H_4) \geq 2/3$.

Let now G_2 be defined as follows. The vertex set of G_2 consists of seven vertices $\{v_1, \ldots, v_7\}$. The edge set of G_2 has four edges: (v_1, v_2, v_5), (v_1, v_3, v_6), (v_1, v_4, v_7) and (v_5, v_6, v_7). Note that vertices v_2, v_3, v_4 have the minimal degree in G_2, therefore we will call them the *low degree vertices* of G_2. We would like to get from H_4 a G_2-free hypergraph H_5 with $\Theta(n_4)$ vertices. Though $ir(G_2) = 5/7 > 2/3$ and therefore Prop. 7 cannot be applied directly, we can observe that if I is an independent set in G_2 of size five, then I should contain all low degree vertices of G_2. Thus, when deleting copies of G_2 from H_4, we either find a large independent set in the union of the low degree vertices of the deleted copies of G_2 or end up with a G_2-free subhypergraph with many vertices. Here is a formal argument, justifying the above statement. As long as H_4 contains a copy of G_2 we delete its vertex set from $V(H_4)$. Denote the resulting subhypergraph by H_5. Let also $G^1, \ldots, G^s$ be the deleted copies of G_2, clearly, $s \leq n_4/7$. Denote $W = \bigcup_{i=1}^{s} V(G^i)$, then $|W| = 7s$. Let I be an independent set in H_4 of size $|I| \geq \frac{2}{3}n_4$. Then $|I \setminus W| \geq |I| + |W| - n_4 \geq 7s - n_4/3$. For $0 \leq j \leq 5$, define $s_j = |\{1 \leq i \leq s : |(V(G^i) \setminus I)| = j\}|$. Then

$$\sum_{j=0}^{5} s_j = s\,, \tag{1}$$

$$\sum_{j=1}^{5} j s_j = |I \setminus W| \geq 7s - \frac{n_4}{3}\,. \tag{2}$$

Let now $L(G^i)$ be the low degree vertices of G^i, $1 \leq i \leq s$, let also $L = \bigcup_{i=1}^{s} L(G^i)$. It is easy to check that if I intersects $V(G^i)$ in five vertices then

$jI \setminus L(G^i)j = 3$, and also if I has four vertices in common with $V(G^i)$ then $jI \setminus L(G^i)j$ 1. Therefore $jI \setminus Lj$ $3s_5 + s_4$. Multiplying inequality (2) by 2 and subtracting (1) multiplied by 7, we get $3s_5 + s_4$ $7s - 2n_4/3$. This implies

$$ir(H_4[L]) \quad \frac{jI \setminus Lj}{jLj} \quad \frac{7s - \frac{2n_4}{3}}{3s} .$$

¿From here we can see that either L contains a large independent set or H_5 has $\Omega(n_4)$ vertices. Indeed, if, say, s $(29/210)n_4$, then jLj $(29/70)n_4$ and $ir(H_4[L])$ $21/29$. Then Prop. 7 can be applied to $H_4[L]$ with G being a single edge ($ir(G) = 2/3$), this way we get an independent set of size at least $(21/29 - 2/3)(29/70)n_4 = n_4/42$. Otherwise (if s $(29/210)n_4$) we have $jV(H_4) \cap Wj$ $n_4/30$, implying that H_5 has $\Omega(n^{40=41}/(\log n)^{9=82})$ vertices, is 3-uniform, 2-colorable and does not contain a copy of G_1 or G_2.

Finally, let n_5 be the number of vertices of H_5. Suppose first that there exists a vertex $v \in V(H_5)$ with degree $d_{H_5}(v)$ $n_5^{8=5}/(\log n_5)^{9=5}$. Then there are two possibilities. If there exists a vertex $u \in V(H_5) \cap fvg$ such that $d_{H_5}(u, v)$ $n_5^{4=5}/(\log n_5)^{9=10})$, then as H_5 is G_1-free, the subset $N_{H_5}(u, v)$ is independent in H. Otherwise, one can find $l = \Omega(n_5^{4=5}/(\log n_5)^{9=10})$ edges $e_1, \ldots, e_l$ of H_5 such that every pair of edges intersects only at v. For 1 i l choose $u_i \in e_i \cap fvg$. Then, as H_5 is G_2-free, the subset $fu_i : 1$ i lg is independent in H. If $d_{H_5}(v) = O(n_5^{8=5}/(\log n_5)^{9=5})$, we can apply procedure *Semidef* to find an independent set of size $\Omega(n_5^{4=5}/(\log n_5)^{9=10})$ in H.

Summarizing the above discussion we deduce the following corollary.

Proposition 8. *Given a 3-uniform 2-colorable hypergraph H on n vertices, one can find in time polynomial in n an independent set in H of size $\Omega(\frac{n^{32=41}}{(\log n)^{81=82}})$.*

Applying Prop. 6 we get the main result of this section.

Theorem 3. *There is a polynomial time algorithm for coloring 3-uniform 2-colorable hypergraphs on n vertices in $O(n^{9=41}(\log n)^{81=82})$ colors.*

5 Concluding Remarks

Despite some progress achieved in this paper, many problems remain open and seem to be quite interesting. One of them is to develop a general approximation algorithm, aiming to match the approximation ratio $O(n(\log\log n)^2/(\log n)^3)$ of the best known graph coloring algorithm due to Halldórsson [11]. Another interesting problem is to come up with a more involved algorithm for the case of r-uniform 2-colorable hypergraphs for r 4. Also, new ideas for the dense case of 3-uniform 2-colorable hypergraphs may lead to a further improvement in this case.

References

1. N. Alon, P. Kelsen, S. Mahajan and H. Ramesh, *Coloring 2-colorable hypergraphs with a sublinear number of colors*, Nordic J. Comput. 3 (1996), 425–439.
2. N. Alon and J. H. Spencer, **The probabilistic method**, Wiley, New York, 1992.
3. B. Berger and J. Rompel, *A better performance guarantee for approximate graph coloring*, Algorithmica 5 (1990), 459–466.
4. A. Blum, *New approximation algorithms for graph coloring*, J. ACM 31 (1994), 470–516.
5. A. Blum and D. Karger, *An $\tilde{O}(n^{3=14})$-coloring algorithm for 3-colorable graphs*, Inform. Process. Lett. 61 (1997), 49–53.
6. R. Boppana and M. M. Halldórsson, *Approximating maximum independent sets by excluding subgraphs*, Bit 32 (1992), 180–196.
7. J. Brown, *The complexity of generalized graph colorings*, Discrete Applied Math. 69 (1996), 257–270.
8. J. Brown and D. Corneil, *Graph properties and hypergraph colourings*, Discrete Math. 98 (1991), 81–93.
9. H. Chen and A. Frieze, *Coloring bipartite hypergraphs*, in: Proc. 5^{th} Intern. IPCO Conference, Lecture Notes in Comp. Sci. 1084 (1996), 345–358.
10. U. Feige and J. Kilian, *Zero knowledge and the chromatic number*, in: Proc. 11^{th} Annual IEEE Conf. on Computational Complexity, 1996.
11. M. M. Halldórsson, *A still better performance guarantee for approximate graph coloring*, Inform. Process. Lett. 45 (1993), 19–23.
12. J. Håstad, *Clique is hard to approximate within n^{1-}* , in: Proc. 37^{th} IEEE FOCS, IEEE (1996), 627–636.
13. D. S. Johnson, *Worst case behaviour of graph coloring algorithms*, in: Proc. 5th S.E. Conf. on Combinatorics, Graph Theory and Computing, Congr. Numer. 10 (1974), 512–527.
14. D. Karger, R. Motwani and M. Sudan, *Approximate graph coloring by semide nite programming*, in: Proc. 26^{th} ACM FOCS, ACM Press (1994), 2–13.
15. P. Kelsen, S. Mahajan and H. Ramesh, *Approximate hypergraph coloring*, in: Proc. 5^{th} Scandinavian Workshop on Algorithm Theory, Lecture Notes in Comp. Sci. 1097 (1996), 41–52.
16. S. Khanna, N. Linial and M. Safra, *On the hardness of approximating the chromatic number*, in: Proc. 2^{nd} Israeli Symposium on Theor. Comp. Sci., IEEE (1992), 250–260.
17. L. Lovász, *Coverings and colorings of hypergraphs*, in: Proc. 4^{th} S.E. Conf. on Combinatorics, Graph Theory and Computing, 1973, Utilitas Math., 3–12.
18. K. Phelps and V. Rödl, *On the algorithmic complexity of coloring simple hypergraphs and Steiner triple systems*, Combinatorica 4 (1984), 79–88.
19. A. Wigderson, *Improving the performance guarantee for approximate graph coloring*, J. ACM 30 (1983), 729–735.

Techniques for Scheduling with Rejection

Daniel W. Engels[?], David R. Karger[??], Stavros G. Kolliopoulos[???], Sudipta Sengupta[??], R. N. Uma[y], and Joel Wein[z]

Abstract. We consider the general problem of scheduling a set of jobs where we may choose not to schedule certain jobs, and thereby incur a penalty for each rejected job. More specifically, we focus on choosing a set of jobs to reject and constructing a schedule for the remaining jobs so as to optimize the sum of the weighted completion times of the jobs scheduled plus the sum of the penalties of the jobs rejected.

We give several techniques for designing scheduling algorithms under this criterion. Many of these techniques show how to reduce a problem with rejection to a (potentially more complex) scheduling problem without rejection. Some of the reductions are based on general properties of certain kinds of linear-programming relaxations of optimization problems, and therefore are applicable to problems outside of scheduling; we demonstrate this by giving an approximation algorithm for a variant of the facility-location problem.

In the last section of the paper we consider a different notion of rejection in the context of scheduling: scheduling jobs with due dates so as to maximize the number of jobs that complete by their due dates, or equivalently to minimize the number of jobs that do not complete by their due date and that thus can be considered "rejected." We investigate the approximability of a simple version of this problem, giving approximation algorithms and characterizing integrality gaps of a class of linear-programming relaxations.

[?] dragon@glenfiddich.lcs.mit.edu. Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Research supported by NSF Contract MIP-9612632.

[??] fkarger, sudiptag@theory.lcs.mit.edu. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Research supported in part by ARPA contract N00014-95-1-1246 and NSF contract CCR-9624239, as well as grants from the Alfred P. Sloane and David and Lucille Packard foundations.

[???] stavros@cs.dartmouth.edu. Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510. Research partially supported by NSF Award CCR-9308701 and NSF Career Award CCR-9624828.

[y] ruma@tiger.poly.edu. Department of Computer Science, Polytechnic University, Brooklyn, NY, 11201. Research partially supported by NSF Grant CCR-9626831.

[z] wein@mem.poly.edu. Department of Computer Science, Polytechnic University, Brooklyn, NY, 11201. Research partially supported by NSF Grant CCR-9626831 and a grant from the New York State Science and Technology Foundation, through its Center for Advanced Technology in Telecommunications.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 490–501, 1998.
c Springer-Verlag Berlin Heidelberg 1998

1 Introduction

Most of traditional scheduling theory begins with a set of n jobs to be scheduled in a particular machine environment so as to optimize a particular optimality criterion. At times, however, a higher-level decision has to be made: given a set of tasks, and limited available capacity, choose only a subset of these tasks to be scheduled, while perhaps incurring some penalty for the jobs that are not scheduled. Knapsack is a "pure" instance of the problem, where we care only about what subset to "accept." In scheduling, however, the set of accepted jobs must also be scheduled at some cost.

In this paper we contribute to the understanding of this problem by studying several scheduling models in which the scheduler can choose to *reject* (not schedule) certain jobs at a penalty. We give several techniques for designing exact and approximation algorithms in this paradigm.

1.1 Problem Definition. For much of this paper we study variants of the following basic scheduling problem. We are given n jobs $j = 1, \ldots, n$, each with a nonnegative processing time p_j, a weight w_j, and a *rejection penalty* e_j. For each job we must decide either to schedule that job (on a machine that can process at most one job at a time) or to reject it. If we schedule job j, we denote its completion time by C_j. If we reject job j, we pay its rejection penalty e_j. Our goal is to choose a subset S of the n jobs to schedule on the machine so as to minimize the sum of the weighted completion times of the scheduled jobs and the penalties of the rejected jobs. We denote the set of rejected jobs by $\overline{S}$, and thus our overall optimality criterion may be denoted as $\sum_{j \in S} w_j C_j + \sum_{j \in \overline{S}} e_j$.

At times we consider scheduling models in which the processing of the jobs is constrained by either *release date* constraints (job j cannot begin processing before a specified release date r_j) or *precedence* constraints ($j \prec k$ is a precedence constraint stating that job k can begin its processing no earlier than job j completes its processing). Precedence constraints impose a partial ordering on the jobs. We also consider models with more than one machine available for processing and other related optimality criteria; we leave their details to be introduced as appropriate in the body of the paper.

Our interest in these models arises primarily from two considerations. First, it is becoming widely understood that in actual manufacturing settings a scheduling algorithm is only one element of a larger decision analysis tool that takes into account a variety of factors such as inventory, potential machine disruptions, etc. [13]. Furthermore, some practitioners [12] have identified as a critical element of this larger decision picture the "pre-scheduling negotiation" in which one considers the capacity of your production environment and then agrees to schedule certain jobs at the requested quality of service (e.g. job turnaround time) and other jobs at a reduced quality of service, while rejecting other jobs altogether.

Typically, the way that current systems handle the integration of objective functions is to have a separate component for each problem element and to integrate these components in an ad-hoc heuristic fashion. Therefore, models that integrate more than one element of this decision process while remaining

amenable to solution by algorithms with performance guarantees have the potential to be very useful elements in this decision process.

Secondly, our work is directly inspired by that of Bartal, Leonardi, Marchetti-Spaccamela, Sgall and Stougie [1] who studied a multiprocessor scheduling problem with the objective of trading off between schedule *makespan* (length) and job rejection cost. Makespan and average weighted completion time are perhaps the two most basic and well-studied of all scheduling optimality criteria; therefore, it is of interest to understand the impact of the "rejection option" on the $\sum w_j C_j$ objective as well.

Since we will be dealing with a number of scheduling models, we will use the scheduling notation introduced by Graham, Lawler, Lenstra and Rinnooy Kan [6] to denote the model being considered.

1.2 Relevant Previous Work. There has been much work on scheduling without rejection to minimize $\sum w_j C_j$. For the simplest variant of the problem, $1||\sum w_j C_j$ (scheduling jobs with no side constraints on one machine to minimize $\sum w_j C_j$) Smith [19] proved that an optimal schedule could be constructed by ordering the jobs by non-decreasing p_j/w_j ratios. More complex variants are typically *NP*-hard, and recently there has been a great deal of work on the development of approximation algorithms for them (e.g. [14,8,5,2,17,16]). A *ρ-approximation algorithm* is a polynomial-time algorithm that always finds a solution of objective function value within a factor of ρ of optimal (ρ is also referred to as the *performance guarantee*).

To the best of our knowledge the only previous work on scheduling models that include rejection in our manner is that of Bartal et. al., who seem to have formally introduced the notion. They considered the problem of scheduling with rejection in a multiprocessor setting with the aim of minimizing the makespan of the scheduled jobs and the sum of the penalties of the rejected jobs [1]. They give a $(1+\phi)(\quad 2.618)$-competitive algorithm for the on-line version, where ϕ is the golden ratio.

1.3 Our Results. Our work differs from that of Bartal et. al. in that we focus in $\sum w_j C_j$ rather than the makespan. More importantly, we focus less on a particular problem and more on giving general techniques that are applicable to a number of problems. We address only the offline setting (the problem is fully specified in advance).

We begin in Section 2 by studying the simplest version of the problem, $1||\sum_S w_j C_j + \sum_{\overline{S}} e_j$ and give a simple and intuitive greedy algorithm that solves certain special cases exactly. We show that the general problem $1||\sum_S w_j C_j + \sum_{\overline{S}} e_j$ is weakly *NP*-Complete, and give a pseudopolynomial-time algorithm based on dynamic programming that solves it exactly. Our dynamic program can be interpreted as capturing the option of assigning a job to a "rejection machine". We conclude Section 2 by giving a direct reduction (using the idea of a "rejection machine") of certain scheduling problems with rejection to more complex problems without.

In Sections 3 and 4 we give another example of how techniques that are used for scheduling problems without rejection can be adapted to handle problem variants with rejection. In contrast to a direct problem-to-problem reduction, however, we examine the structure of certain linear-programming relaxations of such scheduling problems, show how they can be augmented to include a rejection variable to which a job can be assigned, and then give a thresholding technique that allows us to obtain an approximation algorithm for a problem with rejection. This technique is in fact quite general and is applicable to many sorts of optimization problems; we demonstrate its generality by giving an approximation algorithm for a variant of the facility location problem with rejection.

Finally, in Section 5, we move to a different and more traditional notion of rejection in scheduling, which is nonetheless not well understood. In this setting each job has a due date, and a job is essentially "rejected" if it is not scheduled by its due date. Specifically, we consider the problem $1|r_j|\sum U_j$: scheduling jobs with release dates on one machine so as to minimize the number of jobs that miss their deadline or, equivalently, maximize the number of jobs that meet their deadline. This is similar to the problem of scheduling jobs with rejection because the jobs that do not meet their deadline can be considered rejected. Nothing is known about approximating these strongly NP-hard scheduling problems. We give integrality gaps for both maximization and minimization versions of the problem and also give an approximation algorithm for the maximization problem.

2 Scheduling with Rejection

In this section, we focus on the problem of scheduling jobs on a single machine to minimize weighted completion time. If rejection is not considered, the problem is solvable in polynomial time using Smith's rule: schedule the jobs in increasing order of p_j/w_j. We show that adding the option of rejection makes the problem weakly NP-complete. Such problems cannot admit polynomial-time solutions unless $P = NP$. But they can admit *pseudo-polynomial* time algorithms that run fast when the input numbers are small. We give such an algorithm, based on dynamic programming, for our scheduling problem.

Intuitively, the dynamic program can be seen as solving a scheduling problem on two machines, one a "rejection machine" that takes care of all of the rejected jobs and has an unusual cost function for doing so. It is therefore a variant of Rothkopf [15] and Lawler and Moore's [10] pseudo-polynomial algorithms for scheduling to minimize weighted completion time on a fixed number of parallel machines. Indeed, our algorithm also generalizes to schedule with rejection on any fixed number of parallel machines. We use the "rejection machine" intuition again to devise a 3/2-approximation algorithm for the problem of scheduling with rejection on an arbitrary number of machines.

Although we use the two-machine intuition, the rejection cost that the second machine pays is in some sense a simpler function than the weighted completion

time. So for certain special cases, we are able to give simple and efficient polynomial time algorithms based on greedy methods.

2.1 Complexity of Scheduling with Rejection. For any fixed number of processors $m > 1$, $Pmjj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$ is trivially seen to be *NP*-complete by restricting the problem to $Pmjj \sum w_j C_j$, a known weakly *NP*-complete problem [4]. However, we can also prove that adding rejection makes the *single* machine problem *NP*-complete.

Theorem 1. *$1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$ is weakly NP-complete.*

Proof. We reduce the Partition Problem [4] to $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$ in a straightforward manner. Each of the n elements a_i in the Partition Problem correspond to a job J_i in $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$ with weight and processing time equal to a_i and rejection cost equal to $ba_i + \frac{1}{2}a_i^2$, where $b = \frac{1}{2}\sum_{i=1}^{n} a_i$.

2.2 Dynamic Programming. We give an $O(n \sum_{j=1}^{n} w_j)$ time algorithm using dynamic programming to solve $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$. In the full version of the paper, we will show how to modify it to obtain an FPAS for $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$.

To solve our problem, we set up a dynamic program for a harder problem: namely, to find the schedule that minimizes the objective function when the total weight of the scheduled jobs is given. We number the jobs in ascending order of p_j/w_j. Let $\phi_{w;j}$ denote the optimal value of the objective function when the jobs in consideration are $j, j+1, \ldots, n$, and the total weight of the scheduled jobs is w. Note that $\phi_{w;n} = 0$ for any $w \neq w_n$ and $\phi_{w_n;n} = w_n p_n$, forming the boundary conditions for the dynamic program.

Now, consider any optimal schedule for the jobs $j, j+1, \ldots, n$ in which the total weight of the scheduled jobs is w. In any such schedule, there are two possible cases — either job j is rejected or job j is scheduled. So we have $\phi_{w;j} = \min(\phi_{w;j+1} + e_j, \phi_{w-w_j;j+1} + wp_j)$.

Now, observe that the weight of the scheduled jobs can be at most $\sum_{j=1}^{n} w_j$, and the answer to our original problem is $\min\{\phi_{w;1} : 0 \quad w \quad \sum_{j=1}^{n} w_j\}$. Thus, we need to compute exactly $n \sum_{j=1}^{n} w_j$ values $\phi_{w;j}$. We remark that a similar dynamic program solves $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$ when the processing times are equal.

Theorem 2. *Dynamic programming yields an $O(n \sum w_j)$-time (or an $O(n \sum p_j)$-time) algorithm for exactly solving $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$.*

2.3 Special cases. We consider two special cases for which simple greedy algorithms exist to solve $1jj(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$. We give an algorithm for the case when all weights are equal. An algorithm for the case when all processing times are equal is analogous and not presented. Our greedy algorithm, which we call `SCHREJ`, is as follows. We start with all jobs scheduled, i.e., the scheduled set $S = \{1, 2, \ldots, n\}$. (Note that we can optimally schedule the jobs in any subset S using Smith's ordering.) We then reject jobs greedily until we arrive at an

optimal schedule. Note that when we reject a previously scheduled job j, there is a change (positive or negative) in the objective function. We determine a job k that causes the maximum decrease in the the objective function. If there is no such job, then (as we will argue below) we have reached an optimal solution. Otherwise, we remove job k from S (i.e. reject it), and iterate on the remaining jobs in S.

As this algorithms runs, we maintain the set S of currently scheduled jobs. Let $\Delta_j(S)$ denote the amount by which the cost of our schedule changes if we (pay to) reject job $j \in S$. Let $k \in S$ be such that $\Delta_k(S)$ is minimum.

Lemma 1. *During any iteration of* SCHREJ*, for any $j \in S$, we have*

$$\Delta_j(S) = -[w_j \sum_{i \; j;i \in S} p_i \; + \; p_j \sum_{i>j;i \in S} w_i] + e_j$$

The following lemma is an immediate consequence of Lemma 1.

Lemma 2. *The value of $\Delta_j(S)$ increases across every iteration of* SCHREJ*, so long as job j remains in S.*

The following lemma argues that the algorithm can terminate when $\Delta_k(S) \quad 0$.

Lemma 3. *For a set S of jobs, if $\Delta_j(S)$ is non-negative for each $j \in S$, then there is an optimal schedule in which all the jobs in S are scheduled.*

Proof. (sketch) Consider any non-empty set of jobs $R \quad S$. Start with the schedule in which all jobs in S are scheduled, and start rejecting the jobs in R one by one. Observe that the objective function value does not decrease during every such rejection.

For the above lemmas, we have not used the fact that the weights w_j are equal. But this fact is used by the next lemma, which proves that we are justified in moving job k from S to $\overline{S}$ when $\Delta_k(S) < 0$.

Lemma 4. *For a set S of jobs with equal weights, if $\Delta_k(S)$ (the minimum of the $\Delta_j(s)$ for $j \in S$, as computed in* SCHREJ*) is negative, then there is an optimal schedule for the set of jobs in S in which job k is rejected.*

Proof. (sketch) Consider an optimal schedule Γ in which job k is scheduled and the set of rejected jobs is R. Clearly, R is non-empty, otherwise, since $\Delta_k < 0$, we can get a better schedule by rejecting job k. We show that we can improve the schedule Γ by rejecting job k and instead scheduling one of the jobs in R (the one immediately preceding or following job k according to Smith's rule). We compare the objective function value for the two schedules by starting from a schedule in which all jobs are scheduled, and then rejecting the set R of jobs in a particular order. This order is slightly different for the two cases considered.

Theorem 3. *Algorithm* SCHREJ *outputs an optimal schedule for $1|rej|(\sum_S w_j C_j + \sum_{\bar{S}} e_j)$ with equal weights in $O(n^2)$ time.*

Proof. (sketch) By induction. According to the previous lemmas, $SCHREJ$ only rejects job it is safe to reject, and terminates when all remaining jobs must be in the optimal schedule.

2.4 Worst-Case Performance Guarantees. We now turn to several NP-hard variants of the problem and give small-constant-factor approximation algorithms; we do this by reducing the problem with rejection to a scheduling problem without rejection.

First we consider the single machine version of the problem $1||(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$. An instance I can be reduced in an approximation-preserving manner to an instance I' of $R||\sum w_j C_j$ with $n+1$ unrelated machines. Let the machines be indexed $1, 2, \ldots, (n+1)$. Machine $(n+1)$ is the *original* machine with $p_{n+1;j} = p_j$ for job j. Job j has $p_{jj} = \frac{e_j}{w_j}$ on machine j and $p_{ij} = 1$ on machine i where $i = 1, \ldots, n$ and $i \neq j$. We omit further details. A similar reduction can be obtained from $1|r_j|(\sum_{S} w_j C_j + \sum_{\bar{S}} e_j)$ to $R|r_{ij}|\sum w_j C_j$ using the observation that without loss of generality $e_j > w_j r_j$ for all j. Inspection of the argument above reveals that it can be extended to parallel identical machines and parallel unrelated machines. Using the approximation algorithms of Schulz and Skutella [17] for $R||\sum w_j C_j$ and $R|r_{ij}|\sum w_j C_j$ we obtain the following theorem.

Theorem 4. *There exists a $\frac{3}{2}$-approximation algorithm for $R||\sum_{S} w_j C_j + \sum_{\bar{S}} e_j$ and a 2-approximation algorithm for $R|r_j|\sum_{S} w_j C_j + \sum_{\bar{S}} e_j$, both in expectation.*

3 A General Problem P with Rejection

In this section we give a rather general method of converting certain kinds of approximation algorithms for optimization problems without rejection to approximation algorithms for problem variants with rejection. In contrast to the last result in the previous section, however, which was a direct reduction to a scheduling problem in which the solution to that scheduling problem was used as a black box, in this case we exploit the common structure of different linear-programming relaxations used in approximation for scheduling and other optimization problems.

We begin by stating a general "assignment" optimization problem and gaining some understanding of the structure of solutions to two relaxations of this problem. Then we use this structure in a general algorithm for the optimization problem augmented to include the possibility of rejection.

Let us consider the following general problem P. We have two sets of objects O_1 and O_2. We want to assign each object in O_1 to an object in O_2 such that a given set of constraints are satisfied by the assignment and some objective function of the assignment is minimized. We will formulate this problem as an integer program which uses the 0-1 decision variables x_{ij}, where x_{ij} is 1 if object $i \in O_1$ is assigned to object $j \in O_2$ and 0 otherwise. So P is formulated by the following integer program (IP). Let $\mathbf{c}, \mathbf{d} \geq \mathbf{0}$.

$$\begin{array}{lrcll}
\text{minimize} & \mathbf{c^T x} + \mathbf{d^T y} & & & & (3.1)\\
\text{subject to} & \sum_{j} x_{ij} & = & 1 & \forall i \in O_1 & (3.2)\\
& \sum_{i;k \in S(j)} x_{ik} & & \kappa_j & \forall j \in O_2 & (3.3)\\
& \mathbf{A(x\ y)} & & \mathbf{0} & & (3.4)\\
& x_{ij} & \in & \{0,1\} & \forall i \in O_1, \forall j \in O_2 & (3.5)\\
& x_{ij} & = & 0 & (i,j) \in M,\ O_1,\ O_2 & (3.6)
\end{array}$$

Here, $\mathbf{x}\ \mathbf{y}$ denotes the concatenation of vectors $\mathbf{x}, \mathbf{y}$. $\mathbf{A}(\mathbf{x}\ \mathbf{y})\ \ \mathbf{0}$ is a set of additional constraints that have to be satisfied by any feasible assignment. We call the constraints (3.3) *capacity constraints*.

We can replace the integrality constraint on x_{ij} by the constraint $0\ \ x_{ij}\ \ 1$. This essentially means that an object i 2 O_1 can be fractionally assigned to objects in O_2. We will let $(LP1)$ denote the resulting relaxation to (IP).

We will now consider the following additional relaxation to $(LP1)$. Let β be a positive constant that is at most 1. We will further relax the constraint $\sum_j x_{ij} = 1$ to the constraint $\beta \quad \sum_j x_{ij}\ \ 1$. This says that at least a β fraction of object i 2 O_1 must be assigned to some object(s) in O_2. We will call this linear program $(LP2)$.

Let us denote their corresponding optimal solutions by $IP\ , LP1$ and $LP2$ respectively. Since $(LP1)$ is a relaxation of (IP) and $(LP2)$ is a relaxation of $(LP1)$, we have the inequality $LP2\ \ \ LP1\ \ \ IP\ .$

Let A be an algorithm that accepts a feasible solution to $(LP1)$ as input and returns an output with value at most ρ times the input's objective function value (it thus acts as a ρ-approximation algorithm for $(IP1)$). We will show that if problem P satisfies certain Conditions 1 and 2 (to be stated later), then we can use A in an approximation algorithm (with somewhat worse performance) for a version of P that allows rejection. We now state Condition 1.

Condition 1 *There exists a polynomial-time algorithm to convert a feasible solution* $\mathbf{q}$ *to* $(LP2)$, *of value* $v(\mathbf{q})$, *to a feasible solution to* $(LP1)$ *of value within at most* $f(\beta)\ \ v(\mathbf{q})$.

`Case a:` $f(\beta) = \frac{1}{-}$, *if* P *does not have capacity constraints.*

`Case b:` $f(\beta) = \frac{1}{-2}$, *if* P *has capacity constraints.*

We now modify the original problem P to include rejection, and use our structural insights to give an approximation algorithm. We modify P so that each object i 2 O_1 can either be assigned to an object j 2 O_2 or can be rejected for a certain penalty. Let us call this modified problem P_R. The problem P_R can be formulated similar to IP. But now we will have an additional 0-1 decision variable z_i which is 1 if object i 2 O_1 is rejected and 0 otherwise. The objective function to be minimized is now given by $\sum \mathbf{c^T x} + \mathbf{d^T y} + \mathbf{g^T z}$ and the constraint $\sum_j x_{ij} = 1$ in the (IP) is replaced by $\sum_j x_{ij} + z_i = 1$. Let the corresponding linear relaxation be $(LP1_R)$. We introduce a second "decomposability" condition that is natural for problems involving rejection. Let $P(S)$ denote the restriction of the assignment problem P above, such that x_{ij} is defined only for i 2 $S\ \ \ O_1$. Intuitively, we want to capture the notion that the set $O_1 - S$ of objects to be rejected is determined by the objective but not the constraints of P_R.

Condition 2 *Let* $(\mathbf{x}, \mathbf{y})$ *be a feasible solution for* $P(O_1)$. *If* x_{ij} *with* i 2 $O_1 - S$, *is removed from* $\mathbf{x}$ *to obtain* $\mathbf{x}^0$, *the pair of vectors* $(\mathbf{x}^0, \mathbf{y})$ *is feasible for* $P(S)$.

Consider the following algorithm B to get a solution for P_R. First solve $(LP1_R)$. If $\sum_j x_{ij} < \beta$, then include i 2 O_1 in the set $\bar{S}$ of rejected objects. Else include i 2 S. Trivially, $S \cup \bar{S} = O_1$ and $S \setminus \bar{S} = ;$. β 2 $(0, 1]$ will be chosen later.

The resulting fractional solution is a solution to (LP2). Convert it to a solution to (LP1) (as described in Condition 1) and follow algorithm A to assign objects in S to O_2.

Theorem 5. *Let P satisfy Conditions 1(a) and 2. If there is a ρ-approximation algorithm for P w.r.t $(LP1)$, then B is a $(1+\rho)$-approximation algorithm for P_R w.r.t $(LP1_R)$.*

Proof. Let $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ be an optimal solution to $(LP1_R)$ with objective value $LP1_R = \gamma + \delta$, where $\gamma = \mathbf{c^T x} + \mathbf{d^T y}$, and $\delta = \mathbf{g^T z}$. The cost of rejecting objects in $\bar{S}$ is at most $\frac{\delta}{1-\beta}$. The term γ can be written as $\gamma^S + \gamma^{\bar{S}} + \mathbf{d^T y}$, where γ^S is the optimal cost of fractionally assigning objects in S to objects in O_2 and $\gamma^{\bar{S}}$ is the optimal cost of fractionally assigning objects in $\bar{S}$ to objects in O_2. Denote by x_S the vector x restricted to elements x_{ij} such that $i \in S$. By Condition 2, $\mathbf{q} = (\mathbf{x_S}, \mathbf{y})$ is a feasible solution to $(LP2)(S)$ of $P(S)$, with value $v = \gamma^S + \mathbf{d^T y}$. Since Condition 1(a) is satisfied by P, $\mathbf{q}$ can be converted to a feasible solution to $(LP1)(S)$ of value at most αv. Therefore B is a $(\max\{\alpha, \frac{1}{1-\beta}\gamma\})$-approximation algorithm for P_R w.r.t $(LP1_R)$. Note note that $(\max\{\alpha, \frac{1}{1-\beta}\gamma\})$ is minimized for $\beta = \frac{\gamma}{\gamma+1}$ yielding a performance guarantee of $(1+\rho)$.

Theorem 6. *Let P satisfy Conditions 1(b) and 2. If there is a ρ-approximation algorithm for P w.r.t $(LP1)$, then B is a $(2/(\rho+2-\sqrt{\rho(\rho+4)}))$-approximation algorithm for P_R w.r.t $(LP1_R)$.*

4 Applications

Scheduling Problem. We consider the problem of scheduling jobs on a single machine subject to release dates and precedence constraints, $1|r_j, prec|(\sum_S w_j C_j + \sum_{\bar{S}} e_j)$. Hall, Schulz, Shmoys and Wein [8] have given a 3-approximation algorithm for the corresponding problem without rejection $1|r_j, prec|\sum w_j C_j$ based on a time-indexed linear-programming relaxation [20] of the problem. It can be shown that this relaxation of $1|r_j, prec|(\sum_S w_j C_j + \sum_{\bar{S}} e_j)$ satisfies Conditions 1(b) and 2. So for the rejection variant of this problem, by Theorem 6 with $\rho = 3$ and $\beta = 0.791$, we have

Theorem 7. *There is a 4.79-approximation algorithm for $1|r_j, prec|(\sum_S w_j C_j + \sum_{\bar{S}} e_j)$, under the assumption that $\max_j r_j + \sum_{j=1}^{n} p_j$ is polynomial in n.*

A polynomial-size interval-indexed formulation [16,20] can be used to obtain an improved (4.496)-approximation. This formulation is not of the form (IP) but can be shown to satisfy an extension to Theorem 6.

Uncapacitated Facility Location Problem. In this problem, we are given a set of locations $N = \{1, \ldots, n\}$ and subsets $F, D \subseteq N$. We may open a facility at location $i \in F$ for a non-negative cost f_i. For each location $j \in D$, there is a positive integral demand d_j that must be shipped to it. The cost of shipping a unit of demand from a facility at location i to a location j is given by c_{ij} where the c_{ij}'s are non-negative. Since we consider only the *metric* variant of the problem, the c_{ij}'s are symmetric and satisfy the triangle inequality.

We now add the additional constraint that each location $j \in D$ can either be assigned to some facility $i \in F$ in which case it incurs a cost of c_{ij} for each unit of demand shipped or it may be rejected incurring a penalty of e_j.

A sequence of papers [18], [7], [3] give approximation algorithms for the basic problem that are based on a linear programming relaxation. We show that the linear program can be modified and that Conditions 1(a),2 apply, yielding

Theorem 8. *There is a 2.736-approximation algorithm for the uncapacitated facility location problem with rejection.*

5 Integrality Gaps and Approximations for $1|r_j|\sum U_j$

We now move to a more traditional notion of rejection in scheduling, which is nonetheless not well understood. We have one machine and each job has a release date r_j and a due date d_j; a job j is essentially "rejected" ($U_j = 1$) if it cannot be scheduled to complete by its due date. An optimal solution minimizes $\sum U_j$.

5.1 The quality of fractional relaxations. Solving $1|r_j|\sum U_j$ to optimality is strongly *NP*-hard, [9] therefore we are interested in approximations. We distinguish between two optimization metrics. In the minimization problem one seeks to minimize $\sum_{j=1}^{n} U_j$. In the maximization problem one seeks to maximize the number of jobs that meet their deadline, i.e. $\sum_{j=1}^{n}(1 - U_j)$. A heavily used technique in approximation algorithms for scheduling problems is that of solving first a *fractional relaxation.* Typically, a relaxation in a single-machine setting allows jobs to be processed in a preemptive manner once released. We call the resulting schedule S_f, *fractional.* For $1|r_j|\sum U_j$, the contribution U_j of job j to the objective of a fractional relaxation is equal to $1 - z_j$, where $0 \leq z_j \leq 1$ is the fraction of job j that is processed before the deadline d_j. The optimum objective value of a fractional schedule would typically be superoptimal for the problem at hand. Hence if an efficient rounding algorithm is invoked to convert the fractional schedule into a feasible one, it will incur some degradation to the fractional objective. We study gaps between the fractional and the true optima. We use $p_{\max}$ and $p_{\min}$ to denote the maximum and minimum processing times among the input jobs.

Lemma 5. *There is an instance J of $1||\max \sum_{j=1}^{n}(1 - U_j)$ on which the value of a fractional schedule is $\Omega(\ln(p_{\max}/p_{\min}))$ times the true optimum.*

Theorem 9. *There is an instance J of $1|r_j, p_j = p|\min \sum_{j=1}^{n} U_j$ on which the value of a fractional schedule is at most $2/n$ times the true optimum.*

5.2 An IP formulation and approximation algorithms. The results in the previous section indicate that fractional relaxations may give weak bounds for approximation. We show that nonetheless they can yield solutions of good quality for the maximization problem; we give an IP formulation and an approximation algorithm associated with its linear relaxation. The formulation we propose uses interval-indexed variables and thus has polynomial size. We define the set I to contain all disjoint time intervals of the form $(a, b]$ where a, b are consecutive release times or deadlines, i.e. for no other release time or deadline c is it the case that $a < c < b$. Therefore $|I| = O(n)$. The intervals in I are numbered so we refer unambiguously to $i \in I$. If $i = (a, b]$ the *length* $l(i)$ of i is $b - a$. The set $I(j)$ of *legal* intervals for job j is the set of intervals $(a_{j_k}, b_{j_k}]$ in I such that $r_j \leq a_{j_k}$ and $b_{j_k} \leq d_j$. In the ensuing formulation we use the variable x_{ij} to denote the time spent processing job j during interval $i \in I$.

$$\text{minimize} \quad \sum_{j=1}^{n} U_j \tag{5.1}$$

$$\text{subject to} \quad \sum_{i \in I(j)} x_{ij} \geq p_j \qquad j = 1, \ldots, n \tag{5.2}$$

$$\sum_{j=1}^{n} x_{ij} \leq l(i) \qquad i \in I \tag{5.3}$$

$$1 - \frac{\sum_{i \in I(j)} x_{ij}}{p_j} \leq U_j \qquad j = 1, \ldots, n \tag{5.4}$$

$$x_{ij} = 0 \qquad i \notin I(j) \qquad j = 1, \ldots, n \tag{5.5}$$

$$x_{ij} \in Z^+ \tag{5.6}$$

$$U_j \in \{0, 1\} \qquad j = 1, \ldots, n \tag{5.7}$$

It can be seen that the integer program $IP1$ formed by equations (5.1) through (5.7) is a valid relaxation of the problem. Observe that even in an integer solution to $IP1$ a job j may be processed in a preemptive manner. Call LP_{min} the linear relaxation of $IP1$. Let LP_{max} be the linear relaxation of $IP1$ with (1) replaced with "maximize $\sum_{j=1}^{n}(1 - U_j)$". It can be shown that the gaps in Lemma 5 and Theorem 9 apply also for the fractional optima of LP_{min}, LP_{max}.

Lemma 6. *There is an algorithm that outputs in polynomial time a solution to* $1|r_j, p \leq p_j \leq 2p|\max \sum_{j=1}^{n}(1 - U_j)$ *of value at least* $\lceil z/10 \rceil$, *where* z *is the optimum of* LP_{max} *for the problem.*

Theorem 10. *There is an algorithm that outputs in polynomial time a solution to* $1|r_j|\max \sum_{j=1}^{n}(1 - U_j)$ *of value at least* $\lceil z/(10\lceil \log(p_{\max}/p_{\min}) \rceil) \rceil$, *where* z *is the optimum of* LP_{max} *for the problem.*

Acknowledgments. We are thankful to David Shmoys for providing pointers to the literature and giving us access to [11] and to David Williamson for helpful discussions.

References

1. Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. In *Proc. 7th SODA*, 95–103, 1996.
2. C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *Proc. 8th SODA*, 609–618, 1997.
3. F. A. Chudak. Improved approximation algorithms for uncapacitated facility location. In Proc. 6th IPCO 1998. To appear.
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.
5. M. Goemans. Improved approximation algorithms for scheduling with release dates. In *Proc. 8th SODA*, 591–598, 1997.
6. R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.
7. S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. In *Proc. 9th SODA*, 1998.
8. L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, (3):513–544, August 1997.
9. E. L. Lawler. Scheduling a single machine to minimize the number of late jobs. Preprint, Computer Science Division, Univ. of California, Berkeley, 1982.
10. E. L. Lawler and J. M. Moore. A functional equation and its application to resource allocation and sequencing problems. In *Manag. Sci.*, volume 16, 77–84, 1969.
11. E. L. Lawler and D. B. Shmoys. Weighted number of late jobs (preliminary version). To appear in: J.K. Lenstra and D.B. Shmoys (eds.) Scheduling, Wiley.
12. Maxwell. Personal communication. 1996.
13. I. M. Ovacik and R. Uzsoy. *Decomposition Methods for Complex Factory Scheduling Problems.* Kluwer Academic Publishers, 1997.
14. C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Proc. of 4th WADS, LNCS, 955*, 86–97, Berlin, 1995. Springer-Verlag. To appear in Mathematical Programming B.
15. M. H. Rothkopf. Scheduling independent tasks on parallel processors. In *Manag. Sci.*, volume 12, 437–447, 1966.
16. A. S. Schulz and M. Skutella. Random–based scheduling: New approximations and LP lower bounds. In J. Rolim, editor, *Randomization and Approximation Techniques in Computer Science, LNCS, 955*, 119 – 133. Springer, Berlin, 1997.
17. A. S. Schulz and M. Skutella. Scheduling–LPs bear probabilities: Randomized approximations for min–sum criteria. In R. Burkard and G. Woeginger, editors, *Algorithms – ESA'97, LNCS, 1284*, 416 – 429. Springer, Berlin, 1997.
18. D. B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *Proc. of the 29th ACM STOC*, 265–274, 1997.
19. W.E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.
20. M. E. Dyer and L. A. Wolsey. Formulating the single machine sequencing problem with release dates as a mixed integer program. In *Discrete Applied Mathematics*, 26, 255-270, 1990.

Computer-Aided Way to Prove Theorems in Scheduling

S.V. Sevastianov[1] and I.D. Tchernykh[2]

[1] Institute of Mathematics,
Siberian Branch of the Russian Academy of Sciences,
Universitetskii prosp. 4, 630090, Novosibirsk-90, Russia
seva@math.nsc.ru
[2] Department of Algebra and Mathematical Logic,
Novosibirsk State Technical University, pr. Marksa, 20,
630092, Novosibirsk-92, Russia
metal@dtcher.nsu.ru

Abstract. For two scheduling problems ($O3|\,|C_{\max}$ and $AL3|\,|C_{\max}$) tight bounds of the optima localization intervals are found in terms of lower bounds ($\tilde{C}$ and $\hat{C}$, respectively) computable in linear time. The main part of the proof was made with an aid of computer. As a by-product, we obtain linear-time approximation algorithms for solving these problems with worst-case performance ratios 4/3 and 5/3, respectively.

1 Introduction

This paper concerns probably the first experience in using computer programs to prove theorems in scheduling. Not that the authors wanted to share the laurels of Appel and Haken [1] who solved the four-color problem with a help of computer, but we just encountered the fact that we cannot avoid using the computer while solving some scheduling problems.

The history of creating this paper should be started from the article by Gonzalez and Sahni [3], who introduced a new scheduling model called an *open shop* model. Following a standard classification scheme [5], we will denote it by $O|\,|C_{\max}$ and by $Om|\,|C_{\max}$ for an arbitrary and a fixed number m of machines, respectively. It was shown in [3] that the problem of constructing the shortest schedule is NP-hard even for three machines. For the case of two machines, they presented a linear-time algorithm for solving the problem to the optimum.

In 1982 Bárány and Fiala [2] presented a very simple (greedy) algorithm (due to Račsmany) for solving the open shop problem with ratio performance guarantee of 2. This result stayed unsurpassed for more than 10 years, until Chen and Strusevich investigated the open shop problem with three machines with an objective of creating an approximation algorithm with better performance guarantee. They proved ([4]) that any greedy algorithm for $O3|\,|C_{\max}$ has a worst-case performance ratio of $\frac{5}{3}$ and created another algorithm with worst-case performance ratio of $\frac{3}{2}$, the ratio $\frac{3}{2}$ being attained for the best of two schedules constructed.

G. Bilardi et al. (Eds.): ESA'98, LNCS 1461, pp. 502–513, 1998.
c Springer-Verlag Berlin Heidelberg 1998

While investigating this algorithm, we found that in some cases the bound $\frac{3}{2}$ can be improved by applying a third schedule, in the remaining cases this improvement can be achieved by applying a fourth, fifth, etc. schedule, so why should we stop on the bound $\frac{3}{2}$? We started to consider cases, subcases, subsubcases, etc.; after 3 years of intensive research it became clear that we were going a wrong way and our approach should be revised.

What exactly was proved by Chen and Strusevich? To formulate this, we need some notation. Let $l_{\max}$ be the maximum machine load (the sum of processing times of all operations on the most busy machine), $d_{\max}$ be the maximum job length (the sum of processing times of all operations of the longest job), $\tilde{C} \doteq \max\{l_{\max}; d_{\max}\}$. As our objective is to minimize the makespan ($C_{\max}(S)$), the following trivial lower bound on the optimum $C_{\max}$ is evident:

$$C_{\max} \quad \tilde{C}: \tag{1}$$

If we use only this lower bound on the optimum, then to substantiate a ratio performance guarantee $C_{\max}(S)=C_{\max}$ for some , we have to prove that for any problem instance, a schedule S can be constructed that meets the bound $C_{\max}(S) \quad \tilde{C}$.

A result of exactly this type was obtained by Chen and Strusevich. They proved that for any instance of a three-machine open shop, a schedule S can be constructed such that $C_{\max}(S)\ 2\ [\tilde{C}; \frac{3}{2}\tilde{C}]$. Now the question is, what is the minimal interval (in terms of $\tilde{C}$) which contains the optimum of any problem instance? It is evident that we cannot improve the lower bound ($\tilde{C}$) of the above interval. There is a simple instance with 4 jobs and the vectors of processing times $(1; 1; 1); (2; 0; 0); (0; 2; 0); (0; 0; 2)$, for which $C_{\max} = \frac{4}{3}\tilde{C}$ holds. Is it a worst-case upper bound on $C_{\max}$ in terms of $\tilde{C}$? To get an answer, we had written a computer program which output "Yes" after more than 200 hours of running time. Thus, we have found tight bounds of the optima localization interval for all instances of the $O3|jj|C_{\max}$ problem in terms of $\tilde{C}$, which is a contents of Theorem 1.

As a by-product, we present an algorithm with running time $O(n)$ (where n is the number of jobs) which constructs a schedule S with the makespan in the interval $[\tilde{C}; \frac{4}{3}\tilde{C}]$ (which is a contents of Theorem 2). This guarantees a worst-case performance ratio of $\frac{4}{3}$ for our algorithm. As follows from Theorem 1, this algorithm is the best (both in time complexity and in worst-case ratio) among those algorithms that use no lower bounds on the optimum other than (1) to get grounds for their performance.

What is the value of our algorithm in comparison with the approximation scheme for $Om|jj|C_{\max}$ recently presented in [8]? Any algorithm $A_{"}$ from this scheme has a ratio performance guarantee of $(1 + ")$ while its running time is not greater than $O(n)$ plus an additive constant depending on m and $"$. This constant is an upper bound on the time needed for an optimal scheduling of so-called "large" jobs of a given instance. The number of such jobs can be evaluated from above by $10(\frac{m}{"})^{\frac{m}{"}-2}$; in our case $(m = 3;\ " = \frac{1}{3})$ this number is $10 \quad 9^7$ 50000000.

So, for any n 50000000, if we decide to use the algorithm $A_{\frac{1}{3}}$ from the approximation scheme to solve the $O3|jj|C_{\max}$ problem with the ratio performance guarantee of $\frac{4}{3}$, we have to solve it to the optimum. Although, it was clear beforehand that the approximation scheme was not dedicated to practical applications, but rather to determination of the complexity status of the $Om|jj|C_{\max}$ problem. And this purely theoretical question was successfully solved: while it is proved in [9] that for $O|jj|C_{\max}$ problem (i.e., for the open shop problem when m is part of the input) there is no polynomial approximation algorithm with ratio performance guarantee better than $\frac{5}{4}$ (unless $P = NP$), an approximation scheme exists in the case of any fixed m.

Now, a few words about the idea behind the computer-aided proof of Theorem 1. The main insight in the proof can be referred to as "jobs number reduction" and is formalized in Lemma 1. The fact is, that in order to prove Theorem 1 for any n, it is sufficient to prove it for $n = 5$. Thus, a practical algorithm of solving the $O3|jj|C_{\max}$ problem with ratio performance guarantee of $\frac{4}{3}$ is as follows. First, we reduce a given instance into an instance with no more than 5 "aggregated" jobs (preserving the property d_j $\tilde{C}$ while $\tilde{C}$ preserves its initial value, where d_j denotes the length of job J_j); second, for the reduced instance with n 5 jobs we build a schedule S with $C_{\max}(S)$ $\frac{4}{3}\tilde{C}$ which exists due to Theorem 1. We do not even need to find an optimal schedule for this instance (unlike the approximation scheme discussed above).

The idea of jobs number reduction can be applied to other shop scheduling problems for getting results similar to Theorems 1 and 2. To illustrate this in our paper, we apply this method to a three-machine assembly line problem ($AL3|jj|C_{\max}$). For this problem, a similar trivial lower bound $\hat{C}$ on the optimum is defined. It is proved in Theorem 3 that for any $AL3|jj|C_{\max}$ problem instance its optimum belongs to the interval $[\hat{C}; \frac{5}{3}\hat{C}]$, and the bounds of the interval are tight. This result has also a practical application, formulated in Theorem 4. It consists in the fact that for any instance of the $AL3|jj|C_{\max}$ problem, a schedule S with the makespan $C_{\max}(S)$ $\frac{5}{3}\hat{C}$ can be found in linear time. For reference, in [7] an approximation algorithm with better ratio performance of $\frac{3}{2}$ but worse running time $O(n \log n)$ was presented for $AL3|jj|C_{\max}$. So, an improvement of ratio performance guarantee can be bought for the price of running time increasing, and it is fair.

The remainder of the paper is organized as follows. Sect. 2 contains a formalization of the open shop problem and the main Theorem 1. A description of the computer-aided proof of Lemma 2 is given in Sect. 3. An application of the jobs number reduction method to $AL3|jj|C_{\max}$ is described in Sect. 4. Some concluding remarks are given in Sect. 5.

2 Optima Localization for the Open Shop Problem

Let us formally define the notions mentioned in the previous section. Consider the classical open shop scheduling problem. Let $M = \{M_1, \ldots, M_m\}$ be the machine set, $J = \{J_1, \ldots, J_n\}$ be the job set, p_i^j be the processing time of an operation O_i^j of job J_j on machine M_i. Thus, for each job J_j a *vector of processing times* $p^j \doteq (p_1^j, \ldots, p_m^j)$ is defined. In the case of three machines, we will use A, B, C instead of M_1, M_2, M_3, and a_j, b_j, c_j instead of p_1^j, p_2^j, p_3^j, respectively. Let $C_{\max}(S)$ be the makespan (i.e., the maximum completion time) of a schedule S. Let $I_i \doteq \sum_{j=1}^{n} p_i^j$ stand for the load of machine M_i, $I_{\max} \doteq \max_{i=1,\ldots,m} I_i$ be the maximum machine load, $d_j \doteq \sum_{i=1}^{m} p_i^j$ denote the length of job J_j, $d_{\max} \doteq \max_{j=1,\ldots,n} d_j$ stand for the maximum job length, $\tilde{C} \doteq \max\{I_{\max}, d_{\max}\}$, $C_{\max} \doteq \min_S C_{\max}(S)$ be the optimum makespan of a given instance. We wish to find the minimal function $\eta(m)$ such that the inequality

$$C_{\max} \leq \eta(m)\tilde{C}$$

holds for any instance of the $O|m|C_{\max}$ problem. The following simple instance shows that $\eta(m)$ cannot be less than $\frac{3}{2} - \frac{1}{2m}$ for odd m. Indeed, it is sufficient to take $(m+1)$ jobs with the following vectors of processing times: $(1, 1, \ldots, 1)$; $(m-1, 0, \ldots, 0)$; $(0, m-1, 0, \ldots, 0)$; $\ldots$; $(0, \ldots, 0, m-1)$. For this instance, we have $\tilde{C} = m$, whereas

$$C_{\max} = \min_k \max\{k, m-k\} + m - 1 = \left\lceil \frac{m}{2} \right\rceil + m - 1 = \frac{m+1}{2} + m - 1 = \frac{3}{2}m - \frac{1}{2}.$$

The same instance defined for the first $m-1$ machines yields the bound $\eta(m) \geq \frac{3}{2} - \frac{1}{2(m-1)}$ for the case of even m. (It would be strange if this instance — with zero load of one machine — turned out to be the worst one. However, we can suggest nothing better for the case of even m.)

As is shown in the following Theorem 1, this lower bound on $\eta(m)$ is tight for $m = 3$.

Theorem 1 (the main result). *For any instance of the $O3|j|C_{\max}$ problem, its optimum belongs to the interval $[\tilde{C}; \frac{4}{3}\tilde{C}]$ and the bounds of this interval are tight.*

Proof. The tightness of the lower bound $\tilde{C}$ is evident. The "bad" instance described above shows that the upper bound $\frac{4}{3}\tilde{C}$ cannot be improved. It remains to prove that $C_{\max} \leq \frac{4}{3}\tilde{C}$ holds for any instance of the $O3|j|C_{\max}$ problem. This in turn is straightforward from the following two lemmas.

Lemma 1. *For any number of machines m, the supremum $\alpha = \sup C_{\max}/\tilde{C}$ over all instances of the $O|m|jC_{\max}$ problem is attained on an instance with at most $2m-1$ jobs.*

Lemma 2. *For any instance of the $O3|jj|C_{\max}$ problem with 5 jobs, the optimum meets the bound*

$$C_{\max} \quad \frac{4}{3}\tilde{C}: \tag{2}$$

Proof (of Lemma 1). . It is clear that $\sum_{j=1}^{n} d_j = \sum_{i=1}^{m} l_i \quad m\tilde{C}$. Suppose we are given an instance of $Om|jj|C_{\max}$ with $n > 2m-1$ jobs, for which the supremum is attaining. We want to show, that there exists an instance with lesser number of jobs, for which the ratio $C_{\max}/\tilde{C}$ is not less than that for the initial instance. Suppose that there are two jobs J_{j_1} and J_{j_2} in the instance such that d_{j_1} $\tilde{C}/2$; d_{j_2} $\tilde{C}/2$. In this case we combine them to a new large job with processing times $p_i^{j_1} + p_i^{j_2}$ of its operations on machines M_i; $i = 1;:::;m$. So, we unite jobs J_{j_1} and J_{j_2} in a block which cannot be divided in a schedule, and treat this block as a new job instead of J_{j_1} and J_{j_2}. Since any feasible schedule for the modified instance is a feasible schedule for the initial one, this transformation does not lead to decreasing the optimum. It is also evident that this transformation does not affect the value of $\tilde{C}$.

Now assume that the instance contains at most one job J_j with d_j $\tilde{C}/2$. Suppose next that our instance contains more than $(2m-1)$ jobs. (Otherwise we are done.) Choose $(2m-2)$ "large" jobs with $d_j > \tilde{C}/2$. Their total length $\sum d_j > (m-1)\tilde{C}$. Therefore, the total length of the remaining jobs is $\sum d_j < \tilde{C}$ (due to the inequality $\sum_{j=1}^{n} d_j \quad m\tilde{C}$). This implies that all the remaining jobs can be combined into one block without affecting $\tilde{C}$. The resulting instance contains at most $(2m-1)$ jobs. Since this was done without decreasing the value of $C_{\max}$ and changing $\tilde{C}$, the claim follows. □

Note that the transformation described requires at most $O(n)$ time units.

Proof (of Lemma 2). is described below in Sect. 3. □

Thus, the claim of Theorem 1 follows. □

A practical application of this result is described in the following

Theorem 2. *For any $O3|jj|C_{\max}$ problem instance, a schedule S with $C_{\max}(S) \in [\tilde{C}; \frac{4}{3}\tilde{C}]$ can be constructed in linear time.*

3 A Description of the Proof of Lemma 2

Since our computer-aided proof of Lemma 2 cannot be fully presented in this paper, we will give only a description of the proof. In this section we assume that $\tilde{C} = 1$ and measure all operation processing times in these units.

In order to entrust the proof to a computer, we have to reduce the number of cases we need to consider to a finite one. Although we have reduced our problem to a case with a limited number of jobs (the number of machines was fixed in advance), we still have an infinite number of instances. The following observation allows us to avoid considering particular instances.

Note that any feasible schedule for a given instance is determinated by some network which specifies an order of operations for every job and every machine. On the other hand, for any such network it is sufficient to consider the only (so-called *early*) schedule, in which each operation is started as early as possible. The number of such networks is finite (in our case, it does not exceed 15!). We want to show that for any $O3|jj|C_{\max}$ instance there exists a network such that a schedule specified by this network meets bound (2).

As is known from calendar planning, the makespan of the schedule specified by a network G and an instance I is equal to the length of a *critical path* in this network, which, in turn, must be a *complete* path in the network. We will treat a network as a set of its complete paths.

Consider a network G_1 shown in Fig. 1 *a)*.

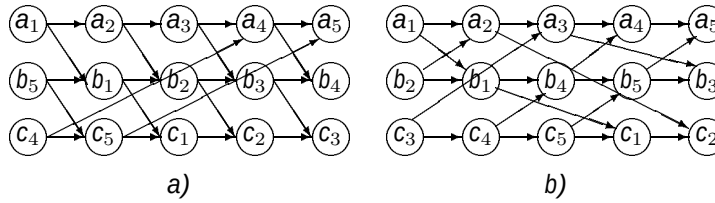


Fig.1. The schemes of *a)* a cyclic network G_1, *b)* and a network of the second type for $O3|jj|C_{\max}$ problem

Similar networks and corresponding schedules will be referred to as *cyclic*. Note that G_1 can be transformed to any cyclic network by applying a proper permutation of jobs and a permutation of machines. Such networks will be denoted by *Cyclic*(;).

Network G_1 contains 21 complete paths: $a_1 \; ! \; a_2 \; ! \; a_3 \; ! \; a_4 \; ! \; a_5$; $a_1 \; ! \; a_2 \; ! \; a_3 \; ! \; a_4 \; ! \; b_4$, etc. We will not consider paths with length evidently not greater than $\tilde{C}$, e.g., $a_1 \; ! \; a_2 \; ! \; a_3 \; ! \; a_4 \; ! \; a_5$. Eliminating such paths, we leave only 16 *non-trivial* complete paths.

For an arbitrary instance I, we do not know beforehand which path in G_1 will become critical. Thus, we should consider all the 16 paths separately, assuming that this particular path became critical in G_1.

For some instances, the network G_1 can be extremely bad, so we will consider also a cyclic network $G_2 \doteq Cyclic((5; 4; 3; 2; 1); (1; 2; 3))$. These two networks ($G_1$ and G_2) will be referred to as *basis* networks.

Now assume that an instance given corresponds to critical paths $p_1; p_2$ in networks G_1 and G_2, respectively. We want to show that for any such combination (p_1 *2* $G_1; p_2$ *2* G_2) and for any particular instance there exists a sequence of schedules such that the makespan of the shortest one among them does not exceed $\frac{4}{3}$. We have to consider every such combination independently to complete the proof of Lemma 2.

How does our program deal with any combination? We will illustrate this on the following example.

Consider a combination $(p_1; p_2)$, where p_i is a critical path from G_i. Let, for instance, $p_1 = a_1 \; ! \; a_2 \; ! \; a_3 \; ! \; a_4 \; ! \; b_4$; $p_2 = a_5 \; ! \; a_4 \; ! \; b_4 \; ! \; b_3 \; ! \; c_3$.

Notation p_i will be further used for not only the actual path, but also for its length. Thus, $p_1 = a_1 + a_2 + a_3 + a_4 + b_4;\ p_2 = a_5 + a_4 + b_4 + b_3 + c_3$. Denote schedules corresponding to networks G_i as S_i. We are under assumption that $C_{\max}(S_1) = p_1;\ C_{\max}(S_2) = p_2$. Summing, we obtain

$$C_{\max}(S_1) + C_{\max}(S_2) = (a_1 + a_2 + a_3 + a_4 + a_5) + (a_4 + b_4) + (b_4 + b_3) + c_3 \quad 4;$$

therefore, $C_{\max}(S_1 \wedge S_2) \quad 2$ (where $S_1 \wedge S_2$ denotes the shortest schedule among S_1 and S_2). This bound is attained for some *critical instance* which meets the following constraints:

$$C_{\max}(S_1) = a_1 + a_2 + a_3 + a_4 + b_4 = 2;\ C_{\max}(S_2) = a_5 + a_4 + b_4 + b_3 + c_3 = 2;$$

$$a_1 + a_2 + a_3 + a_4 + a_5 = 1;\ a_4 + b_4 = 1;\ b_4 + b_3 = 1;\ c_3 = 1:$$

Solving this linear system, we obtain $b_4 = c_3 = a_1 + a_2 + a_5 = 1$, while the remaining operations have zero length. For this critical instance, we can easily construct a schedule S with the makespan of 1. Our next step is to find a network G_3 such that the schedule S corresponds to G_3 and to the critical instance obtained. For this purpose, we can use a network $Cyclic((3;4;5;2;1);(1;2;3))$ Now we should consider 16 more subcases, since G_3 contains 16 (non-trivial) complete paths. Consider, for example, a path $p_3 = a_3 + a_4 + a_5 + b_5 + b_2$. Summing p_2 and p_3, we obtain

$$C_{\max}(S_2) + C_{\max}(S_3) = (a_1 + a_2 + a_3 + a_4 + a_5) + (a_4 + a_3) + (b_5 + b_2 + b_4) \quad 3;$$

therefore, in this case $C_{\max}(S_1 \wedge S_2 \wedge S_3) \quad \frac{3}{2}$. This improved bound is attained for the following instance:

$$b_4 = a_4 = a_3 = c_3 = b_5 + b_2 = \frac{1}{2};\ c_1 + c_2 + c_5 \quad \frac{1}{2};$$

other operations are of zero length.

For this critical instance, we can also find a schedule with the unit makespan. It corresponds to a network $G_4 = Cyclic((5;2;1;4;3);(1;2;3))$. Proceeding in this way, we have to consider 16 complete paths from G_4. Consider, for example, a path $p_4 = a_5 + a_2 + a_1 + a_4 + b_4$. Summing $p_1; p_3$ and p_4 we obtain:

$$C_{\max}(S_1) + C_{\max}(S_3) + C_{\max}(S_4) = 2(a_1 + a_2 + a_3 + a_4 + a_5) + (b_2 + b_4 + b_5) + a_4 \quad 4;$$

therefore $C_{\max}(S_1 \wedge \ldots \wedge S_4) \quad \frac{4}{3}$. Thus, this part of proof is completed, though we still have to consider other complete paths in G_4, then return to G_3 and continue this tree-like process. Although finite, this process is too long to perform it manually. Let us show, how computer can perform this proof. Assume that complete paths in any network G are enumerated in some way, and let $P_k(G_i)$ denote the k-th complete non-trivial path from a network G_i. Note that due to the symmetry of networks G_1 and G_2, we only need to consider 136 combinations $(P_i(G_1); P_j(G_2))$ with $i \quad j$ (since combinations $(P_i(G_1); P_j(G_2))$ and

$(P_j(G_1); P_i(G_2))$ can be transformed to each other by the reverse reordering of jobs).

The program we used to prove Lemma 2 is an implementation of the following

Algorithm A

```
BEGIN
  Let G1 = Cyclic((1;2;3;4;5);(1;2;3)); G2 = Cyclic((5;4;3;2;1);(1;2;3));
  FOR i := 1 TO 16 DO
    FOR j := i TO 16 DO
      BEGIN
        p1 := Pi(G1); p2 := Pj(G2); N := 2; paths := {p1; p2};
        PROVE(N; paths);
      END;
  WRITE("Yes")
END.
```

PROVE is a recursive procedure which branches a proof starting with a current node of the *tree of proof*, specified by the set *paths* and the number N corresponding to a current depth of the tree.

Procedure PROVE(N, *paths*).

Step 1. For N paths contained in *paths* calculate the best provable bound .
Find a critical instance I, for which is attained.
If $\frac{4}{3}$ then RETURN.

Step 2. Find, if possible, a schedule S for I with the makespan not greater than $\frac{4}{3}$.
If schedule S was not found, WRITE("No") and STOP.
Build an improving network G_{N+1} corresponding to schedule S.

Step 3. FOR $k := 1$ TO (number of paths in G_{N+1}) DO
BEGIN
$paths := paths \cup \{P_k(G_{N+1})\}$;
PROVE($N + 1$; *paths*);
$paths := paths \cap \{P_k(G_{N+1})\}$
END;

Note that if Algorithm A has output the "Yes" statement, this means that the claim of Lemma 2 is valid.

Now a few words about the implementation of Steps 1 and 2 of procedure PROVE. Let paths $p_1; \ldots; p_N$ be an input of PROVE. Each path p_i is a linear combination of operations $(a_j; b_j; c_j; j = 1; \ldots; 5)$ with coefficients $\frac{i}{k}$ belonging to $\{0; 1\}$. A problem described at Step 1 can be simulated by the following linear programming problem with 16 variables $(a_j; b_j; c_j;$):

$$! \max$$

$$
\begin{cases}
I_1 = a_1 + a_2 + a_3 + a_4 + a_5 \quad 1; \\
I_2 = b_1 + b_2 + b_3 + b_4 + b_5 \quad 1; \\
I_3 = c_1 + c_2 + c_3 + c_4 + c_5 \quad 1; \\
d_1 = a_1 + b_1 + c_1 \quad 1; \\
\vdots \\
d_5 = a_5 + b_5 + c_5 \quad 1; \\
p_1 = \ {}_1^1 a_1 + \quad + \ {}_{15}^1 c_5 \quad ; \\
\vdots \\
p_N = \ {}_1^N a_1 + \quad + \ {}_{15}^N c_5 \quad ; \\
a_j; b_j; c_j \quad 0:
\end{cases}
$$

Our program uses the simplex algorithm to solve this problem. An implementation of the procedure of searching for an improving network at Step 2 is made as follows. We have a reasonably limited number of networks used. In the beginning we used only cyclic networks, starting with G_1 and G_2 described above. At Step 2 the program searches through all networks known to it in order to find the best one for the current instance, i.e., a network minimizing the makespan of the corresponding schedule. In the case when the minimal makespan is greater than $\frac{4}{3}\tilde{C}$, the program gives the "No" output, and this means that the above limitations on network classes are too strong, and we should add another class of networks to the program's data bank. In fact, the case described occurred only once when the program found an instance with the following vectors of processing time: $(\frac{1}{2}; 0; \frac{1}{2})$; $(0; 1; 0)$; $(\frac{1}{2}; 0; \frac{1}{2})$. It is easy to show that the makespan of any cyclic schedule for this instance is not lesser than $\frac{3}{2}$, while the optimum makespan is equal to 1. We added the second type of networks, shown in Fig. 1*b*. Thus, we used only two types of networks (and 2 3! 5! = 1440 networks in all), and the successful run of the program showed that to prove Lemma 2, it is sufficient to use only these networks. Although it seems to be possible to reduce the running time by adding another proper network type.

Being launched for the first successful time, the program output the "Yes" statement after more than 200 hours of running time. (We were using IBM PC-clone with 100MHz CPU speed). The implementation of Steps 1 and 2 took the largest part of time. After this first launch, a full *Tree of Proof* (i.e., the attributes of the networks found at Step 2) was saved. Now the second version of algorithm A (and also the second version of the program) has the following step instead of Steps 1,2:

Step 1-2. Read the network G_N corresponding to the current node $(N; paths)$. If this node is ending (i.e., without successors) then calculate the bound for paths from *paths* and make sure that it is not greater than $\frac{4}{3}$, otherwise output the "No" statement and STOP. (Actually, this "STOP" event can never happen.)

This version of the program runs considerably faster and can be verified by any meticulous reader. (The PASCAL code of this program can be found via Internet, URL: http://math.nsc.ru/LBRT/k4/seva.html.)

The Tree of Proof saved gives us also an approach to fast search for a schedule S (for any instance with at most 5 jobs) with $C_{\max}(S) \quad \frac{4}{3}\tilde{C}$. Assume, we are

given an instance I with at most 5 jobs. Build schedules S_1 and S_2, corresponding to networks G_1 and G_2. If $C_{\max}(S_1 \wedge S_2) > \frac{4}{3}$, choose the node $(2; fp_1; p_2g)$ from the Tree of Proof, where p_1 and p_2 are critical paths in G_1 and G_2, respectively, corresponding to instance I. Read the corresponding network G_3 and build the schedule S_3. Continue this process until $C_{\max}(S_i) \quad \frac{4}{3}\tilde{C}$. As the depth of the tree does not exceed 10 (as turned out while running the program), this method is very effective.

This completes the description of the proof of Lemma 2.

4 Optima Localization for the Assembly Line Problem

The assembly line problem can be formulated as follows. Let $M; J; p_i^j$ ($A; B; C;$ $a_j; b_j; c_j$ in the case of three machines) have the same meaning as in Sect. 2. Operations of job J_j on machines $M_1; : : : ; M_{m-1}$ can be processed simultaneously. An operation of J_j on machine M_m cannot start until all operations of this job on machines $M_1; : : : ; M_{m-1}$ are finished. Any machine can process at most one operation in time. No preemption is allowed. The objective is to find a feasible schedule with minimal makespan. We will denote this problem as $ALmjjC_{\max}$.

For $m = 2$ this problem is equivalent to the classical Johnson problem and can be solved by Johnson's algorithm ([6]). For $m \quad 3$ it is NP-hard ([7]).

We are presenting the result on optima localization for $AL3jjC_{\max}$, similar to Theorem 1. Let I_i and $\hat{d}_j$ denote the machine load and job length, correspondingly:

$$I_i = \sum_{j=1}^{X} p_i^j; \ \hat{d}_j = \max_{i=1;:::;m-1} p_i^j + p_m^j;$$

$$I_{\max} = \max I_i; \ \hat{d}_{\max} = \max \hat{d}_j; \ \hat{C} = \max fI_{\max}; \hat{d}_{\max}g:$$

For any schedule S, we have $C_{\max}(S) \quad \hat{C}$, therefore $C_{\max} \quad \hat{C}$. We wish to find the minimal function $\hat{}(m)$ such that the inequality $C_{\max} \quad \hat{}(m)\hat{C}$ holds for any $ALmjjC_{\max}$ problem instance. The following simple instance shows that $\hat{}(m)$ cannot be less than $2 - \frac{1}{m}$. Consider m jobs with the following vectors of processing times: $(1; : : : ; 1); \ (m - 1; 0; : : : ; 0; 1); : : : ; (0; : : : ; 0; m - 1; 1)$. For this instance we have $\hat{C} = m$, whereas $C_{\max} = 2m - 1$. The following theorem shows that this bound on $\hat{}(m)$ is tight for $m = 3$.

Theorem 3. *For any instance of $AL3jjC_{\max}$ problem, its optimum belongs to the interval $\hat{C}; \frac{5}{3}\hat{C}$ and the bounds of this interval are tight.*

Proof. Proof of this theorem is fully similar to that of Theorem 1. The tightness of the interval is evident from the existence of the instance described above; the remainder of the proof is straightforward from the following two lemmas:

Lemma 3. *For any number of machines m, the supremum $= \sup C_{\max}=\hat{C}$ over all instances of the $ALmjjC_{\max}$ problem is attained on an instance with at most $2m - 1$ jobs.*

Lemma 4. *For any instance of the $AL3|jj|C_{\max}$ problem with 5 jobs, its optimum meets the bound $C_{\max} \leq \frac{5}{3}\hat{C}$.*

Proofs of these lemmas are analogous to those of Lemmas 1 and 2 and can be omitted. The computer-aided proof of Lemma 4 is more easy than that of Lemma 2, because it is shown in [7] that for $AL|m|jj|C_{\max}$ problem an optimal schedule can be found among permutation ones. So, we had to use the only network type, shown in Fig. 2. □

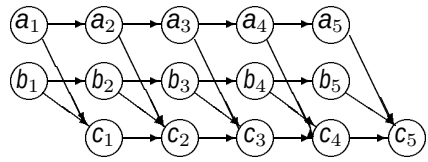


Fig.2. The scheme of a permutation network for $AL3|jj|C_{\max}$ problem

A practical application of this result is described in the following

Theorem 4. *For any $AL3|jj|C_{\max}$ problem instance a schedule S such that $C_{\max}(S) \leq \hat{C}, \frac{5}{3}\hat{C}$ can be constructed in linear time.* □

5 Concluding Remarks

1. We can summarize the approach applied in this paper for proving some properties of a shop scheduling problem as follows.

First, we prove that we can perform a jobs number reduction preserving some parameters (say, $\tilde{C}$, or $\hat{C}$, or $p_{\max}$) and leading to an instance with the limited number of jobs.

Second, we prove with an aid of computer that the desired property (depending on the parameters preserved) holds for any instance with the limited number of jobs.

To apply this approach in its constructive form (given an instance, we need to construct its schedule that meets a desired property), we do as follows.

First, apply the jobs number reduction to a given instance, and next, construct the desired schedule for the resulting instance with a limited number of jobs.

We are convinced that possible applications of the approach just outlined cannot be restricted to two scheduling problems considered in this paper. We would be glad to hear about other useful applications of this method.

2. Chen and Strusevich, being based on their result for $O3|jj|C_{\max}$ ([4]), formulated a conjecture that the makespan of any greedy schedule for $Om|jj|C_{\max}$ problem belongs to the interval $[\tilde{C}, (2-\frac{1}{m})\tilde{C}]$, and the bounds of this interval are tight. (The unimprovability of the upper bound was substantiated by the existence of a simple instance.) For the time being, this conjecture is still unproved, it was just corroborated for the cases $m = 2$ and $m = 3$.

In order to make a stimulus for further investigation of the open shop problem, we are presenting a conjecture also corroborated yet for m 3, and a similar conjecture for the assembly line problem.

Conjecture 1 *For any instance of the* $Omjj C_{\max}$ *problem with an odd number of machines* m*, its optimum* $C_{\max}$ *belongs to the interval* $[\tilde{C}; (\frac{3}{2} - \frac{1}{2m})\tilde{C}]$*, and the bounds of the interval are tight.*

Conjecture 2 *For any instance of the* $ALmjjC_{\max}$ *problem, its optimum* $C_{\max}$ *belongs to the interval* $[\hat{C}; (2 - \frac{1}{m})\hat{C}]$*, and the bounds of the interval are tight.*

Lower bounds are evident, whereas unimprovability of the upper bounds is shown in Sections 2 an 4, respectively.

3. To conclude the paper, we would like to put the most intriguing question: does there exist an analytical proof of Theorem 1 which needs no assistance of a computer?

6 Acknowledgments

We would like to thank our anonymous referee whose constructive remarks helped to eliminate some inaccuracy committed in the paper.

References

1. Appel, K., Haken, W.: Every planar map is four colorable. *A.M.S. Contemporary Math.* 98 (1989), 741 pp.
2. Bárány, I., Fiala, T.: Nearly optimum solution of multimachine scheduling problems. *Szigma - Mat.-Közgaz-dasagi Folyoirat*, 15 (1982) 177–191.
3. Gonzalez, T., Sahni, S.: Open shop scheduling to minimize finish time. *J. Assoc. Comput. Mach.*, 23 (1976) 665–679.
4. Chen, B., Strusevich, V.A.: Approximation algorithms for three-machine open shop scheduling. *ORSA Journal on Computing*, 5 (1993) 321–326.
5. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B.: Sequencing and scheduling: algorithms and complexity. *Handbooks in Operations Research and Management Science*, Amsterdam, 1993, Vol. 4, 445–522.
6. Johnson, S.M.: Optima two- and three-stage production schedules with setup times included. *Naval Res. Logist. Quart.*, 1 (1954), 61–68.
7. Potts, C.N., Sevast'janov, S.V., Strusevich, V.A., Van Wassenhove, L.N.,Zvaneveld, C.M.: The two-stage assembly scheduling problem: complexity and approximation. *Oper. Res.* 43 (1995) 346–355.
8. Sevastianov, S.V., Woeginger, G.J.: Makespan Minimization in Open Shops: a Polynomial Time Approximation. *SFB-Report 68, Mai 1996*, TU Graz, Austria.
9. Williamson, D.P., Hall, L.A., Hoogeveen, J.A., Hurkens, C.A.J., Lenstra, J.K., Sevastianov, S.V., Shmoys, D.B.: Short Shop Schedules. *Oper.Res.*, 45 (1997), No.2, 288–294.

Author Index