Frank Dehne
Jörg-Rüdiger Sack
Roberto Tamassia (Eds.)

LNCS 2125

Algorithms and Data Structures

7th International Workshop, WADS 2001
Providence, RI, USA, August 2001
Proceedings

Springer

Lecture Notes in Computer Science 2125

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Frank Dehne Jörg-Rüdiger Sack
Roberto Tamassia (Eds.)

Algorithms and Data Structures

7th International Workshop, WADS 2001
Providence, RI, USA, August 8-10, 2001
Proceedings

Springer

Volume Editors

Frank Dehne
Jörg-Rüdiger Sack
Carleton University, School of Computer Science
1125 Colonel By Drive, Ottawa, Candada K1S 5B6
E-mail: {dehne/sack}@scs.carleton.ca

Roberto Tamassia
Brown University, Center for Geometric Computing
Providence, RI 02912-1910, USA
E-mail: rt@cs.brown.edu

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Algorithms and data structures : 7th international workshop ; proceedings / WADS 2001, Providence, RI, USA, August 8 - 10, 2001. Frank Dehne ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2125)
ISBN 3-540-42423-7

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

ISSN 0302-9743
ISBN 3-540-42423-7 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Typesetting: Camera-ready by author, data conversion by PTP Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10839948 06/3142 5 4 3 2 1 0

Preface

The papers in this volume were presented at the Seventh Workshop on Algorithms and Data Structures (WADS 2001). The workshop took place August 8–10, 2001 in Providence, Rhode Island, USA. The workshop alternates with the Scandinavian Workshop on Algorithms Theory (SWAT), continuing the tradition of SWAT and WADS starting with SWAT '88 and WADS '89.

In response to the call for papers, 89 papers were submitted. From these submissions, the program committee selected 40 papers for presentation at the workshop. In addition invited plenary lectures were given by the following distinguished researchers: Mikhail J. Atallah, F. Thomson Leighton, and Mihalis Yannakakis.

On behalf of the program committee, we would like to express our appreciation to the invited speakers and to all the authors who submitted papers.

August 2001

Frank Dehne
Jörg-Rüdiger Sack
Roberto Tamassia

Organization

Conference Chair

Roberto Tamassia (Brown)

Program Committee Chairs

Frank Dehne (Carleton)
Jörg-Rüdiger Sack (Carleton)
Roberto Tamassia (Brown)

Program Committee

Alberto Apostolico (Purdue, Padova)
Timothy Chan (Waterloo)
Bruno Codenotti (IMC)
Giuseppe Di Battista (Roma Tre)
Shlomi Dolev (Ben-Gurion)
Martin Farach-Colton
(Rutgers, Google)
Pierre Fraigniaud (LRI)
Harold N. Gabow (Colorado)
Sally A. Goldman (Washington)
Michael T. Goodrich (JHU)
Roberto Grossi (Pisa)
Magnus M. Halldórsson (UVS)
Samir Khuller (Maryland)
Rolf Klein (Bonn)
Jon Kleinberg (Cornell)
Giuseppe Liotta (Perugia)
Ernst W. Mayr (München)
Joseph S. B. Mitchell (SUNY)
Stefan Näher (Trier)
Takao Nishizeki (Tohoku)
Viktor K. Prasanna (USC)
Enrico Puppo (Genova)
Jose Rolim (Geneva)
Jack Snoeyink (UNC)
Ioannis G. Tollis (UTD)
Imrich Vrt'o (SAV)
Dorothea Wagner (Konstanz)
Tandy Warnow (Texas)
Sue Whitesides (McGill)
Peter Widmayer (ETHZ)

Invited Speakers

Mikhail J. Atallah (Purdue)
F. Thomson Leighton (MIT, Akamai)
Mihalis Yannakakis (Bell Labs)

Publicity Chair

Yi-Jen Chiang (Polytechnic)

Local Arrangements Chair

Galina Shubina (Brown)

Administrative Assistant

Frances Palazzo (Brown)

Sponsoring Institutions

Center for Geometric Computing, Brown University
Department of Computer Science, Brown University

Steering Committee

J. Ian Munro (Waterloo)
Frank Dehne (Carleton)
Jörg-Rüdiger Sack (Carleton)
Nicola Santoro (Carleton)
Roberto Tamassia (Brown)

Proceedings Production Editor

Galina Shubina (Brown)

Table of Contents

Session 7B

Session 8A

Session 8B

Session 9 – Invited Talk

Session 10A

Session 10B

Approximation of Multiobjective Optimization Problems

Mihalis Yannakakis

Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Abstract. We discuss problems in multiobjective optimization, in which solutions to a combinatorial optimization problem are evaluated with respect to several cost criteria, and we are interested in the trade-off between these objectives, the so-called Pareto curve. The Pareto curve has typically an exponential number of points. However, it turns out that, under general conditions, there is a polynomially succinct curve that approximates the Pareto curve within any desired accuracy. The central computational question is whether such an approximate curve can be constructed efficiently (in polynomial time). We discuss conditions under which this is the case. We examine in more detail the class of linear multiobjective problems, and relate the multiobjective approximation to the single objective case. We will discuss also problems in multiobjective query optimization.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, p. 1, 2001.

Optimal, Suboptimal, and Robust Algorithms for Proximity Graphs *

Ferran Hurtado[1], Giuseppe Liotta[2], and Henk Meijer[3]

[1] Departament de Matemàtica Aplicada II,
Universitat Politècnica de Catalunya, Barcelona, Spain.
hurtado@ma2.upc.es
[2] Dipartimento di Ingegneria Elettronica e dell'Informazione,
Università di Perugia, Perugia, Italy.
liotta@diei.unipg.it
[3] Department of Computing and Information Science,
Queen's University, Kingston, Ontario, Canada.
henk@cs.queensu.ca

Abstract. Given a set of n points in the plane, any β-skeleton and $[\gamma_0, \gamma_1]$ graph can be computed in quadratic time. The presented algorithms are optimal for β values that are less than 1 and $[\gamma_0, \gamma_1]$ values that result in non-planar graphs. For $\beta = 1$, we show a numerically robust algorithm that computes Gabriel graphs in quadratic time and degree 2. We finally show how a β-spectrum can be computed in optimal $O(n^2)$ time.

1 Introduction

A typical approach to extracting a shape from a given set P of n points is to compute a *proximity graph* of P, i.e. a geometric graph whose vertices are elements of P and where the edges are straight-line segments connecting pairs of points. In a proximity graph of P two points are connected by an edge if and only if their *region of influence* is *empty*, i.e. it does not contain any other element of P. The region of influence of two points u and v of P is a region of the plane that describes a neighbourhood of u and v; the emptiness of the region of influence witnesses that u and v are close enough to each other to be connected by an edge. Depending on the application context, different definitions of region of influence and of corresponding proximity graphs have been proposed in the literature. While the interested reader is referred to the comprehensive survey by Jaromczyk and Toussaint [9], we restrict ourselves to recalling some of the most widely studied proximity graphs.

* Research supported in part by the CNR Project "Geometria Computazionale Robusta con Applicazioni alla Grafica ed al CAD", the project "Algorithms for Large Data Sets: Science and Engineering" of the Italian Ministry of University and Scientific and Technological Research (MURST 40%), by Gen. Cat. SGR000356 and MEC-DGES-SEUID PB98-0933, Spain, and by NSERC, Canada.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 2–13, 2001.

A continuous hierarchy of proximity graphs that includes Gabriel graphs and relative neighbourhood graphs as special cases was first defined in the computational morphology context by Kirkpatrick and Radke [10]. The elements of this infinite family of proximity graphs are called *β-skeletons* and are defined by considering a continuous family of regions of influence indexed by a single real positive parameter β. The region of influence of two points u and v is called the *β-neighbourhood* of u and v. Its area is related to the value β; as β approaches 0, the β-neighbourhood of u and v approaches the line-segment (u, v), while as β increases the β-neighbourhood of u and v becomes larger. The β-neighbourhood of two points can be either *lune-based* or *circle-based*.

Another parametrized family of proximity graphs, known as *γ-graphs* which unifies circle-based β-skeletons, convex hulls, and Delaunay triangulations was defined by Veltkamp [18]. A precise definition of β-skeletons and γ-graphs will be given in Section 2.

This paper is devoted to the study of efficient algorithms for computing β-skeletons and γ-graphs. In order to better explain our contribution, we briefly review some literature about proximity graphs and then list our results. Existing algorithms that compute proximity graphs can be classified according to whether they assume the *Fixed Proximity Scenario* or whether they assume the *Variable Proximity Scenario*.

Fixed Proximity Scenario: In the fixed proximity scenario the input of the problem is a set of points and a definition of closeness between pairs of points; the output is a geometric graph such that two vertices are adjacent if and only if they satisfy the given definition of proximity. For example, a typical problem concerning β-skeletons in this scenario is as follows: Given a set P of n points and a real β , efficiently compute either the lune-based or the circle-based β-skeleton of P.

Variable Proximity Scenario: In this scenario it is not known a priori what closeness measure to use: the application context requires to consider several different definitions of closeness in order to choose the best suited one [14]. In this scenario, the input of the problem is a set of points and a set of definitions of closeness between pairs of points; the output is a set of geometric graphs describing the different definitions of proximity.

In the Fixed Proximity Scenario, optimal $O(n \log n)$ time algorithms are known for computing lune-based β skeletons when $\beta = 1$ and when $\beta = 2$ For $1 < \beta < 2$, an optimal $O(n \log n)$ time algorithm is described in [11]. To our knowledge, there exist only suboptimal algorithms that compute the lune-based β-skeleton when $0 \leq \beta < 1$: the fastest algorithm that we know for this problem requires $O(n^{2.5} \log n)$ time [15]. For values of β in the range $(2, \infty)$, a (suboptimal) $O(n^2)$ time algorithm for lune-based β-skeletons is described in [10,14]. As for circle-based β-skeletons, an optimal $O(n \log n)$ time algorithm is given in [10] for all values of β in the interval $[1, \infty]$, while the same $O(n^{2.5} \log n)$ time algorithm of [15] applies for values of β in the interval $[0, 1)$. The problem of computing γ-graphs has been studied in [18], where an $O(n^3)$ time algorithm

is described for the general case. Under the assumption that no four points are cocircular and that the chosen value of γ gives rise to a γ-graph with no edge crossings, an optimal $O(n \log n)$ time algorithm is also presented in [18].

In the Variable Proximity Scenario, a key problem is that of encoding the entire spectrum of the empty neighbourhoods that can be found in the point set. Given this information, it is easy to compute different proximity graphs with an output sensitive strategy. Suppose for example that one wants to compute all possible (circle-based or lune-based) β-skeletons for values of β in a given range. If the β_1-skeleton of P has already been computed and the next value of β to consider is $\beta_2 > \beta_1$, one would like to construct the β_2-skeleton by a sequence of edge deletions from the β_1-skeleton (since the β_2-neighbourhood contains the β_1-neighbourhood, it follows that the β_2-skeleton is a subgraph of the β_1-skeleton). A technique for efficiently solving this problem is based on a pre-processing step that computes for each pair u, v of a set of points P the largest value $\beta^*(u, v)$ of the parameter β such that u, v are adjacent in the $\beta^*(u, v)$-skeleton. The set of all these maximal values computed for each pairs of points of P is called the *β-spectrum* of the set of points. Once the β-spectrum of a set P of points is known, it is possible to scan all the β-skeletons of P by starting with the β skeleton with $\beta = 0$ and by repeatedly removing the edge with the next smallest $\beta^*(u, v)$ value. A first $O(n^3)$ time algorithm for computing the β-spectrum of a set of n points is given in [10]; the algorithm can be used both for the case that the β-neighbourhood is lune-based and the case that it is circle-based. Very recently, a new algorithm for the computation of the lune-based β-spectrum was presented in [16]. The algorithm in [16] requires $O(n^2 \log n + \rho)$ time, where ρ is a parameter that depends upon the geometry of the point set and it can be $O(n^3)$ for some problem instances (ρ is the size of the so called *witness set*; see the cited paper for more details).

The contribution of this paper is twofold. We present two simple algorithmic strategies that can be applied to solve a number of proximity problems both in the Fixed Proximity Scenario and in the Variable Proximity Scenario. The application of these techniques led to optimal algorithms for the computation of dense β-skeletons and γ-graphs in the Fixed Proximity Scenario and to breaking the $O(n^3)$ barrier for the computation of β-spectra in the Variable Proximity Scenario. It also led to the first robust algorithm for computing Gabriel graphs which uses only double precision arithmetic and requires $o(n^3)$ time. For a formal definition of the adopted model of computation which takes into account the arithmetic precision and for other results where this model is used see, e.g., [3, 4,5,1]. Our results can be listed as follows.

– We exhibit an $O(n^2)$-time algorithm for the computation of the (lune-based or circle-based) β-skeleton of a set of n points in the plane. This is worst case optimal when $0 \leq \beta < 1$ since for these values of β the β-skeleton can have $\Theta(n^2)$ edges. As described above, the previously known time bound for this problem is $O(n^{2.5} \log n)$ [15].

- We extend our technique to the computation of γ-graphs and present an optimal $O(n^2)$ time algorithm for the computation of this family of graphs in the Fixed Proximity Scenario. The previously known bound is $O(n^3)$ [18].
- We give an $O(n^2)$-time algorithm for the computation of the circle-based β-spectrum. Our algorithm is optimal, (the size of the β-spectrum is n^2) and improves over the previously known $O(n^3)$ time algorithm [10].
- We extend the technique of the previous item and obtain an optimal $O(n^2)$-time algorithm for the computation of the lune-based β-spectrum. This result improves over the $O(n^2 \log n + \rho)$-time algorithm in [16].
- As a further application of our technique we show the first algorithm that computes the Gabriel Graph of a set of n points in $O(n^2)$ time and requires only double precision integer arithmetic computations. We remark that the optimal time $O(n \log n)$-time algorithm described in the literature relies on the computation of the Delaunay triangulation which may require an arithmetic precision four times the one used to represent the input data [6].

For reasons of space, some proofs have been omitted in this extended abstract.

2 Preliminaries

In this section we introduce some notation and definitions that we use in subsequent sections. Let P be a set of points in the plane and let u and v be two points of P. The Euclidean distance between u and v is denoted by $d(u, v)$. The *arc* (u, v) is the directed line segment from u to v, and the *edge* (u, v) is the undirected line segment from u to v. Consider a circle of radius r through u and v and let C be one of the two circular arcs connecting u to v on on the circle. We now define rules for associating a real value with C and a real value with the pair u, v.

We associate with C a value that we call the *β-value* of C. If C is smaller than πr, we define the β value of C as $d(u, v)/(2r)$. If C is at least as large as πr, the β value of C is set equal to $2r/d(u, v)$. If $r = \infty$, then one of the two circular arcs connecting u to v coincides with the edge (u, v) and has β value equal to 0, while the other circular arc has β value equal to ∞.

For a given value of β let C_β be the circular arc of value β that lies to the right of the arc (u, v) and let D be the disk such that C_β is a portion of its circumference. We call the portion of D bounded by edge (u, v) and C_β the *right β-region* of u, v and denote is as $C_r(u, v, \beta)$. The region $C_r(u, v, \beta)$ is assumed to include the edge (u, v) but not C_β. For $\beta = \infty$, we define $C_r(u, v, \infty)$ as the open half plane to the left of the arc (u, v), plus the edge (u, v). Finally $C_r(u, v, 0)$ is the empty set. See Figure 1 for an illustration. Similarly we define the *left β-region* of u, v and denote it as $C_l(u, v, \beta)$. We say that the right or the left β-region of two points u, v in a point set P is *empty* if it does not contain any point of P other than u and v. For example, in Figure 1 both $C_l(u, v, \frac{1}{2})$ and $C_r(u, v, 1)$ are empty.

We associate with u, v a value that we call the *right β-value* of u, v and denote it as $\beta_r(u, v)$. The right β-value of u, v is the largest value of β such that

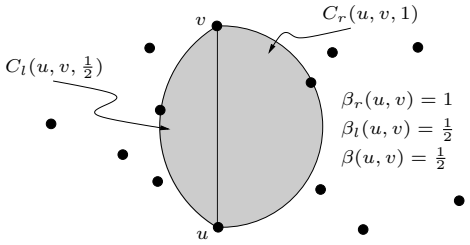


Fig. 1. The right β-value, the left β-value, and the β-value of two points u and v in a point set.

$C_r(u, v, \beta)$ is empty. Similarly we define the *left β-value* of u, v and denote it as $\beta_l(u, v)$. The *β-value* of u, v is the minimum of $\beta_l(u, v)$ and $\beta_r(u, v)$ and is denoted as $\beta(u, v)$. For example, the right β-value of u, v in Figure 1 is $\beta_r(u, v) = 1$, the left β-value is $\beta_l(u, v) = \frac{1}{2}$, and the β-value is $\beta = \frac{1}{2}$.

Property 1. Let u, v be a pair of points in a point set P; we have $\beta_r(u, v) = \beta_l(v, u)$ and $C_r(u, v, \beta) = C_l(v, u, \beta)$.

The *β-spectrum* of a set P of points is the set of all pairs of points of P where each pair is labeled with its β-value. The *circle-based β-neighbourhood* [10] of two points u and v is equal to $C_l(u, v, \beta) \cup C_r(u, v, \beta)$. So the *circle-based β skeleton* of P is a proximity graph such that an edge (u, v) belongs to the graph if and only if $C_l(u, v, \beta) \cup C_r(u, v, \beta)$ is empty. The following property relates β-skeletons to right and left β-values.

Property 2. Let P be a set of points in the plane, let u, v be a pair of points of P, and let β be a real positive value such that $0 \leq \beta \leq \infty$. The β-region of u and v is empty if and only if the right and the left β-values of u, v are such that $\beta_r(u, v) \geq \beta$ and $\beta_l(u, v) \geq \beta$.

If for example we set $\beta = \frac{1}{2}$ and look at points u and v in Figure 1, we can conclude by Property 2 that (u, v) is an edge of the β-skeleton; if we set $\beta > \frac{1}{2}$, then u and v are not adjacent in the β-skeleton of the set of points.

For $0 \leq \beta \leq 1$, the *lune-based β neighbourhood* is the same as the circle-based β neighbourhood. For $\beta > 1$, the lune-based β-neighbourhood of points u and v is the intersection of the two disks through u and v of radius $\beta d(u, v)/2$.

In [18] Veltkamp introduces *γ-graphs* as a parametrized family of proximity graphs which include circle-based β-skeletons as special cases and are defined in terms of two *γ-parameters*, named γ_0 and γ_1. The γ-graph of P is also denoted as the $[\gamma_0, \gamma_1]$-graph of P to explicitly mention the two γ-parameters. Let γ_0 and γ_1 be two real values such that $-1 \leq |\gamma_0| \leq |\gamma_1| \leq 1$ and let u, v be a pair of points of P. The two values γ_0 and γ_1 define two *$[\gamma_0, \gamma_1]$-neighbourhoods* of u, v. The $[\gamma_0, \gamma_1]$-graph of P is a proximity graph such that any two vertices u, v are adjacent if and only if at least one of the two $[\gamma_0, \gamma_1]$-neighbourhoods

of u, v is empty. The two $[\gamma_0, \gamma_1]$-neighbourhoods of u and v can be defined in terms of left and right β-regions: the $[\gamma_0, \gamma_1]$-neighbourhoods of u and v are $C_l(u, v, \beta_0) \cup C_r(u, v, \beta_1)$ and $C_r(u, v, \beta_0) \cup C_l(u, v, \beta_1)$, where the correspondence between the pair (β_0, β_1) and (γ_0, γ_1) is as follows. If $\gamma_i \geq 0$ then $\beta_i = 1/(1-\gamma_i)$ and if $\gamma_i \leq 0$ then $\beta_i = 1 + \gamma_i$ for $i = 0, 1$.

3 Algorithms for the Fixed Proximity Scenario

In this section we first present a simple algorithm for the following problem. Let P be a set of points in the plane and let β be a given positive real number. For each pair of points u, v of P we ask ourselves whether or not the right β-region of u, v contains an element of P. We call this problem the *right β-region problem.* We shall exploit our solution to the right β-region problem to efficiently compute β-skeletons, γ-graphs, and Gabriel graphs with low arithmetic degree.

3.1 The Right β-Region Problem

Our algorithm for the right β-region problem assigns each arc (u, v) a label according to whether the right β-region of u, v is empty or not. The arc (u, v) is labeled *Yes* if $C_r(u, v, \beta)$ contains no points of P, it is labeled *No* otherwise. A technique similar to ours has been used by Beirouti and Snoeyink [2] for computing triangles emptiness in the context of LMT heuristics for minimum weight triangulations. A precise description of our algorithm is as follows.

`Algorithm Right` β`-region`

`Step 1:` For each point $u \in P$ compute an ordered list of the remaining points, sorted in radial order around u in clock-wise direction. Do step 2 for each point $u \in P$.

`Step 2:` Let p_0 be the point closest to u. Let $p_0, p_1, \ldots, p_{n-2}$ be the ordered sequence of points of $P - \{u\}$ radially sorted around u in clock-wise direction. For $i = 0, 1, 2, \ldots, n-2$ do: (arithmetic in indices is done $\pmod{n-1}$):

`Step 2.1:`
- If $p_{i+1} \notin C_r(u, p_i, \beta)$ then label the arc (u, p_i) with a label *Maybe*.
- If $p_{i+1} \in C_r(u, p_i, \beta)$ then do the following:
  1. Label (u, p_i) with a label *No*;
  2. Set $k = i$
  3. Determine j such that $j < k$ and such that j is the largest index for which (u, p_j) has the label *Maybe*. If $j > 0$ and if $p_{i+1} \in C_r(u, p_j, \beta)$ then label (u, p_j) with *No*, set $k = j$ and repeat this instruction, else this instruction is done.

`Step 2.2:` Scan again $p_0, p_1, \ldots, p_{n-2}$; for each arc (u, p_i) $(0 \leq i \leq n-2)$ labeled *Maybe* change its label to *Yes*

`End of Algorithm Right` β`-region`

The following lemmas proof the correctness of the above algorithm and show that its time complexity is $O(n^2)$. The proof of the first lemma follows from elementary geometry.

Lemma 1. *If $i < j$ and the angle between arcs (u, p_i) and (u, p_j) is less than π and if $p_j \notin C_r(u, p_i, \beta)$ then $C_r(u, p_i, \beta) \cap C_r(u, p_j, \infty) \subset C_r(u, p_j, \beta)$.*

Lemma 2. *After* `Algorithm Right` β`-region` *has performed* `Step 2.1` *for a particular value of i, we have that for each value of h with $0 \leq h \leq i$ for which arc (u, p_h) is labeled* Maybe*, $C_r(u, p_h, \beta)$ does not contain any points of $\{p_{h+1}, p_{h+2}, \ldots, p_{i+1}\}$.*

Theorem 1. *Let P be a set of n points in the plane.* `Algorithm Right` β`-region` *solves the Right β-region Problem for P in $O(n^2)$ time.*

Proof. Correctness follows from Lemma 2 and from the fact that since p_0 is the point closest to u, $C_r(u, p_{n-2}, \beta)$ is non-empty if and only if $p_0 \in C_r(u, p_{n-2}, \beta)$. The radial sorting of `Step 1` can be done in $O(n^2)$ time by the algorithm of [13] or the alternative algorithm in [7]. If for each point p_i we maintain a pointer to the last encountered arc with a *Maybe* label, `Step 2.1` will require $O(n)$ time. `Step 2.2` trivially requires $O(n)$ time. Therefore, `Step 2` requires $O(n)$ time and since it is executed n times, `Algorithm Right` β`-region` has an $O(n^2)$ time complexity.

3.2 Computing Proximity Graphs

We will now use `Algorithm Right` β`-region` to compute proximity graphs. We first consider β-skeletons for values of β such that $0 \leq \beta < 1$. In this interval the β-neighbourhood is the same for lune-based and circle-based graphs; therefore in the statement of the next theorem we do not distinguish between lune-based and circle-based β-skeletons.

Theorem 2. *Let P be a set of n points in the plane and let $0 \leq \beta < 1$. There exists an algorithm that computes the circle-based and the lune-based β-skeleton of P in optimal $O(n^2)$ time.*

Proof. If $\beta = 0$, the β-skeleton of P is the complete graph and can be easily computed in $O(n^2)$ time by connecting all pairs of points. If $\beta > 0$, we compute the β-skeleton of P as follows. We first execute `Algorithm Right` β`-region` on P. Note that the definition of right β-region of u, v implicitly contains a notion of direction from u to v and that by Property 1 $C_r(u, v, \beta) = C_l(v, u, \beta)$. Then we compute all edges (u, v) of the β-skeleton by checking if both the arc (u, v) and the arc (v, u) have been labeled *Yes* by `Algorithm Right` β`-region`. The correctness of the algorithm is a consequence of Property 2 and Theorem 1. The bound on the time complexity is a consequence of Theorem 1.

Theorem 3. *Let P be a set of n points in the plane and let γ_0, γ_1 be a pair of real values such that $-1 \leq |\gamma_0| \leq |\gamma_1| \leq 1$. There exists an algorithm that computes the $[\gamma_0, \gamma_1]$-graph of P in optimal $O(n^2)$ time.*

Proof. We compute the $[\gamma_0, \gamma_1]$-graph of P by the following procedure. First the value of β_0 and the value of β_1 corresponding to γ_0 and γ_1 are computed (See Section 2). We execute `Algorithm Right` β`-region` on P twice: once for $\beta = \beta_0$ and a second time for $\beta = \beta_1$. A pair u, v of points is connected by an edge in the $[\gamma_0, \gamma_1]$-graph if one of the following two events happens: (i) The arc (u, v) has been labeled *Yes* by `Algorithm Right` β`-region` when $\beta = \beta_0$ and (v, u) has been labeled *Yes* when $\beta = \beta_1$; or (ii) The arc (u, v) has been labeled *Yes* by `Algorithm Right` β`-region` when $\beta = \beta_1$ and (v, u) has been labeled *Yes* when $\beta = \beta_0$. The bound on the time complexity is a consequence of Theorem 1.

As a further application of `Algorithm Right` β`-region`, we observe that it can also be used to compute circle-based β-skeletons for values of β such that $\beta \geq 1$. When $\beta = 1$ and when the β neighbourhood is a closed set, the β-skeleton of P coincides with the Gabriel graph of P. It can be shown that `Algorithm Right` β`-region` can be slightly modified to compute the Gabriel graph of a set of points in $O(n^2)$ time. This result is not very surprising on its own since an optimal $O(n \log n)$ algorithm for Gabriel graphs (and also for every circle-based β-skeleton with $\beta \geq 1$) is known [8,10]. However, the status of affairs changes if we revisit existing algorithms in terms of their robustness.

The optimal algorithm for Gabriel graphs is based on first computing the Delaunay triangulation and then deleting the edges that are not Gabriel edges. The implementation of a numerically robust code for Delaunay triangulation requires to evaluate the sign of irreducible polynomials of algebraic degree 4 (see, e.g. [6]), thus requiring a numerical precision four times the one required for representing the input data. On the other hand, it is a trivial task to compute the Gabriel graph of a set of points by simply comparing Euclidean distances in $O(n^3)$ time. In this case, the algebraic degree of the polynomials to evaluate is only 2, which can be shown to be optimal. Therefore, we can identify a trade-off between time-complexity and numerical precision required by algorithms that compute Gabriel graphs. In the next theorem we show how to reduce the trade-off.

We adopt *degree model* of computation introduced in [1,3] and analyze the performance of `Algorithm Right` β`-region` in terms of required numerical precision to compute Gabriel graphs. In this model of computation the robustness of a geometric algorithm is evaluated by looking at the irreducible polynomial of highest algebraic degree whose sign is evaluated by the algorithm during its execution. This quantity is called the *degree* of the algorithm and is a measure of the numerical precision that a robust implementation would require to guarantee correct outputs independent of degenerate configurations of the input data. By analyzing the algebraic degree of the geometric tests in the elegant and simple algorithm by Overmars and Welzl [13] we can conclude the following.

Lemma 3. *Let P be a set of n points in the plane. The algorithm by Overmars and Welzl [13] solves the problem of reporting for each $u \in P$ all points of $P - \{u\}$ radially sorted around u in $O(n^2)$ time and degree 2.*

When $\beta = 1$, the geometric tests performed by `Step 2` of `Algorithm Right` β`-region` take into account triplets of input points, say p, q, r, compute the centre c of the disk having p, q as antipodal points, and compare $d(p, c)$ with $d(r, c)$. This corresponds to comparing numbers of at most $2(b + 1)$ bits where b bits is the input precision. We can summarize this discussion as follows.

Theorem 4. *Let P be a set of n points in the plane. There exists an optimal degree 2 algorithm that computes the Gabriel graph of P in $O(n^2)$ time.*

4 Algorithms for the Variable Proximity Scenario

In this section we consider the problem of computing the β-spectrum of a set of points. Before we present the algorithm for computing the β-spectrum, we introduce some new notation and a few lemmas. After that we are in a position to present the algorithm and its proof of correctness.

Let u, v and w be three points in the plane and let $C(u, v, w)$ be the disk through u, v and w. We denote with $C'(u, v, w)$ the subset of $C(u, v, w)$ bounded by chord (u, w) and the circular arc from u, through v ending at w.

Let p be an arbitrary point and let C be a circle through p. Let l_0 be a line through p and tangent to C. Assume that l_0 is horizontal and that C lies above l_0. Let $B = \{p_0, p_1, p_2, \ldots p_k, p_{k+1}\}$ be a set of points that lie above l_0. Assume that points $p_0, p_1, \ldots, p_k$ are counter clock-wise radially sorted around p. Moreover assume that p_0 lies on C and that the remaining points of B lie on or outside C. Let l_i be the line tangent to $C(p, p_{i-1}, p_i)$ through p. Let α_i be the top right angle between l_0 and l_i. We say that $\{p_0, p_1, p_2, \ldots p_k\}$ is an *increasing set* if $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_k$. An increasing set of points is depicted in Figure 2.

In Lemmas 4 - 7 we denote with l the line through p and p_k and with a_i the highest intersection point between l and circle $C(p, p_{i-1}, p_i)$. The highest intersection point of C and l is called a_0. Also, for any two points r and s on line l we say that $r < s$ if $d(p, r) < d(p, s)$. Notice that if $\{p_0, p_1, p_2, \ldots, p_k\}$ is an increasing set of points, we have $a_0 \leq a_1 \leq \ldots \leq a_k$.

Lemma 4. *Let $P = \{p, p_0, p_1, \ldots, p_k\}$ be a set of points such that $\{p_0, p_1, \ldots, p_{k-1}\}$ is an increasing set. Let i be the largest index such that $C_r(p, p_k, \beta_r(p, p_k)) = C'(p, p_i, p_k)$. Then $p_{j-1} \in C'(p, p_j, p_k)$ for $i < j < k$ and $p_{j+1} \notin C'(p, p_j, p_k)$ for $i < j < k - 1$.*

Lemma 5. *Let $P = \{p, p_0, p_1, \ldots, p_k\}$ be a set of points such that $\{p_0, p_1, \ldots, p_{k-1}\}$ is an increasing set. Let i be the smallest index for which $C_r(p, p_k, \beta_r(p, p_k)) = C'(p, p_i, p_k)$. Then $p_{j+1} \in C'(p, p_j, p_k)$ and $p_{j-1} \notin C'(p, p_j, p_k)$ for $0 \leq j < i$.*

Lemma 6. *Let $P = \{p, p_0, p_1, \ldots, p_k\}$ be a set of points such that $\{p_0, p_1, \ldots, p_{k-1}\}$ is an increasing set. If $C_r(p, p_k, \beta_r(p, p_k)) = C'(p, p_i, p_k)$ then $\{p_0, p_1, \ldots, p_i, p_k\}$ is an increasing set of points.*

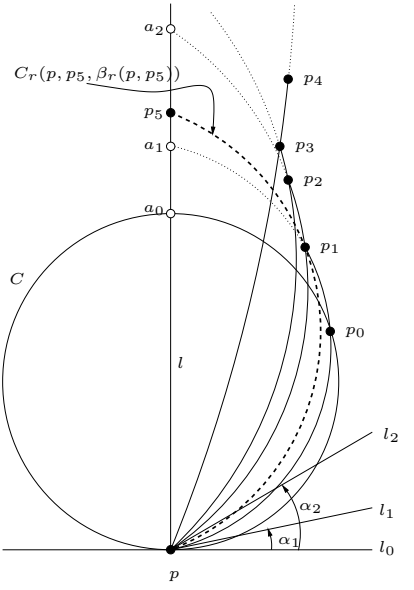


Fig. 2. Increasing set $\{p_0, p_1, \ldots, p_4\}$.

Lemma 7. *Let $P = \{p, p_0, p_1, \ldots, p_k, p_{k+1}\}$ be a set of points such that $\{p_0, p_1, \ldots, p_{k-1}\}$ is an increasing set. If $C_r(p, p_k, \beta_r(p, p_k)) = C'(p, p_i, p_k)$ then $\{p_i, p_k\} \cap C'(p, p_j, p_{k+1}) \neq \emptyset$ for $i < j < k$.*

We are now in a position to describe our algorithm for computing the cricle-based β-spectrum of a set of points. We start with an algorithm for an easier problem that we call the *right β-value problem for a point p* . Let $P = \{p, p_0, p_1, \ldots, p_{n-1}\}$ be a set of points in the plane, let C be a circle through p and p_0 and let l be the line through p tangent to C. Assume that all points in $\{p_0, p_1, \ldots, p_{n-1}\}$ lie outside or on C, and on the same side of l as C does. Also assume that points $p_0, p_1, \ldots, p_{n-1}$ are radially sorted around p in counter clock-wise direction. The right β-value problem for p is to compute the right β-values $\beta_r(p, p_i)$ for $1 \leq i \leq n-1$.

`Algorithm Right` β`-values`

`Step 1:` Set $B = \{p_0, p_1\}$; compute and record $\beta_r(p, p_1)$.
`Step 2:` For $k = 2, 3, \ldots, (n-1)$ do the following:
  `Step 2.1:` Compute point $p_i \in B$ such that $C_r(p, p_k, \beta_r(p, p_k)) = C'(p, p_i, p_k)$.
  `Step 2.2:` Record $\beta_r(p, p_k)$.
  `Step 2.3:` Set $B = \{p_0, p_1, \ldots, p_i, p_k\}$.

`End of Algorithm Right` β`-values`

Lemma 8. *Let $P = \{p, p_0, p_1, \ldots, p_{n-1}\}$ be a set of points in the plane as described above.* `Algorithm Right` β`-values` *solves the right β-value problem in $O(n)$ time.*

Proof. The correctness of `Algorithm Right` β`-values` follows from the following observations. Initially the set B is an increasing set. Lemma 6 shows that after `Step 2.3` set B is still increasing. Moreover, the elements of P that are not in B are not relevant when we compute $\beta_r(p, p_k)$ in `Step 2.1`, as it is shown in Lemma 7. Finally, `Step 2.1` can be performed with a sequential scan of the elements of B. Lemmas 4 and 5 show that the amortized cost of `Step 2` is $O(n)$.

The next lemma shows how to compute the right β-values for all pairs of points in a point set.

Lemma 9. *Let P be a set of n points in the plane. There exists an algorithm that computes the right β-value for each pair of points of P in optimal $O(n^2)$ time.*

Proof. The set of right β-values is computed as follows. We first construct the Delaunay triangulation of P [10]. Let (a, b, c) be a triangle in this triangulation. Without loss of generality assume that (a, b, c) is the clock-wise order of these points around the triangle. Since $C(a, b, c)$ contains no points from P in it interior, we can immediately compute $\beta_r(a, b)$, $\beta_r(b, c)$ and $\beta_r(c, a)$. Secondly we radially sort $P - \{p\}$ around p for each point p of P [13]. Let $\{p_1, p_2, \ldots, p_m\}$ be the set of points in $C_l(b, a, \infty) \cap C_r(b, c, \infty)$, radially sorted in counter clock-wise order around b. We execute `Algorithm Right` β`-values` to obtain $\beta_r(b, p_i)$ for $1 \leq i \leq m$. Repeating this for all three corners of all Delaunay triangles gives us the $\beta_r(u, v)$ for all pairs of points (u, v) of P. The correctness of the algorithm follows from the correctness of `Algorithm Right` β`-values` (Lemma 8).

Concerning the time complexity of our algorithm, we observe that computing the Delaunay triangulation of P and the radially sorted lists can be done in $O(n^2)$ time; since `Algorithm Right` β`-values` is executed $O(n)$ times and since each execution requires $O(n)$ time by Lemma 8, it follows that the computational cost of the algorithm is $O(n^2)$. Since there are n^2 pairs of points there are n^2 right β-values to compute and the time complexity of the algorithm is asymptotically optimal.

The results above easily imply the main result of this section.

Theorem 5. *Let P be a set of n points in the plane. There exists an algorithm that computes the circle-based β-spectrum of P in optimal $O(n^2)$ time.*

We can extend the results of Theorem 5 to compute the lune-based β-spectrum of a set of points.

Theorem 6. *Let P be a set of n points in the plane. There exists an algorithm that computes the lune-based β-spectrum of P in optimal $O(n^2)$ time.*

Acknowledgments. We are most grateful to Gunter Rote for his useful remarks that led us to refine the analysis on the time complexity of `Algorithm Right` β-`value`.

References

1. Robust proximity queries: An illustration of degree-driven algo rithm design. *SIAM J. Comput.*, 28(3):864–889, 1998.
2. R. Beirouti and J. Snoeyink. Implementations of the LMT heuristic for minimum weight triangulation. In *ACM Symposium on Computational Geometry*, pages 96–105, 1998.
3. J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. *SIAM J. Comput.*, (5):1401–1421, 2000.
4. J.-D. Boissonnat and J. Snoeyink. Efficient algorithms for line and curve segment intersection using restricted predicates. *Comput. Geom. Theory Appl.*, 16(1):35–52, 2000.
5. T. Chan. Reporting curve segment intersections using restricted predicates. *Comput. Geom. Theory Appl.*, 16(4), 2000.
6. F. D'Amore, P. Franciosa, and G. Liotta. A robust region approach to the computation of geometric graphs. In *Algortihms–ESA '98*, volume 1461 of *Lecture Notes Comput. Sci.*, pages 175–186, 1998.
7. H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.
8. K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.
9. J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proc. IEEE*, 80(9):1502–1517, Sept. 1992.
10. D. G. Kirkpatrick and J. D. Radke. A framework for computational morphology. In G. T. Toussaint, editor, *Computational Geometry*, pages 217–248. North-Holland, Amsterdam, Netherlands, 1985.
11. A. Lingas. A linear-time construction of the relative neighborhood graph f rom the Delaunay triangulation. *Comput. Geom. Theory Appl.*, pages 199–208, 1994.
12. D. W. Matula and R. R. Sokal. Properties of Gabriel graphs relevant to geographic variation research and clustering of points in the plane. *Geogr. Anal.*, 12(3):205–222, 1980.
13. M. H. Overmars and E. Welzl. New methods for computing visibility graphs. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 164–171, 1988.
14. J. D. Radke. On the shape of a set of points. In G. T. Toussaint, editor, *Computational Morphology*, pages 105–136. North-Holland, Amsterdam, Netherlands, 1988.
15. S. V. Rao and A. Mukhopadhyay. Fast algortihm for computing β-skeletons and their relatives. In *Proc. 8th Annual Int. Symp. on Algorithms and Computation*, volume 1350 of *Lecture Notes Comput. Sci.*, pages 374–383, 1997.
16. S. V. Rao and A. Mukhopadhyay. Efficient algortihms for computing the β-spectrum. In *Proc. 12th Canad. Conf. Comput. Geom.*, pages 91–97, 2000.
17. K. J. Supowit. The relative neighborhood graph with an application to minimum spanning trees. *J. ACM*, 30(3):428–448, 1983.
18. R. C. Veltkamp. The γ-neighborhood graph. *Comput. Geom. Theory Appl.*, 1(4):227–246, 1992.

Optimal Möbius Transformations for Information Visualization and Meshing

Marshall Bern[1] and David Eppstein[2]

[1] Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304, USA
bern@parc.xerox.com
[2] Dept. Inf. & Comp. Sci., UC Irvine, CA 92697-3425, USA
eppstein@ics.uci.edu

Abstract. We give linear-time quasiconvex programming algorithms for finding a Möbius transformation of a set of spheres in a unit ball or on the surface of a unit sphere that maximizes the minimum size of a transformed sphere. We can also use similar methods to maximize the minimum distance among a set of pairs of input points. We apply these results to vertex separation and symmetry display in spherical graph drawing, viewpoint selection in hyperbolic browsing, element size control in conformal structured mesh generation, and brain flat mapping.

1 Introduction

Möbius transformations of d-dimensional space form one of the fundamental geometric groups. Generated by inversions of spheres, they preserve spherical shape as well as the angles between pairs of curves or surfaces. We consider here problems of finding an *optimal Möbius transformation*:

- Given a set of $(d-1)$-dimensional spheres in a d-dimensional unit ball, find a Möbius transformation that maps the ball to itself and maximizes the minimum radius among the transformed spheres.
- Given a set of $(d-1)$-dimensional spheres on a sphere $\mathbb{S}^d$, find a Möbius transformation of $\mathbb{S}^d$ that maximizes the minimum radius among the transformed spheres.
- Given a graph connecting a set of vertices on $\mathbb{S}^d$ or in the unit ball in $\mathbb{E}^d$, find a Möbius transformation that maximizes the minimum edge length.

We develop efficient algorithms for these problems, by formulating them as quasiconvex programs in hyperbolic space. The same formulation also shows that simple hill-climbing methods are guaranteed to find the global optimum; this approach is likely to work well in practice as an alternative to our more complicated quasiconvex programs. We apply these results to the following areas:

- Spherical graph drawing [15]. Any embedded planar graph can be represented as a collection of tangent circles on a sphere $\mathbb{S}^2$; this representation

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 14–25, 2001.

is unique for maximal planar graphs, up to Möbius transformation. Our algorithms can find a canonical spherical realization of any planar graph that optimizes the minimum circle radius or the minimum separation between two vertices, and that realizes any symmetries implicit in the given embedding.
- Hyperbolic browsing [16]. The Poincaré model of the hyperbolic plane has become popular as a way of displaying web sites and other graph models too complex to view in their entirety. This model permits parts of the site structure to be viewed in detail, while reducing the size of peripheral parts. Our algorithms can find a "central" initial viewpoint for a hyperbolic browser, that allows all parts of the sites to be viewed at an optimal level of detail.
- Mesh generation [3, 24]. A principled method of structured mesh generation involves conformal mapping of the problem domain to a simple standardized shape such as a disk, construction of a uniform mesh in the disk, and then inverting the conformal mapping to produce mesh elements in the original domain. Möbius transformations can be viewed as a special class of conformal maps that take the disk to itself. Our algorithms can find a conformal mesh that meets given requirements of element size in different portions of the input domain, while using a minimal number of elements in the overall mesh.
- Brain flat mapping [13]. Hurdal et al. proposed a system for visualizing convoluted brain surfaces, by approximate conformal mapping of those surfaces to a Euclidean disk, sphere, or hyperbolic plane. Our algorithms can find a conformal mapping that minimizes the resulting areal distortion.

We assume throughout that the dimension d of the spaces we deal with is a constant; most commonly in our applications, $d = 2$ or $d = 3$. We omit many details in this extended abstract; see the full paper [2] for details.

2 Preliminaries

2.1 Möbius Transformation and Hyperbolic Geometry

An *inversion* of the set $\mathbb{R}^d \cup \{\infty\}$, generated by a sphere C with radius r, maps to itself every ray that originates at the sphere's center. Each point is mapped to another point along the same ray, so that the product of the distances from the center to the point and to its image equals r^2. The center of C is mapped to ∞ and vice versa. An inversion maps each point of C to itself, transforms spheres to other spheres, and preserves angles between pairs of curves or surfaces. Repeating an inversion produces the identity mapping on $\mathbb{R}^d \cup \{\infty\}$. The set of products of inversions forms a group, the group of *Möbius transformations* on the Euclidean space $\mathbb{E}^d$. If we restrict our attention to the subgroup that maps a given sphere $\mathbb{S}^{d-1}$ to itself, we find the group of Möbius transformations on $\mathbb{S}^{d-1}$.

Although our problem statements involve Euclidean and spherical geometry, our solutions involve techniques from hyperbolic geometry [14], and in particular the classical methods of embedding hyperbolic space into Euclidean space: the *Poincaré model* and the *Klein model*.

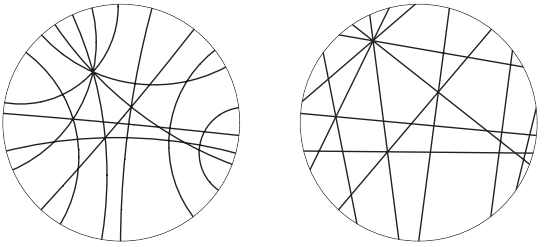


Fig. 1. Poincaré (left) and Klein (right) models of the hyperbolic plane. Analogous models exist for any higher dimensional hyperbolic space.

In both the Poincaré and Klein models, the d-dimensional hyperbolic space $\mathbb{H}^d$ is viewed as an open unit ball in a Euclidean space $\mathbb{E}^d$, while the unit sphere bounding the ball forms a set of points "at infinity". In the Poincaré model (Figure 1, left), the lines of the hyperbolic space are modeled by circular arcs of the Euclidean space, perpendicular to the unit sphere. Hyperplanes are modeled by spheres perpendicular to the unit sphere, and hyperbolic spheres are modeled by spheres fully contained within the unit ball. Two more classes of surfaces are also modeled as Euclidean spheres: *hyperspheres* (surfaces at constant distance from a hyperplane) are modeled by spheres crossing the unit sphere nonperpendicularly, and *horospheres* are modeled by spheres tangent to the unit sphere. The Poincaré model thus preserves spherical shape as well as the angles between pairs of curves or surfaces. Any horosphere or hypersphere divides $\mathbb{H}^d$ into a convex and a nonconvex region; we define a *horoball* or *hyperball* to be the convex region bounded by a horosphere or hypersphere respectively.

In the Klein model (Figure 1, right), the lines of the hyperbolic space map to line segments of the Euclidean space, formed by intersecting Euclidean lines with the unit ball. Hyperplanes thus map to hyperplanes, and convex bodies map to convex bodies. Although the Klein model does not preserve spherical shape, it does preserve flatness and convexity. In the Klein model, spheres are modeled as ellipsoids contained in the unit ball, horospheres are modeled as ellipsoids tangent at one point to the unit ball, and hyperspheres are modeled as halves of ellipsoids tangent in a $(d-1)$-sphere to the unit ball.

The Poincaré and Klein models are not intrinsic to hyperbolic space, rather there can be many such models for the same space. The choice of model is determined by the hyperbolic point mapped to the model's center point, and by an orientation of the space around this center. We call this central point the *viewpoint* since it determines the Euclidean view of the hyperbolic space.

The connection between hyperbolic space and Möbius transformations is this: the isometries of the hyperbolic space are in one-to-one correspondence with the subset of Möbius transformations of the unit ball that map the unit ball to itself, where the correspondence is given by the Poincaré model [14, Theorem 6.3]. Any hyperbolic isometry (or unit-ball preserving Möbius transformation) can be factored into a hyperbolic translation mapping some point of the hyperbolic space

to the viewpoint, followed by a rotation around the viewpoint [14, Lemma 6.4]. Since rotation does not change the Euclidean shape of the objects on the model, our problems of selecting an optimal Möbius transformation of a Euclidean or spherical space can be rephrased as finding an optimal hyperbolic viewpoint.

2.2 Quasiconvex Programming

The viewpoint we seek in our optimal Möbius transformation problems will be expressed as the pointwise maximum of a finite set of *quasiconvex functions*; that is, functions for which the level sets are all convex. To find this point, we use a generalized linear programming framework of Amenta et al. [1] called *quasiconvex programming*.

Define a *nested convex family* to be a map $\kappa(t)$ from nonnegative real numbers to compact convex sets in $\mathbb{E}^d$ such that if $t < t'$ then $\kappa(t) \subset \kappa(t')$, and such that for all t, $\kappa(t) = \bigcap_{t'>t} \kappa(t')$. Any nested convex family κ determines a function $f_\kappa(x) = \inf\{\, t \mid x \in \kappa(t) \,\}$ on $\mathbb{R}^d$, with level sets that are the boundaries of $\kappa(t)$. If f_κ does not take a constant value on any open set, and if $\kappa(t')$ is contained in the interior of $\kappa(t)$ for any $t' < t$, we say that κ is *continuously shrinking*. Conversely, the level sets of any quasiconvex function form the boundaries of the convex sets in a nested convex family, and if the function is continuous and not constant on any open set then the family will be continuously shrinking.

If $S = \{\kappa_1, \kappa_2, \ldots \kappa_n\}$ is a set of nested convex families, and $A \subset S$, let

$$f(A) = \inf\left\{ (t, x) \;\middle|\; x \in \bigcap_{\kappa_i \in A} \kappa_i(t) \right\}$$

where the infimum is taken in the lexicographic ordering, first by t and then by the coordinates of x. Amenta et al. [1] defined a *quasiconvex program* to be a finite set S of nested convex families, with the objective function f described above, and showed that generalized linear programming techniques could be used to solve such programs in linear time for any constant dimension. Due to the convexity-preserving properties of the Klein model, we can replace $\mathbb{E}^d$ by $\mathbb{H}^d$ in the definition of a nested convex family without changing the above result.

3 Algorithms

We now describe how to apply the quasiconvex programming framework described above in order to solve our optimal Möbius transformation problems. In each case, we form a set of nested convex families κ_i, where each family corresponds to a function f_i describing the size of one of the objects in the problem (e.g., the transformed radius of a sphere) as a function of the viewpoint location. The solution to the resulting quasiconvex program then gives the viewpoint maximizing the minimum of these function values. The only remaining question in applying this technique is to show that, for each of the problems we study, the functions of interest do indeed have convex level sets.

3.1 Maximizing the Minimum Sphere

We begin with the simplest of the problems described in the introduction: finding a Möbius transformation that takes the unit ball to itself and maximizes the minimum radius among a set of transformed spheres. Equivalently, we are given a set of spheres in a hyperbolic space, and wish to choose a viewpoint for a Poincaré model of the space that maximizes the minimum Euclidean radius of the spheres in the model. By symmetry, the radius of a sphere in the Poincaré model depends only on its hyperbolic radius, and on the hyperbolic distance from its center to the viewpoint. Thus, in this case, the level sets of the transformed radius are just concentric hyperbolic spheres.

Theorem 1. *Suppose we are given as input a set of n spheres, all contained in the unit ball in $\mathbb{E}^d$. Then we can find the Möbius transformation of the unit ball that maximizes the minimum radius of the transformed spheres, in $\mathcal{O}(n)$ time, by quasiconvex programming.*

If we view the unit sphere itself as being one of the input spheres, then Theorem 1 can be viewed as minimizing the ratio between the radii of the largest and smallest transformed spheres.

Open Problem 1. *Is there an efficient algorithm for finding a Möbius transformation of $\mathbb{E}^d$ minimizing the ratio between the radii of the largest and smallest transformed spheres, when the input does not necessarily include one sphere that contains all the others?*

We can also prove a similar result for radius optimization on the sphere:

Theorem 2. *Suppose we are given as input a set of n spheres in $\mathbb{S}^d$. Then we can find the Möbius transformation of $\mathbb{S}^d$ that maximizes the minimum radius of the transformed spheres, in $\mathcal{O}(n)$ time, by quasiconvex programming.*

3.2 Vertex Separation

We next consider problems of using Möbius transformations to separate a collection of points.

Theorem 3. *Suppose we are given as input a graph with n vertices and m edges, with each vertex assigned to a point on the sphere $\mathbb{S}^d$ or the unit disk in $\mathbb{R}^d$, and with edges represented as great circle arcs or straight line segments respectively. Then we can find the Möbius transformation that maximizes the minimum length of the transformed graph edges, in $\mathcal{O}(m)$ time, by quasiconvex programming.*

Similarly, we can find the Möbius transformation maximizing the minimum distance among a set of n transformed points by applying Theorem 3 to the complete graph K_n. However, the input size in this case is n, while the algorithm of Theorem 3 takes time proportional to the number of edges in K_n, $\mathcal{O}(n^2)$. With care we can reduce the time to near-linear:

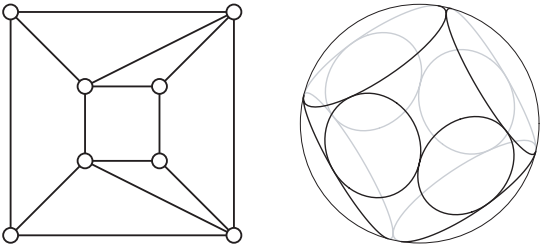


Fig. 2. A planar graph (left) and its coin graph representation (right).

Theorem 4. *Suppose we are given n points in $\mathbb{S}^2$ or the unit disk in $\mathbb{R}^2$. Then we can find the Möbius transformation that maximizes the minimum distance among the transformed points in $\mathcal{O}(n \log n)$ time.*

Proof. The Delaunay triangulation of the points can be computed in $\mathcal{O}(n \log n)$ time, is Möbius-invariant (due to its definition in terms of empty circles), forms a planar graph with $\mathcal{O}(n)$ edges, and is guaranteed to contain the shortest transformed distance among the points. Therefore, applying Theorem 3 to the Delaunay triangulation gives the desired result. □

In higher dimensions, the Delaunay triangulation may be complete, and so gives us no advantage. However we can again reduce the time from quadratic by using a random sampling scheme similar to one from our work on inverse parametric optimization problems [9].

Theorem 5. *Suppose we are given n points in $\mathbb{S}^d$. Then we can find the Möbius transformation that maximizes the minimum distance among the transformed points in randomized expected time $\mathcal{O}(n \log n)$.*

Open Problem 2. *Is there an efficient deterministic algorithm for maximizing the minimum distance among n points in $\mathbb{S}^d$, $d \geq 3$?*

4 Applications

4.1 Spherical Graph Drawing

As is by now well known, any planar graph can be represented by a set of disjoint circles in $\mathbb{S}^2$, such that two vertices are adjacent exactly when the corresponding two circles are tangent [6, 15, 21]. We call such a representation a *coin graph*; Figure 2 shows an example. Although it seems difficult to represent the positions of the coins exactly, fast algorithms for computing numerical approximations to their positions are known [7, 18, 22]. By polar projection, we can transform any coin graph representation on the sphere to one in the plane or vice versa. See Ken Stephenson's web site http://www.math.utk.edu/~kens/ for more information, including software for constructing coin graph representations and a bibliography of circle packing papers.

It is natural to ask for the planar or spherical coin graph representation in which all circles are most nearly the same size. It is NP-hard to determine whether a planar coin graph representation exists in which all circles are equal, or in which the ratio between the maximum and minimum radius satisfies a given bound [5]. However, if the graph is maximal planar, its coin graph representation is unique up to Möbius transformation, and we can apply Theorem 2 to find the optimal spherical coin graph representation. There is also a natural way of obtaining a canonical coin graph representation from a non-maximal embedded planar graph: add a new vertex in each face, connected to all the vertices of the face. Find the coin graph representation of the augmented graph, and delete the circles representing the added vertices. Again, Theorem 2 can then find the optimal Möbius transformation of the resulting coin graph.

Due to the fact that a quasiconvex program only has a single global optimum, the transformed coin graph will display any symmetries present in the initial graph embedding. That is, any homeomorphism of the sphere that transforms the initial embedded graph into itself becomes simply a rotation or reflection of the sphere in the optimal embedding. If the graph has a unique embedding then any isomorphism of the graph becomes a rotation or reflection. For instance, the coin graph representation in Figure 2 (right) has the full symmetry of the underlying graph, while the planar drawing on the left of the figure does not show the symmetries that switch the vertices in the inner and outer squares.

Alternatively, by representing each vertex by the center of its circle, a coin graph representation can be used to find a straight-line drawing of a planar graph, or a drawing on the sphere in which the edges appear as non-crossing great-circle arcs. The algorithms in Section 3.2 can then be used to find a representation maximizing the minimum vertex separation, among all Möbius transformations of the initial vertex positions.

4.2 Hyperbolic Browser

There has been quite a bit of recent work in the information visualization community on *hyperbolic browsers*, techniques for using hyperbolic space to aid in the visualization of large graphs or graph-like structures such as the World-Wide Web [20]. In these techniques, a graph is arranged within a hyperbolic plane [16] or three-dimensional hyperbolic space [19], and viewed using the Klein or Poincaré models, or rendered as it would be seen by a viewer within the hyperbolic space. The main advantage of hyperbolic browsers is that they provide a "fish-eye" view [10] that allows both the details of the current focus of interest and the overall structure of the graph to be viewed simultaneously. In addition, the homogeneous and isotropic geometry of hyperbolic space allows for natural and smooth navigation from one view to another.

Although there are many interesting problems in graph layout for hyperbolic spaces, we are interested in a simpler question: where should one place one's initial focus, in order to make the overall graph structure as clear as possible? Previous work has handled this problem by the simplistic approach of laying out the graph using a rooted subtree such as a breadth-first or depth-first tree, and

placing the focus at the root of the tree. This approach will work well if the tree is balanced, but otherwise deeper parts of the tree may be given a much more crowded initial view. Instead, one could use our techniques to find a viewpoint that shows the whole graph as clearly as possible.

We assume we are given a graph, with vertices placed in a hyperbolic plane. As in [16], we further assume that each vertex has a circular *display region* where information related to that vertex is displayed; different nodes may have display regions of different sizes. It is straightforward to apply Theorem 1 to these display regions; the result is a focus placement for the Poincaré model that maximizes the minimum size of any display region. We can similarly find the Klein model maximizing the minimum diameter or width of a transformed circle, since the level sets of these functions are again simply concentric balls. By applying Theorem 3 we can instead choose the focus for a Poincaré model that maximizes the minimum distance between vertices, either among pairs from the given graph or among all possible pairs.

Open Problem 3. *Is there an efficient algorithm for choosing a Klein model of a hyperbolically embedded graph that maximizes the minimum Euclidean distance between adjacent vertices?*

We can apply similar methods to find a focus in 3-dimensional hyperbolic space that maximizes the minimum solid angle subtended by any display region.

Open Problem 4. *Does there exist an efficient algorithm to find a viewpoint in 3-dimensional hyperbolic space maximizing the minimum angle separating any pair among n given points?*

Even the 2-dimensional Euclidean version of this maxmin angle separation problem is interesting [17]: the level sets are nonconvex unions of two disks, so our quasiconvex programming techniques do not seem to apply.

4.3 Conformal Mesh Generation

One of the standard methods of two-dimensional structured mesh generation [3, 24] is based on *conformal mapping* (that is, an angle-preserving homeomorphism). The idea is to find a conformal map from the domain to be meshed into some simpler shape such as a disk, use some predefined template to form a mesh on the disk, and invert the map to lift the mesh back to the original domain. Conformal meshes have significant advantages: the orthogonality of the grid lines means that one can avoid certain additional terms in the definition of the partial differential equation to be solved [24]. Nevertheless, despite much work on algorithms for finding conformal maps [8, 12, 22, 23, 25] conformal methods are often avoided in favor of quasi-conformal mesh generation techniques that allow some distortion of angles, but provide greater control of node placement [3, 24].

Möbius transformations are conformal, and any two conformal maps from a simply connected domain to a disk can be related to each other via a Möbius

transformation. However, different conformal maps will lead to different structured meshes: the points of the domain mapped on or near the center of the disk will generally be included in mesh elements with the finest level of detail, and points near the boundary will be in coarser mesh elements. Therefore, as we now describe, we can use our optimal Möbius transformation algorithms to find the conformal mesh that best fits the desired level of detail at different parts of the domain, reducing the number of mesh elements created and providing some of the node placement control needed to use conformal meshing effectively.

We formalize the problem by assuming an input domain in which certain interior points p_i are marked with a desired element size s_i. If we find a conformal map f from the domain to a disk, the gradient of f maps the marked element sizes to desired sizes s'_i in the transformed disk: $s'_i = \|f'(p_i)\|$. We can then choose a structured mesh with element size $\min_i s'_i$ in the disk, and transform it back to a mesh of the original domain. The goal is to choose our conformal map in a way that maximizes $\min_i s'_i$, so that we can use a structured mesh with as few elements as possible. Another way of interpreting this is that s'_i can be seen as the radius of a small disk at $f(p_i)$. What we want is the viewpoint that maximizes the minimum of these radii.

By applying a single conformal map, found using one of the aforementioned techniques, we can assume without loss of generality that the input domain is itself a disk. Since the conformal maps from disks to disks are just the Möbius transformations, our task is then to find the Möbius transformation maximizing $\min_i s'_i$. Since s'_i depends only on the hyperbolic distance of the viewpoint from p_i, the level sets for this problem are themselves disks, so we can solve this problem by the same quasiconvex programming techniques as before. Indeed, we can view this problem as a limiting case of Theorem 1 for infinitesimally small (but unequal) sphere radii.

Open Problem 5. *Because of our use of a conformal map to a low aspect ratio shape (the unit disk), rotation around the viewpoint does not significantly affect element size. Howell [12] describes methods for computing conformal maps to high-aspect-ratio shapes such as rectangles. Can one efficiently compute the optimal choice of conformal map to a high-aspect-ratio rectangle to maximize the minimum desired element size? What if the rectangle aspect ratio can also be chosen by the optimization algorithm?*

4.4 Brain Flat Mapping

In order to visualize and understand the complicated strucure of the brain, neuroanatomists have sought methods for stretching its convoluted surface folds onto a flat plane. Hurdal et al. [13] proposed a principled way of performing this stretching via conformal maps: since the surfaces of major brain components such as the cerebellum are topologically disks, one can conformally map these surfaces onto a Euclidean unit disk, sphere, or hyperbolic plane. Hurdal et al. approximate this conformal map by using a fine triangular mesh to represent the brain surface, and forming a coin graph representation of this mesh.

Necessarily, any flat mapping of a curved surface such as the brain's involves some distortions of area, but the distortions produced by conformal mapping can be severe; thus, it would be of interest to choose the mapping in such a way that the distortion is minimized. As we already noted in section 4.3, the remaining degrees of freedom in choosing a conformal mapping can be described by a single Möbius transformation. Thus, we need to formulate a measure of distortion, and find the transformation optimizing that measure.

Since we want to measure area, and the mapping constructed by the method of Hurdal et al. is performed on triangles of a mesh, the most natural quality measure for this purpose seems to be in terms of those triangles: we want to minimize the maximum ratio a/a' where a is the area of a triangle in the initial three-dimensional map, and a' is the area of its image in the flat map. Unfortunately, we have not yet been able to extend our techniques to this quality measure. A positive answer to the following question would allow us to apply our quasiconvex programming algorithms:

Open Problem 6. *Let T be a triangle in the unit disk or sphere, and let C be the set of viewpoints for Möbius transformations that transform T into a triangle of area at least A. Is C necessarily convex?*

Instead of attempting to optimize the area of the triangles, it seems simpler (although perhaps less accurate) to optimize the area of the disks in the coin graph. Under the assumption that the initial triangular mesh has elements of roughly uniform size, it would be desirable that the coin graph representation similarly uses disks of as uniform size as possible. This can be achieved by our linear-time algorithms by applying Theorem 1 in the unit disk, or Theorem 2 in the sphere. In case the triangular mesh is nonuniform, it may be appropriate to apply a weighted version of these theorems, where the weight of each disk is computed from the lengths of edges incident to the corresponding mesh vertex.

5 Conclusions

We have identified several applications in information visualization and structured mesh generation for which it is of interest to find a Möbius transformation that optimizes an objective function, typically defined as the minimum size of a collection of geometric objects. Further, we have shown that these problems can be solved either by local optimization techniques, or by linear-time quasiconvex programming algorithms. For the problems where the input to the quasiconvex program is itself superlinear in size (maximizing the minimum distance between transformed points) we have described Delaunay triangulation and random sampling techniques for solving the problems in near-linear time.

We have listed open problems arising in our investigations throughout the paper. There is also an important problem in the practical application of our algorithms: although there should be little difficulty implementing local optimization techniques for our problems, the linear-time quasiconvex programming

algorithms are based on two primitives that (while constant time by general principles) have not been specified in sufficient detail for an implementation, one to test a new constraint against a given basis and the other to find the changed basis of a set formed by adding a new constraint to a basis. If the basis representation includes the value of the objective function, testing a new constraint is simply a matter of evaluating the corresponding object size and comparing it to the previous value. However, the less frequent basis change operations require a more detailed examination of the detailed structure of each problem, which we have not carried out. For an example of the difficulty of this step see [11]. In practice it may be appropriate to combine the two approaches, using local optimization techniques to find a numerical approximation to the basis change operations needed for the quasiconvex programming algorithms. Especially in the coin graph application, the input to the quasiconvex program is already itself a numerical approximation, so this further level of approximation should not cause additional problems, but one would need to verify that a quasiconvex programming algorithm can behave robustly with approximate primitives.

More generally, there are very few computational geometry algorithms involving hyperbolic geometry (a notable exception being [4]) although many Euclidean constructions such as the Delaunay triangulation or hyperplane arrangements can be translated to the hyperbolic case without difficulty using the Poincaré or Klein models. We expect many other interesting problems and algorithms to be discovered in this area.

Acknowledgements. We thank Hervé Brönnimann, Fan Chung, and Ken Stephenson for helpful discussions. Work of Eppstein was done in part while visiting Xerox PARC and supported in part by NSF grant CCR-9912338.

References

1. N. Amenta, M. Bern, and D. Eppstein. Optimal point placement for mesh smoothing. *J. Algorithms* 30(2):302–322, February 1999, cs.CG/9809081.
2. M. Bern and D. Eppstein. Optimal Möbius transformations for information visualization and meshing. ACM Computing Research Repository, 2001, cs.CG/0101006.
3. M. Bern and P. E. Plassmann. Mesh generation. *Handbook of Computational Geometry*, chapter 6, pp. 291–332. Elsevier, 2000.
4. J.-D. Boissonnat, A. Cérézo, O. Devillers, and M. Teillaud. Output-sensitive construction of the Delaunay triangulation of points lying in two planes. *Int. J. Comp. Geom. & Appl.* 6(1):1–14, 1996.
5. H. Breu and D. G. Kirkpatrick. On the complexity of recognizing intersection and touching graphs of disks. *Proc. 3rd Int. Symp. Graph Drawing*, pp. 88–98. Springer-Verlag, Lecture Notes in Comput. Sci. 1027, 1995.
6. G. R. Brightwell and E. R. Scheinerman. Representations of planar graphs. *SIAM J. Discrete Math.* 6(2):214–229, 1993.
7. C. Collins and K. Stephenson. A circle packing algorithm. Manuscript, September 1997, http://www.math.utk.edu/~kens/ACPA/ACPA.ps.gz.

8. T. A. Driscoll and S. A. Vavasis. Numerical conformal mapping using cross-ratios and Delaunay triangulation. *SIAM J. Sci. Comput.* 19(6):1783–1803, 1998, ftp://ftp.cs.cornell.edu/pub/vavasis/papers/crdt.ps.gz.
9. D. Eppstein. Setting parameters by example. *Proc. 40th Symp. Foundations of Computer Science*, pp. 309–318. IEEE, October 1999, cs.DS/9907001.
10. A. Formella and J. Keller. Generalized fisheye views of graphs. *Proc. 3rd Int. Symp. Graph Drawing*, pp. 243–253. Springer-Verlag, Lecture Notes in Comput. Sci. 1027, 1995, http://www-wjp.cs.uni-sb.de/~formella/dist2.ps.gz.
11. B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. *Inf. Proc. Lett.* 68:33–38, 1998.
12. L. H. Howell. *Computation of Conformal Maps by Modified Schwarz-Christoffel Transformations*. Ph.D. thesis, MIT, 1990, http://www.llnl.gov/CASC/people/howell/lhhphd.ps.gz.
13. M. K. Hurdal, P. L. Bowers, K. Stephenson, D. W. L. Summers, K. Rehm, K. Shaper, and D. A. Rottenberg. Quasi-conformally flat mapping the human cerebellum. Tech. Rep. FSU-99-05, Florida State Univ., Dept. of Mathematics, 1999, http://www.math.fsu.edu/~aluffi/archive/paper98.ps.gz.
14. B. Iversen. *Hyperbolic Geometry*. London Math. Soc. Student Texts 25. Cambridge Univ. Press, 1992.
15. P. Koebe. Kontaktprobleme der konformen Abbildung. *Ber. Verh. Sächs. Akad. Wiss. Leipzig* Math.-Phys. Kl. 88:141–164, 1936.
16. J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for viewing large hierarchies. *Proc. Conf. Human Factors in Computing Systems*, pp. 401–408. ACM, 1995, http://www.parc.xerox.com/istl/projects/uir/pubs/pdf/UIR-R-1995-04-Lamping-CHI95-FocusContext.pdf.
17. J. McKay. Sighting point. sci.math, 20 April 1989, http://www.ics.uci.edu/~eppstein/junkyard/maxmin-angle.html.
18. B. Mohar. A polynomial time circle packing algorithm. *Discrete Math.* 117(1–3):257–263, 1993.
19. T. Munzner. Exploring large graphs in 3D hyperbolic space. *IEEE Comp. Graphics & Appl.* 18(4):18–23, 1997, http://graphics.stanford.edu/papers/h3cga/.
20. T. Munzner and P. Burchard. Visualizing the structure of the world wide web in 3D hyperbolic space. *Proc. VRML '95*, pp. 33–38. ACM, 1995, http://www.geom.umn.edu/docs/research/webviz/webviz/.
21. H. Sachs. Coin graphs, polyhedra, and conformal mapping. *Discrete Math.* 134(1–3):133–138, 1994.
22. W. D. Smith. Accurate circle configurations and numerical conformal mapping in polynomial time, http://www.neci.nj.nec.com/homepages/wds/braegger.ps. Manuscript, December 1991.
23. F. Stenger and R. Schmidtlein. Conformal maps via sinc methods. *Proc. Conf. Computational Methods in Function Theory*, pp. 505–549. World Scientific, 1997, http://www.cs.utah.edu/~stenger/PAPERS/stenger-sinc-comformal-maps.ps.
24. J. F. Thompson, Z. U. A. Warsi, and C. W. Mastin. *Numerical Grid Generation: Foundations and Applications*. North-Holland, 1985.
25. L. N. Trefethen. Numerical computation of the Schwarz-Christoffel transformation. *SIAM J. Sci. Stat. Comput.* 1(1):82–102, 1980.

Using the Pseudo-Dimension to Analyze Approximation Algorithms for Integer Programming

Philip M. Long

Genome Institute of Singapore
1 Research Link
IMA Building
National University of Singapore
Singapore 117604, Republic of Singapore

Abstract. We prove approximation guarantees for randomized algorithms for packing and covering integer programs expressed in certain normal forms. The bounds are in terms of the pseudo-dimension of the matrix of the coefficients of the constraints and the value of the optimal solution; they are independent of the number of constraints and the number of variables. The algorithms take time polynomial in the length of the representation of the integer program and the value of the optimal solution. We establish a related result for a class we call the mixed covering integer programs, which contains the covering integer programs. We describe applications of these techniques and results to a generalization of Dominating Set motivated by distributed file sharing applications, to an optimization problem motivated by an analysis of boosting, and to a generalization of matching in hypergraphs.

1 Introduction

Raghavan and Thompson [RT87] introduced *randomized rounding*. Roughly, their idea was to construct algorithms for integer programming problems as follows:

- solve a similar problem without the integrality constraint, and
- round each variable up or down, using its fractional part as the probability of rounding up.

This technique provides strong approximation guarantees for polynomial-time algorithms for a class of problems called covering and packing integer programs [Rag88,Sri99]. In a covering integer program, for an $m \times n$ matrix A and column vectors c and b, all with only nonnegative entries, the goal is to find $x \in \mathbf{Z}_+^n$ to minimize $c^T x$ subject to $Ax \geq b$. In a packing integer program, it is also assumed that A, c and b have only nonnegative entries, but the goal is to choose $x \in \mathbf{Z}_+^n$ to maximize $c^T x$ subject to $Ax \leq b$. Raghavan [Rag88] asked whether one could exploit algebraic properties of A such as its rank to obtain stronger approximation guarantees.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 26–37, 2001.

In this paper, we report on work along these lines. One can assume without loss of generality that covering and packing integer programs satisfy $A \in [0,1]^{m \times n}$, $c \in [1,\infty)^n$ and $b = (1,1,...,1)^T$ (see [Sri99] and Sections 3 and 5 of this paper). For covering and packing integer programs that are expressed this way, we prove approximation guarantees for efficient algorithms that are independent of the number of variables and constraints, and are in terms of the *pseudo-dimension* [Pol84,Hau92] of A. We also establish a similar result for *mixed covering integer programs*, which are like covering integer programs but without the requirement that the components of A are nonnegative.

The pseudo-dimension can be defined as follows [Hau92]. Say that an $m \times k$ matrix is *full* if the origin in $\mathbf{R}^k$ can be translated so that the rows of the matrix occupy all 2^k orthants. The pseudo-dimension of A is the size of the largest set of columns of A such that the matrix obtained by deleting all other columns is full. A more formulaic definition is given in Section 2.

Since the pseudo-dimension of A is at most its rank [Dud78,Pol84], our analysis implies results like those envisaged by Raghavan. Sometimes, however, the pseudo-dimension of a matrix is much smaller than its rank. For example, the pseudo-dimension of any identity matrix is 1.

Our general results are as follows. All of our algorithms are randomized and, with probability 1/2, achieve the claimed approximations in time polynomial in the number of bits needed to write A and c and the value of the optimal solution. (For many commonly studied combinatorial optimization problems, the value of the optimal solution is bounded by a polynomial in the size of the input [KT94, CK].) The algorithm for covering integer programs outputs a solution whose value is $O(\text{opt}(1 + dr\log(r\text{opt})))$, where d is the pseudo-dimension of A and r is the value of the largest entry in A; since $r \leq 1$, the value of the algorithm's solution is also $O(d\text{opt}\log\text{opt})$. For mixed covering integer programs, the bound is $O(\text{opt}(1 + dr^2\text{opt}))$; here we cannot assume that $r \leq 1$. For packing integer programs, our algorithm obtains a solution whose value is $\Omega\left(\frac{\text{opt}}{(2r\text{opt})^{krd}}\right)$, for a constant $k > 0$ (here once again $r \leq 1$).

We illustrate the application of our general result about covering integer programs using the B-domination problem [NR95,Sri99], a generalization of Dominating Set motivated by distributed file sharing applications. In the B-domination problem, the goal is to locate as few facilities as possible at the nodes of a network so that each node of the network has at least B facilities within one hop. We give a randomized algorithm that, for graphs of constant genus, with probability 1/2, outputs a solution of size $O\left(\text{opt}\left(1 + \frac{\ln \text{opt}}{B}\right)\right)$ in polynomial time.

Our study of mixed covering integer programs was inspired by a learning problem, which can be abstracted as the *minimum majority problem* as follows: given an $m \times n$ matrix A with entries in $\{-1,1\}$, choose $x \in \mathbf{Z}_+^n$ to minimize $\sum_{i=1}^n x_i$ subject to $Ax > 0$. Our algorithm for mixed covering integer programs yields a bound of $O(d\text{opt}^2)$ for this problem. We derive our motivation for this problem from an analysis of the generalization ability of hypotheses output by boosting algorithms [SFBL98]; details are given in Section 6.2.

Our general results about packing integer programs can be applied to simple B-matching [Lov75]. Here, given a family $\mathcal{S}$ of subsets of a finite set X, the goal is to output as many of the sets in $\mathcal{S}$ as possible while ensuring that each element of X is included in at most B of the chosen sets. We give a randomized polynomial-time algorithm for this problem that outputs a solution of size $\Omega((\text{opt}/B)^{1-kd/B})$, where d is the VC-dimension of the dual of the input and $k > 0$ is an absolute constant.

Our work builds on that of Bronnimann and Goodrich [BG95] and Pach and Agarwal [PA95], who established approximation guarantees for polynomial-time algorithms for Set Cover in terms of the VC-dimension of the dual of the input set system. Our analysis of covering integer programs is a generalization of the analysis of Pach and Agarwal. Set Cover can be formulated as a covering integer program, and the pseudo-dimension of the resulting coefficient matrix is the same as the VC-dimension of the dual of the input set system.

Srinivasan [Sri99] showed that if a fractional solution is rounded as originally proposed by Raghavan and Thompson, then the events that the constraints are violated are *positively correlated*, and used this to improve the analysis of randomized rounding for packing and covering integer programs. Recently, he provided *RNC* and *NC* algorithms with the same approximation guarantees [Sri01]. However, his approximation bounds still depend on m.

Baker [Bak94] described a polynomial-time approximation scheme for Dominating Set when the input is restricted to be planar.

It is not hard to see how to use boosting [Sch90 Fre95], together with Lemma 3.3 of [HMP+93], to design an algorithm for the minimum majority problem that outputs a solution with value $O(\text{opt}^2 \log m)$. Since $d \leq \log m$, our bound is never more than a constant factor worse than this, but when $d << \log m$, it is significantly better.

For simple B-matching, the only bounds we know are in terms of opt and $|X|$; the best is $\Omega\left(\frac{\text{opt}}{1+(|X|/\text{opt})^{1/B}}\right)$ [Sri99]. When $d << B << \text{opt} << |X|$ and $d << \log |X|$ (note again that $d \leq \log |X|$), our bound improves on this significantly.

2 Preliminaries

Denote the nonnegative rationals by $\mathbf{Q}_+$, and the nonnegative integers by $\mathbf{Z}_+$.

For a countable set X, a probability distribution D over X, and a predicate ϕ over X, denote by $\mathbf{Pr}_{x \in D}(\phi(x))$ the probability that $\phi(x)$ is true when x is chosen according to D. Define $\mathbf{E}_{x \in D}$ similarly. Denote by D^ℓ the distribution on X^ℓ obtained by sampling ℓ times independently according to D.

For a domain X, and a subset S of X, define χ_S to be the indicator function for S, i.e. function from X to $\{0, 1\}$ for which $\chi_S(x) = 1 \Leftrightarrow x \in S$.

For a domain X, say that a set $\mathcal{F}$ of real-valued functions defined on X *shatters* a sequence $x_1, ..., x_d$ of elements of X if there is a sequence $r_1, ..., r_d$ of real thresholds such that for any $b_1, ..., b_d \in \{\text{above}, \text{below}\}$, there is an $f \in \mathcal{F}$ such that for all $i \in \{1, ..., d\}$, $f(x_i) \geq r_i \Leftrightarrow b_i = \text{above}$. Define the *pseudo-dimension* [Pol84] of $\mathcal{F}$, denoted by $\text{Pdim}(\mathcal{F})$, to be the length of the longest

sequence shattered by $\mathcal{F}$. The *VC-dimension* [VC71] of a set $\mathcal{F}$ of functions from X to $\{0,1\}$, denoted by $\mathrm{VCdim}(\mathcal{F})$, is its pseudo-dimension. The VC-dimension of a family $\mathcal{S}$ of subsets of X is the VC-dimension of $\{\chi_S : S \in \mathcal{S}\}$.

For a real matrix A, define the *pseudo-dimension of* A, denoted by $\mathrm{Pdim}(A)$, by thinking of the rows of A as functions and taking the pseudo-dimension of the resulting class of functions. Specifically, if A is an $m \times n$ matrix, for each $i \in \{1,...,m\}$ define $f_{A,i} : \{1,...,n\} \to \mathbf{R}$ by $f_{A,i}(j) = A_{i,j}$ and let $\mathrm{Pdim}(A) = \mathrm{Pdim}(\{f_{A,i} : i \in \{1,...,m\}\})$.

For a family $\mathcal{S}$ of sets define the *dual* of $\mathcal{S}$, denoted by $\mathrm{dual}(\mathcal{S})$ as follows. For each $x \in \cup_{S\in\mathcal{S}} S$, let $Q_{x,\mathcal{S}} = \{S \in \mathcal{S} : x \in S\}$. Let $\mathrm{dual}(\mathcal{S}) = \{Q_{x,\mathcal{S}} : x \in \cup_{S\in\mathcal{S}} S\}$.

Lemma 1 ([Vap82,Pol84]). *There is a constant $\kappa > 0$ such that for any $r > 0$, any finite set X, any set $\mathcal{F}$ of functions from X to $[0,r]$, any $\epsilon > 0$, and any probability distribution D over X, if $\ell \geq \frac{\kappa r \mathrm{Pdim}(\mathcal{F})}{\epsilon} \ln \frac{r}{\epsilon}$, then*

$$\mathbf{Pr}_{(z_1,...,z_\ell)\in D^\ell}\left(\exists f \in \mathcal{F}, \mathbf{E}_{x\in D}(f(x)) \geq \epsilon \text{ but } \sum_{i=1}^{\ell} f(z_i) < \epsilon\ell/2\right) \leq 1/4.$$

Lemma 2 ([Tal94]). *There is a constant $\kappa > 0$ such that for any real a and b with $a \leq b$ (let $r = b - a$), any finite set X, any set $\mathcal{F}$ of functions from X to $[a,b]$, any $\epsilon > 0$, and any probability distribution D over X, if $\ell \geq \frac{\kappa r^2 \mathrm{Pdim}(\mathcal{F})}{\epsilon^2}$, then*

$$\mathbf{Pr}_{(z_1,...,z_\ell)\in D^\ell}\left(\exists f \in \mathcal{F}, \left|\mathbf{E}_{x\in D}(f(x)) - \frac{1}{\ell}\sum_{i=1}^{\ell} f(z_i)\right| > \epsilon\right) \leq 1/4,$$

Lemma 3. *There is a constant $\kappa > 0$ such that for any $r > 0$, any finite set X, any set $\mathcal{F}$ of functions from X to $[0,r]$, any $\epsilon > 0$, and any probability distribution D over X, and for any $\alpha \geq 1$, if $\ell \geq \frac{\kappa r \mathrm{Pdim}(\mathcal{F})}{\alpha \ln(1+\alpha)\epsilon} \ln \frac{r}{\epsilon}$, then*

$$\mathbf{Pr}_{(z_1,...,z_\ell)\in D^\ell}\left(\exists f \in \mathcal{F}, \mathbf{E}_{x\in D}(f(x)) \leq \epsilon \text{ but } \sum_{i=1}^{\ell} f(z_i) > (1+\alpha)\epsilon\ell\right) \leq 1/4.$$

We are not aware of a reference for Lemma 3. Its proof, whose rough outline follows those of related results (see [Pol84,Hau92,SAB93,AB99]), is omitted due to space contraints.

3 Covering Integer Programs

In a covering integer program, for natural numbers n and m, column vectors $c \in \mathbf{Q}_+^n$ and $b \in \mathbf{Q}_+^m$, and a matrix $A \in \mathbf{Q}_+^{m\times n}$, the goal is to choose $x \in \mathbf{Z}_+^m$ to minimize $c^T x$ subject to $Ax \geq b$.

Srinivasan [Sri99] showed that one can assume without loss of generality that $A \in [0,1]^{m \times n}$ and $b \in [1,\infty)^m$. By dividing each row i of A by b_i, one can further assume w.l.o.g. that each component of b is 1. Furthermore, one can assume that each component of c is positive, since if some $c_j = 0$, one can eliminate the jth variable by deleting all constraints that can be satisfied by making it arbitrarily large. Finally, we can scale c so that its least component is 1. This is summarized in the following.

Definition 1. *A* covering integer program in normal form *is given by a matrix* $A = [0,1]^{m \times n}$ *and a column vector* $c \in [1,\infty)^n$. *The goal is to find a column vector* $x \in \mathbf{Z}^n$ *such that* $x \geq (0,0,...,0)^T$ *and* $Ax \geq (1,1,...,1)^T$ *in order to minimize* $c^T x$.

Theorem 1. *There is a polynomial* q *and a randomized algorithm* R *with the following property. For any covering integer program* (A,c) *in normal form, if* $r = \max_{i,j} A_{i,j}$ *and* L *is the number of bits required to write* A *and* c, *then with probability* $1/2$, *Algorithm* R *outputs a feasible solution* x *in* $q(L, \mathrm{opt}(A,c))$ *time whose solution has cost that is* $O(\mathrm{opt}(A,c)(1 + r\mathrm{Pdim}(A)\log(r\mathrm{opt}(A,c))))$.

Proof Sketch: For the sake of brevity, we will consider an algorithm (let's call it R') that makes use of the knowledge of $\mathrm{Pdim}(A)$. It is not hard to see how to remove the need for this knowledge. Algorithm R' is as follows.

- Solve the linear program obtained by relaxing the integrality constraint. Call the solution u.
- Set $Z = \sum_{i=1}^n u_i$, and $p = u/Z$. Note that p can be interpreted as a probability distribution on $\{1,...,n\}$. Note also that $Z \geq 1/r$, since otherwise all constraints would be violated.
- Let κ be as in Lemma 1 and $\ell = \max\{\lceil 2\kappa \mathrm{Pdim}(A) r Z \ln(rZ)\rceil, \lceil 2Z\rceil\}$. Sample ℓ times at random independently according to p, and, for each j, let x_j be the number of times that j occurs.
- Output $x = (x_1,...,x_n)$.

Choose an input (A,c) and let $r = \max_{i,j} A_{i,j}$, $\mathrm{opt} = \mathrm{opt}(A,c)$, and $d = \mathrm{Pdim}(A)$. Let $a_1,...,a_m$ be the rows of A. Since $Au \geq (1,1,...,1)^T$, we have $Ap \geq (1/Z)(1,1,...,1)^T$. Thus, for each i, we have $\mathbf{E}_{j \in p}(A_{i,j}) = \mathbf{E}_{j \in p}(f_{A,i}(j)) \geq 1/Z$.

Since, for each i, incrementing x_j has the effect of increasing $a_i \cdot x$ by $A_{i,j} = f_{A,i}(j)$, applying Lemma 1 with $\epsilon = 1/Z$, with probability at least $3/4$, for all i, $a_i \cdot x \geq \ell/(2Z) \geq 1$. Thus,

$$\mathbf{Pr}(x \text{ is not feasible }) \leq 1/4. \tag{1}$$

We have

$$\mathbf{E}(c^T x) = \ell c^T p \leq \ell \mathrm{opt}/Z \leq \frac{\max\{\lceil 2\kappa d r Z \ln(rZ)\rceil, \lceil 2Z\rceil\}\mathrm{opt}}{Z}.$$

Thus, Markov's inequality implies that

$$\mathbf{Pr}\left(c^T x > \frac{4\max\{\lceil 2\kappa d(rZ)\ln(rZ)\rceil, \lceil 2Z\rceil\}\mathrm{opt}}{Z}\right) \leq 1/4. \tag{2}$$

Since each $c_i \geq 1$, we have $Z = \sum_{i=1}^n u_i \leq \sum_{i=1}^n c_i u_i \leq \mathrm{opt}$. Combining with (1) and (2) completes the proof. □

4 Mixed Covering Integer Programs

In a *mixed covering integer program*, for natural numbers m and n, column vectors $c \in \mathbf{Q}_+^n$ and $b \in \mathbf{Q}_+^m$, and a matrix $A \in \mathbf{Q}^{m\times n}$, the goal is to choose $x \in \mathbf{Z}_+^m$ to minimize $c^T x$ subject to $Ax \geq b$. Note that to be a mixed covering integer program, the entries of A need not be nonnegative. If $b = (1, 1, ..., 1)^T$ and $c \in [1, \infty)^n$, then we say that the mixed covering integer program is in *normal form*. (This can be seen to be without loss of generality as with covering integer programs.) Note however, that here we cannot assume without loss of generality that the entries of A are at most 1.

Theorem 2. *There is a polynomial q and a randomized algorithm R with the following property. For any mixed covering integer program (A, c) in normal form, if $r = \max_{i,j} |A_{i,j}|$ and L is the number of bits required to write A and c, then with probability $1/2$, Algorithm R outputs a feasible solution x in $q(L, \mathrm{opt}(A, c))$ time whose solution has cost that is $O(\mathrm{Pdim}(A)r^2\mathrm{opt}(A, c)^2 + \mathrm{opt}(A, c))$.*

Proof Sketch: As in the proof of Theorem 1, we will consider an algorithm R' that "knows" $\mathrm{Pdim}(A)$; the algorithm is the same as in that proof, except κ is defined as in Lemma 2, and $\ell = \max\{\lceil 4\kappa\mathrm{Pdim}(A)r^2Z^2\rceil, \lceil 2Z\rceil\}$. We will borrow notation from that proof.

As before, since $Au \geq (1, 1, ..., 1)^T$ and $p = u/Z$, for each $i \in \{1, ..., m\}$, $\mathbf{E}_{j\in p}(f_{A,i}(j)) \geq 1/Z$. Thus

$$\begin{aligned}
\mathbf{Pr}(x \text{ is not feasible}) &\leq \mathbf{Pr}_{(j_1,\ldots,j_\ell)\in p^\ell}\left(\exists i, \mathbf{E}(f_{A,i}) \geq 1/Z \text{ but } \sum_{t=1}^{\ell} f_{A,i}(j_t) < 1\right) \\
&\leq \mathbf{Pr}_{(j_1,\ldots,j_\ell)\in p^\ell}\left(\exists i, \mathbf{E}(f_{A,i}) \geq 1/Z \text{ but } \sum_{t=1}^{\ell} f_{A,i}(j_t) < \frac{\ell}{2Z}\right) \\
&\leq \mathbf{Pr}_{(j_1,\ldots,j_\ell)\in p^\ell}\left(\exists i, \left|\mathbf{E}(f_{A,i}) - \frac{1}{\ell}\sum_{t=1}^{\ell} f_{A,i}(j_t)\right| > \frac{1}{2Z}\right)
\end{aligned}$$

which is at most $1/4$ by Lemma 2. But

$$\mathbf{E}(c^T x) \leq \ell\mathrm{opt}/Z \leq \frac{\max\{\lceil 4\kappa\mathrm{Pdim}(A)r^2Z^2\rceil, \lceil 2Z\rceil\}\mathrm{opt}}{Z}.$$

Applying Markov's inequality and the fact that $Z \leq \mathrm{opt}$ as in the proof of Theorem 1 completes the proof. □

5 Packing Integer Programs

In a packing integer program, for natural numbers n and m, column vectors $c \in \mathbf{Q}_+^n$ and $b \in \mathbf{Q}_+^m$, and a matrix $A \in \mathbf{Q}_+^{m \times n}$, the goal is to choose $x \in \mathbf{Z}_+^m$ to maximize $c^T x$ subject to $Ax \leq b$.

Arguing as for covering, one can assume without loss of generality that entries of A are in $[0,1]$ and $b = (1,1,...,1)^T$. Furthermore, one can also assume in this case that each component of c is positive; here if some $c_j = 0$, you might as well set $x_j = 0$, and thus, the jth variable can be eliminated. Since again we can scale c so that its least component is 1, we arrive at the following.

Definition 2. *A* packing integer program in normal form *is given by a matrix* $A = [0,1]^{m \times n}$ *and a column vector* $c \in [1,\infty)^n$. *The goal is to find a column vector* $x \in \mathbf{Z}^n$ *such that* $x \geq (0,0,...,0)^T$ *and* $Ax \leq (1,1,...,1)^T$ *in order to maximize* $c^T x$.

Theorem 3. *There is a constant* $k > 0$, *a randomized polynomial-time algorithm* R *and a polynomial* q *with the following property. For any packing integer program* (A,c) *in normal form, if* B *is the least integer such that* $\max_{i,j} A_{i,j} \leq 1/B$, L *is the number of bits in the representation of* A *and* c, *and* $d = \mathrm{Pdim}(A)$, *with probability* $1/2$, *Algorithm* R *outputs a feasible solution* x *in* $q(L, \mathrm{opt}(A,c))$ *time whose solution has value that is* $\Omega\left(\frac{\mathrm{opt}(A,c)}{(\mathrm{opt}(A,c)/B)^{kd/B}}\right)$.

Proof Sketch: The fact that the entries of A are at most $1/B$ implies that any x with $\sum_{i=1} x_i \leq B$ is feasible. This, together with the fact that each component of c is at least 1, implies that it is trivial to find a solution of value B. Hence, we can assume without loss of generality that $\frac{\mathrm{opt}}{(\mathrm{opt}/B)^{kd/B}} \geq B$ and therefore, since $\mathrm{opt} \geq B$, that $kd/B \leq 1$.

Again, we will consider an algorithm R' that "knows" $\mathrm{Pdim}(A)$:

- Solve the linear program obtained by relaxing the integrality constraint. Call the solution u.
- Set $Z = \sum_{i=1}^n u_i$, and $p = u/Z$. (Note that $Z \geq B$; otherwise, since the entries of A are at most $1/B$, no constraints would be binding, and u could be improved.)
- Let κ be as in Lemma 3, $d = \mathrm{Pdim}(A)$, $\alpha = (Z/B)^{4\kappa d/B}$, $\ell = \left\lceil \frac{\kappa d (Z/B)\ln(Z/B)}{\alpha \ln(1+\alpha)} \right\rceil$. Sample ℓ times at random independently according to p, and, for each j, let x_j be the number of times that j occurs.
- Output $x = (x_1, ..., x_n)$.

Choose an input (A,c) and let $a_1, ..., a_m$ be the rows of A. Let B, d, α and ℓ be as in the description of Algorithm R', and let $\mathrm{opt} = \mathrm{opt}(A,c)$.

Suppose $\ell = 1$. Again, since the entries in A are at most 1, and the constraints are of the form $a_i \cdot x \leq 1$, then since in this case $\sum_{i=1}^n x_i = 1$, x is certainly feasible.

Suppose $\ell > 1$. Since $Au \leq (1, 1, ..., 1)^T$, for each i, we have $\mathbf{E}(a_i \cdot x) \leq \ell/Z$. Applying part (c) of Lemma 3 (note that since $Z \geq B$, $\alpha \geq 1$), with probability at least 3/4, for all i,

$$a_i \cdot x \leq (1+\alpha)\ell/Z = \frac{1+\alpha}{Z}\left\lceil \frac{\kappa d(Z/B)\ln(Z/B)}{\alpha \ln(1+\alpha)} \right\rceil \leq \frac{4\kappa d(Z/B)\ln(Z/B)}{Z\ln(1+\alpha)} \leq 1$$

where the second inequality holds because $\alpha \geq 1$ and $\ell > 1$.

Thus, whatever the value of ℓ, we have

$$\mathbf{Pr}(x \text{ is not feasible }) \leq 1/4. \tag{3}$$

Applying Chebyshev's inequality yields

$$\mathbf{Pr}\left(c^T x < \frac{\ell c^T u}{Z} - 2\sqrt{\frac{\ell c^T u}{Z}}\right) \leq 1/4. \tag{4}$$

Substituting the value of ℓ and simplifying, we have

$$\frac{\ell c^T u}{Z} \geq \frac{c^T u}{\left(\frac{1}{B}\sum_{i=1}^{n} u_i\right)^{kd/B}} \geq \frac{c^T u}{(c^T u/B)^{kd/B}} \geq \frac{\text{opt}}{(\text{opt}/B)^{kd/B}},$$

since $c^T u \geq \text{opt}$ and $kd/B \leq 1$. Putting this together with (4) and (3) completes the proof. □

6 Applications

In this section, we give examples of the application of our general results.

6.1 Dominating Set and Extensions

The *B-domination problem* [NR95,Sri99] is defined as follows: given a graph $G = (V, E)$, place as few facilities as possible on the vertices of G in such a way that each vertex has at least B facilities in its neighborhood. The neighborhood of a vertex is defined to consist of the vertex and all vertices sharing an edge with it.

Define $\mathcal{N}(G)$ to be the set system consisting of all the neighborhoods in G, i.e.

$$\mathcal{N}(G) = \{\{w : \{v, w\} \in E\} \cup \{v\} : v \in V\}.$$

Theorem 4. *For each natural number B, there is a polynomial-time algorithm A for the B-domination problem such that for any graph G with optimal solution* $\text{opt}(G, B)$*, algorithm A outputs a solution of size*

$$O\left(\text{opt}(G,B)\left(1 + \frac{\text{VCdim}(\mathcal{N}(G))}{B}\log\frac{\text{opt}(G,B)}{B}\right)\right).$$

Proof: If x_v is the number of facilities located at vertex v, the problem is to minimize $\sum_{v \in V} x_v$ subject to the constraints, one for each vertex v, that

$$\left(\left(\sum_{w:\{v,w\} \in E} x_w \right) + x_v \right) / B \geq 1$$

(and that the x_v's are nonnegative integers). Since $\mathcal{N}(G) = \text{dual}(\mathcal{N}(G))$, and scaling all members of a set of functions by a common constant factor does not change its pseudo-dimension, applying Theorem 1 completes the proof. □

The following is an example of how this can be applied. Recall that the genus of a graph is, informally, the number of "handles" that need to be added to the plane before the graph can be embedded without any edge crossings.

Theorem 5. *Choose a fixed nonnegative integer k.*

For each natural number B, there is a polynomial-time algorithm A for the B-domination problem such that for any graph G of genus at most k with optimal solution $\text{opt}(G, B)$, algorithm A outputs a solution of size

$$O\left(\text{opt}(G,B)\left(1 + \frac{\log \text{opt}(G,B)}{B}\right)\right).$$

Proof Sketch: We bound the VC-dimension of $\mathcal{N}(G)$ in terms of the genus of G and apply Theorem 4. Details are omitted from this abstract. □

6.2 Sparse Majorities of Weak Hypotheses

The *minimum majority problem* is to, given an $m \times n$ matrix A with entries in $\{-1, 1\}$, choose $x \in \mathbf{Z}_+^n$ to minimize $\sum_{i=1}^n x_i$ subject to $Ax > 0$. In other words, choose as short a sequence $j_1, ..., j_k$ of columns as possible such that for each row i, a majority of $A_{i,j_1}, ..., A_{i,j_k}$ are 1. The following is an immediate consequence of Theorem 2.

Theorem 6. *There is a randomized polynomial time algorithm for the minimum majority problem that, with probability $1/2$, outputs a solution of cost $O(\text{opt}^2 \text{Pdim}(A))$.*

The minimum majority problem is a restatement of an optimization problem motivated by learning applications. Many learning problems can be modeled as that of approximating a $\{0, 1\}$-valued function using examples of its behavior when applied to randomly drawn elements of its domain [Val84,Hau92]; the approximation is sometimes called a *hypothesis*. *Boosting* [Sch90,Fre95,FS97] is a method for combining "weak hypotheses", which are correct on only a slight majority of the input examples, into a "strong hypothesis", which outputs a weighted majority vote of the weak hypotheses. The key idea of the most influential analysis of the ability of the strong hypothesis to generalize to unseen domain elements [SFBL98] is to use the fact that it can be approximated by a majority of a few of the weak hypotheses. This suggests an alternative approach

to the design of a learning algorithm: try directly to find hypotheses that explain the data well using majorities of as few as possible of a collection of weak hypotheses. This is captured by the minimum majority problem: the columns correspond to examples, the rows to weak hypotheses, and an entry indicates whether a given weak hypothesis is correct on a given example. The goal is to find a small multiset of weak hypotheses whose majority is correct on all of a collection of examples. This direct optimization might provide improved generalization, but even if not, its output should be easier to interpret, which is an important goal for some applications [Qui99].

6.3 Simple B-Matching

The problem of simple B-matching [Lov75] is to, given a family $\mathcal{S}$ of subsets of a finite set X, find a large $\mathcal{T} \subseteq \mathcal{S}$ such that each element of X is contained in at most B of the sets in $\mathcal{T}$.

Theorem 7. *There is a constant k such that for all integers $B \geq 1$ and $d \geq 2$, there is a polynomial time algorithm for the simple B matching problem that, for any input $\mathcal{S}$ such that* $\text{VCdim}(\text{dual}(\mathcal{S})) \leq d$, *outputs a solution of size* $\Omega((\text{opt}(\mathcal{S})/B)^{1-kd/B})$.

Proof: Consider the variant of the simple B matching problem in which multiple copies of sets in $\mathcal{S}$ can be included in the output. This problem can be expressed as a packing integer program in normal form as follows. For each $S \in \mathcal{S}$, include a variable x_S indicating the number of copies of S in the output. Then the goal is to maximize $\sum_{S \in \mathcal{S}} x_S$ subject to the constraints, one for each $x \in X$, that $\left(\sum_{S \in \mathcal{S}: x \in S} x_S\right)/B \leq 1$.

Suppose $\text{opt}(\mathcal{S})$ is the optimal value of the objective function for the original simple B-matching problem. Since the optimal value of the objective function for the multiple-copy variant is at least $\text{opt}(\mathcal{S})$, and since, once again, scaling elements of a set of functions by a common constant factor does not affect its pseudo-dimension, Theorem 3 implies that the value of the solution output by the algorithm described above is $\Omega\left(\frac{\text{opt}(\mathcal{S})}{(\text{opt}(\mathcal{S})/B)^{kd/B}}\right)$. Certainly no more than B copies of any set are included, so if we output one copy of all sets for which $x_S > 0$ we get a solution of size $\Omega\left((\text{opt}(\mathcal{S})/B)^{1-kd/B}\right)$. □

7 Concluding Remark

Other generalizations of the VC-dimension to real and integer-valued functions have been proposed, and results similar to Lemmas 1 and 2 proved for them (see [Dud78,Nat89,Vap89,KS94,BCHL92,ABCH97,BLW96,BL98]). It is easy to see how to prove analogues of Theorems 1, 2 and 3 for any of these. In some cases, these may provide easier analyses or stronger guarantees.

Acknowledgement. We thank Aravind Srinivasan and an anonymous referee for their helpful comments about an earlier draft of this paper. We gratefully acknowledge the support of National University of Singapore Academic Research Fund Grant RP–252–000–070–107.

References

AB99. M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.

ABCH97. N. Alon, S. Ben-David, N. Cesa-Bianchi, and D. Haussler. Scale-sensitive dimensions, uniform convergence, and learnability. *Journal of the Association for Computing Machinery*, 44(4):616–631, 1997.

Bak94. B.S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the Association for Computing Machinery*, 41:153–180, 1994.

BCHL92. S. Ben-David, N. Cesa-Bianchi, D. Haussler, and P. M. Long. Characterizations of learnability for classes of $\{0, \ldots, n\}$-valued functions. *Journal of Computer and System Sciences*, 50(1):74–86, 1992.

BEHW89. A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *JACM*, 36(4):929–965, 1989.

BG95. H. Bronnimann and M. Goodrich. Almost optimal set covers in finite VC-dimension. *Discrete Comput. Geom.*, 14:263–279, 1995.

BL98. P. L. Bartlett and P. M. Long. Prediction, learning, uniform convergence, and scale-sensitive dimensions. *Journal of Computer and System Sciences*, 56(2):174–190, 1998.

BLW96. P. L. Bartlett, P. M. Long, and R. C. Williamson. Fat-shattering and the learnability of real-valued functions. *Journal of Computer and System Sciences*, 52(3):434–452, 1996.

CK. P. Crescenzi and V. Kann. A compendium of NP optimization problems. See http://www.nada.kth.se/viggo/problemlist/compendium.html.

Dud78. R. M. Dudley. Central limit theorems for empirical measures. *Annals of Probability*, 6(6):899–929, 1978.

Fel87. M.R. Fellows. The Robertson-Seymour theorems: A survey of applications. *Contemp. Math.*, 89:1–18, 1987.

Fre95. Y. Freund. Boosting a weak learning algorithm by majority. *Information and Computation*, 121(2):256–285, 1995.

FS97. Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

Hau92. D. Haussler. Decision theoretic generalizations of the PAC model for neural net and other learning applications. *Information and Computation*, 100(1):78–150, 1992.

Hau95. D. Haussler. Sphere packing numbers for subsets of the Boolean n-cube with bounded Vapnik-Chervonenkis dimension. *Journal of Combinatorial Theory, Series A*, 69(2):217–232, 1995.

HMP+93. A. Hajnal, W. Maass, P. Pudlák, M. Szegedy, and G. Turán. Threshold circuits of bounded depth. *Journal of Computer and System Sciences*, 46:129–154, 1993.

KS94. M. J. Kearns and R. E. Schapire. Efficient distribution-free learning of probabilistic concepts. *Journal of Computer and System Sciences*, 48(3):464–497, 1994.

KT94. P.G. Kolaitis and M.N. Thakur. Logical definability of NP optimization problems. *Information and Computation*, 115(2):321–353, 1994.

LLS00. Y. Li, P. M. Long, and A. Srinivasan. Improved bounds on the sample complexity of learning. *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 309–318, 2000.

Lov75. L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

MR95. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

Nat89. B. K. Natarajan. On learning sets and functions. *Machine Learning*, 4:67–97, 1989.

NR95. M. Naor and R. M. Roth. Optimal file sharing in distributed networks. *SIAM J. Comput.*, 24(1):158–183, 1995.

PA95. J. Pach and P.K. Agarwal. *Combinatorial Geometry*. John Wiley and Sons, 1995.

Pol84. D. Pollard. *Convergence of Stochastic Processes*. Springer Verlag, 1984.

Qui99. J.R. Quinlan. Some elements of machine learning. *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 523–524, 1999.

Rag88. P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

Ram30. F.P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematics Society*, 30(2):264–286, 1930.

Roy91. S. Roy. Semantic complexity of relational queries and data independent data partitioning. *Proceedings of the ACM SIGACT-SIGART-SIGMOD Annual Symposium on Principles of Database Systems*, 1991.

RT87. P. Raghavan and C. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

SAB93. J. Shawe-Taylor, M. Anthony, and N. Biggs. Bounding the sample size with the Vapnik-Chervonenkis dimension. *Discrete Applied Mathematics*, 42:65–73, 1993.

Sch90. R. Schapire. The strength of weak learnability. *Machine Learning*, 5:197–227, 1990.

SFBL98. Robert E. Schapire, Yoav Freund, Peter Bartlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. *The Annals of Statistics*, 26(5):1651–1686, 1998.

Sri99. A. Srinivasan. Improved approximation guarantees for packing and covering integer programs. *SIAM Journal on Computing*, 29:648–670, 1999.

Sri01. A. Srinivasan. New approaches to covering and packing problems. *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2001.

Tal94. M. Talagrand. Sharper bounds for Gaussian and empirical processes. *Annals of Probability*, 22:28–76, 1994.

Val84. L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.

Vap82. V. N. Vapnik. *Estimation of Dependencies based on Empirical Data*. Springer Verlag, 1982.

Vap89. V. N. Vapnik. Inductive principles of the search for empirical dependences (methods based on weak convergence of probability measures). *Proceedings of the 1989 Workshop on Computational Learning Theory*, 1989.

VC71. V. N. Vapnik and A. Y. Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and its Applications*, 16(2):264–280, 1971.

On the Complexity of Scheduling Conditional Real-Time Code

Samarjit Chakraborty, Thomas Erlebach, and Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
ETH Zürich, ETH Zentrum, CH-8092 Zürich, Switzerland
{samarjit,erlebach,thiele}@tik.ee.ethz.ch

Abstract. Many real-time embedded systems involve a collection of independently executing event-driven code blocks, having hard real-time constraints. Portions of such codes when triggered by external events require to be executed within a given deadline from the triggering time. The feasibility analysis problem for such a real-time system asks whether it is possible to schedule all such blocks of code so that all the associated deadlines are met even in the worst case triggering sequence. Each such conditional real-time code block can be naturally represented by a directed acyclic graph whose vertices correspond to portions of code having a straight-line flow of control and are associated with execution requirements and deadlines relative to their triggering times, and the edges represent conditional branches. Till now, no complexity results were known for the feasibility analysis problem in this model, and all existing algorithms in the real-time systems literature have an exponential complexity. In this paper we show that this problem is NP-hard under both dynamic and static priorities in the preemptive uniprocessor case, even for a set of only two task graphs. For dynamic-priority feasibility analysis we give a pseudo-polynomial time exact algorithm and a fully polynomial-time approximation scheme for approximate feasibility testing. For the special case where all the execution requirements of the vertices are identical, we present a polynomial time exact algorithm. For static-priority feasibility analysis, we introduce a new sufficient condition and give a pseudo-polynomial time algorithm for checking it. This algorithm gives tighter results for feasibility analysis compared to those known so far.

1 Introduction

Over the years there have been several efforts to correctly model real-time systems and answer scheduling-theoretic questions arising in these models. All of the resulting models are based on an abstract framework in which a real-time system is modelled as a collection of independent *tasks*. Each task generates a sequence of *jobs*, each of which is characterized by a *ready-time*, an *execution requirement*, and a *deadline*. *Hard-real-time systems* require that for each job generated by a task, an amount of processor time equal to the job's execution

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 38–49, 2001.

requirement be assigned to it between its ready-time and its deadline. The feasibility analysis of such a hard-real-time task set is concerned with determining whether it is possible to schedule all the jobs generated by the tasks, such that they meet their deadlines under all possible circumstances.

In the context of most real-time embedded systems, each such real-time task is required to model an event-driven block of code, parts of which are triggered by external events and require to be executed within a given deadline from the triggering time. A natural representation of such a task is a directed acyclic graph whose vertices represent portions of code having a straight-line flow of control, and the edges represent possible conditional branches. The vertices are triggered by external events and have to be executed within their associated deadlines. The feasibility analysis of such a set of task graphs answers whether it is possible to schedule all the graphs so that all the associated deadlines are met even in the worst case triggering sequence. The difficulty of such an analysis lies in the fact that what constitutes a worst case triggering sequence for an individual graph can not be determined in isolation, due to the presence of the conditional branches. To illustrate this, consider the following example taken from [2].

while (*external event*) **do**
  execute code block B_0 {having execution time e_0 and deadline d_0}
  if (C) **then**
    execute code block B_1 {execution time e_1, deadline d_1}
  else
    execute code block B_2 {execution time e_2, deadline d_2}
  end if
end while

In the above block of code, if the condition C depends on some external event, or on the value of a variable which can not be determined at compile time, then the worst case branch here would depend on the other blocks of code executing concurrently with this one. Let $e_1 = 2$, $d_1 = 2$, $e_2 = 4$ and $d_2 = 5$. If another code block is simultaneously executing with $e = 1$ and $d = 1$ then the (e_1, d_1) branch corresponds to the worst case, whereas if $e = 2$ and $d = 5$ then the (e_2, d_2) branch corresponds to the worst case. Hence the usual method followed for the feasibility analysis of hard-real-time systems, of approximating a piece of code by its worst case behaviour does not work in the presence of conditional branches.

In this paper we consider the feasibility analysis of a collection of such code blocks with conditional branches and real-time constraints, and present a series of results on the complexity of various versions of this problem.

1.1 The Model

A task modelling a block of code is represented by a directed acyclic graph with a unique source and a unique sink vertex. Associated with each vertex v is its execution requirement $e(v)$ (which can be determined at compile time),

and deadline $d(v)$. Whenever the vertex v is *triggered*, the code corresponding to it has to be executed (which takes $e(v)$ amount of time) within the next $d(v)$ time units. Each directed edge (u, v) in the graph is associated with a minimum intertriggering separation $p(u, v)$, denoting the minimum amount of time that must elapse before the vertex v can be triggered after the triggering of the vertex u. This can be used to model a possible communication delay between u and v.

The semantics of the execution of such a task graph state that the source vertex can be triggered at any time, and once a vertex u is triggered then the next vertex v can be triggered only if there exists a directed edge (u, v) and at least $p(u, v)$ amount of time has elapsed since the triggering of u. If there are directed edges (u, v_1) and (u, v_2) from the vertex u then only one among v_1 and v_2 can be triggered, after the triggering of u. Therefore, a sequence of vertices $v_1, v_2, \ldots, v_k$ getting triggered at time instants $t_1, t_2, \ldots, t_k$ is legal if and only if there are directed edges (v_i, v_{i+1}) and $t_{i+1} - t_i \geq p(v_i, v_{i+1})$ for $i = 1, \ldots, k-1$. The real-time constraints require that the code corresponding to vertex v_i be executed within the time interval $(t_i, t_i + d(v_i)]$. Note that in general the condition $t_{i+1} \geq t_i + d(v_i)$ may not hold, i.e. a vertex can be triggered before the deadline of the last triggered vertex has elapsed. A consequence of this might be that the code corresponding to a vertex v is executed before that corresponding to a vertex u, although there exists a directed edge (u, v). Since this might not be allowable in most applications, throughout this paper we assume that $t_{i+1} \geq t_i + d(v_i)$ which is equivalent to requiring that $p(u, v) \geq d(u)$. Most of the previous work is based on this assumption and in the real-time systems literature this is referred to as the *frame separation property*.

Task sets and feasibility analysis. A task set $\mathcal{T} = \{T_1, T_2, \ldots, T_k\}$ consists of a collection of task graphs, the vertices of which can get triggered independently of each other. A triggering sequence for such a task set $\mathcal{T}$ is legal if and only if for every task graph T_i, the subset of vertices of the sequence belonging to T_i constitutes a legal triggering sequence for T_i. In other words, a legal triggering sequence for $\mathcal{T}$ is obtained by merging together (ordered by triggering times, with ties broken arbitrarily) legal triggering sequences of the constituting tasks.

The feasibility analysis of a task set $\mathcal{T}$ is concerned with determining whether for all possible legal triggering sequences of $\mathcal{T}$, the codes corresponding to the vertices of the task graphs can be scheduled such that all their associated deadlines are met. In this paper we consider the preemptive uniprocessor version of this problem.

Many scheduling algorithms are implemented by assigning priorities at each time instant (according to some criteria), to all jobs that are ready to execute and then allocating the processor to the highest priority job. Based on this, scheduling algorithms can be broadly classified into either *dynamic-priority* or *static-priority* (also known as fixed-priority) algorithms. Dynamic-priority algorithms allow the switching of priorities between tasks. This means that for two tasks, both having ready jobs at two time instants, at one instant the first task's job might have a higher priority than the second task's job, while at the other instant the priorities might switch. Static-priority algorithms, in contrast to this, do not allow such

priority switching. Here we will be concerned with both dynamic- and static-priority algorithms.

1.2 Our Results

The task model considered in this paper, apart from being of independent interest, forms the core of the *recurring real-time task model* very recently proposed by Baruah in [2,4]. This model is especially suited for accurately modelling conditional real-time code with recurring behaviour, i.e. where code blocks run in an infinite loop, and generalizes many of the previous well known models like the sporadic [8], multiframe [9], generalized multiframe [5], and recurring branching [3]. All of these previous models can be shown [2] to be special cases of the recurring real-time task model. However, the algorithms presented in [2] for the feasibility analysis problem in this model for the preemptive uniprocessor case, both with dynamic and static priorities, have a running time which is exponential in the number of vertices of the task graphs. It was also remarked that the feasibility analysis problem for this model is 'likely to be intractable', and in contrast to the previous (less general) models, no longer runs in pseudo-polynomial time.

The main contribution of this paper is that it answers all the questions raised in [2] and thereby settles the complexity of the feasibility analysis problem for scheduling conditional real-time code. For the ease of presentation, the model we consider here is slightly simpler than that of [2] in the sense that we do not consider the recurring behaviour of the task graphs. We postpone the details of how the results derived here for this simpler model can be extended to the recurring real-time task model, to a full version of this paper. Firstly, we show that the feasibility analysis problem, both for dynamic and static priorities, is NP-hard. For the dynamic-priority feasibility analysis we give a pseudo-polynomial time exact algorithm and a fully polynomial-time approximation scheme for approximate feasibility testing. We also show that for the special case where all the vertices of a task graph have equal execution requirements, this problem can be solved in polynomial time.

For static-priority feasibility analysis Baruah had introduced a sufficient condition in [2]. We give a tighter condition for sufficiency and show that this can be checked in pseudo-polynomial time. Further, our condition is simpler than that of [2]. Our results imply that for all practical purposes the feasibility analysis problem in the recurring real-time task model is efficiently solvable.

We present the hardness results in Section 2. In Section 3 we present the algorithms for dynamic-priority feasibility analysis, followed by those for static-priority feasibility analysis in Section 4. Due to space restrictions all proofs are omitted here; we refer the interested reader to [7].

2 NP-Hardness of Feasibility Analysis

In this section we obtain that both the dynamic- and static-priority feasibility analysis problems for our task model, and therefore for the recurring real-time

task model as well, are NP-hard for the preemptive uniprocessor case. Our proofs rely on a reduction from the knapsack problem which is known to be NP-hard.

Theorem 1. *The dynamic-priority feasibility analysis problem for the task model described in Section 1.1 in a preemptive uniprocessor environment is NP-hard.*

The next result shows that the static-priority feasibility analysis problem is NP-hard. The pseudo-polynomial time algorithm that we present later for this problem, and also the algorithm presented in [2], are based on testing whether a given task from a task set is *lowest-priority feasible*. A task $T \in \mathcal{T}$ is lowest-priority feasible if and only if all the vertices of T can always meet their deadlines with T assigned the lowest priority and all the remaining tasks of $\mathcal{T}$ having any arbitrary priority assignment. The existence of a lowest-priority feasible task in any static-priority feasible task set is given later by Theorem 6.

The next theorem says that the lowest-priority feasibility testing problem is NP-hard, and as a corollary of this it follows that static-priority feasibility analysis is also NP-hard.

Theorem 2. *The problem of determining whether a given task is lowest-priority feasible is NP-hard.*

Corollary 1. *The static-priority feasibility analysis problem is NP-hard.*

3 Dynamic-Priority Feasibility Analysis

A necessary and sufficient condition for the dynamic-priority feasibility of the recurring real-time task model was stated in [2]. It was stated without proof that the condition follows from the *processor demand criterion* introduced in [6]. It is possible to give a simple independent proof [7] showing that the same condition works for our model. It is based on an abstraction of a task, represented by a function called the *demand-bound function*. The demand-bound function of a task T, denoted by $T.dbf(t)$, takes as an argument a real number t and returns the maximum possible cumulative execution requirement by vertices of T that have been triggered by a legal triggering sequence and have both their ready times and deadlines within a time interval of length t. Intuitively, $T.dbf(t)$ denotes the maximum possible execution requirement that can possibly be demanded by T within any time interval of length t, if all its vertices are to meet their deadlines.

Theorem 3. *A task set $\mathcal{T}$ is dynamic-priority feasible if and only if for all $t \geq 0$, $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$.*

We next show that the problem of computing $T.dbf(t)$ for a task T is NP-hard and then give a FPTAS for approximating it, which immediately leads to an approximate decision algorithm for the feasibility analysis problem.

Theorem 4. *The problem of computing $T.dbf(t)$ is NP-hard.*

Given a task graph T we first give a pseudo-polynomial time algorithm for computing $T.dbf(t)$ for any $t \geq 0$, based on dynamic programming. Let there be n vertices in T denoted by $v_1, \ldots, v_n$, and without any loss of generality we assume that there can be a directed edge from v_i to v_j only if $i < j$. Following our notation described in Section 1.1, associated with each vertex v_i is its execution requirement $e(v_i)$ which here is assumed to be integral (a pseudo-polynomial algorithm is meaningful only under this assumption), and its deadline $d(v_i)$. Associated with each edge (v_i, v_j) is the minimum intertriggering separation $p(v_i, v_j)$.

Let $t_{i,e}$ be the minimum time interval within which the task T can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, \ldots, v_i\}$, if all the triggered vertices are to meet their respective deadlines. Let $t^i_{i,e}$ be the minimum time interval within which a sequence of vertices from the set $\{v_1, \ldots, v_i\}$, and ending with the vertex v_i, can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. Lastly, let $E = \max_{i=1,\ldots,n} e(v_i)$. Clearly, nE is an upper bound on $T.dbf(t)$ for any $t \geq 0$. It can be trivially shown by induction that Algorithm 1 correctly computes $T.dbf(t)$, and has a running time of $O(n^3E)$.

Algorithm 1 Computing $T.dbf(t)$

Input: Task graph T, and a real number $t \geq 0$

for $e \leftarrow 1$ to nE **do**

$$t_{1,e} \leftarrow \begin{cases} d(v_1) & \text{if } e(v_1) = e \\ \infty & \text{otherwise} \end{cases}$$

$t^1_{1,e} \leftarrow t_{1,e}$

end for

for $i \leftarrow 1$ to $n-1$ **do**

for $e \leftarrow 1$ to nE **do**

Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}

$$t^{i+1}_{i+1,e} \leftarrow \begin{cases} \min\{t^{i_j}_{i_j, e-e(v_{i+1})} - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + d(v_{i+1}) \mid j = 1, \ldots, k\} \\ \text{if } e(v_{i+1}) < e, \quad d(v_{i+1}) \text{ if } e(v_{i+1}) = e, \quad \text{and} \quad \infty \text{ otherwise} \end{cases}$$

$t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$

end for

end for

$T.dbf(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$

Given this algorithm, any $t \geq 0$, and an $0 < \varepsilon \leq 1$, let T_t be the subgraph of T consisting only of those vertices v_i for which $d(v_i) \leq t$, and let E_t denote the maximum execution requirement of a vertex from among all vertices of T_t. Now we scale all the execution requirements associated with the vertices of T_t by $K = \varepsilon E_t/n$ i.e. $e'(v_i) = \lfloor e(v_i)/K \rfloor$ and run the algorithm with the new $e'(v_i)$s and the graph T_t. By using the same arguments as in the FPTAS for the knapsack problem, it is possible to show that for any $t \geq 0$, the algorithm outputs a value $\geq (1-\varepsilon)T.dbf(t)$ and runs in time $O(n^4/\varepsilon)$, and is therefore an FPTAS for computing $T.dbf(t)$. We denote the result computed by this algorithm by $T.dbf'(t)$.

For our approximate decision algorithm, note that for all $t \geq 0$, there can be at most n distinct values of E_t for any task graph. For each such E_t, we consider the corresponding subgraph that gives rise to this E_t as described above, and scale the execution requirements of the vertices of this subgraph by $K = \varepsilon E_t/n$. In each such subgraph T_t, the number of values of time intervals t' at which the value of $T_t.dbf'(t')$ changes is bounded by $O(n^2/\varepsilon)$, and hence the number of values of time intervals t at which the value of $\sum_{T\in\mathcal{T}} T.dbf'(t)$ changes is bounded by $O(|\mathcal{T}|n^3/\varepsilon)$. Our fully polynomial time approximate decision algorithm for dynamic-priority feasibility analysis is now given as Algorithm 2.

Algorithm 2 Approximate decision algorithm for feasibility analysis

Input: Task set $\mathcal{T}$ and a real $0 < \varepsilon \leq 1$
  decision $\leftarrow$ *YES*
  for all values of t at which $T.dbf'(t)$ changes for any $T \in \mathcal{T}$ **do**
    if $\frac{1}{1-\varepsilon}\sum_{T\in\mathcal{T}} T.dbf'(t) > t$ **then** {Condition $(*)$}
      decision $\leftarrow$ *NO*
    end if
  end for
  return *decision*

Theorem 5. *If a task set $\mathcal{T}$ is infeasible then Algorithm 2 always returns the correct answer. If $\mathcal{T}$ is feasible and $t \geq \frac{1}{1-\varepsilon}\sum_{T\in\mathcal{T}} T.dbf'(t)$ for all values of t, then the algorithm always returns the correct answer YES, otherwise it might return a NO. YES answers are always correct. The running time of the algorithm is $O(|\mathcal{T}|^2 n^5 \varepsilon^{-2} \log n)$, if all task graphs have $O(n)$ vertices.*

For each task T, computing the $t_{n,e}$ values for each of its subgraphs T_t, using Algorithm 1 and the scaled execution requirements requires $O(n^4/\varepsilon)$ time, and these values are stored in a table. Hence computing all such values for all the task graphs in $\mathcal{T}$ takes $O(n^5|\mathcal{T}|/\varepsilon)$ time. For each value of t for which $\sum_{T\in\mathcal{T}} T.dbf'(t)$ changes, computing $T.dbf'(t)$ for any $T \in \mathcal{T}$ requires a binary search to identify the appropriate table corresponding to a subgraph T_t, and then a linear search through the table. Therefore, computing the value of $\sum_{T\in\mathcal{T}} T.dbf'(t)$ for any value of t takes $O(|\mathcal{T}|n^2\varepsilon^{-1}\log n)$ time. Hence the total running time of Algorithm 2 is $O(|\mathcal{T}|^2 n^5 \varepsilon^{-2} \log n)$. The algorithm is overly pessimistic in the sense that for certain feasible task sets it might return a NO. However, for task sets which can be in some sense comfortably scheduled even in the worst case, leaving some idle processor time (which can be parameterized by ε), the algorithm always returns a YES. Therefore, any ε characterizes a class of task sets for which the algorithm errs. Decreasing ε reduces this class of such task sets for which the algorithm errs, at the cost of increasing the running time quadratically in $1/\varepsilon$, thereby giving a fully polynomial-time approximate decision scheme for approximate feasibility testing.

It may be noted that changing Condition $(*)$ in Algorithm 2 to

if $\sum_{T\in\mathcal{T}} T.dbf'(t) > t$ **then**
  decision $\leftarrow$ *NO*
end if

will result in an overly optimistic algorithm which might incorrectly return a YES for certain classes of infeasible task sets. For all feasible task sets it always returns YES, and NO answers are always correct. The task sets for which the algorithm might err are those in which the cumulative execution requirement by tasks of $\mathcal{T}$ within any time interval of length t exceeds the maximum execution requirement that can be feasibly scheduled, by an amount of less than $\varepsilon \sum_{T \in \mathcal{T}} T.dbf(t)$ time units. Again, decreasing ε reduces the class of such task sets, at the cost of the running time increasing linearly in $1/\varepsilon$.

Lastly, it might be noted that Theorem 3 along with Algorithm 1 also imply a pseudo-polynomial time algorithm for dynamic-priority feasibility analysis. To see this, let for any task $T \in \mathcal{T}$, t^T_{max} denote the maximum amount of time elapsed among all execution sequences starting from the source vertex of T and ending at the sink vertex, if every vertex is triggered at the earliest possible time (respecting the minimum intertriggering separations). Let $t_{max} = \max_{T \in \mathcal{T}} t^T_{max}$. It follows from Theorem 3 that $\mathcal{T}$ is dynamic-priority feasible if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $t = 1, \ldots, t_{max}$. $T.dbf(t)$ for any t can be determined in pseudo-polynomial time by Algorithm 1 and clearly, t_{max} is pseudo-polynomially bounded, implying a pseudo-polynomial algorithm for dynamic-priority feasibility analysis.

3.1 Vertices with Equal Execution Requirements

We now show that for the special case where for every task T belonging to a task set $\mathcal{T}$, all the vertices of T have equal execution requirements, the feasibility analysis problem for $\mathcal{T}$ can be solved in polynomial time. This result holds even when all execution requirements and deadlines take values over the reals.

We denote the vertices of a task graph T by $v_1, \ldots, v_n$ and assume that there can be a directed edge from v_i to v_j only if $i < j$. Let $t_{i,k}$ denote the minimum time interval within which exactly k vertices of T from the set $\{v_1, \ldots, v_i\}$ (obviously $k \leq i$) need to be executed as a result of some legal triggering sequence, if they have to meet their associated deadlines. Let $t^i_{i,k}$ denote the minimum time interval within which exactly k vertices of T consisting of v_i and any other $k-1$ vertices from $\{v_1, \ldots, v_{i-1}\}$ need to be executed as a result of some legal triggering sequence, if they have to meet their associated deadlines.

Given any vertex v_i of T, let there be directed edges from the vertices $v_{i_1}, \ldots, v_{i_l}$ to v_i. Then for any $k \leq i$,

$$t^i_{i,k} = \min\{t^{i_j}_{i_j,k-1} - d(v_{i_j}) + p(v_{i_j}, v_i) + d(v_i) \mid j = 1, \ldots, l\} \text{ (and } d(v_i) \text{ if } k = 1)$$

$$t_{i,k} = \min\{t_{i-1,k}, t^i_{i,k}\}$$

Using the fact that $t_{1,1} = t^1_{1,1} = d(v_1)$, it is now possible to compute any $t_{i,k}$ within at most $O(n^3)$ time, where n is the number of vertices in the task graph. Now, if each task graph $T \in \mathcal{T}$ has n_T vertices then let us consider the set $S = \bigcup_{T \in \mathcal{T}} \bigcup_{i=1,\ldots,n_T} \{t_{n_T,i} \text{ for task graph } T\}$. If each vertex of task graph T has an execution requirement of e, then for ant $t \geq 0$, $T.dbf(t) = \max\{ie \mid t_{n_T,i} \leq t\}$. Clearly, the task set $\mathcal{T}$ is feasible if and only if $\sum_{T \in \mathcal{T}} T.dbf(t) \leq t$ for all $t \in S$.

Computing all the necessary dbf values for each task graph and storing them in a table takes $O(n^3)$ time if the number of vertices in any task graph is $O(n)$. Since there are $|\mathcal{T}|$ task graphs, this whole process takes $O(|\mathcal{T}|n^3)$ time. For each value of t, verifying whether the sum of the dbfs exceeds t requires a search through the previously computed tables and takes $O(|\mathcal{T}| \log n)$ time. Since there are $O(|\mathcal{T}|n)$ values of t for which this has to be verified, this takes $O(|\mathcal{T}|^2 n \log n)$ time. Hence the total run time is bounded by $O(|\mathcal{T}|n^3 + |\mathcal{T}|^2 n \log n)$.

4 Static-Priority Feasibility Analysis

The static-priority feasibility analysis of a task set $\mathcal{T}$ is concerned with determining whether there exists an assignment of priorities to the tasks of $\mathcal{T}$ under which they can be scheduled by a static-priority run time scheduler so that all deadlines are met even in the worst case triggering sequence. Any such priority assignment is defined to be a *good* static-priority assignment for $\mathcal{T}$. As mentioned in Section 2 solving this feasibility analysis problem is based on testing whether a given task $T \in \mathcal{T}$ is lowest-priority feasible. Clearly, if there is a procedure for testing lowest-priority feasibility, and the task set $\mathcal{T}$ is static-priority feasible, then $|\mathcal{T}|$ calls to this procedure will be sufficient to identify a lowest-priority feasible task of $\mathcal{T}$. Therefore, if $|\mathcal{T}| = n$ then with $O(n^2)$ calls to this procedure a good static-priority assignment for $\mathcal{T}$ can be determined based on the following theorem.

Theorem 6 (Audsley, Tindell, Burns [1]). *Suppose a task $T \in \mathcal{T}$ is lowest-priority feasible. Then there is a good static-priority assignment for $\mathcal{T}$ if and only if there is a good static-priority assignment for $\mathcal{T} \backslash \{T\}$.*

An algorithm for static-priority feasibility analysis therefore reduces to devising an algorithm for lowest-priority feasibility testing. An algorithm implementing a sufficient condition for lowest-priority feasibility was given by Baruah in [2] for the recurring real-time task model. It is also based on an abstraction of a task, similar to the demand-bound function presented in Section 3, and uses a function called the *request-bound function*. The request-bound function of a task T, denoted by $T.rbf(t)$, takes as an argument a real number t and returns the maximum possible cumulative execution requirement by vertices of T that have been triggered according to some legal triggering sequence and have their ready times within any time interval of length t. Intuitively, $T.rbf(t)$ is an upper bound on the maximum amount of time, within any time interval of length t, for which T can deny the processor to all lower-priority tasks. Based on this function, the following sufficiency condition was given for lowest-priority feasibility testing in [2].

Theorem 7 (Baruah [2]). *A task $T \in \mathcal{T}$ is lowest-priority feasible if $\forall t : \exists t' \leq t$ such that $t' - \sum_{T' \in \mathcal{T} \backslash \{T\}} T'.rbf(t') \geq T.dbf(t)$.*

For any task $T \in \mathcal{T}$, in our task model described in Section 1.1, let t^T_{max} be as described in Section 3. Clearly, T is lowest-priority feasible if the condition given by Theorem 7 is satisfied for all values of $t = 1, \ldots, t^T_{max}$. Although t_{max}

is pseudo-polynomially bounded by the representation of $\mathcal{T}$, the algorithm for computing $T.rbf(t)$ for any t and $T \in \mathcal{T}$ as given in [2] runs in time which is exponential in the number of vertices of T.

We first obtain that the problem of computing $T.rbf(t)$ is NP-hard and then give a modified request-bound function, which we denote by $T.rbf'(t)$, and give a pseudo-polynomial time algorithm for computing it for any value of $t \geq 0$ based on dynamic programming. Using $rbf'(t)$ we then give a new sufficiency condition for testing lowest-priority feasibility. This gives a tighter test compared to that of Theorem 7 in the following sense: for any task set $\mathcal{T}$, if a task $T \in \mathcal{T}$ is returned as lowest-priority feasible by the test in Theorem 7 then it is also returned as lowest-priority feasible by our test, and there exist task sets $\mathcal{T}$ and tasks $T \in \mathcal{T}$ which although being lowest-priority feasible, fail the test in Theorem 7 but are returned as lowest-priority feasible by our test. Lastly, we show that for any task set consisting of exactly two tasks, our test is both a necessary and sufficient condition.

Theorem 8. *The problem of computing $T.rbf(t)$ is NP-hard.*

Our new $T.rbf'(t)$ is similar to $T.rbf(t)$ and returns the maximum possible cumulative execution requirement by vertices of T within any time interval of length t, that have been triggered by a legal triggering sequence. To illustrate the difference between the two functions, consider a task graph T consisting of a single vertex having an execution requirement of 5 and any arbitrary deadline. Whereas $T.rbf(t) = 5$ for any $t \geq 0$ (since the ready time of T is at time 0), $T.rbf'(t) = t$ for $t \leq 5$ and is equal to 5 for any $t > 5$.

Following the notation used in Section 3, given a task graph T, let $t_{i,e}$ denote the minimum time interval within which T can have an execution requirement of exactly e time units due to some legal triggering sequence, considering only a subset of vertices from the set $\{v_1, \ldots, v_i\}$. Let $t^i_{i,e}$ be the minimum time interval within which any execution sequence consisting of vertices from the set $\{v_1, \ldots, v_{i-1}\}$ and ending with the vertex v_i can have an execution requirement of exactly e time units. Now recall the definition of $t^i_{i,e}$ as used in Section 3 for computing $T.dbf(t)$, which is the minimum time interval within which a sequence of vertices from the set $\{v_1, \ldots, v_i\}$, and ending with the vertex v_i can have an execution requirement of exactly e time units, if all the vertices have to meet their respective deadlines. This we denote here by $dbf^i_i(e)$. We assume, as in Section 3, that T consists of n vertices $v_1, \ldots, v_n$ and that there can be a directed edge from v_i to v_j only if $i < j$, and that all the execution requirements are integral. If $E = \max_{i=1,\ldots,n} e(v_i)$, then Algorithm 3 correctly computes $T.rbf'(t)$ and has a running time of $O(n^3E^2)$. Our new sufficiency condition for lowest-priority feasibility is based on the following lemma.

Lemma 1. *Let $T \in \mathcal{T}$ and the task graph corresponding to T have n vertices $v_1, \ldots, v_n$. If each of these vertices v_i is lowest-priority feasible in the task set $\mathcal{T} \backslash \{T\} \bigcup \{v_i\}$, then T is also lowest-priority feasible.*

Algorithm 3 Computing $T.rbf'(t)$

Input: Task graph T, and a real number $t \geq 0$

for $e \leftarrow 1$ to nE **do**

$$t_{1,e} \leftarrow \begin{cases} e & \text{if } e \leq e(v_1) \\ \infty & \text{if } e > e(v_1) \end{cases}$$

$t^1_{1,e} \leftarrow t_{1,e}$

end for

Computing $t_{i+1,e}$**:**

Let there be directed edges from the vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$ to v_{i+1}

Let $t^{i_j,i+1}_{i+1,e}(l) \leftarrow dbf^{i_j}_{i_j}(e - e(v_{i+1}) + l) - d(v_{i_j}) + p(v_{i_j}, v_{i+1}) + e(v_{i+1}) - l$

Let $t^{i_j,i+1}_{i+1,e} \leftarrow \min\{t^{i_j,i+1}_{i+1,e}(l) \mid l = 0, \ldots, e(v_{i+1}) - 1\}$

$t^{i+1}_{i+1,e} \leftarrow \min\{t^{i_j,i+1}_{i+1,e} \mid j = 1, \ldots, k\}$

$t_{i+1,e} \leftarrow \min\{t_{i,e}, t^{i+1}_{i+1,e}\}$

$T.rbf'(t) \leftarrow \max\{e \mid t_{n,e} \leq t\}$

Theorem 9. *A task $T \in \mathcal{T}$ is lowest-priority feasible if for all vertices v belonging to the task graph of T, $\exists 0 \leq t \leq d(v)$ for which $t - \sum_{T' \in \mathcal{T} \setminus \{T\}} T'.rbf'(t) \geq e(v)$.*

It is easy to see that if a task $T \in \mathcal{T}$ is returned as lowest-priority feasible by the test given by Theorem 7 then it also passes the test of Theorem 9. Additionally, if T is returned as lowest-priority feasible, then it is really so. To show that this represents a tigher test, consider a task set consisting of two task graphs T_1 and T_2. T_1 is a simple chain of three vertices with the first two vertices having their execution requirements equal to 1 and deadlines equal to 2, and the third vertex having an execution requirement of 3 and deadline equal to 6. The intertriggering separation on any directed edge (u, v) is equal to the deadline of u. T_2 consists of a single vertex having an execution requirement of 1 and deadline equal to 4. It can be seen that T_2 is indeed lowest-priority feasible and passes the test of Theorem 9, but fails the test given by Theorem 7. Lastly, we show that for any set of exactly two task graphs, the test given by Theorem 9 is both a necessary and sufficient condition.

Theorem 10. *For any task set $\mathcal{T}$ consisting of exactly two task graphs, a task $T \in \mathcal{T}$ is lowest-priority feasible if and only if it satisfies the test given by Theorem 9.*

It now follows from Theorem 6, Algorithm 3, and Theorem 9 that there exists a pseudo-polynomial algorithm for static-priority feasibility analysis that implements the sufficiency condition stated by Theorem 9. Further, the same approach of scaling the execution requirements associated with the vertices as described in Section 3 for the dynamic-priority feasibility analysis, and then using Algorithm 3 with the scaled values will give a polynomial time approximate decision algorithm for static-priority feasibility analysis. Lastly, in the case where for each task graph all the vertices have equal execution times, this problem can also be solved in polynomial time. We skip the details of any of these in this paper.

5 Concluding Remarks

This paper settles the complexity of the feasibility analysis problem involved in scheduling a collection of code blocks with conditional branches and real-time constraints. In particular it shows that although the feasibility analysis of the recently introduced recurring real-time task model is NP-hard, there exists a pseudo-polynomial time exact algorithm and a fully polynomial-time approximation scheme for solving it. All the results presented here pertain to the preemptive uniprocessor version of this problem. It would be natural to extend these results to the non-preemptive and different multiprocessor cases. Following [2], our algorithms were based on an abstraction of a task represented by the demand-bound and the request-bound functions, which in some sense captured the worst case behaviour of a task. It seems unlikely that this same approach might work for any non-preemptive or multiprocessor case, except for very restricted classes of tasks such as those where all vertices have unit execution requirements and time is integral. In more general cases, the worst case triggering sequence of the vertices of a task graph as identified by the demand- or request-bound functions need not be the worst case when the issue of feasibly packing these jobs (on multiple processors, for example) is taken into account.

References

1. N.C. Audsley, K.W. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proc. Euromicro Conf. on Real-Time Systems*, Finland, 1993. IEEE Computer Society Press.
2. S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. To appear in Real-Time Systems.
3. S. Baruah. Feasibility analysis of recurring branching tasks. In *Proc. 10th Euromicro Workshop on Real-Time Systems*, pages 138–145, 1998.
4. S. Baruah. A general model for recurring real-time tasks. In *Proc. Real-Time Systems Symposium*, pages 114–122. IEEE Computer Society Press, 1998.
5. S. Baruah, D. Chen, S. Gorinsky, and A.K. Mok. Generalized multiframe tasks. *Real-Time Systems*, 17(1):5–22, 1999.
6. S. Baruah, R.R. Howell, and L.E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2:301–324, 1990.
7. S. Chakraborty, T. Erlebach, and L. Thiele. On the complexity of scheduling conditional real-time code. Technical Report TIK Report No. 107, ETH Zürich, 2001. `ftp://ftp.tik.ee.ethz.ch/pub/people/samarjit/paper/CET01a.ps.gz`.
8. A.K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, Laboratory for Computer Science, MIT, 1983. Available as Technical Report No. MIT/LCS/TR-297.
9. A.K. Mok and D. Chen. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering*, 23(10):635–645, 1997.

Time Responsive External Data Structures for Moving Points

Pankaj K. Agarwal[1⋆], Lars Arge[1⋆⋆], and Jan Vahrenhold[2⋆⋆⋆]

[1] Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708, USA
{pankaj,large}@cs.duke.edu

[2] Westfälische Wilhelms-Universität Münster, Institut für Informatik, 48149 Münster, Germany
jan@math.uni-muenster.de

Abstract. We develop external data structures for storing points in one or two dimensions, each moving along a linear trajectory, so that a range query at a given time t_q can be answered efficiently. The novel feature of our data structures is that the number of I/Os required to answer a query depends not only on the size of the data set and on the number of points in the answer but also on the difference between t_q and the current time; queries close to the current time are answered fast, while queries that are far away in the future or in the past may take more time.

1 Introduction

I/O-communication, and not internal memory computation time, is often the bottleneck in a computation when working with datasets larger than the available main memory. Recently, external geometric data structures have received considerable attention because massive geometric datasets arise naturally in many applications (see [19,5] and references therein). The need for storing and processing continuously moving data arises in a wide range of applications, including air-traffic control, digital battlefields, and mobile communication systems. Most of the existing database systems, which assume that the data is constant unless it is explicitly modified, are not suitable for representing, storing, and querying continuously moving objects because either the database has to be continuously updated or a query output will be obsolete. A better approach would be to represent the position of a moving object as a function $f(t)$ of time, so that the position changes without any explicit change in the database system and so that the database needs to be updated only when the function $f(t)$ changes (e.g., when the velocity of the object changes).

⋆ Supported in part by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants ITR–333–1050, EIA–9870734, EIA–997287, and CCR–9732787, and by grant from the U.S.-Israeli Binational Science Foundation.

⋆⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA-9972879 and CAREER grant EIA-9984099.

⋆⋆⋆ Part of this work was done while visiting Duke University.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 50–61, 2001.

In this paper, we focus on developing efficient external data structures for storing a set of moving points in one- or two-dimensional space so that range queries over their (future or past) locations can be answered quickly. Our focus is on what we call *time responsive* data structures that have fast response time for near-future or near-past queries but may take more time for queries that are far away in time. Time responsiveness is important in, e.g., air-traffic control, where queries in the near future are more critical than queries far away in the future.

1.1 Problem Statement

Let $S = \{p_1, p_2, \ldots, p_N\}$ be a set of moving points in $\mathbb{R}^d$, $d = 1, 2$. For any time t, let $p_i(t)$ denote the position of p_i at time t, and let $S(t) = \{p_1(t), \ldots, p_N(t)\}$. We will assume that each point p_i is moving along a straight line at some fixed speed, that is, $p_i(t) = \mathbf{a}_i \cdot t + \mathbf{b}_i$ for some $\mathbf{a}_i, \mathbf{b}_i \in \mathbb{R}^d$. We will use t_{now} to denote the current time. We are interested in answering queries of the following form:

Q1. Given a set S of points moving along the y-axis, a y-range $R = [y_1, y_2]$, and a time t_q, report all points of S that lie inside R at time t_q, that is, $S(t_q) \cap R$.

Q2. Given a set S of points moving in the xy-plane, an axis-aligned rectangle R, and a time t_q, report all points of S that lie inside R at time t_q, that is, $S(t_q) \cap R$.

As our main interest is minimizing the number of disk accesses needed to answer query, we will consider the problem in the standard external memory model; see e.g. [5]. This model assumes that each disk access transmits a contiguous block of B units of data in a single *input/output operation* (or *I/O*). The efficiency of a data structure is measured in terms of the amount of disk space it uses (in units of disk blocks) and the number of I/Os required to answer a range query. As we are interested in solutions that are *output sensitive*, our query I/O bounds are not only expressed in terms of N, the number of points in S, but also in terms of K, the number of points reported by the query. The minimum number of disk blocks we need to store N points is $\lceil N/B \rceil$, and at least $\lceil K/B \rceil$ I/Os are needed to report K output points. We refer to these bounds as "linear."

1.2 Previous Results

Recently, there has been a flurry of activity in computational geometry and databases on problems dealing with moving objects. In the computational geometry community, Basch *et al.* [7] introduced the notion of *kinetic data structures*. Their work led to several interesting internal memory results related to moving points; see [12] and references therein. The main idea in the kinetic framework is that even though the points move continuously, the relevant combinatorial structure of the data structure change only at certain discrete times. Therefore the data structure does not need to be updated continuously. Instead *kinetic updates* are performed on the data structure only when certain *kinetic events* occur.

These events have a natural interpretation in terms of the underlying structure. In contrast, in fixed-time-step methods, where the structure is updated at fixed time intervals, the fastest moving object determines an update time step for the entire data structure. Even though the kinetic framework often leads to very query efficient structures, one disadvantage of kinetic data structures is that queries can only be answered at the current time (i.e., in chronological order). Thus while kinetic data structures are very useful in simulation applications they are less suited for answering the types of queries we consider in this paper.

In the database community, a number of practical methods have been proposed for handling moving objects (see [20,16] and the references therein). Almost all of them require $\Omega(N/B)$ I/Os in the worst case to answer a Q1 or Q2 query—even if the query output size is $O(1)$. Kollios *et al.* [13] proposed the first provably efficient data structure, based on partition trees [2,3,15], for queries of type Q1. The structure uses $O(N/B)$ disk blocks and answers queries in $O((N/B)^{1/2+\epsilon} + K/B)$ I/Os, for any $\epsilon > 0$. Agarwal *et al.* [1] extended the result to Q2 queries. Kollios *et al.* [13] also present a scheme that answers a Q1 query using optimal $O(\log_B N + K/B)$ I/Os but using $O(N^2/B)$ disk blocks. These data structures are *time-oblivious*, that is, they do not evolve over time. Agarwal *et al.* [1] were the first to consider kinetic data structure in external memory. Based on external range trees [6], they developed a data structure that answers a Q2 query in optimal $O(\log_B N + K/B)$ I/Os using $O((N/B)\log_B N/(\log_B \log_B N))$ disk blocks—provided, as discussed above, that the queries arrive in chronological order. The amortized cost of a kinetic event is $O(\log_B^2 N)$ I/Os, and the total number of events is $O(N^2)$. They also showed how to modify the structure in order to obtain a tradeoff between the query bound and the number of kinetic events.

Agarwal *et al.* [1] were also the first to propose time responsive data structures in the context of moving points. They developed an $O(N/B)$ space structure for Q1 queries and a $O((N/B)\log_B N/(\log_B \log_B N))$ space structure for Q2 queries, where the cost of a query at time t_q is a monotonically increasing function of the difference between t_{now} and t_q. The query bound never exceeds $O((N/B)^{1/2+\epsilon} + K/B)$. They were only able to prove more specific bounds when the positions and velocities of the points are uniformly distributed inside a unit square.

1.3 Our Results

In this paper we combine the ideas utilized in time-oblivious and kinetic data structures in order to develop the first time responsive external data structures for Q1 and Q2 queries with provably efficient specific query bounds depending on the difference between t_{now} and t_q. Our structures evolve over time, but unlike previous kinetic structures, queries can be answered at any time in the future. Our data structures are of a combinatorial nature and we therefore measure time in terms of kinetic events. We define $\varphi(t)$ to be the number of kinetic events that occur (or have occurred) between t_{now} and t, and our query bounds will depend on $\varphi(t_q)$, the number of kinetic events between the current time and t_q. For

brevity, we will focus on queries in the *future*, i.e., $t_q \geq t_{now}$. The somewhat simpler case of queries in the *past* ($t_q \leq t_{now}$) can be handled similarly.

In Section 3, we describe a time responsive data structure for Q1 queries. A kinetic occurs when two points pass through each other (become equal) and their relative orderings change. Our data structure uses $O((N/B)\log_B N)$ disk blocks and answers a Q1 query with time stamp t_q such that $\varphi(t_q) \leq NB^{i-1}$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os. The expected amortized cost of a kinetic event is $O(\log_B^2 N)$ I/Os. Note that a query at a time t_q with $\varphi(t_q) \leq NB$ can be answered in optimal $O(\log_B N + K/B)$ I/Os. Previously such query efficient structures either used $O(N^2/B)$ space [13] or required the queries to arrive in chronological order [1]. Our data structure is considerably simpler than the one proposed in [1] and does not make any assumptions on the distribution of the trajectories in order to prove a bound on the query time.

In Section 4, we describe a time responsive data structure for Q2 queries. A kinetic event now occurs when the x- or y-coordinates of two points become equal. This data structure uses $O((N/B)\log_B N)$ disk blocks and answers a Q2 query with time stamp t_q such that $\varphi(t_q) \leq NB^i$ in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os. Each kinetic event is handled in $O(\log_B^3 N)$ expected I/Os. If $\varphi(t_q) \leq NB$ the query is answered in $O(\sqrt{N/B}\log_B N + K/B)$ I/Os, an improvement over previous $O((N/B)^{1/2+\epsilon} + K/B)$ I/O structures for non-chronological queries.

2 Preliminaries

Arrangements. Given a set S of N lines in $\mathbb{R}^2$, the *arrangement* $\mathcal{A}(S)$ is the planar subdivision whose vertices are the intersection points of lines, edges are the maximal portions of lines not containing any vertex, and faces are the maximal connected portions of the plane not containing any line of S. $\mathcal{A}(S)$ has $\Theta(N^2)$ vertices, edges, and faces. For each $1 \leq k \leq N$, the *k-level* $\mathcal{A}_k(S)$ of $\mathcal{A}(S)$ is defined as the closure of all edges in $\mathcal{A}(S)$ that have exactly k lines of S (strictly) below them. The k-level is a polygonal chain that is monotone with respect to the horizontal axis. Dey [10] showed that the maximum number of vertices on the k-level in an arrangement of N lines in the plane is $O(Nk^{1/3})$. Recently, Tóth proved a lower bound of $\Omega(N2^{\sqrt{\log k}})$ on the complexity of a k-level [17]. Using a result by Edelsbrunner and Welzl [11], Agarwal *et al.* [2] discussed how a given level of an arrangement of lines can be computed I/O-efficiently.

Lemma 1 (Agarwal *et al.* [2]). *A given level with T vertices in an arrangement of N lines can be computed in $O(N\log_2 N + T\log_2 N\log_B N)$ I/Os.*

B-trees. A *B-tree*, one of the most fundamental external data structures, stores N elements from an ordered domain using $O(N/B)$ disk blocks so that a one-dimensional range query can be answered in $O(\log_B N + K/B)$ I/Os [9]. An element can be inserted/deleted in $O(\log_B N)$ I/Os. A standard B-tree answers

queries only on set of elements currently in the structure. A *persistent* (or multi-version) B-tree, on the other hand, supports range queries in all states (*versions*) of the data structure in $O(\log_B N+K/B)$ I/Os [8,18]. Updates can be performed in $O(\log_B N)$ I/Os on the current structure, and we refer to the structure existing after T updates as the structure existing at *time* T.

Lemma 2 ([8,18]). *A persistent B-tree constructed by performing N updates using $O(\log_B N)$ I/Os each, uses $O(N/B)$ disk blocks and supports range queries at any time in $O(\log_B N + K/B)$ I/Os.*

3 Data Structure for Moving Points in $\mathbb{R}$

In this section we consider queries of type Q1. If we interpret time as the t-axis in the parametric ty-plane, each point in S traces out a line in this ty-plane. Abusing the notation a little, we will use S to denote the resulting set of N lines. A Q1 query then corresponds to reporting all lines of S that intersect a vertical segment (Figure 1 (i).)

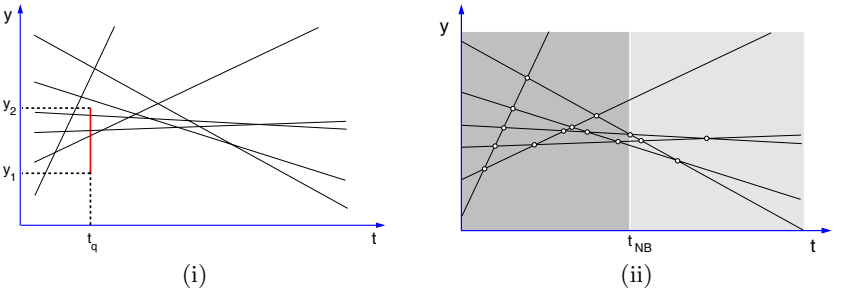


Fig. 1. (i) Lines in ty-plane traced by S. A Q1 query corresponds to finding the lines intersected by segment $[(t_q, y_1), (t_q, y_2)]$. (ii) Two windows for $N = 6$ and $B = 2$.

Since a vertical line $\ell : t = \alpha$ induces a total order on the lines in S—namely the relative ordering of the points in $S(\alpha)$—and since this ordering does not change until two points pass through each other, we can design an efficient data structure using a persistent B-tree as follows: We sweep $\mathcal{A}(S)$ from left to right ($-\infty$ to ∞) with a vertical line, inserting a segment from $\mathcal{A}(S)$ in a persistent B-tree when its left endpoint is encountered and deleting it again when its right endpoint is encountered. We can then answer a Q1 query with range $R = [y_1, y_2]$ by performing a range query with R at time t_q. Since the arrangement is of size $O(N^2)$, Lemma 2 implies that this data structure answers queries in the optimal $O(\log_B N + K/B)$ I/Os using $O(N^2/B)$ space.[1] In order to improve the size

[1] Strictly speaking, Lemma 2 as described in [8] does not hold for line segments since not all segments are above/below-comparable. However, in [5] it is discussed how to extend the result to line segments.

to $O((N/B)\log_B N)$, while at the same time obtaining a time responsive data structure, we divide the *ty*-plane into $O(\log_B N)$ vertical slabs (or *windows*) and store a modified linear-space version of the above data structure in each window.

3.1 Overall Structure

Let $t_1, \ldots, t_v$, $v \leq \binom{N}{2}$, be the sorted sequence of the t-coordinates of the vertices in the arrangement $\mathcal{A}(S)$. For $1 \leq i \leq \lceil \log_B N \rceil$, set $\tau_i = t_{NB^i}$, i.e., NB^i events occur before τ_i. We define the first window $\mathcal{W}_1$ to be the vertical slab $[-\infty, \tau_1] \times \mathbb{R}$, and the ith window $\mathcal{W}_i$, $2 \leq i \leq \lceil \log_B N \rceil$, to be the vertical slab $[\tau_{i-1}, \tau_i] \times \mathbb{R}$. Figure 1 (ii) shows an example of an arrangement of six lines with two windows.

Our data structure consists of a B-tree $\mathcal{T}$ on $\tau_1, \tau_2, \ldots$, as well as a *window structure* $\mathcal{WI}_i$ for each window $\mathcal{W}_i$. In Section 3.2 below we first describe the algorithm for constructing the windows, and in Section 3.3 we then describe the data structure $\mathcal{WI}_i$ that uses $O(N/B)$ disk blocks and answers a query with $t_q \in [\tau_{i-1}, \tau_i]$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os. To answer a Q1 query we first use $\mathcal{T}$ to determine in $O(\log_B N)$ I/Os the window $\mathcal{W}_i$ containing t_q, and then we search $\mathcal{WI}_i$ with R to report all K points of $R \cap S(t)$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os.

3.2 Computing Windows

We describe an algorithm that computes the $O(\log_B N)$ t-coordinates $\tau_1, \tau_2, \ldots$ using $O((N/B)\log_2 N \log_B^2 N)$ I/Os. We cannot afford to compute all vertices of the arrangement $\mathcal{A}(S)$ and then choose the desired t-coordinates. Instead we present an algorithm that, for any $k \in \mathbb{N}$, computes the kth leftmost vertex of $\mathcal{A}(S)$ I/O-efficiently. To obtain the τ_i's we run this algorithm with $k = NB^i$ for $1 \leq i \leq \lceil \log_B N \rceil$.

To find the kth left most vertex we first choose N random vertices of $\mathcal{A}(S)$ and sort them by their t-coordinates. As shown in [14], the merge-sort algorithm can be modified to compute the N vertices without explicitly computing all vertices of $\mathcal{A}(S)$. By using external merge-sort [4] we use $O((N/B)\log_B N)$ I/Os to compute the N vertices, and we can sort them in another $O((N/B)\log_B N)$ I/Os.[2] Let $p_1, p_2, \ldots, p_N$ be the sequence of vertices in the sorted order, and let W_i be the vertical slab defined by vertical lines through p_{i-1} and p_i. A standard probabilistic argument, omitted from this abstract, shows the following.

Lemma 3. *With probability at least* $1 - 1/N$*, each slab* W_i *contains* $O(N)$ *vertices of* $\mathcal{A}(S)$*.*

Suppose we have a procedure COUNT(W_i) that counts using $O((N/B)\log_B N)$ I/Os the number of vertices of $\mathcal{A}(S)$ lying inside a vertical slab W_i. By performing a binary search and using COUNT at each step

[2] To obtain this bound we assume that the internal memory is capable of holding B blocks. The more general bound obtained when the internal memory is of size M will be given in the full paper.

of the search, we can compute in $O((N/B)\log_2 N \log_B N)$ I/Os the index i such that the kth leftmost vertex of $\mathcal{A}(S)$ lies in the vertical strip W_i. Using $\textsc{Count}(W_i)$ again we can check whether W_i contains more than cN vertices of $\mathcal{A}(S)$, where c is the constant hidden in the big-Oh notation in Lemma 3. If it does, we restart the algorithm. By Lemma 3, the probability of restarting the algorithm is at most $1/N$. If W_i contains at most cN vertices, we find all σ_i vertices of $\mathcal{A}(S)$ lying inside the strip W_i in $O((N/B)\log_B N)$ I/Os and choose the desired vertex, using a simple modification of merge-sort; details will appear in the full paper.

What remains is to describe the $\textsc{Count}(W_i)$ procedure. Note that two lines ℓ and ℓ' intersect inside W_i if they intersect its left and right boundaries in different order, i.e., ℓ lies above ℓ' at the left boundary of W_i but below ℓ at the right boundary, or vice-versa. The problem of counting the number of vertices of $\mathcal{A}(S)$ inside W_i reduces to counting the number of pairs of lines in S that have different relative orderings at the two boundaries of W_i. It is well known that the number of such pairs can be counted using a modified version of merge-sort. Hence, we can count the number of desired vertices using $O((N/B)\log_B N)$ I/Os by modifying external merge-sort [4]. Putting everything together, we can compute the τ_i's in a total of $O((N/B)\log_2 N \log_B^2 N)$ I/Os.

3.3 Window Data Structure

Let $\mathcal{W}_i = [\tau_{i-1}, \tau_i] \times \mathbb{R}$ be a window containing NB^i vertices of $\mathcal{A}(S)$. We describe how to preprocess $\mathcal{A}(S)$ into a data structure $\mathcal{WI}_i$ that uses $O(N/B)$ disk blocks and answers a Q1 query in $O(B^{i-1} + \log_B N + K/B)$ I/Os. Since $\mathcal{W}_i$ contains NB^i vertices of $\mathcal{A}(S)$, a persistent B-tree constructed on W_i would use $\Theta(NB^{i-1})$ disk blocks. We therefore instead only build the persistent structure on N/B^i carefully selected levels of $\mathcal{A}(S)$ and build separate structures for each of the N/B^i *bundles* defined by these levels. Our algorithm relies on the following simple lemma about randomly chosen levels of the arrangement $\mathcal{A}(S)$.

Lemma 4. *For a given integer $\xi \in [0, B^i - 1]$, let $\Lambda_\xi = \{\mathcal{A}_{jB^i+\xi}(S) \mid 1 \le j \le N/B^i\}$ be N/B^i levels of $\mathcal{A}(S)$. If the integer ξ is chosen randomly, then the expected number of vertices in all the levels of Λ_ξ whose t-coordinates lie inside $\mathcal{W}_i$ is $O(N)$.*

The preprocessing algorithm works as follows. We choose a random integer $\xi \in [0, B^i - 1]$. For $1 \le j < N/B^i$, let λ_j be the $(jB^i + \xi)$-level of $\mathcal{A}(S)$. Set $\Lambda_\xi = \{\lambda_j \mid 1 \le j < N/B^i\}$. We refer to Λ_ξ as the set of *critical levels*. We compute Λ_ξ using Lemma 1. If during the construction, the size of Λ_ξ becomes more than $2cN$, where c is the constant hidden in the big-Oh notation in Lemma 4, then we abort the construction, choose another random value of ξ, and repeat the above step. This way we make sure that the critical levels are of size $O(N)$. By Lemma 4, the algorithm is aborted $O(1)$ expected times, so the expected number of I/Os needed to construct the critical levels is $O(N \log_2 N \log_B N)$ (Lemma 1). We store Λ_ξ in a persistent B-tree $\mathcal{T}_i$ by sweeping the ty-plane with a vertical line from $t = \tau_{i-1}$ to τ_i so that for a vertical segment $R = [y_1, y_2]$ and a time instance $t_q \in [\tau_{i-1}, \tau_i]$, we can compute the critical levels intersected by R. Since

there are $O(N)$ vertices in Λ_ξ, $\mathcal{T}_i$ uses $O(N/B)$ space and can be constructed in $O((N/B)\log_B N)$ I/Os.

For $1 \le j \le N/B^i$, let *bundle* $\mathcal{B}_j$ be the union of the levels that lie between λ_{j-1} and λ_j, including λ_{j-1}. See Figure 2 (i). For each bundle $\mathcal{B}_j$, we store the set of lines in $\mathcal{B}_j$ at any time $t \in [\tau_{i-1}, \tau_i]$ in a separate persistent B-tree $\mathcal{D}^i_j$. More precisely, we assume that every line in S has a unique identifier and keep the lines ordered by this identifier in $\mathcal{D}^i_j$. Let $\mathcal{D}^i_j(t)$ denote the version of $\mathcal{D}^i_j$ at time t. We sweep the window $\mathcal{W}_i$ from left to right. Initially $\mathcal{D}^i_j$ stores the lines of S in $\mathcal{B}_j$ at time τ_{i-1}. A line leaves or enters $\mathcal{B}_j$ at a vertex of λ_{j-1} or λ_j. Therefore we update $\mathcal{D}^i_j$ and $\mathcal{D}^i_{j+1}$ at each vertex of λ_j. Since Λ_ξ is of size $O(N)$, the total number of I/Os spent in constructing all the bundle structures is $O((N/B)\log_B N)$ and the total space used is $O(N/B)$ blocks. Finally, for every vertex $v = (t_v, y_v)$ on the critical level λ_j, we store a pointer to the roots of structures $\mathcal{D}^i_j(t_v)$ and $\mathcal{D}^i_{j+1}(t_v)$.

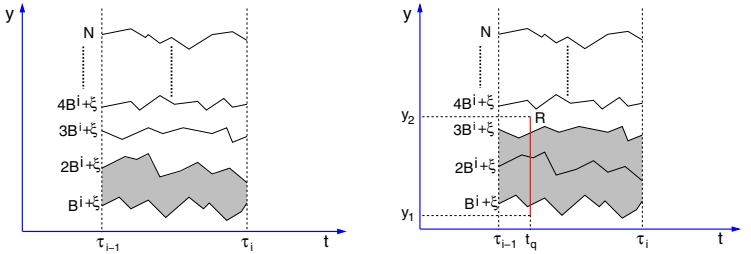


Fig. 2. (i) The N/B^i critical levels in $\mathcal{W}_i = [\tau_{i-1}, \tau_i] \times \mathbb{R}$. Bundle $\mathcal{B}_2$ is shaded. (ii) Query with $R = [y_1, y_2]$ at time t_q. All points in the shaded bundles at time t_q and all relevant points in the bundles containing the endpoints of R are reported.

To answer a Q1 query, we first use $\mathcal{T}_i$ to find the critical levels intersecting R at time t_q in $O(\log_B N)$ I/Os. Next we use the pointers from the vertices on the critical levels to find the bundle structures $\mathcal{D}^i_j$ of bundles completely spanned by R at time t_q. We report all points in these structures using $O(K/B)$ I/Os. Finally, we scan all lines in the (at most) two bundles intersected but not completely spanned by R using $O(B^i/B) = O(B^{i-1})$ I/Os and report the relevant points. In conclusion, we answer the query in $O(B^{i-1} + \log_B N + K/B)$ I/Os; see Figure 2 (ii).

Lemma 5. *Let S be a set of N points moving along the y-axis with fixed velocities, and let $\mathcal{W}_i = [\tau_{i-1}, \tau_i] \times \mathbb{R}$ be a window such that $\mathcal{A}(S)$ has $O(NB^i)$ vertices inside $\mathcal{W}_i$. We can preprocess S into a data structure $\mathcal{WI}_i$ in $O(N \log_2 N \log_B N)$ expected I/Os such that a Q1 query with $t_q \in [\tau_{i-1}, \tau_i]$ can be answered in $O(B^{i-1} + \log_B N + K/B)$ I/Os. The number of disk blocks used by the data structure is $O(N/B)$.*

This completes the description of the basic data structure, but there is one technical difficulty that one has to overcome. Immediately after building our data structure, a query in the first window W_1 can be answered in the optimal $O(\log_B N + K/B)$ I/Os. However, if we do not modify the data structure, the query performance of the structure deteriorates as the time elapses. For example, for $t_{now} > \tau_1$, a query within the first NB events after t_{now} requires $\Omega(B + \log_B N + K/B)$ I/Os. To circumvent this problem, we rebuild the entire structure during the interval $[t_{NB/2}, t_{3NB/4}]$ as though t_{now} were $t_{3NB/4}$, and switch to this structure at time $t_{3NB/4}$. This allows us to always answer a query at time t_q with $\varphi(t_q) \leq NB^i/4$ in $O(B^{i-1} + \log_B N + K/B)$ I/Os. Since we use $O(N \log_2 N \log_B^2 N)$ I/Os to rebuild the structure, we charge $O((\log_2 N \log_B^2 N)/B) = O(\log_B^3 N)$ I/Os to each of the $NB/4$ events to pay for the reconstruction cost. Putting everything together, we obtain the following.

Theorem 1. *Let S be a set of N points moving along the y-axis with fixed velocities. S can be maintained in a data structure using $O(\log_B^3 N)$ expected I/Os per kinetic event such that a Q1 query at time t_q, with $NB^{i-1} \leq \varphi(t_q) \leq NB^i$, can be answered in $O(B^{i-1} + \log_B N + K/B)$ I/Os. The structure can be built in $O(N \log_2 N \log_B^2 N)$ expected I/Os and uses $O((N/B) \log_B N)$ disk blocks.*

Remarks.

(i) Our window structure $\mathcal{WI}_i$ (Lemma 5) can easily be modified to work even if it is built on a set of line segments instead of lines, provided the lines obtained by extending the line segments have $O(NB^i)$ intersections in $\mathcal{W}_i$. We simply construct the critical levels λ_i on the lines but the bundle structures $\mathcal{B}_j$ on the segments.

(ii) Our result can be extended to the case in which the trajectory of each point is a piecewise-linear function of time, i.e., in the ty-plane, S is a set of N polygonal chains with a total of T vertices. Then the query time remains the same and the total size of the data structure is $O(T/B)$ disk blocks.

4 Data Structure for Moving Points in $\mathbb{R}^2$

We now turn to points moving in $\mathbb{R}^2$. Analogously to the $1D$-case, $S(t)$ traces out N lines in xyt-space, and our structure for Q2 queries utilizes the same general ideas as our structure for Q1 queries. While a kinetic event in the $1D$-case corresponds to two points passing each other, an event now occurs when the x- or y-coordinates of two points coincide. We divide the xyt-space into $O(\log_B N)$ horizontal *slices* along the t-axis such that slice $\mathcal{S}_i$ contains $O(NB^i)$ kinetic events; see Figure 3 (i). That is, we choose a sequence $\tau_1 < \tau_2 < \ldots$ of $\log_B N$ time instances so that $O(NB^i)$ events occur in the interval $[\tau_{i-1}, \tau_i]$. We set $\mathcal{S}_i = \mathbb{R}^2 \times [\tau_{i-1}, \tau_i]$. As previously, our data structure consists of a B-tree on $\tau_1, \tau_2, \ldots$, and a linear-space *slice structure* $\mathcal{SL}_i$ for each slice $\mathcal{S}_i$. Below, we will design a linear space data structure $\mathcal{SL}_i$, which can be used to answer a Q2 query with $t_q \in [\tau_{i-1}, \tau_i]$ in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os. Each

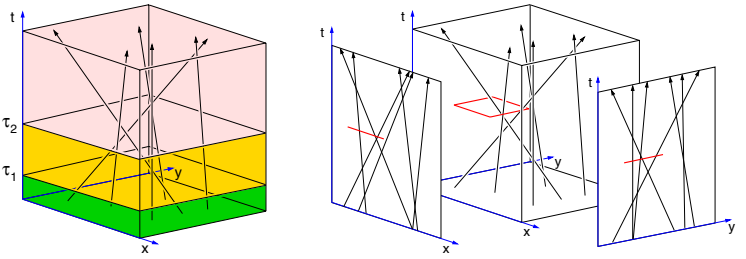


Fig. 3. (i) Three slices (ii) The lines in xyt-space traced by $S(t)$ and their projections.

$\mathcal{SL}_i$ can be constructed in $O(N \log_2 N \log_B N)$ expected I/Os and updated in $O(\log_B^2 N)$ I/Os.

To answer a Q2 query at time t_q with a rectangle R, we first determine the slice $\mathcal{S}_i$ containing t_q and then we query $\mathcal{SL}_i$ to report all K points of $S(t) \cap R$ in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os. Our global data structure uses $O((N/B) \log_B N)$ disk blocks in total. The structure can be constructed in $O(N \log_2 N \log_B^2 N)$ expected I/Os since we can compute the τ_i's as follows. Let $\mathcal{A}^x(S)$ be the arrangement obtained by projecting the lines of S onto the tx-plane (Figure 3 (ii)). Let $k_i = NB^i$ for $1 \le i \le \log_B N$. As in Section 3, we can compute, using $O(N \log_2 N \log_B^2 N)$ I/Os, the time instances τ_i^x, $1 \le i \le \log_B N$, so that the t-coordinate of k_i vertices of $\mathcal{A}^x(S)$ is at most τ_i^x. Similarly, we compute the time instances τ_i^y, $1 \le i \le \log_B N$, so that the t-coordinate of k_i vertices of $\mathcal{A}^y(S)$ is at most τ_i^y. Set $\tau_0^x = \tau_0^y = -\infty$ and $\tau_i = \min\{\tau_i^x, \tau_i^y\}$ for $0 \le i \le \log_B N$. Define the slice $\mathcal{S}_i$ to be $\mathbb{R}^2 \times [\tau_{i-1}, \tau_i]$ for $1 \le i \le \log_B N$. $\mathcal{S}_i$ is guaranteed to contain less than $2NB^i$ events. Finally, as in the $1D$-case, we rebuild the data structure every $\Theta(NB)$ events and obtain the following.

Theorem 2. *Let S be a set of N points moving in the xy-plane with fixed velocities. S can be maintained in a data structure using $O(\log_B^3 N)$ expected I/Os per kinetic event so that a Q2 query at time t_q, with $NB^{i-1} \le \varphi(t_q) \le NB^i$, can be answered in $O(\sqrt{N/B^i}(B^{i-1} + \log_B N) + K/B)$ I/Os. The structure can be built in $O(N \log_2 N \log_B^2 N)$ expected I/Os and uses $O((N/B) \log_B N)$ disk blocks.*

Slice Structure. We now describe our slice data structure $\mathcal{SL}_i$. Answering a Q2 query corresponds to finding the lines traced out by S in xyt-space that intersect a rectangle R on the plane $t = t_q$ parallel to the xy-plane. Note that a line l intersects R if and only if their projections onto the tx- and ty-planes both intersect. See Figure 3 (ii). Let S^x denote the projection of S onto the tx-plane, and let $\mathcal{S}_i^x, \mathcal{S}_i^y$ be the projection of the slice $\mathcal{S}_i$ onto the tx- and ty-planes, respectively. As earlier, we define a set of *critical levels* λ_j of $\mathcal{A}(S^x)$ within the projected window $\mathcal{S}_i^x$ and store it in a persistent B-tree $\mathcal{T}_i$. We choose a random integer $\xi \in [0, \sqrt{NB^i}]$ and set λ_j to be the $(j\sqrt{NB^i} + \xi)$-level of $\mathcal{A}^x(S)$. We have $\sqrt{N/B^i}$ critical levels, and we define *bundle* $\mathcal{B}_j^i$ to be the $\sqrt{NB^i}$ levels between critical level λ_{j-1} and λ_j. Refer to Figure 4 (i). Using Lemma 4, we can prove

that the total expected size of the critical levels is $O(N + NB^i/\sqrt{NB^i}) = O(N)$. For each bundle $\mathcal{B}^i_j$ we construct a $1D$-data structure on S^y, the projection of S onto the ty-plane, as follows.

Fix a bundle $\mathcal{B}^i_j$. A point $p \in S$ can leave and return to $\mathcal{B}^i_j$ several times, i.e., its x-projection may cease to lie between the levels λ_{j-1} and λ_j and then it may appear there again. Therefore, for a point $p \in S$, let $\triangle_1, \ldots, \triangle_r$ be the maximal time intervals, each a subset of $[\tau_{i-1}, \tau_i]$, during which p lies in $\mathcal{B}^i_j$. We map p to a set $S_p = \{e_1, \ldots, e_r\}$ of segments in the ty-plane, where $e_z = \bigcup_{t \in \triangle_z} p^y(t)$ is a segment contained in the ty-projection of the trajectory of p. Define $S^y_j = \bigcup_{p \in S} S_p$; set $N_j = |S^y_j|$. Since the endpoint of a segment in S^y_j corresponds to a vertex of λ_{j-1} or λ_j, $\sum_j N_j = O(N)$. We construct the window structure $\mathcal{WI}^i_j$ on S^y_j. Our construction of slices guarantees that there are $O(NB^i)$ vertices of $\mathcal{A}^y(S)$ in the ty-projection $\mathcal{S}^y_i$ of the slice $\mathcal{S}_i$. Therefore, the remark at the end of Section 3.3, implies that all the bundle structures use a total of $O(N/B)$ disk blocks and that they can be constructed in $O(N \log_2 N \log_B N)$ I/Os.

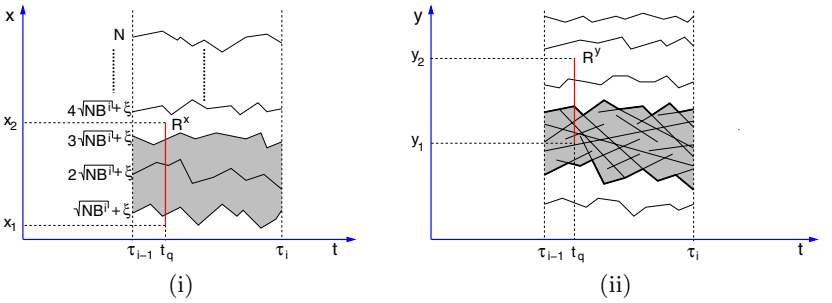


Fig. 4. (i) The $\sqrt{N/B^i}$ critical levels in the arrangement $\mathcal{A}(S^x)$ of the projection of S onto the tx-plane. (ii) One bundle in window structure built on the ty-projection of segments corresponding to points in bundle $\mathcal{B}_j$ in the tx-plane.

To answer a query Q2, we first use $\mathcal{T}_i$ to find the critical levels, and thus bundles, intersecting the tx-projection R^x of R in $O(\log_B N)$ I/Os. For all $O(\sqrt{N/B^i})$ bundles $\mathcal{B}_j$ completely spanned by R^x, we query the corresponding window structure to find all points also intersecting the yt-projection R^y of R using $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os (Lemma 5). Finally, we scan the (at most) two bundles intersected but not completely spanned by R^x using $O(\sqrt{NB^i}/B + K/B) = O(\sqrt{N/B^i} \cdot B^{i-1} + K/B)$ I/O and report the remaining points in R at time t_q. Refer to Figure 4 (i).

Lemma 6. *Let S be a set of N points moving in the xy-plane with fixed velocities, and let $\mathcal{S}_i$ be a time slice $\mathbb{R}^2 \times [\tau_{i-1}, \tau_i]$ that contains NB^i events. We can preprocess S into a data structure $\mathcal{SL}_i$ of size $O(N/B)$ in $O(N \log_2 N \log_B N)$ I/Os such that a Q2 query at time $\tau_{i-1} \le t_q \le \tau_i$ can be answered in $O(\sqrt{N/B^i} \cdot (B^{i-1} + \log_B N) + K/B)$ I/Os.*

References

1. P. K. Agarwal, L. Arge, and J. Erickson, Indexing moving points, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 2000, pp. 175–186.
2. P. K. Agarwal, L. Arge, J. Erickson, P. Franciosa, and J. Vitter, Efficient searching with linear constraints, *Journal of Computer and System Sciences*, 61 (2000), 194–216.
3. P. K. Agarwal and J. Erickson, Geometric range searching and its relatives, in: *Advances in Discrete and Computational Geometry* (B. Chazelle, J. E. Goodman, and R. Pollack, eds.), *Contemporary Mathematics*, Vol. 223, American Mathematical Society, Providence, RI, 1999, pp. 1–56.
4. A. Aggarwal and J. S. Vitter, The Input/Output complexity of sorting and related problems, *Communications of the ACM*, 31 (1988), 1116–1127.
5. L. Arge, External memory data structures, in: *Handbook of Massive Data Sets* (J. Abello, P. M. Pardalos, and M. G. C. Resende, eds.), Kluwer Academic Publishers, 2001. (To appear).
6. L. Arge, V. Samoladas, and J. S. Vitter, On two-dimensional indexability and optimal range search indexing, *Proc. ACM Symp. Principles of Database Systems*, 1999, pp. 346–357.
7. J. Basch, L. J. Guibas, and J. Hershberger, Data structures for mobile data, *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, 1997, pp. 747–756.
8. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, An asymptotically optimal multiversion B-tree, *VLDB Journal*, 5 (1996), 264–275.
9. D. Comer, The ubiquitous B-tree, *ACM Computing Surveys*, 11 (1979), 121–137.
10. T. K. Dey, Improved bounds on planar k-sets and related problems, *Discrete Comput. Geom.*, 19 (1998), 373–382.
11. H. Edelsbrunner and E. Welzl, Constructing belts in two-dimensional arrangements with applications, *SIAM J. Comput.*, 15 (1986), 271–284.
12. L. J. Guibas, Kinetic data structures — a state of the art report, in: *Proc. Workshop Algorithmic Found. Robot.* (P. K. Agarwal, L. E. Kavraki, and M. Mason, eds.), A. K. Peters, Wellesley, MA, 1998, pp. 191–209.
13. G. Kollios, D. Gunopulos, and V. J. Tsotras, On indexing mobile objects, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 1999, pp. 261–272.
14. J. Matoušek, Randomized optimal algorithm for slope selection, *Inform. Process. Lett.*, 39 (1991), 183–187.
15. J. Matoušek, Efficient partition trees, *Discrete Comput. Geom.*, 8 (1992), 315–334.
16. D. Pfoser, C. S. Jensen, and Y. Theodoridis, Novel approaches to the indexing of moving objects trajectories, *Proc. International Conf. on Very Large Databases*, 2000, pp. 395–406.
17. G. Toth, Point sets with many k-sets, *Proc. 16th Annu. Symposium on Computational Geometry*, 2000, pp. 37–42.
18. P. J. Varman and R. M. Verma, An efficient multiversion access structure, *IEEE Transactions on Knowledge and Data Engineering*, 9 (1997), 391–409.
19. J. S. Vitter, Online data structures in external memory, *Proc. Annual International Colloquium on Automata, Languages, and Programming, LNCS 1644*, 1999, pp. 119–133.
20. S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, Indexing the positions of continuously moving objects, *Proc. ACM SIGMOD International Conference on Management of Data*, 2000, pp. 331–342.

Voronoi Diagrams for Moving Disks and Applications

Menelaos I. Karavelas*

Graphics Laboratory, Computer Science Department, Stanford University, Stanford, CA 94305-9035, USA,
karavelas@cs.stanford.edu

Abstract. In this paper we discuss the kinetic maintenance of the Euclidean Voronoi diagram and its dual, the Delaunay triangulation, for a set of moving disks. The most important aspect in our approach is that we can maintain the Voronoi diagram even in the case of intersecting disks. We achieve that by augmenting the Delaunay triangulation with some edges associated with the disks that do not contribute to the Voronoi diagram. Using the augmented Delaunay triangulation of the set of disks as the underlying structure, we discuss how to maintain, as the disks move, (1) the closest pair, (2) the connectivity of the set of disks and (3) in the case of non-intersecting disks, the near neighbors of a disk.

1 Introduction

Geometric objects that move with time appear in many problems in motion planning, geometric modeling, computer simulations of physical systems and virtual environments, robotics, computer graphics and animation, mobile communications, *ad hoc* networks or group communication in military operations. The aim is to answer questions concerning proximity information among the geometric objects, such as find the closest/farthest pair, report all near neighbors, predict collisions or report reachability between a pair of geometric objects. In many cases we can approximate the geometric objects by disks. The problem then reduces to answering proximity questions for a set of disks.

The Voronoi diagram is a data structure that can be used to produce answers to many of these questions. Voronoi diagrams have been successful in robotics applications such as collision detection [11] and retraction motion planning [10]. Dynamic or kinetic Voronoi diagrams for moving objects in the plane have also appeared in the literature. There are papers discussing the maintenance of the Voronoi diagram for a set of points [4], the maintenance of the Voronoi diagram for a set of moving convex polygons [5], as well as for the maintenance of near neighbors of points in sets of moving points [9].

Algorithms for maintaining the Voronoi diagram for sets of non-intersecting moving disks have also appeared. In [2] the Voronoi diagram for disks with respect to the Euclidean metric is considered. The maintenance of the Voronoi

* Supported by NSF grant CCR-9910633.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 62–74, 2001.

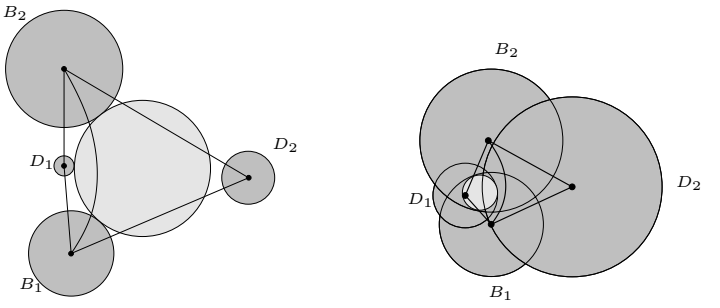


Fig. 1. Left: the edge connecting B_1 and B_2 is locally Delaunay because the exterior tangent ball of B_1, B_2 and D_1 does not intersect D_2. Right: the edge connecting B_1 and B_2 is locally Delaunay because the interior tangent ball of B_1, B_2 and D_1 is not contained in D_2. The exterior/interior tangent ball of B_1, B_2 and D_1 is shown in light gray.

diagram with respect to the power distance, also called the *Power diagram*, is discussed in [6]. The most appealing feature of the Power diagram is that it consists of straight arcs, in contrast to the Euclidean Voronoi diagram that consists of straight or hyperbolic arcs. The main drawback of the Power diagram is that the intersection between a disk and its Voronoi cell may be empty, even if the Voronoi cell is not empty [8]. In the Euclidean Voronoi diagram, however, a disk with non-empty Voronoi cell always intersects its cell [12].

In this paper we tackle the problem of maintaining the Voronoi diagram, or its dual the Delaunay Triangulation (DT) for a set of disks moving in the plane. The major contribution of this paper is that the disks are allowed to intersect. This enables us to not only report collisions between disks, but also to report when the penetration depth between two disks achieves a certain value, or when a disk is wholly contained inside another disk. Moreover, our data structure can be used for maintaining the connectivity of the set of disks as the disks move.

The Voronoi diagram is maintained using the Kinetic Data Structure (KDS) framework introduced in [1]. In the KDS setting one maintains a geometric structure under continuous motion through a set of certificates proving its correctness. An *event queue* is maintained for the failure times of these certificates and at each event the structure of interest and its kinetic proof are appropriately updated.

The kinetization process relies heavily on the fact that the local Delaunay property for the edges in the DT ensures that the triangulation is globally Delaunay. Let e be an edge connecting the disks B_1, B_2 and having the disks D_1, D_2 as its neighbors in the triangulation. The local Delaunay property states that the edge e is an edge of the DT if the exterior tangent ball of B_1, B_2 and D_1 does not intersect the disk D_2, or if the interior tangent ball of B_1, B_2 and D_1 is not contained in D_2 (see Fig. 1). The global Delaunay property states that there exists an edge in the DT between two disks B_1 and B_2 if there exists an exterior tangent ball to B_1 and B_2 that does not intersect any other disk or if

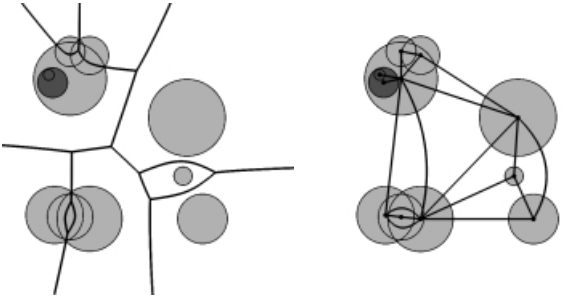


Fig. 2. The Voronoi diagram for a set of disks (left) and the corresponding Augmented Delaunay Triangulation (right). The disks in dark gray are trivial.

there exists an interior tangent ball to B_1 and B_2 that is not contained in any other disk. In this work we prove the relationship between the global and local Delaunay properties.

Using the local Delaunay property, we can maintain the Voronoi diagram for the set of disks using two types of events, one of which appears only in the case of intersecting disks. The data structure that we use is called the *Augmented Delaunay Triangulation* (ADT) of the set of disks. It consists of the Delaunay triangulation of the disks augmented with some additional linear size data structure associated with the disks that do not contribute to the Voronoi diagram (see Fig. 2). We call these disks *trivial*. Since trivial disks exist only when we allow disk intersections, the ADT differs from the DT only when we have disk intersections.

An interesting property of the ADT is that the closest pair of the set of disks is realized between two disks that share an edge in the ADT. Thus, knowing how to maintain the ADT enables us to maintain the closest pair of the set of disks using a tournament tree on the edges of the ADT. The distance between two disks B_1 and B_2 is defined as :

$$\delta(B_1, B_2) = \begin{cases} d(b_1, b_2) - r_1 - r_2, & B_1 \not\subseteq B_2 \text{ and } B_2 \not\subseteq B_1 \\ -2\min\{r_1, r_2\}, & B_1 \subseteq B_2 \text{ or } B_2 \subseteq B_1 \end{cases}, \tag{1}$$

where b_i are the centers of the disks, r_i their radii and $d(\cdot, \cdot)$ denotes the Euclidean metric. If the set of disks does not have any intersecting disks the distance function (1) gives us the closest pair in the usual sense. If there are intersecting disks, then the closet pair with respect to (1) is either the pair of non-trivial disks with *maximum penetration depth* among all intersecting pairs of disks, or the largest trivial disk and its container.

Another important property of the ADT is that a subgraph of the ADT is a spanning subgraph of the connectivity graph of the set of disks. Knowing how to maintain the ADT enables us to maintain the connectivity of the set of disks by maintaining the afore-mentioned spanning subgraph.

Finally, the DT of a set of non-intersecting disks has the property that if we want to find the near neighbors of a disk we only need to look at its neighborhood in the DT. Therefore, in order to maintain the near neighbors of a disk we simply need to look at its neighborhood in the DT and update this neighborhood as the DT changes. We make this statement more precise in Section 7.

The rest of the paper is structured as follows. In Section 2 we introduce the Voronoi diagram for disks, and discuss some of its properties. Section 3 is devoted to proving the relationship between the global and local Delaunay properties. In Section 4 we show how to kinetize the Voronoi diagram. In Section 5 we describe how to maintain the closest pair. In Section 6 we show how to maintain a spanning subgraph of the connectivity graph of the set of disks. In Section 7 we discuss the maintenance of near neighbors of disks. Finally, Section 8 is devoted to conclusions and further work.

2 The Voronoi Diagram for Disks and Its Properties

Let S be a set of n disks B_j, with centers b_j and radii r_j. Let $d(\cdot,\cdot)$ be the Euclidean distance. We define the distance $\delta(p,B)$ between a point $p \in \mathbb{E}^2$ and a disk $B = \{b, r\}$, as $\delta(p,B) = d(p,b) - r$. We define the Voronoi diagram for the set S as follows. For each $i \neq j$, let $H_{ij} = \{y \in \mathbb{E}^2 : \delta(y,B_i) \leq \delta(y,B_j)\}$. Then we define the (closed) Voronoi cell of B_i to be the cell $V_i = \bigcap_{j \neq i} H_{ij}$. The *Voronoi diagram* $\mathrm{VD}(S)$ of S is defined to be the set of points which belong to more that one Voronoi cell. The Voronoi diagram just defined is a subdivision of the plane. It consists of straight or hyperbolic arcs and each Voronoi cell is star-shaped with respect to the center of the corresponding disk. In contrast to the Voronoi diagram for points, there may be disks whose corresponding Voronoi cell is empty. In particular, the Voronoi cell V_i of a disk B_i is empty if and only if B_i is wholly contained in another disk (see [12, Property 2]). A disk whose Voronoi cell has empty interior is called *trivial*, otherwise is called *non-trivial*.

We define the dual of $\mathrm{VD}(S)$ as follows. The vertices are the centers of the non-trivial disks. If $V_i \cap V_j \neq \emptyset$, we add an edge $[b_i, b_j]$ for every open arc α of $V_i \cap V_j$. It turns out that the dual graph is a planar graph and the size of both the Voronoi diagram and its dual graph is $O(n)$ [12, Properties 6 and 7]. If the Voronoi diagram consists of a single connected component and the disks are in general position, the dual graph is a generalized triangulation of the plane. By generalized we mean that the edges of the triangles may be curved arcs or polygonal lines instead of straight line segments. We assume throughout the rest of the paper that the Voronoi diagram consists of only one connected component. We shall refer to the dual graph of $\mathrm{VD}(S)$ as the *Delaunay Graph* $\mathrm{DG}(S)$ of S. If the disks are in general position we refer to the dual graph as the *Delaunay Triangulation* $\mathrm{DT}(S)$ of S.

Let B_i and B_j be two disks such that no disk is contained inside the other. A ball tangent to B_i and B_j that does not contain either of the two is an *exterior tangent ball*. A ball tangent to B_i and B_j that lies in $B_i \cap B_j$ is an *interior*

tangent ball. The following theorem couples the existence of edges in $\mathrm{DG}(S)$ with exterior and interior tangent balls of disks in S.

Theorem 1 (Global Property). *There exists an edge $[b_i, b_j]$ in $DG(S)$ between B_i and B_j if and only if one of the following holds: (1) there exists an exterior tangent ball to B_i and B_j which does not intersect any disk $B_k \in S$, $k \neq i, j$; (2) there exists an interior tangent ball to B_i and B_j, which is not contained in any disk $B_k \in S$, $k \neq i, j$.*

Proof. (Sketch) Let $[b_i, b_j]$ be an edge of $\mathrm{DG}(S)$. Then $V_i \cap V_j$ consists of at least one arc α with non-empty interior. Let y be an interior point of α. Consider the ball C centered at y with radius $|\delta(y, B_i)| = |\delta(y, B_j)|$. Then C is tangent to both B_i, B_j and does not intersect any disk B_k, $k \neq i, j$, if $y \notin B_i \cap B_j$, and is not contained in any disk B_k, $k \neq i, j$, if $y \in B_i \cap B_j$. Conversely, let C be a common tangent ball of B_i, B_j, and let y be its center. If either assumption (1) or assumption (2) of the theorem holds, we have that $\delta(y, B_i) = \delta(y, B_j) < \delta(y, B_k)$, for all $k \neq i, j$. Hence y is an interior point of some arc α of $V_i \cap V_j$, and thus there exists at least one edge $[b_i, b_j]$ in $\mathrm{DG}(S)$. □

In order to account for the trivial disks, we augment the Delaunay triangulation with some additional edges. For a trivial disk D we add an edge between D and its container disk. If D has more than one container we need to add an edge to only one of its containers, chosen arbitrarily. We call this structure the *Augmented Delaunay Triangulation* $\mathrm{ADT}(S)$ of S. The set of additional edges forms a forest, and the root of each tree in the forest is a non-trivial disk. Clearly, the forest has linear size. Hence the size of $\mathrm{ADT}(S)$ is still $O(n)$.

3 The Local Property of the Delaunay Triangulation

In this section we present the local Delaunay property for a set of possibly intersecting disks and we show that the local Delaunay property is a sufficient and necessary condition for a (generalized) triangulation of the set of disks to be globally Delaunay. We only consider non-trivial disks, since trivial disks do not contribute to the Voronoi diagram.

Let π_{ij} be the bisector of B_i and B_j. The bisectors are lines or hyperbolas which can be oriented. We define the orientation to be such that b_i is to the left of π_{ij}. Let $\prec$ be the linear ordering on the points of π_{ij}. Let o_{ij} be the midpoint of the subsegment of $b_i b_j$ that lies either in free space or in $B_i \cap B_j$. We can parameterize π_{ij} as follows: if $p \prec o_{ij}$ then $\zeta_{ij}(p) = -(\delta(p, B_i) - \delta(o_{ij}, B_i))$; otherwise $\zeta_{ij}(p) = \delta(p, B_i) - \delta(o_{ij}, B_i)$. The function ζ_{ij} is a 1–1 and onto mapping from π_{ij} to $\mathbb{R}$.

Let B_i, B_j and B_k be three disks such that no disk is contained inside another. The three disks may have up to eight common tangent balls. Among those we are interested in only two kinds: those balls that do not contain any of the three disks, which we call *exterior tangent balls* and those that are contained entirely in the intersection of the three disks, which we call *interior tangent balls*. Let P_i, P_j,

P_k be the points of tangency of the disks B_i, B_j, B_k with their common tangent ball. If $\mathsf{CCW}(P_i, P_j, P_k) > 0$ we call the common tangent ball the *left tangent ball* of the triple B_i, B_j, B_k. If $\mathsf{CCW}(P_i, P_j, P_k) < 0$, we call the common tangent ball the *right tangent ball* of the triple B_i, B_j, B_k. Note that three disks have at most one left/right exterior/interior tangent ball. Finally, we define $\zeta_{ij}^L(B_k)$ to be the parameter value of the center $c \in \pi_{ij}$ of the left tangent ball of B_i, B_j and B_k. Correspondingly, $\zeta_{ij}^R(B_k)$ is the parameter value of the center of the right tangent ball of B_i, B_j and B_k.

Let $\mathcal{T}(S)$ be a (generalized) triangulation of S that is constructed as follows. The vertices of $\mathcal{T}(S)$ are the centers of the disks in S. An oriented edge e_{ij}^{kl} in $\mathcal{T}(S)$ is an edge that connects the disks B_i and B_j and has as neighbors the disks B_k and B_l to its left and right, respectively. It is possible that the disks B_k and B_l are the same. The disk B_k is called the *left neighbor* of e_{ij}^{kl} and the disk B_l is called the *right neighbor* of e_{ij}^{kl}. Note that the quadruple (i, j, k, l) uniquely defines edges in $\mathcal{T}(S)$, i.e., there can be at most one oriented edge in the triangulation starting from B_i, ending at B_j and having B_k and B_l to its left and right, respectively. The left tangent ball of the triple B_i, B_j, B_k is called the *left (tangent) ball* of e_{ij}^{kl}, and similarly, the right tangent ball of the triple B_i, B_j, B_l is called the *right (tangent) ball* of e_{ij}^{kl}. We assume that for every edge in $\mathcal{T}(S)$ its left and right tangent balls exist. Then we can embed e_{ij}^{kl} with a two-leg polygonal line $b_i x b_j$, where x is a point on π_{ij} with parameter value $\zeta_{ij}(x)$ in between $\zeta_{ij}^L(B_k)$ and $\zeta_{ij}^R(B_l)$. For every triangle $\Delta_{ijk} \in \mathcal{T}(S)$ that connects the disks B_i, B_j and B_k, in counterclockwise order, we associate the left tangent ball $\tilde{\Delta}_{ijk}$ of the triple B_i, B_j, B_k. This is called the *Delaunay ball* of Δ_{ijk}. Note that there is an 1–1 correspondance between triangles Δ in $\mathcal{T}(S)$ and their Delaunay balls $\tilde{\Delta}$.

An edge e_{ij}^{kl} in $\mathcal{T}(S)$ is called *locally Delaunay* if the predicate $\mathsf{InCircle}(B_i, B_j, B_k, B_l)$ is false. A triangle Δ in $\mathcal{T}(S)$ is called locally Delaunay if all its edges are locally Delaunay. The $\mathsf{InCircle}$ predicate is defined below.

Definition 1. *Let B_i, B_j, B_k, B_l be four disks. The predicate $\mathsf{InCircle}(B_i, B_j, B_k, B_l)$ is true if $k \neq l$ and either B_l intersects the exterior left tangent ball of B_i, B_j and B_k, or B_l contains the interior left tangent ball of B_i, B_j and B_k.*

Note that if an edge e_{ij}^{kl} is locally Delaunay then $\zeta_{ij}^L(B_k) > \zeta_{ij}^R(B_l)$. This imples that if a triangle Δ is locally Delaunay, then the center $c_{\tilde{\Delta}}$ of its Delaunay ball $\tilde{\Delta}$ lies in the interior of Δ. We are now ready to prove the main result of this section.

Theorem 2 (Local Property). *A (generalized) triangulation $\mathcal{T}(S)$ is the Delaunay triangulation of S if and only if all the triangles in $\mathcal{T}(S)$ are locally Delaunay.*

Proof. It is straightforward to verify that if a triangulation is globally Delaunay then it is locally Delaunay as well.

Suppose now that we have a triangulation $\mathcal{T}(S)$ that is locally Delaunay but not globally. We assume without loss of generality that for all triangles the corresponding Delaunay balls are interior. If this is not the case we can increase the radii of all the disks by a sufficiently large quantity. The triangulation $\mathcal{T}(S)$ is not affected by this change, other than that all the Delaunay balls become interior.

Since $\mathcal{T}(S)$ is not globally Delaunay there exists a triangle Δ that is locally Delaunay but its Delaunay ball $\tilde{\Delta}$ is contained inside some disk $B = \{b, r\}$. Consider the distance between the disk B and the Delaunay ball $\tilde{\Delta}$. This distance is $\delta(\tilde{\Delta}, B) = d(b, c_{\tilde{\Delta}}) + r_{\tilde{\Delta}} - r$, where $c_{\tilde{\Delta}}$ and $-r_{\tilde{\Delta}}$ are the center and radius of $\tilde{\Delta}$ (interior Delaunay balls are considered to have negative radius). Among all triangles Δ for which $\tilde{\Delta} \subset B$, choose Δ to be the one for which $\delta(\tilde{\Delta}, B)$ is minimized.

Let $e = [b_1, b_2]$ be the (oriented) edge of Δ that the segment $c_{\tilde{\Delta}}b$ intersects (see Fig. 3). Let L be the two-leg polygonal line $b_1 c_{\tilde{\Delta}} b_2$. Since $c_{\tilde{\Delta}}b$ intersects e, b must lie in the half-plane H bounded by L that contains e. Let Δ' be the left neighboring triangle of e. Since both Δ and Δ' are locally Delaunay the quad $Q = b_1 c_{\tilde{\Delta}} b_2 c_{\tilde{\Delta}'}$ is contained inside $\Delta \cup \Delta'$, and clearly b cannot lie inside Q. But then we have $\tilde{\Delta}' \subset B$, and moreover $\delta(\tilde{\Delta}', B) < \delta(\tilde{\Delta}, B)$, which contradicts the fact that $\delta(\tilde{\Delta}, B)$ is minimal. □

4 Kinetizing the Delaunay Triangulation

The framework that we use for maintaining the Voronoi diagram or equivalently the Augmented Delaunay triangulation is the Kinetic Data Structure (KDS) framework. The geometric attribute that we want to maintain as the objects move is called the *configuration function*, e.g., the Voronoi diagram of the set of

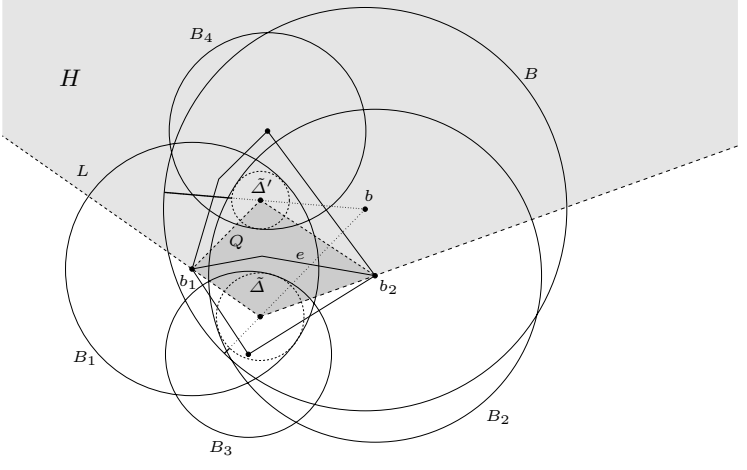


Fig. 3. Proof of the local property.

disks. In the KDS setting we maintain a set of *certificates* that are conditions which ensure the correctness of the configuration function, e.g., that an edge in $\mathrm{DT}(S)$ passes the InCircle test. As the objects move the certificates fail. These are the critical events for the KDS and moreover the corresponding times are the only times that the configuration function can possibly change. When a critical event happens we need to change the set of certificates and possibly the configuration function itself. In order to efficiently update the configuration function we maintain an event queue of the certificates w.r.t. their failure times. When a critical event takes place we remove some certificates from the queue and add some new ones. For more details on KDSs see [1].

Maintaining the Voronoi diagram or its dual, the Delaunay triangulation, for a set of points moving on the plane is straightforward [4]. This is due to the local property of the Delaunay triangulation, which states that if the triangles of the Delaunay triangulation are locally Delaunay, then the triangulation is the Delaunay triangulation. When one of the conditions fails we simply have to do an *edge-flip* operation to restore the correctness of the Delaunay triangulation. The same principle is exploited to maintain the power diagram of non-intersecting moving disks [6] and the Voronoi diagram for rigidly moving polygons [5].

In the case of non-intersecting disks the very same ideas can be used. The local Delaunay property is also true for the Delaunay triangulation of disks, as we showed in the preceding section, and thus the critical events happen at times when four disks are cocircular or when three disks lying on the convex hull of S have a common tangent line. In fact if we add a disk at infinity and connect every disk lying on the convex hull of S with the disk at infinity, the compactified version of the Delaunay triangulation of S consists of triangles only, and every triangle has exactly three neighboring triangles. In this setting, the case of three disks having a common tangent reduces to a cocircularity event with one of the disks being a disk at infinity. When such a cocircularity event happens we only need to perform an edge-flip operation to restore the correctness of the Delaunay triangulation, and its dual the Voronoi diagram.

However, when we allow disk intersections the situation changes considerably. Unlike the points' case, there are disks that are not associated with a particular Voronoi cell, namely the trivial disks. We need to account for these disks, since as the disks move some of the trivial disks may become non-trivial and vice versa. This is done by considering the Augmented Delaunay triangulation instead of the Delaunay triangulation. There are two types of events that change the combinatorial structure of $\mathrm{ADT}(S)$: the *cocircularity* and the *appearance/disappearance* event. Both events are associated with edges of the $\mathrm{ADT}(S)$. In particular, an edge in $\mathrm{DT}(S)$ is associated with a cocircularity and a disappearance event. An edge in $\mathrm{ADT}(S) \setminus \mathrm{DT}(S)$ is associated with an appearance event. We now discuss each type of event separately.

The **cocircularity event** happens when four distinct disks have a common exterior or interior tangent ball. Let B_i, $i = 1, 2, 3, 4$ be the four disks associated with the cocircularity event and let $[b_1, b_3]$ be the edge to be flipped. We need to delete that edge and add the edge $[b_2, b_4]$ (see Fig. 4(left)).

An **appearance event** occurs when a disk B_1 contained inside a disk B_2 is no longer wholly contained inside B_2. There are two possibilities when this happens: (1) B_2 is a trivial disk and (2) B_2 is a non-trivial disk. If B_2 is a trivial disk we delete the edge $[b_1, b_2]$ and add the edge $[b_1, b_3]$, where B_3 is the container disk of B_2. If B_2 is a non-trivial disk we need to first check its neighbors in the DT to see if B_1 is contained in any one of them. If such a neighbor B_3 exists we delete the edge $[b_1, b_2]$ and add the edge $[b_1, b_3]$. If B_1 is not contained in any of the neighbors of B_2, we need to identify the edge $[b_2, b_3]$ that corresponds to the edge of the Voronoi cell of B_2 that the half-line b_2b_1 intersects. Then duplicate this edge and add the edge $[b_1, b_3]$, thus creating two new triangles in $\mathrm{DT}(S)$ (see Fig. 4(right), from right to left).

A **disappearance event** takes place between two disks B_1 and B_2 when, e.g., B_1 becomes wholly contained in B_2. The edge $[b_1, b_2]$ belongs to two triangles with a common third point b_3 corresponding to a disk B_3. When the disappearance event happens we need to delete the edge $[b_1, b_3]$, and identify the two edges $[b_2, b_3]$, thus deleting two triangles from $\mathrm{DT}(S)$ (see Fig. 4(right), from left to right).

In any case, when an edge disappears we deschedule all the events associated with that edge. When an edge appears we schedule all the events corresponding to that edge and reschedule all the cocircularity events, if any, in which the new edge participates.

The number of certificates that we maintain is $O(n)$, since we have a constant number of certificates per edge in $\mathrm{ADT}(S)$, and the number of edges in $\mathrm{ADT}(S)$ is $O(n)$. The time to process the cocircularity and disappearance events is $O(\log n)$. The time to process the appearance event is $O(n)$. If the disks move along pseudo-algebraic trajectories, the total number of cocircularity events that our KDS has to process is $O(n^3\beta(n))$, where $\beta(n) = \lambda_s(n)/n$ and $\lambda_s(n)$ is the maximum length of a (n, s) Davenport-Schinzel sequence for some constant s. The total number

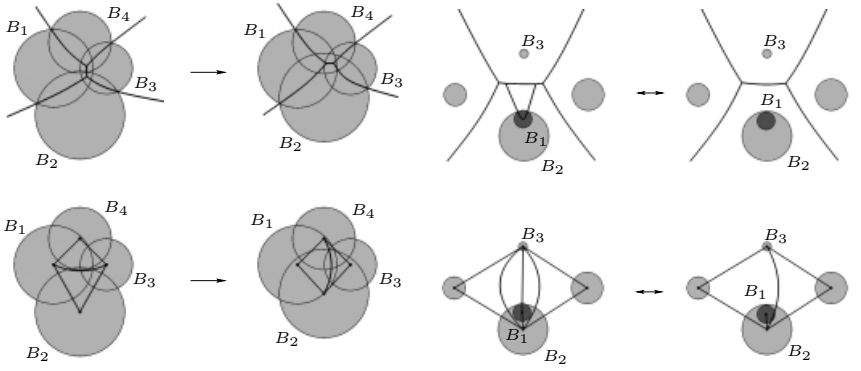


Fig. 4. The cocircularity (left) and appearance/disappearance events (right). Top row: the Voronoi diagram. Bottom row: the Augmented Delaunay triangulation.

of appearance/disappearance events that our KDS processes is obviously $O(n^2)$. Hence the total number of events that have to be processed is $O(n^3\beta(n))$. A lower bound of $\Omega(n^2)$ on the number of events can also be shown.

5 Closest Pair Maintenance

In this section we discuss how to maintain the closest pair of a set S of disks. The distance function that we use is given by relation (1). The trivial way to do the maintenance is to consider all $\binom{n}{2}$ pairs of disks and maintain the one of minimum distance with respect to the distance function (1). However, the Augmented Delaunay triangulation of S has the following property, the proof of which is omitted from this version of the paper.

Theorem 3. *Let B_1, B_2 be the closest pair in S. Then there exists an edge $[b_1, b_2]$ in ADT(S).*

The theorem above suggests that we only need to look at $O(n)$ edges in order to determine the closest pair, namely the edges of ADT(S). In particular, we need to maintain a *tournament tree* T on the edges of ADT(S). Before describing how to actually maintain T we need some definitions. Let t_1 and t_2 be two nodes of T. We say that $t_1 \prec t_2$ if the depth of t_1 is smaller than the depth of t_2 in T, or if t_1 and t_2 are of the same depth and t_1 is to the left of t_2 in T. A node t_1 is *adjacent* to a node t_2 in T if they have the same parent. A node t is a *loser* if its parent is its adjacent node. Finally, a node t is a *winner* if its parent is itself.

The certificates associated with T are the winner-loser relationships. The tree T changes due to changes in the winner-loser relationships or due to changes of the ADT(S), because of cocircularity and appearance/disappearance events. When a winner-loser relationship changes we simply propagate the new winner up the tree, deschedule the old winner-loser relationships and schedule the new ones. During this propagation we visit $O(\log n)$ nodes of the tree and schedule/deschedule $O(\log n)$ certificates in total. Hence the cost per winner-loser relationship change is $O(\log^2 n)$. When an edge disappears we replace the corresponding leaf node with the last loser leaf node and delete the last winner and loser leaf nodes. Then we propagate the last loser leaf node up the tree as in the case of a winner-loser relationship change. Again this takes $O(\log^2 n)$ time. Finally, when an edge appears we create two new leaf nodes in the tree: one for the new edge and one for the first leaf node. We attach the two new nodes under the current first leaf node and propagate the winner between these two nodes up the tree. Once again this takes $O(\log^2 n)$ time.

The number of changes in a single winner-loser relationship is constant for disks moving along pseudo-algebraic trajectories of constant degree. Hence the number of events that we have to process is dominated by the number of combinatorial changes of ADT(S), which is $O(n^3\beta(n))$. All, but the appearance event, are processed in $O(\log^2 n)$ time; the appearance event is processed in $O(n)$ time.

6 Kinetic Connectivity of Disks

In this section we discuss how to kinetically maintain the connectivity for a set of disks of different radii. The problem for unit disks has already been studied in [3].

The connectivity graph K of a set of disks S is defined as follows. The vertices of K are the centers of the disks in S. Two disks share an edge in K if they intersect. Let G be the subgraph of $\mathrm{ADT}(S)$ defined as follows. An edge e in $\mathrm{ADT}(S)$ belongs to G if and only if it is an edge between two non-trivial intersecting disks or between a trivial disk and its container disk. Clearly, G is a subgraph of the connectivity graph K of S (modulo multiple edges between two disks in $\mathrm{DT}(S)$). The important property of G is captured by the following theorem, the proof of which is omitted from this version of the paper.

Theorem 4. *If $B_1, B_2 \in S$ belong to the same connected component in K, then there exists a path in G that connects B_1 and B_2.*

In other words G is a spanning subgraph of K. This is really important since the size of K is $\Omega(n^2)$ in the worst case, whereas the size of G is $O(n)$. We can now maintain the connectivity of the disks using the dynamic graph data structure of Holm, de Lichtenberg and Thorup [7]. This data structure supports edge insertions and deletions in $O(\log^2 n)$ amortized time, and connectivity queries in $O(\log n / \log\log n)$ time. The graph that we maintain is the graph G defined above. Once we have $\mathrm{ADT}(S)$, maintaining G is really simple. First we color the edges of $\mathrm{ADT}(S)$ as follows: edges between non-intersecting non-trivial disks are *green*, edges between intersecting non-trivial disks are *orange* and edges between trivial disks and their containers are *red*. Clearly, G is the union of orange and red edges. The color of an edge changes if the corresponding disks become tangent. In particular, whenever a green edge becomes an orange edge or whenever an orange or a red edge appears we simply add it to G. Whenever an orange edge becomes a green edge or whenever an orange or a red edge disappears we simply delete it from G. Since the cost per insertion/deletion of edge in G is $O(\log^2 n)$, in the amortized sense, the cost per update of G is $O(\log^2 n)$ (amortized), except when we have an appearance event, in which case the update cost is $O(n)$. The number of times that two disks, moving along pseudo-algebraic trajectories of constant degree, can become tangent is constant. This implies that the number of events due to disk tangencies is $O(n^2)$. The total number of events for maintaining G is thus dominated by $O(n^3\beta(n))$, which is the number of times that the Delaunay triangulation of the set of disks can change combinatorially.

7 Near Neighbor Maintenance

Suppose that we have a set S of *non-intersecting* moving disks. Let P be a disk in S for which we want to know the disks in S that are within a certain, possibly time varying, distance R_P from P. Let C_P be the disk centered at P with radius R_P. The obvious approach is to maintain the distance from P to every other

disk in S and keep those that intersect C_P. In fact, we can do better than that. If we are maintaining the DT of S, the only disks that can enter or exit C_P are those that are end points of edges of the DT crossing C_P exactly once. This is the essence of the following theorem, the proof of which we omit from this paper.

Theorem 5. *Let $\mathcal{T}(S)$ be the DT(S) and let $P \in S$. If a disk $Q \in S$ enters/exists the disk C_P at some time t_0, then there exists an edge in $\mathcal{T}(S)$ between Q and some disk that intersects C_P.*

Maintaining the near neighbors of P then reduces to maintaining the DT of S and updating the set E_P of DT edges, one end disk of which intersects C_P and the other does not. The set E_P changes when disks enter or exit C_P. Edge flips due to the maintenance of DT(S) may also change E_P. In case we want to maintain the k-nearest neighbors of P the same idea applies with two slight modifications: (1) the distance R_P is defined to be the distance of the center of P from P_k, where P_k is the k-th nearest neighbor of P and (2) the edges of DT(S) adjacent to P_k are all included in E_P.

We omit the details of the algorithms since they are essentially the same as the corresponding algorithms for points in [9].

8 Conclusion

In this paper we presented how to kinetically maintain the Voronoi diagram for a set of disks moving in the plane. The key steps in the kinetization process were the introduction of the Augmented Delaunay triangulation and the establishment of the relationship between the local and global Delaunay properties. We showed how to maintain the closest pair of the set of disks and how to maintain a spanning subgraph of the connectivity graph of the set of disks using the Augmented Delaunay triangulation as the underlying structure. Finally, if the disks do not intersect, we discussed how to maintain the disks that are within a prescribed distance from a reference disk or how to maintain the k-nearest neighbors of a reference disk.

We strongly believe that the results presented in this paper can be generalized to more general *additively weighted Voronoi diagrams*, in which the weights can be positive as well as non-positive. We would also like to extend the results presented here to general smooth convex objects or to environments where obstacles are present. Finally, the best known lower bound on the number of combinatorial changes of the DT is $\Omega(n^2)$, whereas our upper bound is $O(n^3\beta(n))$. Given this upper bound, the algorithms presented here for maintaining the DT, the closest pair and disk connectivity are not efficient; it would be of interest to find kinetic data structures that solve these problems efficiently, or prove a tighter lower or upper bound on the number of combinatorial changes of the DT.

Acknowledgments. The author wishes to thank Leonidas J. Guibas, Olaf Hall-Holt, Aristides Gionis and Natasha Gelfand for useful discussions.

References

1. J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. *J. Algorithms*, 1:1–28, 1999.
2. M. Gavrilova and J. Rokne. Swap conditions for dynamic Voronoi diagrams for circles and line segments. *CAGD*, 16:89–106, 1999.
3. L. J. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic connectivity for unit disks. In *Proc. 16th ACM Symp. on Computat. Geom.*, pages 331–339, 2000.
4. L. J. Guibas, J. S. B. Mitchell, and T. Roos. Voronoi diagrams of moving points in the plane. In G. Schmidt and R. Berghammer, editors, *Proc. 17th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 570 of *LNCS*, pages 113–125. Springer, 1991.
5. L. J. Guibas, J. Snoeyink, and L. Zhang. Compact Voronoi diagrams for moving convex polygons. In Magnús M. Halldórsson, editor, *Proc. 7th SWAT*, volume 1851 of *LNCS*, pages 339–352. Springer, 2000.
6. L. J. Guibas and L. Zhang. Euclidean proximity and power diagram. In *Proc. 10th CCCG*, 1998.
7. J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proc. 30th ACM STOC*, pages 79–89, 1998.
8. H. Imai, M. Iri, and K. Murota. Voronoi diagram in the Laguerre geometry and its applications. *SIAM J. Comput.*, 14(1):93–105, 1985.
9. M. I. Karavelas and L. J. Guibas. Static and kinetic geometric spanners with applications. In *Proc. 12th ACM-SIAM SODA*, pages 168–176, 2001.
10. J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, 1991.
11. M. C. Lin and J. F. Canny. Efficient algorithms for incremental distance computation. In *Proc. IEEE Intern. Conf. Robot. Autom.*, volume 2, pages 1008–1014, 1991.
12. M. Sharir. Intersection and closest-pair problems for a set of planar discs. *SIAM J. of Comput.*, 14(2):448–468, May 1985.

Fast Fixed-Parameter Tractable Algorithms for Nontrivial Generalizations of Vertex Cover *

Naomi Nishimura[1], Prabhakar Ragde[1†], and Dimitrios M. Thilikos[2]

[1] Department of Computer Science, University of Waterloo,
Waterloo, Ontario, Canada, N2L 3G1.
{nishi, plragde}@uwaterloo.ca, FAX (519) 885-1208.
[2] Departament de Llenguatges i Sistemes Informàtics,
Universitat Politècnica de Catalunya, Campus Nord,
Mòdul C5, c/Jordi Girona Salgado, 1-3. E-08034, Barcelona, Spain.
sedthilk@lsi.upc.es.

Abstract. Our goal in this paper is the development of fast algorithms for recognizing general classes of graphs. We seek algorithms whose complexity can be expressed as a linear function of the graph size plus an exponential function of k, a natural parameter describing the class. Our classes are of the form $\mathcal{W}_k(\mathcal{G})$, graphs that can be formed by augmenting graphs in $\mathcal{G}$ by adding at most k vertices (and incident edges). If $\mathcal{G}$ is the class of edgeless graphs, $\mathcal{W}_k(\mathcal{G})$ is the class of graphs with a vertex cover of size at most k.

We describe a recognition algorithm for $\mathcal{W}_k(\mathcal{G})$ running in time $O((g+k)|V(G)| + (fk)^k)$, where g and f are modest constants depending on the class $\mathcal{G}$, when $\mathcal{G}$ is a minor-closed class such that each graph in $\mathcal{G}$ has bounded maximum degree, and all obstructions of $\mathcal{G}$ (minor-minimal graphs outside $\mathcal{G}$) are connected. If $\mathcal{G}$ is the class of graphs with maximum degree bounded by D (not closed under minors), we can still recognize graphs in $\mathcal{W}_k(\mathcal{G})$ in time $O(|V(G)|(D+k) + k(D+k)^{k+3})$.

Our results are obtained by considering minor-closed classes $\mathcal{G}$ for which all obstructions are connected graphs, and showing that the size of any obstruction for $\mathcal{W}_k(\mathcal{G})$ is $O(tk^7 + t^7k^2)$, where t is a bound on the size of obstructions for $\mathcal{G}$.

1 Introduction

One of the principal goals of algorithmic graph theory is to determine elegant and efficient ways to characterize classes of graphs. A particularly enticing approach applies readily to graph classes closed under minor containment (defined formally in Section 2). In their seminal work on graph minors, Robertson and Seymour [RS85b] proved that for any minor-closed graph class $\mathcal{G}$, there is a finite number of minor-minimal graphs (the set of *obstructions*, or *obstruction set*) in

* Research supported by the Natural Sciences and Engineering Research Council of Canada, Ministry of Education and Culture of Spain (grant number MEC-DGES SB98 0K148809), and EU project ALCOM-FT (IST-99-14186).

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 75–86, 2001.

the set of graphs outside of $\mathcal{G}$. As a consequence, G is in $\mathcal{G}$ if and only if no graph in the obstruction set of $\mathcal{G}$ is a minor of G; if the obstruction set of $\mathcal{G}$ is known, there exists a polynomial-time recognition algorithm for $\mathcal{G}$ [RS85a,RS95].

Unfortunately, finding the obstruction set of a class $\mathcal{G}$ is unsolvable in general [FRS87,FL94,van90] and appears to be a hard structural problem even for simple graph classes, largely due to the rapid explosion in the size of the obstructions [TUK94,Ram95,Thi00]. Following a brute force approach, it is possible to build a computer program enumerating graphs and searching among them for obstructions. A crucial drawback of this method is that there is no general way to bound the search space [van90,FL94]; more sophisticated methods are also possible [Din95]. It is thus necessary to determine upper bounds on the sizes of the obstructions for special graph classes. Results of this type have been obtained for graphs with bounded treewidth or pathwidth [Lag98] as well as more general graphs [Lag93].

In this paper we settle the question of the combinatorial growth for graph classes created from simpler ones. We augment a graph class by adding at most k vertices (and adjacent edges) to each graph in the class. More formally, for each graph class $\mathcal{G}$ and integer $k \geq 1$, we define $\mathcal{W}_k(\mathcal{G})$ to consist of graphs G which are *within k vertices of $\mathcal{G}$* ([DF99], page 196), namely all graphs G such that the vertices of G can be partitioned into sets S_1 and S_2, where $|S_1| \leq k$ and the subgraph of G induced on the vertices in S_2 is in the class $\mathcal{G}$. The notion of "within k" can be used to easily define the classes of graphs with vertex cover of size at most k, or graphs with vertex feedback set of size at most k: $\mathcal{G}$ is the class of edgeless graphs (i.e. $\text{ob}(\mathcal{G}) = \{K_2\}$), or the class of forests (i.e. $\text{ob}(\mathcal{G}) = \{K_3\}$), respectively. Moreover, if $\mathcal{G}$ is closed under taking of minors, then so is $\mathcal{W}_k(\mathcal{G})$ [FL88,LP98]. We conjecture that the sizes of obstructions for $\mathcal{G}$ and $\mathcal{W}_k(\mathcal{G})$ are related.

Conjecture 1. If $\mathcal{G}$ is a minor-closed graph class whose obstructions have no more than t vertices, then the size of the obstructions for $\mathcal{W}_k(\mathcal{G})$ depends only on k and t.

We show that the above conjecture is true for any class $\mathcal{G}$ in which no graph has degree greater than a fixed constant D, provided the class is minor-closed with all obstructions connected (in the proof of Lemma 8, we show that this is equivalent to the class being closed under disjoint union). In particular, we prove that for such a $\mathcal{G}$, any graph in the obstruction set of $\mathcal{W}_k(\mathcal{G})$ has size bounded by a polynomial in k, D, and (for $D \geq 3$) C, where C is an upper bound on the length of paths of degree-two vertices ("chains") in the obstruction set of $\mathcal{G}$. As intermediate steps in proving this result, we develop lemmas demonstrating the existence of leafy trees in sparse (bounded degree) graphs, constituting results of independent interest.

Making use of advances in construction of fixed-parameter tractable algorithms [DF99,Nie98], we employ the method of "reduction to a problem kernel" to obtain recognition algorithms such that an exponential function on k contributes only an additive term to the overall complexity. This builds on previous

work [Meh84,BG93]; our results can be seen as generalizations of the algorithms designed for the vertex-cover problem [BFR98,NR99,CKJ99]. Fast algorithms of this type have been generated for other problems [DF95,FPT95,KST99,DFS99, CCDF97,MR99].

We make use of our upper bound on the size of obstructions to obtain an $O((g(t)+k)|V(G)|+(f(k,t))^k)$-time algorithm recognizing $\mathcal{W}_k(\mathcal{G})$ when $\mathcal{G}$ satisfies the following three conditions: $\mathcal{G}$ is closed under taking of minors; each graph in $\mathcal{G}$ has degree bounded by D; and the obstructions for $\mathcal{G}$ are connected and of size bounded by t. Here g and f are polynomial functions with multiplicative constants depending on the class $\mathcal{G}$. Finally, we demonstrate that similar time complexities can be achieved even when the closure restriction is removed. We present an $O(|V(G)|(D+k)+k(D+k)^{k+3})$-time algorithm for the recognition of $\mathcal{W}_k(\mathcal{G})$, where $\mathcal{G}$ is the class of graphs with maximum degree bounded by D.

After establishing notation in Section 2, we demonstrate in Section 3 that we can obtain a bound on the size of a graph G when the disjoint union of stars is excluded as a minor and there are bounds on the degree and length of chains. Building on this result, in Section 4 we establish a bound on the size of obstructions for $\mathcal{W}_k(\mathcal{G})$. Sections 5 and 6, making use of these results, establish polynomial-time fixed-parameter tractable algorithms. In this conference version, many proofs are omitted or only sketched.

2 Preliminaries

We make use of standard graph-theoretic notation. A graph G has vertex set $V(G)$ and edge set $E(G)$. We define $\mathrm{nbr}_G(S)$ to be $\{v \in V(G) \mid \exists w \notin S, (v,w) \in E(G)\}$, and $G[S]$ to be the subgraph of G induced on S. For $v \in V(G)$, we define the *degree of* v *in* G or $\deg_G(v)$ to be $|\mathrm{nbr}_G(\{v\})|$, the set of *pendant vertices* $\mathrm{pend}(G)$ to be $\{v \in V(G) \mid \deg_G(v) = 1\}$, and the set of *internal vertices* $\mathrm{int}(G)$ to be $V(G) \setminus \mathrm{pend}(G)$. The degree bound of the graph G, denoted $\Delta(G)$, is $\max_{v \in V(G)} \deg_G(v)$. For $\mathcal{G}$ a finite graph class, we define $\text{max-size}(\mathcal{G}) = \max\{|V(G)| \mid G \in \mathcal{G}\}$; if $\Delta(\mathcal{G}) = \max\{\Delta(G) \mid G \in \mathcal{G}\}$ can be bounded above by a constant, $\mathcal{G}$ is *bounded degree*.

At times we will alter a graph by replacing paths by edges. We call a path of a graph G an *a-chain* if it has length (number of edges) a, all its internal vertices have degree two in G, and its end vertices are either adjacent or have degree not equal to two. We denote as $\mathrm{chain}(G)$ the largest a for which there exists an a-chain in G, setting $\mathrm{chain}(G) = 0$ when $E(G) = \emptyset$. A graph G is *resolved* if $\mathrm{chain}(G) = 1$. Finally, for any graph class $\mathcal{G}$, we define $\mathrm{chain}(\mathcal{G})$ to be $\max\{\mathrm{chain}(G) \mid G \in \mathcal{G}\}$.

Throughout this paper we will use $\mathcal{G}$ to denote a graph class that is closed under taking of minors or, equivalently, a *minor-closed* graph class. A graph G is a *minor of* a graph H if a graph isomorphic to G can be formed from H by a series of edge and vertex deletions and edge contractions. The *contraction* of an edge $e = (u,v)$ in G results in a graph G', in which u and v are replaced by a new vertex v_e ($V(G') = \{v_e\} \cup V(G) \setminus \{u,v\}$) and in which for every neighbor w of u

or v in G, there is an edge (w, v_e) in G'. An *obstruction* of $\mathcal{G}$ is a minor-minimal graph outside $\mathcal{G}$; we use $\mathrm{ob}(\mathcal{G})$ to denote the set of obstructions of $\mathcal{G}$, or the *obstruction set* of $\mathcal{G}$. We use $G \preceq H$ and $G \leq H$ to denote that G is a minor of or an induced subgraph of H, respectively.

We use K_r to denote the complete graph on r vertices and $K_{r,s}$ to denote the complete bipartite graph with r and s vertices in the two parts of the bipartition. The graph $K_{1,r}$ is also known as a *star*. We use L_r^{k+1} to denote the graph consisting of $k+1$ disjoint copies of $K_{1,r}$.

3 Excluding Disjoint Stars

We first establish Theorem 1, which shows that the absence of a L_r^{k+1} minor combined with a bound on the degree and on the length of a chain results in a graph of bounded size. The statement of the theorem looks complicated; the important fact is that the bound obtained is polynomial in k, r, the degree bound, and the chain length. In Section 4 we will make use of the theorem by showing that any sufficiently large graph with the aforementioned restrictions must contain L_r^{k+1} as a minor.

Theorem 1. *Any L_r^{k+1}-minor-free connected graph G where $\Delta(G) \leq D$ for $D \geq 3$ and $\mathrm{chain}(G) \leq C$ has at most $f(k, r, D, C)$ vertices, where $\gamma = \max\{r-1, D\}$ and*

$$f(k,r,D,C) = \begin{cases} (k+1)(D+1), & \text{if } r = 1 \\ (D+1)(k(D^2+1)+1), & \text{if } r = 2 \\ \frac{5}{2}(D+1)(k(D^2+1)+1))(CD+2) & \text{if } r = 3 \\ 5(CD+2)(\gamma(1+\gamma)(k(4(\gamma-1)^2+1)+1)-1) & \text{if } r \geq 4. \end{cases}$$

Proof. For each value of r, we prove the theorem by contradiction, assuming that $|V(G)| > f(k, r, D, C)$ and showing that as a consequence $L_r^{k+1} \preceq G$. The cases $r = 1$ and $r = 2$ follow by simple reasoning about greedy algorithms to find L_r^{k+1}. For the case in which $r = 3$, we find a minor of G which is a resolved tree with sufficiently many vertices, and then show that this tree contains L_r^{k+1}, by using the following Ramsey-theoretic lemmas.

Lemma 1. *If T is a tree, R is the set of vertices of degree at least r in T, and $|R| > k((\Delta(T))^2 + 1)$, then $L_r^{k+1} \preceq T$.* □

Lemma 2. *For G connected, $|\mathrm{int}(G)| \geq \max\{\frac{|\mathrm{pend}(G)|}{\Delta(G)}, \frac{|V(G)|}{\Delta(G)+1}\}$.* □

Lemma 3. *Any connected graph G contains as a minor a connected resolved graph H such that $|V(H)| > \frac{2|V(G)|}{\mathrm{chain}(G)\Delta(G)+2}$ and $\Delta(H) = \Delta(G)$.* □

Lemma 4. *Any connected resolved graph G contains as a minor a resolved tree T such that $|V(T)| \geq \frac{|V(G)|}{5}$ and $\Delta(T) \leq \Delta(G)$.* □

To conclude the case $r = 3$, we apply Lemmas 3 and 4 to conclude that G contains as a minor a resolved tree T such that $\Delta(T) \leq \Delta(G)$ and $|V(T)| > \frac{2|V(G)|}{5(\text{chain}(G)\Delta(G)+2)} \geq \frac{2|V(G)|}{5(CD+2)} \geq (D+1)(k(D^2+1)+1))$. Each internal vertex of T has degree at least 3. By Lemma 2, $|\text{int}(T)| \geq (k(\Delta(G)^2+1)+1) \geq (k(\Delta(T)^2+1)+1)$. Applying Lemma 1, we conclude that $L_r^{k+1} \preceq T \preceq G$, as needed to obtain a contradiction.

For the case $r \geq 4$, we apply the same reasoning as above, but continue with the following lemma.

Lemma 5. *For any resolved tree T, if $|V(T)| \geq 2$, then $|\text{pend}(T)| \geq \frac{1}{2}|V(T)|+1$.*

Proof sketch. Induction on the number of pendant vertices. □

Applying the line of argument from the $r = 3$ case shows that when $r \geq 4$, G contains as a minor a resolved tree T where $|\text{pend}(T)| > \frac{|V(G)|}{5(\text{chain}(G)\Delta(G)+2)} + 1 \geq \frac{|V(G)|}{5(CD+2)} + 1 \geq \gamma(1+\gamma)(k((2\gamma-2)^2+1)+1)$. This allows us to find a L_r^{k+1} minor. We construct a tree U from T by iteratively finding and contracting an edge (u,v), $u,v \notin \text{pend}(T)$, where $\deg_T(u) + \deg_T(v) \leq r+1$. Any vertex in U that is not the result of a contraction has degree at most $\Delta(T)$. A vertex in U that is the result of a contraction of an edge e will have degree at most $r-1$, as the sum of the degrees of the endpoints of e is at most $r+1$. We conclude that $\Delta(U) \leq \max\{r-1, \Delta(T)\} = m$. Clearly $\text{pend}(U) = \text{pend}(T)$ and for any edge $(u,v) \in E(U)$, $u,v \notin \text{pend}(U)$, it must be the case that $\deg_U(u)+\deg_U(v) \geq r+2$. Since $U \preceq T$, it will suffice to prove that $L_r^{k+1} \preceq U$.

By Lemma 2, $|\text{int}(U)| \geq \frac{|\text{pend}(U)|}{\Delta(U)} > (1+m)(k(2m-2)^2+1)$. Elementary arguments show that $U[\text{int}(U)]$ contains a matching M where $|M| \geq \frac{|\text{int}(U)|}{\Delta(U)+1} > k(2m-2)^2+1$. We form a new tree U' from U by contracting all the edges in M. Since the matching consists of edges in $U[\text{int}(U)]$, for each edge $(u,v) \in M$, neither u nor v is in $\text{pend}(U)$. For any edge $(u,v) \in M$, $\deg_U(u)+\deg_U(v) \geq r+2$. Consequently, the contraction of an edge in M will result in a new vertex of degree at least r in U'. For R the set of vertices in U' with degree at least r, clearly $|R| \geq |M|$, or $|R| > k(2m-2)^2+1$.

Since the degree of a vertex w in U' formed by contracting an edge (u,v) will be $\deg_U(u) + \deg_U(v) - 2$, and $\Delta(U) \leq m$, we can conclude that $\deg_{U'}(w) \leq 2m-2$. If a vertex is not an endpoint of a contracted edge, its degree in U' will be the same as its degree in U and therefore it will be at most $\Delta(T)$. As $\Delta(T) \geq 3$, we can conclude that $\Delta(U') \leq 2m-2$, or $|R| > k(\Delta(U')^2+1)$. Applying Lemma 1, we have the contradiction $L_r^{k+1} \preceq U' \preceq U \preceq T \preceq G$. □

We remark that an easy corollary of the proof in the case $r \geq 4$ is that any connected resolved graph of at least $10k$ vertices contains a spanning tree with at least k leaves. This remark can improve the time complexity for the algorithm solving the k-Leaf Spanning Tree problem ([DF95]; [DF99], pages 40–42), from $O(n + (2k)^{4k})$ to $O(n + (10k)^{2k})$.

4 Sizes of Obstructions

Since the function f in the statement of Theorem 1 is polynomial in its arguments, the following theorem gives a bound on the size of obstructions of within-k graph classes that is polynomial in k, the degree bound, and chain(ob($\mathcal{G}$)).

Theorem 2. *If $\mathcal{G}$ is a bounded degree minor-closed graph class then* max-size(ob($\mathcal{W}_k(\mathcal{G})$)) $\leq f(k, \Delta(\mathcal{G})+1, k+\Delta(\mathcal{G})+1, (k+1)\text{chain}(\text{ob}(\mathcal{G})))$.

Proof. Work of Dinneen [Din97] shows that for any minor-closed disjoint-union-closed graph class $\mathcal{G}$, if a graph in ob($\mathcal{W}_k(\mathcal{G})$) is the disjoint union of $r+1$ nonempty connected graphs $C_0, \ldots, C_r$, then there exists a partition of $k+1$ into at most $r+1$ integers $k_0, k_1, \ldots, k_r$ such that for $i = 0, \ldots, r$, $C_i \in \text{ob}(\mathcal{W}_{k_i}(\mathcal{G}))$. Thus it suffices to prove this theorem for the connected obstructions in ob($\mathcal{W}_k(\mathcal{G})$).

We first bound chain(ob($\mathcal{W}_k(\mathcal{G})$)) and $\Delta((\text{ob}(\mathcal{W}_k(\mathcal{G})))$. Consider a graph $H \in \text{ob}(\mathcal{W}_k(\mathcal{G})))$ and an edge $e \in E(H)$. We form a new graph $H' = (V(H), E(H) - \{e\})$ by removing e from H. Since H' is smaller than H, $H \in \text{ob}(\mathcal{W}_k(\mathcal{G}))$, and ob($\mathcal{W}_k(\mathcal{G})$) is a set of minor-minimal elements, we can conclude that $H' \in \mathcal{W}_k(\mathcal{G})$. Consequently, by the definition of $\mathcal{W}_k(\mathcal{G})$, we can partition $V(H')$ into sets S_1 and S_2 where $|S_1| \leq k$ and $H'[S_2] \in \mathcal{G}$. It is not difficult to see that $\Delta(H') \leq \Delta(H'[S_2]) + |S_1|$. Since that $\Delta(H'[S_2]) \leq \Delta(\mathcal{G})$, $|S_1| \leq k$, and $\Delta(H) \leq \Delta(H') + 1$, we have $\Delta(H) \leq k + \Delta(\mathcal{G}) + 1$. The bound on chain($H$) comes from the following lemma.

Lemma 6. *For any minor-closed disjoint-union-closed graph class $\mathcal{G}$,* chain(ob($\mathcal{W}_k(\mathcal{G})$)) $\leq (k+1)\text{chain}(\text{ob}(\mathcal{G}))$. □

If $L = L^{k+1}_{\Delta(\mathcal{G})+1}$ is a minor of H, then since H is connected and L is not, L must be a proper minor of H. But this leads to a contradiction. Since ob($\mathcal{W}_k(\mathcal{G})$) is defined to be a minor-minimal set of obstructions for $\mathcal{W}_k(\mathcal{G})$, $L \in \mathcal{W}_k(\mathcal{G})$. By the definition of $\mathcal{W}_k(\mathcal{G})$, L must contain a set S, $|S| \leq k$, such that $G' = L[V(L) - S] \in \mathcal{G}$. Clearly, $\Delta(G') \leq \Delta(\mathcal{G})$ and therefore $K_{1,\Delta(\mathcal{G})+1}$ cannot be a subgraph of G'. S must then contain at least one vertex in each of the disjoint copies of $K_{1,\Delta(\mathcal{G})+1}$ as a subgraph in L, as otherwise G' contains a copy of $K_{1,\Delta(\mathcal{G})+1}$ as a subgraph. Since $|S| \leq k$ and the number of copies of $K_{1,\Delta(\mathcal{G})+1}$ is $k+1$, we obtain a contradiction.

We can thus conclude that H is L-minor free. Therefore, Theorem 1 implies that $|V(H)| \leq f(k, \Delta(\mathcal{G})+1, k+\Delta(\mathcal{G})+1, (k+1)\text{chain}(\text{ob}(\mathcal{G})))$. □

For $\Delta(\mathcal{G}) = 0$, Theorem 2 implies that the obstructions for the class of graphs with vertex cover at most k have size at most $(k+1)(k+2)$. In the case where $\Delta(\mathcal{G}) = 1$, the upper bound is $(k+3)(k((k+2)^2+1)+1)$ or $O(k^4)$; in the case $\Delta(\mathcal{G}) = 2$, the upper bound is $\frac{5}{2}(k+4)(k((k+3)^2+1)+1)(\text{chain}(\text{ob}(\mathcal{G}))(k+3)+2)$ or $O(\text{chain}(\text{ob}(\mathcal{G}))k^5)$; and when $\Delta(\mathcal{G}) \geq 3$, the upper bound is in $O(\text{chain}(\text{ob}(\mathcal{G}))k^2(k+\Delta(\mathcal{G}))^5)$.

A minor-closed graph class $\mathcal{G}$ has bounded degree if and only if $K_{1,\Delta(\mathcal{G})+1}$ is one of its obstructions. This implies that if t is the size of the biggest obstruction in ob($\mathcal{G}$), $\Delta(G) \leq t$. Since chain(ob($\mathcal{G}$)) $\leq t$, we can now conclude the following.

Corollary 1. *If $\mathcal{G}$ is a bounded degree minor-closed disjoint-union-closed graph class then* $\text{max-size}(\text{ob}(\mathcal{W}_k(\mathcal{G}))) = O(tk^7 + t^6k^2)$ *where* $t = \text{max-size}(\text{ob}(\mathcal{G}))$.

5 Recognizing Graphs within k Vertices of a Bounded-Degree Minor-Closed Class

When k is considered to be part of the input and $\mathcal{G}$ is characterized by a nontrivial property, the problem of deciding whether $G \in \mathcal{W}_k(\mathcal{G})$ is NP-complete [LP80]. In contrast, when k is viewed as a parameter and $\mathcal{G}$ is any bounded degree minor-closed disjoint-union-closed graph class, we are able to obtain a fast fixed-parameter tractable algorithm, as shown in this section.

Theorem 3. *Let $\mathcal{G}$ be any minor-closed graph class with the following properties: (1) $\mathcal{G}$ contains only graphs of degree bounded by $D \geq 3$; (2)* $\text{chain}(\text{ob}(\mathcal{G})) \leq C$*; (3) all obstructions of $\mathcal{G}$ are connected; and (4) checking whether $G \in \mathcal{G}$ can be executed in $O(|V(G)| \cdot \sigma_{\mathcal{G}})$ time where $\sigma_{\mathcal{G}}$ is a constant depending on the class $\mathcal{G}$. Then, for any k, there exists an $O((D+k+\sigma_{\mathcal{G}})|V(G)| + \sigma_{\mathcal{G}}(f(k, D+1, D+k, C(k+1)))^k)$-time algorithm that decides if a graph G is in $\mathcal{W}_k(\mathcal{G})$ and, if so, produces a set $S \subseteq V(G), |S| \leq k$, such that $G[V(G) - S] \in \mathcal{G}$.*

Proof. We propose the following algorithm.

1 $A \leftarrow \{v \in V(G) \mid \deg_G(v) > D + k\}$.
2 If $|A| > k$ then answer NO, else $H \leftarrow G[V(G) - A]$.
3 $k' \leftarrow k - |A|$.
4 While there exists in H a β-chain where $\beta > C(k'+1)$,
replace it with a $(C(k'+1))$-chain.
5 Remove from H all its connected components that are in $\mathcal{G}$.
6 If $|V(H)| \geq f(k', D+1, D+k, C(k'+1))$ then answer NO
7 Answer YES iff $H[V(H) - S'] \in \mathcal{G}$ for some $S' \subseteq V(H), |S| \leq k'$
$(S \leftarrow A \cup S')$

The following technical lemmas are needed to prove correctness.

Lemma 7. *For any minor-closed graph class $\mathcal{G}$, for any $a \geq \text{chain}(\text{ob}(\mathcal{G}))$, and for any $G \in \mathcal{G}$, the subdivision of any edge in any a-chain of G results in a graph that is also a member of $\mathcal{G}$.* □

Lemma 8. *If $\mathcal{G}$ is a minor-closed disjoint-union-closed graph class, then, for any $k \geq 0$, no graph in* $\text{ob}(\mathcal{W}_k(\mathcal{G}))$ *contains a member of $\mathcal{G}$ as a connected component.* □

To see that the algorithm is correct, we first observe that if there exists a solution, it will contain every vertex of degree greater than $D + k$ and hence $|A| \leq k$ (step 2). The problem is then reduced to the question of whether $H = G[V(G) - A]$ is in $\mathcal{W}_{k'}(\mathcal{G})$, for $\Delta(H) \leq D + k$. As a consequence of Lemma 6

and property 2 of $\mathcal{G}$, $\text{chain}(\text{ob}(\mathcal{W}_{k'}(\mathcal{G}))) \leq C(k'+1)$. By combining this fact with Lemma 7 (for $a = C(k'+1)$), since $\mathcal{G}$ is minor-closed we can conclude that step 4 preserves membership in $\mathcal{W}_{k'}(\mathcal{G})$ and results in a graph H where $\text{chain}(H) \leq C(k'+1)$. Property 3 of $\mathcal{G}$ and Lemma 8 imply that membership in $\mathcal{W}_{k'}(\mathcal{G})$ is invariant under the removal from H of all its connected components that are members of $\mathcal{G}$. In this way, the problem is reduced to determining whether $J \in \mathcal{W}_{k'}(\mathcal{G})$ where $\text{chain}(J) \leq C(k'+1)$ and $\Delta(J) \leq D+k$, where for $J_1, \ldots, J_m$ the connected components of J, $J_i \notin \mathcal{G}$. Each J_i must then contain at least one vertex of a possible solution S.

If $J \in \mathcal{W}_{k'}(\mathcal{G})$, there exists a partition $(S_1, \ldots, S_m)$ of S where for $i = 1 \ldots m$, $S_i = S \cap V(J_i)$, $k_i = |S_i|$, $J_i[V(J_i) - S_i] \in \mathcal{G}$, and therefore $J_i \in \mathcal{W}_{k_i}(\mathcal{G})$. By the proof of Theorem 2, for $i = 1, \ldots, m$, J_i is $L_{D+1}^{k_i+1}$-minor free. Since for $i = 1, \ldots, m$, $\Delta(J_i) \leq D+k$ and $\text{chain}(J_i) \leq C(k'+1)$, we apply Theorem 1 to show that $V(J_i) \leq f(k_i, D+1, D+k, C(k'+1))$. Since J is composed of its connected components, $|V(J)| \leq \sum_{i=1,\ldots,m} f(k_i, D+1, D+k, C(k'+1))$. The maximum value of $|V(J)|$ is achieved when $m = 1$, due to the restrictions that $\forall_{1 \leq i \leq m}\ k_i \geq 1$ and $\sum_{i=1}^{m} k_i = k'$, as f is a monotonically increasing function and hence $f(k_1) + f(k_2) \leq f(k_1 + k_2)$. We can then conclude that $|V(J)| \leq f(k', D+1, D+k, C(k'+1))$, justifying step 6.

To determine the complexity of the algorithm, we first observe that steps 1 through 3 run in $O(|V(G)|(D+k))$ time and step 4 in $O(|V(G)|)$ time. As a consequence of property 4 of $\mathcal{G}$, step 5 can be executed in $O(|V(G)|\sigma_{\mathcal{G}})$ time. Step 7 can be implemented by checking whether $J[V(J) - S'] \in \mathcal{G}$ for all sets $S' \subseteq V(J), |S| \leq k'$, in $O(\sigma_{\mathcal{G}}(f(k', D+1, D+k, C(k'+1)))^{k'})$ time. The overall time complexity of the algorithm is thus in $O((D+k+\sigma_{\mathcal{G}})|V(G)| + \sigma_{\mathcal{G}}(f(k, D+1, D+k, C(k+1)))^k)$, as claimed. □

As we mentioned at the end of Section 4, both $\Delta(\mathcal{G})$ and $\text{chain}(\text{ob}(\mathcal{G}))$ are bounded by $t = \text{max-size}(\text{ob}(\mathcal{G}))$ and hence the complexity of Theorem 3 can be rewritten as $O((t+k+\sigma_{\mathcal{G}})|V(G)| + \sigma_{\mathcal{G}}(f(k, t+1, t+k, t(k+1)))^k) = O((t+k+\sigma_{\mathcal{G}})|V(G)| + \sigma_{\mathcal{G}}(O(tk^2(t+k)^5)^k)$.

Suppose now that $\mathcal{G}$ satisfies conditions 1–3 of Theorem 3 and that $\text{ob}(\mathcal{G})$ is known. As $K_{1,\Delta(\mathcal{G})+1} \in \text{ob}(\mathcal{G})$, all the graphs in $\mathcal{G}$ have pathwidth bounded by D [BRST91]. Using standard techniques on graphs with bounded treewidth [Arn85], one can construct an algorithm that checks whether $H \leq G$ in $O(\chi(t, |V(H)|)|V(G)|)$ time, where t is a bound on the treewidth of G and χ is a super-polynomial function. Consequently, if we know $\text{ob}(\mathcal{G})$, then each of the checks for membership in $\mathcal{G}$ applied in steps 5 and 7 can be done in $O(|\text{ob}(\mathcal{G})| \cdot \chi(D, \text{max-size}(\text{ob}(\mathcal{G})))|\mathcal{G}|)$ time and $\sigma_{\mathcal{G}}$ can be replaced by $|\text{ob}(\mathcal{G})| \cdot \chi(t, \text{max-size}(\text{ob}(\mathcal{G})))$. As a result, condition 4 will be satisfied.

6 Recognizing Graphs within k Vertices of a Bounded-Degree Class

In this section we extend our techniques to provide a solution to a generalization of the vertex cover problem that is not closed under taking of minors. Let $\mathcal{G}_D$

denote the class of graphs with maximum degree at most D. We define the problem ALMOST D-BOUNDED GRAPH as deciding whether or not a graph G is in $\mathcal{W}_k(\mathcal{G}_D)$, and if so, finding a set $S \subseteq V(G)$ where $|S| \leq k$ and $G[V(G)-S] \in \mathcal{G}_D$. If $D = 0$, the problem is simply VERTEX COVER. To the best of our knowledge, the best algorithm so far for the general problem can be derived as a subcase of the $\Pi_{i,j,k}$ GRAPH MODIFICATION PROBLEM [Cai96]. The running time obtained is $O((D+1)^k|V(G)|^{D+1})$. The function given below improves on this bound (Theorem 4) but the algorithm which uses it as a subroutine is even faster.

```
Function Search_D(G, k)
1   If k < 0 then return NO
2   If Δ(G) > D + k, choose v ∈ V(G) with deg_G(v) > D + k
       if Search_D(G − v, k − 1) =NO, then return NO
       else return {v} ∪ Search_D(G − v, k − 1)
3   If Δ(G) ≤ D, then return ∅
4   Choose v ∈ V(G) with D < d_G(v) ≤ D + k.
       (If there exists no such vertex in G then return NO)
5   If, for some u ∈ N_G(v) ∪ {v}, Search_D(G − u, k − 1) is a set
       then return that set; else return NO
```

```
Algorithm
1   A ← {v ∈ V(G) | deg_G(v) > D + k}
2   If |A| > k then return NO, else H ← G[V(G) − A].
3   k' ← k − |A|.
4   F ← {v ∈ V(H) | deg_H(v) ≤ D}, C ← V(H) − F
5   B ← {v ∈ F | v adjacent, in H, to a vertex in C}
6   If |B ∪ C| ≥ (D + k + 1)k' + (D + k)(D + k + 1)k' then return NO
7   return A∪Search_D(H[B ∪ C], k')
```

Theorem 4. *For $\mathcal{G}_D$ the class of graphs with maximum degree at most $D \geq 0$, function* $\mathsf{Search}_D(G,k)$ *solves* ALMOST D-BOUNDED GRAPH *in time* $O(|V(G)|(D+k)^{k+1})$. □

Theorem 5. *For $\mathcal{G}_D$ the class of graphs with maximum degree at most $D \geq 0$, the algorithm above solves* ALMOST D-BOUNDED GRAPH *in time* $O(|V(G)|(D+k) + k(D+k)^{k+3})$.

Proof. If $G \in \mathcal{W}_k(\mathcal{G}_D)$, then there exists a set $S, |S| \leq k$ where $\Delta(G[V(G)-S]) \leq D$, and hence all the vertices of G that are not in S have degree at most $D+k$, yielding $A \subseteq S$. If the set A has more than k vertices, then G cannot be a member of $\mathcal{W}_k(\mathcal{G}_D)$ (steps 1 and 2). It is now easy to see that $G \in \mathcal{W}_k(\mathcal{G}_D)$ if and only if $H \in \mathcal{W}_{k-|A|}(\mathcal{G}_D)$. Therefore, the problem is reduced to the question of whether or not $H \in \mathcal{W}_{k'}(\mathcal{G}_D)$, where $\Delta(H) \leq D+k$. The vertices of $V(H)$ that are not in $B \cup C$ have all degree at most D and are adjacent to vertices that have degree at most D. As a consequence $H \in \mathcal{W}_{k'}(\mathcal{G}_D)$ if and only if $H[B \cup C] \in \mathcal{W}_{k'}(\mathcal{G}_D)$.

The graph $H' = H[B \cup C]$ has maximum degree at most $D + k$ and any vertex of degree at most D is adjacent to a vertex of degree greater than D in H'.

We let B' be the vertices in $V(H') - S$ of degree at most D and let $C' = V(H') - S - B'$. Each vertex in C' will be adjacent to at least one vertex in S as otherwise $H'[V(H') - S]$ will contain a vertex of degree greater than D. Since $\Delta(S) \leq \Delta(H') \leq k + D$, C' will contain at most $D + k$ neighbours of each vertex in S, or a total of at most $k'(D + k)$ vertices. We have established that $|C'| + |S| \leq (D + k + 1)k'$. We let J be the vertices of B' that are adjacent to vertices in $C' \cup S$ and observe that since each vertex in $C' \cup S$ can have at most $k + D$ neighbours in J, $|J| \leq (D + k)(|C'| + |S|) \leq (D + k)(D + k + 1)k'$. Any vertex in B' is adjacent to a vertex in C'. Therefore $B' = J$ and $|V(H')| = |C'| + |S| + |B'| \leq (D + k + 1)k' + (D + k)(D + k + 1)k'$. By Theorem 4, step 7 requires time $O(|V(H')|(D + k)^{k+1}) = O(k(D + k)^{k+3})$. Since steps 1–6 can be trivially implemented in $O((D + k)|V(G)|)$ time, the total running time is as claimed. □

7 Future Work

Ideally, we would like to remove the restriction of closure under disjoint union from our results. As a consequence of Dinneen's work [Din97], an obstruction of $\mathcal{W}_k(\mathcal{G})$ can have at most $k + 1$ components if $\mathcal{G}$ is a minor-closed disjoint-union-closed graph class. Unfortunately, without this restriction, $\mathcal{G}$ may have disconnected obstructions, invalidating the proof of Lemma 8. It would be an interesting result, requiring new techniques, to determine an upper bound on the number of connected components in an obstruction of $\mathcal{W}_k(\mathcal{G})$ when $\mathcal{G}$ is not a disjoint-union-closed graph class. It would also be nice to use the approach of Section 6 to find a better algorithm for the general $\Pi_{i,j,k}$-Graph Modification Problem.

References

Arn85. Stefan Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability – A survey. *BIT*, 25:2–23, 1985.

BFR98. R. Balasubramanian, Michael Fellows, and Venkatesh Raman. An improved fixed-parameter algorithm for vertex cover. *Inform. Proc. Letters*, 65:163–168, 1998.

BG93. Jonathan F. Buss and Judy Goldsmith. Nondeterminism within P. *SIAM J. Computing*, 22:560–572, 1993.

BRST91. Daniel Bienstock, Neil Robertson, Paul D. Seymour, and Robin Thomas. Quickly excluding a forest. *J. Comb. Theory Series B*, 52:274–283, 1991.

Cai96. Leizhen Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Inform. Proc. Letters*, 58:171–176, 1996.

CCDF97. Liming Cai, Jianer Chen, Rodney Downey, and Michael Fellows. Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, 84:119–138, 1997.

CKJ99. J. Chen, I.A. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. In *Proceedings of the 25th International Workshop on Graph-Theoretical Concepts in Computer Science*, pages 313–324. Springer-Verlag, 1999.

DF95. Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM J. Comput.*, 24:873–921, 1995.

DF99. Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Springer-Verlag, 1999.

DFS99. Rodney G. Downey, Michael R. Fellows, and Ulrike Stege. Computational tractability: the view from Mars. *Bulletin of the European Association of Theoretical Computer Science*, 69:73–97, 1999.

Din95. Michael J. Dinneen. *Bounded Combinatorial Width and Forbidden Substructures.* PhD thesis, Department of Computer Science, University of Victoria, 1995.

Din97. Michael J. Dinneen. Too many minor order obstructions (for parametrized lower ideals). *Journal of Universal Computer Science*, 3(11):1199–1206, 1997.

FL88. Michael R. Fellows and Michael A. Langston. Nonconstructive tools for proving polynomial-time decidability. *J. ACM*, 35:727–739, 1988.

FL94. Michael R. Fellows and Michael A. Langston. On search, decision, and the efficiency of polynomial-time algorithms. *J. Comp. Syst. Sc.*, 49(3):769–779, 1994.

FPT95. Martin Farach, Teresa Przytycka, and Mikkel Thorup. On the agreement of many trees. *Inform. Proc. Letters*, 55:297–301, 1995.

FRS87. Harvey Friedman, Neil Robertson, and Paul D. Seymour. The metamathematics of the graph minor theorem. *Contemporary Mathematics*, 65:229–261, 1987.

KST99. Haim Kaplan, Ron Shamir, and Robert E. Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM J. Comput.*, 28(5):1906–1922, 1999.

Lag93. Jens Lagergren. An upper bound on the size of an obstruction. In Neil Robertson and Paul Seymour, editors, *Graph Structure Theory, Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference, Seattle WA, June 1991*, pages 601–621, Providence, RI, 1993. American Math. Soc. Contemp. Math. 147.

Lag98. Jens Lagergren. Upper bounds on the size of obstructions and intertwines. *J. Combin. Theory Ser. B*, 73:7–40, 1998.

LP80. J. M. Lewis and Christos H. Papadimitriou. The node-deletion problem for hereditary properties is NP-complete. *J. Comp. Syst. Sc.*, 20:219–230, 1980.

LP98. Michael A Langston and Barbara C. Plaut. On algorithmic applications of the immersion order. An overview of ongoing work presented at the Third Slovenian International Conference on Graph Theory. *Discrete Mathematics*, 182(1-3):191–196, 1998.

Meh84. Kurt Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms.* Springer Verlag, Berlin, 1984.

MR99. Meena Mahajan and Venkatesh Raman. Parameterizing above guaranteed values: maxsat and maxcut. *J. Algorithms*, 31(2):335–354, May 1999.

Nie98. Rolf Niedermeier. Some prospects for efficient fixed parameter algorithms. In *Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM'98)*, volume 1521 of *Lecture Notes in Computer Science*, pages 168–185, 1998.

NR99. Rolf Niedermeier and Peter Rossmanith. Upper bounds for vertex cover further improved. In *C. Meinel, S. Tison, editors, Proceedings of the 16th Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *Lecture Notes in Computer Science*, pages 561–570, 1999.

Ram95. Siddharthan Ramachandramurthi. A lower bound for treewidth and its consequences. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Proceedings 20th International Workshop on Graph Theoretic Concepts in Computer Science WG'94*, volume 903 of *Lecture Notes in Computer Science*, pages 14–25. Springer Verlag, 1995.

RS85a. Neil Robertson and Paul D. Seymour. Disjoint paths – A survey. *SIAM J. Alg. Disc. Meth.*, 6:300–305, 1985.

RS85b. Neil Robertson and Paul D. Seymour. Graph minors — A survey. In I. Anderson, editor, *Surveys in Combinatorics*, pages 153–171. Cambridge Univ. Press, 1985.

RS95. Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory Series B*, 63:65–110, 1995.

Thi00. Dimitrios M. Thilikos. Algorithms and obstructions for linear-width and related search parameters. *Discrete Applied Mathematics*, 105:239–271, 2000.

TUK94. Atsushi Takahashi, Shuichi Ueno, and Yoji Kajitani. Minimal acyclic forbidden minors for the family of graphs with bounded path-width. *Discrete Mathematics*, 127(1/3):293–304, 1994.

van90. Jan van Leeuwen. Graph algorithms. In *Handbook of Theoretical Computer Science, A: Algorithms and Complexity Theory*, pages 527–631, Amsterdam, 1990. North Holland Publ. Comp.

Deciding Clique-Width for Graphs of Bounded Tree-Width
(Extended Abstract)

Wolfgang Espelage, Frank Gurski*, and Egon Wanke

Department of Computer Science, D-40225 Düsseldorf, Germany
{espelage,gurski,wanke}@cs.uni-duesseldorf.de

Abstract. We show that there exists a linear time algorithm for deciding whether a graph of bounded tree-width has clique-width k for some fixed integer k.

1 Introduction

The clique-width of a graph is defined by a composition mechanism for vertex-labeled graphs, see [CO00]. The operations are the vertex disjoint union, the addition of edges between all pairs of vertices with a given pair of labels, and the relabeling of vertices. The clique-width of a graph G is the minimum number of labels needed to define G. Graphs of bounded clique-width are especially interesting from an algorithmic point of view. A lot of NP-complete graph problems can be solved in polynomial time for graphs of bounded clique-width if the composition of the graph is explicitly given. For example, all graph properties which are expressible in monadic second order logic with quantifications over vertices and vertex sets (MSO_1-logic) are decidable in linear time on graphs of bounded clique-width, see [CMR00]. The MSO_1-logic has been extended by counting mechanisms which allow the expressibility of optimization problems concerning maximal or minimal vertex sets, see [CMR00]. All these graph problems expressible in extended MSO_1-logic can be solved in polynomial time on graphs of bounded clique-width. Furthermore, a lot of NP-complete graph problems which are not expressible in MSO_1-logic or extended MSO_1-logic like Hamiltonicity and certain partitioning problems can also be solved in polynomial time on graphs of bounded clique-width, see [KR01,Wan94].

If a graph G has clique-width at most k then the edge complement $\overline{G}$ has clique-width at most $2k$, see [CO00]. Distance hereditary graphs have clique-width at most 3, see [GR00]. The set of all graphs of clique-width at most 2 is the set of all labeled cographs. The clique-width of permutation graphs, interval graphs, grids and planar graphs is not bounded by some fixed integer k, see [GR00]. An arbitrary graph with n vertices has clique-width at most $n - r$, if $2^r < n - r$, see [Joh98]. The recognition problem for graphs of clique-width

* The work of the second author was supported by the German Research Association (DFG) grant WA 674/9-1.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 87–98, 2001.

at most k is still open for $k \geq 4$. Clique-width of at most 3 is decidable in polynomial time, see [CHL+00]. Clique-width of at most 2 is decidable in linear time, see [CPS85].

A famous class of graphs for which a lot of NP-complete graph problems can be solved in polynomial time is the class of graphs of bounded tree-width, see Bodlaender [Bod98] for a survey. For every fixed integer k, it is decidable in linear time whether a given graph G has tree-width k, see [Bod96]. All graph properties expressible in monadic second order logic with quantifications over vertex sets and edge sets (MSO_2-logic) are decidable in linear time for graphs of bounded tree-width by dynamic programming, see [Cou90]. The MSO_2-logic has also been extended by counting mechanisms to express optimization problems which can then be solved in polynomial time for graphs of bounded tree-width, see [ALS91].

Every graph of tree-width at most k has clique-width at most $2^{k+1} + 1$, see [CO00]. Since the set of all cographs already contains all complete graphs, the set of all graphs of clique-width at most 2 does not have bounded tree-width. In [GW00], it is shown that every graph of clique-width k which does not contain the complete bipartite graph $K_{n,n}$ for some $n > 1$ as a subgraph has tree-width at most $3k(n-1) - 1$.

A simple algorithm to decide a graph property on a graph of bounded tree-width can be obtained from a partition of all l-terminal graphs into a finite number of equivalence classes, see, for example, [Bod98]. An l-terminal graph is a graph with a list of l distinct vertices called terminals. Two l-terminal graphs G and H can be combined to a graph $G \circ H$ by taking the disjoint union of G and H and then identifying the i-th terminal of G with the i-th terminal of H for $1 \leq i \leq l$. They are called equivalent with respect to a graph property Π if for all l-terminal graphs J the answer to Π is the same for $G \circ J$ and $H \circ J$. A graph property Π is decidable in linear time on a graph of bounded tree-width if there is a finite number of equivalence classes with respect to Π for all l-terminal graphs and all $l \geq 0$. The linear time algorithm first computes a binary tree-decomposition T for G and then bottom-up the equivalence class for every l-terminal graph G' represented by a complete subtree T' of T. The equivalence class of G' defined by subtree T' with root u' is computable in time $O(1)$ from the classes of the two l-terminal graphs defined by the two subtrees in $T' - \{u'\}$.

In this paper, we prove that the graph property "clique-width at most k" divides the set of all l-terminal graphs into a finite number of equivalence classes. This implies that there exists a linear time algorithm for deciding "clique-width at most k" for graphs of bounded tree-width. Since every graph of tree-width r has clique-width at most $2^{r+1} + 1$, there is also a linear time algorithm for computing the clique-width of a graph of bounded tree-width by testing "clique-width at most k" for $k = 1, \ldots, 2^{r+1} + 1$. The proofs[1] of lemma 1, 5, 6, 7, and theorem 1 are omitted due to space limitations. Note that it remains open whether the clique-width k property is expressible in MSO_2-logic and whether "clique-width at most k" is decidable in polynomial time for arbitrary graphs.

[1] A complete version can be found at `www.cs.uni-duesseldorf.de/~wanke`.

2 Basic Definitions

We work with finite undirected *graphs* $G = (V_G, E_G)$, where V_G is a finite set of *vertices* and $E_G \subseteq \{\{u, v\} \mid u, v \in V_G,\ u \neq v\}$ is a finite set of *edges*. Graph $J = (V_J, E_J)$ is a *subgraph* of G if V_J is a subset of V_G and E_J is a subset of $E_G \cap \{\{u, v\} \mid u, v \in V_J,\ u \neq v\}$. J is an *induced subgraph* of G if additionally $E_J = \{\{u, v\} \in E_G \mid u, v \in V_J\}$. To distinguish between the vertices of (non-tree) graphs and trees, we call the vertices of the trees simply *nodes*.

The notion of clique-width for labeled graphs is defined by Courcelle and Olariu in [CO00]. Let $[k] := \{1, \ldots, k\}$ be the set of all integers between 1 and k. A *k-labeled graph* $G = (V_G, E_G, \mathrm{lab}_G)$ is a graph (V_G, E_G) whose vertices are labeled by some mapping $\mathrm{lab}_G : V_G \to [k]$. A labeled graph $J = (V_J, E_J, \mathrm{lab}_J)$ is a subgraph of G if $V_J \subseteq V_G$, $E_J \subseteq E_G \cap \{\{u, v\} \mid u, v \in V_J,\ u \neq v\}$ and $\mathrm{lab}_J(u) = \mathrm{lab}_G(u)$ for all $u \in V_J$. The labeled graph which consists of a single vertex labeled by $t \in [k]$ is denoted by $\bullet_t$.

Definition 1 (Clique-width, [CO00]). *Let k be some positive integer. The class CW_k of labeled graphs is recursively defined as follows.*

1. *The single vertex graph $\bullet_t$ for some $t \in [k]$ is in CW_k.*
2. *Let $G = (V_G, E_G, lab_G) \in CW_k$ and $J = (V_J, E_J, lab_J) \in CW_k$ be two vertex disjoint labeled graphs. Then $G \oplus J := (V', E', lab')$ defined by $V' := V_G \cup V_J$, $E' := E_G \cup E_J$, and*

$$lab'(u) \ := \ \begin{cases} lab_G(u) & \text{if } u \in V_G \\ lab_J(u) & \text{if } u \in V_J \end{cases}, \ \forall u \in V'$$

   is in CW_k.
3. *Let $i, j \in [k]$ be two distinct integers and $G = (V_G, E_G, lab_G) \in CW_k$ be a labeled graph then*
   a) $\rho_{i \to j}(G) := (V_G, E_G, lab')$ defined by

$$lab'(u) \ := \ \begin{cases} lab_G(u) & \text{if } lab_G(u) \neq i \\ j & \text{if } lab_G(u) = i \end{cases}, \ \forall u \in V_G$$

   is in CW_k and
   b) $\eta_{i,j}(G) \ := \ (V_G, E', lab_G)$ defined by

$$E' := E_G \cup \{\{u, v\} \mid \ u, v \in V_G,\ u \neq v,\ lab(u) = i,\ lab(v) = j\}$$

   is in CW_k.

The clique-width *of a labeled graph G is the smallest integer k such that $G \in CW_k$.*

An expression X built with the operations $\bullet_t, \oplus, \rho_{i \to j}, \eta_{i,j}$ for integers $t, i, j \in [k]$ is called a *k-expression* or *expression* for short. To distinguish between an expression and the graph defined by the expression, we denote by $\mathrm{val}(X)$ the

graph defined by expression X. That is, CW_k is the set of all graphs $\mathrm{val}(X)$, where X is a k-expression.

The *clique-width* of an *unlabeled* graph $G = (V_G, E_G)$ is the smallest integer k such that there is some labeling $\mathrm{lab}_G : V_G \to [k]$ such that $G' = (V_G, E_G, \mathrm{lab}_G)$ has clique-width k. Since a relabeling of vertices does not change the clique-width of the graph, we consider an unlabeled graph as a labeled graph with all vertices labeled 1. This allows us to use the notation "graph" without any confusion for labeled and unlabeled graphs.

We next define a so-called *normal form* for a k-expression. This normal form does not restrict the graphs that can be defined by k-expressions but helps us to prove our main result.

To keep the definition of our normal form as simple as possible, we enumerate the vertices in $\mathrm{val}(X)$ for some k-expression X as follows. If $G = \mathrm{val}(\bullet_t)$, then the single vertex in G is the first vertex of G. Let $G = \mathrm{val}(Y_1 \oplus Y_2)$. If $\mathrm{val}(Y_1)$ has n vertices and $\mathrm{val}(Y_2)$ has m vertices, then the i-th vertex of G is the i-th vertex of $\mathrm{val}(Y_1)$ if $i \leq n$ and the $(i-n)$-th vertex of $\mathrm{val}(Y_2)$ if $i > n$. The i-th vertex of $\mathrm{val}(\eta_{i,j}(Y))$ and $\mathrm{val}(\rho_{i\to j}(Y))$ is the i-th vertex of $\mathrm{val}(Y)$. We say two expressions X and Y are *equivalent*, denoted by $X \equiv Y$, if $\mathrm{val}(X)$ and $\mathrm{val}(Y)$ are isomorphic with regard to the order of the vertices, that is,

1. $\mathrm{val}(X)$ and $\mathrm{val}(Y)$ have the same number n of vertices,
2. the i-th vertex in $\mathrm{val}(X)$, $1 \leq i \leq n$, has the same label as the i-th vertex in $\mathrm{val}(Y)$, and
3. there is an edge between the i-th and j-th vertex in $\mathrm{val}(X)$, $1 \leq i, j \leq n$, if and only if there is an edge between the i-th and j-th vertex in $\mathrm{val}(Y)$.

Otherwise X and Y are *not equivalent*, denoted by $X \not\equiv Y$.

Definition 2 (Normal form). *The* normal form *for k-expressions is defined as follows.*

1. *The k-expression $\bullet_t$ for some $t \in [k]$ is in normal form.*
2. *If Y_1 and Y_2 are two k-expressions in normal form then the k-expression*

$$\rho_{i_n\to j_n}(\cdots \rho_{i_1\to j_1}(\eta_{i'_{n'},j'_{n'}}(\cdots \eta_{i'_1,j'_1}(Y_1 \oplus Y_2)\cdots))\cdots)$$

for $i_1, j_1, \ldots, i_n, j_n, i'_1, j'_1, \ldots, i'_{n'}, j'_{n'} \in [k]$ is in normal form if the following properties hold.

   a) For every edge insertion operation $\eta_{i'_{l'},j'_{l'}}$, $1 \leq l' \leq n'$,

$$\eta_{i'_{l'},j'_{l'}}(Y_1 \oplus Y_2) \not\equiv Y_1 \oplus Y_2, \quad \eta_{i'_{l'},j'_{l'}}(Y_1) \equiv Y_1, \quad \eta_{i'_{l'},j'_{l'}}(Y_2) \equiv Y_2,$$

   and for every edge insertion operation $\eta_{i'_{k'},j'_{k'}}$, $1 \leq k' < l'$, $\{i'_{l'}, j'_{l'}\} \neq \{i'_{k'}, j'_{k'}\}$.

   b) For every relabeling operation $\rho_{i_l\to j_l}$, $1 \leq l \leq n$, graph

$$val(\rho_{i_{l-1}\to j_{l-1}}(\cdots \rho_{i_1\to j_1}(\eta_{i'_{n'},j'_{n'}}(\cdots \eta_{i'_1,j'_1}(Y_1 \oplus Y_2)\cdots))\cdots))$$

   has a vertex labeled by i_l, a vertex labeled by j_l, and $i_l \notin \{j_1, \ldots, j_{l-1}\}$, and at least one of the two indices i_l and j_l is in $\{i'_1, j'_1, \ldots, i'_{n'}, j'_{n'}\}$.

c) If there are two relabeling operations $\rho_{i_l \to j_l}$ *and* $\rho_{i_k \to j_k}$, $1 \le l < k \le n$, *such that* $j_l = j_k$ *then* $i_l \in \{i'_1, j'_1, \ldots, i'_{n'}, j'_{n'}\}$ *or* $i_k \in \{i'_1, j'_1, \ldots, i'_{n'}, j'_{n'}\}$.

d) If $\rho_{i_l \to j_l}$, $1 \le l \le n$, *is a relabeling operation then*

i. if val(Y_1) *has a vertex labeled by* i_l *then*

$$\begin{aligned} &\rho_{i_l \to j_l}(\cdots \rho_{i_1 \to j_1}(\eta_{i'_{n'}, j'_{n'}}(\cdots \eta_{i'_1, j'_1}(\quad Y_1 \oplus Y_2\ )\cdots))\cdots) \\ \not\equiv\ &\rho_{i_l \to j_l}(\cdots \rho_{i_1 \to j_1}(\eta_{i'_{n'}, j'_{n'}}(\cdots \eta_{i'_1, j'_1}(\ \rho_{i_l \to j_l}(Y_1) \oplus Y_2\ )\cdots))\cdots) \end{aligned}$$

and

ii. if val(Y_2) *has a vertex labeled by* i_l *then*

$$\begin{aligned} &\rho_{i_l \to j_l}(\cdots \rho_{i_1 \to j_1}(\eta_{i'_{n'}, j'_{n'}}(\cdots \eta_{i'_1, j'_1}(\ Y_1 \oplus \quad Y_2\ )\cdots))\cdots) \\ \not\equiv\ &\rho_{i_l \to j_l}(\cdots \rho_{i_1 \to j_1}(\eta_{i'_{n'}, j'_{n'}}(\cdots \eta_{i'_1, j'_1}(\ Y_1 \oplus \rho_{i_l \to j_l}(Y_2)\ )\cdots))\cdots). \end{aligned}$$

In an expression in normal form a relabeling operation is never done immediately before an edge insertion operation. Furthermore, edge insertion operations and relabeling operations are done as soon as possible, see 2.(a) and 2.(d). The rest of the restrictions are only to avoid redundant composition steps.

The following lemma shows that every graph of clique-width k can be defined by some k-expression in normal form.

Lemma 1. *For every graph* G *of clique-width* k *there is a* k*-expression for* G *in normal form.*

Here we should give the following remark. Assume an expression

$$X = \rho_{i_n \to j_n}(\cdots \rho_{i_1 \to j_1}(Y)\cdots)$$

is in normal form, where $Y = \eta_{i'_{n'}, j'_{n'}}(\cdots \eta_{i'_1, j'_1}(Z)\cdots)$ for some expression Z. If we choose a relabeling operation $\rho_{i_l \to j_l}$ for some l, $1 \le l \le n$, and substitute all labels i_l in Y by j_l and all labels j_l in Y by i_l to get an expression denoted by $Y_{i_l \leftrightarrow j_l}$, then the new expression

$$X' = \rho_{i_n \to j_n}(\cdots \rho_{i_1 \to j_1}(Y_{i_l \leftrightarrow j_l})\cdots)$$

is also in normal form and defines the same graph as before, i.e., $X \equiv X'$.

Definition 3 (Expression tree). *The expression tree* $T = (V_T, E_T, lab_T)$ *of a* k*-expression* X *is an ordered rooted tree whose nodes are labeled by the operations of the expression and whose arcs are directed from the sons to the fathers towards the root of* T.

The expression tree T *of expression* $\bullet_t$ *consists of a single node* r *(the root of* T*) labeled by* $\bullet_t$. *The* expression tree T *of* $\eta_{i,j}(X)$ *and* $\rho_{i \to j}(X)$ *consists of the expression tree* T' *of expression* X *with an additional node* r *(the root of* T*) labeled by* $\eta_{i,j}$ *or* $\rho_{i \to j}$, *respectively, and an additional arc from the root of* T' *to node* r. *The* expression tree T *of* $X_1 \oplus X_2$ *consists of the disjoint union of the expression trees* T_1 *and* T_2 *of* X_1 *and* X_2, *respectively, with an additional node* r *(the root of* T*) labeled by* $\oplus$ *and two additional arcs from the roots of* T_1 *and*

T_2 to node r. The root of T_1 is the left son of r and the root of T_2 is the right son of r.

A node of T labeled by $\bullet_t$, $\eta_{i,j}$, $\rho_{i\to j}$, or $\oplus$ is called a leaf, edge insertion node, relabeling node, *or* union node, *respectively.*

For every expression X, there is a one-to-one correspondence between the leaves of the expression tree $T = (V_T, E_T, \mathrm{lab}_T)$ of X and the vertices of the graph $G = (V_G, E_G, \mathrm{lab}_G) = \mathrm{val}(X)$. The i-th vertex in $\mathrm{val}(X)$ corresponds to the i-th leaf of T counted from left to right.

For some node u of T, let $T(u)$ be the subtree of T induced by node u and all nodes v of T from which there is a directed path to u in T. Tree $T(u)$ is always an expression tree. The expression X' of $T(u)$ defines a (possibly relabeled) subgraph G' of G. The vertices of G' are the vertices of G corresponding to the leaves of $T(u)$. The edges of G' and the labels of the vertices of G' are defined by expression X' which is a sub-expression of X. The subgraph G' of G is also denoted by $G(T, u)$ or $G(u)$, if tree T is known from the context.

Definition 4 (l-terminal graph). *An* l-terminal graph *is a triple $G = (V_G, E_G, P_G)$ where (V_G, E_G) is a graph and $P_G = (x_1, \ldots, x_l)$ is a sequence of $l \geq 0$ mutually distinct vertices of V_G. The vertices in P_G are called* terminal vertices *or* terminals. *Vertex x_i, $1 \leq i \leq l$, is called the i-th* terminal *of G. The vertices in $V_G - P_G$ are called the* inner vertices *of G.*

The operation $\circ$ maps two l-terminal vertex disjoint graphs H and J to some graph $H \circ J$, by taking the disjoint union of H and J, then identifying corresponding terminals, i.e., for $i = 1, \ldots, l$, identifying the i-th terminal of H with the i-th terminal of J, and removing multiple edges.

Terminal graphs are also called *sourced graphs*, see [ALS91]. The composition mechanism can easily be extended to labeled graphs under the assumption that the i-th terminal of H and the i-th terminal of J have the same label.

Definition 5 (Replaceability). *Let Π be a graph property, i.e., $\Pi : \mathcal{G} \to \{0, 1\}$, where $\mathcal{G}$ is the set of all graphs. Two l-terminal graphs G_1 and G_2 are called* replaceable *with respect to Π, denoted by $G_1 \sim_{\Pi,l} G_2$, if for every l-terminal graph H, $\Pi(G_1 \circ H) = \Pi(G_2 \circ H)$.*

If $\sim_{\Pi,l}$ divides the set of all l-terminal graphs into a finite number of equivalence classes then Π is decidable in linear time for all graphs of tree-width l by dynamic programming algorithms. The input for the decision algorithms is a binary rooted decomposition tree of width l. Such a decomposition tree of bounded width can be computed in linear time for every graph of bounded tree-width, see [Bod96]. The dynamic programming algorithms simply compute bottom-up for every complete subtree with root u the equivalence class of the corresponding l-terminal graph from the equivalence classes of the subtrees of the two sons of u.

3 The Main Result

In this section, we show that every $\sim_{\Pi,l}$, $l \geq 0$, has a finite number of equivalence classes where Π is the graph property "clique-width at most k". In the

following, we consider the case where we have two l-terminal labeled graphs $H = (V_H, E_H, P_H, \mathrm{lab}_H)$ and $J = (V_J, E_J, P_J, \mathrm{lab}_J)$ such that

$$G = (V_G, E_G, \mathrm{lab}_G) = H \circ J$$

has clique-width at most k. We partition the vertex set V_G of G into three disjoint sets U_H, U_J, U_P such that $U_H \cup U_J \cup U_P = V_G$. Vertex set $U_H = V_H - P_H$ contains the inner vertices from H, vertex set $U_J = V_J - P_J$ contains the inner vertices from J, and vertex set U_P contains the terminals from H and J. Vertex set U_P has exactly l vertices because the l terminals of H are identified with the l terminals of J. Graph G does not have any edge between a vertex of U_H and a vertex of U_J.

Let $T = (V_T, E_T, \mathrm{lab}_T)$ be a k-expression tree for $G = H \circ J$. The subtree T_P of T is defined by the l leaves of T that correspond to the l vertices of U_P and by all nodes of T to which there is a path from these leaves. The root of T_P is the root of T. In general, tree T_P is not an expression tree because it does not represent a valid expression.

The outline of the proof is the following. We will show that for each such pair H, J, where $G = H \circ J$ has clique-width at most k there is always at least one k-expression tree T for G such that T_P has a very special form. This special form allows us to bound the information how H and J are combined. The size of this information will depend only on k and l but not on the size of H or J. Let us call this information the *connection type* of T_P.

The definition of the connection type will imply the following. If there are two systems (H_1, J_1, T_1) and (H_2, J_2, T_2) such that the connection type of T_1 and the connection type of T_2 are equivalent, then the two graphs $H_1 \circ J_2$ and $H_2 \circ J_1$ will also have clique-width at most k. If for every system (H_1, J_1, T_1) for H_1 and some J_1 there is some system (H_2, J_2, T_2) for H_2 and some J_2 such that the connection type of T_1 and the connection type of T_2 are equivalent and vice versa, then $H_1 \sim_{\Pi_k, l} H_2$, where Π_k is the property of clique-width at most k. Since there is only a bounded number of mutually different connection types, it follows that there is only a bounded number of equivalence classes for all l-terminal graphs with respect to equivalence relation $\sim_{\Pi_k, l}$.

Lemma 2. *There is always a k-expression tree T for G that satisfies the following property.*

Property 1. Let u_1 be a union node of T_P such that one of its sons u_0 is in T_P and the other son u_0' is not in T_P. Then the vertices of $G(u_0')$ are either all from U_H or all from U_J.

Proof. Since $G(u_0')$ does not contain vertices from U_P, we know that the vertices of $G(u_0')$ are all from $U_H \cup U_J$. If the vertices of $G(u_0')$ are not all from U_H or not all from U_J then let T_1 and T_2 be the k-expression trees that define the subgraphs of $G(u_0')$ induced by the vertices of U_H and U_J, respectively. We replace the subtree $T(u_0')$ by T_1, insert a new union node v_0 between u_1 and u_0, and make the root of T_2 to the second son of v_0 which is not in T_P. The resulting tree obviously defines the same graph as before but u_1 now satisfies property 1.

The transformation of an arbitrary expression tree T into normal form as in the proof of lemma 1 (omitted in this version) does not change property 1 of T. That is, we can additionally assume that we have a k-expression tree T for $G = H \circ J$ in normal form which satisfies property 1.

Let u_1 be a union node of T_P such that one of its sons u_0 is in T_P and the other son u_0' is not in T_P. We define $\xi(u_1) := H$ or $\xi(u_1) := J$ if the vertices of $G(u_0')$ are all from U_H or all from U_J, respectively. In all other cases and in the case where u_1 is not a union node, we say $\xi(u_1)$ is undefined. By lemma 2, we can assume that $\xi(u_1)$ is defined for all union nodes u_1 of T_P for which exactly one of their sons is in T_P.

The tree T_P consists of exactly $2l-1$ maximal paths $p = (u_1, \ldots, u_{s'})$, $s' \geq 1$, such that u_1 is a leaf of T_P or a union node with two sons in T_P and all vertices $u_2, \ldots, u_{s'}$ of the path have exactly one son in T_P. The last node $u_{s'}$ is either the root of T_P or a son of some union node whose sons are both in T_P. All the graphs $G(u_s)$ for $s = 1, \ldots, s'$ contain the same vertices of U_P. Such a path is called a 1-*path* or *path of type* 1. Every node of T_P is in exactly one of these $2l-1$ paths of type 1.

We divide every 1-path p of T_P into a path of type 1.a and a path of type 1.b. The paths of type 1.a are the maximal paths q whose first node u_1 is a leaf of T_P or a union node whose sons u_0' and u_0 are both in T_P and all other nodes of q are edge insertion or relabeling nodes. The remaining parts of T_P are called paths of type 1.b. There are exactly $2l-1$ paths of type 1.a and at most $2l-1$ paths of type 1.b. For every union node u_1 in a 1.b-path there is either $\xi(u_1) = H$ or $\xi(u_1) = J$.

A maximal subpath $(u_1, \ldots, u_{r'}, \ldots, u_{s'})$ of a 1.b-path p such that u_1 is a union node, $u_2, \ldots, u_{r'}$ are edge insertion nodes, and $u_{r'+1}, \ldots, u_{s'}$ are relabeling nodes, is called a *frame* of p. Every frame has at most $\binom{k}{2} + k$ nodes, because there are one union node u_1, at most $\binom{k}{2}$ edge insertion nodes $u_2, \ldots, u_{r'}$, and at most $k-1$ relabeling nodes $u_{r'+1}, \ldots, u_{s'}$.

For some node u_s of the k-expression tree T, let $L_H(u_s)$, $L_J(u_s)$, and $L_P(u_s)$ be the sets of the labels of the vertices of $G(u_s)$ which are from U_H, U_J, and U_P, respectively. The intersections $L_H(u_s) \cap L_J(u_s)$, $L_P(u_s) \cap L_H(u_s)$, $L_P(u_s) \cap L_J(u_s)$, and $L_P(u_s) \cap L_H(u_s) \cap L_J(u_s)$ are abbreviated by

$$L_{H \cap J}(u_s), \quad L_{P \cap H}(u_s), \quad L_{P \cap J}(u_s), \quad \text{and} \quad L_{P \cap H \cap J}(u_s),$$

respectively.

Lemma 3. *There is always a k-expression tree T for G in normal form which satisfies property 1 and 2.*

Property 2. Let u_s be a relabeling node of T_P labeled by $\rho_{i \to j}$ and let u_{s-1} be the son of u_s. If $i \in L_P(u_{s-1})$ then $j \in L_P(u_{s-1})$.

Proof. Assume T_P is in normal form and satisfies property 1. Let $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ be a frame of T_P such that $u_2, \ldots, u_{r'}$ are edge insertion nodes, and $u_{r'+1}, \ldots, u_{s'}$ are relabeling nodes. Let u_s, $r' < s \leq s'$, be labeled by $\rho_{i \to j}$

and let $i \in L_P(u_{s-1})$. We additionally assume that $T(u_{s-1})$ already satisfies the lemma.

If $j \notin L_P(u_{s-1})$ then we replace in subtree $T(u_{r'})$ simultaneously every label i by label j and every label j by label i. The resulting subtree $T(u_{r'})$ is still in normal form and satisfies property 1 and 2. The resulting tree $T(u_s)$ is now also in normal form, satisfies property 1 and 2, and defines the same graph as before.

By lemma 3, we can assume that for every son u_{s-1} of some relabeling node u_s of T_P $L_P(u_s) \subseteq L_P(u_{s-1})$. If $L_P(u_s) = L_P(u_{s-1})$, then the reverse inclusion holds true for the sets $L_{P\cap H}(u_s)$ and $L_{P\cap J}(u_s)$, i.e., $L_{P\cap H}(u_s) \supseteq L_{P\cap H}(u_{s-1})$ and $L_{P\cap J}(u_s) \supseteq L_{P\cap J}(u_{s-1})$. This allows us to divide every 1.b-path p into paths of type 2.a and paths of type 2.b as follows. The 2.a-paths are the frames $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ of p for which at least one of the following properties holds.

1. There is some relabeling node u_s, $r' < s \leq s'$, such that $|L_P(u_s)| < |L_P(u_{s-1})|$, $|L_{P\cap H}(u_s)| > |L_{P\cap H}(u_{s-1})|$, or $|L_{P\cap J}(u_s)| > |L_{P\cap J}(u_{s-1})|$.
2. $|L_{P\cap H}(u_1)| > |L_{P\cap H}(u_0)|$ or $|L_{P\cap J}(u_1)| > |L_{P\cap J}(u_0)|$, where u_0 is the son of u_1 in T_P.

The 2.b-paths are the remaining parts of p. In a 2.b-path p all the sets $L_P(u_s)$ are equal, all the sets $L_{P\cap H}(u_s)$ are equal, all the sets $L_{P\cap J}(u_s)$ are equal, and all the sets $L_{P\cap H\cap J}(u_s)$ are equal, for all nodes u_s of p including the son u_0 of the first node u_1 which is in T_P.

For a frame $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ of a 2.b-path let

$$L_P(q) = L_P(u_1), \quad L_{P\cap H}(q) = L_{P\cap H}(u_1), \quad L_{P\cap J}(q) = L_{P\cap J}(u_1).$$

We use q instead of some node of q as the argument to emphasize that the sets above are equal for all nodes of q including the son u_0 of the first node of q which is in T_P. It is easy to see that for every 1.b-path p there are at most $3k$ paths of type 2.b and at most $3k-1$ paths of type 2.a.

Lemma 4. *There is always a k-expression tree T for G in normal form that satisfies property 1, 2, and additionally property 3.*

Property 3. Let p be a 2.b-path of T_P, and $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ be a frame of p such that $u_2, \ldots, u_{r'}$ are edge insertion nodes, $u_{r'+1}, \ldots, u_{s'}$ are relabeling nodes, and u_s, $r' < s \leq s'$, is labeled by $\rho_{i\to j}$. If $i \in L_{H\cap J}(u_{s-1})$ then $j \in L_{H\cap J}(u_1)$.

Proof. Assume T is in normal form and satisfies property 1 and 2. Let $i \in L_{H\cap J}(u_{s-1})$.

If $j \in L_P(q)$ then the assumption $i \in L_{H\cap J}(u_{s-1})$ and the relabeling $\rho_{i\to j}$ at node u_s imply $j \in L_{P\cap H\cap J}(u_s) = L_{P\cap H\cap J}(q)$ and thus $j \in L_{H\cap J}(u_1)$.

If $j \notin L_P(q)$ and $j \notin L_{H\cap J}(u_1)$ then we replace in subtree $T(u_{r'})$ simultaneously every label i by label j and every label j by label i. The resulting subtree $T(u_{r'})$ is still in normal form and satisfies property 1, 2, and 3. The resulting tree $T(u_{s'})$ is now also in normal form, satisfies property 1, 2, and 3, and defines the same graph as before.

For some node u_s of T_P and some label $j \in [k]$ let $\text{forb}_P(u_s, j)$ be the set of all labels $i \in L_P(u_s)$ such that graph $G(u_s)$ has two non adjacent vertices labeled by i and j. If the set $\text{forb}_P(u_s, j)$ is empty then either graph $G(u_s)$ has no vertex labeled by j or every vertex of $G(u_s)$ labeled by j is adjacent to every vertex of $G(u_s)$ labeled by some label of $L_P(u_s)$. Note that for every node u_s of T_P, set $L_P(u_s)$ is non-empty.

Let u_s be a relabeling node of T_P labeled by $\rho_{i\to j}$ and let u_{s-1} be the son of u_s. If $i \notin L_P(u_{s-1})$ then obviously

$$\text{forb}_P(u_s, j) = \text{forb}_P(u_{s-1}, j) \cup \text{forb}_P(u_{s-1}, i).$$

Lemma 5. *Assume T is in normal form and satisfies property 1, 2, and 3. Let p be a 2.b-path of T_P, let $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ be a frame of p, and let u_0 be the son of u_1 which is in T_P. If a node u_s, $r' < s \leq s'$, is labeled by $\rho_{i\to j}$ and if $i \in L_{H\cap J}(u_{s-1})$ then $\text{forb}_P(u_0, j) \subsetneq \text{forb}_P(u_{s'}, j)$.*

Lemma 5 allows us to divide every 2.b-path p into paths of type 3.a and paths of type 3.b as follows. The 3.a-paths are the frames $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ of p for which there is some relabeling node u_s, $r' < s \leq s'$, or some label $i \in L_P(q) \cup L_{H\cap J}(u_0)$, such that $|L_{H\cap J}(u_s)| \neq |L_{H\cap J}(u_{s-1})|$, $|L_{H\cap J}(u_1)| > |L_{H\cap J}(u_0)|$, or $|\text{forb}_P(u_{s'}, i)| \neq |\text{forb}_P(u_0, i)|$, where u_0 is the son of u_1 in T_P. The 3.b-paths are the remaining parts of p. In a 3.b-path p all the sets $L_{H\cap J}(u_s)$ are equal for all nodes u_s of p including the son of the first node which is in T_P. For a frame $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ of a 3.b-path let $L_{H\cap J}(q) = L_{H\cap J}(u_{s'})$ and $\text{forb}_P(q, j) = \text{forb}_P(u_{s'}, j)$. Then for all frames q of a 3.b-path including the frame immediately before the first frame of q all the set $L_{H\cap J}(q)$ and $\text{forb}_P(q, j)$ are equal for all $j \in [k]$.

Lemma 5 implies that the number of 3.a-paths and 3.b-paths in some 2.b-path can be estimated by some constant c depending only on k.

Lemma 6. *Let $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ be a frame of a 3.b-path p such that $u_2, \ldots, u_{r'}$ are edge insertion nodes, $u_{r'+1}, \ldots, u_{s'}$ are relabeling nodes, and u_s, $r' < s \leq s'$, is labeled by $\rho_{i\to j}$.*

1. *If $\xi(u_1) = H$ then $i \in L_H(u_{s-1}) - L_P(q) - L_J(u_{s-1})$ and $j \in L_H(u_{s-1})$.*
2. *If $\xi(u_1) = J$ then $i \in L_J(u_{s-1}) - L_P(q) - L_H(u_{s-1})$ and $j \in L_J(u_{s-1})$.*

Lemma 7. *There is always a k-expression tree T for G in normal form that satisfies property 1, 2, 3, and additionally property 4.*

Property 4. Every 3.b-path p is divided into at most $3 \cdot 2^{2(k+1)}$ paths p' such that for all frames $q = (u_1, \ldots, u_{r'}, \ldots, u_{s'})$ of p' all $\xi(u_1)$ are equal.

Lemma 7 allows us to divide every 3.b-path into $2^{2(k+1)}$ paths of type 4.a and $2 \cdot 2^{2(k+1)}$ paths of type 4.b. The paths of type 4.a are the frames p_1. The paths of type 4.b are the paths $p_{2,1}$ and $p_{2,2}$, where for all the frames q of $p_{2,1}$ all $\xi(q)$ are equal and for all the frames q of $p_{2,2}$ all $\xi(q)$ are equal.

Let us now summarize how the paths of T_P are partitioned. We have

1. exactly $2l-1$ paths of type 1.a,
2. at most $(2l-1)\cdot(3k-1)$ paths of type 2.a,
3. at most $(2l-1)\cdot 3k\cdot c$ paths of type 3.a,
4. at most $(2l-1)\cdot 3k\cdot c\cdot 2^{2(k+1)}$ paths of type 4.a, and
5. at most $(2l-1)\cdot 3k\cdot c\cdot 2\cdot 2^{2(k+1)}$ paths of type 4.b.

Every node of T_P is in exactly one of these paths. The paths of type 1.a, 2.a, 3.a, and 4.a have at most $\binom{k}{2}+k$ nodes. For all frames $q=(u_1,\ldots,u_{s'})$ in a path of type 4.b all $\xi(u_1)$ are equal, all sets $\mathrm{forb}_P(u_{s'},j)$ are equal for all $j\in[k]$, and all sets $L_P(q)$ and $L_{H\cap J}(q)$ are equal.

Definition 6 (Connection type). *Assume T_P satisfies property 1, 2, 3, and 4. Then the* connection type *C_P of T_P is a labeled tree obtained from T_P by replacing every 4.b-path $p=(u_1,\ldots,u_{s'})$ by some* special node *v and two edges (u_1,v) and $(v,u_{s'})$. Every usual node u_s of T_P will be labeled by its clique-width operation and additionally by $\xi(u_s)$, $L_P(u_s)$, $L_J(u_s)$, $L_H(u_s)$, and all $forb_P(u_s,j)$ for all $j\in[k]$. Each leaf u_s of T_P represents a vertex v of G obtained by joining two terminal vertices, one from the l-terminal graph H and one from the l-terminal graph J. If v is obtained by joining the i-th terminal of H with the i-th terminal of J then leaf u_s is additionally labeled by index i.*

Two connection types $C_{P,1}$, $C_{P,2}$ are called equivalent *if there is a bijection b between the nodes of $C_{P,1}$ and $C_{P,2}$ such that*

1. *node u_{s-1} is a son (left son, right son) of node u_s in $C_{P,1}$ if and only if node $b(u_{s-1})$ is a son (left son, right son, respectively) of node $b(u_s)$ in $C_{P,2}$,*
2. *if node u_s of $C_{P,1}$ is a special node then node $b(u_s)$ of $C_{P,2}$ is a special node,*
3. *node u_s of $C_{P,1}$ and node $b(u_s)$ of $C_{P,2}$ are equally labeled.*

Theorem 1. *Let (H_1,J_1,T_1) and (H_2,J_2,T_2) be two systems such that*

1. *H_1,H_2,J_1,J_2 are l-terminal graphs,*
2. *$H_1\circ J_1$ and $H_2\circ J_2$ have clique-width at most k,*
3. *T_1 and T_2 are k-expression trees for $H_1\circ J_1$ and $H_2\circ J_2$, respectively, in normal form which satisfy property 1, 2, 3, and 4,*
4. *and the two connection types $C_{P,1}$ and $C_{P,2}$ of $T_{P,1}$ and $T_{P,2}$ are equivalent.*

Then $H_1\circ J_2$ and $H_2\circ J_1$ have also clique-width at most k.

By theorem 1 and the fact that there is only a finite number of connection types for fixed integers l and k it follows that the equivalence relation $\sim_{\Pi_k,l}$ has a finite number of equivalence classes. For an l-terminal graph H let $\mathcal{C}(H)$ be the set of all connection types C_P defined as follows. A connection type C_P is in $\mathcal{C}(H)$ if there is some l-terminal graph J such that $H\circ J$ has clique-width k and there is a k-expression tree T for $H\circ J$ such that T_P is in normal form and satisfies property 1, 2, 3, and 4, and C_P is the connection type of T_P. If for two l-terminal graphs H_1 and H_2 the sets $\mathcal{C}(H_1)$ and $\mathcal{C}(H_2)$ are equal, then H_1 and H_2 are replaceable with respect to the clique-width k property.

References

ALS91. S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

Bod96. H.L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

Bod98. H.L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998.

CHL$^+$00. D.G. Corneil, M. Habib, J.M. Lanlignel, B. Reed, and U. Rotics. Polynomial time recognition of clique-width at most three graphs. In *Proceedings of Latin American Symposium on Theoretical Informatics (LATIN '2000)*, volume 1776 of *LNCS*. Springer-Verlag, 2000.

CMR00. B. Courcelle, J.A. Makowsky, and U. Rotics. Linear time solvable optimization problems on graphs of bounded clique width. *Theory of Computing Systems*, 33(2):125–150, 2000.

CO00. B. Courcelle and S. Olariu. Upper bounds to the clique width of graphs. *Discrete Applied Mathematics*, 101:77–114, 2000.

Cou90. B. Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.

CPS85. D.G. Corneil, Y. Perl, and L.K. Stewart. A linear recognition algorithm for cographs. *SIAM Journal on Computing*, 14(4):926–934, 1985.

GR00. M.C. Golumbic and U. Rotics. On the clique-width of some perfect graph classes. *International Journal of Foundations of Computer Science*, 11(3):423–443, 2000.

GW00. F. Gurski and E. Wanke. The tree-width of clique-width bounded graphs without $K_{n,n}$. In *Proceedings of Graph-Theoretical Concepts in Computer Science*, volume 1938 of *LNCS*, pages 196–205. Springer-Verlag, 2000.

Joh98. Ö. Johansson. Clique-decomposition, NLC-decomposition, and modular decomposition - relationships and results for random graphs. *Congressus Numerantium*, 132:39–60, 1998.

KR01. D. Kobler and U. Rotics. Polynomial algorithms for partitioning problems on graphs with fixed clique-width. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 468–476. ACM-SIAM, 2001.

Lap98. D. Lapoire. Recognizability equals definability, for every set of graphs of bounded tree-width. In *Proceedings of 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *LNCS*, pages 618-628. Springer-Verlag, 1998.

Wan94. E. Wanke. k-NLC graphs and polynomial algorithms. *Discrete Applied Mathematics*, 54:251–266, 1994.

Complexity Bounds for Vertical Decompositions of Linear Arrangements in Four Dimensions

Vladlen Koltun*

School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
vladlen@tau.ac.il

Abstract. We prove tight and near-tight combinatorial complexity bounds for vertical decompositions of arrangements of linear surfaces in four dimensions. In particular, we prove a tight upper bound of $\Theta(n^4)$ for the vertical decomposition of an arrangement of n hyperplanes in four dimensions, improving the best previously known bound [7] by a logarithmic factor. We also show that the complexity of the vertical decomposition of an arrangement of n 3-simplices in four dimensions is $O(n^4\alpha(n)\log n)$, improving the best previously known bound [3] by a near-linear factor. We believe that the techniques used for obtaining these results can also be extended to analyze decompositions of arrangements of fixed-degree algebraic surfaces (or surface patches) in four dimensions.

1 Introduction

Given a collection Γ of n fixed-degree algebraic surfaces in $\mathbb{R}^d$, its arrangement [1, 8] is denoted by $\mathcal{A}(\Gamma)$. The complexity of a single cell in $\mathcal{A}(\Gamma)$ may be as large as $\Omega(n^{d-1})$. There exist many geometric algorithms that require arrangements to be decomposed into cells of constant description complexity (that is, defined in terms of a constant number of polynomial equalities and inequalities of constant maximum degree). As a result, devising a decomposition scheme that decomposes arrangements into as few as possible such cells is an intensively studied problem.

The most efficient general-purpose decomposition scheme is the *vertical decomposition*, which is suitable for arrangements of fixed-degree algebraic surfaces is any dimension. We define vertical decompositions in Section 2, and refer the reader to [3] for details on the general scheme. The vertical decomposition was originally introduced in the context of two-dimensional problems, and was extended to higher dimensions in the late 1980's [3]. The complexity of the vertical decomposition of an arrangement of n triangles (or planes) in $\mathbb{R}^3$ is known to be $O(n^2\alpha(n)\log n + K)$, where $K = O(n^3)$ is the complexity of the undecomposed arrangement [10]. However, there is still a substantial gap between the known upper and lower bounds for the complexity of vertical decompositions in dimensions higher than three.

* This work was supported by a grant from the Israeli Academy of Sciences (center of excellence).

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 99–110, 2001.

The number of cells in the vertical decomposition of an arrangement of n fixed-degree algebraic surfaces in $\mathbb{R}^4$ is only known to be $O(n^5\beta(n))$ [3], where $\beta(n)$ is an extremely slow-growing function of n (which also depends on d and on the maximum degree of the polynomials that define the surfaces of Γ), related to Davenport-Schinzel sequences [9]. The problem of improving this upper bound (which is larger than the known lower bound by a near-linear factor) has been stated as an important open problem numerous times [1,3,7,8,9], but is still open for more than a decade.

Such an improvement has immediate wide-ranging algorithmic applications, as it would automatically improve the (asymptotic) running time of many algorithms that utilize decomposed 4-dimensional arrangements. These include algorithms for point location, range searching, robot motion planning, and a variety of geometric optimization problems (see [9, Section 8.3] and the references therein). An interesting result in this direction is that of Guibas et al. [7], who showed that the vertical decomposition of an arrangement of n hyperplanes in four dimensions has complexity $O(n^4 \log n)$.

In this paper, we prove a tight bound of $\Theta(n^4)$ for the case of hyperplanes, just mentioned. Moreover, we bound the complexity of the vertical decomposition of an arrangement of n 3-simplices in four dimensions by $O(n^4\alpha(n)\log n)$. This improves the best previously known upper bound for this setting [3] by a near-linear factor.

We note that an arrangement of simplices as above can be decomposed into $\Theta(n^4)$ cells by extending the simplices into hyperplanes, and decomposing the resulting hyperplane arrangement using the bottom-vertex simplicial decomposition [4]. In light of this special decomposition scheme for linear arrangements, the principal significance of this paper is in the fact that it introduces general techniques for analyzing *vertical decompositions* in four dimensions.

2 Vertical Decompositions in Four Dimensions

The construction. Denote the coordinates by x, y, z and w. Given a collection Γ of 3-simplices in $\mathbb{R}^4$, the vertical decomposition of $\mathcal{A}(\Gamma)$, denoted by $\mathcal{V}(\Gamma)$, is constructed as follows.

For each $S \in \Gamma$, erect a 3D z-vertical visibility wall on the boundary of S (a 2D piecewise linear surface denoted by ∂S), which is defined as the union of all z-vertical (visibility) segments that have an end-point on ∂S and are interior-disjoint from all simplices of Γ. Also, for each pair $S, T \in \Gamma$, erect a 3D z-vertical visibility wall on the 2D (linear) surface $S \cap T$ in a similar fashion. This results in a decomposition of $\mathcal{A}(\Gamma)$ into (not necessarily convex) z-vertical prisms, such that the floor (respectively, ceiling) of each prism, if it exists, is contained in a single simplex of Γ. We denote this decomposition by $\mathcal{V}_1(\Gamma)$.

For a prism P of $\mathcal{V}_1(\Gamma)$, the simplex containing its floor (respectively, ceiling) is denoted by P_F (respectively, P_C). Projecting P onto its floor or ceiling (in the z-direction) results in a 3D polyhedron, denoted by P_{3D}. We first decompose P_{3D} by erecting 2D y-vertical visibility walls on each of its edges. For an edge f

of P_{3D}, the wall erected on it is defined as the union of all y-vertical (visibility) segments that have an end-point on f and are fully contained in P_{3D}. We then decompose P by erecting z-vertical 3D visibility walls on each such y-vertical 2D visibility wall of P_{3D}. Repeating this process for all prisms P results in a decomposition of $\mathcal{V}_1(\Gamma)$, which we denote by $\mathcal{V}_2(\Gamma)$.

We further refine $\mathcal{V}_2(\Gamma)$ as follows. For a prism Q of $\mathcal{V}_2(\Gamma)$, consider its z-projection Q_{3D} (which is a y-vertical prism in $\mathbb{R}^3$). Projecting Q_{3D} onto its floor or ceiling (in the y-direction) results in a 2D polygon Q_{2D}, which we decompose by erecting 0, 1, or 2 x-vertical (possibly infinite) visibility segments on each vertex v of Q_{2D}, defined in analogy to the above. We then erect y-vertical 2D walls (inside Q_{3D}) on each such x-vertical segment of Q_{2D}. Subsequently, z-vertical 3D walls (inside Q) are erected on each such y-vertical 2D wall of Q_{3D}. Repeating this process for each prism Q of $\mathcal{V}_2(\Gamma)$ decomposes $\mathcal{V}_2(\Gamma)$ into cells of constant description complexity: each such cell is a convex polyhedron bounded by up to 8 facets.

This completes the construction of the vertical decomposition $\mathcal{V}(\Gamma)$. Similar constructions can be formulated for arrangements of general algebraic surfaces in $\mathbb{R}^4$, and for arrangements of simplices or algebraic surfaces in higher dimensions. We refer the reader to [3] for more details on the vertical decomposition scheme in those more general settings.

Preliminary analysis. The following two observations will be helpful in analyzing the complexity of $\mathcal{V}(\Gamma)$. First, it is easy to see that the complexity of $\mathcal{V}(\Gamma)$ is asymptotically the same as the complexity of $\mathcal{V}_2(\Gamma)$, and it is thus sufficient to bound the latter in order to bound the former. Second, the complexity of $\mathcal{V}_2(\Gamma)$ is asymptotically the same as the complexity of $\mathcal{V}_1(\Gamma)$ *plus* the number of y-vertical visibility events inside all the projection polyhedra P_{3D} of the prisms P of $\mathcal{V}_1(\Gamma)$. (Such an event is said to happen between two edges, f and f', of P_{3D} if they intersect a common y-vertical line l, and the segment $s \subset l$ that connects f and f' on this line lies completely inside P_{3D}.) We start by bounding the complexity of $\mathcal{V}_1(\Gamma)$.

Lemma 1. *Given a collection Γ of n 3-simplices in $\mathbb{R}^4$, the complexity of $\mathcal{V}_1(\Gamma)$ is $O(n^4\alpha(n))$. If Γ consists of hyperplanes, the complexity of $\mathcal{V}_1(\Gamma)$ is $O(n^4)$.*

Proof. During the construction of $\mathcal{V}_1(\Gamma)$, a z-vertical visibility wall is erected on $S \cap T$, for all $S, T \in \Gamma$. By construction, this wall is bounded from above by the lower envelope of the part of $\mathcal{A}(\Gamma)$ that lies above $S \cap T$ (within the z-vertical hyperplane spanned by $S \cap T$). Similarly, it is bounded from below by the upper envelope of the part of $\mathcal{A}(\Gamma)$ that lies below $S \cap T$. The complexity of these upper and lower envelopes is $O(n^2\alpha(n))$ [5]. Consequently, the complexity of the wall erected on $S \cap T$ is also $O(n^2\alpha(n))$.

The same arguments imply that the complexity of the visibility wall erected on each ∂S (for $S \in \Gamma$) is also $O(n^2\alpha(n))$. Since there are n such boundaries ∂S and $O(n^2)$ such intersections $S \cap T$, the overall number of features that are created during the construction of $\mathcal{V}_1(\Gamma)$ is $O(n^4\alpha(n))$.

If Γ is a collection of hyperplanes, $S \cap T$ is a 2-dimensional plane (for all $S, T \in \Gamma$), and the complexity of the above-mentioned envelopes is $O(n^2)$. This is because the envelopes are a portion of the zone of $S \cap T$ within the cross-section of $\mathcal{A}(\Gamma)$ in the vertical hyperplane spanned by $S \cap T$; the complexity of such a zone is $O(n^2)$ [6]. The number of created features in this case is thus $O(n^4)$. □

Visibility events. In light of the above, the bulk of this paper is devoted to bounding the number of y-vertical visibility events inside all polyhedra P_{3D}. To this end, we first classify the faces and edges of a projected prism P_{3D}, obtained as the z-vertical projection of a prism P of $\mathcal{V}_1(\Gamma)$. Each face of P_{3D} is a z-vertical projection of a face of P, which is a part of a 3-dimensional z-vertical wall erected on a 2-dimensional feature of $\mathcal{A}(\Gamma)$ (during the construction of $\mathcal{V}_1(\Gamma)$). These walls can be of three types.

- An (upward) visibility wall erected on $P_F \cap S$ (for some $S \in \Gamma$) that touches P_C (from below). Faces of P_{3D} that are projections of parts of such walls are said to be *red*.
- A (downward) visibility wall erected on $P_C \cap T$ (for some $T \in \Gamma$) that touches P_F (from above). Corresponding faces of P_{3D} are said to be *green*.
- A visibility wall erected on ∂U (for some $U \in \Gamma$) that touches P_F (from above) and P_C (from below). (Intuitively, the boundary of U is partly "floating" between P_F and P_C and the z-vertical wall erected on it reaches both P_F and P_C.) Corresponding faces of P_{3D} are said to be *blue*. (Note that in the case of hyperplanes there are no blue faces.)

Any edge of P_{3D} is incident to two faces of P_{3D}. Such edges can thus be classified into four types, depending on the types of the incident faces.

1. Edges incident to two red (or two green) faces. The two faces, by definition, correspond to parts of the visibility walls erected on $P_F \cap S$ and $P_F \cap T$ (respectively, on $P_C \cap S$ and $P_C \cap T$), for some $S, T \in \Gamma$. An edge incident to both of them thus corresponds to the common part of these two walls, which is the visibility wall erected on $P_F \cap S \cap T$ (respectively, $P_C \cap S \cap T$). We will denote such edges mnemonically by E_3 to signify that they are formed by an intersection of three simplices.
2. Edges incident to a red and a green face. Such an edge corresponds to the common part of two walls, one erected (upward) on $P_F \cap S$ and another erected (downward) on $P_C \cap T$, for some $S, T \in \Gamma$. This part is composed of z-vertical segments that touch both $P_F \cap S$ and $P_C \cap T$. We will denote such edges by E_{22} to signify that they are formed by interaction of two intersections, each of two simplices.
3. Edges incident to a red (or green) face and a blue face. Such an edge corresponds to the common part of two walls, one erected on $P_F \cap S$ (respectively, on $P_C \cap S$) and another erected on ∂T, for some $S, T \in \Gamma$. This part is composed of z-vertical segments that touch $P_F \cap S$ from above (respectively, $P_C \cap S$ from below), pass through ∂T and touch P_C (respectively, P_F). We

will denote such edges by E_{21} to signify that they are formed by an intersection of two simplices, and by a boundary of a third simplex.
An interesting special case is the one in which $S = T$, i.e. one face corresponds to $P_F \cap S$ or $P_C \cap S$, and another to ∂S. Such edges correspond to a visibility wall erected on $P_F \cap \partial S$ or $P_C \cap \partial S$, and will be denoted by E_2.

4. Edges incident to two blue faces. Such an edge corresponds to the common part of the visibility walls erected on ∂S and ∂T, for some $S, T \in \Gamma$. We will denote such edges by E_{11} to signify that they are formed by two boundaries of simplices.
An interesting special case is the one in which $S = T$. In this case the edge is incident to two blue faces that correspond to walls erected on two incident 2-dimensional features of ∂S, and it therefore corresponds to a visibility wall erected on a 1-dimensional feature of ∂S. Such edges will be denoted by E_1.

To recap, the possible mnemonic representations of the edges of P_{3D} are E_3, E_{22}, E_{21}, E_2 (a special case of E_{21}), E_{11}, and E_1 (a special case of E_{11}).

Each y-vertical visibility event inside P_{3D} corresponds to a y-vertical segment s that lies completely inside P_{3D}, such that s connects a point p on an edge of P_{3D} to a point p' on another edge of P_{3D}. Notice that p (as well as p') is a z-vertical projection of a specific z-vertical segment e (respectively, e'), whose bottom end-point lies on P_F, and whose top end-point lies on P_C. By construction, e and e' lie inside a common yz-parallel (2-dimensional) plane, which we denote by $\Pi_{e,e'}$. The 2-dimensional feature of $\mathcal{V}_2(\Gamma)$ that corresponds to s (that is, the "wall" that is erected on s inside P) is the trapezoid that has e and e' as its bases. This trapezoid is necessarily disjoint from Γ in its interior. We will sometimes denote a y-vertical visibility event by (e, e'), where e and e' are as above, and where the points on e have a smaller y-coordinate than the points on e'.

Fig. 1 provides an exhaustive visual catalogue of the possible types of y-vertical visibility events. There are 10 such types, determined by the type of edge that contains the point p and the type of edge that contains the point p'. For example, one such type of events is E_3E_{22}, in which the edge containing p is of type E_3 and the edge containing p' is of type E_{22} (or vice versa; this type can also be denoted by $E_{22}E_3$). For each type of events, the figure shows one representative configuration of simplices inside the plane $\Pi_{e,e'}$. Notice that each event is "defined" by six simplices — P_F, P_C, and four others.

Fig. 1 does not show events that involve E_2 and E_1 edges. Since these types of edges are special cases, the number of such events will be bounded by the analysis of events that involve the more general E_{21} and E_{11} edges.

Three types of events — E_3E_3, $E_{21}E_3$, and $E_{21}E_{21}$ — have alternative configurations, as illustrated in Fig. 2. These simpler configurations are not special cases of the configurations that are shown in Fig. 1(a), Fig. 1(d), and Fig. 1(f), respectively. Nevertheless, it turns out that the events portrayed in Fig. 1 are much harder to treat, and our analysis of their number can also be applied, in simplified form, to bounding the number of events that are in the alternative configurations. We will thus not treat these configurations explicitly.

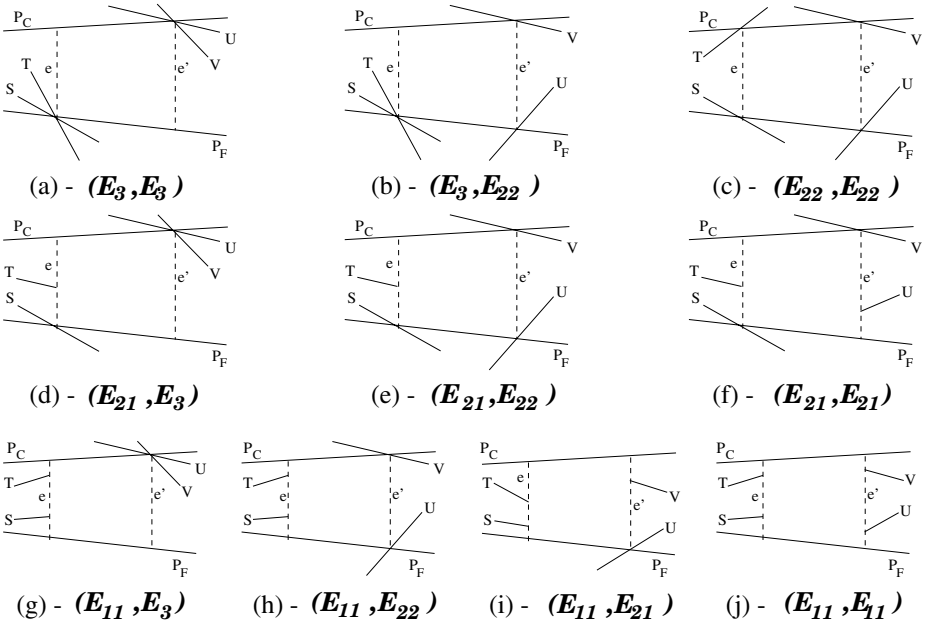


Fig. 1. All the possible types of y-vertical visibility events (up to symmetry).

3 Visibility Events in Arrangements of Hyperplanes

Analyzing vertical decompositions of arrangements of hyperplanes is simpler than in the (more general) case of 3-simplices, since hyperplanes do not have boundaries, and thus only three kinds of y-vertical visibility events can occur. Using the notation introduced in the previous section, these are E_3E_3, E_3E_{22} and $E_{22}E_{22}$ events. In order to prove the main result of this section, stated below, it is therefore sufficient to analyze only events of these three kinds.

Theorem 1. *The number of cells in the vertical decomposition of an arrangement of n hyperplanes in four dimensions is $O(n^4)$.*

In the following sequence of three lemmas, we analyze each kind of events in turn, and prove that there can only be $O(n^4)$ events of each kind. This is accomplished by charging each event to features of $\mathcal{A}(\Gamma)$ or of $\mathcal{V}_1(\Gamma)$, or to events that were analyzed previously.

Lemma 2. *The number of E_3E_3 events in an arrangement of n hyperplanes in 4-space is $O(n^4)$.*

Proof. For any three hyperplanes $P_F, S, T \in \Gamma$, we slide a point a on the line $P_F \cap S \cap T$ at constant speed (in any of the two possible directions, say in the positive direction of the w-axis, from $-\infty$ to $+\infty$). Consider a 2-dimensional yz-parallel plane Π_a that is attached to the point a, so that it contains a at all

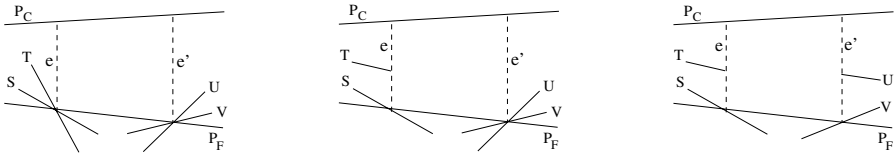


Fig. 2. Alternative (easier to analyze) configurations of some y-vertical visibility events.

times during the sliding. The plane Π_a "sweeps" (a part of) $\mathcal{A}(\Gamma)$ in a fixed direction at a fixed speed. Thus, it contains a dynamic arrangement of lines, such that each line moves with a fixed speed, and the slope of each line is fixed. Each such moving line corresponds to a hyperplane; two moving lines intersect in a moving point, corresponding to an intersection of two hyperplanes, that also moves along a linear trajectory with a fixed speed. At discrete moments in time, during the sliding, three lines intersect in a point — such "critical events" correspond to intersections of three hyperplanes that are swept by Π_a. Three of the moving lines are always in degenerate configuration, as they intersect in a point at all times. These are the lines (that correspond to) P_F, S, and T. (Abusing the notation slightly, we will denote $X \cap \Pi_a$, for $X \in \Gamma$, simply by X, and similarly for any other feature.)

During this sliding, we associate each E_3E_3 event that involves $P_F \cap S \cap T$ with a pair (U, Λ), where $U \in \Gamma$ and Λ is an edge in the (4-dimensional) zone of U. Since we go over all triples (P_F, S, T) in this fashion, all E_3E_3 events in $\mathcal{V}_2(\Gamma)$ will be associated with such pairs at the end of the process. We will prove that each such pair can only be charged (i.e. associated with) a constant number of times, which will imply that the number of E_3E_3 events is asymptotically the same as the number of such pairs in the whole arrangement. The latter number is $O(n^4)$, since there are n hyperplanes U, and the zone of each contains $O(n^3)$ edges Λ [6].

Consider one E_3E_3 event (e, e') that involves $P_F \cap S \cap T$. Assume, without loss of generality, that the bottom end-point of the segment e lies on $P_F \cap S \cap T$, and the top end-point of e' lies on an intersection of three other hyperplanes of Γ, $P_C \cap U \cap V$, such that an upward z-vertical ray obtained by extending e upwards hits U before hitting V (see Fig. 1(a)). By definition, there is a specific moment in time (denote it by t_0) during the sliding, at which the plane Π_a coincides with the plane $\Pi_{e,e'}$. At time $t_0 + \varepsilon$, where $\varepsilon > 0$ is arbitrarily small, the y-coordinate of the point $P_C \cap V$ is either smaller or larger than the y-coordinate of the point $P_C \cap U$. These two possibilities distinguish between E_3E_3 events of type A and type B, respectively; see Fig. 3.

Suppose the event (e, e') is of type A. Notice that, immediately after time t_0, U is "separated" (within Π_a) from $P_F \cap S \cap T$ by P_C and V, in the sense that U does not intersect the wedge bounded by P_C and V that contains $P_F \cap S \cap T$ at time t_0. Due to linearity of the trajectories, this implies that no event of type E_3E_3 involving both $P_F \cap S \cap T$ and U can happen after time t_0 while $P_F \cap S \cap T$

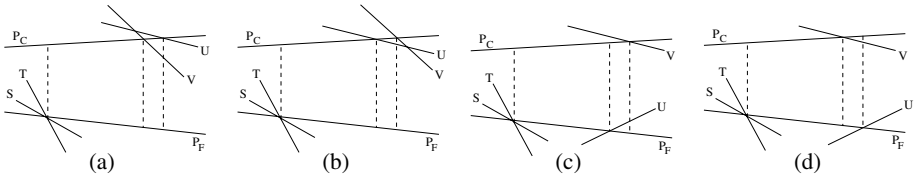


Fig. 3. The types of E_3E_3 and E_3E_{22} events. (a) and (b) show E_3E_3 events of types A and B, respectively, while (c) and (d) similarly show E_3E_{22} events of types A and B. All illustrations show a representative configuration inside the plane Π_a at time $t_0 + \varepsilon$.

lies inside this wedge. Notice that $P_F \cap S \cap T$ cannot cease to lie inside this wedge before intersecting either P_C or V. Such intersection corresponds to a vertex of $\mathcal{A}(\Gamma)$.

Let $\Lambda \subset P_F \cap S \cap T$ be the edge of $\mathcal{A}(\Gamma)$ that contains the bottom end-point of e, at the time (e, e') materializes. The preceding argument implies that no E_3E_3 event involving Λ and U can happen after time t_0. We associate the event (e, e') with the pair (U, Λ). Uniqueness of this association is ensured by the fact that, after the first such charge is made to some pair (U, Λ) (during sliding on the edge Λ), no other E_3E_3 event of Type A involving U and Λ can occur.

Events (e, e') of type B are handled by a fully symmetric charging argument, in which the direction of the sliding of a is reversed. We have thus shown that each E_3E_3 event can be associated with a pair (U, Λ) as described above, such that each such pair (U, Λ) is associated with at most one event of type A and at most one event of type B. This completes the proof of the lemma. □

Remark 1. A $G_{3,3}$ event is said to happen between two points, $p \in P_F \cap S \cap T$ and $p' \in P_C \cap U \cap V$, for some $P_F, P_C, S, T, U, V \in \Gamma$, if p and p' lie inside a common yz-parallel plane, and p and p' belong to the same cell of $\mathcal{A}(\Gamma)$; see Fig. 4(a). The proof of Lemma 2 can be modified in a straightforward fashion to bound the number of $G_{3,3}$ events by $O(n^4)$.

Lemma 3. *The number of E_3E_{22} events in an arrangement of n hyperplanes in 4-space is $O(n^4)$.*

Proof. For any $P_F, P_C, S, T \in \Gamma$, we slide a z-vertical segment a, such that its bottom end-point lies on $P_F \cap S \cap T$ and its top end-point lies on P_C, at constant speed (in any of the two possible directions), with the yz-parallel plane Π_a attached to it.

During the sliding, each E_3E_{22} event (e, e') that involves a will be associated with either a vertex of $\mathcal{A}(\Gamma)$, or a feature of $\mathcal{V}_1(\Gamma)$, or a $G_{3,3}$ event. Since we go over all quadruples (P_F, P_C, S, T) in this fashion, all E_3E_{22} events in $\mathcal{V}_2(\Gamma)$ will be associated with such features at the end of the process. The complexity of $\mathcal{A}(\Gamma)$ is $O(n^4)$; Lemma 1 bounds the number of features of $\mathcal{V}_1(\Gamma)$ by $O(n^4)$, while Remark 1 states that the number of $G_{3,3}$ events is also $O(n^4)$. As we will

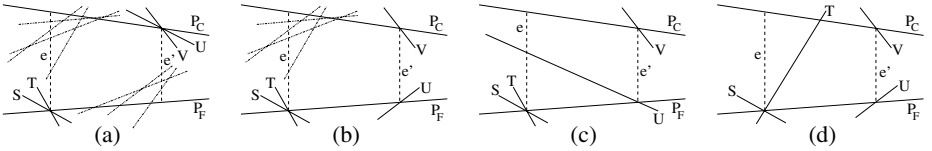


Fig. 4. Generalized visibility events. (a) shows a $G_{3,3}$ event, while (b–d) show three $G_{3,22}$ events. Notice that in the case of $G_{3,3}$ and $G_{3,22}$ events, as opposed to E_3E_3 and E_3E_{22} events, the trapezoid that has e and e' as its bases is not necessarily empty.

show, each such feature can only be charged (i.e. associated with) a constant number of times, which implies that the number of E_3E_{22} events is also $O(n^4)$.

Consider one E_3E_{22} event (e, e'), such that a coincides with e at time t_0. (All E_3E_{22} events (e, e'), such that a coincides with e', can be handled analogously.) Suppose that the bottom end-point of e' lies on $P_F \cap U$ and the top one lies on $P_C \cap V$, for some $U, V \in \Gamma$. The y-coordinate of the point $P_F \cap U$ (inside the plane Π_a) at time $t_0 + \varepsilon$ is either smaller or larger than the y-coordinate of the point $P_C \cap V$. These two possibilities distinguish between E_3E_{22} events of type A and type B, respectively, see Fig. 3. Assume that the event (e, e') is of type A. The other case can be handled by a symmetric argument with the direction of the sliding reversed, as in the proof of Lemma 2.

Claim. No E_3E_{22} event (g, g') can occur, such that the segment a coincides with g, after time t_0 and before some (moving) line X (corresponding to a hyperplane $X \in \Gamma$, which may be U) intersects either (1) the top end-point of a on P_C (Fig. 5(a)), or (2) the bottom end-point of a on $P_F \cap S \cap T$ (Fig. 5(b)), or (3) $P_C \cap V$ (Fig. 5(c)).

Proof. After time t_0 and before any of the above events happens, the relative interior of the segment connecting $P_C \cap V$ and the top end-point of a is disjoint from all hyperplanes of Γ (other than P_C). Therefore, in any E_3E_{22} event (g, g') as specified in the claim, the top end-point of g' has to lie on $P_C \cap V$ (while the bottom end-point has to lie on P_F). However, in this case g' is necessarily intersected by U, which prevents (g, g') from being an E_3E_{22} event. □

The claim implies that we can associate the event (e, e') with the first time (after time t_0) when some line X causes one of the three events listed in the claim to happen. Notice that an event of type (3) is a $G_{3,3}$ event, and that an event of type (2) is a vertex of $\mathcal{A}(\Gamma)$. Notice also that after time t_0 and prior to any of the events specified in the Claim, the segment a is disjoint in its interior from all hyperplanes of Γ. This ensures that any event of type (1) corresponds to a feature of $\mathcal{V}_1(\Gamma)$. Thus, we can associate the event (e, e') with a $G_{3,3}$ event, or a vertex of $\mathcal{A}(\Gamma)$, or a feature of $\mathcal{V}_1(\Gamma)$. As in the proof of Lemma 2, each of these events can be charged at most twice, once for each direction of the sliding of a. This implies that the number of E_3E_{22} events is $O(n^4)$. □

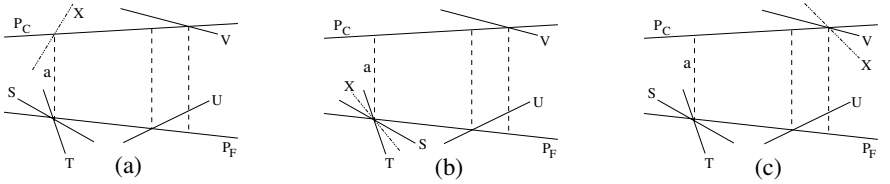


Fig. 5. The three types of events that are associated with E_3E_{22} events and occur at a later time.

Remark 2. A $G_{3,22}$ event is said to happen between three points, $p_1 \in P_F \cap S \cap T$, $p_2 \in P_F \cap U$, and $p_3 \in P_C \cap V$, for some $P_F, P_C, S, T, U, V \in \Gamma$, if all three points lie inside a common yz-parallel plane, the segment (p_1, p_2) is disjoint from all hyperplanes of Γ other than P_F, and the segment (p_2, p_3) is z-vertical and is disjoint from all hyperplanes of Γ; see Fig. 4(b–d). By combining ideas from the proofs of Lemmas 2 and 3, the number of $G_{3,22}$ events can be bounded by $O(n^4)$. We omit the details due to space limitations.

Lemma 4. *The number of $E_{22}E_{22}$ events in an arrangement of n hyperplanes in 4-space is $O(n^4)$.*

Proof. For any four hyperplanes $P_F, P_C, S, T \in \Gamma$, we slide a z-vertical segment a, such that its bottom end-point lies on $P_F \cap S$ and its top end-point lies on $P_C \cap T$, at constant speed (in any of the two possible directions), with the yz-parallel plane Π_a attached to it, as above. Assuming general position, the locus of points on $P_C \cap T$ that are intersected by the z-vertical hyperplane spanned by $P_F \cap S$ is a line. This implies that the trajectory of a is linear, and Π_a contains a dynamic arrangement of lines, as in the proof of Lemma 3.

Consider one $E_{22}E_{22}$ event (e, e'), such that a coincides with e at time t_0. Suppose that the bottom end-point of e' lies on $P_F \cap U$ and the top one lies on $P_C \cap V$, for some $U, V \in \Gamma$. Assume, without loss of generality, that the y-coordinate of the point $P_F \cap U$ (inside the plane Π_a) at time $t_0 + \varepsilon$ is smaller than the y-coordinate of the point $P_C \cap V$ (see Fig. 6(a)).

Similarly to the claim in the proof of Lemma 3, no $E_{22}E_{22}$ event (g, g') can occur, such that the segment a coincides with g, after time t_0 and before some (moving) line X (corresponding to a hyperplane $X \in \Gamma$, which may be U) intersects either (1) the top end-point of a on $P_C \cap T$ (Fig. 6(b)), or (2) the bottom end-point of a on $P_F \cap S$ (Fig. 6(c)), or (3) $P_C \cap V$ (Fig. 6(d)).

We can thus associate the event (e, e') with the first time some line X causes one of these three events to happen. Notice that an event of type (3) is a $G_{3,22}$ event, while the first (in time) event of type (1) or (2) corresponds to a feature of $\mathcal{V}_1(\Gamma)$. Thus, we can associate the event (e, e') with a $G_{3,22}$ event or a feature of $\mathcal{V}_1(\Gamma)$. As above, none of these events is charged more than twice, which implies that the number of $E_{22}E_{22}$ events is $O(n^4)$. □

Lemmas 2–4 bound the number of all y-vertical visibility events by $O(n^4)$, thereby implying Theorem 1.

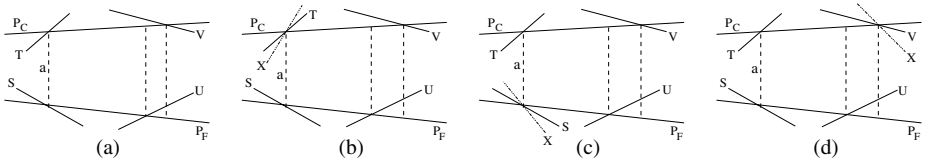


Fig. 6. The analysis of $E_{22}E_{22}$ events. (a) illustrates the configuration inside Π_a at time $t_0 + \varepsilon$; (b–d) depict the three types of events that are associated with $E_{22}E_{22}$ events and occur at a later time.

4 Visibility Events in Arrangements of Simplices

The analysis in the case of simplices is considerably more complex and technical than in the case of hyperplanes, since all of the 10 types of events illustrated in Fig. 1 can occur. In this case we have:

Theorem 2. *The number of cells in the vertical decomposition of an arrangement of n 3-simplices in four dimensions is $O(n^4\alpha(n)\log n)$.*

The theorem is proved in two stages. In the first stage we analyze all types of events, with the exception of E_3E_3 events, and prove that the number of events of each type is $O(n^4\alpha(n))$. This stage uses ideas similar to those introduced in the previous section. That is, each event is charged to a feature of $\mathcal{A}(\Gamma)$, or of $\mathcal{V}_1(\Gamma)$, or to an event of a type that was analyzed previously. Special care is taken to ensure that each such feature or event is only charged at most a constant number of times.

For establishing the stated complexity bound, we cannot charge events to pairs (U, Λ), where U is a simplex of Γ and Λ is an edge in the zone of U, as we did in the proof of Lemma 2. The reason is that the zone of a 3-simplex in an arrangement of n 3-simplices in 4-space is only known to have complexity $O(n^3 \log n)$ [2,10], and not $O(n^3)$, as in the case of hyperplanes. This implies that the overall number of pairs (U, Λ) as above is $O(n^4 \log n)$. This, however, is not good enough, due to the use of the Tagansky technique in the second stage of the proof (see below), which adds a logarithmic overhead factor to the bound that is proved in the first stage. Thus, to achieve the asserted $O(n^4\alpha(n)\log n)$ overall bound, we have to prove a bound of $O(n^4\alpha(n))$ in the first stage.

Accordingly, a charging scheme similar to those used in Lemmas 3 and 4 is presented for most types of events. Unfortunately, the charging schemes are much more involved, and so is their analysis. Due to space limitations, all the details are omitted from this extended abstract, and will appear in the full version of the paper.

In the second stage of the proof, we bound the number of E_3E_3 events using the technique introduced by Tagansky for analyzing substructures in arrangements of linear surfaces [10]. Details are again omitted due to space limitations, and we only describe the general idea here.

A charging scheme is presented, such that each E_3E_3 event is charged either to a feature of $\mathcal{A}(\Gamma)$, or to a feature of $\mathcal{V}_1(\Gamma)$, or to an event of one of the

types that were analyzed in the first stage, or to a *1-level* E_3E_3 event (see [10] for definitions). To utilize the Tagansky technique, we carefully bound the number of times each 1-level E_3E_3 event is charged in this fashion. In particular, we prove that although each E_3E_3 event makes 4 'units' of charge, each 1-level E_3E_3 event is charged by at most 2 such units. As a result, we obtain a recurrence for the number of E_3E_3 events, which does solve to the asserted bound of $O(n^4\alpha(n)\log n)$, thus completing the proof of Theorem 2.

5 Conclusion

We have presented improved upper bounds for vertical decompositions of arrangements of hyperplanes and 3-simplices in four dimensions. The current focus of our work is to utilize the ideas and techniques introduced in this paper to prove a near-quartic upper bound on the complexity of vertical decompositions of arrangements of fixed-degree algebraic surfaces in four dimensions. Such a result appears to be attainable, and would have significant algorithmic applications.

Acknowledgements. The author wishes to thank Micha Sharir for his help in preparing this paper, as well as Danny Halperin for helpful discussions concerning the studied problems.

References

1. P. K. Agarwal and M. Sharir. Arrangements and their applications. In J.-R. Sack and J. Urrutia, editors, *Handbook of Comput. Geom.*, pages 49–119. 2000.
2. B. Aronov and M. Sharir. Castles in the air revisited. *Discrete Comput. Geom.*, 12:119–150, 1994.
3. B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. A singly-exponential stratification scheme for real semi-algebraic varieties and its applications. *Theoret. Comput. Sci.*, 84:77–105, 1991. Also in Proc. ICALP (1989), pp. 179–193.
4. K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17:830–847, 1988.
5. H. Edelsbrunner. The upper envelope of piecewise linear functions: Tight complexity bounds in higher dimensions. *Discrete Comput. Geom.*, 4:337–343, 1989.
6. H. Edelsbrunner, R. Seidel, and M. Sharir. On the zone theorem for hyperplane arrangements. *SIAM J. Comput.*, 22(2):418–429, 1993.
7. L. J. Guibas, D. Halperin, J. Matoušek, and M. Sharir. On vertical decomposition of arrangements of hyperplanes in four dimensions. *Discrete Comput. Geom.*, 14:113–122, 1995.
8. D. Halperin. Arrangements. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Comput. Geom.*, pages 389–412. 1997.
9. M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.
10. B. Tagansky. A new technique for analyzing substructures in arrangements of piecewise linear surfaces. *Discrete Comput. Geom.*, 16:455–479, 1996.

Optimization over Zonotopes and Training Support Vector Machines

Marshall Bern[1] and David Eppstein[2]

[1] Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304
bern@parc.xerox.com
[2] Univ. of California, Irvine, Dept. Inf. & Comp. Sci., Irvine, CA 92697.
eppstein@ics.uci.edu

Abstract. We make a connection between classical polytopes called zonotopes and Support Vector Machine (SVM) classifiers. We combine this connection with the ellipsoid method to give some new theoretical results on training SVMs. We also describe some special properties of C-SVMs for $C \to \infty$.

1 Introduction

A statistical classifier algorithm maps a set of *training vectors*—positively and negatively labeled points in $\mathbb{R}^d$—to a decision boundary. A Support Vector Machine (SVM) is a classifier algorithm in which the decision boundary depends on only a subset of training vectors, called the *support vectors* [18]. This limited dependence on the training set helps give SVMs good *generalizability*, meaning that SVMs are resistant to overtraining even in the case of large d. Another key idea associated with SVMs is the use of a *kernel function* in computing the dot product of two training vectors. For example, the usual dot product $v \cdot w$ could be replaced by $k(v, w) = (v \cdot w)^2$ (quadratic kernel) or by $k(v, w) = \exp(-\|v - w\|^2)$ (radial basis function). The kernel function [14] in effect maps the original training vectors in $\mathbb{R}^d$ into a higher-dimensional (perhaps infinite-dimensional) *feature space* $\mathbb{R}^{d'}$; a linear decision boundary in $\mathbb{R}^{d'}$ then determines a nonlinear decision surface back in $\mathbb{R}^d$. For good introductions to SVMs see the tutorial by Burges [3] or the book by Cristianini and Shawe-Taylor [6].

The basic *maximum margin* SVM applies to the case of linearly separable training vectors, and divides positive and negative vectors by a farthest-apart pair of parallel hyperplanes, as shown in Figure 1(a). The decision boundary itself is typically the hyperplane halfway between the boundaries. Computational geometers might expect that the extension of the SVM to the non-separable case would divide positive and negative vectors by a least-overlapping pair of half-spaces bounded by parallel hyperplanes, as shown in Figure 1(b). This generalization, however, may be overly sensitive to outliers, and hence the method of choice is a more robust *soft margin* classifier, called a C-SVM [4,18] or ν-SVM [13] depending upon the precise formulation. Parameter C is a user-chosen penalty for errors.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 111–121, 2001.

Computing the maximum margin classifier for n vectors in $\mathbb{R}^d$ amounts to solving a quadratic program (QP) with about d variables and n linear constraints. If the feature vectors are not *explicit* (that is, kernel functions are being used), then the usual Lagrangian formulation gives a QP with about $n+d$ variables and linear constraints. Similarly, the soft margin classifier—with or without explicit feature vectors—is computed in a Lagrangian formulation with about $n+d$ variables and linear constraints. The jump from d to $n+d$ variables can have a great impact on the running time and choice of QP algorithm. Recent results in computational geometry [8,11] give fast QP algorithms for the case of large n and small d, algorithms requiring about $\mathcal{O}(nd) + (\log n)\exp(\mathcal{O}(\sqrt{d}\,))$ arithmetic operations. The best bound on the number of arithmetic operations for a QP with $n+d$ variables and constraints is about $\mathcal{O}((n+d)^3 L)$, where L is the precision of the input data [16].

In this paper, we show that the jump from d to $n+d$ is not necessary for soft margin classifiers with explicit feature vectors. More specifically, we describe training algorithms with running time near linear in n and polynomial in d and input precision, for two different scenarios: C set by the user and $C \to \infty$. The second scenario also introduces a natural measure of separability of point sets. Our algorithms build upon a geometric view of soft margin classifiers [1,5] and the ellipsoid method for convex optimization. Due to their reliance on explicit feature vectors and the ellipsoid method, and also due to the fact that SVMs are more suited to the case of moderate n and large d than to the case of large n and small d, our algorithms have little practical importance. On the other hand, our results should be interesting theoretically. We view the soft margin classifier as a problem defined over a zonotope, a type of polytope that admits an especially compact description. Accordingly, our algorithms have lower complexity than either the vertex or facet descriptions of the polytopes.

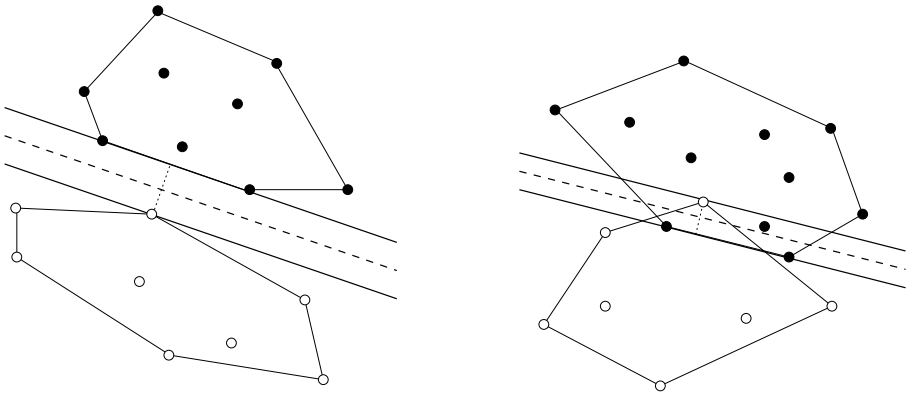


Fig. 1. (a) The maximum margin SVM classifier for the separable case. The dashed line shows the decision boundary. (b) The most natural generalization to the non-separable case is not popular.

2 SVM Formulations

We adopt the usual SVM notation and mostly follow the presentation of Bennett and Bredensteiner [1]. The training vectors are $x_1, x_2, \ldots, x_n$, points in $\mathbb{R}^d$. The corresponding labels are $y_1, y_2, \ldots, y_n$, each of which is either $+1$ or -1. Let $I_+ = \{\, i \mid y_i = +1 \,\}$ and $I_- = \{\, i \mid y_i = -1 \,\}$. We use w and x to denote vectors in $\mathbb{R}^d$ and b to denote a scalar. We use the dot product notation $w \cdot x$, but in this section $w \cdot x$ could be standing in for the kernel function $k(w, x)$.

In the maximum margin SVM we seek parallel hyperplanes defined by the equations $w \cdot x = b_+$ and $w \cdot x = b_-$ such that $w \cdot x_i \le b_-$ for all $i \in I_-$ and $w \cdot x_i \ge b_+$ for all $i \in I_+$. The signed distance between these two hyperplanes—the *margin*—is $\frac{b_+ - b_-}{\|w\|}$ and hence can be maximized by minimizing $\|w\|^2 - (b_+ - b_-)$.

$$\min_{w,\, b_+,\, b_-} \quad \|w\|^2 - (b_+ - b_-) \qquad \text{subject to} \tag{1}$$
$$x_i \cdot w \ge b_+ \quad \text{for} \quad i \in I_+, \qquad x_i \cdot w \le b_- \quad \text{for} \quad i \in I_-.$$

A popular choice for the decision boundary is the plane halfway between the parallel hyperplanes, $w \cdot x = (b_+ + b_-)/2$, and hence each unknown vector x is classified according to the sign of $w \cdot x - (b_+ + b_-)/2$.

In the linearly separable case, we can set $b_+ = 1 - b$ and $b_- = -1 - b$ (thereby rescaling w) and obtain the following optimization problem, the standard form in most SVM treatments [3].

$$\min_{w,\, b} \quad \|w\|^2 \qquad \text{subject to} \tag{2}$$
$$x_i \cdot w + b \ge 1 \quad \text{for} \quad i \in I_+, \qquad x_i \cdot w + b \le -1 \quad \text{for} \quad i \in I_-.$$

Notice that this QP has $d+1$ variables and n linear constraints. At the solution, w is a linear combination of x_i's, $2/\|w\|$ gives the margin, and $w \cdot x + b = 0$ gives the halfway decision boundary.

The dual problem to maximizing the distance between parallel hyperplanes separating the positive and negative convex hulls is to minimize the distance between points inside the convex hulls. Thus the dual in the separable case is the following.

$$\min_{\alpha_i} \left\| \sum_{i \in I_+} \alpha_i x_i - \sum_{i \in I_-} \alpha_i x_i \right\|^2 \text{ subject to } \; 0 \le \alpha_i \le 1, \; \sum_{i \in I_+} \alpha_i = 1, \; \sum_{i \in I_-} \alpha_i = 1. \tag{3}$$

Karush-Kuhn-Tucker (complementary slackness) conditions show that the optimizing value of w for (1) is given by the optimizing values of α_i for (3): $w = \sum_{i \in I_+} \alpha_i x_i - \sum_{i \in I_-} \alpha_i x_i$. The vectors x_i with $\alpha_i > 0$ are called the *support vectors*.

The soft margin SVM adds slack variables to formulation (1), and then penalizes solutions proportional to the sum of these variables. Slack variable ξ_i measures the *error* for training vector x_i, that is, how far x_i lies on the wrong

side of the parallel hyperplane for x_i's class.

$$\min_{w,\, b_+,\, b_-,\, \xi_i} \quad \|w\|^2 + (b_+ - b_-) + \mu \sum_{i=0}^{n} \xi_i \qquad \text{subject to} \tag{4}$$
$$\xi_i \geq 0 \ \ \forall i, \quad x_i \cdot w \geq b_+ - \xi_i \ \text{ for } \ i \in I_+, \quad x_i \cdot w \leq b_- + \xi_i \ \text{ for } \ i \in I_-.$$

The standard C-SVM formulation [18] again sets $b_+ = 1 - b$ and $b_- = -1 - b$.

$$\min_{w,\, b,\, \xi_i} \quad \|w\|^2 + C \sum_{i=0}^{n} \xi_i \qquad \text{subject to} \tag{5}$$
$$\xi_i \geq 0 \ \ \forall i, \quad x_i \cdot w + b \geq 1 - \xi_i \ \text{ for } \ i \in I_+, \quad x_i \cdot w + b \leq -1 + \xi_i \ \text{ for } \ i \in I_-.$$

In formulation 5, the decision boundary is $w \cdot x = b$. Formulation (4), however, does not set the decision boundary, but only its direction. Crisp and Burges [5] write that because "originally the sum of ξ_i's term arose in an attempt to approximate the number of errors", the best option might be to run a "simple line search" to find the decision boundary that actually minimizes the number of training set errors.

The dual of formulation (4) in the separable case minimizes the distance between points inside "reduced" or "soft" convex hulls [1,5].

$$\min_{\alpha_i} \left\| \sum_{i \in I_+} \alpha_i x_i - \sum_{i \in I_-} \alpha_i x_i \right\|^2 \text{ subject to } \ 0 \leq \alpha_i \leq \mu, \ \sum_{i \in I_+} \alpha_i = 1, \ \sum_{i \in I_-} \alpha_i = 1. \tag{6}$$

See Figure 2. The *reduced convex hull* of points x_i, $i \in I_+$, is the set of convex combinations of $\alpha_i x_i$ with each $\alpha_i \leq \mu$. (Notice that in (4) there is no reason to consider $\mu > 1$.) We shall say more about reduced convex hulls in the next section.

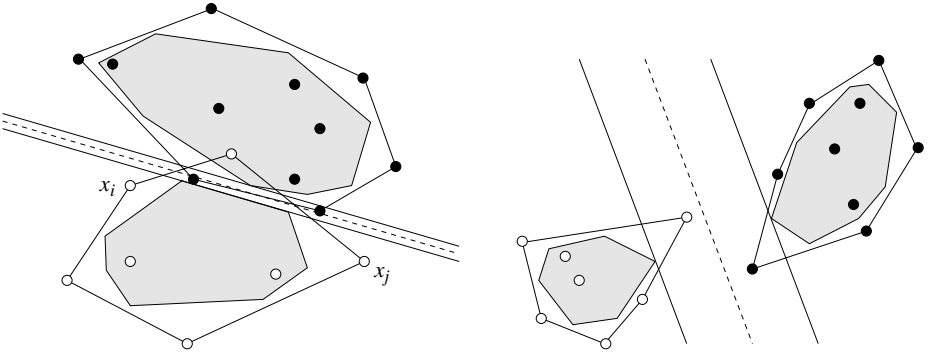


Fig. 2. (a) Soft margin SVMs maximize the margin between reduced convex hulls. (b) Although the soft margin is often explained as a way to handle non-separability, it can help in the separable case as well.

The dual view highlights a slight difference between formulations (4) and (5). Formulation (4) allows the direct setting of the reduced convex hulls. Parameter μ limits the influence of any single training point; if the user expects no more than four outliers in the training set, then an appropriate choice of μ might be $1/9$ in order to ensure that the majority of the support vectors are non-outliers. If the reduced convex hulls intersect, the solution to (4) is the least-overlapping pair of half-spaces, as in Figure 1(b). Formulation (5) is also always feasible—unlike the standard hard margin formulation (2)—but it never allows the reduced convex hulls to intersect. As $C \to \infty$ the reduced convex hulls either fill out their convex hulls (the separable case) or continue growing until they asymptotically touch (the non-separable case).

3 Reduced Convex Hulls and Zonotopes

Assume $0 \leq \mu \leq 1$ and define the positive and negative reduced convex hulls by

$$H_{+\mu} = \Big\{ \sum_{i \in I_+} \alpha_i x_i \;\Big|\; \sum_{i \in I_+} \alpha_i = 1, \quad 0 \leq \alpha_i \leq \mu \Big\},$$

$$H_{-\mu} = \Big\{ \sum_{i \in I_-} \alpha_i x_i \;\Big|\; \sum_{i \in I_-} \alpha_i = 1, \quad 0 \leq \alpha_i \leq \mu \Big\}.$$

Figure 3 shows the reduced convex hull of three points x_1, x_2, and x_3 for various values of μ. The reduced convex hull grows from the centroid at $\mu = 1/3$ to the convex hull at $\mu = 1$; for $\mu < 1/3$ it is empty. In Figure 2, μ is a little less than $1/2$.

A reduced convex hull is a special case of a *centroid polytope*, the locus of possible weighted averages of points each with an unknown weight within a certain range [2]. For reduced convex hulls, each weight α_i has the same range $[0, \mu]$ and the sum of the weights is constrained to be 1. In [2] we related centroid polytopes in $\mathbb{R}^d$ to special polytopes, called zonotopes, in $\mathbb{R}^{d+1}$. We repeat the connection here, specialized to the case of reduced convex hulls.

Let v_i denote $(x_i, 1)$, the vector in $\mathbb{R}^{d+1}$ that agrees with x_i on its first d coordinates and has 1 as its last coordinate. Define

$$Z_{+\mu} = \Big\{ \sum_{i \in I_+} \alpha_i v_i \;\Big|\; 0 \leq \alpha_i \leq \mu \Big\}.$$

Polytope $Z_{+\mu}$ is a Minkowski sum[1] of line segments of the form $S_i = \{ \alpha_i v_i \mid 0 \leq \alpha_i \leq \mu \}$. The Minkowski sum of line segments is a special type of convex polytope called a *zonotope* [2,7]. Polytope $H_{+\mu}$ is the cross-section of $Z_{+\mu}$ with the $(d+1)$-st coordinate (which by construction is also $\sum_i \alpha_i$) equal to one. Of course, $H_{-\mu}$ can also be related to a zonotope in the same way. The following lemmas state the property of zonotopes and reduced convex hulls that underlies our algorithms. Lemma 2 is implicit in Keerthi et al.'s iterative nearest-point approach to SVM training [9].

[1] The *Minkowski sum* of sets A and B in $\mathbb{R}^{d+1}$ is $\{p + q \mid p \in A \text{ and } q \in B\}$.

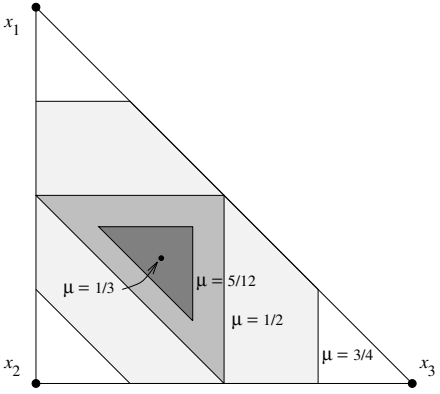


Fig. 3. The reduced convex hull of 3 points ranges from the centroid to the convex hull.

Lemma 1. *Let Z be a zonotope that is the Minkowski sum of n line segments in $\mathbb{R}^d$. There is an algorithm with $\mathcal{O}(nd)$ arithmetic operations for optimizing a linear function over Z.*

Proof. Assume that we are trying to find a vertex v in zonotope Z extreme in direction w, that is, that maximizes the dot product $w \cdot v$. Assume that Z is the Minkowski sum of line segments of the form $S_i = \{\, \alpha_i v_i \mid 0 \le \alpha_i \le \mu \,\}$, where $v_i \in \mathbb{R}^{d+1}$. We simply set each α_i independently to 0 or μ, depending upon whether the projection of v_i onto w is negative or positive. □

Lemma 2. *There is an algorithm with $\mathcal{O}(nd)$ arithmetic operations for optimizing a linear function over a reduced convex hull of n points in $\mathbb{R}^d$.*

Proof. Assume that we are trying to find a vertex x in zonotope $H_{+\mu}$ extreme in direction w. Order the x_i's with $y_i = +1$ according to their projection onto vector w, breaking ties arbitrarily. In decreasing order by projection along w, set the corresponding α_i's to μ until doing so would violate the constraint that $\sum_{i \in I_+} \alpha_i = 1$. Set α_i for this "transitional" vector to the maximum value allowed by this constraint, and finally set the remaining α_i's to 0. Then $x = \sum_{i \in I_+} \alpha_i x_i$ maximizes $w \cdot x$. □

An interesting combinatorial question asks for the worst-case complexity of a reduced convex hull $H_{+\mu}$. The vertex x of $H_{+\mu}$ that is extreme for direction w can be associated with the set of x_i's for which $\alpha_i > 0$. If $\mu = 1/k$, then as in Lemma 2, x's set is the first k points in direction w, a set of k points that can be separated from the other $n-k$ points by a hyperplane normal to w. And conversely, each separable set of k points defines a unique vertex of H_μ. Hence the maximum number of vertices of $H_{+\mu}$ is equal to the maximum number of *k-sets* for n points in $\mathbb{R}^d$, which is known to be $\omega(n^{d-1})$ and $o(n^d)$ [17,19]. In [2] we showed that a more general centroid polytope in which each point x_i has α_i

between 0 and μ_i (that is, different weight bounds for different points) may have complexity $\Theta(n^d)$.

We can also apply the argument in the proof of Lemma 2 to say something about the optimizing values of the variables in (4) and (6) for the non-separable case. (Alternatively we can derive the same statements from the Karush-Kuhn-Tucker conditions.) Each of H_+ and H_- has a transition in the sorted order of the x_i's when projected along the normal w to the parallel pair of hyperplanes. For x_i with $i \in I_+$, $\alpha_i = 0$ if $x_i \cdot w$ lies on the "right" side of the transition, $0 \leq \alpha_i \leq \mu$ if $x_i \cdot w$ coincides with the transition, and $\alpha_i = \mu$ if x_i lies on the "wrong" side of the transition. Of course an analogous statement holds for x_i for $i \in I_-$. As usual, the support vectors are those x_i with $\alpha_i > 0$. Thus all training set errors are support vectors. In Figure 2(a) there are six support vectors: two transitional unfilled dots (marked x_i and x_j) and one wrong-side unfilled dot, along with one transitional and two wrong-side filled dots.

4 Ellipsoid-Based Algorithms

We first assume that μ has been fixed in advance, perhaps using some knowledge of the expected number of outliers or the desired number of support vectors. We give an algorithm for solving formulation (6).

One approach would be to compute the vertices of $H_{+\mu}$ and $H_{-\mu}$ and then use formulation (1) with positive and negative training vectors replaced by the vertices of $H_{+\mu}$ and $H_{-\mu}$ respectively. However, the number of vertices of $H_{+\mu}$ and $H_{-\mu}$ may be very large, so this algorithm could be very slow.

So instead we exploit a polynomial-time equivalence between separation and optimization (see for example [15], chapter 14.2). The input to the *separation problem* is a point q and a polytope P (typically given by a system of linear inequalities). The output is either a statement that q is inside P or a hyperplane separating q and P. The input to the *optimization problem* is a direction w and a polytope P. The output is either a statement that P is empty, a statement that P is unbounded in direction w, or a point in P extreme for direction w. The two problems are related by projective duality,[2] and a subroutine for solving one can be used to solve the other in a number of calls that is polynomial in the dimension d and the input precision, that is, the number of bits in q or w plus the maximum number of bits in an inequality defining P.

In our case, the polytope is not given by inequalities, but rather as a Minkowski sum of line segments; this presentation has an impact on the required precision. If the input precision is L, the maximum number of bits in one of the feature vectors x_i, then the maximum number of bits in a vertex of the polytope is $\mathcal{O}(d^2 L \log n)$. What is new is the $\mathcal{O}(\log n)$ term, resulting from the fact that a vertex of a zonotope is a sum of up to n input vectors.

[2] The more famous direction of this equivalence is that separation—which can be solved directly by checking each inequality—implies optimization. This result is a corollary of Khachiyan's ellipsoid method.

Theorem 1. *Given n explicit feature vectors in $\mathbb{R}^d$ and μ with $1/n \le \mu \le 1$, there is a polynomial-time algorithm for computing a soft margin classifier, with the number of arithmetic operations linear in n and polynomial in d, L, and* $\log n$.

Proof. As in [9], consider the polytope P that is the Minkowski sum of $H_{+\mu}$ and $-H_{-\mu}$, that is, $P = \{\, v_+ - v_- \mid v_+ \in H_{+\mu} \text{ and } v_- \in H_{-\mu} \,\}$. We are trying to minimize over P the convex quadratic objective function $\|v\|^2$, that is, the length of a line segment between $H_{+\mu}$ and $H_{-\mu}$.

For a given direction w, we can find the solution $v = v_+ - v_-$ to the linear optimization problem for P by using Lemma 2 to find the v_+ optimizing w over $H_{+\mu}$ and the v_- optimizing w over $H_{-\mu}$. Now given a point $q \in \mathbb{R}^d$, we can use this observation and the polynomial-time equivalence between separation and optimization to solve the separation problem for q and P in time linear in n and polynomial in d and L. We can use this solution to the separation problem for P as a subroutine for the ellipsoid method (see [10,15]) in order to optimize $\|v\|^2$ over P. Given an optimizing choice of $v = v_+ - v_-$, it is easy to find the best pair of parallel hyperplanes and a decision boundary, either the C-SVM decision boundary or some other reasonable choice within the parallel family. □

Now assume that we are in the non-separable case. We shall show how to solve for the maximum μ for which the reduced convex hulls have non-intersecting interior, that is, the μ for which the margin is 0. This choice of μ corresponds to $C \to \infty$ and the objective function simplifying to $\sum_i \xi_i$ in formulation (5).

This choice of μ has two special properties. First, among all settings of C, $C \to \infty$ tends to give the fewest support vectors. To see this, imagine shrinking the shaded regions in Figure 2(a). Support vectors are added each time one of the parallel hyperplanes crosses a training vector. On the other hand, a support vector may be lost occasionally when the number of reduced convex hull vertices on the parallel hyperplanes changes, for example, if the vertex supporting the upper parallel line in Figure 2(a) slipped off to the right of the segment supporting the lower parallel line.

Second, the μ for which the margin is zero gives a natural measure of the separability of two point sets. For simplicity, let $|I_+| = |I_-| = n/2$ and normalize the zero-margin μ by $\mu^* = (\mu - 2/n)/(1 - 2/n)$. The separability measure μ^* runs from 0 to 1, with 0 meaning that the centroids coincide and 1 meaning that the convex hulls have disjoint interiors. Computing the zero-margin μ as the maximum value of a dual variable α_i using formulation (5) above is no harder than training a C-SVM, and in the case of explicit features, it should be significantly easier, as we now show.

We can formulate the problem as minimizing μ subject to

$$\sum_{i \in I_+} \beta_i x_i = \sum_{i \in I_-} \beta_i x_i, \qquad \sum_{i \in I_+} \beta_i = 1, \qquad \sum_{i \in I_-} \beta_i = 1, \qquad 0 \le \beta_i \le \mu.$$

As above, let v_i denote $(x_i, 1)$, the vector in $\mathbb{R}^{d+1}$ that agrees with x_i on its first d coordinates and has 1 as its last coordinate. Letting $\alpha_i = \beta_i/\mu$, we can rewrite

the problem as maximizing

$$1/\mu = \sum_{i \in I_+} \alpha_i = \sum_{i \in I_-} \alpha_i$$

subject to

$$\sum_{i \in I_+} \alpha_i v_i = \sum_{i \in I_-} \alpha_i v_i, \qquad 0 \le \alpha_i \le 1. \tag{7}$$

Yet another way to state the problem is to ask for the point with maximum $(d+1)$-st coordinate in $Z_+ \cap Z_-$, where

$$Z_+ = \Big\{ \sum_{i \in I_+} \alpha_i v_i \;\Big|\; 0 \le \alpha_i \le 1 \Big\}, \qquad Z_- = \Big\{ \sum_{i \in I_-} \alpha_i v_i \;\Big|\; 0 \le \alpha_i \le 1 \Big\}.$$

Polytopes Z_+ and Z_- are each zonotopes, Minkowski sums of line segments of the form $S_i = \{ \alpha_i v_i \mid 0 \le \alpha_i \le 1 \}$.

Theorem 2. *Let Z_1 and Z_2 be zonotopes defined by a total of n line segments in $\mathbb{R}^d$. There is an algorithm for optimizing a linear objective function over $Z_1 \cap Z_2$, with the number of arithmetic operations linear in n and polynomial in d, L, and $\log n$.*

Proof. Given a point q and zonotope Z_i, $1 \le i \le 2$, we can use Lemma 1 and the polynomial-time equivalence between separation and optimization to solve the separation problem for q and Z_i in time linear in n and polynomial in d, L and $\log n$. We can solve the separation problem for the intersection of zonotopes $Z_1 \cap Z_2$ simply by solving it separately for each zonotope. We now use the equivalence between separation and optimization in the other direction to conclude that we can also solve the optimization problem for an intersection of zonotopes. □

The proof of the following result then follows from the ellipsoid method in the same way as the proof of Theorem 1.

Corollary 1. *Given n explicit feature vectors in $\mathbb{R}^d$, there is a polynomial-time algorithm for computing the maximum μ for which $H_{+\mu}$ and $H_{-\mu}$ are linearly separable, with the number of arithmetic operations linear in n and polynomial in d, L, and $\log n$.*

Theorem 1 and Corollary 1 can be extended to some cases of implicit feature vectors. For example, the quadratic kernel $k(v, w) = (v \cdot w)^2$ for vectors $v = (v_1, v_2)$ and $w = (w_1, w_2)$ in $\mathbb{R}^2$ is equivalent to an ordinary dot product in $\mathbb{R}^3$, namely $k(v, w) = \Phi(v) \cdot \Phi(w)$, where $\Phi(v) = \left(v_1^2, \sqrt{2} v_1 v_2, v_2^2\right)$. In general [3], a polynomial kernel $k(v, w) = (v \cdot w)^p$ amounts to lifting the training vectors from $\mathbb{R}^d$ to $\mathbb{R}^{d'}$ where $d' = \binom{d+p-1}{p}$. Radial basis functions, however, give $d' = \infty$, and the SVM training problem seems to necessarily involve $n + d$ variables. (The rather amazing part is that it is a combinatorial optimization problem at all!)

5 Discussion and Conclusions

In this paper we have connected SVMs to some recent results in computational geometry and mathematical programming. These connections raise some new questions, both practical and theoretical.

Currently the best practical algorithms for training SVMs, Platt's sequential minimal optimization (SMO) [12] and Keerthi et al.'s nearest point algorithm (NPA) [9], can be viewed as interior-point methods that iteratively optimize the margin over line segments. Both algorithms make use of heuristics to find line segments close to the exterior, meaning line segments with α_i weights set to either 0 or C.

Computational geometry may have a practical algorithm to contribute for the case of n large and d small, say $n \approx 100,000$ and $d \approx 20$: the generalized linear programming (GLP) paradigm of Matoušek et al. [8,11]. The training vectors need not actually live in $\mathbb{R}^d$ for small d, so long as the GLP dimension of the problem is small, where the GLP dimension is the number of support vectors in any subproblem defined by a subset of the training vectors.

On the theoretical side, we are wondering about the existence of strongly polynomial algorithms for QP problems over zonotopes. Due to the combinatorial equivalence of zonotopes and arrangements, the graph diameter of a zonotope is known to be only $\mathcal{O}(n)$; polynomial graph diameter is of course a necessary condition for the existence of a polynomial-time simplex-style algorithm.

Acknowledgments. David Eppstein's work was done in part while visiting Xerox PARC, and supported in part by NSF grant CCR-9912338. We would also like to thank Yoram Gat for a number of helpful discussions.

References

1. K.P. Bennett and E.J. Bredensteiner. Duality and geometry in SVM classifiers. *Proc. 17th Int. Conf. Machine Learning*, Pat Langley, ed., Morgan Kaufmann, 2000, 57–64.
2. M. Bern, D. Eppstein, L. Guibas, J. Hershberger, S. Suri, and J. Wolter. The centroid of points with approximate weights. *3rd European Symposium on Algorithms*, Corfu, 1995. Springer Verlag LNCS 979, 1995, 460–472.
3. C.J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, Vol. 2, No. 2, 1998, 121–167. http://svm.research.bell-labs.com/SVMrefs.html
4. C. Cortes and V. Vapnik. Support vector networks. *Machine Learning*, 1995, 273–297.
5. D.J. Crisp and C.J.C. Burges. A geometric interpretation of ν-SVM classifiers. *Advances in Neural Information Processing Systems 12*. S.A. Solla, T.K. Leen, and K.-R. Müller, eds. MIT Press, 1999. http://svm.research.bell-labs.com/SVMrefs.html
6. N. Cristianini and J. Shawe-Taylor. *Support Vector Machines*. Cambridge U. Press, 2000.

7. H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, Springer Verlag, 1987.
8. B. Gärtner. A subexponential algorithm for abstract optimization problems. *SIAM J. Computing* 24 (1995), 1018–1035.
9. S.S. Keerthi, S.K. Shevade, C. Bhattacharyya, and K.R.K. Murthy. A fast iterative nearest point algorithm for support vector machine classifier design. *IEEE Trans. Neural Networks* 11 (2000), 124–136. http://guppy.mpe.nus.edu.sg/∼mpessk/
10. M.K. Kozlov, S.P. Tarasov, L.G. Khachiyan. Polynomial solvability of convex quadratic programming. *Soviet Math. Doklady* 20 (1979) 1108–1111.
11. J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. Tech. Report B 92-17, Freie Univ. Berlin, Fachb. Mathematik, 1992
12. J.C. Platt. Fast training of support vector machines using sequential minimal optimization. Chapter 12 of *Advances in Kernel Methods: Support Vector Learning*, B. Schölkopf, C. Burges, and A. Smola, eds. MIT Press, 1998, 185–208. http://www.research.microsoft.com/∼jplatt
13. B. Schölkopf, A.J. Smola, R. Williamson, and P. Bartlett. New support vector algorithms. *Neural Computatation* Vol. 12, No. 5, 2000, 1207–1245. NeuroCOLT2 Technical Report NC2-TR-1998-031. 1998.
http://svm.first.gmd.de/papers/tr-31-1998.ps.gz
14. B. Schölkopf, S. Mika, C.J.C. Burges, P. Knirsch, K.-R. Müller, G. Rätsch, and A.J. Smola. Input space vs. feature space in kernel-based methods. *IEEE Trans. on Neural Networks* 10 (1999) 1000-1017.
http://svm.research.bell-labs.com/SVMrefs.html
15. A. Schrijver. *Theory of Linear and Integer Programming* John Wiley & Sons, 1986.
16. M.J. Todd. Mathematical Programming. Chapter 39 of *Handbook of Discrete and Computational Geometry*, J.E. Goodman and J. O'Rourke, eds., CRC Press, 1997.
17. G. Tóth. Point sets with many k-sets. *Proc. 16th Annual ACM Symp. Computational Geometry*, 2000, 37–42.
18. V. Vapnik. *Statistical Learning Theory*. Wiley, 1998.
19. R.T. Živaljević and S.T. Vrećica. The colored Tverberg's problem and complexes of injective functions. *J. Comb. Theory, Series A*, 61 (1992) 309–318.

Reporting Intersecting Pairs of Polytopes in Two and Three Dimensions*

Pankaj K. Agarwal[1], Mark de Berg[2], Sariel Har-Peled[3], Mark H. Overmars[2], Micha Sharir[4], and Jan Vahrenhold[5]

[1] Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708, USA.
pankaj@cs.duke.edu.
[2] Institute of Information and Computing Sciences, Utrecht University, P.O.Box 80.089, 3508 TB Utrecht, The Netherlands.
{markdb,markov}@cs.ruu.nl.
[3] Department of Computer Science, DCL 2111, University of Illinois, 1304 West Springfield Ave., Urbana, IL 61801, USA.
sariel@cs.uiuc.edu.
[4] School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel; and Courant Institute of Mathematical Sciences, New York University, New York, NY 10012, USA. sharir@math.tau.ac.il.
[5] Westfälische Wilhelms-Universität Münster, Institut für Informatik, 48149 Münster, Germany.
jan@math.uni-muenster.de.

Abstract. Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of m convex polytopes in $\mathbb{R}^d$, for $d = 2, 3$, with a total of n vertices. We present output-sensitive algorithms for reporting all k pairs of indices (i, j) such that P_i intersects P_j. For the planar case we describe a simple algorithm with running time $O(n^{4/3} \log n + k)$, and an improved randomized algorithm with expected running time $O((n \log m + k)\alpha(n) \log n)$ (which is faster for small values of k). For $d = 3$, we present an $O(n^{8/5+\varepsilon} + k)$-time algorithm, for any $\varepsilon > 0$. Our algorithms can be modified to count the number of intersecting pairs in $O(n^{4/3} \log^{O(1)} n)$ time for the planar case, and in $O(n^{8/5+\varepsilon})$ time and $\mathbb{R}^3$.

1 Introduction

Computing intersections in a set of geometric objects is a fundamental problem in computational geometry. A basic version of this problem is when the objects

* P.A. was also supported by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants EIA–9870724, EIA–997287, and CCR–9732787, and by a grant from the U.S.-Israeli Binational Science Foundation. M.S. was supported by NSF Grant CCR-97-32101, by a grant from the Israel Science Fund (for a Center of Excellence in Geometric Computing), by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University, and by a grant from the U.S.-Israeli Binational Science Foundation.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 122–134, 2001.

are line segments in the plane. Indeed, computing the intersecting pairs in a set of n line segments was one of the first problems studied in computational geometry: Already in 1979, Bentley and Ottmann [7] described an algorithm for this problem with $O((n+k)\log n)$ running time, where k is the number of intersecting pairs of segments. Since then much research has been done on this problem, culminating in optimal—that is, with $O(n\log n+k)$ running time—deterministic algorithms by Chazelle and Edelsbrunner [12] and Balaban [5], and simpler randomized algorithms by Clarkson and Shor [14] and Mulmuley [19].

Another well-studied variant of the problem is the red-blue intersection problem. Here one is given a set of red segments and a set of blue segments, and the goal is to report all bichromatic intersections. If there are no monochromatic intersections, then the problem can be solved in $O(n\log n+k)$ time by applying an optimal standard line-segment intersection algorithm; when the red segments and the blue segments both form simply connected subdivisions, then the problem can even be solved in $O(n+k)$ time [15]. The situation becomes considerably more complicated when there are monochromatic intersections. Applying a standard line-segment intersection algorithm will not lead to an output-sensitive algorithm because it may report a quadratic number of monochromatic intersections even when there are no bichromatic intersections. Somehow one has to avoid processing all the monochromatic intersections. Agarwal and Sharir [3] showed that one can detect whether the two sets intersect in $O(n^{4/3+\varepsilon})$ time.[1] Later Agarwal [1] and Chazelle [9] gave $O(n^{4/3}\log^{O(1)} n+k)$-time algorithm to report all k red-blue intersections. Basch *et al.* [6] presented a deterministic $O(\lambda_{t+2}(n+k)\log^3(n))$ algorithm for the case where the set of red segments is connected and the set of blue segments is connected; this algorithm also works for the case of Jordan arcs, each pair of which intersect at most t times. Its running time is $O(\lambda_{t+2}(n+k)\log^3(n))$, where $\lambda_s(n)$, the maximum length of an (n,s) Davenport-Schinzel sequence, is an almost linear function of n for any fixed s. This bound was later improved for the case of segments to $O((n+k)\log^2(n)\log\log n)$ by Brodal and Jacob [8]. Har-Peled and Sharir [17] give a randomized algorithm with $O(\lambda_{t+2}(n+k)\log n)$ running time for the case of Jordan arcs, as above.

We are interested in the case in which the input consists of convex polygons in the plane. We want to compute all intersecting pairs of polygons. More formally, we are given a set $\mathcal{P}=\{P_1,\ldots,P_m\}$ of m convex polygons in $\mathbb{R}^2$ with a total of n vertices, and we want to report all k pairs of indices i,j such that P_i intersects P_j. (The polygons are considered to be 2-dimensional regions, so two polygons intersect also in the case that one of them is fully contained inside the other.) If each polygon P_i has constant complexity, then the number of intersections between pairs of edges will not exceed the total number of intersecting pairs of polygons by more than a constant factor, and one can solve the problem in $O(n\log n+k)$ time, by a straightforward modification of the algorithms mentioned above for reporting segment intersections. If the given polygons do

[1] The meaning of a bound like this is that for any $\varepsilon>0$ there exists a constant $c=c(\varepsilon)$ that depends on ε, so that the bound holds with c as the constant of proportionality.

not have constant complexity, then the problem becomes considerably harder because the intersection of a pair of the given polygons can have many vertices. Regarding each input polygon as a collection of segments will thus not lead to an output-sensitive algorithm in this case.

Gupta *et al.* [16] nevertheless managed to develop an output-sensitive algorithm for this case that runs in time $O(n^{4/3+\varepsilon}+k)$. The algorithm first computes a trapezoidal decomposition for each polygon. Then it computes, using a multilevel partition tree, those pairs of intersecting trapezoids such that the leftmost intersection point of the trapezoids is also the leftmost intersection point of the corresponding polygons. This way it is ensured that each intersecting pair of polygons is reported exactly once.

We develop two new algorithms for this problem. The first algorithm is randomized and combines hereditary segment trees [13] with the above mentioned red-blue intersection algorithm of Har-Peled and Sharir [17]. Its expected running time is $O((n \log m + k)\alpha(n) \log n)$ and it is significantly faster than the algorithm of Gupta *et al.* when $k = o(n^{4/3})$. In addition, the algorithm also works for convex splinegons (that is, convex shapes whose boundary is composed of Jordan arcs) with only a minor increase in running time; this is not the case for the algorithm of Gupta *et al.* Our algorithm can be made deterministic at the expense of an additional polylogarithmic factor.

Our second algorithm has $O(n^{4/3} \log n + k)$ running time, and is thus slightly faster than our first algorithm for $k = \Omega(n^{4/3})$. It is related to the algorithm of Gupta *et al.*—it uses partition trees and similar techniques to search for the rightmost intersection points of intersecting pairs of polygons—but it is conceptually simpler and it has a slightly better running time.

The main advantage of our approach over Gupta *et al.*'s is that it generalizes to the 3-dimensional version of the problem: Given a set $\mathcal{P} = \{P_1, \ldots, P_m\}$ of m convex polytopes in $\mathbb{R}^3$ with a total of n vertices, report all k pairs of indices (i, j) such that P_i intersects P_j. For this problem, no subquadratic algorithm was known. We generalize our second 2-dimensional algorithm, and obtain an algorithm with running time $O(n^{8/5+\varepsilon}+k)$, for any $\varepsilon > 0$. Such a generalization seems hard for the algorithm of Gupta *et al.*, as the vertical decomposition of a convex polytope can have quadratic complexity. Note that our algorithm for the 3-dimensional case has the same running time as the best known algorithm for the much simpler problem of reporting all intersecting pairs in a set of triangles in $\mathbb{R}^3$ [2].

2 The Planar Case

Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of m convex polygons in the plane, with a total of n vertices. For simplicity, we assume that none of the polygons has a vertical edge and that all the vertex coordinates are distinct; we can enforce this in $O(n \log n)$ time by applying a suitable rotation. For a polygon P_i, we define ℓ_i to be the leftmost point of P_i and r_i to be the rightmost point of P_i (since there are no vertical edges, ℓ_i and r_i are uniquely defined). They partition the boundary

of P_i into two convex chains: the *upper chain*, denoted U_i, and the *lower chain*, denoted L_i.

We first describe an algorithm whose running time is near-linear in n and k, and then a worst-case optimal algorithm for the case of large k; its worst-case running time is $O(n^{4/3} \log n + k)$.

2.1 A Near-Linear Randomized Algorithm

We present a randomized algorithm that reports, in $O((n \log m + k)\alpha(n) \log n)$ expected time, all k intersecting pairs of polygons in $\mathcal{P}$. For each polygon P_i, we define s_i to be the segment connecting ℓ_i to r_i; we call s_i the *spine* of P_i. Let $\mathcal{SP}$ denote the set of all the spines.

Our algorithm starts by constructing a *hereditary* segment tree T on (the x-projections of) the spines of $\mathcal{SP}$ [13]. Each node v of T is associated with a vertical strip W_v and with a subset $\mathcal{SP}(v)$ of spines. A spine s_i intersecting W_v is *short* at v if at least one of its endpoints lies in the interior of W_v, otherwise it is *long*. The set $\mathcal{SP}(v)$ is the subset of spines that intersect W_v and are short at the parent of v. If v is the root, then $\mathcal{SP}(v) = \mathcal{SP}$. Let $\mathcal{P}(v) = \{P_i \mid s_i \in \mathcal{SP}(v)\}$. A polygon is short (resp., long) at v if its spine is short (resp., long) at v. As shown in [13], $\sum_v |\mathcal{P}(v)| = O(m \log m)$.

We assume that $\mathcal{SP}(v)$ and $\mathcal{P}(v)$ are clipped to within W_v. At each node v of the tree, we will report all pairs (i, j) such that

($\star$) the rightmost intersection point of P_i and P_j lies inside W_v and P_i is long at v.

The following lemma is straightforward from the structure of hereditary segment trees.

Lemma 1. *For every pair of intersecting polygons P_i and P_j, there is exactly one node v of T at which property ($\star$) holds.*

Let k_v be the number of pairs that satisfy property ($\star$) at a node v. Then $\sum_v k_v = k$. Our procedure will ensure that a pair (i, j) is reported only once, at the node where ($\star$) is satisfied, but it will spend roughly $O(\log n)$ time for each intersecting pair.

Fix a node v. Let $\mathcal{P}_L \subseteq \mathcal{P}(v)$ denote the subset of long polygons at v, and let $\mathcal{P}_S \subseteq \mathcal{P}(v)$ denote the subset of short polygons at v. Denote the set of spines of $\mathcal{P}_L$ by $\mathcal{SP}_L$, the set of their upper chains by $\mathcal{U}_L$, and the set of their lower chains by $\mathcal{L}_L$. The sets $\mathcal{SP}_S$, $\mathcal{U}_S$, and $\mathcal{L}_S$ are defined analogously for the short polygons. Again, all these objects are clipped to within W_v. Let n_v denote the total number of edges in (the clipped) $\mathcal{P}_L$ and $\mathcal{P}_S$. As above, the structure of hereditary segment trees implies that $\sum_v n_v = O(n \log m)$. Finally, we define R_S to be the set of right endpoints of the spines in $\mathcal{SP}(v)$ that lie in the interior of W_v. Note that every point in R_S is the right endpoint of an (unclipped) original spine in $\mathcal{SP}$. Let μ_v be the number of intersection points between $\mathcal{SP}_L$ and $\mathcal{SP}(v) \cup \partial\mathcal{P}(v)$ plus the number of intersection points between the upper

(resp. lower) chains of $\mathcal{P}_L$ and the lower (resp. upper) chains of $\mathcal{P}(v)$, where $\partial\mathcal{P}(v) = \left\{\partial P \;\middle|\; P \in \mathcal{P}(v)\right\}$. We have:

Lemma 2. $\sum_{v\in T} \mu_v = O(k)$.

Since all the vertex coordinates are distinct, there exists at most one spine in $\mathcal{SP}(v)$ whose right endpoint r_i lies on the right boundary of W_v. We can easily compute in $O(n_v)$ time all polygons of $\mathcal{P}(v)$ that contain r_i. We now describe how we report all the other pairs that satisfy $(\star)$ at v.

We construct, in $O(n_v \log n_v + \mu_v)$ time, the arrangement $\mathcal{A} = \mathcal{A}(\mathcal{SP}_L)$ of the spines of the long polygons [12]. We also add the vertical lines bounding W_v to $\mathcal{A}$. Each face f of $\mathcal{A}$ is a convex polygon, so we can compute the intersections between a line and ∂f in $O(\log n_v)$ time. We preprocess $\mathcal{A}$, in $O((n_v+\mu_v)\log n_v)$ time, for planar point-location queries [20]. For each edge e of $\mathcal{P}(v)$, we locate its left endpoint in $\mathcal{A}$ and then trace it through $\mathcal{A}$, spending $O(\log n_v)$ time at each face of $\mathcal{A}$ that e intersects.

For each face $f \in \mathcal{A}$, we report the pairs (i, j) that satisfy $(\star)$ and for which the rightmost point of $P_i \cap P_j$ lies inside f. This is accomplished in the following three stages.

(a) Report all pairs (i, j) such that $P_i q \in \mathcal{P}_L$ contains the right endpoint $r_j \in R_S$ and $r_j \in f$.
(b) Report all pairs (i, j) such that the lower chain of $P_i \in \mathcal{P}_L$ intersects the upper chain of $P_j \in \mathcal{P}(v)$ and the rightmost point of their intersection lies inside f.
(c) Report all pairs (i, j) such that the upper chain of $P_i \in \mathcal{P}_L$ intersects the lower chain of $P_j \in \mathcal{P}(v)$ and the rightmost point of their intersection lies inside f.

It is easily verified that stages (a)–(c) indeed report all the desired intersections. Since (b) and (c) are symmetric, we omit the description of (c).

Containments of rightmost points. Let $R(f) \subset R_S$ be the subset of right endpoints that lie inside f. We wish to report all pairs (i, j) such that $r_j \in R(f)$ lies inside $P_i \in \mathcal{P}_L$. Let $\mathcal{P}(f) \subseteq \mathcal{P}_L$ denote the set of long polygons that contain f in their interior (i.e., for a polygon $P \in \mathcal{P}(f)$, we have $f \subseteq P$), and let $\mathcal{Q}(f) \subseteq \mathcal{P}_L$ denote the set of polygons whose boundaries intersect f. Let n_f denote the number of vertices of the polygons in $\mathcal{Q}(f)$ that lie inside f, and let n'_f denote the number of edges in $\mathcal{Q}(f)$ that intersect f but their endpoints do not lie inside f. Then $\sum_f n_f \le n_v$ and $\sum_f n'_f \le \mu_v$. Obviously, $|\mathcal{Q}(f)| \le n_f + n'_f$. Since we have already traced the edges of $\mathcal{P}_L(v)$ through $\mathcal{A}$, we have $\mathcal{Q}(f)$ at our disposal. However, we do not store $\mathcal{P}(f)$ explicitly for each face f because the resulting storage could be quite large.

Note that every point in $R(f)$ lies inside every polygon in $\mathcal{P}(f)$, so we report every pair in $\mathcal{P}(f) \times R(f)$. In order to compute the polygons of $\mathcal{P}(f)$, we perform a plane sweep over $\mathcal{A}$ and the collection of long polygons. The events of the sweep are (i) all the vertices of $\mathcal{A}$; (ii) left and right endpoints of polygons in $\mathcal{P}_L$; and (iii)

intersections of boundaries of polygons in $\mathcal{P}_L$ with the edges of $\mathcal{A}$. The number of events is $O(n_v + \mu_v + k_v)$. The $\mathcal{C}$-structure consists of the intersections of the sweep line with the edges of $\mathcal{A}$. Each interval between consecutive intersections represents a face f of $\mathcal{A}$; we store there the current set $\mathcal{Q}(f)$. Each polygon P_i of $\mathcal{P}_L$ appears in at most two intervals of the $\mathcal{C}$-structure, and we associate with it the union E_i of the intervals of the $\mathcal{C}$-structure that lie *strictly* between these two intervals. We finally convert the $\mathcal{C}$-structure into a segment tree, which stores the intervals E_i.

Updating the $\mathcal{C}$-structure at each sweep event is easy, and takes logarithmic time. When we reach a point $r_j \in R_S$, we simply report all the ℓ_j intervals E_i that contain the interval of the face of $\mathcal{A}$ that contains r_j. This can be done in time $O(\ell_j + \log n_v)$. In total, this step takes time $O((n_v + \mu_v + k_v) \log n_v)$.

Next, for every point $r_j \in R(f)$, we report the polygons in $\mathcal{Q}(f)$ that contain r_j. We build a *union tree* Ψ on the polygons in $\mathcal{Q}(f)$, which is a minimum-height binary tree whose leaves store the polygons of $\mathcal{Q}(f)$. Each node ξ of Ψ is associated with the subset $\mathcal{Q}_\xi \subseteq \mathcal{Q}(f)$ of polygons that are stored at the leaves of the subtree rooted at ξ. Let ν_ξ be the total number of vertices of the polygons in $\mathcal{Q}_\xi$ that lie in the interior of f, and let ν'_ξ be the number of edges of the polygons in $\mathcal{Q}_\xi$ that intersect f but whose endpoints do not lie inside f; we have $\sum_\xi \nu_\xi = O(n_f \log n_v)$ and $\sum_\xi \nu'_\xi = O(n'_f \log n_v)$. Let L_ξ (resp., U_ξ) denote the set of maximal connected portions of the lower (resp., upper) chains of the polygons in $\mathcal{Q}_\xi$ that lie inside f. At each node ξ, we compute the lower envelope $\mathcal{L}_\xi$ of L_ξ and the upper envelope $\mathcal{U}_\xi$ of U_ξ. These envelopes have $O((\nu_\xi+\nu'_\xi)\alpha(n_v))$ breakpoints. If we have already computed the lower and upper envelopes of the children of ξ, then $\mathcal{L}_\xi, \mathcal{U}_\xi$ can be computed in an additional $O((\nu_\xi + \nu'_\xi)\alpha(n_v))$ time. We store the sequences of breakpoints of L_ξ (and U_ξ) in an array, sorted from left to right. For each breakpoint, we store the segment that appears on the envelope immediately to its left. We also apply fractional cascading [10] so that for a given x-coordinate x_0, if we know the breakpoint of $\mathcal{L}_\xi$ (resp. $\mathcal{U}_\xi$) that is immediately to the right of x_0, we can compute, in $O(1)$ time, the breakpoints of $\mathcal{L}_\zeta, \mathcal{L}_\eta$ (resp. $\mathcal{U}_\zeta, \mathcal{U}_\eta$) that lie to the right of x_0, where ζ, η are the children of ξ. The total time spent in preprocessing Ψ is $O((n_f + n'_f)\alpha(n_v) \log n_v)$.

For each point $r_j \in R(f)$, we find all polygons in $\mathcal{Q}(f)$ containing r_j by traversing the union tree in a top-down manner. Suppose we are at a node ξ of Ψ. Since f is not crossed by any spine, r_j does not lie in any polygon of $\mathcal{Q}_\xi$ if and only if r_j lies below all the chains in L_ξ (i.e., lies below $\mathcal{L}_\xi$) and above all the chains in U_ξ (i.e., lies above $\mathcal{U}_\xi$). We thus find the breakpoints of $\mathcal{L}_\xi, \mathcal{U}_\xi$ that lie immediately to the right of r_j and determine in $O(1)$ time whether r_j lies below $\mathcal{L}_\xi$ and above $\mathcal{U}_\xi$. If the answer is yes, we conclude that r_j does not lie in any polygon of $\mathcal{Q}_\xi$, and we stop. If ξ is a leaf and r_j lies inside the only polygon, say P_i, in $\mathcal{Q}_\xi$, then we return the pair (i, j). If ξ is not a leaf and r_j lies inside a polygon of $\mathcal{Q}_\xi$, we recursively visit the children of ξ. Suppose r_j lies inside k_j polygons of $\mathcal{Q}(f)$, then the query procedure visits $O(1+k_j \log n_v)$ nodes of Ψ. It spends $O(\log n_v)$ time at the root and $O(1)$ at any other node, so the time spent in processing r_j is $O((1 + k_j) \log n_v)$. Hence, the algorithm spends

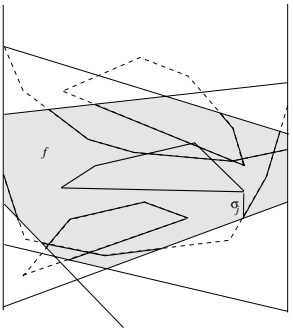


Fig. 1. A face f of $\mathcal{A}$, the associated sets $U(f)$, $L(f)$, and $\mathcal{SP}(f)$, and an added vertical segment.

$O((n_f + n'_f)\alpha(n_v) + \sum_{r_j \in R(f)}(1 + k_j)) \log n_v)$ time at face f. Summing over all the faces of $\mathcal{A}$, we obtain that the total time spent in reporting the pairs that satisfy condition (a), over all faces f of $\mathcal{A}$, is $O((n_v + \mu_v + k_v)\alpha(n_v) \log n_v)$.

Intersections between long lower chains and upper chains. For a face f of $\mathcal{A}$, let $L(f)$ denote the set of maximal connected portions of the chains in $\mathcal{L}_L$ that lie inside f, let $U(f)$ denote the set of maximal connected portions of upper chains of (short and long) polygons in $\mathcal{P}(v)$ that lie inside f, and let $\mathcal{SP}(f)$ denote the set of portions of short spines inside f. Since we have traced the edges of $\mathcal{P}(v)$ through $\mathcal{A}$, the sets $L(f)$ and $U(f)$ are already available for all faces f. We will report all pairs (i, j) that satisfy $(\star)$ and whose rightmost intersection points lie inside f. See Figure 1 for an illustration.

The endpoints of all chains in $L(f)$ lie on ∂f because they are portions of long chains. Let A_f be the set of edges that constitute $L(f)$ and ∂f; set $a_f = |A_f|$. The union of A_f is connected. If both endpoints of a chain $\gamma \in U(f)$ lie in the interior of f, then γ is the entire upper chain of a short polygon P_j. In this case, we add a vertical segment σ_j from the right endpoint r_j of P_j downwards until it meets ∂f. Let B_f denote the union of the set of edges that constitute $U(f)$ and ∂f, and the set of vertical segments that we have just added; set $b_f = |B_f|$. By construction, the union of B_f is also connected because all the upper chains in $U(f)$ are connected to ∂f after introducing the vertical segments. Since the unions of A_f and of B_f are both connected, we can use the randomized algorithm of Har-Peled and Sharir [17] to compute all I_f intersection points between the segments of A_f and of B_f that lie in the interior of f, in $O((a_f + b_f + I_f)\alpha(n_v) \log n_v)$ expected time.

The total expected running time spent in reporting the pairs that satisfy property (b) is $\sum_f O((a_f + b_f + I_f)\alpha(n_v) \log n_v)$. Each endpoint of a segment of A_f or of B_f is either a vertex of $\mathcal{P}(v)$, or an intersection point of a long spine and an edge of $\mathcal{P}(v)$, or the lower endpoint of a vertical segment σ_j. Therefore, $\sum_f (a_f + b_f) = O(n_v + \mu_v)$. The expected running time is thus $O((n_v + \mu_v + \sum_f I_f)\alpha(n_v) \log n_v)$.

We call an intersection point of $e \in A_f$ and $e' \in B_f$ *real* if e is an edge of a lower chain in $L(f)$ and e' is an edge of an upper chain in $U(f)$; otherwise we call the intersection point *virtual*. We report a pair (i, j) if there exists an edge e_i of P_i in A_f and an edge e_j of P_j in B_f such that the intersection point of e_i and e_j is the rightmost vertex of $P_i \cap P_j$.

Each real intersection point is an intersection point of $\mathcal{L}_L$ and the upper chains of $\mathcal{P}(v)$, so the total number of real intersection points, summed over all faces of $\mathcal{A}$, is $O(\mu_v)$. Since ∂f does not intersect the relative interior of any segment in $U(f)$ or $L(f)$, a virtual intersection point is an intersection point $e \cap e'$, where e is an edge of the lower chain of a long polygon P_i and e' is the vertical segment σ_j emanating from the right endpoint r_j of (the upper chain of) a short polygon P_j. We can ignore intersections on ∂f because they correspond to degenerate intersections between A_f and B_f, and, in any case, their number is only $O(\mu_v)$. Since P_i is a long polygon, its spine s_i is in $\mathcal{SP}_L$. Therefore, s_i lies above the interior of the face f and thus above r_j. The intersection of e and σ_j implies that r_j is inside P_i. We charge the intersection point $e \cap e'$ to the pair (i, j). Each pair (i, j) is charged by at most one virtual intersection point and the pair (i, j) is reported at v, therefore the total number of virtual intersection points, summed over all faces of $\mathcal{A}$, is at most k_v. Hence, $\sum_f I_f = O(k_v + \mu_v)$, and the total expected time spent in executing stage (b) is $O((n_v + k_v + \mu_v)\alpha(n_v) \log n_v)$.

We have thus described procedures for reporting all intersecting pairs that satisfy properties (a)–(c) at a node v of T. The total expected time we spend at v is $O((n_v + k_v + \mu_v)\alpha(n_v) \log n_v)$. Since $\sum_v n_v = O(n \log m)$, $\sum_v k_v = k$, and $\sum_v \mu_v = O(k)$ (Lemma 2), we obtain the following result.

Theorem 1. *Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be m convex polygons in the plane with a total of n vertices. All k pairs of indices (i, j) such that P_i intersects P_j can be reported in $O((n \log m + k)\alpha(n) \log n)$ expected time.*

Remark 1. (i) To get a worst-case time bound instead of an expected time bound, we can replace the algorithm of Har-Peled and Sharir [17] used in the second part of the algorithm by an algorithm of Basch *et al.* [6]. This will increase the time bound by a polylogarithmic factor.

(ii) The algorithm also works when the boundaries of the polygons are composed of Jordan arcs instead of straight edges, provided the polygons are still convex. If t is the maximum number of times any pair of Jordan arcs intersect, the running time of the algorithm becomes $O((\lambda_{t+2}(n) \log m + \lambda_{t+2}(k)) \log n)$.

2.2 An Alternative Deterministic Algorithm

Let P_i and P_j be two intersecting polygons of $\mathcal{P}$. As above, the rightmost vertex of $P_i \cap P_j$ is either r_i, or r_j, or an intersection point of the upper chain of P_i with the lower chain of P_j, or an intersection point of the lower chain of P_i with the upper chain of P_j. Using this observation, we can report the intersecting pairs of polygons as follows.

Let $V = \{r_i \mid 1 \le i \le m\}$. We first report all intersecting pairs of polygons for which the rightmost vertex of the intersection polygon is the rightmost vertex of one of the two polygons. A vertex r_i is the rightmost vertex of $P_i \cap P_j$ if and only if $r_i \in P_j$. For each P_i, we therefore report $P_i \cap V$. Using the range-searching data structure of Matoušek [18], we preprocess V, in time $O((m^{2/3}n^{2/3}+n)\log n)$, into a data structure of size $O(m^{2/3}n^{2/3}+n)$, and query it with each P_i. For a polygon P_i, all μ_i points of $P_i \cap V$ can be reported in time $O(|P_i|(m^{2/3}/n^{1/3})\log n + \mu_i)$. Hence, the total time spent in this step is $O(m^{2/3}n^{2/3}\log n + n\log n + \mu)$ where $\mu = \sum_{i=1}^{m} |P_i \cap V| \le k$.

Next, we report the pairs (i, j) such that the rightmost vertex of $P_i \cap P_j$ is an intersection point of an edge of P_i with an edge of P_j. Let U be the set of segments in the upper chains of the polygons in $\mathcal{P}$, and let L be the set of segments in the lower chains of these polygons. We compute all ν intersecting pairs of segments between U and L. This can be accomplished in $O(n^{4/3}\log^{2/3} n + \nu)$ time [1,9]. Suppose that an edge e of the upper chain of P_i and an edge e' of the lower chain of P_j intersect. We check in $O(1)$ time whether $e \cap e'$ is the rightmost vertex of $P_i \cap P_j$, and, if so, report the pair (i, j). Since an upper chain intersects a lower chain in at most two points, the number of intersections between U and L is at most $2k$, where k is the number of intersecting pairs of polygons in P.

Hence, we obtain the following result.

Theorem 2. *Let $\mathcal{P}$ be a set of m convex polygons in the plane with a total of n vertices. All k pairs of indices (i, j) such that P_i intersects P_j can be reported in $O(n^{4/3}\log n + k)$ time.*

Remark 2. (i) Since the data structure in [18] can count the number of points lying inside a k-gon in time $O(k(m^{2/3}/n^{1/3})\log n)$ time using $O((m^{2/3}n^{2/3} + m)\log n)$ preprocessing and the number of intersections between L and U can be counted in time $O(n^{4/3}\log n)$ time, the number of intersecting pairs of polygons can be counted in time $O(n^{4/3}\log n)$.

(ii) As in Agarwal and Sharir [4], we can use a more sophisticated data structure to improve the running time of the algorithm to $O(m^{2/3}n^{2/3}\log^c n + k)$, for an appropriate constant c.

3 The Three-Dimensional Case

Let $\mathcal{P} = \{P_1, \ldots, P_m\}$ be a set of m convex polytopes in $\mathbb{R}^3$ with a total of n vertices. We present an algorithm, with running time $O(n^{8/5+\varepsilon} + k)$, for any $\varepsilon > 0$, which reports all k pairs of indices (i, j) such that P_i intersects P_j. Our approach is similar to the algorithm described in Section 2.2. We compute the *bottom* vertex, i.e., the vertex with the minimum z-coordinate, of each nonempty intersection polytope $P_{ij} = P_i \cap P_j$, and report the corresponding pairs (i, j). The bottom vertex of an intersection polytope P_{ij} is either the bottom vertex of P_i, or the bottom vertex of P_j, or the intersection point of an edge of P_i and a face of P_j, or the intersection point of a face of P_i and an edge of P_j. In the two

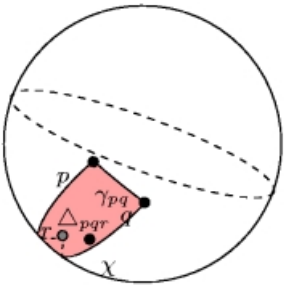


Fig. 2. An arc γ_{pq} and a spherical triangle $\triangle pqr$.

latter cases, the intersection has to satisfy some additional properties, which we describe and exploit below.

Let b_i be the bottom vertex of P_i, and let $V = \{b_i \mid 1 \leq i \leq m\}$. We first report all pairs (i, j) such that the bottom vertex of P_{ij} is the bottom vertex of P_i or of P_j. A vertex $b_i \in V$ is the bottom vertex of P_{ij} if and only $b_i \in P_j$. Therefore, for each $P_j \in \mathcal{P}$, we need to compute and report $P_j \cap V$. As in Section 2.2, we can accomplish this in time $O(m^{3/4}n^{3/4}\log^c n + \mu)$, for some constant c, where $\mu = \sum_{i=1}^{m} |P_j \cap V| \leq k$, using the range-searching algorithm of Matoušek [18].

Next, we report all pairs (i, j) such that the bottom vertex of (the nonempty) P_{ij} is an edge-face intersection. Let E and F denote the sets of edges and of faces, respectively, of the polytopes in $\mathcal{P}$. Using the data structure of Agarwal and Matoušek [2], we can compute, in $O(n^{8/5+\varepsilon})$ time, for any $\varepsilon > 0$, a family of pairs $\mathcal{F} = \{(E_1, F_1), \ldots, (E_r, F_r)\}$, such that

(i) $E_i \subseteq E$ and $F_i \subseteq F$, for all $1 \leqslant i \leqslant r$;
(ii) every edge in E_i crosses every face of F_i, for all $1 \leqslant i \leqslant r$;
(iii) for every crossing edge-face pair $(e, f) \in E \times F$, there is an i so that $e \in E_i$ and $f \in F_i$; and
(iv) $\sum_{i=1}^{u}(|E_i| + |F_i|) = O(n^{8/5+\varepsilon})$.

We will describe an algorithm that, for a given pair (E_i, F_i), computes, in time $O((|E_i| + |F_i|)\log^2 n + \nu_i)$, all ν_i pairs $(e, f) \in E_i \times F_i$ such that $e \cap f$ is the bottom vertex of the corresponding intersection polytope. Repeating this procedure for all pairs of $\mathcal{F}$, we report, in time $O(n^{8/5+\varepsilon} + \nu)$ (for a slightly larger, but still arbitrarily small $\varepsilon > 0$), all ν pairs (i, j) such that the bottom vertex of P_{ij} is the intersection of an edge-face pair.

Consider a pair (E_i, F_i) from the family $\mathcal{F}$. For each edge $e \in E_i$ (resp., each face $f \in F_i$), let $P_e \in \mathcal{P}$ (resp., $P_f \in \mathcal{P}$) be the polytope containing e (resp., f). Let $\mathbb{S}^2$ be the unit sphere of directions in $\mathbb{R}^3$, and let $\chi = (0, 0, -1)$ be the south pole of $\mathbb{S}^2$. For two points $p, q \in \mathbb{S}^2$ that are not antipodal, let $\gamma_{pq} \subset \mathbb{S}^2$ be the shorter arc of the great circle passing through p and q. For three points $p, q, r \in \mathbb{S}^2$ no two of which are antipodal, let $\triangle pqr$ be the smaller spherical triangle formed by the arcs γ_{pq}, γ_{qr}, and γ_{pr}. See Figure 2.

Let $\mathbf{n}_f$ denote the outward unit normal of the face f. For an edge e, let γ_e be the great circular arc representing all outward normals to the planes supporting P_e at e. The endpoints ξ and η of γ_e are the outward normals of the faces of P_e incident upon e, and $\gamma_e = \gamma_{\xi\eta}$. For an edge $e \in E_i$ and a face $f \in F_i$, let $\tau_{ef} = \triangle\xi\eta\mathbf{n}_f$ be the spherical triangle formed by γ_e, $\gamma_{\xi\mathbf{n}_f}$, and $\gamma_{\eta\mathbf{n}_f}$; τ_{ef} is the set of outward normals supporting $P_e \cap P_f$ at the vertex $e \cap f$. The following lemma is straightforward but crucial to our analysis.

Lemma 3. *For a pair $(e, f) \in E_i \times F_i$, the intersection point $e \cap f$ is the bottom vertex of $P_e \cap P_f$ if and only if $\chi \in \tau_{ef}$.*

In order to find the edge-face pairs with the above property, we define a spherical triangle Δ_e for each edge $e \in E_i$ as follows. Let p and q be the antipodal points of the endpoints of γ_e, and let $\overline{\gamma}_e$ be the antipodal arc of γ_e, i.e., the set of points that are antipodal to the points on γ_e. We define $\triangle_e$ to be the spherical triangle $\triangle pq\chi$, which is bounded by the arcs $\overline{\gamma}_e, \gamma_{p\chi}$, and $\gamma_{q\chi}$. We also define W_e to be the spherical wedge that contains the arc $\overline{\gamma}_e$ and is formed by the meridians passing through p and q. Finally, let H_e be the hemisphere containing $\triangle_e$ and bounded by the great circle containing γ_e and $\overline{\gamma}_e$ (this circle is the set of normals to the planes passing through the edge e). Then $\triangle_e = H_e \cap W_e$.

It can be easily checked that $\chi \in \tau_{ef}$ if and only if $\mathbf{n}_f \in \triangle_e$, which implies the following lemma.

Lemma 4. *For a given pair $(e, f) \in E_i \times F_i$, the intersection point $e \cap f$ is the bottom vertex of $P_e \cap P_f$ if and only if $\mathbf{n}_f \in \triangle_e$.*

Let $\Delta = \{\triangle_e \mid e \in E_i\}$ and $N = \{\mathbf{n}_f \mid f \in F_i\}$. For each $\triangle_e \in \Delta$, we wish to report $\triangle_e \cap N$. Recall that $\triangle_e = W_e \cap H_e$. We thus preprocess N into a two-level data structure—the first level reports, for any query $\triangle_e$, all points of $W_e \cap N$ as the union of $O(\log |F_i|)$ canonical subsets, and the second level reports all points of the canonical subsets that lie inside H_e. More precisely, we proceed as follows. We sort the points in N by their longitudes and construct a minimum-height binary tree T on the sorted point set (we omit the easy details concerning the handling of the circularity of this order). Each node u of T is associated with the subset $N_u \subseteq N$ of points that are stored at the leaves of the subtree rooted at u. We preprocess N_u for hemisphere reporting queries, where each query reports all points of N_u lying inside a query hemisphere $H \subset \mathbb{S}^2$. By using a halfplane reporting structure [11], we can preprocess N_u, in $O(|N_u| \log |N_u|)$ time, into a data structure of size $O(|N_u|)$, so that a hemisphere query can be answered in $O(\log |N_u| + t)$ time, where t is the output size. We attach this structure at u as its secondary structure. The total time spent in preprocessing N is $O(|F_i| \log^2 |F_i|)$. For an edge $e \in A$, we report $\triangle_e \cap N$ as follows. By searching with the longitudes of the endpoints of $\overline{\gamma}_e$, we first find, in $O(\log |F_i|)$ time, a set U_e of $O(\log |F_i|)$ nodes of T, so that $\bigcup_{u \in U_e} N_u = W_e \cap N$. For each node $u \in U_e$, we report all t_u points of $N_u \cap \triangle_e$ in $O(\log |F_i| + t_u)$ time, by searching with H_e in the secondary structure attached to u. Therefore the total time spent in reporting all t_e points of $\triangle_e \cap N$ is $O(\log^2 |F_i| + t_e)$. Hence, the overall time spent in

reporting all ν pairs of $E_i \times F_i$ such that $e \cap f$ is the bottom vertex of $P_e \cap P_f$ is $O((|E_i| + |F_i|) \log^2 |F_i| + \nu)$.

Summing up all the bounds, and replacing ε by a slightly larger, but still arbitrarily small constant, we obtain the following.

Theorem 3. *Given a set $\mathcal{P}$ of m polytopes in $\mathbb{R}^3$ with a total of n vertices, we can report all k pairs of indices (i, j) such that P_i and P_j intersect, in time $O(n^{8/5+\varepsilon} + k)$, for any constant $\varepsilon > 0$.*

Remark 3. The above algorithm can also be modified to count, in $O(n^{8/5+\varepsilon})$ time, the number of all intersecting pairs of polytopes in $\mathcal{P}$.

4 Conclusions

In this paper, we presented output-sensitive algorithms for reporting all intersecting pairs of convex polygons / polytopes in two and three dimensions. For the planar case, we presented a near-linear-time algorithm for this problem.

An open question is whether there exists an $o(m^2)$-time algorithm for reporting all pairs of intersecting polytopes in a set $\mathcal{P}$ of m convex polytopes in $\mathbb{R}^4$.

References

[1] P. K. Agarwal, Partitioning arrangements of lines: II. Applications, *Discrete Comput. Geom.*, 5 (1990), 533–573.

[2] P. K. Agarwal and J. Matoušek, On range searching with semialgebraic sets, *Discrete Comput. Geom.*, 11 (1994), 393–418.

[3] P. K. Agarwal and M. Sharir, Red-blue intersection detection algorithms, with applications to motion planning and collision detection, *SIAM J. Comput.*, 19 (1990), 297–321.

[4] P. K. Agarwal and M. Sharir, Ray shooting amidst convex polygons in 2D, *J. Algorithms*, 21 (1996), 508–519.

[5] I. Balaban, An optimal algorithm for finding segment intersections, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 211–219.

[6] J. Basch, L. J. Guibas, and G. Ramkumar, Sweeping lines and line segments with a heap, *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, pp. 469–472.

[7] J. L. Bentley and T. A. Ottmann, Algorithms for reporting and counting geometric intersections, *IEEE Trans. Comput.*, C-28 (1979), 643–647.

[8] G. Brodal and R. Jacob, Dynamic planar convex hull with optimal query time and $o(\log n \cdot \log \log n)$ update time, *Proc. 7th Scand. Workshop Algorithm Theory*, 2000, pp. 57–70.

[9] B. Chazelle, Cutting hyperplanes for divide-and-conquer, *Discrete Comput. Geom.*, 9 (1993), 145–158.

[10] B. Chazelle and L. Guibas, Fractional cascading: I. A data structuring technique, *Algorithmica*, 1 (1986), 133–162.

[11] B. Chazelle, L. Guibas, and D. T. Lee, The power of geometric duality, *BIT*, 25 (1985), 76–90.

[12] B. Chzelle and H. Edelsbrunner, An optimal algorithm for intersecting line segments in the plane, *J. ACM*, 39 (1992), 1–54.
[13] B. Chzelle, H. Edelsbrunner, L. Guibas, and M. Sharir, Algorithms for bichromatic line segment problems and polyhedral terrains, *Algorithmica*, 11 (1994), 116–132.
[14] K. L. Clarkson and P. Shor, Applications of random sampling in computational geometry, II, *Discrete Comput. Geom.*, 4 (1989), 387–421.
[15] U. Finke and K. Hinrichs, Overlaying simply connected planar subdivisions in linear time, *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, 1995, pp. 119–126.
[16] P. Gupta, R. Janardan, and M. Smid, Efficient algorithms for counting and reporting pairwise intersections between convex polygons, *Inform. Process. Lett.*, 69 (1999), 7–13.
[17] S. Har-Peled and M. Sharir, On-line point location in planar arrangements and its applications, *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001, pp. 57–66.
[18] J. Matoušek, Range searching with efficient hierarchical cuttings, *Discrete Comput. Geom.*, 10 (1993), 157–182.
[19] K. Mulmuley, A fast planar partition algorithm, I, *J. Symbolic Comput.*, 10 (1990), 253–280.
[20] N. Sarnak and R. E. Tarjan, Planar point location using persistent search trees, *Commun. ACM*, 29 (1986), 669–679.

Seller-Focused Algorithms for Online Auctioning

Amitabha Bagchi[1], Amitabh Chaudhary[1], Rahul Garg[2], Michael T. Goodrich[1], and Vijay Kumar[3]*

[1] Dept of Computer Science, Johns Hopkins University, 3400 N Charles St, Baltimore MD 21218, USA
[2] IBM India Research Lab, Hauz Khas, New Delhi 110016, India
[3] Amazon.com, 605 5th Avenue South, Seattle, WA 98104, USA

Abstract. In this paper we provide an algorithmic approach to the study of online auctioning. From the perspective of the seller we formalize the auctioning problem as that of designing an algorithmic strategy that fairly maximizes the revenue earned by selling n identical items to bidders who submit bids online. We give a randomized online algorithm that is $O(\log B)$-competitive against an oblivious adversary, where the bid values vary between 1 and B per item. We show that this algorithm is optimal in the worst-case and that it performs significantly better than any worst-case bounds achievable via deterministic strategies. Additionally we present experimental evidence to show that our algorithm outperforms conventional heuristic methods in practice. And finally we explore ways of modifying the conventional model of online algorithms to improve competitiveness of other types of auctioning scenarios while still maintaining fairness.

1 Introduction

Although auctions are among the oldest forms of economic activity known to mankind, there has been a renewed interest in auctioning as the Internet has provided a forum for economic interaction on an unprecedented scale. Indeed, a number of web sites have been created for supporting various kinds of auctioning mechanisms. For example, at www.priceline.com, users present bids on commodity items without knowledge of prior bids, and the presented bids must be immediately accepted or rejected by the seller. Alternately, web sites such as www.ebay.com and www.ubid.com allow bidding on small lots of non-commodity items, with deadlines and exposure of existing bids. The rules for bidding vary considerably, in fact, even in how equal bids for multiple lots are resolved. Interestingly, it is a simple exercise to construct bidding sequences that result in suboptimal profits for the seller. For example, existing rules at www.ubid.com allow a \$100 bid for 10 of 14 items to beat out two \$70 bids for 7 items each. Thus, we feel there could be considerable interest in algorithmic strategies that allow sellers to maximize their profits without compromising fairness.

* A significant portion of this work was done while this author was at IBM's India Research Lab.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 135–147, 2001.

Since Internet auctioning requires a great deal of trust, notions of fairness and legitimacy are fundamental properties for an auction [22]. Indeed, there is considerable previous mathematical treatments of auctioning that classify various types of auctions, together with evolving criteria of fairness (e.g., see [23,19, 9]). Still, the algorithmic aspects of auctioning have been largely neglected.

1.1 The Topic of Interest

Given the interactive nature of Internet auctioning today, we feel it most appropriate to study auctioning strategies from an online algorithms perspective. That is, algorithms must make immediate decisions based on existing, incomplete information, and are not allowed to delay responses to wait for future offers.

Moreover, given that existing auctioning web sites must implement what are essentially algorithmic rules for accepting or rejecting bids, in this paper we focus on algorithmic strategies for sellers. Even so, we restrict our study to strategies that are honest and fair to buyers. For example, we would consider as unacceptable a strategy that uses a fictitious external bidder that causes a real bidder to offer more than he would had there been less bidding competition. Besides being unethical, dishonest or unfair strategies are ultimately detrimental to any Internet auction house anyway, since their discovery drives away bidders.

1.2 Previous Related Work

Offline scenarios for auctioning, where all bids are collected at one time, such as in sealed bid auctions, have been studied and understood in terms of knapsack problems, for which the algorithms community has produced considerable previous work [20,12,13]. We are not aware of much previous work on online auctioning strategies, however.

The general area of online algorithms [7] studies combinatorial optimization problems where the problem instance is presented interactively over time but decisions regarding the solution must be made immediately. Even though such algorithms can never know the full problem instance until the end of the sequence of updates (whose arrival itself might not even be known to the online algorithm), online algorithms are typically compared to optimal offline algorithms. We say that an online algorithm is c-competitive with respect to an optimal offline algorithm if the solution determined by the online algorithm differs from that of the offline algorithm by at most a factor of c in all cases.[1] The goal, therefore, in online algorithm design is to design algorithms that are c-competitive for small values of c. Often, as will be the case in this paper, we can prove worst-case lower bounds on the competitive ratio, c, achievable by an online algorithm. Such proofs typically imply an adversary who constructs input sequences that lead online algorithms to make bad choices. In this paper, we restrict our attention to oblivious adversaries, who can have knowledge of the

[1] As a matter of convention we can never have a c-competitive algorithm for $c < 1$ (such algorithms would instead be called $1/c$-competitive)

online algorithm we are using, but cannot have access to any random bits that it may use.

In work that is somewhat related to online auctioning, Awerbuch, Azar and Plotkin ([3]) study online bandwidth allocation for throughput competitive routing in networks. Their approach can be viewed as a kind of bidding strategy for bandwidth, but differs from our study, since in bandwidth allocation the problem of communication path determination is at least as difficult as that of bandwidth capacity management. Leonardi and Marchetti-Spaccamela ([14]) generalize the result of Awerbuch *et al.*, but in a way that is also not directly applicable to online auctioning, since, again, there is no notion of path determination in online auctioning.

Work for online call control [1,4,15] is also related to the problems we consider. In online call control, bandwidth demands made by phone calls must be immediately accepted or rejected based on their utility and on existing phone line usage. In fact, our work uses an adaptation of an algorithmic design pattern developed by Awerbuch *et al.* [4] and Lipton [15], which Awerbuch *et al.* call "classify-and-select." In applying this pattern to an online problem, one must find a way to partition the optimization space into q classes such that, for each class, one can construct a c-competitive algorithm (all using the same value of c). Combining all of these individual algorithms gives an online algorithm with a competitive ratio that is $O(cq)$. Ideally, the individual c-competitive algorithms should be parameterized versions of the same algorithm, and the values c and q should be as small as possible. Indeed, the classify-and-select pattern is best applied to problems that can be shown to require competitive ratios that are $\Omega(cq)$ in the worst case against an oblivious adversary.

1.3 Our Results

We consider several algorithmic issues regarding online auctioning, from the viewpoint of the seller, in this paper. We begin, in Section 2, by defining the *multiple-item B-bounded online auctioning problem* in which bidders bid on multiple instances of a single item with each bidder allowed to bid for as many items as he or she wants to. We present an online algorithm for this problem that is $O(\log B)$-competitive with an oblivious adversary. The upper bound result presented in this sections is based on adaptations of the classify-and-select design pattern [4] to the specific problem of online auctioning.

In Section 3 we show that it is not possible for any deterministic algorithm to provide a satisfactory competitive ratio for this problem. Moreover, we show that the algorithm we give in Section 2 is "optimal" in the sense that no randomized algorithm can achieve a competitive ratio of $o(\log B)$. To do this we derive lower bounds, based on novel applications of Yao's "randomness shifting" technique [24] , that show the competitive ratios for our algorithm is worst-case optimal.

In order to show that our algorithm performs well in practice we undertook a number of experiments. The results, detailed in Section 4, demonstrate that our algorithm handles different types of input sequences with ease and is vastly

superior to other online strategies in the difficult case where the bids vary greatly in size and benefit.

Finally, in Section 5 we discuss a possible modification of our model with an eye towards making it more flexible as regards its online nature so as to gain in terms of competitiveness. In particular, we allow the algorithm to "buffer" a certain number of bids before making a decision. We show that by buffering only $O(\log B)$ bids it is possible to be c-competitive with an oblivious adversary for the case in which we are selling a single item, for a constant c.

2 Multiple-Item B-Bounded Online Auctioning

In this section we introduce the *multiple-item B-bounded* online auctioning problem. We have n instances of the item on sale and the bids which come in for them offer varying benefit per item. Each bid can request any number of items and offer a given benefit for them. The objective is to maximize the profit that can be earned from the sequence of bids with the additional requirement that the seller accept or reject any given bid before considering any future bids, if they exist.

The *price density* of a bid is defined as the ratio of the benefit offered by the bid to the number of instances of the item that the bid wants to buy. In other words the price density is the average price per item the bidder is willing to pay. The range of possible price densities that can be offered is between 1 and B, inclusively. This restriction is made without loss of generality in any scheme for single-item bidding that has bounded bid magnitude, as we can alternately think of B as the ratio of the highest and lowest bids that can possibly be made on this item. A sequence of bids need not contain the two extreme values, 1 and B, and any bid after the first need not be larger than or equal to the previous bid.

We assume that the algorithm knows the value of B. We discuss at the end of this section how this assumption can be dispensed with.

For this problem we propose an algorithm that uses an adaption of a random choice strategy of Awerbuch and Azar [2] together with the "classify and select" technique of [4], where we break the range of possible optimization values into groups of ranges and select sets of bids which are good in a probabilistic sense based on these ranges. Our algorithm is described in Figure 1.

Theorem 1. Price_And_Pack *is an $O(\log B)$ competitive algorithm for the multiple item B-bounded online auctioning problem.*

The proof of this theorem is in Appendix A. ∎

An important thing to note is that here the algorithm has to know the range of the input i.e. the algorithm has to be aware of the value of B. It is possible to dispense with this assumption to get a slightly weaker result following [4]. In other words, it is possible to give an $O((\log B)^{1+\epsilon})$ competitive algorithm, for any $\epsilon > 0$, which does not know the value of B beforehand. We do not

Algorithm Price_And_Pack

- Select i uniformly at random from the integers 0 to $\log B - 1$.
- If i is 0 then set $pd_r = 1$ else set $pd_r = 2^{i-1}$.
- Define a bid as legitimate if it has a price density of at least pd_r
  - Toss a fair coin with two outcomes before any bid comes in.
  - If the coin has landed heads then wait for a legitimate bid for more than $n/2$ items to come in rejecting all smaller bids and all illegitimate bids.
  - Else keep accepting legitimate bids till there is capacity to satisfy them. Reject all illegitimate bids.

Fig. 1. Price_And_Pack: Auctioning multiple items with bids of varying benefit.

detail it here because it does not provide any further insight into the problem of auctioning.

In the next section we give lower bounds which will show that *Price_And_Pack* gives the best possible competitive ratio for the this problem.

3 Lower Bounds for the Online Auctioning Problem

We consider the version of the online auctioning problem in which there is only one item to be auctioned and the range of possible prices that can be offered for this item is between 1 and B, inclusively. We call this the *single-item B-bounded* online auctioning problem. We give lower bounds for this problem. Upper bounds for this problem are given in [7]. It is easy to see that a lower bound on any algorithm for the single-item problem is a lower bound for the multiple-item problem as well.

In this section we first prove that no deterministic algorithm can in the worst case have a competitive ratio better than the maximum for the single-item problem. More precisely, we show that every deterministic algorithm must have a worst-case competitive ratio that is $\Omega(B)$. This lower bound is based on the fact that a seller does not know in advance how many bids will be offered. Even so, we also show that even if the seller knows in advance the number of bids in the input sequence, any deterministic algorithm is limited to a competitive ratio that is $\Omega(\sqrt{B})$ in the worst case.

Theorem 2. *Any deterministic algorithm for the single-item B-bounded auctioning problem has a competitive ratio that is $\Omega(B)$ in the worst case.*

Proof: For a given deterministic algorithm A we construct an adversarial input sequence I_A in the following way: Let the first bid in I_A be of benefit 1. If A accepts this bid, then I_A is the sequence $\{1, B\}$. In this case, on the sequence I_A, the deterministic algorithm A gets a benefit of 1 unit while the offline optimal algorithm would pick up the second bid thereby earning a benefit of B units.

If A does not accept this first bid, then I_A is simply the sequence $\{1\}$. In this case A earns 0 units of revenue while the optimal offline algorithm would accept the bid of benefit 1. ■

Of course, B is the worst competitive ratio that is possible for this problem, so this theorem implies a rather harsh constraint on deterministic algorithms. Admittedly, the above proof used the fact, perhaps unfairly, that the seller does not know in advance the number of bids that will be received. Nevertheless, as we show in the following theorem, even if the number of bids is known in advance, one cannot perform much better.

Theorem 3. *Any deterministic algorithm for the single-item B-bounded online auctioning problem, where the number of bids is known in advance, has a competitive ratio that is $\Omega(\sqrt{B})$ in the worst case.*

Proof: Consider the input sequence $I_{base} = \{1, 2, 4, \ldots 2^i \ldots B/2, B\}$. For any deterministic algorithm A we construct our adversarial sequence I_A based on what A does with I_{base}.

We recall here that since we are considering the single-item problem, any deterministic algorithm essentially picks at most one of the bids in the input sequence.

Suppose A accepts some bid $2^i \leq \sqrt{B}$. Then we choose I_A to be the same as I_{base}. In this case A's benefit is less than $\sqrt{B}$, whereas an optimal offline algorithm would earn B units thereby making A an $\Omega(\sqrt{B})$ competitive algorithm.

If A accepts some bid $2^i > \sqrt{B}$, on the other hand, then we choose I_A to be $\{1, 2, 4, \ldots 2^{i-1}, 1, 1, \ldots\}$, i.e., we stop increasing the sequence just before A accepts and then pad the rest of the sequence with bids of benefit 1. [2] This way A can get no more than 1 unit of benefit while the optimal offline algorithm gets 2^{i-1} which we know is at least $\sqrt{B}$.

If A accepts none of the bids in I_{base} then it is not a competitive algorithm at all (i.e. it earns 0 revenue while the optimal offline algorithm earns B units) and so we need not worry about it at all. ■

It is easy to see that the deterministic algorithm that either picks up a bid of benefit at least $\sqrt{B}$ or, if it does not find such a bid, picks up the last bid, whatever it may be, succeeds in achieving a competitive ratio of $O(\sqrt{B})$.

Theorem 2 tells us that no deterministic algorithm can effectively compete with an oblivious adversary in the worst case, if the number of bids is not known in advance. Indeed, although the proof used a sequence that consisted of either one bid or two, the proof can easily be extended to any sequence that is either of length n or $n+1$. This bleak outlook for deterministic algorithm is not improved much by knowing the number of bids to expect, however, as shown in Theorem 3.

Furthermore we show that even randomization does not help us too much. We can use Yao's principle [24] to show that no randomized algorithm can be more competitive against an oblivious adversary than *Sell_One*.

Theorem 4. *Any randomized algorithm for the single-item B-bounded online auctioning problem is $\Omega(\log B)$-competitive in the worst case.*

The proof is sketched in Appendix B. ■

[2] Bids are not always increasing. For a single item there is no issue at all if the bids are always increasing and the number of bids is known. Just wait for the last bid.

4 Experimental Results

In order to give an idea of the efficacy of *Price_And_Pack* we present the results of simulated auctions which use this algorithm.

The input sequences were generated by selecting each bid from a given probability distribution. The three distributions used were: Normal, Poisson and Uniform. Both the number of items being bid for and the price density offered by the bid were chosen from the same distribution.

We chose three different combinations of n and B and generated 100 input sequences for each combination. To get a good approximation to the average benefit of *Price_And_Pack* we ran the algorithm 1000 times on each instance and averaged the benefit over all these runs.

We determined a lower bound on the amount of revenue obtained by our algorithm compared to the maximum possible revenue. To do this we implemented an offline algorithm which has been shown to be a 2 approximation [11]. By dividing the revenue obtained by *Price_And_Pack* by 2 times the revenue obtained by the offline algorithm we were able to provide a number which is effectively a lower bound on the actual ratio.

(n, B)	Distribution	Expected ratio $(1/\log B)$	Ratio
(50, 1024)	Uniform	.1	.31
	Normal		.69
	Poisson		.61
(2000, 1024)	Uniform	.1	.34
	Normal		.62
	Poisson		.7
(2000, 2048)	Uniform	.09	.34
	Normal		.61
	Poisson		.69

Fig. 2. Price_And_Pack v/s the optimal offline algorithm.

The numbers in Figure 2 show that in practice *Price_And_Pack* performs quite well compared to the optimal offline algorithm and significantly better than the bound of $O(\log B)$ would suggest. We see that in the two distributions which tend to cluster sample points near the mean, i.e. Normal and Poisson, the algorithm does especially well. However these distributions provide fairly regular input instances. The real power of *Price_And_Pack* is on view when the input instances have widely varying bids.

To demonstrate this we compared the performance of a simple *Greedy* heuristic with the performance of *Price_And_Pack*. *Greedy* simply accepts bids while it has the capacity to do so. In Figure 3 we present the results in terms of the percentage extra revenue *Price_And_Pack* is able to earn over the *Greedy* heuristic.

Distribution	(n, B)	Average %
Uniform	(50, 1024)	25
	(2000, 1024)	28.5
	(2000, 2048)	27.1
Normal	(50, 1024)	0.5
	(2000, 1024)	0.7
	(2000, 2048)	0.5
Poisson	(50, 1024)	1.4
	(2000, 1024)	0.1
	(2000, 2048)	0.3

Fig. 3. Price_And_Pack over **Greedy**: % advantage.

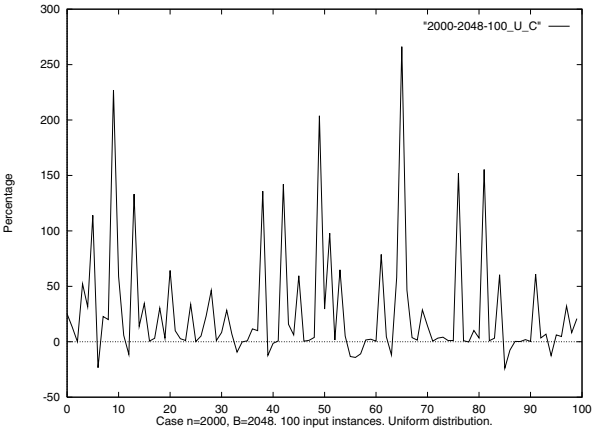


Fig. 4. Price_And_Pack over **Greedy**: % advantage in 100 individual runs.

We see that when the bids are comparable to each other (i.e. when they are generated by the Normal or Poisson distribution) then *Price_And_Pack* does not do significantly better than *Greedy* but when the bids vary widely in size (i.e. when they are generated by the Poisson distribution) then *Price_And_Pack* definitely outperforms *Greedy*.

In Figure 4 we graph the percentage extra revenue earned by *Price_And_Pack* in 100 different input instances for a given choice of n and B. It is clear from the graph that *Price_And_Pack* consistently outperforms *Greedy*.

5 Improving Competitiveness by Giving Intermediate Information

In this section we look at a way of modifying the auctioning model to improve competitiveness. Taxonomies of auctions (for eg. [23], [19]) have classified auctions along three broad categories: bid types, clearance mechanisms and intermediate information policies. We look at this lattermost classification to help us improve competitiveness.

In the preceding sections we saw that in the conventional online model, where every bid has to be accepted or rejected before the next bid comes in, we are limited to a competitive ratio of $\Omega(\log B)$. However it is possible to do better if we relax the online model slightly. In the model under consideration so far every bid has to be accepted or rejected immediately, or, more precisely, before the next bid comes in. However in real life auctions this is not always the case. Most auctions do release some intermediate information. For example in the outcry type of auction the current highest bid is announced. This amounts to informing those who bid at that level that their bid is still under consideration, although it might yet be beaten out by a better bid.

The problem with the outcry auction is that, in the case of a monotonically increasing sequence of bids, each bid is asked to hold on and then rejected i.e. $O(B)$ bids are made to wait till a new bid comes in before being rejected. This is clearly unacceptable since from the point of view of a bidder an intermediate notification that the bid is still under consideration is tantamount to saying that this bid has a reasonable chance of success. However if the bidder knows that $O(B)$ bids could be asked to hold then he might not consider this chance of success reasonable.

So, the model we propose is that only a certain small number of bids can be asked to wait without a definite notification of acceptance or rejection. We can think of these bids being buffered in a buffer which will need to contain only one item at a time and will not be allowed to hold more than a certain small number of items in the course of the auction. We call this structure a *k-limited access buffer* or a k-LAB. We denote the highest bid held in the LAB by $H(LAB)$.

That this relaxation is useful becomes immediately evident when we consider that a $\log B$-LAB allows us to become constant competitive deterministically with the optimal algorithm in the case where we want to sell one item and we know the number of bids. We give the algorithm for this in Figure 5. We have to view this in light of the fact that in Theorem 3 we showed that in a purely online setting it is not possible to do better than $\Omega(\sqrt{B})$ deterministically for this problem.

Algorithm LAB_Sell_One_N

- $LAB \leftarrow b_0$
- For each bid b_i for i going from 1 to N do
  - if $b_i > 2.H(LAB)$ then $LAB \leftarrow b_i$ else reject b_i
- Accept the bid in the LAB.

Fig. 5. LAB_Sell_One_N: Auctioning a single item with a buffer when the number of bids is known.

Theorem 5. LAB_Sell_One_N *is* $\frac{1}{2}$ *competitive with the optimal.*

Proof: It is quite easy to see that the highest bid b_{off} which is the benefit of the offline algorithm would not be put in the buffer only if a bid which was at

least half its benefit were already in there. Since a bid of at least half its benefit is already in the online algorithm's buffer therefore its benefit will be at least half of the online's. ■

Acknowledgements. The authors would like to thank Leslie Hall and two anonymous referees.

References

1. R. Adler and Y. Azar. Beating the logarithmic lower bound: randomized preemptive disjoint paths and call control algorithms. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1–10, 1999.
2. B. Awerbuch and Y. Azar. Blindly competitive algorithms for pricing and bidding. Manuscript.
3. B. Awerbuch, Y Azar, and S. Plotkin. Throughput competitive on-line routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 32–40. IEEE, 3–5 November 1993.
4. B. Awerbuch, Y. Bartal, A. Fiat, and A. Rosen. Competitive non-preemptive call-control. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, 23-25 Jan 1994.
5. A. Bagchi, A. Chaudhary, R. Garg, M. T. Goodrich, and V. Kumar. Online algorithms for auctioning problems. Technical report, IBM India Research Lab, 2000.
6. S. Ben-David and A. Borodin. A new measure for the study of on-line algorithms. *Algorithmica*, 11:73–91, 1994.
7. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.
8. D. Dooly, S. Goldman, and S. Scott. Tcp dynamic acknowledgment delay: Theory and practice. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing,*, pages 289–298, May 1998.
9. R. Engelbrecht-Wiggans, M. Shubik, and R. M. Stark, editors. *Auctions, Bidding, and Contracting: Uses and Theory.* New York University Press, 1983.
10. E. F. Grove. Online bin packing with lookahead. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 430–436, 1995.
11. Dorit S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, Boston, MA., 1997.
12. O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and subset sum problems. *Journal of the ACM*, 22:463–468, 1975.
13. E. L. Lawler. Fast approximation algorithms for knapsack problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 206–213, 1977.
14. S. Leonardi and A. Marchetti-Spaccamela. On-line resource management with applications to routing and scheduling. In *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, pages 303–314, 1995.
15. R. J. Lipton and A. Tomkins. Online interval scheduling. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 302–11, 23-25 Jan 1994.
16. G. Lueker. Average-case analysis of off-line and on-line knapsack problems. *Journal of Algorithms*, 29(2):277–305, November 1998.

17. A. Marchetti-Spaccamela and C. Vercellis. Stochastic on-line knapsack problems. *Mathematical Programming*, 68(1):73–104, Jan 1995.
18. R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
19. M. H. Rothkopf, A Pekec, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1998.
20. S. Sahni. Approximation algorithms for the 0/1 knapsack problem. *Journal of the ACM*, 22:115–124, 1975.
21. T. Sandholm. An algorithm for optimal winner determination in combinational auctions. Technical Report WUCS-99-01, Department of Computer Science, Washington University, January 1999.
22. Charles W. Smith. *Auctions: The Social Construction of Value*. The Free Press, 1989.
23. P. R. Wurman, M. P. Wellman, and W. E. Walsh. A parametrization of the auction design space. *Games and Economic Behaviour*, To appear.
24. A. C-C. Yao. Probabilistic computations: Towards a unified measure of complexity. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 222–227, 1977.
25. T. Yeh, C. Kuo, C. Lei, and H. Yen. Competitive analysis of on-line disk scheduling. *Theory of Computing Systems*, 31:491–506, 1998.

A Proof of Theorem 1

Let the optimal offline algorithm OPT achieve profit density p on a given input sequence I. So if the optimal algorithm sells $n' \leq n$ items, its total profit is $n'p$. Let j be the largest integer such that $2^j \leq 4p/5$. Define $\alpha = \frac{2^j}{p}$. We say that *Price_And_Pack* chooses i *correctly*, if the chosen value of i equals j. It is easy to see that i is chosen correctly with probability $1/\log B$. In that event, bids of price density greater than $p\alpha$ are legitimate while the rest are not. Note that $\alpha \in (2/5, 4/5]$.

Let I_p be a subset of I, comprising all bids in I which have price density greater than $p\alpha$.

Lemma 1. *The sum of the revenues obtained by the optimal algorithm running on I_p is no less than $n'p(1-\alpha)$ where p is the profit density of OPT on I and n' is the number of items it sells.*

Proof: Suppose that OPT sells some $n_{lt} \leq n'$ instances to bids in $I - I_p$, and let rev_{ge} be the revenue earned by OPT from items which were sold to bids in I_p. Clearly,

$$rev_{ge} + n_{lt}.p\alpha \geq n'p$$

this gives us

$$rev_{ge} \geq n'p - n_{lt}.p\alpha$$

and since $n_{lt} \leq n'$ we get

$$rev_{ge} \geq n'p(1-\alpha)$$

Since rev_{ge} is the revenue obtained from a subset of the bids in I_p, the result follows. ■

Proof of Theorem 1:

We consider the following three cases, and show that in each case the expected revenue of *Price_And_Pack* is at least $\frac{np}{10 \log B}$.

Case 1: There is a bid of size greater than $n/2$ in I_p.

With probability at least $1/\log B$, *Price_And_Pack* chooses i correctly. With probability $1/2$ *Price_And_Pack* chooses to wait for a bid of size greater than size $n/2$. Thus, with probability at least $\frac{1}{2 \log B}$, *Price_And_Pack* will accept a bid of size at least $n/2$ and price density at least αp.

So in this case the expected revenue of *Price_And_Pack* is at least $\frac{np\alpha}{4 \log B}$. Since the revenue earned by OPT is np, and $\alpha > 2/5$, in this case *Price_And_Pack* is $10 \log B$ competitive with OPT.

Case 2: There is no bid of size greater than $n/2$ in I_p, and the total number of items demanded by the bids in I_p is more than $n/2$.

With probability $1/2$ *Price_And_Pack* will choose to accept bids of any size. If it also chooses i correctly (the probability of which is $1/\log B$), it will sell at least $n/2$ instances [3], and earn a revenue of at least $p\alpha$ units for every item sold.

Thus, with probability $1/2 \log B$, *Price_And_Pack* sells at least $n/2$ instances to bids whose price densities are no smaller than $p\alpha$. This means that, in this case, the expected revenue of *Price_And_Pack* is at least $\frac{np\alpha}{4 \log B} > \frac{np}{10 \log B}$, which makes it $10 \log B$ competitive with OPT.

Case 3: There is no bid of size greater than $n/2$ in I_p, and taken together the bids in I_p demand no more than $n/2$ instances.

Again, with probability $1/2$ *Price_And_Pack* decides to accept all bids, and with probability $1/\log B$, i is chosen correctly. Thus, with probability $1/2 \log B$ our algorithm accepts all bids in I_p, and, by Lemma 1, earns a revenue no smaller than $n'p(1-\alpha)$ where n' is the number of items sold by OPT. So its expected revenue is at least $\frac{n'p(1-\alpha)}{2 \log B} \geq \frac{n'p}{10 \log B}$, which makes it $10 \log B$ competitive with OPT in this case. ∎

B Proof Sketch for Theorem 4

We use Yao's Principle [24,18] to show a lower bound for all randomized algorithms. To do this we give a probability distribution over the input space and determine the expected benefit of the best possible deterministic algorithm on this probabilistic input. The competitiveness of this expectation against the expected benefit of the optimal offline algorithm for this probabilistically distributed input will be, by Yao's Principle, a lower bound on the competitiveness of any randomized algorithm for this problem. Due to lack of space we simply describe the probability distribution on the inputs here. The entire proof is available in [5].

[3] If *Price_And_Pack* accepts all bids in I_p, it sells at least $n/2$ instances. If it rejects any bid in I_p, it must not have enough capacity left to satisfy it. But then at least $n/2$ instances must have been sold, since any bid in I_p — in particular the rejected bid — is of size no more than $n/2$.

Consider the following input sequence which we will be calling the base sequence or I_{base}: $\{1, 2, 4, \ldots B/2, B\}$. Our set of input sequences will be derived from I_{base} by truncating it at a given point and substituting bids of revenue 1 for the tail of the input sequence. All other inputs occur with probability 0. The set of inputs, $\mathcal{I} = \{I_1, I_2 \ldots I_{\log B}\} \cup \{I_f\}$ and associated probabilities are described as:

- $I_i = \{1, 2, 4, \ldots 2^i, 1 \ldots 1\}$ occurs with probability $P_i = \frac{1}{2^{i+1}}$. Each I_i has $\log B + 1$ bids.
- $I_f = \{1\}$ occurs with probability $P_f = \frac{B+1}{2B}$.

■

Competitive Analysis of the LRFU Paging Algorithm

Edith Cohen[1], Haim Kaplan[2], and Uri Zwick[2]

[1] AT&T Labs–Research, 180 Park Avenue, Florham Park, NJ 07932 USA.
edith@research.att.com
[2] School of computer science, Tel Aviv University, Tel Aviv, Israel.
{haimk,zwick}@math.tau.ac.il

Abstract. We present a competitive analysis of the LRFU paging algorithm, a hybrid of the LRU (Least Recently Used) and LFU (Least Frequently Used) paging algorithms. We show that the competitive ratio of LRFU is $k + \left\lceil \frac{\log(1-\lambda)}{\log \lambda} \right\rceil - 1$, where $\frac{1}{2} \leq \lambda \leq 1$ is the decay parameter used by the LRFU algorithm, and k is the size of the cache. This supplies, in particular, the first natural paging algorithms that are competitive but are not optimally competitive, answering a question of Borodin and El-Yaniv. Although LRFU, as it turns out, is not optimally competitive, it is expected to behave well in practice, especially in web applications, as it combines the benefits of both LRU and LFU.

1 Introduction

Paging (cache replacement) algorithms had been extensively studied and deployed. Two important applications are operating systems virtual memory paging and caching of Web content. The input to a paging algorithm is a sequence of requests to different *pages* and the size of the *cache*. The algorithm decides which page to evict when a request arrives for a page which is not presently in the cache and the cache is full. Often, the choice of a paging algorithm can have a significant effect on performance.

Two important paging algorithms are *Least Recently Used* (LRU) and *Least Frequently Used* (LFU). LRU is arguably the most commonly deployed in practice. One such example is the popular Squid [7] Web caching software. When LRU has to evict a page from its cache, it chooses to evict the least-recently requested page. LRU exploits temporal locality in request sequences and the *recency* property which states that recently-requested objects are more likely to be requested next. LFU is another policy that seems to perform well on Web request sequences. LFU evicts the page that was requested the fewest number of times. LFU comes in two flavors: *in-cache* LFU which counts the number of times a page was requested since it entered the cache, and *perfect* LFU which counts the total number of requests made to the page since the start of the sequence. LFU exploits the *frequency* property of request sequences which states that pages that were requested more times are more likely to be requested next.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 148–154, 2001.

Recent studies (e.g. [2]) comparing cache replacement algorithms concluded that LRU and LFU exhibit comparable performance on Web request sequences. In general, performance depends on the interplay between the recency and frequency properties of the particular sequence. Frequency seems to matter more for larger caches and recency has a more pronounced impact with smaller cache sizes. Similar behavior was observed for page request sequences in operating systems [6]. Motivated by these observations, Lee et al. [6] defined a spectrum of hybrid policies between LRU and LFU. They named these policies LRFU_λ, with the parameter λ varying between $\lambda = 0$ (LRU) and $\lambda = 1$ (LFU). Their experiments based on simulations using page requests generated by different application programs demonstrated that the performance curve as a function of λ is smooth and the dependence on λ is bitonic: the miss-rate first decreases and then increases. Thus typically LRFU_λ outperforms at least one of the endpoints and with the best choices of λ, LRFU_λ often outperforms both LRU and LFU. Their results show that LRFU_λ does indeed provide a desirable spectrum in terms of performance.

Competitive analysis had been the leading theoretical tool for analyzing the performance of paging algorithms [1]. A paging algorithm A is said to have a *competitive ratio* of at most c if on any request sequence, the number of *misses* of A is at most c times the number of misses of the optimal offline algorithm, plus some constant. A *miss* occurs when a requested page is not in the cache. The competitive ratio of LRU and LFU demonstrates both strengths and shortcoming of the ability of competitive analysis to capture actual behavior. The LRU algorithm is optimally competitive. That is, its competitive ratio of k is the best possible by any deterministic paging algorithm (k is the maximum number of pages that can fit in the cache). On the other hand, the factor k obtained on worst-case adversarial sequences is very far from the typical ratio on actual sequences. Moreover, LFU, which performs comparably to LRU on Web sequences has an unbounded competitive ratio (is not competitive). Actual sequences, as opposed to worst-case sequences, typically exhibit the recency and frequency properties exploited by LRU and LFU.

Interestingly, most natural deterministic paging algorithms fall in one of these two categories, either they are optimally competitive like LRU, or they are not competitive like LFU. An open question posed by Borodin and El-Yaniv [1] (open question 3.2 on page 43) is the existence of a natural deterministic paging algorithm with a finite but not optimal competitive ratio. It seems that most conceivable hybrids of a non-competitive algorithm with an optimally-competitive algorithms (such as partition the cache between the two policies) are not competitive.

Our main contribution here is a tight competitive analysis of LRFU_λ. Our analysis reveals that as λ increases we obtain all integral values between the competitive ratio of LRU (k) and that of LFU (∞). This solves the open problem posed by Borodin and El-Yaniv. The full spectrum of values also suggests that LRFU_λ is the "right" hybrid of LRU and LFU. In this sense the competitive analysis supports and complements the experimental results.

2 The LRFU$_\lambda$ Paging Algorithm

The LRFU$_\lambda$ paging policy, for $0 < \lambda \leq 1$, is defined as follows:

1. Each page p currently in the cache has a value $v(p)$ associated with it.
2. Upon a request for a page q, the following operations are performed:
   a) If a page has to be evicted to make room for q, then the page with the smallest value is evicted. (Ties are broken arbitrarily.) The new page q is temporarily given the value $v(q) \leftarrow 0$.
   b) The values of the pages in the cache are updated as follows:

$$v(p) \leftarrow \begin{cases} \lambda v(p) & \text{if } p \neq q, \\ 1 + \lambda v(p) & \text{if } p = q. \end{cases}$$

It is easy to see that if a page p is in the cache at time t and if the hits to this page, since it last entered the cache, were at times $t - j_1, t - j_2, \ldots, t - j_n$, then its value at time t is $v(p) = \sum_{i=1}^{n} \lambda^{j_i}$. Note that the value $v(p)$ of a page is always *smaller* than $\sum_{j=0}^{\infty} \lambda^j = \frac{1}{1-\lambda}$.

In particular, we get that if $0 < \lambda \leq \frac{1}{2}$, then LRFU$_\lambda$ behaves just like LRU. And, if $\lambda = 1$, then LRFU$_\lambda$ behaves just like LFU. It is well known that LRU has an optimal competitive ratio of k, while LFU is not competitive. We show that as λ increases from $\frac{1}{2}$ to 1, the competitive ratio of LRFU$_\lambda$ increases from k to ∞ and assumes all possible integral values in this range.

3 Competitiveness of LRFU$_\lambda$

We obtain the following result:

Theorem 1. *The competitive ratio of* LRFU$_\lambda$ *for a cache of size k, where $\frac{1}{2} \leq \lambda < 1$, is*

$$k + \left\lceil \frac{\log(1-\lambda)}{\log \lambda} \right\rceil - 1 \, .$$

Another way of stating this result is as follows: the competitive ratio of LRFU$_\lambda$ is $k + \ell$, where $\ell = \left\lceil \frac{\log(1-\lambda)}{\log \lambda} \right\rceil - 1$. Note that $\frac{\lambda^\ell}{1-\lambda} > 1$ while $\frac{\lambda^{\ell+1}}{1-\lambda} \leq 1$. Thus, ℓ has the following property: if p was not referenced in the last $\ell + 1$ time units, then $v(p) < 1$, and ℓ is the smallest number with this property. Theorem 1 follows from Lemmas 1 and 3 that we obtain next.

3.1 Upper Bound on the Competitive Ratio

Lemma 1. *For every $\frac{1}{2} \leq \lambda < 1$,* LRFU$_\lambda$ *is $(k + \ell)$-competitive, where $\ell = \left\lceil \frac{\log(1-\lambda)}{\log \lambda} \right\rceil - 1$.*

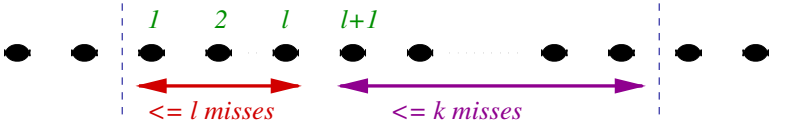


Fig. 1. LRFU_λ incurs at most $k + \ell$ misses per k-phase.

Proof. As in the classical analysis of LRU (see [1, Chapter 3), we break the sequence of requests into *k-phases*. A k-phase is a maximal contiguous block of requests in which at most k different pages are requested. The first k-phase starts at the beginning of the sequence and ends just before the $(k+1)$-st distinct page is referenced. The second block begins immediately after the first block and so on. It is easy to see that any caching policy must incur, on average, at least one miss for each k-phase. (More precisely, any caching policy must incur at least one miss on each *shifted k-phase*, where a shifted k-phase is a sequence of requests that begins with the second request of an ordinary k-phase and ends with the first request of the next ordinary k-phase.) To show that LRFU_λ is $(k + \ell)$-competitive, we show that LRFU_λ incurs at most $k + \ell$ misses on each k-phase.

We first claim that under the LRFU_λ policy, a page p that enters the cache *after* the first ℓ requests of a k-phase, stays in the cache until the end of the phase. This follows from the interpretation of ℓ given after the statement of Theorem 1. When p enters the cache, and assigned the value 1, the value $v(q)$ of a page q that is currently in the cache, but was not referenced yet in the current k-phase, must satisfy $v(q) < 1$. If a miss occurs in the k-phase after p enters the cache, then there is at least one page q in the cache that was not referenced yet in the k-phase. (If all k pages in the cache were referenced in the current phase then a subsequent miss would end the phase.) As $v(q) < v(p)$, p will not be evicted.

It follows, therefore, that at most ℓ misses occur at the first ℓ requests of each k-phase (at most one miss per request), and at most k misses occur at the rest of the k-phase (at most one miss per each page that enters the cache after the ℓ-th request). (See Figure 1.) This completes the proof of the lemma.

Note that if $\lambda = \frac{1}{2}$ then $\ell = 0$ (i.e., $v(p) \geq 1$ if and only if p is the last page referenced), and Lemma 1 states that the competitive ratio of $\mathrm{LRFU}_{1/2} = \mathrm{LRU}$ is k. Lemma 1 is therefore an extension of the classical competitive analysis of the LRU policy. An easy extension of Lemma 1 is the following:

Lemma 2. *For every* $\frac{1}{2} \leq \lambda < 1$ *and* $1 \leq h \leq k$, LRFU_λ *with a cache of size* k *is* $(k + \ell)/(k - h + 1)$*-competitive, where* $\ell = \left\lceil \frac{\log(1-\lambda)}{\log \lambda} \right\rceil - 1$, *versus an offline adversary that uses a cache of size* h.

Proof. Follows from the fact any algorithm that uses a cache of size h must incur at least $k - h + 1$ misses on each shifted k-phase.

3.2 Lower Bound on the Competitive Ratio

We now show that the bound given in Lemma 1 is tight.

Lemma 3. *For every* $\frac{1}{2} \leq \lambda < 1$, LRFU_λ *is* at most $(k+\ell)$*-competitive, where* $\ell = \left\lceil \frac{\log(1-\lambda)}{\log \lambda} \right\rceil - 1$.

Proof. For every $k \geq 1$ and $\frac{1}{2} \leq \lambda < 1$, we construct an infinite sequence of requests such that the ratio between the number of misses incurred by LRFU_λ and the minimum possible number of misses needed to serve the sequence approaches $k + \ell$. As in the classical case of LRU, only $k + 1$ distinct pages are needed for this purpose.

Our sequences are again partitioned into k-phases. The state of the cache at the end of each k-phase is isomorphic to the state of the cache at the beginning of the phase. (This is explained below in more detail.) We show that LRFU_λ incurs $k + \ell$ misses in each phase. As there is a total of only $k + 1$ pages, the sequence of requests can be served with only one miss per k-phase. (At the first miss of a phase, the optimal algorithm evicts the page that is not going to be referenced in that phase.) The claim of the lemma would thus follow.

Let L (for Large) be a sufficiently large number such that after L consecutive requests to a page p we have $v(p) > \lambda^{-\ell}$ and $v(q) < 1$ for any page q such that $q \neq p$. Such a number exists as by the definition of ℓ we have $\lambda^{-\ell} < \frac{1}{1-\lambda}$.

The first initializing k-phase is composed of the following page requests:

$$p_2, p_3, p_4, \ldots, p_{k+1}^L \ ,$$

where p_i^L stands for L consecutive requests of the page p_i. At the end of this phase, the pages in the cache, in decreasing order of their values, are $p_{k+1}, p_k, \ldots, p_2$, and their values satisfy $v(p_{k+1}) > \lambda^{-\ell}$ and $v(p_i) < 1$, for $2 \leq i \leq k$.

The second phase in now composed of the following page requests:

$$p_{(1)}, p_{(2)}, \ldots, p_{(k+\ell)}^L \ ,$$

where the parentheses around the indices indicate that they should be interpreted modulo k (i.e., $p_{(k+1)} = p_1$, and more generally $p_{(i)} = p_{1+(i-1) \bmod k}$.

What happens to the cache when this sequence of requests is served? As in the beginning of the phase $\lambda^{-\ell} < v(p_{k+1}) < \frac{1}{1-\lambda} \leq \lambda^{-(\ell+1)}$, (the last inequality follows from the definition of ℓ), we get that during the first ℓ requests of the phase, page p_{k+1} still has the largest value among all the pages of the cache, and the page entering the cache becomes the page with the *second* largest value. The page evicted is always the next page requested, and thus each request causes a miss.

When the $(\ell+1)$-st request of the phase is served, the value of $v(p_{k+1})$ drops below 1. The page entering the cache now becomes the page with the largest weight. The rank of p_{k+1} in the cache decreases by 1 at each step. The page

Requested page	Cache content by decreasing value								
	$k+1$	k	$k-1$				4	3	2
1	$k+1$	1	k	$k-1$			5	4	3
2	$k+1$	2	1	k	$k-1$			5	4
					$\cdots$				
					$\cdots$				
ℓ	$k+1$	ℓ	$\ell-1$		k	$k-1$		$\ell+3$	$\ell+2$
$\ell+1$	$\ell+1$	$k+1$	ℓ	$\ell-1$		k	$k-1$		$\ell+3$
					$\cdots$				
					$\cdots$				
k	k	$k-1$		$\ell+1$	$k+1$	ℓ	$\ell-1$		2
1	1	k	$k-1$		$\ell+1$	$k+1$	ℓ		3
2	2	1	k	$k-1$		$\ell+1$	$k+1$		4
					$\cdots$				
					$\cdots$				
ℓ	ℓ	$\ell-1$		2	1	k	$k-1$		$\ell+1$

Fig. 2. A typical phase of the worst-case sequence for LRFU_λ

evicted is again the next page requested and thus each request still causes a miss. While serving the request for $p_{(k+\ell)}$, page p_{k+1} is finally evicted from the cache.

Overall, there are $k+\ell$ misses in this phase. The content of the cache during such a phase is depicted in Figure 2. (In the figure, it is assumed, for simplicity, that $\ell < k$. The argument given above does not rely on this assumption.)

Now, the state of cache at the end of the second phase is similar to its state at the end of the first phase. One page, namely $p_{(k+\ell)}$, has value larger than $\lambda^{-\ell}$, while all other pages have values that are less than 1. We can therefore use a sequence of requests similar to the one used in the second state, and again create a k-phase in which LRFU_λ incurs $k+\ell$ misses. This process can be repeated over and over again, resulting with the promised infinite sequence of requests.

4 Open Problems

Web objects may vary significantly in both size and cost of a miss. This motivated the development of cache replacement algorithms that account for varying page sizes and fetching costs [5,8,3,4]. Experiments showed that the optimally-competitive Landlord algorithm performs well on Web caching sequences [8,3]. Some experiments, however, show that it can still be outperformed by perfect LFU [2]. Thus, it would be interesting to extend our results to this more general situation and obtain natural hybrid policies.

References

1. A. Borodin and R. El Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.
2. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.
3. P. Cao and S. Irani. Cost-Aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
   `http://www.usenix.org/events/usits97`.
4. E. Cohen and H. Kaplan. LP-based analysis of greedy-dual-size. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1999.
5. S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proc. 29th Annual ACM Symposium on Theory of Computing*. ACM, 1997.
6. D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. In *Proceedings of the ACM SIGMETRICS'99 Conference*, 1999.
7. Squid internet object cache.
   `http://squid.nlanr.net/Squid`.
8. N. Young. On line file caching. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*. ACM-SIAM, 1998.

Admission Control to Minimize Rejections

Avrim Blum[1], Adam Kalai[2], and Jon Kleinberg[3]

[1] Carnegie Mellon University, Pittsburgh PA 15213,
avrim@cs.cmu.edu
[2] Carnegie Mellon University, Pittsburgh PA 15213,
akalai@cs.cmu.edu
[3] Cornell University, Ithaca NY 14853,
kleinber@cs.cornell.edu

Abstract. Admission control (call control) is a well-studied online problem. We are given a fixed graph with edge capacities, and must process a sequence of calls that arrive over time, accepting some and rejecting others in order to stay within capacity limitations of the network. In the standard theoretical formulation, this problem is analyzed as a benefit problem: the goal is to devise an online algorithm that accepts at least a reasonable fraction of the maximum number of calls that could possibly have been accepted in hindsight. This formulation, however, has the property that even algorithms with optimal competitive ratios (typically $O(\log n)$) may end up rejecting the vast majority of calls even when it would have been possible in hindsight to reject only very few.
In this paper, we instead consider the goal of approximately *minimizing* the number of calls *rejected*. This is much more natural for real-world settings in which rejections are intended to be rare events. In order to avoid trivial lower-bounds, we assume *preemption* is allowed and that calls are given to the algorithm as fixed paths. We show that in a number of cases, we can in fact achieve a competitive ratio of 2 for rejections (so if the optimal in hindsight rejects 0 then we reject 0; if the optimal rejects r then we reject at most $2r$). For other cases we get worse but nontrivial bounds. For the most general case of fixed paths in arbitrary graphs with arbitrary edge capacities, we achieve matching $\Theta(\sqrt{m})$ upper and lower bounds. We also show a connection between these problems and online versions of the vertex-cover and set-cover problems (our factor-2 results give 2-approximations to slight generalizations of the vertex cover problem, much as [AAA99] show hardness results for the benefit version based on the hardness of approximability of independent set).

1 Introduction

In the well-studied *admission control* (or *call control*) problem, our job is to manage a network G (a graph with edge capacities) in the presence of online requests for communication (calls). Requests for communication may be accepted or rejected, and the goal of an online algorithm is to accept as many as possible while staying within the edge capacities of the network.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 155–164, 2001.

This problem has typically been studied as a *benefit problem*. That is, one compares the number of calls that could have been accepted in hindsight to the number actually accepted by the online algorithm, and tries to minimize this ratio. A number of papers have produced good bounds for this metric, such as the work of Awerbuch, Azar and Plotkin [AAP93] for the high-capacity setting, and Awerbuch et al. [AGLR94] for trees and other specific networks. A serious problem with viewing call-control as a benefit problem, however, is that even with, say, an $O(\log n)$ competitive ratio that would normally be considered quite good, it is possible that the algorithm may route only a $1/(\log n)$ fraction of the calls even if a solution routing nearly all of them is possible.[1] For many of the natural applications of admission control, even a modest constant fraction of rejections would be deemed unacceptable performance. Thus, for these types of applications, the benefit formulation appears fundamentally flawed.

In this paper we depart from the benefit metric and instead set our sights on the goal of *minimizing* the number of calls *rejected*. That is, if OPT (the optimal strategy in hindsight) rejects 0 then we should reject 0. If OPT rejects a small number, then we should reject only a small multiple of that. What we show is that for several natural cases, we can in fact achieve a competitive ratio of 2 for rejections. For other versions we can achieve worse but still non-trivial bounds. Of course, approximately minimizing rejections suffers from the reverse problem that the algorithm may accept no calls even if in hindsight it was possible to accept, say, half of them. However, in many applications, even optimal performance in such a case would be unacceptable: if one's network required one to reject a significant fraction of calls, then the correct response would be to upgrade the network. It is these types of settings that motivate our work.

We assume in our results that the online algorithm is allowed *preemption*: at any time we may preempt (reject) requests that had previously been accepted, and simply count it as if the request had been rejected from the start. This is one of the standard models and is necessary to achieve any nontrivial bound on rejections. A second assumption we make is that each request is for a fixed path. That is, the requests can be thought of as a sequence of paths $p_1, p_2, \ldots$, and the decision made by the online algorithm is just whether to accept or reject each path, and does not involve routing. Again, if routing is part of the algorithm's job, then even in very simple settings, no nontrivial bound is possible for our performance metric.[2] Our results are then as follows:

[1] In fact, prior to Leonardi et al. [LMSPR98] the situation was even worse. Depending on the types of requests made, many of the randomized algorithms would, with probability $1 - 1/(\log n)$, accept no calls at all. That is, the variance of possible benefits was high compared to the expectation.

[2] Consider a 4-cycle ABCD with capacity c on each edge. Imagine that we are given c calls connecting the diagonally opposite nodes A and C, and then we are given either c calls connecting A and B, or else c calls connecting A and D, with equal probability. Every on-line algorithm rejects $c/2$ calls in expectation, while it was possible to reject none off-line. Similarly, with n separate 4-cycles, the on-line algorithm rejects $nc/2$ calls in expectation, while OPT rejects none.

Admission control on a line: When the underlying graph is a line, we can achieve a competitive ratio of 2 for any set of edge capacities. That is, the algorithm will reject at most twice as many as the minimum possible in hindsight.

Admission control on a general graph: For general graphs, we can achieve a competitive ratio of 2 if all edge capacities are 1 (the disjoint paths case). This extends to a ratio of $c + 1$ if all edge capacities are $\leq c$. For arbitrary capacities, we give a different algorithm that achieves a competitive ratio of $O(\sqrt{m})$, where m is the number of edges, which we match with an $\Omega(\sqrt{m})$ lower bound.

An interesting aspect of the rejection measure is that the easiest cases are when capacities are *low*. This is the opposite of the situation for the benefit measure, where low capacities are difficult and higher capacities make the problems easier.

1.1 Related Work

As discussed above, existing work on admission control has primarily focused on the problem of maximizing the number of accepted calls, rather than minimizing the number of rejected calls. (See the surveys by Plotkin [Plo95] and Leonardi [Leo98].) The one exception we are aware of is the work of Kamath, Palmon, and Plotkin [KPP96], who provide performance guarantees in terms of a competitive ratio on rejections. Their setting is quite different from ours, however. Most significantly, they assume the input to be probabilistically generated, not worst-case. They consider calls that are generated according to a Poisson process, with exponentially distributed holding times, and also assume the maximum bandwidth of a call to be very small relative to the available edge capacity (i.e. large capacities). Moreover, their model does not allow pre-emption.

Routing on fixed paths, as we consider here, was studied by Alon, Arad, and Azar [AAA99] under the traditional measure of maximizing benefit. Of course, in linear networks, and more generally in tree networks, one is necessarily routing on fixed paths; for trees, work of Awerbuch et al. [ABFR94] provides $O(\log n)$-competitive algorithms for for maximizing benefit. (See also the earlier work of [GG92, GGK+93] for linear networks, and the improved probabilistic guarantees obtained by Leonardi et al. [LMSPR98].)

A number of previous papers have considered the performance gains obtainable by allowing pre-emption, in the context of maximizing benefit. Adler and Azar [AA99] show that allowing pre-emption leads to an $O(1)$-competitive algorithm for benefit maximization in linear networks, when the benefit of a call is defined to be proportional to the bandwidth it consumes.

1.2 Notation and Definitions

We are given a graph G, which may be directed or undirected, with m edges and n nodes. Each edge e has an integer capacity $c_e > 0$. We are also given a sequence of requests, $p_1, p_2, \ldots$, each of which is a simple path in the graph. Each

path may either be accepted or rejected. The requirement is that for every edge e, the number of unrejected requests that have edge e should be no larger than c_e. We will call a set of rejection decisions *valid* if it satisfies this requirement.

In the off-line problem, we must simply find a small valid set of rejections. In the on-line problem, we are given requests one at a time, and we much choose to accept or reject the requests on-line so that the set of accepted requests never exceeds the capacity of any edge. We also allow our online algorithm to preempt an earlier request, i.e. we may reject a request after already accepting it. However, we may not accept a request after rejecting it.

Let OPT be a minimum valid set of rejections. We say that an algorithm is k-competitive (k may be a function of m, n, and c) if the number of requests *rejected* by this algorithm is at most $k|\text{OPT}|$.

One final note: Our algorithms will sometimes decide to reject some requests even when not strictly necessary. Because we have preemption, these can always be implemented in a lazy manner. That is, such requests are marked but not actually rejected until a new request arrives that causes a conflict with it.

2 Preliminaries: Set-Cover and Vertex-Cover

A well-known result for the set-cover problem is that if every point is in at most k sets, then there is a simple k-approximation algorithm: pick an arbitrary uncovered point, take all $\leq k$ sets that cover it, and repeat. The case $k = 2$ corresponds to vertex cover.

A slight generalization of the $k = 2$ case is a setting in which a point may potentially be covered by many sets $s_1, s_2, \ldots$, but where we are guaranteed that some two of those sets s_i, s_j cover their union. Then one can achieve a 2-approximation as follows: pick an arbitrary uncovered point p, find two sets that cover the union of all sets covering p, take those two sets and repeat. This is a 2-approximation because each time two sets are chosen, they can be charged to whatever set s_p in the optimal solution is used to cover p. Because the two sets chosen by the algorithm contain s_p, we are guaranteed that each selection of two sets is charged to a unique set in the optimal cover.

Some of the results below can be viewed as an online version of this algorithm and guarantee.

3 Admission Control on a Line

We begin with the special case of a line graph. Each edge e has some arbitrary capacity c_e. A request corresponds to an interval on this line and the capacities limit the number of intervals covering any given edge that may be accepted. We show a 2-competitive algorithm, based on the set-cover idea above. The idea is that whenever a new request cannot be accepted due to capacity constraints, we look at (an arbitrary) one of the edges that would go over capacity, and throw out the two requests p_l and p_r covering that edge that extend farthest to the

left and farthest to the right, respectively. (One of these may or may not be the current request.) We then accept the current request if we did not throw it out. To be more precise:

1. If a request can be accepted, accept it.
2. If a request cannot be accepted, then choose an arbitrary edge e that would be put over capacity.
   a) Among the unrejected requests that contain e (including the current request), let p_l be one that extends furthest to the left.
   b) Among the unrejected requests that contain e (including the current request), let p_r be one that extends furthest to the right.
3. Reject p_l and p_r, and accept the current request if it is not one of $\{p_l, p_r\}$.

Theorem 1. *The above algorithm is 2-competitive.*

Proof. Consider some optimal valid rejection set OPT. Each time the algorithm rejects a pair of requests $\{p_l, p_r\}$, we will modify OPT by adding at most 1 request to it, in order to maintain an invariant that OPT is a superset of the requests rejected by the online algorithm. We do this as follows. Each time the online algorithm reaches case 2, we know that OPT must have rejected at least one request p_{opt} of those being considered by the online algorithm (i.e., at least one of those covering edge e that have not yet been rejected by the online algorithm). Therefore, when the online algorithm rejects p_l and p_r, we know that (viewing paths as sets of edges) $p_l \cup p_r \supseteq p_{opt}$. Therefore, if we put p_l and p_r into OPT, and then remove p_{opt} if neither p_l nor p_r had been in OPT already, this only adds 1 to the size of OPT, maintains its status as a valid rejection set, and maintains our invariant. So, if OPT_{init} is the true offline optimal, OPT_{final} is the final OPT set achieved by the above transformation, and t is the number of requests rejected by the online algorithm, then $t \leq |\text{OPT}_{final}| \leq |\text{OPT}_{init}| + t/2$, and therefore $t \leq 2|\text{OPT}_{init}|$. □

Another way of viewing this argument is that each time the algorithm rejects two requests p_l and p_r, we give OPT a "two-fer", allowing it to reject those two requests for the price of 1. Since OPT must reject some request contained in their union, it might as well take the offer. Inductively, at the end of the game, OPT has rejected the exact same set as the online algorithm, but at half the cost.

The above algorithm and analysis also applies if the underlying graph is a cycle.

4 General Graphs

4.1 The Low Capacity Case

Theorem 2. *On a general graph G, if every edge e has capacity $c_e \leq c$, then there is a simple $(c+1)$-competitive algorithm.*

Proof. The algorithm is just an online version of the k-approximation to set cover:

1. If a request can be accepted, accept it.
2. If a request cannot be accepted, then choose the first edge e that would be over capacity. Reject the current request along with the c_e (unrejected) other requests that contain e.

This algorithm rejects sets of requests of size $\leq c+1$ that all share an edge. These sets are disjoint. Any valid rejection set must include at least one request from each of these sets. Therefore, the algorithm achieves a competitive ratio of $c+1$. □

Thus, if all edges have capacity 1 (the disjoint paths case) we have a 2-competitive algorithm.

4.2 General Capacities

The above algorithm gets worse as the capacities in the graph become large. Can we achieve a bound independent of the capacities for general graphs? The connection to set-cover suggests that perhaps we could achieve an $O(\log m)$ bound. However, it turns out that the online nature of the problem makes that impossible. What we show instead are a set of matching $\Theta(\sqrt{m})$ upper and lower bounds. We begin with the lower bound.

Theorem 3. *There is a $\Omega(\sqrt{m})$ lower bound on the competitive ratio of any online algorithm for general graphs with arbitrary capacities. This holds for randomized algorithms as well.*

Proof. For clarity, we will use a multigraph for the lower bound. The multigraph consists of $k+1$ vertices $\{0, 1, \ldots, k\}$ arranged in a line, with k edges connecting each vertex to the next. So the total number of edges is k^2. Each edge has capacity k^{k-1}.

We begin by seeing k^k paths of length k, one for each possible route between vertex 0 and vertex k. By design, these will fill all edges exactly to capacity. We then see k single-edge paths: the first path is a random edge between vertex 0 and vertex 1; the second is a random edge between vertex 1 and vertex 2, and so on.

The offline algorithm needs only to reject one path, namely the path among the first k^k that happens to match the sequence of k single-edge paths seen at the end. However, any online algorithm must reject at least $k/2$ in expectation. That is because if $j < k$ paths have been rejected so far, then the next single-edge path seen has at least a $(k-j)/k$ chance of causing its edge to go over capacity. Therefore, the competitive ratio of any online algorithm is at least $k/2$ which is $\Omega(\sqrt{m})$. □

One can also give the above argument using a standard (non-multi) graph. For example, we can have the underlying graph be the complete graph and initially see all $n!$ Hamiltonian paths that (by design) fill all edges exactly to capacity. We then see n edges one at a time that together make up a random Hamiltonian path. The optimal offline algorithm again rejects just one path: namely, the Hamiltonian path among the first $n!$ corresponding to the final sequence of n edges. However, any online algorithm will need to reject at least $\Omega(n)$ paths in expectation.

We now give a matching $O(\sqrt{m})$ upper bound. Specifically, we present a $4\sqrt{m}$-competitive algorithm for an arbitrary multigraph with m edges. The algorithm is as follows, starting with zero "chips" on every edge and $R = 0$.

1. If a request covers at least $\sqrt{m}$ edges that have chips, then
   a) Reject the request
   b) Remove one chip from each of the request's edges (that have chips)
2. If a request cannot be accepted (some edge would be over capacity), then
   a) Reject the request
   b) $R = R + 1$
   c) If R is a multiple of $\sqrt{m}$, then
      i. Add a chip to each edge
      ii. Reapply Step 1 to every accepted request so far.
3. Else, accept the request.

Theorem 4. *The above algorithm is $O(\sqrt{m})$-competitive.*

Analysis. Observe that the number of rejections from Step 1 (line 1a) is no more than the number of rejections from Step 2 (line 2a). This is because, after R Step-2 rejections, we have placed no more than $R/\sqrt{m}$ chips on each edge, and every Step-1 rejection removes at least $\sqrt{m}$ chips. Thus, to show that the algorithm is $4\sqrt{m}$-competitive, we will show that $R \leq 2|OPT|\sqrt{m}$. From here on, when we refer to a rejection, we mean a Step-2 rejection.

Just before we perform each rejection, we "blame" it on an individual edge in one of OPT's rejections, as follows. Let e be the first of the edges that would have gone over capacity had we accepted the request. We blame the rejection on the first OPT rejection that has not yet been rejected, has e, and has not yet been blamed for e. Not only must there be some such OPT rejection to blame, but it must come no later in the request sequence than the current request. To see this, say we have had e as a blame edge t times before, and we have rejected r of OPT's rejections that have e. Then, including the current request, we must have seen $c_e + r + t + 1$ requests that contain e. OPT must also reject at least $r + t + 1$ requests with e, and we have rejected r of these and blamed t of them for e, leaving at least 1 previous OPT rejection to blame.

Also notice that after $|OPT|$ chips have been removed from an edge, we will not blame any more rejections on that edge. This is because the total number of requests that have an edge does not exceed $c_e + |OPT|$. Finally, it suffices

to show that no OPT rejection is blamed for more than $\sqrt{m}$ rejections after $R = |OPT|\sqrt{m}$. At this point, we have placed $|OPT|$ chips on each edge. Fix an OPT rejection. Look at the first rejection after $R = |OPT|\sqrt{m}$ blamed on it. Since we did not reject the OPT rejection in Step 1, it has less than $\sqrt{m}$ edges with chips. All of its other edges must have had at least $|OPT|$ chips removed, so they will never blamed again. Thus, we will blame at most $\sqrt{m}$ edges on each OPT rejection after $R = |OPT|\sqrt{m}$, which implies $R \leq 2|OPT|\sqrt{m}$ and the total number of rejections is at most $4|OPT|\sqrt{m}$.

5 The Offline Case

It is interesting to consider the *offline* version of our problem because of its connection to set-cover. For the off-line problem, we know the *excess* of each edge, i.e. the number of calls that include that edge minus its capacity. Let n_e be the excess of edge e. Our goal is to reject the fewest requests such that each edge e is contained in at least n_e rejections. This can be thought of as generalization of a set cover problem, where each point e has an associated number n_e, and instead of the usual goal of covering each point at least once, a legal cover must cover point e at least n_e times.

Let us define $N = \sum_e n_e$; that is, N is the total sum of the excesses. Then, the usual analysis of greedy set-cover gives us an $O(\log N)$ approximation to this generalized problem. In particular, if we imagine placing n_e chips on point e, then the greedy set-cover algorithm becomes: take the set that covers the most points of those with chips on them, remove one chip from each point covered, and repeat. If the optimal solution uses k sets, then at each step, the greedy algorithm must remove at least a $1/k$ fraction of the chips remaining, giving us the $O(\log N)$ ratio.

A natural question is whether this upper bound can be improved to $O(\log m)$ where m is the number of points (edges). This would be strictly better than what is achievable for the online problem. It is not clear if the greedy algorithm can be used to achieve this, but we *can* get $O(\log m)$ via randomized rounding as follows.

Formulate the problem as a linear programming problem, where $0 \leq f_i \leq 1$ is the fraction of set s_i to take (the fraction of the ith request to reject). Our objective is to minimize $\sum f_i$ subject to the chip (capacity) constraints $\sum_{i:e \in s_i} f_i \geq n_e$, for every point e. The minimum value of the objective will be no larger than k, the value of the optimal integer solution. To round, we choose each set independently, with probability of choosing set s_i equal to $\min(1, 5f_i \log m)$.

Next, Chernoff bounds imply that a given point e is covered at least n_e times with probability $\geq 1 - 1/m^2$. To see this, first note that we do not have to worry about those sets that have $5f_i \log m \geq 1$, because we will select them for certain. So ignore these sets. Let z_e be the sum of the f_i for the remaining sets s_i that have e, and we can assume $z_e \geq 1$ else we are already done. We expect to select $5z_e \log m$ sets covering e, and the only way in which we could fail is if the number we actually select is less than z_e. By the multiplicative Chernoff bounds, this

happens with probability less than

$$e^{-\left(1-\frac{1}{5\log m}\right)^2(5z_e\log m)/2} \leq \frac{1}{m^2}$$

for sufficiently large m. Thus with probability at least $1-m/m^2$ we will have *all* points covered the desired number of times. Furthermore, the expected number of sets chosen is $O(k\log m)$, as desired.

6 Conclusions

We have shown that in a number of natural cases, we can achieve good, or at least nontrivial bounds for minimizing the number of rejections in admission control. One open question left by these results is that our $\Omega(\sqrt{m})$ lower bound requires an exponential number of requests and exponential size capacities. Perhaps one might be able to achieve bounds that are logarithmic in m if we also allow logarithmic dependence on the maximum capacity c.

Acknowledgements. This research was supported in part by NSF grants CCR-9732705 and CCR-0085982, NSF Faculty Early Career Development Award CCR-9701399, a David and Lucile Packard Foundation Fellowship, an ONR Young Investigator Award, and an IBM Graduate Fellowship.

References

AA99. R. Adler and Y. Azar. Beating the logarithmic lower bound: randomized preemptive disjoint paths and call control algorithms. In *Proceedings of 10th SODA*, pages 1–10, 1999.

AAA99. N. Alon, U. Arad, and Y. Azar. Independent sets in hypergraphs with applications to routing via fixed paths. In *Proceedings of 2nd APPROX*, pages 16–27, 1999.

AAP93. Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive online routing. In *Proc. 34th Symp. Foundations of Computer Science*, pages 32–40, 1993.

ABFR94. Baruch Awerbuch, Yair Bartal, Amos Fiat, and Adi Rosén. Competitive non-preemptive call control. In *Proc. 5th Symp. on Discrete Algorithms*, pages 312–320, 1994.

AGLR94. Baruch Awerbuch, Rainer Gawlick, Tom Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proc. 35th Symp. Foundations of Computer Science*, pages 412–423, 1994.

GG92. Juan A. Garay and I. S. Gopal. Call preemption in communications networks. In *Proc. INFOCOM '92*, pages 1043–1050, 1992.

GGK+93. Juan A. Garay, I. S. Gopal, Shay Kutten, Yishay Mansour, and M. Yung. Efficient on-line call control algorithms. In *Proc. 2nd Israel Symp. on Theory of Computing and Systems*, pages 285–293, 1993.

KPP96. A. Kamath, O. Palmon, and Serge Plotkin. Routing and admission control in general topology networks with poisson arrivals. In *Proc. 7th Symp. on Discrete Algorithms*, pages 269–278, 1996.

Leo98. S. Leonardi. On-line network routing. In A. Fiat and G. Woeginger, editors, *On-line Algorithms*. Springer-Verlag, 1998.

LMSPR98. Stefano Leonardi, Alberto Marchetti-Spaccamela, A. Presciutti, and Adi Rosèn. On-line randomized call-control revisited. In *Proc. 9th Symp. on Discrete Algorithms*, pages 323–332, 1998.

Plo95. Serge Plotkin. Competitive routing in atm networks. *IEEE J. Selected Areas in Communications*, pages 1128–1136, 1995.

Secure Multi-party Computational Geometry

Mikhail J. Atallah and Wenliang Du

Department of Computer Sciences and
Center for Education and Research in Information Assurance and Security
Purdue University
West Lafayette, IN 47907
{mja,duw}@cs.purdue.edu

Abstract. The general secure multi-party computation problem is when multiple parties (say, Alice and Bob) each have private data (respectively, a and b) and seek to compute some function $f(a, b)$ without revealing to each other anything unintended (i.e., anything other than what can be inferred from knowing $f(a, b)$). It is well known that, in theory, the general secure multi-party computation problem is solvable using circuit evaluation protocols. While this approach is appealing in its generality, the communication complexity of the resulting protocols depend on the size of the circuit that expresses the functionality to be computed. As Goldreich has recently pointed out [6], using the solutions derived from these general results to solve specific problems can be impractical; problem-specific solutions should be developed, for efficiency reasons. This paper is a first step in this direction for the area of computational geometry. We give simple solutions to some specific geometric problems, and in doing so we develop some building blocks that we believe will be useful in the solution of other geometric and combinatorial problems as well.

1 Introduction

The growth of the Internet opens up tremendous opportunities for cooperative computation, where the answer depends on the private inputs of separate entities. These computations could even occur between mutually untrusting entities. The problem is trivial if the context allows the conduct of these computations by a trusted entity that would know the inputs from all the participants; however if the context disallows this then the techniques of secure multi-party computation become very relevant and can provide useful solutions.

In this paper we investigate how various computational geometry problems could be solved in a cooperative environment, where two parties needs to solve a geometric problem based on their joint data, but neither wants to disclose its private data to the other party. Some of the problems we solve in this framework are:

Problem 1. (Point-Inclusion) Alice has a point z, and Bob has a polygon P. They what to determine whether z is inside P, without revealing to each other

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 165–179, 2001.

any more than what can be inferred from that answer. In particular, neither of them allowed to learn such information about the relative position of z and P as whether z is at the northwest side of P, or whether z is close to one of the borders of P, etc.

Problem 2. (Intersection) Alice has a polygon A, and Bob has a polygon B; they both want to determine whether A and B intersect (not where the intersection occurs).

Problem 3. (Closest Pair) Alice has M points in the plane, Bob has N points in the plane. Alice and Bob want to jointly find two points among these $M + N$ points, such that their mutual distance is smallest.

Problem 4. (Convex Hulls) Alice has M points in the plane, Bob has N points in the plane. Alice and Bob want to jointly find the convex hulls for these $M+N$ points; however, neither Alice nor Bob wants to disclose any more information to the other party than what could be derived from the result.

Of course all of the above problems, as well as other computational geometry problems, are special cases of the general Secure Multi-party Computation problem [16,9,6]. Generally speaking, a secure multi-party computation problem deals with computing a function on any input, in a distributed network where each participant holds one of the inputs, ensuring that no more information is revealed to a participant in the computation than can be computed from that participant's input and output.

In theory, the general secure multi-party computation problem is solvable using circuit evaluation protocol [16,9,6]. While this approach is appealing in its generality, the communication complexity of the protocol it generates depends on the size of the circuit that expresses the functionality F to be computed, and in addition, involves large constant factors in their complexity. Therefore, as Goldreich points out in [6], using the solutions derived by these general results for special cases of multi-party computation can be impractical; special solutions should be developed for special cases for efficiency reasons. This is a good motivation for seeking special solutions to computational geometry problems, solutions that are more efficient than the general theoretical solutions.

Due to page limitations, we include detailed solutions to only two of the above problems: point-inclusion problem and intersection problem. Our work assumes that all parties are semi-honest; informally speaking, a semi-honest party is one who follows the protocol properly with the exception that it keeps a record of all its intermediate computations and might try to derive other parties' private inputs from the record. We also assume that adding a random number to an x effectively hides x. The assumption is known to be true in a finite field, in the infinite case, our protocols can be considered heuristic or approximation.

1.1 Related Work

The history of the multi-party computation problem is extensive since it was introduced by Yao [16] and extended by Goldreich, Micali, and Wigderson [9], and by many others. These works use a similar methodology: each functionality F is represented as a Boolean circuit, and then the parties run a protocol for every gate in the circuit. While this approach is appealing in its generality and simplicity, the protocols it generates depend on the size of the circuit. This size depends on the size of the input and on the complexity of expressing F as a circuit. If the functionality F is complicated, using the circuit evaluation protocol will typically not be practical. However, if F is a simple enough functionality, using circuit a evaluation protocol can be practical.

The existing protocols listed below serve as important building blocks in our solutions. Our paper [5] contains some primitives for general scientific problems, that could be used as subroutines by some of our computations (as special cases), however the next section will give better solutions for the special cases that we need than the general ones given in [5] (more on this later).

The Circuit Evaluation Protocol. In a circuit evaluation protocol, each functionality is represented by a Boolean circuit, and the construction takes this Boolean circuit and produces a protocol for evaluating it. The protocol scans the circuit from the input wires to the output wires, processing a single gate in each *basic step*. When entering each basic step, the parties hold shares of the values of the input wires, and when the step is completed they hold shares of the output wire.

1-out-of-N Oblivious Transfer. Goldreich's circuit evaluation protocol uses the 1-out-of-N Oblivious Transfer, and our protocols in this paper also heavily depends on this protocol. An 1-out-of-N Oblivious Transfer protocol [7,4] refers to a protocol where at the beginning of the protocol one party, Bob has N inputs $X_1, \ldots, X_N$ and at the end of the protocol the other party, Alice, learns one of the inputs X_i for some $1 \leq i \leq N$ of her choice, without learning anything about the other inputs and without allowing Bob to learn anything about i. An efficient 1-out-of-N Oblivious Transfer protocol was proposed in [11] by Naor and Pinkas. By combining this protocol with the scheme by Cachin, Micali and Stadler [8], the 1-out-of-N Oblivious Transfer protocol could be achieved with polylogarithmic (in n) communication complexity.

Homomorphic Encryption Schemes

We need a public-key cryptosystems with a *homomorphic* property for some of our protocols: $E_k(x) * E_k(y) = E_k(x+y)$. Many such systems exist, and examples include the systems by Benaloh [2], Naccache and Stern [10], Okamoto and Uchiyama [13], Paillier [14], to mention a few. A useful property of homomorphic encryption schemes is that an "addition" operation can be conduced based on the encrypted data without decrypting them.

Yao's Millionaire Problem

This is another protocol used as a primitive in our solutions; The purpose of the protocol is to compare two private numbers (i.e., determine which is larger). This private comparison problem was first proposed by Yao [15] and is referred as Yao's Millionaire Problem (because two millionaires wish to know who is richer, without revealing any other information about their net worth). The early cryptographic solution by Yao [15] has communication complexity that is exponential in the number of bits of the numbers involved, using an untrusted third party. Cachin proposed a solution [3] based on the Φ-hiding assumption. His protocol uses an untrusted third party that can misbehave on its own (for the purpose of illegally obtaining information about Alice's or Bob's private vectors) but does not collude with either participant. The communication complexity of Cachin's scheme is $O(\ell)$, where ℓ is the number of bits of each input number.

2 New Building Blocks

In this section, we introduce two secure two-party protocols, a scalar product protocol, and a vector dominance protocol. Apart from serving as building blocks in solving the secure two-party computational geometry problems considered later in the paper, these two protocols are of independent interest and will be useful in solving other problems as well.

2.1 Scalar Product Protocol

Our paper [5] contains a matrix product protocol that could be used for scalar product (as a special case), but the scalar product protocol given below is better than using the general matrix multiplication protocol. We use $X \cdot Y$ to denote the scalar product of two vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$, $X \cdot Y = \sum_{k=1}^{n} x_k y_k$. Our definition of the problem is slightly different more general: We assume that Alice has the vector X and Bob has the vector Y, and the goal of the protocol is for Alice (but not Bob) to get $X \cdot Y + v$ where v is random and known to Bob only (of course without either side revealing to the other the private data they start with). Our protocols can easily be modified to work for the version of the problem where the random v is given ahead of time as part of Bob's data (the special case $v = 0$ puts us back in the usual scalar product definition). The purpose of Bob's random v is as follows: If $X \cdot Y$ is a partial result that Alice is not supposed to know, then giving her $X \cdot Y + v$ prevents Alice from knowing the partial result (even though the scalar product has in fact been performed); later, at the end of the multiple-step protocol, the effect of v can be effectively "subtracted out" by Bob without revealing v to Alice (this should become clearer with example protocols that we later give).

Problem 5. (Scalar Product Problem) Alice has a vector $X = (x_1, \ldots, x_n)$ and Bob has a vector $Y = (y_1, \ldots, y_n)$. Alice (but not Bob) is to get the result of $u = X \cdot Y + v$ where v is a random scalar known to Bob only.

We have developed two protocols, and we will present both of them here.

Scalar Product Protocol 1. Consider the following naive solution: Alice sends p vectors to Bob, only one of which is X (the others are arbitrary). Then Bob computes the scalar products between Y and each of these p vectors. At the end Alice uses the 1-out-of-N oblivious transfer protocol to get back from Bob the product of X and Y. Because of the way oblivious transfer protocol works, Alice can decide which scalar product to get, but Bob could not learn which one Alice has chosen. There are many drawbacks to this approach: If the value of X has certain public-known properties, Bob might be able to differentiate X from the other $p-1$ vectors, but even if Bob is unable to recognize X his chances of guessing it is an unacceptably low 1 out of p.

The above drawbacks can be fixed by dividing vector X into m random vectors $V_1, \ldots, V_m$ of which it is the sum, i.e., $X = \sum_{i=1}^{m} V_i$. Alice and Bob can use the above naive method to compute $V_i \cdot Y + r_i$, where r_i is random number and $\sum_{i=1}^{n} r_i = v$ (see Figure 1). As a result of the protocol, Alice gets $V_i \cdot Y + r_i$ for $i = 1, \ldots, m$. Because of the randomness of V_i and its position, Bob could not find out which one is V_i. Certainly, there is 1 out p possibility that Bob can guess the correct V_i, but since X is the sum of m such random vectors, the chance that Bob guesses the correct X is 1 out p^m, which could be very small if we chose p^m large enough.

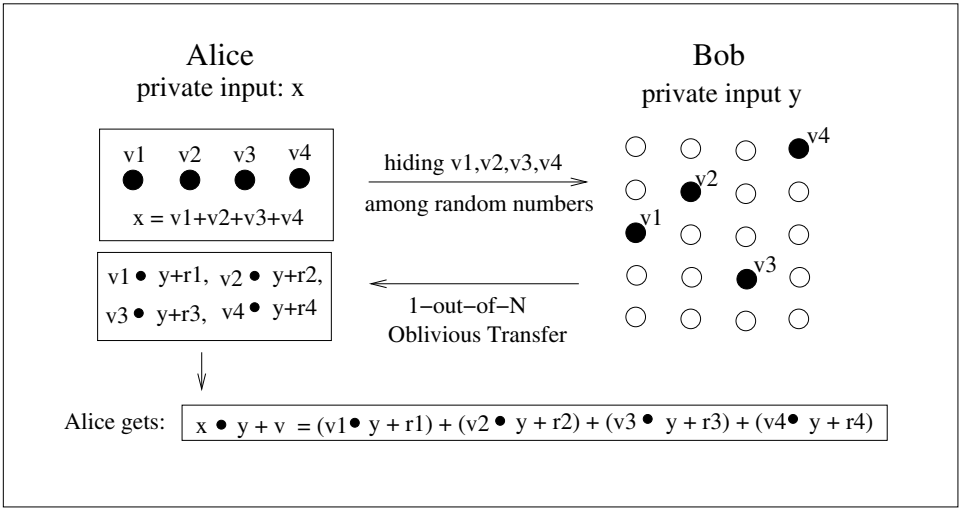


Fig. 1. Scalar Product Protocol 1

After Alice gets $V_i \cdot Y + r_i$ for $i = 1, \ldots, n$, she can compute $\sum_{i=1}^{m}(V_i \cdot Y + r_i)$ $= X \cdot Y + v$. The detailed protocol is described in the following:

Protocol 1 *(Two-Party Scalar Product Protocol 1)*
Inputs: Alice has a vector $X = (x_1, \ldots, x_n)$, and Bob has a vector $Y = (y_1, \ldots, y_n)$.

Outputs: Alice (but not Bob) gets $X \cdot Y + v$ where v is a random scalar known to Bob only.

1. Alice and Bob agree on two numbers p and m, such that p^m is so large that conducting p^m additions is computationally infeasible.
2. Alice generates m random vectors, $V_1, \ldots, V_m$, such that $X = \sum_{j=1}^{m} V_i$.
3. Bob generates m random numbers $r_1, \ldots, r_m$ such that $v = \sum_{j=1}^{m} r_j$.
4. For each $j = 1, \ldots, m$, Alice and Bob conduct the following sub-steps:
   a) Alice generates a secret random number k, $1 \leq k \leq p$.
   b) Alice sends $(H_1, \ldots, H_p)$ to Alice, where $H_k = V_j$, and the rest of H_i's are random vectors. Because k is a secret number known only to Alice, Bob does not know the position of V_j.
   c) Bob computes $Z_{j,i} = H_i \cdot Y + r_j$ for $i = 1, \ldots, p$.
   d) Using the 1-out-of-N Oblivious Transfer protocol, Alice gets $Z_j = Z_{j,k} = V_j \cdot Y + r_j$, while Bob learns nothing about k.
5. Alice computes $u = \sum_{j=1}^{m} Z_j = X \cdot Y + v$.

How is privacy achieved:

- If Bob chooses to guess, his chance of guessing the correct X is p^m.
- The purpose of r_j is to add randomness to $V_j \cdot Y$, thus preventing Alice from deriving information about Y.

Scalar Product Protocol 2. In the following discussion, we define $\pi(X)$ as another vector whose elements are random permutation of those of vector X.

We begin with two observations. First, a property of the scalar product $X \cdot Y$ is that $\pi(X) \cdot \pi(Y) = X \cdot Y$, regardless of what π is. Secondly, if Bob sends a vector $\pi(V)$ to Alice, where π and V are known only to Bob, Alice's chance of guessing the position of any single element of the vector V is 1 out of n (n is the size of the vector); Alice's chance of guessing the positions of all of the elements of the vector V is 1 out of $n!$.

A naive solution would be to let Alice get both $\pi(X)$ and $\pi(Y)$ but not π. Let us ignore for the time being the drawback that Alice gets the items of Y in permuted order, and let us worry about not revealing π to Alice: Letting Alice know $\pi(X)$ allows her to easily figure out the permutation function π from knowing both X and $\pi(X)$. In order to avoid this problem, we want to let Alice know only $\pi(X + R_b)$ instead of $\pi(X)$, where R_b is a random vector known only to Bob. Because of the randomness of $X + R_b$, to guess the correct π, Alice's chance is only 1 out of $n!$. Therefore to get the final scalar product, Bob only needs to send $\pi(Y)$ and the result of $R_b \cdot Y$ to Alice, who can compute the result of the scalar product by using

$$X \cdot Y = \pi(X + R_b) \cdot \pi(Y) - R_b \cdot Y$$

Now we turn our attention to the drawback that giving Alice $\pi(Y)$ reveals too much about Y (for example, if Alice is only interested in a single element

of the vector Y, her chance of guessing the right one is an unacceptably low 1 out of n). One way to fix this is to divide Y to m random pieces, $V_1, \ldots, V_m$, with $Y = V_1 + \ldots + V_m$; then Bob generates π random permutations $\pi_1, \ldots, \pi_m$ (one for each "piece" V_i of Y) and lets Alice know $\pi_i(V_i)$ and $\pi_i(X + R_b)$ for $i = 1, \ldots, m$. Now in order to guess the correct value of a single element of Y, Alice has to guess the correct position of V_i in each one of the m rounds; the possibility of a successful guessing becomes 1 out of n^m.

Now, let us consider the unanswered question: how could Alice get $\pi(X+R_b)$ without learning π or R_b? We do this with a technique based on a homomorphic public key system, that was used in [1] in a different context (to compute the minimum value in a vector that is the difference of Alice's private vector and Bob's private vector). Recall that an encryption scheme is *homomorphic* if $E_k(x)*E_k(y) = E_k(x+y)$. A good property of homomorphic encryption schemes is that "addition" operation can be conduced based on the encrypted data without decrypting them. Based on the homomorphic public key system, we have the following Permutation Protocol (where, for a vector $Z = (z_1, \ldots, z_n)$, we define $E(Z) = (E(z_1), \ldots, E(z_n))$, $D(Z) = (D(z_1), \ldots, D(z_n))$):

Protocol 2 *(Permutation Protocol)*
Inputs: Alice has a vector X. Bob has a permutation π and a vector R.
Output: Alice gets $\pi(X + R)$.

1. Alice generates a key pair for a homomorphic public key system and sends the public key to Bob. The corresponding encryption and decryption is denoted as $E(\cdot)$ and $D(\cdot)$.
2. Alice encrypts $X = (x_1, \ldots, x_n)$ using her public key and sends $E(X) = (E(x_1), \ldots, E(x_n))$ to Alice.
3. Bob computes $E(R)$, then computes $E(X) * E(R) = E(X + R)$; Bob then permutes $E(X + R)$ using the random permutation function π, thus getting $\pi(E(X + R))$; Bob sends the result of $\pi(E(X + R))$ to Alice.
4. Alice computes $D(\pi(E(X + R))) = \pi(D(E(X + R))) = \pi(X + R)$.

Based on Secure Two-Party Permutation Protocol, we have developed the following scalar product protocol:

Protocol 3 *(Secure Two-Party Scalar Product Protocol 2)*
Inputs: Alice has a secret vector X, Bob has a secret vector Y.
Output: Alice gets $X \cdot Y + v$ where v is a random scalar known to Bob only.

1. Bob's set up:
   a) Bob divides Y to m random pieces, s.t. $Y = V_1 + \ldots + V_m$.
   b) Bob generates m random vectors $R_1, \ldots, R_m$, let $v = \sum_{i=1}^{m} V_i \cdot R_i$.
   c) Bob generates m random permutations $\pi_1, \ldots, \pi_m$.
2. For each $i = 1, ..., m$, Alice and Bob do the following:
   a) Using Secure Two-Party Permutation Protocol, Alice gets $\pi_i(X + R_i)$ without learning either π_i or R_i.

b) Bob sends $\pi_i(V_i)$ to Alice.
c) Alice computes $Z_i = \pi_i(V_i) \cdot \pi_i(X + R_i) = V_i \cdot X + V_i \cdot R_i$

3. Alice computes $u = \sum_{i=1}^{m} Z_i = \sum_{i=1}^{m} V_i \cdot X + \sum_{i=1}^{m} V_i \cdot R_i = X \cdot Y + v$

How is privacy achieved:

- The purpose of R_i is to prevent Alice from learning π_i.
- The purpose of π_i is to prevent Alice from learning V_i. Although Alice learns a random permutation of the V_i, she does not learn more because of the randomness of V_i. Without π_i, Alice could learn each single value of V_i.
- If Alice chooses to guess, in order to successfully guess all of the elements in Y, her chance is $(\frac{1}{n!})^m$.
- Alice's chance of successfully guessing just one elements of Y is n^m. For example, in order to guess the kth element of Y, Alice has to guess the the corresponding elements in $\pi_i(V_i)$ for all $i = 1, \ldots, m$; for each i, the chance is $\frac{1}{n}$.
- A drawback of this protocol is that the information about $\sum_{i=1}^{n} y_i$ is disclosed because the random permutation does not help to hide this information.

Comparison of These Two Protocols. The communication cost of Protocol 3 is $4m * n$, where m is a security parameter (so that $\mu' = n^m$ is large enough). The communication cost of Protocol 1 is $p * t * n$, where $p \geq 2$ and t are security parameters such that $\mu'' = p^t$ is large enough. Setting $\mu' = \mu'' = \mu$ for the sake of comparison, the communication cost of Protocol 3 is $4 \log \mu \frac{n}{\log n}$ and the communication cost of Protocol 1 is $\frac{p \log \mu}{\log p} n$. When n is large, Protocol 3 is more efficient than Protocol 1.

2.2 Secure Two-Party Vector Dominance Protocol

Definition 1 *(Vector Dominance)* Let $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$; if for all $i = 1, \ldots, n$ we have $a_i > b_i$, then we say that A dominates B and denote it by $A \succ B$.

Problem 6. (Secure Two-Party Vector Dominance Problem) Alice has a vector $A = (a_1, \ldots, a_n)$ and Bob has a vector $B = (b_1, \ldots, b_n)$. Alice wants to know whether A dominates B. Note in the case where A does not dominate B, neither Alice nor Bob should learn the relative ordering of any individual a_i, b_i pair (i.e., whether $a_i < b_i$ or not).

The Protocol. We first give an outline of the protocol, then discuss each step in details.

Protocol 4 *(Secure Two-Party Vector Dominance Protocol)*
Inputs: Alice has a vector $A = (a_1, \ldots, a_n)$, Bob has a vector $B = (b_1, \ldots, b_n)$.

1. Inputs Disguise: Using a disguise technique (described later), Alice gets the disguised input $A' = (a'_1, \ldots, a'_{4n})$, and Bob gets the disguised input $B' = (b'_1, \ldots, b'_{4n})$. Let V_A
$$V_A = (\overbrace{1, \ldots, 1}^{2n}, \overbrace{0, \ldots, 0}^{2n})$$
2. Private Permutation: Bob generates a random permutation π and a random vector R. Using the Permutation Protocol (Protocol 2), Alice gets $A'' = \pi(A' + R)$. Bob also computes $B'' = \pi(B' + R)$, $V'_A = \pi(V_A)$.
3. Yao's Millionaire Comparison: Alice and Bob use Yao's Millionaire protocol as subroutine to compare A''_i with B''_i, for $i = 1, \ldots, 4n$, where A''_i (resp., B''_i) is the ith element of vector A'' (resp., B''). At the end, Alice gets the result $U = \{u_1, \ldots, u_{4n}\}$, where $u_i = 1$ if $A''_i > B''_i$, otherwise $u_i = 0$.
4. Dominance Testing: Alice and Bob use a private equality-testing protocol to compares U with V'_A: If $U = V'_A$, then A dominates B; otherwise, A does not dominate the B. (Note: when we later use this protocol for the intersection protocol, this step must be skipped.)

Outputs: If the Dominance Testing step needs to be skipped, Alice outputs U and Bob outputs V'_A. Otherwise, Alice and Bob each output the dominance testing results.

Step 1: Inputs Disguise

For convenience, we assume a_i and b_i for $i = 1, \ldots, n$ are integers; however our scheme can be easily extended to the non-integer case. The disguised inputs are the followings:

$$\begin{aligned} A' = (&2a_1, \ldots, 2a_n, (2a_1 + 1), \ldots, (2a_n + 1), \\ &-2a_1, \ldots, -2a_n, -(2a_1 + 1), \ldots, -(2a_n + 1)) \end{aligned} \tag{1}$$

$$\begin{aligned} B' = (&(2b_1 + 1), \ldots, (2b_n + 1), 2b_1, \ldots, 2b_n, \\ &-(2b_1 + 1), \ldots, -(2b_n + 1), -2b_1, \ldots, -2b_n) \end{aligned} \tag{2}$$

The purpose of the inputs disguise is to get the same number of $a'_i > b'_i$ situations as that of $a'_i < b'_i$ situations; therefore, nobody knows how many a_i's are larger than b_i's and vice versa. The disguise is based on the fact that if $a_i > b_i$, then $2a_i > 2b_i + 1$, $(2a_i + 1) > 2b_i$, $-2a_i < -(2b_i + 1)$, and $-(2a_i + 1) < -2b_i$, which generates two $>$'s, and two $<$'s.

Step 2: Private Permutation: This step is fully discussed in Secure Two-Party Permutation Protocol (Protocol 2).

Step 3: Yao's Millionaire Comparison

Alice now has $A'' = \pi(A' + R) = (a''_1, \ldots, a''_{4n})$, Bob has $B'' = \pi(B' + R) = (b''_1, \ldots, b''_{4n})$. They can use Yao's Millionaire Protocol to compare each a''_i with b''_i. Actually it is an one-side (asymmetric) version of it because only Alice learns the result. So at the end of this step, Alice gets $U = (u_1, \ldots, u_{4n})$, where for $i = 1, \ldots, 4n$, $u_i = 1$ if $a'_i > b'_i$, otherwise $u_i = 0$.

Step 4: Dominance Testing

Because V_A is exactly what U should be if vector A dominates B, we only need to find out whether $U = V_A$. Alice cannot just send U to Bob because it will allow Bob to find out the relationship between a_i and b_i for each $i = 1, \ldots, n$. So we need a way for Alice and Bob to determine whether Alice's U equals Bob's V_A without disclosing each person's private input to the other person.

This comparison problem is well studied, and was thoroughly discussed by Fagin, Naor, and Winkler [12]. Several methods for it were discussed in [12,11]. For example, the following is part of the folklore:

Protocol 5 *(Equality-Testing Protocol)*
Inputs: Alice has U, Bob has V_A.
Outputs: $U = V_A$ iff $E_B(E_A(U)) = E_A(E_B(V_A))$.

1. Alice encrypts U with a commutative encryption scheme, and gets $E_A(U)$; Alice sends $E_A(U)$ to Bob.
2. Bob encrypts $E_A(U)$, and gets $E_B(E_A(U))$; Bob sends the result back to Alice.
3. Bob encrypts V_A, gets $E_B(V_A)$; Bob sends $E_B(V_A)$ to Alice.
4. Alice encrypts $E_B(V_A)$, gets $E_A(E_B(V_A))$.
5. Alice compares $E_B(E_A(U))$ with $E_A(E_B(V_A))$.

3 Secure Two-Party Geometric Computations

In the following, we want to illustrate how the building blocks we studied earlier can be put together to solve geometric problems. Many other geometric problems are amenable to such solutions; and in fact we suspect that the solutions we give below can be further improved.

3.1 Secure Two-Party Point-Inclusion Problem

We will look at how the point-location problem is solved in a straightforward way without worrying about the privacy concern. The computation cost of this straightforward solution is $O(n)$. Although we know the computation cost of the best algorithm for the point-location problem is only $O(\log n)$, we are concerned that the "binary search" nature of that solution might lead to the disclosure of partial information. Therefore, for a preliminary result, we focus on the $O(n)$ solution. The algorithm works as follows:

1. Find the leftmost vertex l and the rightmost vertex r of the polygon.
2. Decide whether the point $p = (\alpha, \beta)$ is above all the edges on the lower boundary of the polygon between l and r.
3. Decide whether the point α is below all the edges on the upper boundary of the polygon between l and r.
4. If the above two tests are both true, then the point is inside the polygon, otherwise it is outside (or on the edge) of the polygon.

If we use $f_i(x, y) = 0$ for the equation of the line boundary of the polygon, where $f_i(x, y) = 0$ for $i = 1, \ldots, m$ represent the edges on the lower part of the boundary and $f_i(x, y) = 0$ for $i = m+1, \ldots, n$ represent the edges on the upper part of the boundary, then our goal is to decide whether $f_i(\alpha, \beta) > 0$ for all $i = 1, \ldots, m$ and $f_i(\alpha, \beta) < 0$ for all $i = m+1, \ldots, n$.

The Protocol. First, we need to find a way to compute $f_i(\alpha, \beta)$ without disclosing Alice's $p : (\alpha, \beta)$ to Bob or Bob's f_i to Alice. Moreover, no party should learn the result of $f_i(\alpha, \beta)$ for any i because that could disclose the relationship between the location and the edge. Since $f_i(\alpha, \beta)$ is a special case of scalar product, we can use Secure Two-Party Scalar Product Protocol to solve this problem. In this protocol, we will let both party share the result of $f_i(\alpha, \beta)$, namely, one party will have u_i, the other party will have v_i, and $u_i = f_i(\alpha, \beta) + v_i$; therefore nobody learns the value of $f_i(\alpha, \beta)$, but they can find out whether $f_i(\alpha, \beta) > 0$ by comparing whether $u_i > v_i$, which could be done using Yao's Millionaire Protocol [15,3].

However, we cannot use Yao's Millionaire Protocol for each (u_i, v_i) pair individually because that would disclose the relationship between u_i and v_i, thus reveal too much information. In fact, all we want to know is whether $(u_1, \ldots, u_n)$ dominates $(v_1, \ldots, v_n)$. This problem can be solved using the *Vector Dominance Protocol* (Protocol 4)

Based on the Scalar Product Protocol and Vector Dominance Protocol, we have the following Secure Two-Party Point-Inclusion Protocol:

1. Bob generates n random numbers $v_1, \ldots, v_n$.
2. Alice and Bob use Scalar Product Protocol to compute $u_i = f_i(\alpha, \beta) + v_i$, for $i = 1, \ldots, m$ and compute $u_i = -f_i(\alpha, \beta) + v_i$ for $i = m+1, \ldots, n$. According to the scalar product protocol, Alice will get $(u_1, \ldots, u_n)$ and Bob will get $(v_1, \ldots, v_n)$. Bob will learn nothing about u_i and (α, β); Alice will learn nothing about v_i and the function $f_i(x, y)$.
3. Alice and Bob use the *Vector Dominance Protocol* to find out whether vector $A = (u_1, \ldots, u_n)$ dominates $B = (v_1, \ldots, v_n)$. According to the Vector Dominance Protocol, if A does not dominate B, no other information is disclosed.

Claim. If $A = (u_1, \ldots, u_n)$ dominates $B = (v_1, \ldots, v_n)$, then the point $p = (\alpha, \beta)$ is inside the polygon; otherwise, the point is outside (or on the edge) of the polygon.

3.2 Secure Two-Party Intersection Problem

Two polygons intersect if (1) one polygon is inside another, or (2) at least one edge of a polygon intersects with one edge of another polygon. Since (1) can be decided using the Point-Inclusion Protocol, we only focus on (2).

We will first look at how the intersection problem could be solved in a straightforward way ($O(n^2)$) without worrying about the privacy concern. For

the same reason we decide not to use the more efficient $O(n)$ algorithm because of the concern about the partial information disclosure. The algorithm works as follows:

1. For each pair (e_i, e'_j), decide whether e_i intersects with e_j, where e_i is an edge of polygon A and e'_j is an edge of polygon B,
2. If there exists an edge $e_i \in A$ and an edge $e'_j \in B$, such that e_i intersects with e'_j, then A and B intersect.

We use $f_i(x,y), (x_i,y_i), (x'_i,y'_i)$, for $i = 1,\ldots,n_a$, to represent each edge of the polygon A, where $f_i(x,y)$ is the equation of the line containing that edge, (x_i,y_i) and (x_i,y_i) represents the two endpoints of the edge. We use $g_i(x,y)$, for $i = 1,\ldots,n_b$, to represent each edge of the polygon B.

The Protocol. During the testing of whether two edges intersect with each other, obviously, nobody should learn the result of each individual test; otherwise, he knows which of his edge intersects with the other party's polygon. In our scheme, Alice and Bob conduct these n^2 testings, but nobody knows the result of each individual test, instead, they share the results of each test, namely each of them gets a seemly-random piece of the result. One has to obtain both pieces in order to know the result of each test. At the end, all these shared pieces are put together in a way that only a single result is generated, to show only whether the two polygon boundaries intersect or not.

First, let us see how to conduct such a secure two-party testing of the intersection. Assume Alice has a edge $f_1(x,y) = 0$, where $f_1(x,y) = a_1x + b_1y + c_1$, and $a_1 \geq 0$; the two endpoints of the edge are (x_1,y_1) and (x'_1,y'_1). Bob has a line $f_2(x,y) = 0$, where $f_2(x,y) = a_2x + b_2y + c_2$, $a_2 \geq 0$; the two endpoints of the edge are (x_2,y_2) and (x'_2,y'_2). According to the geometries, f_1 and f_2 intersect if and only if f_1's two endpoints (x_1,y_1), (x'_1,y'_1) are on the different sides of f_2, and f_2's two endpoints (x_2,y_2) and (x'_2,y'_2) are on the different sides of f_1. In another words, f_1 and f_2 intersect if and only if one of the following expressions is true:

- $f_1(x_2,y_2) > 0 \wedge f_1(x'_2,y'_2) < 0 \wedge f_2(x_1,y_1) > 0 \wedge f_2(x'_1,y'_1) < 0$
- $f_1(x_2,y_2) > 0 \wedge f_1(x'_2,y'_2) < 0 \wedge f_2(x_1,y_1) < 0 \wedge f_2(x'_1,y'_1) > 0$
- $f_1(x_2,y_2) < 0 \wedge f_1(x'_2,y'_2) > 0 \wedge f_2(x_1,y_1) > 0 \wedge f_2(x'_1,y'_1) < 0$
- $f_1(x_2,y_2) < 0 \wedge f_1(x'_2,y'_2) > 0 \wedge f_2(x_1,y_1) < 0 \wedge f_2(x'_1,y'_1) > 0$

We cannot let either party know the results of $f_1(x_2,y_2)$, $f_1(x'_2,y'_2)$, $f_2(x_1,y_1)$, or $f_2(x'_1,y'_1)$ (in the following discussion, we will use $f(x,y)$ to represent any of these expressions). According to the Scalar Product Protocol, we can let Alice know the result of $f(x,y) + r$, and let Bob know r, where r is a random number generated by Bob. Therefore, nobody knows the actual value of $f(x,y)$, but Alice and Bob can still figure out whether $f(x,y) > 0$ by comparing $f(x,y) + r$ with r.

Let $u_1 = f_1(x_2,y_2) + r_1$, $u'_1 = f_1(x'_2,y'_2) + r'_1$, $u_2 = f_2(x_1,y_1) + r_2$, and $u'_2 = f_2(x'_1,y'_1) + r'_2$. Alice has (u_1,u'_1,u_2,u'_2) and Bob has (r_1,r'_1,r_2,r'_2). Then f_1 and f_2 intersect if and only if one of the following expressions is true:

- $u_1 > r_1 \wedge u'_1 < r'_1 \wedge u_2 > r_2 \wedge u'_2 < r'_2$
- $u_1 > r_1 \wedge u'_1 < r'_1 \wedge u_2 < r_2 \wedge u'_2 > r'_2$
- $u_1 < r_1 \wedge u'_1 > r'_1 \wedge u_2 > r_2 \wedge u'_2 < r'_2$
- $u_1 < r_1 \wedge u'_1 > r'_1 \wedge u_2 < r_2 \wedge u'_2 > r'_2$

Our next step is to compute each of the above expressions. As before, nobody should learn the individual comparison results, just the aggregate. Let us use E to denote any one of the above expressions. Using the Vector Dominance Protocol, we can get Alice to know a random piece t, and Bob to know another random piece s, such that E is true if and only if $t = s$.

Now Alice has $4 * n^2$ numbers $(t_1, \ldots, t_{4n^2})$, Bob has $(s_1, \ldots, s_{4n^2})$. We want to know whether there exists an $i = 1, \ldots, 4n^2$, such that $t_i = s_i$. Although there are some other approaches to achieve this, we believe using the circuit evaluation protocol is efficient in this case, because the size of the circuit is small (linear in the number of the items). The security of the circuit evaluation protocol guarantees that only the final results–`yes` or `no`–will be disclosed; nobody learns any other information, such as how many t_i's equal to s_i's, and which $t_i = s_i$.

The following is the outline of the protocol:

1. Let $m = n_a * n_b$.
2. For each pair of edges, perform the following sub-protocol. Suppose the index of this edge pair is i, for $i = 1, \ldots, m$; suppose $(f, (x_1, y_1), (x'_1, y'_1)) \in A$ and $(g, (x_2, y_2), (x'_2, y'_2)) \in B$ are two edges.
   a) Using the scalar product protocol, Alice gets $U = (u_1, u'_1, u_2, u'_2)$, and Bob gets $R = (r_1, r'_1, r_2, r'_2)$, where $u_1 = f(x_2, y_2) + r_1$, $u'_1 = f(x'_2, y'_2) + r'_1$, $u_2 = g(x_1, y_1) + r_2$, and $u'_2 = g(x'_1, y'_1) + r'_2$.
   b) Using the Vector Dominance Protocol, Alice gets $t_{i,1}, t_{i,2}, t_{i,3}, t_{i,4}$, and Bob gets $s_{i,1}, s_{i,2}, s_{i,3}, s_{i,4}$
3. Alice has $(t_{1,1}, t_{1,2}, t_{1,3}, t_{1,4}, \ldots, t_{m,1}, t_{m,2}, t_{m,3}, t_{m,4})$, and Bob has $(s_{1,1}, s_{1,2}, s_{1,3}, s_{1,4}, \ldots, s_{m,1}, s_{m,2}, s_{m,3}, s_{m,4})$. Alice and Bob uses circuit evaluation method to find out whether there exists $i \in \{1, \ldots, m\}$, $j \in \{1, \ldots, 4\}$, such that $t_{i,j} = s_{i,j}$.

4 Applications

The following two scenarios describe some potential applications of the problems we have discussed in this paper.

1. Company A decided that expanding its market share in some region will be very beneficial after a costly market research; therefore A is planning to do this. However A is aware of that another competing company B is also planning to expand its market share in some region. Strategically, A and B do not want to compete against each other in the same region, so they want to know whether they have a region of overlap? Of course, they do not want to give away location information because not only does this information cost both companies a lot of money, but it can also cause significant damage

to the company if it were disclosed to other parties: for example, a larger competitor can immediately occupy the market there before A or B even starts; or some real estate company can actually raise their price during the negotiation if they know A or B is very interested in that location. Therefore, they need a way to solve the problem while maintaining the privacy of their locations.

2. A country decides to bomb a location x in some other country; however, A does not want to hurt its relationship with its friends, who might have some areas of interests in the bombing region: for example, those countries might have secret businesses, secret military bases, or secret agencies in that area. Obviously, A does not want to disclose the exact location of x to all of its friends, except the one who will definitely be hurt by this bombing; on the other hand, its friends do not want to disclose their secret areas to A either, unless they are in the target area. How could they solve this dilemma? If each secret area is represented by a secret polygon, the problem becomes how to decide whether A's secret point is within B's polygon, where B represents one of the friend countries. If the point is not within the polygon, no information should be disclosed, including the information such as whether the location is at the west of the polygon, or within certain longitude or latitude. Basically it is "all-or-nothing": if one will be bombed, it knows all; otherwise it knows nothing.

5 Conclusions and Future Work

In this paper, we have considered several secure two-party computational geometry problems and presented some preliminary work for solving such problems. For the purpose of doing so, we have also presented two useful building blocks, Secure Two-Party Scalar Product Protocol and Secure Two-Party Vector Dominance Protocol.

In the protocols for the Point-Inclusion problem and the Intersection problem, we use an inefficient algorithm to decide whether a point is side a polygon (or whether two polygon intersect) although more efficient solutions exist, because of the concern about information disclosure. In our future work, we will study how to take advantage of those efficient solutions without degrading the privacy.

References

1. Wenliang Du, Mikhail J. Atallah and Florian Kerschbaum. Protocols for secure remote database access with approximate matching. *Submitted to a journal*, 2001.
2. J. Benaloh. Dense probabilistic encryption. In *Proceedings of the Workshop on Selected Areas of Cryptography*, pages 120–128, Kingston, ON, May 1994.
3. C. Cachin. Efficient private bidding and auctions with an oblivious third party. In *Proceedings of the 6th ACM conference on Computer and communications security*, pages 120–127, Singapore, November 1-4 1999.

4. G. Brassard, C. Crépeau and J. Robert. All-or-nothing disclosure of secrets. In *Advances in Cryptology - Crypto86, Lecture Notes in Computer Science*, volume 234-238, 1987.
5. Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative scientific computations. In *14th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 11-13 2001.
6. O. Goldreich. Secure multi-party computation (working draft). Available from http://www.wisdom.weizmann.ac.il/home/oded/public_html/foc.html, 1998.
7. S. Even, O. Goldreich and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28:637–647, 1985.
8. C. Cachin, S. Micali and M. Stadler. Computationally private information retrieval with polylogarithmic communication. *Advances in Cryptology: EUROCRYPT '99, Lecture Notes in Computer Science*, 1592:402–414, 1999.
9. O. Goldreich, S. Micali and A. Wigderson. How to play any mental game. In *Proceedings of the 19th annual ACM symposium on Theory of computing*, pages 218–229, 1987.
10. D. Naccache and J. Stern. A new cryptosystem based on higher residues. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 59–66, 1998.
11. M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation (extended abstract). In *Proceedings of the 31th ACM Symposium on Theory of Computing*, pages 245–254, Atanta, GA, USA, May 1-4 1999.
12. R. Fagin, M. Naor and P. Winkler. Comparing information without leaking it. *Communication of the ACM*, 39:77–85, 1996.
13. T. Okamoto and S. Uchiyama. An efficient public-key cryptosystem. In *Advances in Cryptology – EUROCRYPT 98*, pages 308–318, 1998.
14. P. Paillier. Public-key cryptosystems based on composite degree residue classes. In *Advances in Cryptology – EUROCRYPT 99*, pages 223–238, 1999.
15. A.C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science*, 1982.
16. A.C. Yao. How to generate and exchange secrets. In *Proceedings 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.

The Grid Placement Problem*

Prosenjit Bose[1], Anil Maheshwari[1], Pat Morin[2], and Jason Morrison[1]

[1] School of Computer Science, Carleton University
Ottawa, ON, Canada, K1S 5B6
{bose, maheshwa, morrison}@scs.carleton.ca
[2] School of Computer Science, McGill University
Montréal, Québec, Canada, H3A 2K6
morin@cgm.cs.mcgill.ca.

Abstract. We consider the problem of placing a regular grid over a set of points in order to minimize (or maximize) the number of grid cells not containing any points. We give an $O(n \log n)$ time and $O(n)$ space algorithm for this problem. As part of this algorithm we develop a new data structure for solving a problem on intervals that is interesting in its own right and may have other applications.

1 Introduction

The grid placement problem relates the positioning of a regular rectangular grid over a pointset. The focus of the problem is to place the grid and then count the number of grid cells containing at least one point. The problem has two variations. In the first variation the task is to calculate the grid position which minimizes the number of empty cells. The second variation calculates the position which maximizes the number of empty cells. Figure 1 shows an example solution for both problems.

This grid placement problem arises in *gridding/interpolation* where one attempts to derive a regular grid of data points from an irregular set of sample points. In this setting, one wants to minimize the number of grid cells not containing any sample point.

Another example application of this problem is in the layout of maps into books. In this case each grid cell corresponds to a page in the book and each point represents map features. In this case, minimizing the number of empty cells decreases the number of features on many pages. This in turn makes the pages easier to label and reduces the number of uninteresting pages.

In this paper, we give an $O(n \log n)$ time algorithm for solving the grid placement problem for a set of n points and a grid of arbitrary size. As part of this algorithm we develop a new data structure for solving a problem on intervals that is interesting in its own right and may have other applications.

* This research was partly supported the Natural Sciences and Engineering Research Council of Canada, the Government of Ontario and by NCE–GEOIDE. An extended abstract of this paper was presented at the European Workshop on Computational Geometry 2001 in Berlin.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 180–191, 2001.

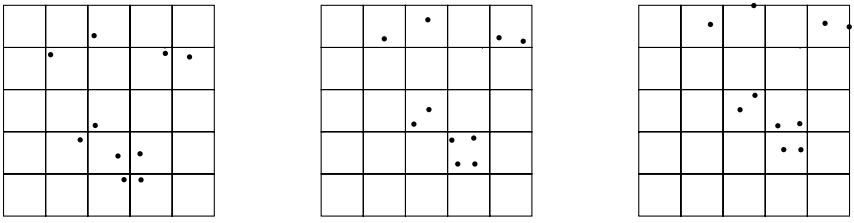


Fig. 1. A grid placement (a) with the minimum number of empty cells, (b) with the maximum number of empty cells and (c) in standard position.

In previous related work, by Asano and Tokuyama [2] and Agarwal et al. [1] the problem was to minimize the maximum number of points in a given cell. In the problem that was considered the grid was allowed to rotate and change in size. Asano and Tokuyama produced a result of $O(b^2n^3 \log n)$ time where b^2 is the number of grid cells. Agarwal et al. gave a randomized algorithm which works in $\tilde{O}(\min(B^{\frac{1}{2}}n^{\frac{5}{3}} \log^{\frac{7}{3}} n + B^{\frac{3}{2}}\Delta n \log^3 n, n^2))$ time where the maximal number of points in a cell is $\frac{n}{B} + \Delta$ and B is the number of grid cells. It should be noted that in both of these cases the only grids considered had an equal number of rows and columns.

In the following sections we give a discussion of the solution to the minimization problem and leave the trivial extension to the maximization problem to the reader. Our discussion formally defines the problem and describes and proves correctness of a basic algorithm. This basic algorithm is then refined into our final solution and analyzed.

2 Problem Definition and Analysis

We start by defining the set of n points in the problem to be the points $P = \{p_1, p_2, \ldots, p_n\}$ embedded in the plane. The location of a point p_i in the plane is represented as (p_{ix}, p_{iy}). A grid is defined as a packed tiling of regular rectangles called cells. The sides for each of the cells are assumed to be parallel to the x and y axes of the plane and have dimensions (c_x, c_y). A specific instance of a grid, G is defined by its size and its position. The size of G is defined by the number of cell columns (c) and rows (r). The *position* of the grid (G_x, G_y) is the location of the cell corner which has minimal x and y coordinates (i.e., the lower left corner). A grid position is allowable if and only if every point of P is contained in some cell of G.

There are a variety of allowable grid positions relative to the pointset P. In standard position (G_{sx}, G_{sy}), as shown in Figure 1c, the uppermost and rightmost points in P are coincident with the uppermost and rightmost boundaries of G, respectively. We finish the problem definition by assuming that in the standard position the leftmost column and bottommost row of cells do not contain

any points. Later we will show how to remove this assumption. Thus the problem is to find an allowable grid position which minimizes the number of cells that do not contain points from P.

Examining allowable grid positions yields the interesting result given in Lemma 1.

Lemma 1. *Given two distinct allowable positions* $a = (a_x, a_y)$ *and* $b = (a_x + h_x, a_y + h_y)$ *where* $h_x = ic_x$, $h_y = jc_y$ *and* $\{i, j\} \in \mathcal{N}$, *the number of non-empty cells is identical for both positions.*

Proof: Since both positions are allowable the grid at both positions a and b must contain all points. This further implies that the grid located at a must intersect the grid located at b and that only intersecting cells can contain points. Also, since i and j are integers then any cell can only intersect at most one other cell. Thus, the sets of cells which overlap form a one-to-one correspondence. This establishes a bijection between non-empty cells at a and non-empty cells at b. Therefore there are the same number of non-empty cells at both locations. □

Using Lemma 1 it is possible to limit the number of allowable grid positions that need to be considered. The exact space of allowable grid positions is defined in Theorem 1.

Theorem 1. *The only allowable grid positions that need to be considered are the "kernel" positions defined by the bottom, left cell in standard position.*

Proof: The proof is divided into two parts. First that all of the kernel positions are allowable and second that all other allowable positions are equivalent to some kernel position. Given that the standard position is allowable, that the bottommost row is empty and that the leftmost column is also empty, it is clear that all the kernel positions are allowable.

It remains to be proven that all other allowable positions are equivalent to some kernel position. Let A be the set of allowable positions. By the definition of the standard position any translation of the grid down and/or to the left will result in at least one point p_i being outside of the grid. Thus all elements $a = (a_x, a_y)$ of A must be expressible by non-negative translation from the standard grid position. That is, $a = (G_{sx} + h_x, G_{sy} + h_y)$ where $\{h_x, h_y\} >= 0$. Given Lemma 1 the theorem follows trivially. □

3 Basic Algorithm

The algorithm itself consists of two main stages. In the first stage, we calculate the regions of the kernel for which each cell is empty. In the second stage, we overlay these empty regions and calculate a position contained in the minimum number of empty regions.

For the cell $C_{(i,j)}$[1] we define $P(C_{(i,j)})$ to be the subset of P contained in $C_{(i,j)}$ at standard position. Since only the kernel positions are considered, then

[1] Each cell $C_{(i,j)}$ is labeled by it's row and column with $C_{(1,1)}$ in the bottom left and $C_{(r,c)}$ in the upper right.

$C_{(i,j)}$ can only contain elements from $P(C_{(i,j)})$, $P(C_{(i+1,j)})$, $P(C_{(i,j+1)})$ and $P(C_{(i+1,j+1)})$. For $C_{(i,j)}$ to be empty at the grid position a, the cell must not include any element of any of these sets.

To understand the empty region for $C_{(i,j)}$ first consider only the set $P(C_{(i,j)})$. A point $X = (X_x, X_y)$ is dominated by the set $P(C_{(i,j)})$ if and only if there exists an element p_i of $P(C_{(i,j)})$ such that $p_{ix} > X_x$ and $p_{iy} > X_y$. The cell $C_{(i,j)}$ does not contain an element of $P(C_{(i,j)})$ if and only if the lower left corner of $C_{(i,j)}$ is not dominated by $P(C_{(i,j)})$. Thus the maximal elements [6,4] define the region in which the cell does not contain any points of $P(C_{(i,j)})$. By symmetry the same can be done for $P(C_{(i+1,j)})$, $P(C_{(i,j+1)})$ and $P(C_{(i+1,j+1)})$ and the intersection of these four regions define the empty region for $C_{(i,j)}$. See Figure 2 for an example.

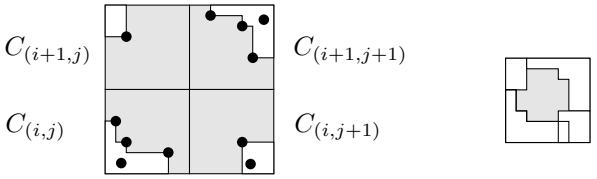


Fig. 2. Using maximal elements to compute the empty region for $C_{(i,j)}$.

The intersection of these four regions is done in a particular manner to avoid added complexity in the second stage. First, the appropriate set of maximal elements is calculated for each of the four cells[2]. Then the intersection of the regions for $P(C_{(i,j)})$ and $P(C_{(i+1,j)})$ is calculated followed by the intersection for the remaining two regions. The result of these intersections are two orthogonal convex polygons.

Each of these polygons is stored as a list corresponding to the events of a vertical sweep from the top to bottom of the kernel. At each event in the vertical sweep, the line interval resulting from the intersection of the sweep line and the polygon is stored. The vertical and horizontal position of the event and the corresponding width of the interval is stored with each of the intervals. It is assumed that the list of intervals is sorted by its vertical position. The two polygons are then intersected by merging the two lists to create a new list of intervals and their insertion and deletion heights. The results of such a process are displayed in Figure 3.

Suppose $|P(C_{(i,j)}) \cup P(C_{(i+1,j)}) \cup P(C_{(i,j+1)}) \cup P(C_{(i+1,j+1)})| = k_{(i,j)}$. The maximal elements can be computed in $O(k_{(i,j)} \log k_{(i,j)})$ time [6,4], and the intersection of the four empty regions can also be done in $O(k_{(i,j)})$ time as described above. Furthermore, any point in P is used in the computation of the empty regions for at most 4 grid cells. Therefore, the overall running time thus far is $O(\sum k_{(i,j)} \log k_{(i,j)}) = O(n \log n)$.

[2] If one of the pointsets to be calculated does not exist due to grid boundary then the pointset can be considered empty.

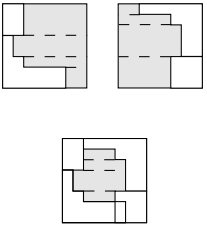


Fig. 3. The two preliminary intersections and the final result of intersecting the empty regions for $C_{(i,j)}$ including all intervals created.

In the second stage of the algorithm the empty regions for all of the cells have to be overlayed and a position with the minimum number of empty regions found. This could be done by computing an arrangement of rectilinear polygons, but this would take $\Omega(n^2)$ time and space. Instead, we perform a vertical plane sweep [3,7] of the kernel from top to bottom. There are two types of events during the sweep, the addition of an interval at it's maximum height and the deletion of an interval at it's minimum height. At each point in time we would like to keep track of a point that is contained in the least number of intervals. The minimum such value, over all points in time, corresponds to a point in the kernel that is contained in the least number of empty regions.

Since the problem has reduced to insertion and deletion of intervals in a sweep of the plane, then the second stage of the algorithm yields a solution by solving the following problem. Given that a set of one dimensional intervals I are inserted and deleted at times $\{t_1, t_2, t_3, \ldots, t_s\}$, find the time t and the location x where the minimum number of intervals overlap.

The algorithm will maintain a tree structure T which will answer queries about how many intervals are overlapping at a given x-coordinate and what the minimum overlapping intervals are. A complete binary search tree is created with a leaf for every x-coordinate in P plus the maximal x-coordinates for the kernel. The tree is assumed to have enough information to perform searches on x-coordinates ($right$, $left$, $parent(p)$, key, $\ldots$). Additionally, every vertex v in the tree will hold values $stab[v]$ and $ptr[v]$.

In a leaf vertex u, $stab[u]$ will contain the number of intervals that overlap at its corresponding x-coordinate. At an internal vertex $stab[v] = \min(stab[u])$ such that u is any leaf of the sub-tree rooted at v. Each vertex v also contains a pointer $ptr[v]$ to its child v' that has the minimum $stab$ value. In all future discussions, we will use the notation v' to denote the child of v with minimum $stab$ value. For all leaves $ptr[v] = v$.

With this structure an interval is inserted by incrementing $stab[u]$ at each leaf u whose key is contained by the interval (all intervals are closed). Then all ancestors v of the updated leaves are updated in a bottom up fashion. Deletion of an interval is done in an identical fashion by decrementing the necessary leaves.

For the purpose of our algorithm and its analysis each interval update is further broken into two updates. An interval $[x_1, x_2]$ is equivalent to $[x_1, \infty) \setminus$

(x_2, ∞). Therefore the insertion of $[x_1, x_2]$ is broken into inserting $[x_1, \infty)$ and deleting $[x_3, \infty)$ where x_3 is the minimal x-coordinate greater than x_2. Deletion of an interval is done similarly. Thus all of the interval updates which define the empty regions must be converted into two updates.

The algorithm is run by first sorting all of the interval updates by y-coordinate. Two null events are also added to occur at the maximal y-coordinates of the kernel. Any events with identical y-coordinate are then grouped together. The answer to the problem is then found by processing each group of events in order. After each group, excepting the last group, compare the known minimum value with the value at the root of the tree. If it is less than the current minimum then make it the new minimum. Also follow the chain of vertices connected by ptr values from $root[T]$ to vertex u. Record $key[u]$ as the x-Coordinate and the y-coordinate of that event group as the y-coordinate of the minimum.

4 Refined Algorithm

The main problem with the Basic Algorithm is the complexity of each interval update. For any given update $[x, \infty)$ let the vertex chain $V(x) = \{v_k, \ldots, v_1, v_0\}$ be the vertices along the path from $root[T] = v_k$ to the leaf v_0 whose $key[v] = x$. Then consider updating the tree T with $[x, \infty)$ and a value val ($val = 1$ for insertion, $val = -1$ for deletion). The basic algorithm requires every vertex in any right sub-tree of the path $V(x)$ be updated. The refined algorithm reduces the cost of updating by postponing the updating for any such sub-tree until absolutely necessary. This is achieved by performing an update on the root of a sub-tree and setting a mark bit for that root. This mark bit signifies that the node has been updated and its descendents have not.

In refining the data structure for T, a different algorithm is required to update the interval $[x, \infty)$ with a given value val. The refined $Update\_Interval$ algorithm is broken conceptually into two phases. In the first phase the path $V(x)$ is followed from root to leaf performing necessary updates to T (see lines 1-17 of Algorithm 1). These necessary updates involve unmarking all vertices on $V(x)$ and marking the root of all right sub-trees of $V(x)$. The algorithm then updates the leaf node v_0 by adding val to $stab[v_0]$.

The second phase of the algorithm then traverses the vertices $V(x) = v_i$ from leaf to root updating $ptr[v_i]$ appropriately and updating $stab[v_i]$ to be the value stored in the child of v_i that has the minimal $stab$ value (see lines 19-25 of Algorithm 1).

To prove the correctness of our algorithm we define the following variables:

$$\Delta_v = \begin{cases} stab[v], & \text{if } leaf[v] = TRUE; \\ stab[v] - stab[v'], & \text{if } mark[v] = TRUE; \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

$$STAB_i[x] = \begin{cases} \text{Num. of intervals overlapping } x \\ \text{after the } i^{th} \text{ interval update.} \end{cases} \tag{2}$$

Algorithm 1 *Update_Interval*(x, val) – Adds $[x, \infty)$ of value val to tree T

```
1: v ← root[T]
2: d ← 0
3: while leaf[v] = FALSE do
4:    if mark[v] = TRUE then
5:       Unmark(v) (see Algorithm 2)
6:    end if
7:    d ← d + 1
8:    if key[v] < x then
9:       v ← right[v]
10:   else {interval begins in the left sub-tree of v}
11:      if leaf[right[v]] = FALSE then
12:         mark[right[v]] ← TRUE
13:      end if
14:      v ← left[v]
15:   end if
16: end while
17: stab[v] ← stab[v] + val
18: while d ≠ 0 do
19:   if left[p[v]] = v then
20:      stab[right[p[v]]] ← stab[right[p[v]]] + val
21:   end if
22:   v ← p[v]
23:   d ← d − 1
24:   stab[v] ← stab[v']
25:   ptr[v] ← v'
26: end while
```

Algorithm 2 *Unmark*(v)

```
Require: leaf[v] = FALSE
1: δ ← stab[v] − stab[v']
2: stab[right[v]] ← stab[right[v]] + δ
3: stab[left[v]] ← stab[left[v]] + δ
4: if leaf[right[v]] = FALSE then
5:    mark[right[v]] ← TRUE
6:    mark[left[v]] ← TRUE
7: end if
8: mark[v] ← FALSE
```

Property 1. The tree T is said to have Property 1 after the i^{th} interval update if and only if Equation 3 is true where $V(x) = \{v_k, \ldots, v_1, v_0\}$ is a path of vertices from $root[T] = v_k$ to a leaf v_0 whose $key[v_0] = x$.

$$\sum_{j=0}^{k} \Delta_{v_j} = STAB_i[x]. \tag{3}$$

Lemma 2. *Given T such that Property 1 holds, then the property will still hold after performing procedure $Unmark(v)$ (see Algorithm 2) on any marked vertex v.*

Proof: Since $Unmark(v)$ only affects the *stab* values of $v, right[v]$ and $left[v]$ then only $\Delta_v, \Delta_{right[v]}$ and $\Delta_{left[v]}$ are affected. Since v is a marked vertex before $Unmark(v)$ is run and an unmarked vertex after then any sum involving Δ_v must decrease by Δ_v. However, $Unmark(v)$ also increases the values of $\Delta_{right[v]}$ and $\Delta_{left[v]}$ by the same amount and each sum must contain one of these two values. □

Theorem 2. *T always has Property 1 after the i^{th} interval update of Algorithm 1.*

Proof: The proof is by induction on i.

Base Case (After initialization): Before the first update there are no intervals in T. Thus if T is initialized with $mark[v] = FALSE$, $stab[v] = 0$ and $ptr[v] = left[v]$ then Property 1 is trivially true.

Induction Hypothesis: Assume that it is true for the i^{th} interval update.

$(i+1)^{st}$ Interval Update: We prove that Property 1 is true for the $(i+1)^{st}$ update in three cases, for vertices v with $key[v] = x$, $key[v] < x$ and $key[v] > x$.

The first case considers the leaf v_0 representing the leftmost portion of the interval being updated (i.e., $key[v_0] = x$). After line 16 all vertices in the path $V(x)$ are unmarked. It follows from the definition of Δ_v that $\Delta_v = 0, \forall v \in V(x)$ except v_0 before the $(i+1)^{st}$ update. By Lemma 2 the expression $\sum_{j=0}^{k} \Delta_{v_j} = STAB_i(x)$ must be true before line 17. Combining these two equations with Equation 1 yields that $\Delta_{v_0} = STAB_i(x) = stab[v_0]$. Thus, after line 17 $stab[v_0] = STAB_i(x) + val = STAB_{i+1}(x)$, which proves the first case.

The second case considers, without loss of generality any leaf u_0 with *key* value less than the leftmost portion of the interval being updated (i.e., $key[u_0] = w < x$). Again, it follows from the definition of Δ_v that $\Delta_v = 0$, $\forall v \in V(x)$ except v_0 before the $(i+1)^{st}$ update. By Lemma 2 the expression $\sum_{j=0}^{k} \Delta_{v_j} = STAB_i(x)$ must be true before line 17. Let the lowest common ancestor $lca(u_0, v_0) = u_h$. By definition of $lca()$, all vertices $\{u_h, u_{h+1}, \ldots, u_k\} \in V(x)$. Thus $\sum_{j=0}^{h-1} \Delta_{u_j} = STAB_i(w)$. Since vertices $\{u_{h-1}, u_{h-2}, \ldots, u_0\}$ are unaffected by the $(i+1)^{st}$ update, then after line 26: $\sum_{j=0}^{h-1} \Delta_{u_j} = \sum_{j=0}^{k} \Delta_{u_j} = STAB_i(w)$. Since $STAB_i(w) = STAB_{i+1}(w)$ then the second case is proven.

The third case follows the same logic as the second case with only one difference. On lines 19 and 20 the value of $stab[u_{h-1}]$ is incremented by val. Thus after line 26: $\sum_{j=0}^{h-1} \Delta_{u_j} = \sum_{j=0}^{k} \Delta_{u_j} = STAB_i(w) + val$. Since $STAB_{i+1}(w) = val + STAB_i(w)$ then the third case is proven. □

Lemma 3. *After the i^{th} interval update, the pointer $ptr[v]$ at any vertex v equals its child vertex v' with the minimum stab value.*

Proof: by induction on the number of interval updates(i).

Base Case (after initialization): Before the first update there are no intervals in T. Thus if T is initialized with $mark[v] = FALSE$, $stab[v] = 0$ and $ptr[v] = left[v]$ then Property 1 is trivially true.

Induction Hypothesis: Assume that it is true for the i^{th} interval update.

Proof of $(i+1)^{st}$ update: During the $(i+1)^{st}$ update the only vertices whose *stab* values change are the vertices on the path $V(x)$ or their children. Since the property holds after the i^{th} update the path vertices are the only vertices whose *ptr* values may need adjustment. Since these values are adjusted on line 25 then the property must be true for the $(i+1)^{st}$ update. □

Consider the path $v_k = root[T], v_{k-1} = ptr[v_k], \ldots, v_0 = ptr[v_1]$ and let $x = key[v_0]$. By Theorem 2 after the i^{th} insertion $STAB_i(x) = \sum_{j=0}^{k} \Delta_{v_j}$. Furthermore, by the definition of Δ_v and Lemma 3, $\Delta_{v_i} = stab(v_i) - stab(v_{i-1})$. Therefore,

$$STAB_i(x) = \sum_{j=0}^{k} \Delta_{v_j} \tag{4}$$

$$\Delta_{v_i} = stab[v_i] - stab[v_{i-1}] \tag{5}$$

By using Equations 4 and 5 together:

$$STAB_i(x) = stab[v_0] + \sum_{j=1}^{k} stab[v_i] - stab[v_{i-1}] \tag{6}$$

$$= stab[v_k] \tag{7}$$

Theorem 3. *Given the definition of v_k stated above, $stab[v_k]$ contains the appropriate minimal value after the i^{th} interval update.*

Proof: Consider any leaf vertex u_0 such that $key[u_0] = w$. By rearranging Equation 3 we arrive at Equation 10

$$STAB_i(w) = \sum_{j=0}^{k} \Delta_{u_j} \tag{8}$$

$$= stab[u_0] + \sum_{j=1}^{k} stab[u_j] - stab[u'_j] \tag{9}$$

$$= stab[u_k] + \sum_{j=0}^{k-1} stab[u_j] - stab[u'_{j+1}] \tag{10}$$

By the definition of u'_{j+1} the value $stab[u_j] - stab[u'_{j+1}]$ is minimized when $u_j = u'_{j+1}$. By Lemma 3 $ptr[u_{j+1}] = u'_{j+1}$ and therefore $STAB_i(x)$ is minimal. The proof is concluded because $stab[v_k] = STAB_i(x)$ by Equation 7. □

It is clear that the running time of *Update_Interval* is $O(\log n)$. Furthermore, Eq. (3) shows how to find a point overlapped by the least number of intervals in $O(\log n)$ time. Since these are exactly the operations required to solve the grid placement problem, we obtain our main result.

Theorem 4. *The grid placement problem can be solved in $O(n \log n)$ time and $O(n)$ space.*

All that remains to be shown is that the points in P can be preprocessed using this time and space to allow two specific types of queries. First, given a point in $P(C_{(i,j)})$, report all points in $P(C_{(i,j)})$ in $O(|P(C_{(i,j)})|)$ time. Second, given a point in $P(C_{(i,j)})$, report points in $P(C_{(i,j+1)})$, $P(C_{(i+1,j)})$ and $P(C_{(i+1,j+1)})$ in $O(1)$ time, returning null pointers should any of those sets be empty.

To perform this task we use a linked-list of points in which each point will have pointers to points in the cells above and to the right of its cell. As described previously these pointers may be null if the cells mentioned are empty. If the list is sorted to contain points from any specific cell in a contiguous block then the first requirement is satisfied. This is trivially accomplished by sorting the points primarily by the row they fall into and secondly by the column they fall into. This sorting is achieved in $O(n \log n)$ time and the structure contains a constant amount of space per point so the $O(n)$ space requirement is also fulfilled.

It remains to be shown how to preprocess the points to obtain the required pointers to points in the neighbouring cells. This is accomplished by first sorting the points primarily by the column and then by the row they lie in. In this format points in vertically adjacent cells are in adjacent contiguous blocks in the list. By passing through the list and remembering a pointer to a point in the previous contiguous block it is possible to scan through the points once and set the appropriate pointers for the vertically adjacent cells.

The remaining pointers to the horizontally adjacent cells can be computed similarly by sorting the list by row and then column. This method requires $O(n \log n)$ time and $O(n)$ space thereby proving Theorem 4.

5 Removing the Assumption

Finally we address the cases wherein the bottom row and leftmost column of the grid G are not necessarily empty in standard position. The algorithm still applies, however the kernel of allowable positions changes. Instead of mapping allowable positions into the entire bottom/leftmost grid cell $C_{(1,1)}$ the allowable positions map into a sub-region R of $C_{(1,1)}$. It is easily shown that there are only three configurations for R which are all shown in Figure 4.

First the algorithm must calculate R using the point(s) $p_{min(x)}$ and $p_{min(y)}$ which have the least x and y-coordinates respectively. These points are then used to calculate the region R as shown in Figure 4. During the algorithm the

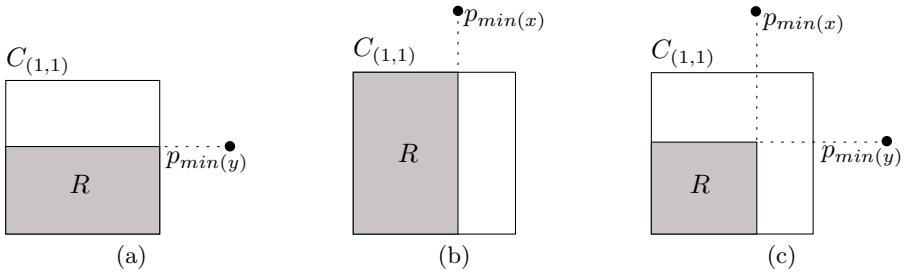


Fig. 4. The two points $p_{min(x)}$ and $p_{min(y)}$ define three possible configurations for the region R. (a) column 1 is empty, (b) row 1 is empty and, (c) neither are empty.

empty regions are calculated as described previously. Then the list intervals are modified to only include those portions which intersect R. The structure T is then based solely on x-coordinates which appear in the modified interval list.

Since all of these operations require linear time in the number of points and involve at most constant additional space none of the previous theorems are changed.

6 Conclusions

We have studied the grid placement problem and given an $O(n \log n)$ time and $O(n)$ space algorithm. This was accomplished by reducing it to the following data structuring problem: Maintain a set of intervals with endpoints drawn from a set of $O(n)$ points under insertions and deletions so that at any point in time a point included in the least (or most) number of intervals can be reported in $O(\log n)$ time. We have developed a new lazy data structure for this problem that requires $O(n)$ space and operates in $O(\log n)$ time per operation (insert, delete and query).

Since the acceptance of this paper, the authors have discovered a similar work by Lee [5] on the interval problem. In the paper, Lee solves the maximum clique problem on an interval graph. That is, his structure reports the maximum number of overlapping intervals. In our interval problem we also provide a point contained in the intersection of these intervals. There is no clear indication of how this would be accomplished using Lee's structure. It can be achieved by reporting all of the intervals contained in the maximum clique and calculating their intersection. However, the algorithm that Lee sketches for reporting the maximum clique requires $O(n \log n)$ space and each query requires $\Omega(k + \log n)$ time where k is the size of the maximum clique.

References

1. P. Agarwal, B. Bhattacharya, and S. Sen. Output-sensitive algorithms for uniform partitions of points. In *Proceedings of the 10th International Symposium on Algorithms and Computation*, 1999.

2. T. Asano and T. Tokuyama. Algorithms for projecting points to give the most uniform distribution with applications to hashing. *Algorithmica*, 9:572–590, 1993.
3. J. L. Bentley. Algorithms for Klee's rectangle problems. Carnegie-Mellon University, Department of Computer Science, unpublished notes, 1977.
4. H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *JACM*, 22(4):469–476, 1975.
5. D. T. Lee. Maximum clique problem of rectangle graphs. In F. Preparata, editor, *Advances in Computing Research*, volume 1, pages 91–107. JAI Press Inc., 1983.
6. F. P. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985.
7. J. van Leeuwen and D. Wood. The measure problem for rectangular ranges in d-space. *J. Algorithms*, 2:282–300, 1981.

On the Reflexivity of Point Sets

Esther M. Arkin[1], Sándor P. Fekete[2], Ferran Hurtado[3], Joseph S.B. Mitchell[1], Marc Noy[3], Vera Sacristán[3], and Saurabh Sethia[4]

[1] Department of Applied Mathematics and Statistics, State University of New York, Stony Brook, NY 11794-3600, USA, {estie, jsbm}@ams.sunysb.edu.
[2] Department of Mathematical Optimization TU Braunschweig, Pockelsstr. 14, D-38106 Braunschweig, Germany, sandor.fekete@tu-bs.de.
[3] Departament de Matemàtica Aplicada II, Universitat Politècnica de Catalunya, Pau Gargallo, 5, E-08028, Barcelona, Spain, {hurtado, noy, vera}@ma2.upc.es.
[4] Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA, saurabh@cs.sunysb.edu.

Abstract. We introduce a new measure for planar point sets S. Intuitively, it describes the combinatorial distance from a convex set: The *reflexivity* $\rho(S)$ of S is given by the smallest number of reflex vertices in a simple polygonalization of S. We prove various combinatorial bounds and provide efficient algorithms to compute reflexivity, both exactly (in special cases) and approximately (in general). Our study naturally takes us into the examination of some closely related quantities, such as the *convex cover* number $\kappa_1(S)$ of a planar point set, which is the smallest number of convex chains that cover S, and the *convex partition* number $\kappa_2(S)$, which is given by the smallest number of disjoint convex chains that cover S. We prove that it is NP-complete to determine the convex cover or the convex partition number, and we give logarithmic-approximation algorithms for determining each.

1 Introduction

In this paper, we study a fundamental combinatorial property of a discrete set, S, of points in the plane: What is the minimum number, $\rho(S)$, of *reflex vertices* among all of the *simple polygonalizations* of S? A *polygonalization* of S is a closed tour on S whose straight-line embedding in the plane defines a connected cycle without crossings, i.e., a simple polygon. A vertex of a simple polygon is *reflex* if it has interior angle greater than π. We refer to $\rho(S)$ as the *reflexivity* of S.

In general, there are many different polygonalizations of a point set, S. There is always at least one: simply connect the points in angular order about the center of mass. A set S has precisely one polygonalization, if and only if it is in convex position, but usually there is a great number of them. Studying the set of polygonalizations (e.g., counting them, enumerating them, or generating a random element) is a challenging active area of investigation in computational geometry.

The reflexivity $\rho(S)$ quantifies, in a combinatorial sense, the degree to which the set of points S is in convex position. See Figure 1 for an example.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 192-204, 2001.

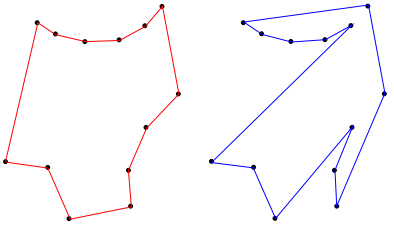


Fig. 1. Two polygonalizations of a point set, one (left) using 7 reflex vertices and one (right) using only 3 reflex vertices.

We conduct a formal study of reflexivity, both in terms of its combinatorial properties and in terms of an algorithmic analysis of the complexity of computing it, exactly or approximately. Some of our attention is focussed on the closely related *convex cover number* of S, which gives the minimum number of convex chains (subsets of S in convex position) that are required to cover all points of S. For this question, we distinguish two cases: The *convex cover number*, $\kappa_1(S)$, is the smallest number of convex chains to cover S; the *convex partition number*, $\kappa_2(S)$, is the smallest number of convex chains with pairwise-disjoint convex hulls to cover S. (Note that nested chains are feasible for a convex cover, but not for a convex partition.)

Motivation. In addition to the fundamental nature of the questions and problems we address, we are also motivated to study reflexivity for several other reasons:

(1) An application motivating our original investigation is that of meshes of low stabbing number and their use in performing ray shooting efficiently. If a point set S has low reflexivity or convex partition number, then it has a triangulation of low stabbing number, which is much lower than the general $O(\sqrt{n})$ upper bound guaranteed to exist ([1,12]). (e.g., a constant reflexivity implies a logarithmic stabbing number triangulation)

(2) Classifying point sets by their reflexivity may give us some structure for dealing with the famously difficult question of counting and exploring the set of all polygonalizations of S. See [11] for some references to this problem.

(3) There are several applications in computational geometry in which the number of reflex vertices of a polygon can play an important role in the complexity of algorithms. If one or more polygons are *given* to us, there are many problems for which more efficient algorithms can be written with complexity in terms of "r" (the number of reflex vertices), instead of "n" (the total number of vertices), taking advantage of the possibility that we may have $r \ll n$ for some practical instances. (See, e.g., [13,16].) The number of reflex vertices also plays an important role in convex decomposition problems for polygons (see, e.g., [17]).

(4) Reflexivity is intimately related to the issue of convex cover numbers, which has roots in the classical work of Erdős and Szekeres [8,9], and has been studied more recently by Urabe [22,23].

(5) Our problems are related to some problems in curve (surface) reconstruction, where the goal is to obtain a "good" polygonalization of a set of sample points. (See [3,5,6].)

Related Work. The study of convex chains in finite planar point sets is the topic of classical papers by Erdős and Szekeres [8,9], who showed that any point set of size n has a convex subset of size $t = \Omega(\log n)$. This is closely related to the convex cover number κ_1, since it implies an asymptotically tight bound on $\kappa_1(n)$, the worst-case value for sets of size n. There are still a number of open problems related to the exact relationship between t and n; see, for example, [21] for recent developments. Other issues have been considered, such as the existence and computation (see [7]) of large "empty" convex subsets (i.e., with no points of S interior to their hull); this is related to the convex partition number, $\kappa_2(S)$. It was shown by Horton [15] that there are sets with no empty convex chain larger than 6, so this implies that $\kappa_2(n) \geq n/6$. Tighter worst-case bounds were given by Urabe [22,23].

Another possibility is to consider a simple polygon having a given set of vertices, that is "as convex as possible". This has been studied in the context of TSP tours of a point set S, where convexity of S provides a trivial optimal tour. Convexity of a tour can be characterized by two conditions. If we drop the global condition (i.e., no crossing edges), but keep the local condition (i.e., no reflex vertices), we get "pseudoconvex" tours. In [10] it was shown that any set with $|S| \geq 5$ has such a pseudoconvex tour. It is natural to require the global condition of simplicity instead, and minimize the number of local violations – i.e., the number of reflex vertices. As in the paper [10], this draws a close connection to angles in a tour, a problem that has also been studied by Aggarwal et al. [2]. We will see in Section 4 that there are further connections.

The number of polygonalizations on n points is in general exponential in n. To give tight bounds on the maximum value attainable for a given n has also been object of intensive research ([11]). The minimum number of reflex vertices among all the polygonalizations of a point set S is the *reflexivity* of S, a concept we introduce in this work.

Main Results. The main results of this work include:

- Tight bounds on the worst-case reflexivity in a number of cases, including the general case and the case of onion depth 2.
- Upper and lower bounds on reflexivity, convex cover number, convex partition number, and their relative behavior. We obtain exact worst-case values for small cardinalities.
- Proofs of NP-completeness for computing convex cover and convex partition numbers.
- Algorithmic results yielding $O(\log n)$ approximations for convex cover number, convex partitioning number, and (Steiner) reflexivity. We also give efficient exact algorithms for cases of low reflexivity.

Throughout this extended abstract we omit many proofs and details, due to space limitations. We refer the reader to the full paper, available on the internet.

2 Preliminaries

Throughout this paper, S will be a set of n points in the plane $\Re^2$. Let P be any polygonalization of S. We say that P is *simple* if edges may only share common endpoints, and each endpoint is incident to exactly two edges. Let $\mathcal{P}$ be the set of all polygonalizations of S. Note that $\mathcal{P}$ is not empty, since any point set S having $n \geq 3$ points has at least one polygonalization (e.g., the star-shaped polygonalization obtained by sorting points of S angularly about a point interior to the convex hull of S).

A simple polygon P is a closed Jordan curve, subdividing the plane into an unbounded and a bounded component. We say that the bounded component is the *interior* of P. A *reflex vertex* of a simple polygon is a common endpoint of two edges, such that the interior angle between these edges is larger than π. We say that an angle is *convex* if it is not reflex. We define $r(P)$ to be the number of reflex vertices in P, and $c(P)$ to be the number of non-reflex, i.e., convex vertices in P. We define the *reflexivity* of a planar point set S to be $\rho(S) = \min_{P \in \mathcal{P}} r(P)$. Similarly, the *convexivity* of a planar point set S is defined to be $\chi(S) = \max_{P \in \mathcal{P}} c(P)$. Note that $\chi(S) = n - \rho(S)$.

The *convex hull* $CH(S)$ of a set S is the smallest convex set that contains all elements of S; the convex hull elements of S are the members of S that lie on the boundary of the convex hull. The *layers* of a point set S are given by repeatedly removing all convex hull elements, and considering the convex hull of the remaining set. We say that S has k *layers* or *onion depth* k if this process terminates after precisely k layers. A set S forms a *convex chain* (or is in *convex position*) if it has only one layer. A *Steiner point* is a point not in the set S that may be added to S in order to improve some structure of S. We define the *Steiner reflexivity* $\rho'(S)$ to be the minimum number of reflex vertices of any simple polygon with vertex set $V \supset S$. Similarly, we can define the *Steiner convexivity* $\chi'(S)$. (Furthermore, Steiner points can be required to lie within the convex hull, or be arbitrary. In this abstract, we do not elaborate on this difference.)

We use the notation $\max_{S:|S|=n} \rho(S) = \rho(n)$ and $\chi(n) = \min_{S:|S|=n} \chi(S)$ for the worst-case values for point sets of size n. For a given finite set S, let $\mathcal{C}_1$ be the family of all sets of convex chains, such that each element of S is part of at least one chain. We say that a set of chains $C \in \mathcal{C}_1$ is called a *convex cover* of S. Similarly, $\mathcal{C}_2 \subset \mathcal{C}_1$ is the family of all convex covers of S for which the convex hulls of any two chains are mutually disjoint. Then we define the *convex cover number* $\kappa_1(S) = \min_{C \in \mathcal{C}_1} |C|$ as the smallest size of a convex cover of S, and the *convex partition number* $\kappa_2(S) = \min_{C \in \mathcal{C}_2} |C|$. Again, we denote by $\kappa_1(n)$ and $\kappa_2(n)$ the worst-case values for sets of size n.

Finally, we state a basic property of polygonalizations of point sets. The proof is straightforward.

Lemma 1. *In any polygonalization of S, the points on the convex hull of S are always convex vertices, and they occur in the polygonalization in the same order in which they occur along the convex hull.*

3 Combinatorial Bounds

In this section we establish several combinatorial results on reflexivity and convex cover numbers.

One of our main combinatorial results establishes an upper bound on the reflexivity of S that is tight in terms of the number n_I of points *interior* to the convex hull of S. Given that the points of S that are vertices of the convex hull are required to be convex vertices in any (non-Steiner) polygonalization of S, the bound in terms of n_I seems to be quite natural.

Theorem 1. *Let S be a set of n points in the plane, n_I of which are interior to the convex hull $CH(S)$. Then, $\rho(S) \leq \lceil n_I/2 \rceil$.*

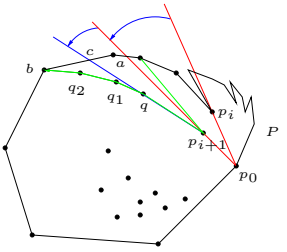


Fig. 2. Computing a polygonalization with at most $\lceil n_I/2 \rceil$ reflex vertices.

Proof. We describe a polygonalization in which at most half of the interior points are reflex. We begin with the polygonalization of the convex hull vertices that is given by the convex polygon bounding the hull. We then iteratively incorporate other (interior) points of S into the polygonalization. Fix a point p_0 that lies on the convex hull of S. At a generic step of the algorithm, the following invariants hold: (1) our polygonalization consists of a simple polygon, P, whose vertices form a subset of S; and (2) all points $S' \subset S$ of S that are not vertices of P lie interior to P; in fact, the points S' all lie within the subpolygon, Q, to the left of the diagonal p_0p_i, where p_i is a vertex of P such that the subchain of ∂P from p_i to p_0 (counter-clockwise) together with the diagonal p_0p_i forms a convex polygon (Q). Define p_{i+1} to be the first point of S' that is encountered when sweeping the ray p_0p_i counter-clockwise about its endpoint p_0. Then, we sweep the subray with endpoint p_{i+1} further counter-clockwise, about p_{i+1}, until we encounter another point, q, of S'. (If $|S'| = 1$, we can readily incorporate p_{i+1} into the polygonalization, increasing the number of reflex vertices by one.) Now, the ray $p_{i+1}q$ intersects the boundary of P at some point $c \in ab$ on the boundary of Q.

We now modify P to include interior points p_{i+1} and q (and possibly others as well) by replacing the edge ab with the chain $(a, p_{i+1}, q, q_1, \ldots, q_k, b)$, where

the points q_i are interior points that occur along the chain we obtain by "pulling taut" the chain (q, c, b) (i.e., by continuing, in the "gift wrapping" fashion, to rotate rays counter-clockwise about each interior point q_i that is hit until we encounter b). In this way we incorporate at least two new interior points (of S') into the polygonalization P, while creating only one new reflex vertex (at p_{i+1}). It is easy to check that the invariants (1) and (2) hold after this step. □

In fact, the upper bound of Theorem 1, $\rho(S) \leq \lceil n_I/2 \rceil$, is *tight*, as we now argue based on the special configuration of points, $S = S_0(n)$, in Figure 3. The set $S_0(n)$ is defined for any integer $n \geq 6$, as follows: $\lceil n/2 \rceil$ points are placed in convex position (e.g., forming a regular $\lceil n/2 \rceil$-gon), forming the convex hull $CH(S)$, and the remaining $n_I = \lfloor n/2 \rfloor$ interior points are also placed in convex position, each one placed "just inside" $CH(S)$, near the midpoint of an edge of $CH(S)$. The resulting configuration $S_0(n)$ has two layers in its convex hull. Lemma 2, below, shows that $\rho(S_0(n)) \geq \lceil n_I/2 \rceil \geq \lfloor n/4 \rfloor$.

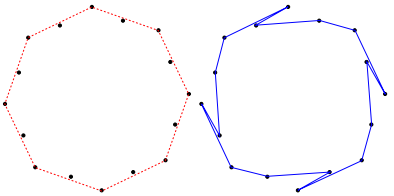


Fig. 3. Left: The configuration of points, $S_0(n)$, which has reflexivity $\rho(S_0(n)) \geq \lceil n_I/2 \rceil$. Right: A polygonalization having $\lceil n_I/2 \rceil$ reflex vertices.

Lemma 2. *For any* $n \geq 6$, $\rho(S_0(n)) \geq \lceil n_I/2 \rceil \geq \lfloor n/4 \rfloor$.

Proof. Denote by x_i the points on the convex hull, $i = 1, ..., \lceil n/2 \rceil$, and v_i the points "just inside" the convex hull, $i = 1, ..., \lfloor n/2 \rfloor$, where v_i is along the convex hull edge (x_i, x_{i+1}).

From Lemma 1 we know that points x_i are connected in their order around the convex hull, and are all convex vertices in any polygonalization. Consider an arbitrary pocket of a polygonalization, having lid (x_j, x_{j+1}) and let m_j denote the number of interior points that go to this pocket (if the convex hull edge (x_j, x_{j+1}) belongs to the polygonalization, with a slight abuse of notation we can consider it as a pocket with $m_j = 0$). Observe that $m_j > 0$ implies that v_j belongs to this particular pocket. If $m_j = 1$ the pocket contains a single interior point, namely v_j, and then v_j is a reflex vertex in this polygonalization. To complete our proof we will show that if this pocket contains more interior points, among them only v_j will be a convex point.

We use the following simple fact: Given a set of points, all but one of which is in convex position, all polygonalizations of this set have a unique reflex vertex, namely the point not on the convex hull.

The pocket with lid (x_j, x_{j+1}) includes points x_j, x_{j+1}; if v_j is not the only interior point included in this pocket, then this pocket together with the lid is a polygon as in the simple fact. Therefore the polygon which is the pocket has only one reflex vertex, v_j, and when considered "inside-out" as a pocket of the original polygon, only v_j among the interior points is a convex vertex.

Therefore the number of reflex vertices in a pocket is in any case at least $\lceil m_j/2 \rceil$, and we have

$$\begin{aligned}\rho(S_0(n)) &\geq \sum \lceil m_j/2 \rceil \\ &\geq \left\lceil \sum (m_j/2) \right\rceil = \lceil n_I/2 \rceil \geq \lfloor n/4 \rfloor.\end{aligned}$$

□

Remark. Since $n_I \leq n$, the corollary below is immediate from the theorem. The gap in the bounds for $\rho(n)$, between $\lfloor n/4 \rfloor$ and $\lceil n/2 \rceil$, remains an intriguing open problem. While our combinatorial bounds are tight in terms of n_I (the number of points of S whose convexivity/reflexivity is not forced by the convex hull of S), they are not yet tight in terms of n.

Corollary 1. $\lfloor n/4 \rfloor \leq \rho(n) \leq \lceil n/2 \rceil$.

Steiner Points. If we allow Steiner points in the polygonalizations of S, the reflexivity of S may decrease; in fact, we have examples of point sets with reflexivity $\rho(S) = r$, where the introduction is reduced by a factor of 2 from the no-Steiner case: $\rho'(S) = r/2$. At this point, it is unclear whether this estimate characterizes a worst case. We believe it does:

Conjecture 1. For a set S of points in the plane, we have $\rho'(S) \geq \rho(S)/2$.

In terms of the cardinality n of S, we obtain the following combinatorial bounds; a proof can be found in the full version of the paper:

Theorem 2. *For a set S of n points in the plane, we have $\rho'(S) \leq \lceil n/3 \rceil$.*

By a careful analysis of our example in Figure 3, we can show the following:

Theorem 3. *If one only allows Steiner points that are interior to $CH(S)$, then any Steiner polygonalization of $S_0(n)$ has at least $\lceil n/4 \rceil$ reflex vertices.*

Two-Layer Point Sets. Let S be a point set that has onion depth 2. It is clear from our repeated use of the example in Figure 3 that this is a natural case that is a likely candidate for worst-case behavior. With a very careful analysis of this case, we are able to obtain tight bounds on the worst-case reflexivity in terms of n:

Theorem 4. *Let S be a set of n points having two layers. Then $\rho(S) \leq \lceil n/4 \rceil$, and this bound is tight.*

A proof can be found in the full version of our paper.

Furthermore, we can prove the following lower bound for polygonalizations with a very special structure that may be useful in an inductive proof in more general cases:

Lemma 3. *For a point set with onion depth 2 a polygonalization with at most* $\lceil \frac{n}{3} \rceil$ *reflex vertices exists such that none of the edges exist in the interior of the inner layer of the onion.*

Convex Cover Numbers. As we noted in the introduction, it was shown by Erdős and Szekeres [8,9] that any set of n points in the plane has a convex chain of size $O(\log n)$. Moreover, they have shown that there are sets of size $2^t + 1$ without a convex chain of $t + 3$ points. This implies the following:

Theorem 5. $\kappa_1(n) = \Theta(n/\log n)$.

Proof. For point sets with $\kappa_1(n) = \Omega(n/\log n)$, consider the sets constructed by Erdős and Szekeres. These have a largest chain of size $O(\log n)$, and the lower bound follows.

To see that there always is a cover with $O(n/\log n)$ chains, consider a greedy cover in the following way. Let $S_0 = S$, and for each S_i, remove a largest chain, yielding S_{i+1}. By the result of Erdős and Szekeres, each removed chain has size $\Omega(\log |S_i|)$; the lower bound for the size of S_i remains constant until $\lceil \log |S_i| \rceil$ decreases, i.e., until we have removed at least half of the points. Furthermore, any largest convex chain in S_i has at least 3 points, so the iteration must terminate after removing a series of chains of size at least 3. This yields a total number of at most $O(\sum_{i=3}^{\log n} \lceil \frac{n}{2^{\log n+1-i} i} \rceil) = O(\sum_{i=3}^{\log n} \lceil \frac{2^{i-1}}{i} \rceil)$ chains. A straightforward induction over q shows that $\sum_{i=3}^{q} \frac{2^{i-1}}{i} \leq 2\frac{2^q}{q}$, so the claim follows. □

Even better bounds are known for the disjoint cover number:

Theorem 6 (Urabe [22]).
$\lceil (n-1)/4 \rceil \leq \kappa_2(n) \leq \lceil 2n/7 \rceil$.

The lower bound can be seen directly from our Figure 3; the upper bound is the result of a detailed construction in [22].

Now we discuss the relationship between the different measures for a set S.

The ratio $\kappa_1(S)$ and $\kappa_2(S)$ for a set S may be as big as $O(n)$: Our example in Figure 3 has $\kappa_1(S) = 2$, but $\kappa_2(S) \geq n/4$. However, there is a very tight lower bound of $\kappa_2(S)$ in terms of $\rho(S)$:

Theorem 7. *For a planar set S, we have the estimates $\kappa_1(S) \leq \kappa_2(S) \leq \rho(S) + 1$, and these upper bounds are best possible.*

Proof. The upper bound for $\kappa_1(S)$ by $\kappa_2(S)$ is trivial. The upper bound for $\kappa_2(S)$ by $\rho(S)$ can be found in [4]. □

One can construct examples with $2\kappa_2(S) = \rho(S)$, which is the worst example we know of, see the full paper. On the other hand, it is a surprisingly difficult open problem to prove that there is *some* bounded ratio between $\kappa_2(S)$ and $\rho(S)$:

Conjecture 2. For a set S of points in the plane, we have $\rho(S) = O(\kappa_2(S))$.

However, it is not hard to see that the estimate $\rho'(S) = O(\kappa_2(S))$ holds (see the proof of Corollary 2), so a proof of Conjecture 2 would follow from the validity of Conjecture 1.

Small Point Sets. It is natural to consider the exact values of $\rho(n)$, $\kappa_1(n)$, and $\kappa_2(n)$ for small values of n. Table 1 below shows some of these values, which we obtained through (sometimes tedious) case analysis. Oswin Aichholzer has recently applied his software that enumerates point sets of size n of all distinct order types to verify our results; in addition, he has obtained the result that $\rho(10) = 3$. (Values of $n \geq 11$ seem to be intractable for enumeration.)

n	$\rho(n)$	$\kappa_1(n)$	$\kappa_2(n)$
≤ 3	0	1	1
4	1	2	2
5	1	2	2
6	2	2	2
7	2	2	2
8	2	2	3
9	3	3	3

Table 1. Worst-case values of ρ, κ_1, κ_2 for small values of n.

4 Complexity

Theorem 8. *It is NP-complete to decide whether for a planar point set S the convex partition number $\kappa_2(S)$ is below some threshold k.*

Proof. We give a reduction of Planar 3 Sat, which was shown to be NP-complete by Lichtenstein (see [18]). See Figure 4 for the overall proof idea, and the full paper for proof details. □

Theorem 9. *It is NP-complete to decide whether for a planar point set S the convex cover number $\kappa_1(S)$ is below some threshold k.*

Proof. Our proof uses a reduction of the problem 1-in-3 SAT. (It is inspired by the hardness proof for the Angular Metric TSP given in [2].) See Figure 5 for the proof idea, with bold lines corresponding to appropriate dense point sets. Proof details can be found in the full paper. □

So far, the complexity status of determining the reflexivity of a point set remains open. However, the close relationship between convex cover number and reflexivity leads us to believe the following:

Conjecture 3. It is NP-complete to determine the reflexivity $\rho(S)$ of a point set.

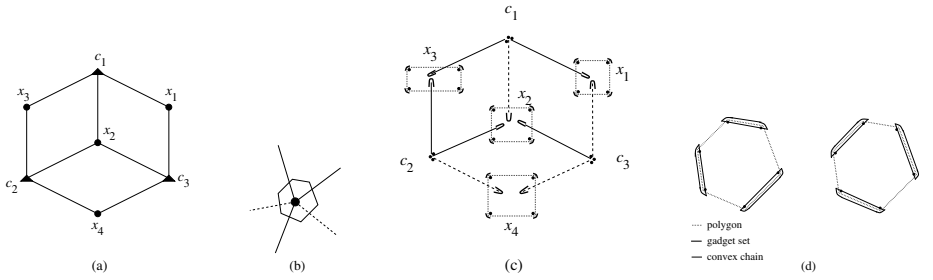


Fig. 4. (a) A straight-line embedding of the occurrence graph for the 3 SAT instance $(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_4})$; (b) a polygon for a variable vertex; (c) a point set S_I representing the PLANAR 3 SAT instance I; (d) joining point sets along the odd or even polygon edges.

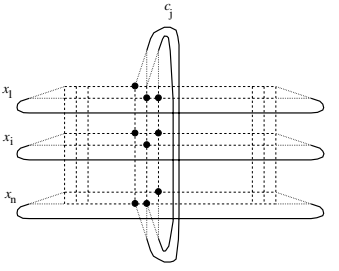


Fig. 5. A point set S_I for a 1-in-3 SAT instance I. Pivot points are shown for the clause $(x_1 \vee \overline{x_i} \vee x_n)$.

5 Algorithms

In this section, we provide a number of algorithmic results. Since some of the methods are rather technical, we can only give proof sketches in this extended abstract.

Theorem 10. *Given a set S of n points in the plane, in $O(n \log n)$ time one can compute a polygonalization of S having at least $\chi(S)/2$ convex vertices, where $\chi(S)$ is the convexivity of S.*

Proof. (sketch) The algorithm of Theorem 1 is constructive, producing a polygonalization of S having at most $n_I/2 \leq n/2$ reflex vertices, and thus at least $n/2$ convex vertices (thereby giving a 2-approximation for convexivity). In order to obtain the stated time bound, we must implement the algorithm efficiently. We utilize a dynamic convex hull data structure ([14]), to be able to obtain the points $q, q_1, \ldots, q_k$, efficiently (in amortized $O(\log n)$ time per point). □

Theorem 11. *Given a set S of n points in the plane, the convex cover number, $\kappa_1(S)$, can be computed approximately, within a factor of $O(\log n)$, in polynomial time.*

Proof. (sketch) We use a greedy set cover heuristic. At each stage, we need to compute a largest convex subset among the remaining (uncovered) points of S. This can be done in polynomial time using the dynamic programming methods of [20]. □

Theorem 12. *Given a set S of n points in the plane, the convex partition number, $\kappa_2(S)$, can be computed approximately, within a factor of $O(\log n)$, in polynomial time.*

Proof. (sketch) Let $C^* = \{P_1, \ldots, P_{k^*}\}$ denote an optimal solution, consisting of $k^* = \kappa_2(S)$ disjoint convex polygons whose vertices are the set S. Following the method of [19], we partition each of these polygons into $O(\log n)$ vertical trapezoids whose x-projection is a "canonical interval" (one of the $O(n)$ such intervals determined by the segment tree on S).

For the algorithm, we use dynamic programming to compute a minimum-cardinality partition of S into a disjoint set, C', of (empty) convex subsets whose x-projections are canonical intervals. Since the optimal solution, C^*, can be converted into at most $k^* \cdot O(\log n)$ such convex sets, we know we have obtained an $O(\log n)$-approximate solution to the disjoint convex partition problem. □

Corollary 2. *Given a set S of n points in the plane, its Steiner reflexivity, $\rho'(S)$, can be computed approximately, within a factor of $O(\log n)$, in polynomial time.*

A proof can be found in the full paper.

Special Cases. For small values of r, we have devised particularly efficient algorithms that check if $\rho(S) \leq r$ and, if so, produce a witness polygonalization having at most r vertices. Of course, the case $r = 0$ is trivial, since that is equivalent to testing if S lies in convex position (which is readily done in $O(n \log n)$ time, which is worst-case optimal). It is not particularly surprising that for any fixed r one can obtain an $n^{O(r)}$ algorithm, e.g., by enumerating over all r-element subsets that correspond to reflex vertices, along with all possible neighboring segments incident on these vertices, etc. The factor in front of r in the exponent, however, is not so trivial to reduce. In particular, the straightforward method applied to the case $r = 1$ gives $O(n^5)$ time. With a more careful analysis of the cases $r = 1, 2$, we obtain:

Theorem 13. *Given a set S of n points in the plane, in $O(n \log n)$ time one can determine if $\rho(S) = 1$, and, if so, produce a witness polygonalization. Furthermore, $\Omega(n \log n)$ is a lower bound.*

For $r = 2$, a careful analysis of how two pockets can interact also yields a very efficient algorithm; in the full paper we prove:

Theorem 14. *Given a set S of n points in the plane, in $O(n^2)$ time one can determine if $\rho(S) = 2$, and, if so, produce a witness polygonalization.*

Acknowledgments. We thank Adrian Dumitrescu for valuable input on this work. The collaboration between UPC and SUNY Stony Brook was made possible by a grant from the Joint Commission USA-Spain for Scientific and Technological Cooperation Project 98191. E. Arkin acknowledges support from the NSF (CCR-9732221) and HRL Laboratories. S. Fekete acknowledges travel support from the Hermann-Minkowski-Minerva Center for Geometry at Tel Aviv University. F. Hurtado, M. Noy, and V. Sacristán acknowledge support from CUR Gen. Cat. 1999SGR00356, and Proyecto DGES-MEC PB98-0933. J. Mitchell acknowledges support from HRL Laboratories, NSF (CCR-9732221), NASA (NAG2-1325), Northrop-Grumman, Sandia, Seagull Technology, and Sun Microsystems.

References

1. P. K. Agarwal. Ray shooting and other applications of spanning trees with low stabbing number. *SIAM J. Comput.*, **21**, 540–570, 1992.
2. A. Aggarwal, D. Coppersmith, S. Khanna, R. Motwani, and B. Schieber. The angular-metric traveling salesman problem. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 221–229, Jan. 1997.
3. N. Amenta, M. Bern, and D. Eppstein. The crust and the β-skeleton: Combinatorial curve reconstruction. *Graphical Models and Image Processing*, **60**, 125–135, 1998.
4. B. Chazelle. *Computational geometry and convexity.* Ph.D. thesis, Dept. Comput. Sci., Yale Univ., New Haven, CT, 1979. Carnegie-Mellon Univ. Report CS-80-150.
5. T. K. Dey and P. Kumar. A simple provable algorithm for curve reconstruction. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 893–894, Jan. 1999.
6. T. K. Dey, K. Mehlhorn, and E. A. Ramos. Curve reconstruction: Connecting dots with good reason. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 197–206, 1999.
7. D. P. Dobkin, H. Edelsbrunner, and M. H. Overmars. Searching for empty convex polygons. *Algorithmica*, **5**, 561–571, 1990.
8. P. Erdős and G. Szekeres. A combinatorial problem in geometry. *Compositio Math.*, **2**, 463–470, 1935.
9. P. Erdős and G. Szekeres. On some extremum problem in geometry. *Ann. Univ. Sci. Budapest*, **3-4**, 53–62, 1960.
10. S. P. Fekete and G. J. Woeginger. Angle-restricted tours in the plane. *Comp. Geom. Theory Appl.*, **8**, 195–218, 1997.
11. A. García, M. Noy, and J. Tejel. Lower bounds for the number of crossing-free subgraphs of K_n. In *Proc. 7th Canad. Conf. Comput. Geom.*, pages 97–102, 1995.
12. J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *J. Algorithms*, **18**, 403–431, 1995.
13. S. Hertel and K. Mehlhorn. Fast triangulation of the plane with respect to simple polygons. *Inf. Control*, **64**, 52–76, 1985.
14. J. Hershberger and S. Suri. Applications of a semi-dynamic convex hull algorithm. *BIT*, **32**, 249–267, 1992.
15. J. Horton. Sets with no empty convex 7-gons. *Canad. Math. Bull.*, **26**, 482–484, 1983.
16. F. Hurtado and M. Noy. Triangulations, visibility graph and reflex vertices of a simple polygon. *Comput. Geom. Theory Appl.*, **6**, 355–369, 1996.

17. J. M. Keil. Polygon decomposition. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 491–518. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
18. D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, **11**, 329–343, 1982.
19. J. S. B. Mitchell. Approximation algorithms for geometric separation problems. Technical report, Department of Applied Mathematics, SUNY Stony Brook, NY, July 1993.
20. J. S. B. Mitchell, G. Rote, G. Sundaram, and G. Woeginger. Counting convex polygons in planar point sets. *Inform. Process. Lett.*, **56**, 191–194, 1995.
21. J. Pach (ed.). *Discrete and Computational Geometry*, 19, Special issue dedicated to Paul Erdös, 1998.
22. M. Urabe. On a partition into convex polygons. *Discrete Appl. Math.*, **64**, 179–191, 1996.
23. M. Urabe. On a partition of point sets into convex polygons. In *Proc. 9th Canad. Conf. Comp. Geom.*, pages 21–24, 1997.

A $\frac{7}{8}$-Approximation Algorithm for Metric Max TSP

Refael Hassin[1] and Shlomi Rubinstein[1]

Department of Statistics and Operations Research,
School of Mathematical Sciences,
Tel-Aviv University, Tel-Aviv 69978, Israel.
{hassin,shlomiru}@post.tau.ac.il

Abstract. We present a randomized approximation algorithm for the metric undirected MAXIMUM TRAVELING SALESMAN PROBLEM. Its expected performance guarantee approaches $\frac{7}{8}$ as $n \to \infty$, where n is the number of vertices in the graph.

1 Introduction

Let $G = (V, E)$ be a complete (undirected) graph with vertex set V, $|V| = n$, and edge set E. Since we only deal with asymptotic bounds, we assume without loss of generality that n is even. For $e \in E$ let $w(e) \geq 0$ be its weight. For $E' \subseteq E$ we denote $w(E') = \sum_{e \in E'} w(e)$. For a random subset $E' \subseteq E$, $w(E')$ denotes the expected value. The MAXIMUM TRAVELING SALESMAN PROBLEM (Max TSP) is to compute a Hamiltonian circuit (a *tour*) with maximum total edge weight. If the weights $w(e)$ satisfy the triangle inequality, we call the problem METRIC MAXIMUM TRAVELING SALESMAN PROBLEM.

We denote the weight of an optimal tour by *opt*. In [1] a randomized polynomial algorithm is given for Max TSP that guarantees for any $r < \frac{25}{33}$ a solution of expected weight at least r *opt*. A paper by Kostochka and Serdyukov [2] contains an algorithm with a performance guarantee of $\frac{5}{6}$ for the metric Max TSP.

We build on ideas from a $\frac{3}{4}$-approximation algorithm for Max TSP by Serdyukov in [3] and a $\frac{5}{6}$-approximation algorithm for the metric case by Kostochka and Serdyukov [2]. We present a randomized approximation algorithm for the metric problem with expected performance guarantee that approaches $\frac{7}{8}$ as $n \to \infty$.

A *perfect matching* is a subgraph in which each vertex in V has a degree of exactly 1. A *cycle cover*, or *binary 2-matching*, is a subgraph in which each vertex in V has a degree of exactly 2. A *maximum cycle cover* is one with maximum total edge weight. The problem of computing a maximum cycle cover is a relaxation of Max TSP and therefore the weight of a maximum cycle cover is an upper bound on *opt*. A *subtour* in this paper is a subgraph with no non-Hamiltonian cycles or vertices of degree greater than 2.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 205–209, 2001.

2 Algorithms by Serdyukov and Kostochka

Serdyukov's algorithm for the general Max TSP starts by computing a maximum cycle cover $\mathcal{C} = \{C_1, ..., C_s\}$ and a maximum perfect matching M. Then it sequentially, for $i = 1, ..., s$, transfers from C_i to M an edge so that M remains a subtour. Finally it completes $\mathcal{C}$ into a tour T_1 and M into a tour T_2 and returns the tour with maximum weight between T_1 and T_2.

The crucial observation is that it is always possible to transfer an edge from C_i to M as required. The performance guarantee follows easily from $w(\mathcal{C}) \geq opt$ and $w(M) \geq \frac{1}{2}opt$. Thus, $w(T_1) + w(T_2) \geq \frac{3}{2}opt$ and therefore $\max\{w(T_1), w(T_2)\} \geq \frac{3}{4}opt$.

Algorithm Old_Metric given in Figure 1 is a randomized algorithm which resembles an algorithm by Kostochka and Serdyukov [2] for the metric Max TSP.

Old_Metric
input *A complete undirected graph $G = (V, E)$ with weights satisfying the triangle inequality.*
returns *A tour.*
begin
Compute a maximum cycle cover $\mathcal{C} = \{C_1, ..., C_s\}$.
Delete from each cycle $C_1, ..., C_s$ a minimum weight edge.
Let u_i and v_i be the ends of the path P_i that results from C_i.
Give each path a random orientation and form a tour T by adding connecting edges between the head of P_i and the tail of P_{i+1} ($P_{s+1} \equiv P_1$).
return *The tour T.*
end *Old_Metric*

Fig. 1. Old_Metric algorithm

Since $|C_i| \geq 3$:

$$w(u_i, v_i) \leq \frac{1}{3}w(C_i).$$

By the triangle inequality,

$$\begin{aligned} w(T) &= \sum_{i=1}^{s} w(P_i) \\ &\quad + \frac{1}{4}\sum_{i=1}^{s}(w(u_i, u_{i+1}) + w(v_i, v_{i+1}) + w(u_i, v_{i+1}) + w(v_i, u_{i+1})) \\ &\geq \sum_{i=1}^{s} w(P_i) + \frac{1}{2}\sum_{i=1}^{s} w(u_i, v_i) \end{aligned}$$

$$= \sum_{i=1}^{s} w(C_i) - \frac{1}{2} \sum_{i=1}^{s} w(u_i, v_i)$$
$$\geq \frac{5}{6} \sum_{i=1}^{s} w(C_i) \geq \frac{5}{6} opt.$$

3 A New Algorithm

Our algorithm is motivated by the following idea. The value of a random perfect matching in a graph with weights satisfying the triangle inequality is at least half the weight of a maximum perfect matching, and consequently at least a quarter of the weight of the longest tour. The total weight of a maximum cycle cover, a maximum perfect matching, and a random matching is therefore at least $\frac{14}{8} opt$. If the corresponding edges could be decomposed into two Hamiltonian cycles then the longer of these cycles would have weight of at least $\frac{7}{8} opt$. The problem with this approach is that these edges may not be decomposable into two tours. The algorithm which we propose below contains a modified approach which guarantees creation of two tours.

The proposed algorithm is given in Figure 2.

New_Metric
input *A complete undirected graph* $G = (V, E)$ *with weights* w_e $e \in E$ *satisfying the triangle inequality.*
returns *A tour.*
begin
Compute a maximum cycle cover $\mathcal{C} = \{C_1, ..., C_s\}$.
Compute a maximum perfect matching M.
for $i = 1, ..., s$:
Identify $e, f \in E \cap C_i$ *such that both* $M \cup \{e\}$ *and* $M \cup \{f\}$ *are subtours.*
Randomly choose $g \in \{e, f\}$ *(each with probability* $1/2$*).*
$P_i := C_i \setminus \{g\}$.
$M := M \cup \{g\}$.
end for
Complete $\cup_{i=1}^{s} P_i$ *into a tour* T_1 *as in Algorithm Old_Metric.*
Let $S :=$*set of end nodes of paths in* M.
Compute a random perfect matching M_S *over* S.
Delete an edge from each cycle in $M \cup M_S$.
Arbitrarily complete $M \cup M_S$ *into a tour* T_2.
return *The tour* T *with maximum weight between* T_1 *and* T_2.
end *New_Metric*

Fig. 2. New algorithm for metric Max TSP

Theorem 1. *The expected weight of the tour returned by Algorithm New_Metric satisfies* $w(T) \geq (\frac{7}{8} - O(\frac{1}{\sqrt{n}}))opt$.

Proof: As before, $w(\mathcal{C}) \geq opt$ and $w(M) \geq opt/2$.

The algorithm selects (sequentially, for $i = 1, ..., s$) a pair of edges, which we call *candidates*, from each cycle of $\mathcal{C}$ and deletes one of them, where the selection of the edge to be deleted is with probability 1/2. Let $\alpha w(\mathcal{C})$ be the weight of the edges that were candidates for deletion, $0 \leq \alpha \leq 1$. The expected weight of the edges that were actually deleted is $(\alpha/2)w(\mathcal{C})$. However, as in Algorithm Old_Metric, half of this weight is regained when connecting the resulting paths to a tour T_1. Hence,

$$w(T_1) \geq \Big[1 - \frac{\alpha}{2} + \frac{\alpha}{4}\Big] w(\mathcal{C}) \geq \Big(1 - \frac{\alpha}{4}\Big) opt. \tag{1}$$

The algorithm adds to M one of the pair of candidates from each cycle of $\mathcal{C}$. The expected weight of the added edges is $(\alpha/2)w(\mathcal{C})$. Note that if a vertex v is incident to two candidates then certainly $v \notin S$. If v is not incident to a candidate then certainly $v \in S$. Finally, if v is incident to one candidate then $v \in S$ with probability 1/2 (which is the probability that this candidate is *not* chosen).

Let $|S| = k + 1$. For $i \in S$, exactly one edge from $\{(i, j) | j \in S \setminus i\}$ is chosen to M_S. Thus, for an edge $(i, j) \in E \cap (S \times S)$ the probability that this edge will be selected to M_S is $1/k$. If (i, j) is selected, charge its weight $w(i, j)$ in the following manner: Suppose that i is incident to edges $e', e'' \in \mathcal{C}$. If none of these edges was a candidate, charge $w(i, j)/4$ to each of e' and e''. If one of them , say e' was a candidate, charge $w(i, j)/2$ to e'' (and nothing to e'). Note that it cannot be that both e' and e'' were candidates since in such a case $i \notin S$. The expected weight charged to an edge $(g, h) \in \mathcal{C}$ that was not a candidate is then $(1/k)[\sum_{r \in S \setminus g} w(r, g)/4 + \sum_{r \in S \setminus h} w(r, h)/4]$. Note that the 1/4 factor arises also in the case that the vertex, say g, is incident with a candidate edge on $\mathcal{C}$, since in such a case $g \in S$ with probability 1/2 and then it gets half of the weight of the edge of M_S which is incident to it. By the triangle inequality, $w(r, g) + w(r, h) \geq w(g, h)$ so that the above sum is at least $w(g, h)/4$. We conclude that $w(M_S) \geq w(\mathcal{C})(1 - \alpha)/4$ and consequently

$$w(M \cup M_S) \geq \Big(0.5 + \frac{\alpha}{2} + \frac{1 - \alpha}{4}\Big) w(\mathcal{C}) = \Big(\frac{3 + \alpha}{4}\Big) opt.$$

Finally, the algorithm deletes edges from cycles in $M \cup M_S$. We claim that $|S| \geq n/3$. The reason is that the perfect matching M computed by the algorithm had all n vertices of V with degree 1. Then, one candidate from each cycle of $\mathcal{C}$ was added to M. The number of added edges is equal to the number of cycles which is at most $n/3$. Therefore, after the addition of these edges the degrees of at most $2n/3$ vertices became 2, while at least $n/3$ vertices remained with degree 1. The latter vertices are precisely the set S, and this proves that $|S| \geq n/3$. Since M_S is a random matching, the probability that an edge of $M \cup M_S$ is

contained in a cycle whose size is smaller than $\sqrt{n}$ is at bounded from above by

$$\frac{1}{\frac{n}{3}} + \frac{1}{\frac{n}{3}-1} + \cdots + \frac{1}{\frac{n}{3}-\sqrt{n}} \leq \frac{\sqrt{n}}{\frac{n}{3}-\sqrt{n}} = O\Big(\frac{1}{\sqrt{n}}\Big).$$

(The j-th term in the left-hand side of this expression bounds the probability that a cycle containing exactly j edge from M_S is created.) Therefore, the expected weight of edges deleted in this step is $O(1/\sqrt{n})w(M \cup M_S)$ and

$$w(T_2) \geq \Big(\frac{3+\alpha}{4}\Big)\Big(1 - O\Big(\frac{1}{\sqrt{n}}\Big)\Big)opt. \tag{2}$$

Combining (1) and (2) we get that when $\alpha \leq 1/2$, $w(T_1) \geq (7/8)opt$ and when $\alpha \geq 1/2$ $w(T_2) \geq (7/8)(1 - O(1/\sqrt{n}))opt$. Thus,

$$w(T) = \max\{w(T_1), w(T_2)\} \geq \Big(\frac{7}{8} - O\Big(\frac{1}{\sqrt{n}}\Big)\Big)opt.$$

■

References

1. R. Hassin and S. Rubinstein, "Better approximations for Max TSP", *Information Processing Letters* **75** (2000), 181-186.
2. A. V. Kostochka and A. I. Serdyukov, "Polynomial algorithms with the estimates $\frac{3}{4}$ and $\frac{5}{6}$ for the traveling salesman problem of the maximum" (in Russian), *Upravlyaemye Sistemy* **26** (1985) 55-59.
3. A. I. Serdyukov, "An algorithm with an estimate for the traveling salesman problem of the maximum" (in Russian), *Upravlyaemye Sistemy* **25** (1984) 80-86.

Approximating Multi-objective Knapsack Problems

Thomas Erlebach[1], Hans Kellerer[2], and Ulrich Pferschy[2]

[1] Computer Engineering and Networks Laboratory, ETH Zürich, CH-8092 Zürich, Switzerland, erlebach@tik.ee.ethz.ch
[2] University of Graz, Department of Statistics and Operations Research, Universitätsstr. 15, A-8010 Graz, Austria, {hans.kellerer,pferschy}@uni-graz.at

Abstract. For multi-objective optimization problems, it is meaningful to compute a set of solutions covering all possible trade-offs between the different objectives. The multi-objective knapsack problem is a generalization of the classical knapsack problem in which each item has several profit values. For this problem, efficient algorithms for computing a provably good approximation to the set of all non-dominated feasible solutions, the Pareto frontier, are studied.
For the multi-objective 1-dimensional knapsack problem, a fast fully polynomial-time approximation scheme is derived. It is based on a new approach to the single-objective knapsack problem using a partition of the profit space into intervals of exponentially increasing length. For the multi-objective m-dimensional knapsack problem, the first known polynomial-time approximation scheme, based on linear programming, is presented.

1 Introduction

The knapsack problem is a classical problem in combinatorial optimization. In the original version of the problem, the input consists of a knapsack with a certain capacity and a set of items, each of which has a weight and a profit. A feasible solution is a selection of items that can be put into the knapsack, i.e., the sum of the weights of the selected items must not exceed the knapsack capacity. The goal is to maximize the total profit, i.e., the sum of the profits of the selected items. This knapsack problem is also called the 0–1 knapsack problem. It is *NP*-hard, but it was one of the first problems for which a fully polynomial-time approximation scheme was known.

In spite of its comparatively long history, variants of the knapsack problem are still being studied intensively. In this paper, we are interested in the multi-objective m-dimensional knapsack problem. This problem generalizes the classical knapsack problem in two respects: First, each item has t different profits, and instead of trying to compute a single solution with maximum profit, we want to compute a set of feasible solutions covering all possible trade-offs between the different profit values. Second, each item has m different weights,

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 210–221, 2001.

the knapsack has m different capacities, and the knapsack constraints must be satisfied for each of the m weights resp. capacities.

Zitzler and Thiele have presented experimental comparisons of various evolutionary algorithms for the multi-objective m-dimensional knapsack problem [13]. Contrary to their approach, we are interested in polynomial-time algorithms with provable approximation guarantees for this problem. We obtain a fully polynomial-time approximation scheme for the multi-objective 1-dimensional knapsack problem and a polynomial-time approximation scheme for the multi-objective m-dimensional knapsack problem.

Multi-objective optimization problems are frequently encountered in practice. Often there are several different criteria measuring the "quality" of a solution, and it is not possible to select a most important criterion or to combine the criteria into a single objective function. In the context of knapsack problems consider e.g. a government agency which has to choose a subset out of a given list of different projects subject to monetary restrictions. Each project requires a certain budget (possibly also human resources of the agency, office space, etc.) and yields a certain profit for different objectives such as employment effect, infrastructure, side effects for private economy, social effects and public opinion. In such applications, the decision maker may want to have an algorithm to compute a set of good solutions (instead of only one solution) with various trade-offs between the different criteria, so that she can select the most desirable solution after inspecting the various alternatives. A discussion of different concepts of multi-objective optimization can be found in the recent textbook by Ehrgott [1].

2 Preliminaries

2.1 Multi-objective Optimization

Let I be an instance of an optimization problem with t objectives. If S is a feasible solution, let $V_k(S)$ denote the value of S with respect to the k-th objective, $1 \leq k \leq t$. We assume throughout this paper that all objectives are maximization criteria and that the number t of objectives is a constant.

A feasible solution S_1 *weakly dominates* a feasible solution S_2 if $V_k(S_1) \geq V_k(S_2)$ for all $1 \leq k \leq t$. We say that a set $\mathcal{P}$ of feasible solutions for I is a *Pareto frontier* (sometimes also called *Pareto curve*) if, for any feasible solution S for I, $\mathcal{P}$ contains a solution that weakly dominates S. A set of feasible solutions is called *reduced* if it does not contain two different solutions S_1 and S_2 such that S_1 weakly dominates S_2. Usually, a Pareto frontier is assumed to be reduced.

For $\rho \geq 1$, a solution S_1 is called a *ρ-approximation* of a solution S_2 if $V_k(S_1) \geq V_k(S_2)/\rho$ for all $1 \leq k \leq t$. A set $\mathcal{F}$ of feasible solutions for I is called a *ρ-approximation of the Pareto frontier* if, for every feasible solution S for I, $\mathcal{F}$ contains a feasible solution S' that is a ρ-approximation of S. Note that ρ always satisfies $\rho \geq 1$. The closer ρ is to 1, the better the approximation of the Pareto frontier.

An algorithm that runs in polynomial time in the size of the input and that always outputs a ρ-approximation of the Pareto frontier is called a *ρ-approximation algorithm*. A *polynomial-time approximation scheme* (PTAS) for the Pareto frontier is a family of algorithms that contains, for every fixed constant $\varepsilon > 0$, a $(1+\varepsilon)$-approximation algorithm A_ε. If the running-time of A_ε is polynomial in the size of the input and in ε^{-1}, the family of algorithms is called a *fully polynomial-time approximation scheme* (FPTAS).

The Pareto frontier of an instance of a multi-objective optimization problem may contain an arbitrarily large number of solutions. To the contrary, for every $\varepsilon > 0$ there exists a $(1+\varepsilon)$-approximation of the Pareto frontier that consists of a number of solutions that is polynomial in the size of the instance and in ε^{-1} (under reasonable assumptions). An explicit proof for this observation has recently been given by Papadimitriou and Yannakakis [9]. Consequently, a PTAS or FPTAS for a multi-objective optimization problem does not only have the advantage of computing a provably good approximation in polynomial time, but also has a good chance of presenting a reasonably small set of solutions to the user.

2.2 Multi-objective Knapsack Problems

We will consider two types of multi-objective knapsack problems. In the first case, the classical 0–1 knapsack problem is simply extended by introducing t profit values for every item. Hence, an instance I of the *multi-objective knapsack problem* consists of the following:

- a number n of items
- for each item i a weight $w_i \in \mathbb{N}$ and t profits $p_{ki} \in \mathbb{N}$, $k = 1, \ldots, t$.
- a knapsack with capacity $c \in \mathbb{N}$.

A feasible solution is a subset S of the given items satisfying the weight constraint $\sum_{i \in S} w_i \leq c$.

In the more general *multi-objective m-dimensional knapsack problem* there are m weights for every item. More precisely, an instance of this problem consists of the following:

- a number n of items
- for each item i there are m weights $w_{ki} \in \mathbb{N}$, $k = 1, \ldots, m$, and t profits $p_{ki} \in \mathbb{N}$, $k = 1, \ldots, t$.
- a knapsack with m capacities $c_k \in \mathbb{N}$, $k = 1, \ldots, m$.

Here, a feasible solution is a subset S of the given items satisfying all of the following m constraints:

$$\sum_{i \in S} w_{ki} \leq c_k \text{ for } k = 1, \ldots, m.$$

Throughout this paper, we assume that m is a constant.

In both cases, the t objective values of a feasible solution S are $V_1(S)$, ..., $V_t(S)$ with $V_k(S) = \sum_{i \in S} p_{ki}$.

3 Related Work

3.1 Single-objective Knapsack Problems

The existence of an FPTAS for the classical 0–1 knapsack problem has been known right from the beginning of research on approximation algorithms (cf. [8]). However, as soon as a second weight constraint is added, thus extending the problem to the *2-dimensional knapsack problem*, the existence of an FPTAS would imply $P = NP$ as shown by Korte and Schrader [3]. Hence, we can only hope to develop a PTAS for the multi-objective m-dimensional knapsack problem.

Indeed, for the single-objective m-dimensional knapsack problem a PTAS was presented by Frieze and Clarke [2].

Theorem 1 (Frieze and Clarke, 1984). *For every constant m, there exists a polynomial-time approximation scheme for the m-dimensional 0–1 knapsack problem.*

3.2 Multi-objective Knapsack Problems

An extensive study of multi-objective combinatorial optimization problems was carried out by Safer and Orlin [11,12]. In [11], they study necessary and sufficient conditions for the existence of an FPTAS for a multi-objective optimization problem. Their approach is based on the concept of VPP algorithms and VPP reductions (VPP stands for value-pseudo-polynomial). It allows to obtain an FPTAS for a multi-objective optimization problem either by designing a VPP algorithm for it or by using a VPP reduction to a problem for which a VPP algorithm has already been found. In [12], they apply these techniques to obtain FPTASs for various multi-objective network flow, knapsack, and scheduling problems. In particular, they prove the existence of an FPTAS for the multi-objective knapsack problem, which we study in Section 4. However, it should be noted that their FPTAS is obtained using a VPP reduction, which in general does not lead to an FPTAS with a good running-time. In particular, their approach involves the solution of many different scaled versions of the given instance using a VPP algorithm. In [12, p. 26] they write: "Note that the results presented are existential. The characteristics of a particular problem must be considered carefully in order to develop a practical algorithm for its solution." In Section 4 we develop such an FPTAS that is tailored to the multi-objective knapsack problem and provides a better running-time than the FPTAS resulting from the existence proof in [12].

A detailed study of different dynamic programming approaches for the optimal solution of the multi-objective knapsack problem was recently given by Klamroth and Wiecek [6]. A branch and bound algorithm for the bicriteria knapsack problem ($t = 2$) was recently given by Ehrgott [1].

Further results concerning the existence of an FPTAS for a multi-objective optimization problem are given by Papadimitriou and Yannakakis [9]. They show

that an FPTAS exists if and only if a certain *gap problem* can be solved efficiently. For the case of linear objective functions and a feasible set consisting of integer vectors satisfying certain conditions, they show that there is an FPTAS if there is a pseudopolynomial-time algorithm for the exact version of the problem (determining for a single linear objective function and a target value B whether a feasible solution with objective value exactly B exists).

4 An FPTAS for the Multi-objective Knapsack Problem

The standard procedure to construct an FPTAS for the knapsack problem and its relatives is to start with an optimal dynamic programming scheme and transform it into an FPTAS by appropriate scaling techniques. Let us therefore begin by recalling an optimal dynamic programming scheme for the 0–1 knapsack problem and extending it to the multi-objective 0–1 knapsack problem.

4.1 An Optimal Dynamic Programming Algorithm

We will use dynamic programming by reaching in the profit space. The dynamic programming function W is defined by $W[v] = w$ and indicates that there exists a subset of items with profit v and weight w, where w is minimal among all such sets. It is well known that this function can be computed by going through the recursion

$$W[v + p_i] = \min\{W[v + p_i], W[v] + w_i\}$$

for $i = 1, \ldots, n$ after initializing $W[0] = 0$ and $W[v] = c + 1$ for $v \geq 1$. The first expression in the minimum denotes the weight of the best possible solution with profit $v + p_i$ so far not including item i whereas the second expression packs item i into the knapsack.

To bound the length of the dynamic programming array we need to find an upper bound UB on the optimal solution value. This can be done easily, e.g. by running the classical greedy algorithm with performance guarantee 1/2. Altogether the running time of this straightforward approach is $O(n\, UB)$, since we try to add every item to every profit value.

To generalize this algorithm to the multi-objective case we can basically expand it to the t-dimensional profit space. First of all we have to compute upper bounds on each of the t objective functions. They can be obtained easily, e.g. by running the greedy algorithm separately for every objective. Thus, t upper bounds $UB_1, \ldots, UB_t$ are computed. For convenience we also introduce $UB_{\max} := \max\{UB_1, \ldots, UB_t\}$.

Then we define the following dynamic programming function for $v_k = 0, 1, \ldots, UB_k$ and $k = 1, \ldots, t$:

$$W[v_1, \ldots, v_t] = w \qquad \Longleftrightarrow$$

there exists a subset of items with profit v_k in the k-th objective for $k = 1, \ldots, t$ and weight w, where w is minimal among all such sets.

After setting all function values to $c+1$ in the beginning (indicating that the corresponding combination of profits can not be reached yet) except $W[0,\ldots,0]$ which is set to 0, the function W can be computed by considering the items in turn and performing for every item i the analogous update operation for all feasible combinations of profit values (i.e. $v_k + p_{ki} \leq UB_k$):

$$W[v_1 + p_{1i}, \ldots, v_t + p_{ti}] = \min\{W[v_1 + p_{1i}, \ldots, v_t + p_{ti}], W[v_1, \ldots, v_t] + w_i\}$$

As above this operation can be seen as trying to improve the current weight of an entry by adding item i to the entry $W[v_1, \ldots, v_t]$.

It remains to extract the Pareto frontier from W. This can be done in a straightforward way: Each entry of W satisfying $W[v_1, \ldots, v_t] \leq c$ corresponds to a feasible solution with objective values $v_1, \ldots, v_t$. The set of items leading to that solution can be determined easily if standard bookkeeping techniques are used. The feasible solutions are collected and the resulting set of solutions is reduced by discarding solutions that are weakly dominated by other solutions in the set.

The correctness of this approach follows from classical dynamic programming theory. Its running time is given by going through the complete profit space for every item which is $O(n \prod_{k=1}^{t} UB_k)$ and hence $O(n(UB_{\max})^t)$.

4.2 Developing an FPTAS

An obvious strategy to develop an FPTAS for the multi-objective knapsack problem would be to scale the profits and apply the optimal dynamic programming algorithm after scaling, thus adapting the FPTAS for the 0–1 knapsack problem to the multi-objective case. However, this approach runs into difficulties. Since the error allowed for a $(1+\varepsilon)$-approximation is a relative error and hence depends on the solution value in every objective criterion of any given solution, the classical partitioning of the profit space into intervals of equal size does not work. Note that usually the set of items is partitioned into items with large and small profits. Then the scaled dynamic programming is performed only for the large items. The small items are finally added by the greedy algorithm. To guarantee that the relative error contributed by the small items to a solution, which might have an extremely small profit in one dimension, is not too large, the set of small items must be chosen to contain only items with extremely small profits in that dimension. But then the set of large items spans a wide profit range. Partitioning this range into a bounded number of appropriately small equal sized intervals is an impossible task, since the ratio between the smallest and largest objective value of one criterion (i.e. the upper bound on the profit space) can not be bounded.

To make the extension of an FPTAS from one to m dimensions possible, we require an algorithm which computes a feasible solution with a relative ε-error for every possible profit value in every objective. In the following we will present an FPTAS for the 0–1 knapsack problem fulfilling this particular property and briefly show how it can be extended to an FPTAS for the multi-objective knapsack problem.

The description of this FPTAS that is derived from the dynamic programming scheme at the beginning of Section 4.1 is relatively simple. The profit space between 1 and UB is partitioned into u intervals

$$[1, (1+\varepsilon)^{\frac{1}{n}}),\ [(1+\varepsilon)^{\frac{1}{n}}, (1+\varepsilon)^{\frac{2}{n}}),\ [(1+\varepsilon)^{\frac{2}{n}}, (1+\varepsilon)^{\frac{3}{n}}),\ \ldots, [(1+\varepsilon)^{\frac{u-1}{n}}, (1+\varepsilon)^{\frac{u}{n}})$$

with $u := \lceil n \log_{1+\varepsilon} UB \rceil$. Note that u is of order $O(1/\varepsilon \cdot n \log UB)$ and hence polynomial in the length of the encoded input. The main point of this construction is to guarantee that in every interval the upper endpoint is exactly $(1+\varepsilon)^{\frac{1}{n}}$ times the lower endpoint.

To achieve an FPTAS we adapt the above dynamic programming algorithm to the partitioned profit space. Instead of considering function W for all integer profit values, we consider only the value 0 and the u lower endpoints of intervals as possible profit values. The definition of the resulting dynamic programming function $\tilde{W}$ is slightly different from W. An entry $\tilde{W}[\tilde{v}] = w$, where $\tilde{v}$ is the lower endpoint of an interval of the partitioned profit space, indicates that there exists a subset of items with weight w and a profit of *at least* $\tilde{v}$.

The update operation where item i is added to every entry $\tilde{W}[\tilde{v}]$ is modified in the following way. We compute the profit attained from adding item i to the current entry. The resulting value $\tilde{v} + p_i$ is *rounded down* to the nearest lower endpoint of an interval $\tilde{u}$. The weight in the corresponding dynamic programming function entry is compared to $\tilde{W}[\tilde{v}]+w_i$. The minimum of the two values is stored as (possibly updated) function value $\tilde{W}[\tilde{u}]$.

The running time of this approach is bounded by $O(nu)$ and hence by $O(1/\varepsilon \cdot n^2 \log UB)$.

Theorem 2. *The above algorithm computes a $(1+\varepsilon)$-approximation for every reachable profit value of a 0–1 knapsack problem.*

Proof. The correctness of the statement is shown by induction over the set of items. In particular, we will show the following claim.

> **Claim:** After performing all update operations with items $1, \ldots, i$ for some $i \in \{1, \ldots, n\}$ for the optimal function W and the approximate function $\tilde{W}$ there exists for every entry $W[v]$ an entry $\tilde{W}[\tilde{v}]$ with
>
> $$\text{(i)} \quad \tilde{W}[\tilde{v}] \leq W[v] \quad \text{and} \qquad \text{(ii)} \quad (1+\varepsilon)^{\frac{i}{n}}\, \tilde{v} \geq v.$$

Evaluating (ii) for $i = n$ immediately yields the desired result.

To prove the Claim for $i = 1$ we add item 1 into the empty knapsack. Hence we get an update of the optimal function with $W[p_1] = w_1$ and of the approximate function with $\tilde{W}[\tilde{v}] = w_1$ where $\tilde{v}$ is the largest interval endpoint not exceeding p_1. Property (i) holds with equality, whereas (ii) follows from the fact that p_1 and $\tilde{v}$ are in the same interval and hence $(1+\varepsilon)^{\frac{1}{n}}\, \tilde{v} \geq p_1$.

Assuming the Claim to be true for $i-1$ we can show the properties for i by investigating the situation after trying to add item i to every function entry.

Clearly, those entries of W which were not updated by item i fulfill the claim by the induction hypothesis since (ii) holds for $i-1$ and even more so for i.

Now let us consider an entry of W which was actually updated, i.e. item i was added to $W[v]$ yielding $W[v+p_i]$. Since the Claim is true before considering item i, there exists $\tilde{W}[\tilde{v}]$ with $(1+\varepsilon)^{\frac{i-1}{n}}\tilde{v} \geq v$, i.e., $\tilde{v} \geq v/(1+\varepsilon)^{\frac{i-1}{n}}$.

In the FPTAS $\tilde{v}+p_i$ is computed and rounded down to some lower interval endpoint $\tilde{u}$. From the interval construction we have $(1+\varepsilon)^{\frac{1}{n}}\,\tilde{u} \geq \tilde{v}+p_i$. Putting things together this yields

$$(1+\varepsilon)^{\frac{1}{n}}\,\tilde{u} \geq v/(1+\varepsilon)^{\frac{i-1}{n}} + p_i \geq (v+p_i)/(1+\varepsilon)^{\frac{i-1}{n}}.$$

Moving terms around, this proves that $\tilde{u}$ satisfies (ii) for $v+p_i$. Property (i) follows immediately by applying the claim for $W[v]$ since

$$\tilde{W}[\tilde{u}] = \min\{\tilde{W}[\tilde{u}], \tilde{W}[\tilde{v}]+w_i\} \leq W[v]+w_i = W[v+p_i]. \qquad \square$$

The main point of developing the above FPTAS was the approximation of every reachable profit value and not only the optimal solution value, since this will allow us to extend the FPTAS to the multi-objective case. In itself, this new FPTAS for the single-objective 0–1 knapsack problem does not improve on the previously best known FPTASs. For the 0–1 knapsack problem in general, the best known FPTAS is due to Kellerer and Pferschy [4,5]. For the special case where $n \leq 1/\varepsilon$, the currently best FPTAS by Magazine and Oguz [7] runs in asymptotic time of $O(n^2 \log n \cdot 1/\varepsilon)$. Depending on the relationship between $\log n$ and $\log UB$ for a given family of instances, our new FPTAS may be preferable.

The space requirement of our new FPTAS is $O(n+u)$, i.e. $O(n \log UB \cdot 1/\varepsilon)$, after avoiding the explicit storage of subsets for every dynamic programming entry (see [10]). This is higher than the $O(n \cdot 1/\varepsilon)$ required by [7].

It should be noted that the performance of this new FPTAS for the single-objective knapsack problem can be further improved for the general case of $n > 1/\varepsilon$ by introducing the usual partitioning of items into small items ($p_i \leq \varepsilon UB/2$) and large items ($p_i > \varepsilon UB/2$). Applying the interval structure to the set of large items we can start with an interval $[\varepsilon UB/2,\ \varepsilon UB/2 \cdot (1+\varepsilon)^{\frac{1}{n}})$ and continue as above to generate further intervals by multiplying with $(1+\varepsilon)^{\frac{1}{n}}$. In this case, the number of intervals is of order $O(1/\varepsilon \cdot n \log(1/\varepsilon))$ since the value of UB cancels out in the computation. Hence, the total running time would be $O(n \log n + 1/\varepsilon \cdot n^2 \log(1/\varepsilon))$.

Furthermore, we can increase the range of the intervals by replacing the multiplier $(1+\varepsilon)^{\frac{1}{n}}$ by $(1+\varepsilon)^{\varepsilon}$. Recall that in the proof of Theorem 2 the value $\frac{1}{n}$ was only necessary because every entry of the dynamic programming function may correspond to a sequence of up to n items. However, dealing only with large items, there can be at most $O(1/\varepsilon)$ items contributing to any dynamic programming entry. This reduces the number of intervals to $O(1/\varepsilon^2 \log(1/\varepsilon))$.

A further improvement is attained by using the reduction of the set of large items given in [4] as a preprocessing step. It follows that only $1/\varepsilon^2$ large

items need to be considered at all. Performing an update operation for each of them with the reduced number of intervals yields an improved running time of $O(n \log n + 1/\varepsilon^4 \log(1/\varepsilon))$ with $O(n + 1/\varepsilon^2 \log(1/\varepsilon))$ space.

The extension of this FPTAS to the multi-objective problem can now be performed in a completely analogous way as for the optimal dynamic programming scheme of Section 4.1. The partitioning of the profit space is done for every dimension, which yields $u_k := \lceil n \log_{1+\varepsilon} UB_k \rceil$ intervals for every objective k. Instead of adding an item i to all lower interval endpoints $\tilde{v}$ as in the one-dimensional case we now have to add it to every possible t-tuple $(\tilde{v}_1, \ldots, \tilde{v}_t)$, where $\tilde{v}_k$ is either 0 or a lower interval endpoint of the k-th profit space partitioning. The resulting objective values $\tilde{v}_1 + p_{1i}, \ldots, \tilde{v}_t + p_{ti}$ are all rounded down for every objective to the nearest interval endpoint. At the resulting t-tuple of lower interval endpoints a comparison to the previous dynamic programming function entry is performed.

Since the "projection" of this algorithm to a single objective yields exactly the FPTAS for the 0–1 knapsack problem as discussed above, the correctness of this method follows from Theorem 2. The running time of this approach is clearly bounded by $O(n \prod_{k=1}^{t} u_k)$ and hence $O(n(1/\varepsilon \cdot n \log UB_{\max})^t)$. Summarizing, we have shown the following statement.

Theorem 3. *For every constant t, the above algorithm is an FPTAS for the multi-objective knapsack problem with t objectives.*

5 A Polynomial-Time Approximation Scheme

In this section, we present a polynomial-time approximation scheme for the multi-objective m-dimensional knapsack problem. Recall that t denotes the number of objective functions and that we assume that both m and t are constants. We make use of some of the ideas from the PTAS for the single-objective m-dimensional knapsack problem due to Frieze and Clarke [2].

Let $\varepsilon > 0$ be given. Choose $\delta > 0$ and $\mu > 0$ to be positive constants such that $(1+\delta)(1+\mu) \leq 1+\varepsilon$. For example, set $\delta = \mu = \min\{1/3, \varepsilon/3\}$.

Let an instance I of the multi-objective m-dimensional knapsack problem be given. First, the algorithm computes for each k, $1 \leq k \leq t$, a $(1+\delta)$-approximation U_k of the m-dimensional 0–1 knapsack problem with objective p_{k*} (i.e., with $p_i = p_{ki}$ for all items i), using Theorem 1. Note that any feasible solution S to I satisfies $0 \leq V_k(S) \leq (1+\delta)V_k(U_k)$ for all $1 \leq k \leq t$. Therefore, the set of all objective vectors of all feasible solutions to I is contained in

$$\mathcal{U} = [0, (1+\delta)V_1(U_1)] \times [0, (1+\delta)V_2(U_2)] \times \cdots \times [0, (1+\delta)V_t(U_t)].$$

Define $u_k = \lceil \log_{1+\delta} V_k(U_k) \rceil$. With respect to the k-th objective, we consider the following (lower) bounds for objective values:

- $u_k + 1$ lower bounds $\ell_{kr} = (1+\delta)^{r-1}$, for $1 \leq r \leq u_k + 1$, and
- the lower bound $\ell_{k0} = 0$.

The algorithm enumerates all tuples $(r_1, r_2, \ldots, r_{t-1})$ of $t-1$ integers satisfying $0 \leq r_k \leq u_k + 1$ for all $1 \leq k < t$. Note that the number of such tuples is polynomial in the size of the input if t and δ are constants. Roughly speaking, the algorithm considers, for each tuple $(r_1, r_2, \ldots, r_{t-1})$, the subspace of solutions S satisfying $V_k(S) \geq \ell_{kr_k}$ for all $1 \leq k < t$, and tries to maximize $V_t(S)$ among all such solutions.

For this purpose, the algorithm proceeds as follows. Let $h = \lceil (t+m-1)(1+\mu)/\mu \rceil$. Let $A_1, A_2, \ldots, A_t$ be subsets of $\{1, 2, \ldots, n\}$ with cardinality at most h. The algorithm enumerates all possibilities for choosing such sets $A_1, A_2, \ldots, A_t$. Intuitively, the set A_k represents the selected items of highest profit with respect to p_{k*}. Let $A = A_1 \cup A_2 \cup \cdots \cup A_t$. For $1 \leq k \leq t$, let

$$F_k = \{j \in \{1, 2, \ldots, n\} \setminus A_k \mid p_{kj} > \min\{p_{kq} \mid q \in A_k\}\}.$$

Intuitively, F_k is the set of all items that cannot be put into the knapsack if A_k are the items with highest profit with respect to p_{k*} in the solution. Let $F = F_1 \cup F_2 \cup \cdots \cup F_m$. If $F \cap A \neq \emptyset$, the current choice of the sets A_k is not consistent, and the algorithm continues with the next possibility of choosing the sets $A_1, A_2, \ldots, A_t$.

So let us assume that $F \cap A = \emptyset$. Consider the following linear programming relaxation:

$$\begin{aligned}
\text{(LP)} \quad & \max \sum_{j=1}^{n} p_{tj} x_j \\
& \text{s.t.} \sum_{j=1}^{n} w_{kj} x_j \leq c_k, \quad \text{for } k = 1, 2, \ldots, m \\
& \sum_{j=1}^{n} p_{kj} x_j \geq \ell_{kr_k}, \quad \text{for } k = 1, 2, \ldots, t-1 \\
& x_j = 0, \quad j \in F \\
& x_j = 1, \quad j \in A \\
& 0 \leq x_j \leq 1, \quad j \in \{1, 2, \ldots, n\} \setminus (F \cup A)
\end{aligned}$$

Let $\hat{x}$ be an optimal (basic feasible) solution vector for (LP). Such a vector $\hat{x}$ can be computed in polynomial time, if it exists. (If no such vector exists, proceed to the next combination of sets $A_1, \ldots, A_t$.) As (LP) has only $t+m-1$ non-trivial constraints, at most $t+m-1$ components of $\hat{x}$ are fractional. Now an integral vector $\bar{x}$ is obtained by rounding down $\hat{x}$, i.e., we set $\bar{x}_j = \lfloor \hat{x}_j \rfloor$. From the integral vector $\bar{x}$, we obtain a solution S to the multi-objective m-dimensional knapsack problem by letting $S = \{j \in \{1, 2, \ldots, n\} \mid \bar{x}_j = 1\}$.

For each possibility of choosing $A_1, A_2, \ldots, A_t$ consistently, we either find that (LP) has no solution, or we compute an optimal fractional solution to (LP) and obtain a rounded integral solution S. We output all integral solutions that are obtained in this way (if any). The procedure is repeated for every tuple $(r_1, r_2, \ldots, r_{t-1})$. This completes the description of the algorithm.

Theorem 4. *For every pair of constants m and t, the above algorithm is a polynomial-time approximation scheme for the t-objective m-dimensional knapsack problem.*

Proof. We show that the algorithm described above is a $(1+\varepsilon)$-approximation algorithm for the multi-objective m-dimensional knapsack problem.

First, we argue that the running-time is polynomial. Initially, the $(1+\delta)$-approximation algorithm for the single-objective m-dimensional 0–1 knapsack problem is called t times. This gives the solutions $U_1,\ldots,U_t$ and the numbers $u_1,\ldots,u_t$. Then our algorithm enumerates all tuples of $t-1$ numbers $r_1,\ldots,r_{t-1}$ satisfying $0 \le r_k \le u_k + 1$ for all $1 \le k \le t-1$. There are $O(u_1 u_2 \cdots u_{t-1})$ such tuples. As each u_i is polynomial in the size of the input, the total number of tuples is also bounded by a polynomial. For each tuple, all combinations of choosing sets $A_1,\ldots,A_t$ of cardinality at most h are enumerated. There are $O(n^{t(t+m-1)(1+\mu)/\mu})$ such combinations. For each combination, the linear program (LP) is created and solved in polynomial time. Since m, t, δ and μ are constants, the overall running-time is polynomial in the size of the input.

Now, we analyze the approximation ratio. Consider an arbitrary feasible solution G. We have to show that the output of the algorithm contains a solution S that is a $(1+\varepsilon)$-approximation of G. For $1 \le k \le t-1$, define $r_k = \max\{r \mid \ell_{kr} \le V_k(G)\}$. It follows that $\ell_{kr_k} \le V_k(G) \le \ell_{kr_k}(1+\delta)$. For $1 \le k \le t$, let A_k be a set that contains $\min\{h, |G|\}$ items in G with largest profit p_{k*}. When the algorithm considers the tuple $(r_1, r_2, \ldots, r_{t-1})$ and the sets $A_1,\ldots,A_t$, the linear program (LP) is feasible, because G constitutes a feasible solution. Therefore, the algorithm obtains a fractional solution S_f and outputs a rounded integral solution S for this tuple $(r_1, r_2, \ldots, r_{t-1})$ and for this combination of sets $A_1,\ldots,A_t$. If $|G| \le h$, the rounded solution S is at least as good as G with respect to all t objectives, because it contains G as a subset. Thus, the solution S output by the algorithm weakly dominates G in this case.

Now assume that $|G| > h$. Consider objective k. We have that $V_k(S_f) \ge V_k(G)/(1+\delta)$. For $1 \le k < t$, this is because S_f is a feasible solution to (LP). For $k = t$, we even have $V_t(S_f) \ge V_t(G)$, because S_f is an optimal fractional solution to (LP).

Furthermore, we claim that $V_k(S) \ge V_k(S_f)/(1+\mu)$. Consider some objective k, $1 \le k \le t$. When S is obtained from S_f, at most $t+m-1$ items are lost, because S_f has at most $t+m-1$ fractional items, as we noted above. Let $\bar{p}$ be the smallest profit among the h items with highest profit in G with respect to objective k. We have $V_k(S_f) \ge h\bar{p}$ and $V_k(S) \ge V_k(S_f) - (t+m-1)\bar{p}$. Combining these two inequalities, we get

$$\begin{aligned} V_k(S) &\ge V_k(S_f) - \frac{t+m-1}{h} h\bar{p} \ge V_k(S_f) - \frac{t+m-1}{h} V_k(S_f) \\ &= V_k(S_f)\left(1 - \frac{t+m-1}{h}\right) \ge V_k(S_f)\left(1 - \frac{\mu}{1+\mu}\right) = V_k(S_f)/(1+\mu). \end{aligned}$$

So we have $V_k(S) \geq V_k(S_f)/(1+\mu) \geq V_k(G)/((1+\mu)(1+\delta))$. Since we have chosen μ and δ such that $(1+\mu)(1+\delta) \leq (1+\varepsilon)$, we obtain that S is a $(1+\varepsilon)$-approximation of G.

As the above argument is valid for any feasible solution G, we have shown that the set of solutions output by our algorithm is indeed a $(1+\varepsilon)$-approximation of the Pareto frontier. □

Acknowledgement. We are grateful to Eckart Zitzler for bringing the multi-objective knapsack problem to our attention.

References

1. M. Ehrgott, *Multicriteria Optimization*, Lecture Notes in Economics and Mathematical Systems **491**, Springer, 2000.
2. A. Frieze and M. Clarke, "Approximation algorithms for the m-dimensional 0–1 knapsack problem: Worst-case and probabilistic analyses", *European Journal of Operational Research* **15** (1984), 100–109.
3. B. Korte and R. Schrader, "On the Existence of Fast Approximation Schemes", *Nonlinear Programming* **4** O.L. Mangasarian, R.R. Meyer, S.M. Robinson (ed.), Academic Press 1981, 415–437.
4. H. Kellerer and U. Pferschy, "A new fully polynomial time approximation scheme for the knapsack problem", *Journal of Combinatorial Optimization* **3** (1999), 59–71.
5. H. Kellerer and U. Pferschy, "Improved dynamic programming in connection with an FPTAS for the knapsack problem", Technical Report 05/1999, Faculty of Economics, University of Graz, 1999, submitted.
6. K. Klamroth and M.M. Wiecek, "Dynamic programming approaches to the multiple criteria knapsack problem", *Naval Research Logistics* **47** (2000), 57–76.
7. M.J. Magazine and O. Oguz, "A fully polynomial approximation algorithm for the 0–1 knapsack problem", *European Journal of Operational Research* **8** (1981), 270–273.
8. S. Martello and P. Toth, *Knapsack Problems*, J. Wiley & Sons, 1990.
9. C.H. Papadimitriou and M. Yannakakis, "On the approximability of trade-offs and optimal access of web sources". In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science FOCS'00* (2000), 86–92.
10. U. Pferschy, "Dynamic programming revisited: Improving knapsack algorithms", *Computing* **63** (1999), 419–430.
11. H.M. Safer and J.B. Orlin, "Fast approximation schemes for multi-criteria combinatorial optimization", Working Paper 3756–95, MIT, January 1995.
12. H.M. Safer and J.B. Orlin, "Fast approximation schemes for multi-criteria flow, knapsack, and scheduling problems", Working Paper 3757–95, MIT, January 1995.
13. E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach", *IEEE Transactions on Evolutionary Computation* **3** (1999), 257-271.

Visual Ranking of Link Structures
Extended Abstract

Ulrik Brandes and Sabine Cornelsen

Department of Computer & Information Science, University of Konstanz,
Box D 188, 78457 Konstanz, Germany.
{Ulrik.Brandes,Sabine.Cornelsen}@uni-konstanz.de

Abstract. Methods for ranking World Wide Web resources according to their position in the link structure of the Web are receiving considerable attention, because they provide the first effective means for search engines to cope with the explosive growth and diversification of the Web. We show that layouts for effective visualization of an underlying link structure can be computed in sync with the iterative computation utilized in all popular such rankings. Our visualizations provide valuable insight into the link structure and the ranking mechanism alike. Therefore, they are useful for the analysis of query results, maintenance of search engines, and evaluation of Web graph models.

1 Introduction

The directed graph induced by the hyperlink structure of the Web has been recognized as a rich source of information. Understanding and exploiting this structure has a proven potential to help dealing with the explosive growth and diversification of the Web. Probably the most widely recognized example of this kind is the PageRank index employed by the Google search engine [6].

PageRank is but one of many models and algorithms to rank Web resources according to their position in a link structure (see, e.g., [25,20,9,1,5,8]). Our goal is to supplement rankings with a meaningful visualization of the graph they are computed on.

While graph visualization is an active area of research as well [10,19], its integration with quantitative analysis is only beginning to receive attention. It is, however, rather difficult to understand the determinants of a particular ranking if a visualization is prepared without explicitly taking it into account.

A design for graph visualizations showing vertex prominence in the structural context is introduced in [4], where the vertical dimension of the layout space is reserved to represent exactly the prominence of each vertex, but since horizontal layout is done by an adaptation of the Sugiyama framework for layered graph drawing [27] it does not scale to graphs with more than a few hundred vertices.

In the present application, it is also highly desirable that dense subgraphs be clustered, since on the Web they typically correspond to related resources. A well-known class of graph layout algorithms that exhibit these properties are

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 222–233, 2001.

spring embedders, and the range for which they are practical has recently been extended to sparse graphs with several thousands of vertices [13,15,29]. Since they require relatively complex memory management, however, they are less suitable in our situation.

Instead, we apply a spectral graph layout approach, because of its formal and computational similarities with common ranking techniques. Background on link-based ranking is given in Sect. 2. The actual layout definition and computation are described in Sect. 3. In Sect. 4, we discuss applications and provide examples on generated and real-world data.

2 Structural Ranking of Web Resources

The structural features of the Web are captured in a directed graph $G = (V, E)$, where the set V of vertices represents the set of resources on the Web, and there is a directed edge $(u, v) \in E$ from a resource u to a resource v, if u contains a hyperlink to v. All graphs considered in this paper are assumed to be connected. We do not allow parallel edges, but loops and a positive real weight ω_{uv} for every edge. Let $A(G) = A = (A_{uv})_{u,v \in V}$ be the *adjacency matrix* of a graph, i.e. $A_{uv} = \omega_{uv}$ if $(u, v) \in E$, and $A_{uv} = 0$ otherwise. The *indegree* (*outdegree*), d_v^+ (d_v^-), of a vertex $v \in V$ is $\sum_{u:(u,v)\in E} A_{uv}$ ($\sum_{w:(v,w)\in E} A_{vw}$).

We will frequently use graph-related matrices, such as the adjacency matrix, and their *spectra*, i.e. the set of eigenvalues and associated eigenvectors. For background on matrix computations see, e.g., [14].

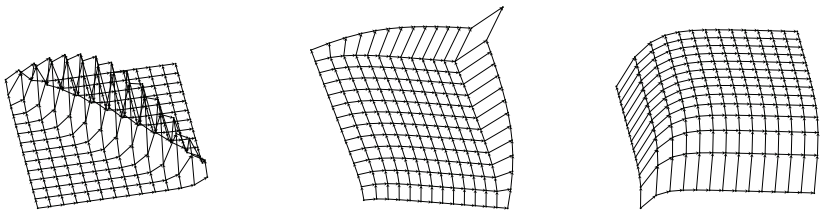
Any real-valued vector $p = (p_v)_{v \in V}$ defined on the vertices of a graph is called a *prominence index*, where p_v is the *prominence* of vertex v. A *ranking* is obtained from a prominence index by ordering the vertices according to non-increasing prominence.

Many models have been proposed to capture an explicitly or implicitly defined notion of a vertex's prominence in a graph [18,17,11,3,12,25,20,1,9,8, and many more]. Though in general only defined for undirected graphs, we first outline *eigenvector centrality* [2], because it nicely illustrates some important commonalities of the three popular rankings that we discuss below.

Assume that the prominence of a vertex is understood to be proportional to the combined prominence of its neighbors, $\lambda p_v = \sum_{u:\{u,v\}\in E} \omega_{uv} p_u$, $v \in V$, where the constant λ is introduced so that the system of equations has a non-zero solution. This definition yields the eigensystem of the transposed adjacency matrix,

$$\lambda p = Ap, \qquad (\textit{eigenvector centrality})$$

and every eigenvector of A gives a ranking of the vertices for the above notion of prominence, though the *principal eigenvector*, i.e. the one associated with the eigenvalue of largest magnitude, is generally preferred [3,12]. The principal eigenvector can be obtained by power iteration, which starts with any non-



(a) authority (b) PageRank (c) Katz's status

Fig. 1. Prominence indices on a directed grid

zero vector and iteratively multiplies the matrix with the current solution, e.g. $p^{(0)} \leftarrow \mathbf{1}$ and

$$p^{(k+1)} \leftarrow A \cdot p^{(k)}.$$

Since the matrices considered here originate from large and sparse graphs, multiplication is carried out by computing $p_v^{(k+1)} \leftarrow \sum_{u:\{u,v\}\in E} p_u^{(k)}$ for every $v \in V$. In this and in the iterations to follow, we tacitly assume that each vector $p^{(k)}$ is normalized (e.g., to the length of the starting vector) before the next vector is computed. This serves to avoid numerical difficulties with numbers growing out of range, and does not affect the relative prominence of vertices.

More elaborate indices defined on directed graphs are discussed below. In Fig. 1 they are illustrated on an acyclic grid. The grid is placed in a plane and each grid point is then lifted according to its prominence.

Hubs and authorities [20]. A natural notion of prominence for a Web resource is the extent to which it is referred to by other Web pages, in particular by those pages that specialize in listing useful resources. In turn, the property of being such a list of useful resources is a notion of prominence in itself. In these complementary and mutually reinforcing notions prominent resources are called *authorities* (resources with useful information) and *hubs* (pages with useful links).

The hub score of a page is proportional to the combined authority of the resources it links to, and the authority of a resource is proportional to the combined hub score of the pages linking to it. In practice, hub and authority scores are thus computed by iterating $p^{(0)} \leftarrow \mathbf{1}$ and

$$\begin{aligned} p^{(2k+1)} &\leftarrow A^T \cdot p^{(2k)} \\ p^{(2k+2)} &\leftarrow A \cdot p^{(2k+1)}. \end{aligned}$$

For $h^{(k)} = p^{(2k)}$ and $a^{(k)} = p^{(2k+1)}$, the alternating iteration can be written as

$$h^{(k+1)} \leftarrow AA^T \cdot h^{(k)} \qquad (\textit{hubs})$$

$$a^{(k+1)} \leftarrow A^TA \cdot a^{(k)}. \qquad (\textit{authorities})$$

In this formulation, it is easy to see that the hub and authority indices in a graph with adjacency matrix A correspond to eigenvector centrality in the weighted undirected graphs with adjacency matrix AA^T and A^TA, respectively.

As can be seen in Fig. 1(a), vertices on and above the falling diagonal of the grid have the highest authority, because they are in the midst of the undirected graph induced by A^TA. Compare this to the undirected graph induced by AA^T, indicating why the best hubs are found on and below this diagonal.

PageRank [5]. In another variant of eigenvector centrality the contribution of each vertex to another vertex's prominence is weighted by its outdegree, $p_v = \sum_{u:(u,v)\in E} \frac{p_u}{d_u^-}$ (see e.g. [24,7]). If we require p to be a probability distribution over the set of vertices, this notion has a nice interpretation as the stationary distribution of the simple random walk on the graph (or random surfer on the Web, if you will), in which each edge leaving a vertex is chosen with equal probability.

Let $M = D_-^{-1}A$ be the adjacency matrix normalized so that the rows sum to one, where D_- is the diagonal matrix with the outdegrees on the diagonal. Then, M is a stochastic matrix of transition probabilities, and a stationary distribution $p = M^T \cdot p$ satisfies the above notion of prominence. However, if a vertex has outdegree zero, the computation breaks down, and strongly connected components may cause an overdue increase of the prominence of their vertices. This so-called "sink problem" can be avoided by introducing an escape mechanism. Let $\hat{p}$ be an a-priori probability distribution over the vertices (e.g., user preferences or general popularity of a resource), then with probability ω the random walk picks an edge of the graph whereas with the remaining probability, it jumps to any other vertex according to $\hat{p}$. The index is thus defined by

$$\begin{aligned} p &= \omega M^T p + (1-\omega)\hat{p} && (\textit{PageRank}) \\ &= \left(\omega M^T + (1-\omega)\hat{p}\cdot \mathbf{1}^T\right)\cdot p. \end{aligned}$$

The second equality holds because p is a probability distribution. From the second expression it can be seen that PageRank is the eigenvector centrality of a weighted graph with a complete set of additional escape edges. This modified matrix is irreducible and aperiodic so that the iteration $p^{(0)} \leftarrow \frac{1}{n}\mathbf{1}$ and

$$p^{(k+1)} \leftarrow \left(\omega M + (1-\omega)\mathbf{1}\cdot \hat{p}^T\right)^T \cdot p^{(k)}$$

converges to a unique prominence vector. On the grid in Fig. 1(b), the random surfer may jump to any vertex, but is most likely to walk towards the upper and right side of the grid, from where the only continuation is towards the upper right corner.

Katz's status index [18]. As a generalization of simply using indegrees to measure 'status' in social networks, the prominence of a vertex is determined by the number of directed paths of arbitrary length ending in the vertex, where the influence of longer paths is attenuated by a decay factor. Recall that the entries of

the k-th power of the adjacency matrix of an unweighted graph give the number of paths of length k between every pair of vertices. Therefore, this notion of prominence is determined by

$$p = \left(\sum_{k=1}^{\infty} (\alpha A^T)^k \right) \cdot \mathbf{1}, \qquad (\textit{Katz's status})$$

where parameter α corresponds to the fraction of status that is passed along a directed edge. For sufficiently small values of α (a convenient choice is $\frac{1}{\Delta^+ + 1}$, where Δ^+ is the maximum indegree of any vertex in the graph), the sum converges to $(I - \alpha A^T)^{-1} - I$. Therefore, the status vector can be obtained by solving $\left(\alpha^{-1} I - A^T\right) \cdot p = d^+$. Solving this system of linear equations directly is prohibitive for large graphs. Standard sparse matrix approaches approximate a solution iteratively. The update step in Jacobi iteration, for instance, yields $p^{(k+1)} \leftarrow \alpha A^T \cdot p^{(k)} + \alpha d^+$. This iteration nicely reflects the underlying notion of adding contributions from vertices farther and farther away. The same can be observed in Fig. 1(c), where the attenuated influence from vertices in the lower left does not suffice to discriminate the prominence of vertices in the upper right any more.

In a sense, the above definitions of prominence are contained in the following generic formulation of status in networks [17]. It puts a twist on eigenvector centrality through the addition of an a-priori prominence vector $\hat{p}$,

$$p = A^T p + \hat{p}. \qquad (\textit{Hubbel's status})$$

By choosing appropriate weights and a-priori prominences, we obtain eigenvector centrality and PageRank. Reordering, we have $p = (I - A^T)^{-1} \cdot \hat{p}$, provided the inverse exists. If it does, it equals $\sum_{k=0}^{\infty} (A^T)^k$, and therefore $p = \left(\sum_{k=0}^{\infty} (A^T)^k\right) \cdot \hat{p} = \left(I + \sum_{k=1}^{\infty} (A^T)^k\right) \cdot \hat{p}$. With uniform edge weights and $\hat{p} = \mathbf{1}$ we obtain a prominence index in which every component is by one larger than Katz's status index.

3 Spectral Graph Layout

In the previous section we emphasized formal similarities in the definition of several popular prominence indices. In practice, all of them are computed by some variant of sparse matrix power iteration, i.e. by iterating over all vertices, and, for each vertex, combining the current scores of its neighbors. For really large graphs, most of the running time is spent on data transfer between internal and external memory. It is thus desirable not to cause additional swapping.

In this section, we introduce a layout algorithm that produces meaningful layouts while synchronously operating on the same data that prominence implementations do, because it is based on the same principles as the ranking algorithms. It is therefore a simple matter to augment an existing system for ranking resources to compute a layout of the graph on the fly.

Layout with eigenvectors. For the layout, we consider the undirected, simple graph obtained by omitting directions, loops, and multiple edges. Recall that directions are already represented in the prominence dimension. Let A be the adjacency matrix of the skeleton and $D = D^+ = D^-$ its diagonal *degree matrix*. We consider the *Laplacian matrix* $L = D - A$, which has interesting applications in many areas (see, e.g., [23]). This matrix is interesting for graph layout, since minimizing the associated quadratic form

$$x^T Lx = \sum_{\{u,v\} \in E} (x_u - x_v)^2,$$

corresponds to minimizing the squared distance between pairs of adjacent vertices, if x is interpreted as a vector of vertex positions. This corresponds to spring embedding with zero-length springs and no repelling forces. Clearly, the minima are obtained by assigning the same position to all vertices (recall that we assume connectedness of the graph).

These undesirable solutions can be avoided by fixing some selected vertices at distinct positions. Minimization subject to these boundary conditions is the famous *barycentric layout* model of Tutte [28]. However, it requires some a-priori knowledge about which vertices should be fixed where.

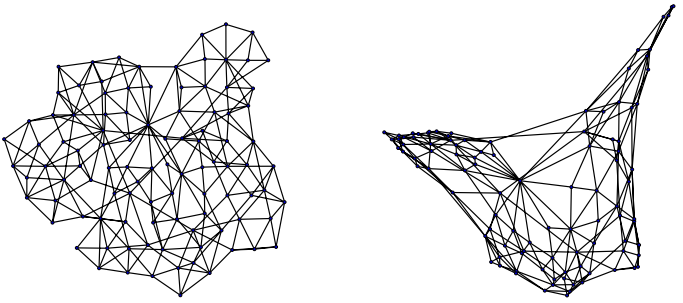
Note that the undesired minima $x = c\mathbf{1}$ are the eigenvectors associated with eigenvalue zero, i.e. $Lx = 0$. More generally, if (λ, x) is any eigenpair of L, then $\lambda = \frac{x^T Lx}{x^T x}$. We therefore want to minimize

$$\frac{x^T Lx}{x^T x} \text{ subject to } x \perp \mathbf{1},$$

since the eigenvectors of a symmetric matrix are orthogonal. Hence, the desired solution is the normalized eigenvector associated with the second-smallest eigenvalue of L. This vector is called the *Fiedler vector* and, because of its distance minimization property, often used in graph partitioning (see, e.g., [26]). For the same reason, it yields a useful one-dimensional layout of a graph, because edges are short and hence dense subgraphs are clustered. If a rank visualization in three dimensions is desired (cf. Fig. 1), a reasonable choice for the second free dimension is the normalized eigenvector minimizing the objective function subject to being orthogonal to $\mathbf{1}$ and the first solution.

An example of two-dimensional layouts obtained by a typical spring embedder and two eigenvectors of L is given in Fig. 2. While the spring embedder produces more uniform edge lengths, the eigenvectors emphasize structural clustering of vertices.

Computing the eigenvectors. Eigenvectors associated with the smallest eigenvalues of large sparse matrices are usually computed using Lanczos' method. However, all popular prominence indices are computed using a variant of the much simpler power iteration, which only gives an eigenvector associated with the eigenvalue of largest magnitude. Since we want to synchronize layout and prominence computation, we consider the matrix $L' = 2\Delta \cdot I - L$ instead of L



(a) spring embedder (b) spectral layout

Fig. 2. Two-dimensional layouts of a random planar triconnected graph

itself, where Δ is the maximum degree of any vertex. The crucial observation is that the eigenvectors of L and L' are identical, but the order of the magnitudes of their corresponding eigenvalues is reversed.

Straightforward application of power iteration on L' returns the principal eigenvector of L', which is the trivial eigenvector $\frac{1}{\sqrt{n}}\mathbf{1}$ of L. Power iteration on a vector that is orthogonal to the principal eigenvector yields an eigenvector of the second-largest eigenvalue of L', and hence the desired layout for the first dimension. Iterating on a vector that is orthogonal to both the trivial eigenvector and the approximate solution for the first dimension yields the second dimension.

A vector y is orthogonalized with respect to another vector x by setting $y \leftarrow y - \frac{x^T \cdot y}{x^T \cdot x}\, x$. Orthogonalization with respect to the trivial eigenvector $\frac{1}{\sqrt{n}}\mathbf{1}$ is even easier, since it corresponds to subtracting, from each entry of y, the mean of all its entries. To obtain vectors x and y for a two-dimensional layout we thus carry out the following augmented power iteration on random starting vectors $x^{(0)}, y^{(0)}$ that are repeatedly orthogonalized with respect to $\mathbf{1}$ and to one another

$$x^{(k+1)} \leftarrow L' \cdot x^{(k)}; \qquad x^{(k+1)} \leftarrow x^{(k+1)} - \frac{1}{n} \sum_{v \in V} x_v^{(k+1)}$$

$$y^{(k+1)} \leftarrow L' \cdot y^{(k)}; \qquad y^{(k+1)} \leftarrow y^{(k+1)} - \frac{1}{n} \sum_{v \in V} y_v^{(k+1)}$$

$$y^{(k+1)} \leftarrow y^{(k+1)} - \frac{x^{(k+1)T} \cdot y^{(k+1)}}{x^{(k+1)T} \cdot x^{(k+1)}}\, x^{(k+1)}$$

Intuitively, the layout is centered, rectified, and (due to the tacitly assumed normalization) zoomed after each multiplication with L'. The last two lines are omitted if only one dimension needs to be determined for the layout. Though convergence may be slower than in the prominence computations, a few iterations are usually sufficient for a reasonable layout.

4 Application to Web Graph Models and Query Results

We demonstrate our visualization approach on two different kinds of data, random Web graphs generated according to slightly modified versions of the *evolving copying models* of [21] and the *small-world model* of [30], and a real-world example obtained from an AltaVista query. Our C++-implementations use the *Library of Efficient Data Types and Algorithms* (LEDA) [22].

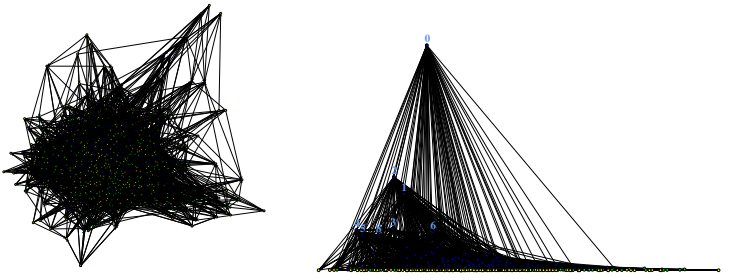
Web graph models. In the *linear growth model*, a graph grows one vertex at a time. At each time step, a prototype is chosen among already existing vertices, and a new vertex is generated. This new vertex is then assigned a fixed number of outgoing edges. With some fixed probability, the ith of these edges points to a randomly selected vertex among those already existing (creation case), and with the remaining probability it points to the same vertex as the ith outgoing edge of the prototype vertex (copying case). Our generator does not introduce multiple edges, and if a prototype happens to not have enough outgoing edges, no edge is introduced in the copying case. Clearly, all graphs evolving like this are acyclic.

In the *exponential growth model*, a graph grows by a fixed fraction of its current size at each time step. New vertices receive a fixed number of loops, and for each already existing edge, its target receives a new incoming edge for which, with some fixed probability, the source is chosen uniformly at random from the new vertices, and otherwise from the existing vertices with probability proportional to their current outdegree. We used a simpler model in which existing vertices are chosen uniformly at random as well.

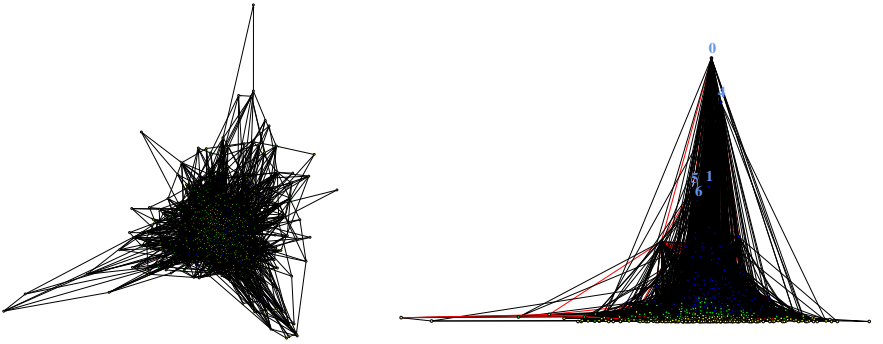
In the *small-world model*, we initially generate a cyclic sequence of vertices and let a vertex link to a fixed number of predecessors and successors. Then, each edge is rewired with some small probability by chosing a new destination uniformly at random.

Figure 3 shows spectral layouts of graphs generated according to these models and rankings replacing the vertical dimension with PageRank as an example of a prominence index. There is no visible clustering in the evolving copying models. Moreover, the prominence of resources appears to be correlated with their age (also with the other indices outlined in Sect. 2). The figures thus graphically support the conclusion of [21] that *death processes*, i.e. the occasional deletion of vertices and edges, might be necessary for the evolving copying models to be realistic. In the small-world model, the spectral layout reveals a cycle crumpled by chords, and the ranking shows that the model yields a rather egalitarian structure.

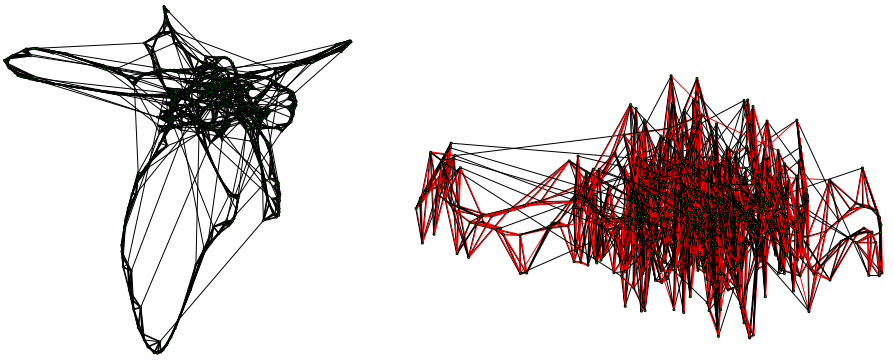
Query results. The data for this example was compiled in a way similar to the HITS algorithm [20]. We asked the AltaVista search engine for pages containing the word "java" and used the first 200 URLs it returned as the root set. It was then expanded by asking AltaVista for pages containing links to resources in the root set (backward extension), and adding resources linked to by pages in the root set (forward extension). The graph was completed by adding edges



(a) Linear growth evolving copying model [21]



(b) Exponential growth evolving copying model [21]



(c) Small-world model [30]

Fig. 3. Web graph models (2D spectral layout and 1D spectral layout with PageRank)

between pages in the resulting set of vertices. The computations were carried out on the only large component of this graph from which some poorly connected vertices were removed to prevent extreme clustering. The graph has more than 5000 vertices and 15000 edges.

In Fig. 4, this graph is shown twice, with vertices positioned vertically according to the Fiedler vector, and horizontally according to one of two prominence indices. Again, links to less prominent resources are colored red.

The most prominent resources match the expectations, but there are some less expected recommendations as well. It is clearly visible that some of these serve distinct user groups, like the japanese directory in the upper right. Note that we may not conclude that vertically close vertices are closely connected without zooming into the image. However, it is safe to assume that vertically separated vertices are likely to be distant in the structure. This feature can serve to distinguish query results which contain a keyword that is used in different contexts (see the "jaguar"-query example in [20]).

Figure 4 also shows that the top authorities are surprisingly distinguished from the rest of the graph, and quite different from our expectations. Most of them are located at `Stars.com`, a large repository for developers ("Web Developer's Virtual Library"). Since they are well connected among each other, it is by virtue of our layout approach that their vertical position is similar, and thus this phenomenon could be detected by visual exploration. In the figure, resources at this site are colored yellow. Not surprisingly, vertices with high hub scores are from this site as well. This simple example graphically explains why the original HITS algorithm does not consider links within a site.

5 Conclusions

We have proposed a method for Web graph visualization that provides unambiguous identification of prominent resources while showing the entire graph and its clustering. The layout of our visualizations can be computed synchronously with common link-based rankings. Speed-up techniques such as in [16] that reorganize storage to reduce external memory access therefore directly apply to the layout algorithm as well.

We expect our visualizations to be particularly useful for visual inspection of rankings, for teaching and communicating ranking procedures, and for evaluation and illustration of stochastic models of the Web graph. The main advantage of spectral graph layout, the correspondence with distance minimization and hence with clustering, becomes a drawback in cases where the underlying undirected graph is poorly connected, since denser subgraphs will be clustered in a very small interval. This problem is addressed in the full version of this paper, which also contains an analysis of the convergence properties of the iterative computation.

Acknowledgments. We thank Marco Gaertler for collecting the "java"-query data used in Sect. 4, and Stephen Muth for discussions regarding the iterative computation of Katz's status index.

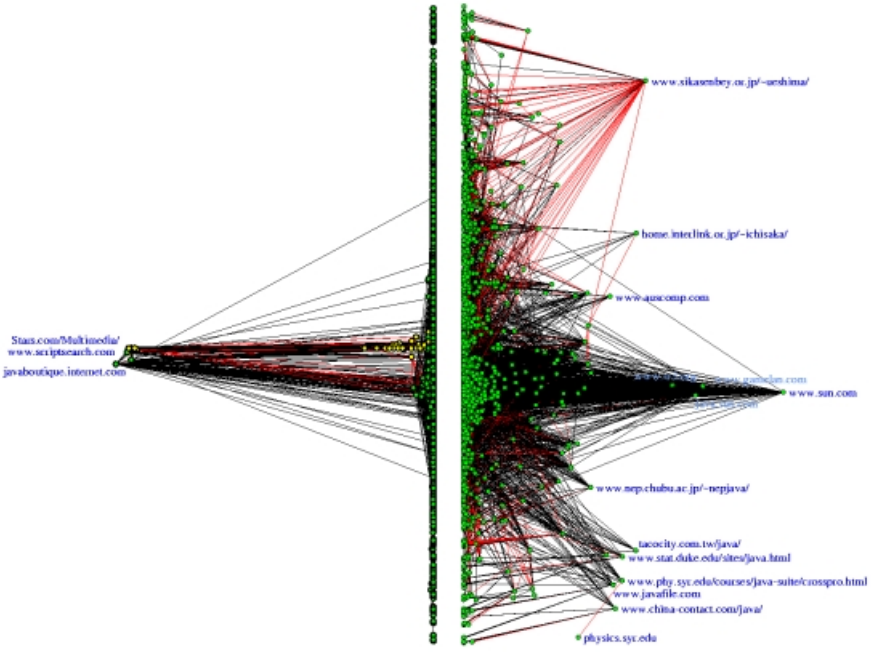


Fig. 4. Authority and PageRank visualization of "java" query result

References

1. K. Bharat and M. R. Henzinger. Improved algorithms for topic distillation in a hyperlinked environment. In *Proc. 21st Ann. Intl. ACM SIGIR Conf. Research and Development in Information Retrieval*, pp. 104–111, 1998.
2. P. Bonacich. Factoring and weighting approaches to status scores and clique identification. *Journal of Mathematical Sociology*, 2:113–120, 1972.
3. P. Bonacich. Power and centrality: A family of measures. *American Journal of Sociology*, 92:1170–1182, 1987.
4. U. Brandes and D. Wagner. Contextual visualization of actor status in social networks. In *Data Visualization 2000. Proc. 2nd Joint Eurographics and IEEE TCVG Symp. Visualization*, pp. 13–22. Springer, 2000.
5. S. Brin, R. Motwani, L. Page, and T. Winograd. What can you do with a web in your pocket? *IEEE Bulletin of the Technical Committee on Data Engineering*, 21(2):37–47, June 1998.
6. S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
7. R. S. Burt. *Toward a Structural Theory of Action: Network Models of Social Structure, Perception, and Action.* Academic Press, 1982.
8. S. Chakrabarti, B. E. Dom, R. Kumar, P. Raghavan, S. Rajagopalan, A. S. Tomkins, D. Gibson, and J. M. Kleinberg. Mining the Web's link structure. *IEEE Computer*, 32(8):60–67, 1999.

9. S. Chakrabarti, B. E. Dom, P. Raghavan, S. Rajagopalan, D. Gibson, and J. M. Kleinberg. Automatic resource compilation by analyzing hyperlink structure and associated text. *Computer Networks and ISDN Systems*, 30(1–7):65–74, 1998.
10. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
11. L. C. Freeman. Centrality in social networks: Conceptual clarification. *Social Networks*, 1:215–239, 1979.
12. N. E. Friedkin. Theoretical foundations for centrality measures. *American Journal of Sociology*, 96(6):1478–1504, 1991.
13. P. Gajer, M. T. Goodrich, and S. G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In *Proc. 8th Intl. Symp. Graph Drawing*, LNCS 1984:211–221. Springer, 2001.
14. G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins University Press, 3rd ed., 1996.
15. D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. In *Proc. 8th Intl. Symp. Graph Drawing*, LNCS 1984:183–196. Springer, 2001.
16. T. H. Haveliwala. Efficient computation of pagerank. Technical Report 1999-31, Database Group, Stanford University, 1999.
17. C. H. Hubbell. An input-output approach to clique identification. *Sociometry*, 28:377–399, 1965.
18. L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18:39–43, 1953.
19. M. Kaufmann and D. Wagner (eds). *Drawing Graphs: Methods and Models*, LNCS Tutorial 2025. Springer, 2001.
20. J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.
21. R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. S. Tomkins, and E. Upfal. Stochastic models for the Web graph. In *Proc. 41st Ann. IEEE Symp. Foundations of Computer Science*, pp. 57–65, 2000.
22. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
23. B. Mohar. Some applications of Laplace eigenvalues of graphs. In *Graph Symmetry*. NATO ASI Series C 497, pp. 225–275. Kluwer, 1997.
24. G. Pinski and F. Narin. Citation influence for journal aggregates of scientific publications: Theory, with applications to the literature of physics. *Information Processing and Management*, 12(5):297–312, 1976.
25. P. Pirolli, J. Pitkow, and R. Rao. Silk from a sow's ear: Extracting usable structures from the Web. In *Proc. ACM Conf. Human Factors in Computing Systems*, pp. 118–125, 1996.
26. D. A. Spielman and S.-H. Teng. Spectral graph partitioning works: Planar graphs and finite element meshes. In *Proc. 37th Ann. IEEE Symp. Foundations of Computer Science*, pp. 96–105, 1996.
27. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981.
28. W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society, Third Series*, 13:743–768, 1963.
29. C. Walshaw. A multilevel algorithm for force-directed graph drawing. In *Proc. 8th Intl. Symp. Graph Drawing*, LNCS 1984:171–182. Springer, 2001.
30. D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.

A Simple Linear Time Algorithm for Proper Box Rectangular Drawings of Plane Graphs*

Xin He

Department of Computer Science and Engineering
State University of New York at Buffalo, Buffalo, NY 14260, USA
xinhe@cse.buffalo.edu

Abstract. In this paper we introduce a new drawing style of a plane graph G, called *proper box rectangular* (PBR) drawing. It is defined to be a drawing of G such that every vertex is drawn as a rectangle, called a box, each edge is drawn as either a horizontal or a vertical line segment, and each face is drawn as a rectangle. We establish necessary and sufficient conditions for G to have a PBR drawing. We also give a simple linear time algorithm for finding such drawings. The PBR drawing is closely related to the *box rectangular* (BR) drawing defined by Rahman, Nakano and Nishizeki [17]. Our method can be adapted to provide a new algorithm for solving the BR drawing problem.

1 Introduction

The problem of "nicely" drawing a graph G has received increasing attention [6]. Such drawings are useful in visualizing planar graphs and find applications in fields such as computer graphics, VLSI layout, algorithm animation and so on. Among different drawing styles, the *orthogonal drawing* has attracted much attention due to its applications in circuit layouts, database diagrams, entity-relationship diagrams etc. [3,7,10,14,18,19,20,22]. A survey of this area was given in [17]. The definitions and examples given in this section are from [17].

In an orthogonal drawing of a plane graph G, each vertex is drawn as an integer grid point and each edge is drawn as a sequence of alternate horizontal and vertical line segments along grid lines as illustrated in Fig 1 (a). Every plane graph G with maximum degree ≤ 4 has an orthogonal drawing. However, a plane graph with maximum degree ≥ 5 has no orthogonal drawing.

A *box-orthogonal drawing* of a plane graph G is a drawing of G on an integer grid such that each vertex is drawn as a rectangle, called a *box*, and each edge is drawn as a sequence of alternate horizontal and vertical line segments along grid lines as illustrated in Fig 1 (b). Some of the boxes may be degenerated (i.e. points). A box-orthogonal drawing is a natural generalization of an orthogonal drawing. Every plane graph has a box-orthogonal drawing even if its maximum degree is ≥ 5. The box-orthogonal drawing is studied in [1,4,5,8,15,21].

* Research supported in part by NSF Grant CCR-9912418.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 234–245, 2001.

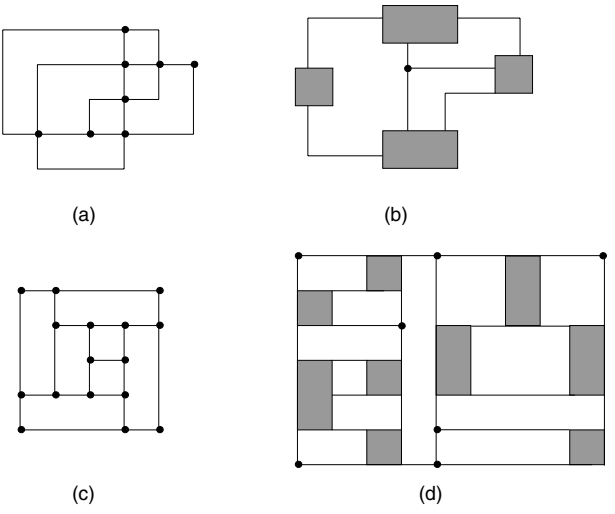


Fig. 1. (a) An orthogonal drawing, (b) a box-orthogonal drawing, (c) a rectangular drawing, and (d) a box-rectangular drawing.

An orthogonal drawing of a plane graph G is called a *rectangular drawing* if each edge of G is drawn as a straight line segment without bends and the contour of each face of G is drawn as a rectangle as illustrated in Fig 1 (c). Since rectangular drawings have applications in VLSI floor-planning, this subject has been extensively studied in [13,2,9,11,16].

A *box-rectangular* (BR) drawing was introduced in [17]: It is a drawing of G on an integer grid such that each vertex is drawn as a rectangle, called a *box* (which may be degenerated) and the contour of each face is drawn as a rectangle, as illustrated in Fig 1 (d). (A degenerated box will be called a *point*. A non-degenerated box will be called a *real box*). Several applications of BR drawings are given in [17]. A linear time algorithm for finding such a drawing is presented in [17]. Necessary and sufficient conditions for the existence of a BR drawing of a plane graph are also given in [17]. These conditions are rather complicated.

The BR drawing defined in [17] allows vertices be drawn as points. This is not desirable for applications such as floor-planning. A BR drawing is called a *proper box rectangular* (PBR) drawing if every vertex of G is drawn as a real box. In this paper, we establish necessary and sufficient conditions for a graph G to have a PBR drawing. We also present a linear time algorithm for constructing PBR drawings. Although BR drawing and PBR drawing are similar, our approach for solving this problem is totally different from that in [17]. Our necessary and sufficient conditions are logically simple and clean. Our algorithm is conceptually simpler. With slight modification, our method can be applied to the BR drawing also. This provides another linear time algorithm and another set of necessary and sufficient conditions for a graph to have BR drawing. Moreover, our method can be easily adapted to solve other similar drawing problems.

2 Preliminaries

Throughout the paper, $G = (V, E)$ denotes a connected plane graph with no selfloops, but may have multiple edges. Let $n = |V|$ and $m = |E|$. The embedding of G divides the plane into a number of regions. The unbounded region is called the *exterior face*. Other regions are called *interior faces*. The *degree* of a face F of G is the number of edges on its boundary.

Let $G^* = (V^*, E^*)$ denote the dual graph of G. For each edge e in G, e^* denotes its dual edge. For a subset $E_1 \subseteq E$, E_1^* denotes the set of the dual edges in G^* corresponding to the edges in E_1. If a cycle C of G contains k vertices, it is a k-cycle. A *triangle* (*quadrangle*, resp.) is a 3-cycle (4-cycle, resp.) A cycle C divides the plane into its interior region and exterior region. If there is at least one vertex in its interior region, C is called a *non-empty cycle* of G.

Consider a cut vertex v of G. Suppose v is an interior vertex. Let F be the interior face of G such that v appears on its boundary more than once. For each connected component D of $G - \{v\}$, the subgraph of G induced by $V(D) \cup \{v\}$ is called an *extended component* of $G - \{v\}$. Let H be an extended component of $G - \{v\}$ surrounded by F (see Fig 2 (a)). It is impossible to draw G on the plane so that the face F is a rectangle since there is no room to draw the vertices in H. Suppose v is an exterior vertex. Let H_1 be one extended component of $G - \{v\}$ and H_2 be the union of other extended components of $G - \{v\}$ (see Fig 2 (b)). In any PBR drawing of G, the box representing v must touch both the top and the bottom boundary (or both the left and the right boundary). Thus, in order to obtain a PBR drawing of G, we can recursively find a PBR drawing of H_1 while designating v as two adjacent corners (this forces the rectangle for v spans the whole length of the drawing of H_1) and a similar PBR drawing for H_2; then merge the rectangle for v in the drawing of H_1 and the rectangle for v in the drawing of H_2. This gives a PBR drawing of G (see Fig 2 (c)). Thus, without loss of generality, we will always assume G is biconnected from now on.

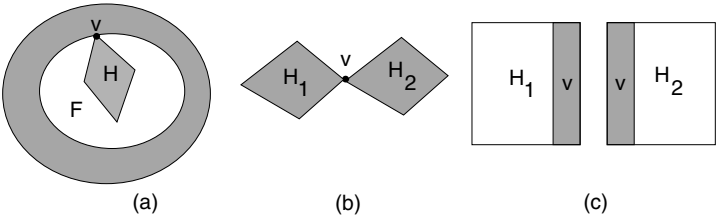


Fig. 2. (a) Interior cut vertex v, (b) exterior cut vertex v, (c) drawings of H_1 and H_2.

Our PBR drawing algorithm is based on the concept of the *rectangular dual* defined as follows. Let R be a rectangle. A *rectangular subdivision system* of R is a partition of R into a set $\Phi = \{R_1, \ldots, R_n\}$ of non-intersecting smaller rectangles such that no four rectangles in Φ meet at the same point. A *rectangular dual*

of a graph $G = (V, E)$ is a rectangular subdivision system Φ and a one-to-one correspondence $f : V \to \Phi$ such that two vertices u and v are adjacent in G iff their corresponding rectangles $f(u)$ and $f(v)$ share a common boundary. Fig 3 (c) illustrates a rectangular dual of the plane graph in Fig 3 (b).

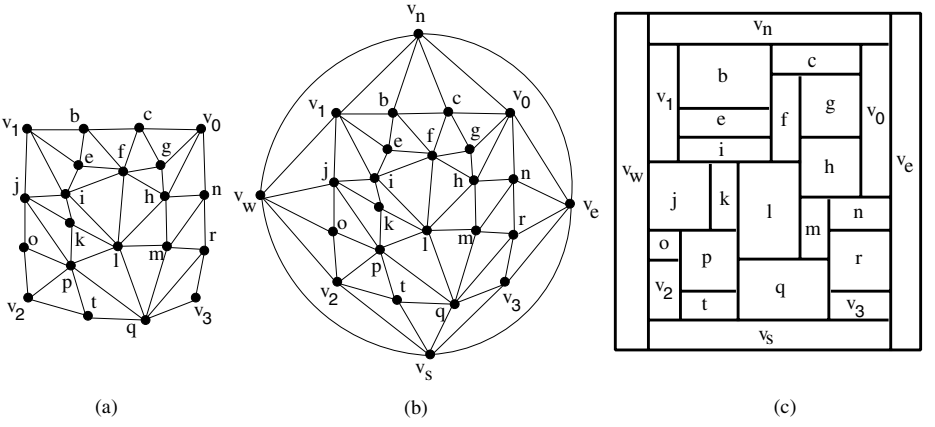


Fig. 3. Rectangular dual of plane graphs.

Consider a plane graph $H = (V, E)$. Let v_0, v_1, v_2, v_3 be four vertices on the exterior face of H in counterclockwise order. Let P_i $(i = 0, 1, 2, 3)$ be the four paths on the exterior face of H consisting of the vertices between v_i and v_{i+1} (addition is mod 4). We seek a rectangular dual R_H of H such that the four vertices v_0, v_1, v_2, v_3 correspond to the four corner rectangles of R_H and the vertices on P_0 (P_1, P_2, P_3, resp.) correspond to the rectangles located along the north (west, south, east, resp.) boundary of R_H. Necessary and sufficient conditions for testing if H has a rectangular dual were discussed in [2,12,13].

To simplify the problem, we modify H as follows: Add four new vertices v_n, v_w, v_s, v_e and connect v_n (v_w, v_s, v_e, resp.) to every vertex on P_0 (P_1, P_2, P_3, resp.) Then add new edges $(v_n, v_w), (v_w, v_s), (v_s, v_e), (v_e, v_n)$. Let G be the resulting graph. (Fig 3 (a) shows a graph H and Fig 3 (b) shows the corresponding graph G). It is easy to see that H has a rectangular dual R_H with v_0, v_1, v_2, v_3 corresponding to the four corner rectangles iff G has a rectangular dual R.

If G has a rectangular dual R, then every face of G, except the exterior face, must be a triangle (since no four rectangles of R meet at the same point). Moreover, since at least four rectangles are needed to fully enclose some non-empty area on the plane, any non-empty cycle of G must have length at least 4. The following theorem was given in [12].

Theorem 1. *A plane graph $G = (V, E)$ has a rectangular dual R with four rectangles on the boundary of R iff the following conditions hold: (1) Every*

interior face of G is a triangle and the exterior face of G is a quadrangle; (2) G has no non-empty triangles; and (3) G has no multiple edges.

A graph satisfying the three conditions in Theorem 1 is called a *proper triangular planar* (PTP for short) graph.

Theorem 2. [9,11] *Given a plane graph G, in linear time, we can check the conditions in Theorem 1, and construct a rectangular dual of G if these conditions are satisfied.*

3 PBR Drawing with Designated Corner Vertices

In this section we establish necessary and sufficient conditions for the existence of a PBR drawing of a plane graph G when the four corner vertices are designated. We also give a linear time algorithm for finding such a drawing if it exists.

Let G be a biconnected plane graph. Let v_0, v_1, v_2, v_3 be the four vertices on the exterior face (in counterclockwise order) of G designated as corner vertices. These designated vertices may be repeated. For example, if $v_0 = v_1$, then in the PBR drawing of G, a single box occupying two corners represents the vertex $v_0 = v_1$. We use P_i $(i = 0, 1, 2, 3)$ to denote the four paths on the exterior face of G consisting of the vertices between (and including) v_i and v_{i+1}.

Definition 1. Let G be a biconnected plane graph with designated corner vertices v_0, v_1, v_2, v_3. The *extended graph* G_x is obtained from G as follows:

1. Add four new vertices v_n, v_w, v_s, v_e and connect v_n (v_w, v_s, v_e, resp.) to every vertex on P_0 (P_1, P_2, P_3, resp.) Add four new edges $(v_n, v_w), (v_w, v_s), (v_s, v_e)$, (v_e, v_n). For each interior face F of G, add a new vertex v_F in F and connect it to the vertices on the boundary of F.
2. For each edge e in the original graph G, delete e and add a new edge between v_{F_1} and v_{F_2} where F_1 and F_2 are the two faces of G with e on their boundary.

Fig 4 (b) shows the extended graph G_x of the graph G in Fig 4 (a). The vertices of G will be called *original vertices*. The vertices (edges, resp.) added in Step 1 are called *added vertices* (edges resp.) of G_x and denoted by V_a (E_a, resp.) The edges introduced in Step 2 are called *dual edges* and denoted by E_d. Thus $G_x = (V \cup V_a, E_a \cup E_d)$. We list the structures of G_x below.

- After step 1, every interior face of the resulting graph $G_a = (V \cup V_a, E \cup E_a)$ is a triangle, and the exterior face of G_a is a quadrangle. Consider any edge $e = (u, v)$ in the original graph G. Let F_1 and F_2 be the two faces in G with e on their boundaries. In G_a, e is a diagonal of the quadrangle $Q = \{u, v, v_{F_1}, v_{F_2}\}$. In Step 2, $e = (u, v)$ is replaced by another diagonal (v_{F_1}, v_{F_2}) of Q. Thus, all internal faces of G_x are still triangles.
- Each original vertex is incident only to the added vertices in G_x. For each added vertex v_F that corresponds to an interior face F with degree k in G, the degree of v_F is $2k$ in G_x, and the edges incident to v_F alternate between the added edges and the dual edges.

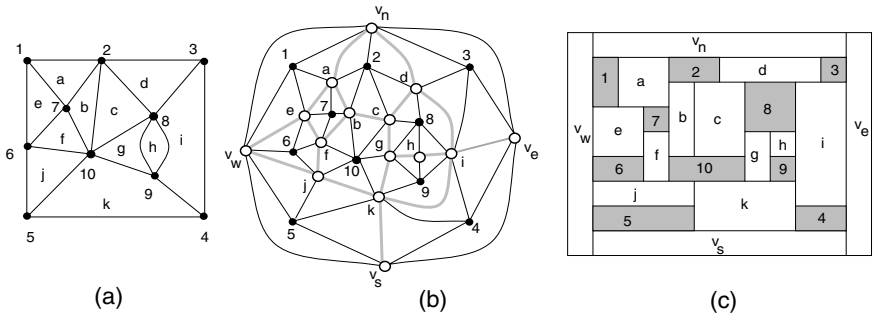


Fig. 4. (a) G, (b) G_x (the added vertices are drawn as empty circles, the dual edges are drawn as light curves), (c) a rectangular dual of G_x which is also a PBR drawing of G after removing the four rectangles corresponding to v_n, v_w, v_s, v_e.

Lemma 1. *G has a* PBR *drawing iff G_x has a rectangular dual.*

The proof is omitted due to space limitation.

The extended graph G_x can be easily constructed in linear time from the embedding data structure of G. Thus by Lemma 1, Theorems 1 and 2, we have:

Theorem 3. *Let G be a biconnected plane graph with four designated vertices v_0, v_1, v_2, v_3. In linear time, we can test whether G has a* PBR *drawing with v_0, v_1, v_2, v_3 as the four corner vertices, and construct one if it exists.*

Next we present necessary and sufficient conditions for the existence of a PBR drawing solely in terms of G. An *edge-cut* of $G = (V, E)$ is a *minimal* subset $X \subseteq E$ such that the graph $G - X = (V, E - X)$ is disconnected. If $|X| = k$, we say X is a k-edge-cut. A *2-mixed-cut* of G is a pair $\{v, e\}$ where $v \in V$ and $e \in E$ such that $G - \{e\} - \{v\}$ is disconnected.

Lemma 2. *Let $G = (V, E)$ be a biconnected plane graph and X a 2-edge-cut or a 3-edge-cut of G. Then X contains either 0 or 2 exterior edges of G.*

Proof. Consider the dual graph G^* of G. Let v_o^* be the vertex in G^* corresponding to the exterior face of G. For $k = 2$ or 3, a subset $X \subseteq E$ is a k-edge-cut of G iff its corresponding set X^* of dual edges is a k-cycle in G^*. If X contains an exterior edge of G, then v_o^* is on the cycle X^*. To complete the cycle, X^* must contain exactly two dual edges corresponding to two exterior edges of G. So X must contain exactly two exterior edges. □

Let $X = \{e_1, e_2\}$ be a 2-edge-cut (or $X = \{e_1, e_2, e_3\}$ a 3-edge-cut, respectively). By Lemma 2, X can be classified as follows:

- X is an *interior edge-cut* if all of its edges are interior edges;
- X is a *cross exterior edge-cut* if $e_1 \in P_i$, $e_2 \in P_j$ and $|i - j| = 2$.
- X is a *corner exterior edge-cut* if $e_1 \in P_i$, $e_2 \in P_j$ and $|i - j| = 1$ or 3.
- X is a *1-side exterior edge-cut* if $e_1 \in P_i$, $e_2 \in P_j$ and $i = j$.

Theorem 4. *A biconnected plane graph G has a* PBR *drawing with v_0, v_1, v_2, v_3 as the four corner vertices iff the following hold:*

1. *All 2-edge-cuts of G are cross exterior edge-cuts; and*
2. *All 3-edge-cuts of G are cross exterior or corner exterior edge-cuts; and*
3. *G has no 2-mixed-cut.*

Proof. **If:** If we can show the extended graph G_x is a PTP graph, then G has a PBR drawing by Theorem 1 and Lemma 1. As noted before, all interior faces of G_x are triangles and the exterior face of G_x is a quadrangle.

Next we show G_x has no multiple edges. Towards a contradiction, suppose G_x has two multiple edges which form a 2-cycle C. Suppose that C contains an added edge (v, v_F) in G_x between an original vertex v and an added vertex v_F. This can happen only if v appears on the boundary of the face F (of G) more than once. Then v is a cut vertex of G and this contradicts the assumption that G is biconnected. Thus C must consist of two dual edges e_1^* and e_2^*. Let e_1 and e_2 be the two edges in G corresponding to e_1^* and e_2^*, respectively. If both e_1 and e_2 are interior edges of G, then $\{e_1, e_2\}$ is an interior 2-edge-cut of G. This contradicts our assumption. Suppose e_1 is an exterior edge of G. Then C contains one of exterior vertices, say v_n, of G_x. Hence $e_1 \in P_0$. Since all dual edges incident to v_n corresponds to edges in P_0, e_2 must be in P_0 also. Thus $\{e_1, e_2\}$ is a 1-side exterior 2-edge-cut of G. This contradicts our assumption.

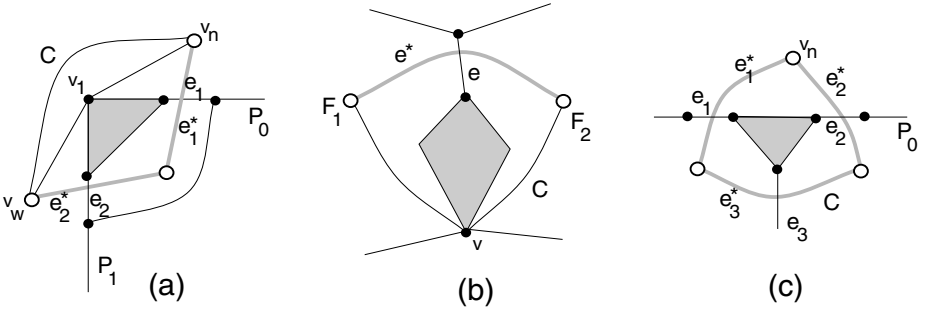


Fig. 5. The illustrations of the proof of the "if" part of Theorem 4.

Next we show that G_x has no non-empty triangles. Towards a contradiction, suppose G_x has a non-empty triangle C.

Case 1: C contains an exterior edge of G_x, say (v_n, v_w). The other two edges of C must share a common end vertex w in G_x. The only original vertex adjacent to both v_n and v_w in G_x is the corner vertex v_1. However, the triangle consisting of the three edges $(v_n, v_w), (v_n, v_1), (v_w, v_1)$ is an empty triangle in G_x. Thus the common vertex w must be an added vertex and the other two edges of C must be two dual edges e_1^* and e_2^*, where the edge e_1 corresponding to e_1^* is on P_0 and the edge e_2 corresponding to e_2^* is on P_1 (see Fig 5 (a)). Then $\{e_1, e_2\}$ is a corner exterior 2-edge-cut of G. This contradicts our assumption.

Case 2: C contains no exterior edge of G_x, but contains at least one added edge, say (v, v_{F_1}) between an original vertex v and an added vertex v_{F_1} in G_x. Since the original vertex v is adjacent to only the added vertices, C must consist of v and two added vertices v_{F_1} and v_{F_2} in G_x. Then the three edges of C are: (v, v_{F_1}), (v, v_{F_2}) and the dual edge $e^* = (v_{F_1}, v_{F_2})$. Let e be the edge of G corresponding to e^*. Then $\{v, e\}$ is a 2-mixed-cut of G (see Fig 5 (b)). This contradicts our assumption.

Case 3: C contains no added edges. Then C consists of three dual edges e_1^*, e_2^*, e_3^*. Let e_1, e_2, e_3 be the three edges in G corresponding to e_1^*, e_2^*, e_3^*, respectively. If e_1, e_2, e_3 are all interior edges, they form an interior 3-edge-cut of G. This contradicts our assumption. Suppose e_1 is an exterior edge, say on P_0. Then C contains v_n. Since all dual edges incident to v_n correspond to the edges in P_0, another edge of C must be in P_0. Then $\{e_1, e_2, e_3\}$ form a 1-side exterior 3-edge-cut (see Fig 5 (c)), a contradiction again.

Since all cases lead to contradictions, G_x has no non-empty triangles. So G_x is a PTP graph.

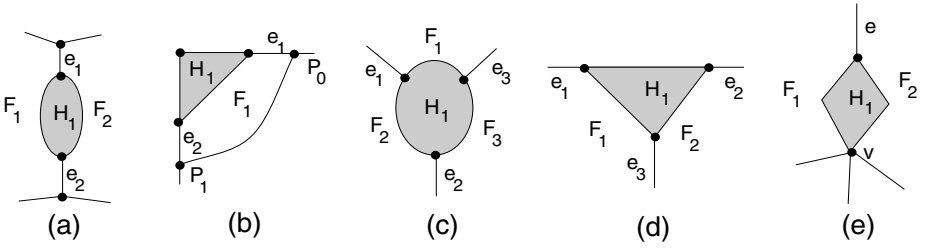


Fig. 6. The illustrations of the proof of the "only if" part of Theorem 4.

Only If: We show that if any of the three conditions fails, then G has no PBR drawing. Suppose that G has a 2-edge-cut X that is not a cross exterior edge-cut. There are three possible cases:

- X consists of two interior edges e_1, e_2. Let F_1, F_2 be the two faces of G with e_1 and e_2 on their boundary. Let H_1 and H_2 be the two connected components of $G - \{e_1, e_2\}$. It is impossible to draw the faces F_1, F_2 and the vertices in H_1 and H_2 as rectangles (see Fig 6 (a)).
- X consists of two exterior edges e_1, e_2 located on the paths P_i and P_j where $|i - j| = 1$ or 3. Let F_1 be the interior face of G with e_1 and e_2 on its boundary. Let H_1 and H_2 be the two connected components of $G - \{e_1, e_2\}$. It is impossible to draw the face F_1 and the vertices in H_1 and H_2 as rectangles (see Fig 6 (b)).
- X consists of two exterior edges e_1, e_2 located on the same path P_i. Let F_1 be the interior face of G with e_1 and e_2 on its boundary. Let H_1 and H_2 be the two connected components of $G - \{e_1, e_2\}$. It is impossible to draw the face F_1 and the vertices in H as rectangles.

Similarly, if G has a 3-edge-cut $X = \{e_1, e_2, e_3\}$ that is neither a cross nor a corner exterior edge-cut, or if G has a 2-mixed cut $\{v, e\}$, we can show G has no PBR drawing (see Fig 6 (c), (d) and (e).) □

4 PBR Drawing with No Designated Corner Vertices

In this section, we consider the case where no vertices of G are designated as corner vertices. Let $G = (V, E)$ be a biconnected plane graph. Let C_o be the cycle that form the exterior face of G. Let $u_1, u_2, \ldots, u_t$ be the exterior vertices of G in counterclockwise order. Let $e_1, e_2, \ldots, e_t$ be the exterior edges of G where $e_i = (u_i, u_{i+1})$ $(1 \leq i \leq t-1)$ and $e_t = (u_t, u_1)$.

We want to choose four vertices v_0, v_1, v_2, v_3 on C_o so that G has a PBR drawing with v_0, v_1, v_2, v_3 as corner vertices. To make this possible, we must choose the corner vertices so that the conditions in Theorem 4 are satisfied. If this is possible we say G is *feasible*. In this section, we describe a linear time algorithm for checking if G is feasible. Note that the notions of 2-mixed-cuts and internal edge-cuts are defined solely in terms of the plane graph G, and are independent from the choice of v_0, v_1, v_2, v_3. On the other hand, the notions of cross, corner and 1-side exterior edge-cuts depend on both G and the choice of v_0, v_1, v_2, v_3.

Let G^* be the dual graph of G. Let v_o^* be the vertex in G^* corresponding to the exterior face of G. Let e_i^* $(1 \leq i \leq t)$ be the dual edge in G^* corresponds to e_i. The *enlarged dual graph* of G, denoted by $\bar{G}^*$, is obtained from G^* as follows: (a) Remove the vertex v_o^* from G^*; (b) add a cycle C_o^* consisting of t new vertices v_i^* $(1 \leq i \leq t)$; (c) make the edge e_i^* $(1 \leq i \leq t)$ incident to v_i^*. See Fig 7 (a). Note that if we shrink the cycle C_o^* into a single vertex (and delete selfloops), then $\bar{G}^*$ becomes G^*. The method described in this section can be presented using G^* only. However, it is easier to illustrate the ideas by using $\bar{G}^*$. Note that each vertex v_i^* $(1 \leq i \leq t)$ on C_o^* corresponds to the exterior edge e_i of G, and each exterior edge (v_{i-1}^*, v_i^*) on C_o^* corresponds to the exterior vertex u_i on C_o. An *arc* of $\bar{G}^*$ is a continuous section of C_o^*. Let A be an arc of $\bar{G}^*$. Later in this section by saying "choose a corner vertex in A", we mean "choose a vertex in C_o that corresponds to an edge in A as a corner vertex".

A *bridge 2-path* (3-path, resp.) of $\bar{G}^*$ is a path P in $\bar{G}^*$ consisting of two (three, resp.) edges such that its two end vertices are on C_o^* and its internal vertex (vertices, resp.) are interior vertices of $\bar{G}^*$. Let $\hat{G}^*$ be the subgraph of $\bar{G}^*$ consisting of C_o^* and the edges in all bridge 2- and 3-paths (see Fig 7 (b)). It is easy to see that $\hat{G}^*$ can be obtained by first constructing the subgraph of $\bar{G}^*$ induced by the vertices on the cycle C_o^* and the vertices adjacent to C_o^*, then deleting all degree-1 vertices.

Note that a subset $X \subseteq E$ forms an exterior 2-edge-cut (3-edge-cut, resp.) iff the corresponding set X^* of dual edges forms a bridge 2-path (3-path, resp.) of $\hat{G}^*$. For any bridge path P, the two end vertices u, v of P divide C_o^* into two arcs A_1 and A_2 (u and v belong to both A_1 and A_2), which are called the two arcs *defined by* P.

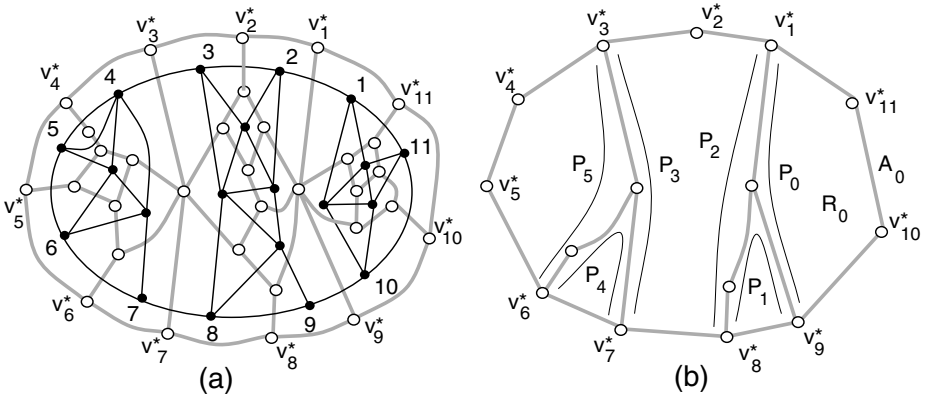


Fig. 7. (a) A graph G (drawn as solid lines and dark circles) and the corresponding graph $\bar{G}^*$ (drawn as empty circles and light lines); (b) the graph $\hat{G}^*$.

Let P be a bridge 2-path consisting of two dual edges e_i^*, e_j^*. Then $X = \{e_i, e_j\}$ is a 2-edge-cut of G. Let A_1 and A_2 be the two arcs defined by P. In order to make X a cross exterior edge-cut of G, we must choose two corner vertices on A_1 and two corner vertices on A_2.

Let P be a bridge 3-path consisting of three dual edges e_i^*, e_j^*, e^*. Then $X = \{e_i, e_j, e\}$ is an exterior 3-edge-cut of G. Let A_1 and A_2 be the two arcs defined by P. In order to make X a corner exterior edge-cut of G, we must choose either exactly one corner vertex on A_1 or exactly one corner vertex on A_2. In order to make X a cross exterior edge-cut of G, we must choose two corner vertices on A_1 and two corner vertices on A_2.

As we have seen, each exterior 2- or 3-edge-cut X of G corresponds to a bridge 2-path or 3-path of $\hat{G}^*$. To ensure that X satisfies the conditions in Theorem 4, it demands either 1 or 2 corner vertices on certain arcs. Since we can choose exactly four corner vertices, if the total demand for corner vertices exceeds 4, then G is not feasible. In the following, we describe how to verify this condition.

Let P be a bridge path and A_1 and A_2 be the two arcs defined by P. We say that another bridge path P' is on A_1 if both end vertices of P' are in A_1. P is called a *primary* bridge path if no other bridge paths are on one arc defined by P, and this arc is the *empty arc* of P. (If there is only one bridge path P in $\hat{G}^*$, then P is a primary bridge path and both arcs defined by P are empty arcs.)

Let $\hat{G}_2^*$ be the subgraph of $\hat{G}^*$ consisting of the cycle C_o^* and all bridge 2-paths. Let H be the graph obtained from $\hat{G}_2^*$ as follows. For each internal face F in $\hat{G}_2^*$, there is a node v_F in H. Two nodes v_{F_1} and v_{F_2} are adjacent in H iff the two corresponding faces F_1 and F_2 share a common boundary in $\hat{G}_2^*$. $\hat{G}_2^*$ is a *ladder* if H is a path. The two faces of $\hat{G}_2^*$ corresponding to the two end nodes of H are called the two *end regions* of $\hat{G}_2^*$.

For an example, consider the graph shown in Fig 7 (b). $\hat{G}^*$ has four primary bridge paths: P_0, P_1, P_4 and P_5. The corresponding graph $\hat{G}_2^*$ consists of the

cycle C_o^* and the bridge 2-paths P_0 and P_3. The corresponding graph H is a path consisting of three nodes. So $\hat{G}_2^*$ is ladder. One end region of $\hat{G}_2^*$ is the region bounded by the arc $\{v_1^*, v_{11}^*, v_{10}^*, v_9^*\}$ and the path P_0. The other end region of $\hat{G}_2^*$ is the region bounded by the arc $\{v_3^*, v_4^*, v_5^*, v_6^*, v_7^*\}$ and the path P_3.

Theorem 5. *G is feasible iff one of the following two conditions holds:*

1. *$\hat{G}^*$ has no bridge 2-paths and there are at most four primary bridge 3-paths.*
2. *$\hat{G}^*$ has at least one bridge 2-path, and (a) $\hat{G}_2^*$ is a ladder; and (b) All primary bridge 3-paths are in the two end regions of $\hat{G}_2^*$; and (c) Each end region contains at most two primary bridge 3-paths.*

Proof. (1) Assume $\hat{G}^*$ has no bridge 2-paths. Suppose that there are at most $l \leq 4$ primary bridge 3-paths. We can choose one corner vertex on the empty arc of each primary bridge 3-path. (If $l < 4$, the other $4 - l$ corner vertices can be chosen arbitrarily). This way, for any bridge 3-path P, at least one corner vertex is chosen on each of the two arcs defined by P. Thus G is feasible.

Suppose that there are more than four primary bridge 3-paths. Regardless how we choose the four corner vertices, for at least one primary bridge 3-path P, no corner vertex is chosen on the empty arc of P. Thus G is not feasible.

(2) Assume $\hat{G}^*$ has at least one bridge 2-path.

Suppose the conditions 2 (a), 2 (b) and 2 (c) hold. We can choose one corner vertex on the empty arc of each primary bridge 3-path. If less then four corner vertices are chosen, we can choose the remaining corner vertices so that exactly two corner vertices are chosen in each of the two end regions of $\hat{G}_2^*$. This way, exactly two corner vertices are chosen in each arc defined by every bridge 2-path, and at least one corner vertex is chosen in each arc defined by every bridge 3-path. Hence G is feasible.

Suppose the condition 2 (a) fails. We must choose 2 corner vertices on the empty arc defined by every bridge 2-paths. Since $\hat{G}^*$ is not a ladder, there are at least 3 primary bridge 2-paths in $\hat{G}_2^*$. So we must choose at least six 6 vertices.

Suppose the condition 2 (b) or 2 (c) fails. When we try to choose at least 1 corner vertex on the empty arc defined by each primary bridge 3-path, and exactly 2 corner vertices on each arc defined by each bridge 2-path, we end up choosing at least 3 corner vertices on an arc defined by a bridge 2-path.

Thus if one of the conditions 2 (a), 2 (b) or 2 (c) fails, G is not feasible. □

In the graph $\hat{G}^*$ shown in Fig 7 (b), the condition 2 (b) fails and hence the corresponding graph G is not feasible. The proof of the following theorem is omitted.

Theorem 6. *We can test whether a biconnected plane graph G is feasible in linear time.*

References

1. P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE. Tran. on Comp.*, 49(8):826–840, 2000.

2. J. Bhasker and S. Sahni. A linear algorithm to find a rectangular dual of a planar triangulated graph. *Algorithmica*, 3:247–278, 1988.
3. T. Biedl and G. Kant. A better heuristic for orthogonal graph drawings. *Computational Geometry: Theory and Applications*, 9:159–180, 1998.
4. T. Biedl and M. Kaufmann. Area-efficient static and incremental graph drawings. In *Proc. 5th European Symp. on Algorithms*, LNCS 1284, pages 87–52, 1997.
5. G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In *Proc. of GD'99, LNCS 1731*, pages 297–310, 1999.
6. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice-Hall, Inc., 1999.
7. G. Di Battista, P. Liotta, and F. Vargiu. Spirality and optimal orthogonal drawings. *SIAM. J. Comput.*, 27(6):1764–1811, 1998.
8. U. Fossmeier and M. Kauffmann. Drawing high degree graphs with low bend numbers. In *Proc. Graph Drawing'95*, LNCS 1027, pages 254–266, 1995.
9. Xin He. On finding the rectangular duals of planar triangulated graphs. *SIAM J. Comput.*, 22(6):1218–1226, 1993.
10. G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16:4–32, 1996.
11. G. Kant and X. He. Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. *Theoretical Computer Science*, 172:175–193, 1997.
12. K. Koźmiński and E. Kinnen. Rectangular dual of planar graphs. *Networks*, 15:145–157, 1985.
13. K. Koźmiński and E. Kinnen. Rectangular dualization and rectangular dissections. *IEEE Trans. on Circuits and Systems*, 35(11):1401–1415, 1988.
14. A. Papakostas and I. G. Tollis. Algorithms for area-efficient orthogonal drawings. *Computational Geometry: Theory and Applications*, 9(1/2):83–110, 1998.
15. A. Papakostas and I. G. Tollis. Efficient orthogonal drawings of high degree graphs. *Algorithmica*, 26:100–125, 2000.
16. M. S. Rahman, S. Nakano, and T. Nishizeki. Rectangular grid drawings of plane graphs. *Comp. Geom. Theo. Appl.*, 10(3):203–220, 1998.
17. M. S. Rahman, S. Nakano, and T. Nishizeki. Box-rectangular drawings of plane graphs. Journal of Algorithms, 37, pp. 363-398, 2000.
18. M. S. Rahman, S. Nakano, and T. Nishizeki. A linear algorithm for bend-optimal orthogonal drawings of triconnected cubic plane graphs. *J. of Graph Algorithms and Applications*, 3(4):31–62, 1999.
19. J. M Six, K. G. Kakoulis, and I. G. Tollis. Refinement of orthogonal graph drawings. In *Proc. Graph Drawing (GD'98)*, pages 302–315, 1999.
20. R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Computing*, 16(3):421–444, 1987.
21. R. Tamassia, G. Di Battista, and C. Batini. Automatic graph drawing and readability of diagrams. *IEEE Tran. Sys., Man, and Cybernetics*, 18(1):61–79, 1988.
22. R. Tamassia and I. G. Tollis. Planar grid embedding in linear time. *IEEE Trans. Circuits and Systems*, 36:1230–1234, 1989.

Short and Simple Labels for Small Distances and Other Functions

Haim Kaplan[1] and Tova Milo[1]

School of computer science, Tel Aviv University, Tel Aviv, Israel.
{haimk,milo}@math.tau.ac.il

Abstract. We present a labeling scheme for rooted trees which allows to compute, from the label of v alone, unique identifiers for the ancestors of v that are at distance at most d from v. For any constant d our labeling scheme produce labels of length $\log n + O(\sqrt{\log n})$, and for $d \in O(\sqrt{\log n})$ the labels are still of length $O(\log n)$.
In particular, given the labels of two nodes u and v we can determine from the labels alone whether u is the parent of v or vice versa, whether u and v are siblings, and whether u and v are at distance at most d from each other.
The need for such labeling scheme arises in several application areas, including in particular communication networks and search engines for large collections of Web XML files. In the latter application XML files are viewed as trees, and typical queries ask for XML files containing a particular set of nodes with specific ancestor, parent, or sibling relationships among them.

1 Introduction

We present an algorithm that given a tree T assigns a unique identifier, $\mathrm{id}(v)$, and a unique label, $\mathrm{lab}(v)$, for every $v \in T$. These identifiers and labels have the property that given the label of a node v we can compute from this label alone the identifiers of all ancestors of v which are at distance at most d from v. The maximum length of a label or identifier that our algorithm produce is $\log n + O(d\sqrt{\log n})$ bits, where n is the number of nodes in the tree. Thus for every constant d our labels are of length $\log n + O(\sqrt{\log n})$. This is close up to a second order additive term to the minimum of $\log n$ bits required to identify the nodes. Given the label of v we can construct the labels of all d ancestors of v starting from v in $O(1)$ time per ancestor.

In particular, our labeling scheme (with $d = 1$) allows parent and sibling queries. I.e. given the labels of two nodes u and v we can determine in constant time whether one is the parent of the other, and whether they are siblings (have a common parent). Furthermore, given the labels of two nodes u and v we can determine whether the distance between u and v is at most d. In case the distance is indeed at most d we can calculate it exactly from the two labels. This latter result is in contrast with a recent lower bound of $\Omega(\log^2 n)$ on the maximum label length if the labeling scheme allows to compute the distance between any

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 246–257, 2001.

pair of nodes [6]. Our result of this paper shows that if we limit ourselves to small distances then much shorter labels suffice.

Our labeling scheme is such that from the label of v alone we can compute the identifiers of the d closest ancestors of v. Since according to our scheme the label of a node v contains its identifier we can use it to determine if a node u is an ancestor of a node v at distance at most d from v from the labels of u and v[1]. In contrast, one may be interested in a labeling scheme that allow to determine whether u is an ancestor of v, using the labels of both v and u, for any pair of nodes u and v without any restriction on their distance. We call a labeling scheme that allows such queries a *labeling scheme for ancestor queries*. Notice that a labeling scheme for ancestor queries does not have any immediate implications for distance queries. The labeling scheme we describe in this paper does not support arbitrary ancestor queries. However, we can combine the technique presented in this paper with a recent labeling scheme for ancestor queries suggested by Abiteboul et al [2] and subsequently improved by Alstrup and Rauhe [3]. The resulting labeling scheme will then allow to identify the d closest ancestors of any node, and, additionally, to answer ancestor queries between *every* pair of nodes. Although the combined scheme is more complicated than the one we present here, the maximum label length is still $\log n + O(d\sqrt{\log n})$ (the constant factors are somewhat worse.)

Our result builds upon the tree decomposition technique developed by Abiteboul at el in [2]. We decompose the tree by applying the tree decomposition algorithm recursively $\sqrt{\log n}$ times. Based on the resulting decomposition of the tree we assign unique identifiers and labels to the nodes. The identifier of a node v is a concatenation of identifiers of the representatives of v from the $\sqrt{\log n}$ recursive levels. The main technical contributions of the paper are the followings. 1) We separate the notions of an identifiers of a node and a label of a node, and 2) We develop a technique that exploits the recursive structure of the identifiers and labels to compute the identifiers of the ancestors of v from the label of v. To simplify the presentation, we demonstrate our technique in this extended abstract using particularly simple identifiers that allows to identify the ancestors of a node v that are at distance at most d from v. But as already mentioned, we can combine our technique with more complicated identifiers derived from the work of Abiteboul et al [2] and Alstrup and Rauhe [3], to obtain a scheme that also supports ancestor queries.

1.1 Related Work

Early work on labeling schemes for graphs focused on *adjacency labeling schemes* where we want to determine whether u and v are adjacent based solely on their labels. Kannan, Naor, and Rudich [10] suggested $O(\log n)$ adjacency labeling schemes for a number of graph families, including trees, and various intersection graph families.

[1] We compute from the label of v the identifiers of its d closest ancestors and check whether one of them is the identifier of u.

For trees an adjacency labeling scheme is a labeling scheme that allows parent queries. Kannan et al [10] suggested the following simple scheme which we call the *pair scheme*. According to the *pair scheme* we number the nodes of T in some arbitrary order, by consecutive integers starting from 1. Then we label each internal node v by a pair $(n(v), n(p(v)))$, where $n(v)$ and $n(p(v))$ are the numbers of v and its parent respectively. Clearly node v is a parent of node u if v's first attribute equals to u's second attribute. It is easy to see that the length of the labels generated by this scheme is at most $2\log(n)$. The labeling scheme we suggest in this paper (with $d = 1$) improves substantially on the $2\log n$ bound of Kannan et al, and produce labels of length $\log n + O(\sqrt{\log n})$ bits. One can view our technique as an extension of the simple pair scheme. At a high level our algorithm partitions the tree into small subtrees and uses the pair scheme within each such subtree with special care of how to glue these independent pair schemes together.

The notion of *adjacency labeling scheme* has been extended by Peleg [9] who studied various *distance labeling schemes*. A distance labeling scheme allows to determine, or at least approximate, the distance between u and v from their labels. Peleg [9] gave a distance labeling scheme for trees producing labels of length $O(\log^2 n)$, and an approximate distance labeling scheme for general graphs. The approximate scheme for general graphs produce distances that are accurate up to a factor of $\sqrt{8k}$ with labels of length $O(\log^2 nkn^{1/k})$. Further extensive work on distance labeling schemes has been done by Gaviolle et al [6] and Peleg et al [8]. These papers establish lower and upper bounds on the label length for several graph families. In particular Gaviolle et al [6] show a lower bound of $\Omega(\log^2 n)$ on the label length of any distance labeling scheme for trees. Our result of this paper shows that if we limit ourselves to small distances then labels much shorter than $O(\log^2 n)$ suffice.

Recent work has also been focused on labeling scheme for ancestor queries. A simple such labeling first suggested by Santoro and Khatib [11] produce labels of length at most $2\log n$. Abiteboul et al [2] have recently presented a labeling scheme for ancestor queries with labels of length $\frac{3}{2}\log n + O(\log\log n)$. Alstrup and Rauhe [3] subsequently showed how to improve the bound of Abiteboul et al, to $\log n + O(\sqrt{\log n})$, by incorporating alphabetic trees [7] into their algorithm. A similar scheme has also been discovered independently by Zwick and Thorup [12] in the context of a routing application. As already mentioned our result builds upon the tree decomposition technique of Abiteboul et al as presented in [2]. We can obtain a labeling scheme for parent queries by taking the ancestor labeling scheme of Alstrup and Rauhe [3] and add to the label of v the height of v in the appropriate subtree that contains it. The label length would still be $\log n + O(\sqrt{\log n})$. This approach, however, does not seem to extend to allow sibling queries or distance queries, even for small distances.

1.2 Motivation

Applications for most of the labeling schemes that we mentioned in Section 1.1, as well as the new schemes suggested in this paper arise from two main areas.

The first is routing in communication networks and the second is the design of search engines for large collections of XML files.

Communication Networks. Distance labeling schemes are useful in situations when a router knows the label[2] of the destination but does not know or does not have the time to access the topology of the entire network. Such router may want to choose between several routing options (or routing algorithms) based on the distance to the destination. In case the labels are carefully designed to allow distance or limited distance queries, the router can figure out the distance to the destination without accessing the topology of the network. A concrete application of distance labeling schemes for connection establishment in ATM networks is described in [9].

A distance labeling scheme can also be useful to limit flooding during broadcast or multicast of information. In such context a router may want to switch to unicast or TTL (Time To Live) based broadcast when all destinations reachable from it are relatively close. A close relation between ancestor labeling schemes and routing in trees is suggested by the work of Zwick and Thorup [12].

Note that when the label of the destination (or labels of the destinations) is transmitted with the packet, long labels will increase the bandwidth. Each node also stores the labels of all destinations it may need to send data to, so if the labels are long these tables may become too large. Consequently, one would like to use as compact labels as possible.

XML search engines. The Web is constantly growing and contains a huge amount of useful information. To retrieve such data, people typically use search engines which provide full-text indexing services (the user gives a few words and the engine returns documents containing those words). The new emerging XML Web-standard [13] allows for the development of advanced search engines that support more sophisticated queries. XML allows to describe the semantic nature of the document components, enabling users not only to ask full-text queries but also utilize the document structure to ask for more specific data [15].

The key observation is that Web documents obeying the XML standard can be viewed as trees (basically the parse tree of the document). Typical queries over such documents then amount to testing various inclusion and direct inclusion of document items, which correspond to ancestor, parent, sibling, and similar relationships among the analogous tree nodes [5,14,1].

XML Web-search engines process such queries using an index structure that summarizes this structural information [16]. Each node in the document tree is represented in the index by some logical id (label), with the labels being assigned to the nodes such that, given the labels of two nodes, the index can determine fast (by looking only at the labels and without any access to the actual file) whether one node is the parent or the ancestor of the other. To provide good performance it is essential that the index structure (or at least a large part of it) resides in main memory. Observe that we are talking here about huge numbers - thousands of giga bytes of labels [4]. Since a main factor which determines the index size is the length of the labels, reducing this length, even

[2] Labels can also be used to identify the nodes in the network.

by a constant factor, is a critical issue, contributing both memory cost reduction and performance improvement.

We use the RAM model of computation, and assume that the size of a computer word is $\Omega(\log n)$ (hence the basic operations on the labels can be performed in constant time). In particular we assume that the RAM model includes the ability to perform bit manipulations like shift, bitwise AND, OR, XOR, and base-two discrete logarithm in constant time.

The paper is organized as follows: In section 2 we recalling the main principle of the tree decomposition algorithm of [2]. Section 3 demonstrates our techniques by giving a very simple scheme for parent queries using just one iteration of the tree decomposition algorithm. Section 4 shows how to reduce the length of the labels by adapting our technique to recursive pruning of the tree. Finally we show in Section 5 how to extend our ideas so we can compute identifiers for d closest ancestors. We conclude and suggest directions for further research in Section 6. Most proofs are omitted from this extended abstract.

2 The Prune&Contract Algorithm

In this section we give an overview of the tree decomposition algorithm of [2] and its properties. This will serve later as the basis of our new labeling scheme. We refer to this algorithm as the *prune&contract* algorithm. We assume that the input tree is such that each internal node has at least two children[3]. We denote by n the number of nodes in a tree T and by l the number of leaves. Since we assume that each node in T has at least two outgoing edges $n \leq 2l$.

The algorithm uses a parameter x that controls the size of the pruned subtrees, and has three phases. In our description below we omit some details that can be found in [2].

- The first phase prunes from T all subtrees having l^x leaves or less, by cutting the edges leading to these subtrees[4]. For example consider the tree in Figure 1(a). The tree has 27 leaves. When $x = \frac{1}{3}$ all subtrees containing three or less leaves are prunes. The dashed lines in Figure 1 indicate the pruned subtrees.
- The second phase groups the pruned subtrees into forests each having between l^x and $2l^x$ leaves[5]. We do the grouping such that the subtrees in each forest are either all children of a single node (as in the case of the forests B-E in Figure 1(b)), or they are the set of children of some path of nodes (as in the case of forest A; the nodes of the path here are denoted by a_1-a_3). The paths are such that their tail node is not a leaf of the remaining tree (in the above example a_3 is not a leaf - it has a non pruned child).

[3] If this is not the case to begin with we add a child to each internal node having only one child. This transformation at most doubles the number of nodes. (Since the labels lengths are in term of $\log n$ this adds at most one one bit to the label).

[4] For some technical reasons we in fact also cut some subtrees having more than l^x but still at most $2l^x$, for details see [2].

[5] Some limited number of forests may in fact contain a smaller number of leaves.

– The third phase adds to the remaining part of the tree a *representative-leaf* per each pruned forest. When the parents of the trees in the forest form a path we first contract the path into a single node and attach the new representative-leaf to that node. Otherwise we attach the leaf to the parent of the forest. An exception is when there is only one single forest below such a parent node. In this case rather than adding a new leaf, the parent itself serves as the representative of the forest. To continue with the above example, in Figure 1(c) the leaves a-e represent the forests A-E resp., and the node a^* corresponds to the contracted path a_1-a_3. The nodes $a, c-e$ are new leaves, while b belonged to the original tree.

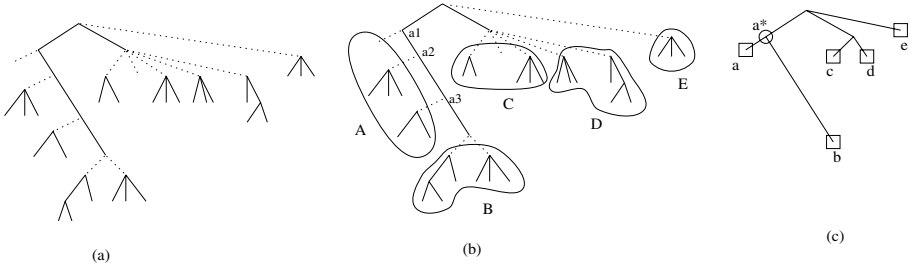


Fig. 1. The pruning algorithm

Let $\overline{T}$ denotes the tree resulting from the above algorithm after pruning, contraction, and adding representatives. The following Lemma proved in [2] bounds the number of leaves in $\overline{T}$ and the length of the contracted paths. We shall use these lemma to establish the bound on the label length.

Lemma 1. *[2] (1) The length of each path contracted while constructing $\overline{T}$ is at most $2l^x$. (2) The number of leaves in $\overline{T}$ is at most $3l^{1-x}$ where l is the number of leaves in T.*

We shall use the following definition of the *representative* of a node to define our labeling schemes.

Definition 1. *Let v be some node in T. The* representative *node of v in $\overline{T}$, denoted by $r(v)$, is defined as follows. If v was contracted as part of a path into some node c in $\overline{T}$, or belongs to some pruned forest under such path, then $r(v) = c$. If v was pruned from T (but not under a contracted path) then it is represented in $\overline{T}$ by the representative-leaf corresponding to the forest to which v belongs. Otherwise, i.e. if $v \in T \cap \overline{T}$, then $r(v) = v$.*
Finally, for each node $w \in \overline{T}$ that represents pruned vertices we denote by $F(w)$ the forest of nodes of T that w represents.

We number each contracted path sequentially starting from 0 such that nodes closer to the root have larger numbers. For a vertex v that belongs to a contracted path we define the *index of v* as its sequential number on its contracted path.

3 A Simple Two Level Parent Labeling Scheme

First we use the prune&contract algorithm with a parameter x that will be determined later to decompose the tree into pruned subtrees of size about n^x and a remaining tree $\overline{T}$ of size about n^{1-x}. We number the nodes of $\overline{T}$ and the nodes of each pruned forest independently. Using these numbers we define a unique identifier $\mathrm{id}(v)$ for each $v \in T$. The identifier of v consists of two fields. The first field is the number of $r(v)$ in $\overline{T}$. If v is in a pruned subtree then the second field contains the label of v in its pruned subtree. Otherwise, the second field contains the index of v on its contracted path (or zero in case v is not on a contracted path). The second field also contains a bit which is set iff it contains a number of a pruned node rather than an index or zero.

The label of v, $\mathrm{lab}(v)$, consists of $\mathrm{id}(v)$ together with an additional information that allows us to compute the identifier of the parent of v from the identifier of v. The definition of this additional information follows from the following lemma.

Lemma 2. *During the prune&contract algorithm exactly one of the following cases holds.*

1. *The vertices v and $p(v)$ are on the same contracted path. In this case v and $p(v)$ share the same representative in $\overline{T}$ and the index of $p(v)$ is one greater than the index of v.*
2. *Vertices v and $p(v)$ are in the same pruned subtree. In this case v and $p(v)$ share the same representative in $\overline{T}$ and have different numbers in the numbering of their pruned subtree.*
3. *Vertex v is in a pruned subtree and vertex $p(v)$ is on a contracted path. In this case v and $p(v)$ share the same representative in $\overline{T}$.*
4. *Vertex v is in a pruned subtree and is represented by $p(v)$.*
5. *Vertex v is either (1) the last node on a contracted path, (2) a root of a pruned subtree that is represented by a new leaf added to $\overline{T}$, or (3) belongs to $\overline{T}$. Vertex $p(v)$ either belongs to $\overline{T}$ or is the first vertex on a contracted path. In this case $r(p(v))$ is the parent of $r(v)$ in $\overline{T}$.*

The structure of $\mathrm{lab}(v)$ is as follows. In addition to $\mathrm{id}(v)$, $\mathrm{lab}(v)$ contains a *merge bit* which is set iff v and $p(v)$ have a common representative in $\overline{T}$ (cases (1-4) above). In case v and $p(v)$ do not have a common representative (case (5)) in $\overline{T}$ we also keep the number of $r(p(v))$ in $\overline{T}$. Otherwise, if v and $p(v)$ do have a common representative we also keep two bits indicating which of cases (1), (2), (3), or (4) occurred. When case (2) occurred we also keep the number of $p(v)$ in $F(r(p(v)))$, and when case (3) occurred we keep the index of $p(v)$ in its contracted path.

It is rather straightforward to check now that given $\mathrm{lab}(v)$ we can compute $\mathrm{id}(p(v))$: If the merge bit is not set we obtain $\mathrm{id}(p(v))$ by storing the number of $r(p(v))$ in $\overline{T}$ in the first field of $\mathrm{id}(p(v))$ and writing zero into the second field. If the merge bit is set then the way we obtain $\mathrm{id}(p(v))$ depends on which of the cases (1), (2), (3), or (4) happens. In case (1) we obtain $\mathrm{id}(p(v))$ by incrementing

the index in the second field of $\mathrm{id}(v)$. In case (2) we obtain $\mathrm{id}(p(v))$ by replacing the content of the second field in $\mathrm{id}(v)$ with the number of $p(v)$ in its pruned forest. In case (3) we replace the second field in $\mathrm{id}(v)$ with the index of $p(v)$ on its contracted path and turn off the prune bit. In case (4) we simply write zero in the second field.

The length of the Labels. By inspecting the definition of the labels it is easy to see that the largest labels are in cases (5), (3), and (2). One can bound the length of the label in each of these cases using Lemma 1. To balance out the lengths of these three kinds of labels we pick the pruning parameter to be 1/2. The maximum length of a label is then $\frac{3}{2}\log n + O(1)$.

In summary we proved in this section the following theorem

Theorem 1. *The labeling and identification scheme presented in this section produce labels of length $\frac{3}{2}\log n + O(1)$ bits such that from the label of a node v we can obtain the identifier of $p(v)$ in constant time.*

The following is an immediate corollary.

Corollary 1. *Using the labeling scheme described in this section one can determine from the labels of u and v alone whether one is the parent of the other, and whether u and v have a common parent.*

4 Smaller Labels via Recursion

In this section we show how to obtain shorter labels for parent queries by applying the prune&contract algorithm recursively. As in Section 3, we define a unique identifier $\mathrm{id}(v)$ for each node v. The label of v consists of the identifier of v together with additional information that allows us to compute the identifier of $p(v)$ from the identifier of v. By Applying the prune&contract algorithm recursively k times, we obtain k levels of pruned forests and a final remaining tree T_k. To identify a node v we need a number from each pruning level. However, taking advantage of the locality of the parent relation, we will see that to produce the identifier of the parent of v it suffices to store *one* additional piece of information from *one* particular pruning level. We first recall the recursive prune&contract process which is similar to the one of [2]. Then we explain how to use it in order to construct the identifiers and the labels. By setting $k = \sqrt{\log n}$ and fixing appropriate pruning thresholds we obtain labels of length at most $\log n + O(\sqrt{\log n})$.

Recursive pruning. A prune&contract process with k recursive levels works as follows. Let $x_1, \ldots, x_k$ be a sequence of numbers that we shall fix later. First we process T using the prune&contract algorithm with x_1 as the pruning threshold. Let $T_1 \equiv \overline{T}$ be the resulting tree. Then we apply the prune&contract algorithm again to T_1 with a threshold x_2 and denote the resulting tree $\overline{T_1}$ by T_2. We continue recursively and at each step define T_i, $1 \leq i \leq k$, to be the tree $\overline{T_{i-1}}$ obtained by applying the prune&contract algorithm to T_{i-1} with threshold x_i. (With T_0 being the original tree T.) We extend the definition of a representative,

$r(v)$, of a node $v \in T$ from Section 2 such that $r^0(v) = v$ and $r^i(v) = r(r^{i-1}(v))$ for any $1 \leq i \leq k$.

As before we number the nodes of T_k consecutively starting from one. Then we do the same for the pruned forests: For each leaf or contracted node $w \in T_i$ such that $F(w)$ is not empty we number the nodes in $F(w)$ consecutively starting from one. Based on these numberings we define the identifiers and the labels of the nodes of T.

We start by defining the identifier of a node v, $\mathrm{id}(v)$. The identifier of v consists of $k+1$ fields ordered from field k down to field 0, and denoted by $f_k, \ldots, f_0$. Field f_i corresponds to the i-th iteration of the prune&contract algorithm. Field f_i starts with a *prune bit* that is set iff $r^i(v)$ was pruned. If $r^i(v)$ was pruned then this bit is followed by the number of $r^i(v)$ in its pruned forest. Otherwise, the prune bit is followed by the index of $r^i(v)$ on its contracted path or by zero if $r^i(v)$ was not contracted. It is easy to prove by induction on the number of iterations of the prune&contract algorithm that these identifiers are indeed unique.

The label of v consists of the identifier of v together with additional information that is needed to produce the identifier of the parent of v from the identifier of v. To define that additional information we need the following definition of the *merge-level* of a vertex v, and the subsequent lemmas.

Definition 2. *We define the* merge-level of a vertex $v \in T$ *as the maximum* j *such that* $r^j(v) \neq r^j(p(v))$. *We denote the merge level of a vertex* v *by* $m(v)$.

Let $m = m(v)$, the following lemma characterizes the possible relations between $r^m(v)$ and $r^m(p(v))$.

Lemma 3. *If* $m = m(v) < k$ *then exactly one of the followings occurred,*

1. *Vertices* $r^m(v)$ *and* $r^m(p(v))$ *are in the same pruned subtree of* T_m.
2. *Vertices* $r^m(v)$ *and* $r^m(p(v))$ *are in the same contacted path of* T^m.
3. *Vertex* $r^m(v)$ *is in a pruned subtree of* T^m *and vertex* $r^m(p(v))$ *is on a contracted path of* T^m.
4. *Vertex* $r^m(v)$ *is the root of a pruned subtree of* T^m *and* $r^m(p(v)) = r^{m+1}(p(v))$ *is a leaf of* T_{m+1} *representing* v.

If $m(v) = k$ *then* $r^k(p(v))$ *is the parent of* $r^k(v)$.

The following lemma characterizes what can happen to $r^i(v)$ and $r^i(p(v))$ when $i < m(v)$. Its proof follows from the definition of $m(v)$ and the definition of the prune&contract algorithm.

Lemma 4. *(i) For every* $i < m(v)$ *either* $r^i(p(v)) = r^{i+1}(p(v))$ *or* $r^i(p(v))$ *is the first node on a contracted path to which* $r^{i+1}(p(v))$ *corresponds.*

(ii) For every $i < m(v)$ *either (1)* $r^i(v) = r^{i+1}(v)$, *(2)* $r^i(v)$ *is the last vertex on a contracted path to which* $r^{i+1}(v)$ *corresponds, or (3)* $r^i(v)$ *is a leaf added to* T_i *to represent a pruned forest,* $r^{i-1}(v)$ *is a root of a tree in that forest.*

Now we are ready to define the additional information, denote by addi(v), that we add to id(v) in order to produce the identifier of $p(v)$ from the identifier of v. The label of v consists of both id(v) and addi(v). The first field of addi(v) contains the merge-level of v. We denote this merge level by m. If $m(v) = k$ then the second field contains the number of $r(p(v))$ in T_k. Otherwise the second field contains two bits indicating which of the cases in lemma 3 occurred. If Case (1) occurred then addi(v) contains also the number of $r^m(p(v))$ in its pruned forest. If Case (3) occurred then addi(v) contains also the index of $r^m(p(v))$ on its contracted path.

Given id(v) and addi(v) we compute id($p(v)$) as follows. If the merge level of v is k then field f_k of id($p(v)$) contains the number of $r^k(p(v))$ in T_k which is stored in addi(v), and all fields f_i for $i < k$ of id($p(v)$) are zero. So let $m = m(v)$ and assume $m < k$. By the definition of the identifiers, and the definition of the merge level, fields f_i for $i > m$ of id($p(v)$), are the same as the corresponding fields of id(v). By the first part of Lemma 4 fields f_i of id($p(v)$) for $i < m$ are zero. To compute field f_m of id($p(v)$) we check addi(v) to see which of the cases of Lemma 3 occurred at iteration m of the prune&contract algorithm. If Case (1) occurred then field f_m of id($p(v)$) contains the number of $r^m(p(v))$ in its pruned forest. We copy this number from addi(v) where it is stored, and also turn on the prune bit of this field. If Case (2) occurred we obtain field f_m of id($p(v)$) by adding one to the index stored in field f_m of id(v). If Case (3) occurred field f_m of id($p(v)$) contains the index of $r^m(p(v))$ in its contracted path that is stored in addi(v), with the pruned bit turned off. If Case (4) occurred then field f_m of id($p(v)$), including the pruned bit is zero.

The length of the labels. Each field in the label is of fixed length that is determined by the maximum possible value that can be assigned to that field. Since addi(v) contains information from a single but arbitrary recursive level it follows that the total label length is minimized if we choose the pruning thresholds $x_1, \ldots, x_k$ to be the same. The label length then is $(1 + \frac{1}{k+1}) \log n + O(k)$. Therefore by setting $k = \sqrt{\log n}$ we get labels of length $\log n + O(\sqrt{\log n})$. Since we assumed that a computer word can hold $\log n$ bits then it is straightforward to check that we can compute id($p(v)$) from id(v) in constant time.

The following theorem and its corollary summarize the results of this section.

Theorem 2. *The identification and labeling scheme we presented in this section assigns identifiers and labels of length* $\log n + O(\sqrt{\log n})$ *to the nodes of a tree such that given the label of a node* v *we can compute the identifier of* $p(v)$ *in constant time.*

Corollary 2. *One can label the nodes of a tree* T *with labels of length* $\log n + O(\sqrt{\log n})$ *such that given the labels of* u *and* v *we can determine from the labels alone whether one is the parent of the other, and whether* u *and* v *are siblings.*

5 Obtaining the Identifiers of the d Closest Ancestors

We can extend the algorithm from the previous section to produce labels from which we can compute the identifiers of all ancestors of v which are at distance at most d from v. We use the recursive prune&contract process and the identifiers as defined in Section 4. The label of v contains $\mathrm{id}(v)$ and an extended additional field, $\mathrm{addi}_d(v)$, that will allow to compute the identifiers of d ancestors of v. The extended label of a vertex $\mathrm{lab}_d(v)$ consists of $\mathrm{id}(v)$ and $\mathrm{addi}_d(v)$.

The field $\mathrm{addi}_d(v)$ is a concatenation of the fields $\mathrm{addi}(p^i(v))$, for $0 \leq i \leq d-1$. Given $\mathrm{id}(v)$ and $\mathrm{addi}(v)$ we use the algorithm of Section 4 to compute $\mathrm{id}(p(v))$. Then we take $\mathrm{id}(p(v))$ and $\mathrm{addi}(p(v))$ and compute $\mathrm{id}(p^2(v))$ using the same algorithm. We continue in a similar way d times until we have computed $\mathrm{id}(p^d(v))$. Since each of the fields $\mathrm{addi}(p^i(v))$ is of length $O(\sqrt{\log n})$, the length of $\mathrm{addi}_d(v)$ is $O(d\sqrt{\log n})$, and our labels for computing the identifiers of d ancestors are of length $\log n + O(d\sqrt{\log n})$. We conclude with the main result of this Section.

Theorem 3. *There are sets of unique identifiers* $\mathrm{id}(v)$ *and unique labels* $\mathrm{lab}(v)$ *that one can assign to the nodes of the tree* T *such that from* $\mathrm{lab}(v)$ *we can compute* $\mathrm{id}(p^i(v))$ *for* $1 \leq i \leq d$. *The maximum length of an identifier or a label is* $\log n + O(d\sqrt{\log n})$ *bits.*

Our labeling scheme can be used for distance queries. Given the label of u and the label of v we can determine whether u and v are at distance at most d from each other. We do that by producing the identifiers of the d closest ancestors of v and the d closest ancestors of u. If there is no intersection among these lists of identifiers then we conclude that the distance between u and v is larger than d. Otherwise we find the identifier, $\mathrm{id}(z)$, of the lowest common ancestor, z, of u and v. By summing up the sequential number of z on the list of the ancestors of v and the sequential number of z on the list of the ancestors of u we obtain the distance between u and v. Notice that if u and v are of distance at most d from each other then their lowest common ancestor is of distance at most d from each of them and therefore it will show up in the list of the d closest ancestors to v and in the list of the d closest ancestors to u. So we have established the following corollary of Theorem 3.

Corollary 3. *The labeling scheme of Theorem 3 allows to determine the distance between* u *and* v *if this distance is less than* d *or figure out that the distance between* u *and* v *is larger than* d. *This labeling scheme also allows to determine whether* u *is an ancestor of* v *and at distance* j *from* v *for any* $j \leq d$.

6 Conclusions

We have presented an algorithm that assigns identifiers and labels to the nodes of a tree of length $O(\log n + d\sqrt{\log n})$ such that given a label of a node v one can compute the identifier of the d immediate ancestors of v. In particular our

labeling scheme allows to determine the distance between u and v from their labels if this distance is less than d, or figure out that the distance is larger than d.

When d is small then our labels are particularly short but as d gets larger our labels may become even linear in the number of nodes. In contrast, Peleg in [9] shows a labeling scheme for trees whose labels are of length $O(\log^2 n)$ which allows to determine any distance between u and v. So one may try to develop a labeling scheme, that works the same for every d, and is as efficient as ours for small d but degrades to no more than $O(\log^2 n)$ for larger values of d. Another line for further research is to implement our algorithms and test their performance on real XML data.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan-Kaufmann, 340 Pine Street, Sixth Floor San Francisco, CA 94104, October 1999.
2. S. Abiteboul, H. Kaplan, and T. Milo. Compact labeling schemes for ancestor queries. In *Proc. 12th Symp. Discrete Algorithms (SODA'01)*, January 2001.
3. Stephen Alstrup and Theis Rauhe, January 2001. Private communication at SODA'01.
4. D. Butler. Souped-up search engines. *Nature*, 405:112–115, May 2000.
5. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. In *International World Wide Web Conference*, 1999.
6. Cyril Gavoille, David Peleg, Stephane Perennes, and Ran Raz. Distance labeling in graphs. In *Proc. 12th Symp. Discrete Algorithms (SODA'01)*, January 2001.
7. T. C. Hu and C. Tucker. Optimum computer search trees. *SIAM J. Appl. Math.*, 21:514–532, 1971.
8. M. Katz, N. Katz, and D. Peleg. Distance labeling schemes for well-separated graph classes. In S. Tison H. Reichel, editor, *STACS'00*, volume 1170 of *Lecture Notes in Computer Science*, pages 370–381. Springer Verlag, 2000.
9. David Peleg. Proximity-preserving labeling schemes and their applications. In *Graph-Theoretic Concepts in Computer Science, 25th International Workshop, WG '99*, volume 1665 of *Lecture Notes in Computer Science*, pages 30–41. Springer Verlag, 1999.
10. S. Rudich S. Kannan, M. Naor. Implicit representation of graphs. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 334–343, 1988.
11. N. Santoro and R. Khatib. Labeling and implicit routing in networds. *The Computer J.*, 28:5–8, 1985.
12. M. Thorup and U. Zwick. Compact routing schemes. In *To appear in the Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.
13. W3C. Extensible markup language (xml) 1.0. http://www.w3.org/TR/REC-xml.
14. W3C. Extensible stylesheet language (xsl). http://www.w3.org/Style/XSL/.
15. W3C. The w3c query languages workshop, dec 1998, boston, massachussets. http://www.w3.org/TandS/QL/QL98/cfp.html.
16. Xyleme. A dynamic data warehouse for the xml data of the web. http://www.xyleme.com.

Fast Boolean Matrix Multiplication for Highly Clustered Data

Andreas Björklund[1] and Andrzej Lingas[2]

[1] Department of Computer Science, Lund Institute of Technology, 22100 Lund.
Andreas.Bjorklund@cs.lth.se.
[2] Department of Computer Science, Lund University, 22100 Lund.
Andrzej.Lingas@cs.lth.se.

Abstract. We consider the problem of computing the product of two $n \times n$ Boolean matrices A and B. For an $n \times n$ Boolean matrix C, let G_C be the complete weighted graph on the rows of C where the weight of an edge between two rows is equal to its Hamming distance, i.e., the number of entries in the first row having values different from the corresponding entries in the second one. Next, let $MWT(C)$ be the weight of a minimum weight spanning tree of G_C. We show that the product of A with B as well as the so called witnesses of the product can be computed in time $\tilde{O}(n(n + \min\{MWT(A), MWT(B^t)\}))$ [1].

1 Introduction

Since Strassen published his first subcubic algorithm for arithmetic matrix multiplication [1], a lot of work in this area has been done. The best asymptotic upper bound on the number of arithmetic operations necessary to multiply two $n \times n$ matrices is presently $O(n^{2.376})$ due to Coppersmith and Winograd [5]. Since Boolean matrix multiplication is trivially reducible to arithmetic $0-1$-matrix multiplication [1], the same asymptotic upper bound holds in the Boolean case. Unfortunately, the aforementioned substantially subcubic algorithms for arithmetic matrix multiplication are based on algebraic approaches difficult to implement.

In [9], Schnorr and Subramanian have shown that the Boolean product of two $n \times n$ random Boolean matrices can be determined by a simple combinatorial algorithm with high probability in time $\tilde{O}(n^2)$. Consequently, they raised the question of whether or not there exist a substantially subcubic combinatorial algorithm for Boolean matrix multiplication (the fastest known combinatorial algorithm for this problem is due to Bash et al. [3] and runs in time $O(n^3/\log^2 n)$).

In this paper, we give a further evidence that such a combinatorial algorithm might exists. Namely, we provide a combinatorial algorithm for Boolean matrix multiplication which is substantially subcubic in case the rows of the first $n \times n$ matrix or the columns of the second one are highly clustered, i.e., their minimum spanning tree in the Hamming metric has low cost. More exactly,

[1] $\tilde{O}(f(n))$ means $O(f(n)poly - \log n)$ and B^t stands for the transposed matrix B.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 258–263, 2001.

our algorithm runs in time $\tilde{O}(n(n+c))$ where c is the minimum of the costs of the minimum spanning trees for the rows and the columns, respectively, in the Hamming metric. It relies on the fast methods for computing an approximate minimum spanning tree in the L_1 and L_2 metrics given in [7,8].

We also observe that the simple combinatorial algorithm from [9] for the Boolean product of two $n \times n$ runs in time $O(n(n+r))$ if at least one of the matrices is r-sparse, i.e., contains at most r entries set to 1.

If an entry with indices $i,\ j$ of the Boolean product of two Boolean matrices A and B is equal to 1 then any index k such that $A[i,k]$ and $B[k,j]$ are equal to 1 is a *witness* to this. Quite recently, Alon and Naor [2] and Galil and Margalit [6] have shown that the witnesses for the Boolean matrix product of two $n \times n$ Boolean matrices (i.e., for all its nonzero entries) can be computed in time $\tilde{O}(n^{2.376})$ by repeatedly applying the aforementioned algorithm of Coppersmith and Winograd for arithmetic matrix multiplication [5]. The combinatorial algorithms for Boolean matrix multiplication presented in this paper yield the witnesses directly without any extra asymptotic time-cost.

Our paper is structured as follows. The next section presents known fact on approximating minimum spanning tree in the L_1 and L_2 metrics. Section 3 contains our algorithm for fast Boolean matrix multiplication for highly clustered data and its analysis. Finally, Section 4 presents the observations on Boolean matrix multiplication for sparse matrices.

2 Approximate MST in the Hamming Metric

For $c \geq 1$ and a finite set S of points in a metric space, an *c-approximate minimum spanning tree for S* is a spanning tree in the complete weighted graph on S, with edge weights equal to the distances between the endpoints, whose total weight is at most c times the minimum.

In [7] (section 4.3), Indyk and Motwani in particular considered the bichromatic ϵ-approximate closest pair problem for n points in R^d with integer coordinates in $O(1)$ under the L_p metric, $p \in \{1,2\}$. They showed that there is a dynamic data structure for this problem which supports insertions, deletions and queries in time $O(dn^{1/(1+\epsilon)})$ and requires $O(dn + n^{1+1/(1+\epsilon)})$-time preprocessing. In consequence, by a simulation of Kruskal's algorithm they deduced the following fact.

Fact 1. For $\epsilon > 0$, a $1+\epsilon$-approximate minimum spanning tree for a set of n points in R^d with integer coordinates in $O(1)$ under the L_1 or L_2 metric can be computed by a Monte Carlo algorithm in time $O(dn^{1+1/(1+\epsilon)})$.

In [8] Indyk, Schmidt and Thorup reported even slightly more efficient (by a poly-log factor) reduction of the problem of finding a $1+\epsilon$-approximate minimum spanning tree to the bichromatic ϵ-approximate closest pair problem via an easy simulation of Prim's algorithm.

3 Boolean Matrix Multiplication via MST

For an i-th row of A and a j-th column of B, the set of witnesses is the set of all indices k in $\{1, ..., n\}$ such that $A[i, k] = 1$ and $B[k, j] = 1$. An $n \times n$ matrix W such that $W[i, j]$ is the set of witnesses of the i-th row of A and the j-th column of B is called a witness matrix for the product of A and B.

The idea of our combinatorial algorithm for witnesses of Boolean matrix product is simple. First, we compute an approximate spanning tree of the rows of A (or, the columns of B, alternatively) in the Hamming metric. Then, we fix a traversal of the tree and precompute the set differences between the sets of entries set to one for consecutive neighboring rows in the traversal. Finally, for each column of B, we traverse the tree and compute the set of witnesses for the cdot product of the traversed row of A with the column of B from that for previously traversed row of A and the column of B.

Algorithm 1

Input: $n \times n$ Boolean matrices A and B;
Output: witnesses for the Boolean product of A and B.

1. for $i = 1, ..., n$ **do**
$A1_i \leftarrow \{k | A[i, k] = 1\}$
2. Compute an $O(\log n)$-approximate minimum weight spanning tree T_A of the graph G_A;
3. Fix a traversal of T_A of length linear in the size of T_A;
4. $i_0 \leftarrow$ the number of the row corresponding to the firstly traversed node of T_A;
5. $i \leftarrow i_0$;
6. while traversing the tree T_A **do**
begin
$l \leftarrow i$;
$i \leftarrow$ the number of the row of A corresponding to the next node of T_A on the traversal;
$D1_{i,l} \leftarrow A1_i \setminus A1_l$;
$D1_{l,i} \leftarrow A1_l \setminus A1_i$;
end
7. for $j = 1, ..., n$ **do**
$i \leftarrow i_0$
$W[i, j] \leftarrow$ the set of witnesses for the i-th row of A and j-th column of B;
while traversing the tree T_A **do**
begin
$l \leftarrow i$;
$i \leftarrow$ the number of the row of A corresponding to the next node of T_A on the traversal;
if i has been already visited **then go to** E;
$W[i, j] \leftarrow W[l, j]$;
for each $k \in D1_{i,l}$ **do**

if $B[k,j]=1$ **then** insert k into $W[i,j]$;
$W[i,j] \leftarrow W[i,j] \setminus D1_{l,i}$
if $W[i,j]$ is non-empty **then** output (i,j,w) where w is an element in $W[i,j]$
E: **end**

Lemma 1. *Algorithm 1 is correct, i.e., it outputs witnesses for the product of the Boolean matrices A and B.*

Lemma 2. *Algorithm 1 can be implemented in time $O(n(n+MWT(A)))+t(n)$ where $t(n)$ is the time taken by the construction of the $O(\log n)$-approximate minimum weight spanning tree in step 2.*

Proof. By representing the sets $A1_i$ of indices with monotone lists, we can compute each of the set differences $D1_{i,l}$, $D1_{l,i}$ in time $O(n)$. This combined with the linear in n length of the traversal T_A implies an $O(n^2)$-time implementation of steps 1 and 6. To implement step 7 efficiently we use search trees to form the sets $W[i,j]$. Then, the cost of transforming $W[l,j]$ into $W[i,j]$ becomes $O(|D1_{i,l} \cup D1_{l,i}| \log n)$ and consequently the overall time-cost of step 7 is $O(n(n+\log n \sum_i |D1_{i,l} \cup D1_{l,i}|))$ which is $\tilde{O}(n(n+MWT(A)))$.

Note that the L_1 metric for points in R^n with 0, 1-coordinates coincides with the n-dimensional Hamming metric. It follows from Fact 1 with ϵ set to $O(\log n)$ that an $O(\log n)$-approximate minimum spanning tree for a set of n vectors in the n-dimensional Hamming metric can be computed by a Monte Carlo algorithm in time $\tilde{O}(n^2)$. Also, the transposed product of matrices A and B is equal to the product of the transposed matrix B with the transposed matrix A. Hence, Lemmata 1, 2 yield our main result.

Theorem 1. *Let A, B be two $n \times n$ Boolean matrices. The product of A with B as well as witnesses for the product can be computed in expected time $\tilde{O}(n(n+\min\{MWT(A), MWT(B^t)\}))$ where B^t stands for the transposed matrix B.*

4 The Sparse Case

It is well known that if one of the two input $n \times n$ Boolean matrices to multiply is *sparse*, i.e., has the number r of entries set to 1 substantially smaller than n^2 then the witness matrix can be computed almost from the definition in substantially sub-cubic time. Simply, we can form for each row or column of the sparse matrix a list of indices of entries holding ones, and for each column or row of the other matrix a search tree for ones in it. Now, to determine the witnesses for the (i,j) entry of the output matrix is enough to search the indices in the list of ones in the i-th row of the first matrix in the search tree for ones in the j-th column of the other matrix or *vice versa*. Since each search costs $O(\log n)$, this method requires $O(n(n+r)\log n)$ operations. Note that the results of the previous section yield an analogous upper bound in the sparse case up to a logarithmic factor.

Interestingly, one can obtain a slightly better result in the sparse case by considering the so called column-row and row-column matrix representations (cf. [9]).

A *column-row* (CR for short) representation of a Boolean $n \times n$ matrix A is a sequence $\{R_k(A)\}_{k=1}^{n}$ such that $R_k(A)$ is a list of the numbers of the rows of A that have 1 on the k-th position (i.e., in the k-th column) in the increasing order. A *row-column* (RC for short) $C_k(A)$, $k = 1, ..., n$ representation of A is defined symmetrically. The following lemma is straightforward.

Lemma 3. *CR and RC representations of a Boolean $n \times n$ matrix A can be constructed in time $O(n^2)$.*

Theorem 2. *Let A, B be two Boolean $n \times n$ matrices. If A or B has at most r entries set to 1 then the sets of witnesses for the Boolean product of A and B can be determined in time $O(n(n + r))$.*

Proof. To compute the witness matrix W set all its entries initially to en empty set. Construct a CR representation $R_k(A)$, $k = 1, ..., n$, of A and an RC representation $C_k(B)$, $k = 1, ..., n$, of B. For $k = 1, ..., n$, all $i \in R_k(A)$ and $j \in C_k(B)$, augment $W[i, j]$ by (i, j). The correctness of setting W follows directly from the definitions of CR and RC. By Lemma 3, the number of operations performed is $O(n^2 + \sum_{k=1} |R_k(A)||C_k(B)|)$ which is $O(n^2 + nr)$.

5 Final Remark

It follows from the existence of the so called Hadamard matrices [4] that there is an infinite sequence of $n_i \times n_i$ matrices A_i, B_i such that the Hamming distance between any pair of rows of A_i or columns of B_i is $\Omega(n_i)$. Then, the cost of the corresponding minimum spanning trees is $\Omega((n_i)^2)$. Thus, our combinatorial algorithm for Boolean matrix multiplication presented in Section 3 does not break the cubic upper bound in the general case. However, in many applications of Boolean matrix multiplication where the rows or columns respectively tend to be more clustered the aforementioned scenario would be unlikely.

References

1. A.V. Aho, J.E. Hopcroft and J.D. Ullman. The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, Massachusetts, 1974).
2. N. Alon and M. Naor. Derandomization, Witnesses for Boolean Matrix Multiplication and Construction of Perfect hash functions. Algorithmica 16, pp. 434-449, 1996.
3. J. Basch, S. Khanna and R. Motwani. On Diameter Verification and Boolean Matrix Multiplication. Technical Report, Standford University CS department, 1995.
4. P.J. Cameron. Combinatorics. Cambridge University Press 1994.
5. D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. J. of Symbolic Computation 9 (1990), pp. 251-280.

6. Z. Galil and O. Margalit. Witnesses for Boolean Matrix Multiplication and Shortest Paths. Journal of Complexity, pp. 417-426, 1993.
7. P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. Proceedings of the 30th ACM Symposium on Theory of Computing, 1998.
8. P. Indyk, S.E. Schmidt, and M. Thorup. On reducing approximate mst to closest pair problems in high dimensions. *Manuscript*, 1999.
9. C.P. Schnorr and C.R. Subramanian. Almost Optimal (on the average) Combinatorial Algorithms for Boolean Matrix Product Witnesses, Computing the Diameter. Randomization and Approximation Techniques in Computer Science. Second International Workshop, RANDOM'98, Lecture Notes in Computer Science 1518, pp. 218-231.

Partitioning Colored Point Sets into Monochromatic Parts

Adrian Dumitrescu[1] and János Pach[2]

[1] University of Wisconsin at Milwaukee,
ad@cs.uwm.edu
[2] Courant Institute, NYU and Hungarian Academy of Sciences,
pach@cims.nyu.edu

Abstract. It is shown that any two-colored set of n points in general position in the plane can be partitioned into at most $\lceil \frac{n+1}{2} \rceil$ monochromatic subsets, whose convex hulls are pairwise disjoint. This bound cannot be improved in general. We present an $O(n \log n)$ time algorithm for constructing a partition into fewer parts, if the coloring is unbalanced, i.e., the sizes of the two color classes differ by more than one. The analogous question for k-colored point sets ($k > 2$) and its higher dimensional variant are also considered.

1 Introduction

A set of points in the plane is said to be in *general position*, if no three of its elements are collinear. Given a two-colored set S of n points in general position in the plane, let $p(S)$ be the minimum number of monochromatic subsets S can be partitioned into, such that their convex hulls are pairwise disjoint. Let

$$p(n) = \max\{p(S) : S \subset \mathbb{R}^2 \text{ is in general position }, |S| = n\}.$$

The following related quantities were introduced by Ron Aharoni and Michael Saks [8]. Given a set S of w white and b black points in general position in the plane, let $g(S)$ denote the number of edges in a largest non-crossing matching of S, where every edge is a straight-line segment connecting two points of the same color. Let

$$g(n) = \min\{g(S) : S \subset \mathbb{R}^2 \text{ is in general position}, |S| = n\}.$$

Aharoni and Saks asked if it is always possible to match all but a constant number of points, i.e., if $g(n) \geq n/2 - O(1)$ holds. It was shown in [8] that the answer to this question is in the negative. However, according to our next result (proved in Section 2), this inequality holds for $p(n)$. In other words: one can partition a set of n points in the plane into $n/2 + O(1)$ monochromatic parts whose convex hulls are disjoint; however if the size of each part is restricted to at most two points, this is not possible (there are configurations which require $(1/2 + \delta)n - O(1)$ parts, $\delta > 0$).

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 264–275, 2001.

Theorem 1. *Let $p(n)$ denote the smallest integer p with the property that every 2-colored set of n points in general position in the plane can be partitioned into p monochromatic subsets whose convex hulls are pairwise disjoint. Then we have*

$$p(n) = \left\lceil \frac{n+1}{2} \right\rceil .$$

For *unbalanced* colorings, i.e., when the sizes of the two color classes differ by more than one, we prove a stronger result. Slightly abusing notation, for any $w \leq b$, we write $p(w, b)$ for the minimum number of monochromatic subsets, into which a set of w white and b black points can be partitioned, so that their convex hulls are pairwise disjoint. Obviously, we have $p(w, b) \leq p(w + b)$.

Theorem 2. *For any $w \leq b$, we have*

$$w + 1 \leq p(w, b) \leq 2 \left\lceil \frac{w}{2} \right\rceil + 1. \tag{1}$$

More precisely,

(i) if $w \leq b \leq w + 1$ or w is even, we have $p(w, b) = w + 1$;
(ii) for all odd $w \geq 1$ and $b \geq 2w$, we have $p(w, b) = w + 2$;
(iii) for all odd $w \geq 3$ and $w+2 \leq b \leq 2w-1$, we have $w+1 \leq p(w, b) \leq w+2$.

We prove this theorem in Section 3, where we also present an $O(n \log n)$ time algorithm which computes a partition meeting the requirements.

For k-colored point sets with $k \geq 3$, the functions $p_k(n)$ and $g_k(n)$ can be defined similarly. In Section 4, we prove

Theorem 3. *For any $k, n \geq 3$, let $p_k(n)$ denote the smallest integer p with the property that every k-colored set of n points in general position in the plane can be partitioned into p monochromatic subsets whose convex hulls are pairwise disjoint. Then we have*

$$\left\lceil \left(1 - \frac{1}{k}\right) n + \frac{1}{k} \right\rceil \leq p_k(n) \leq \left(1 - \frac{1}{k + 1/6}\right) n + O(1).$$

We also provide an $O(n \log n)$ time algorithm for computing such a partition.

In Section 5, we discuss the analogous problem for $k = 2$ colors, but in higher dimensions. A set of $n \geq d+1$ points in d-space is said to be in *general position*, if no $d + 1$ of its elements lie in a hyperplane.

A sequence $a_1 a_2 \ldots$ of integers between 1 and m is called an (m, d)-*Davenport-Schinzel sequence*, if (i) it has no two consecutive elements which are the same, and (ii) it has no *alternating* subsequence of length $d + 2$, i.e., there are no indices $1 \leq i_1 < i_2 < \ldots < i_{d+2}$ such that

$$a_{i_1} = a_{i_3} = a_{i_5} = \ldots = a, \quad a_{i_2} = a_{i_4} = a_{i_6} = \ldots = b,$$

where $a \neq b$. Let $\lambda_d(m)$ denote the maximum length of an (m, d)-Davenport-Schinzel sequence (see [4], [14]). Obviously, we have $\lambda_1(m) = m$, and it is easy

to see that $\lambda_2(m) = 2m - 1$, for every m. It was shown by Hart and Sharir [9] that $\lambda_3(m) = O(n\alpha(n))$, where $\alpha(n)$ is the extremely slowly growing functional inverse of Ackermann's function, and that this estimate is asymptotically tight. They also proved that $\lambda_d(m)$ is only slightly superlinear in m, for every fixed $d > 3$. (For the best currently known bounds of this type, see [1].)

For any fixed d, let

$$\mu_d(n) = \min\{m : \lambda_d(m) \geq n\}.$$

Thus, we have that

$$\mu_2(n) = \left\lceil \frac{n+1}{2} \right\rceil,$$

and $\mu_d(n)$ is only very slightly sublinear in n, for any $d \geq 3$.

Theorem 4. *For any $n > d \geq 2$, let $p^{(d)}(n)$ denote the smallest integer p with the property that every 2-colored set of n points in general position in d-space can be partitioned into p monochromatic subsets whose convex hulls are pairwise disjoint.*

For a fixed $d \geq 2$, we have

(i) $p^{(d)}(n) \leq \frac{n}{d} + O(1)$;
(ii) $p^{(d)}(n) \geq \mu_d(n)$.

2 Proof of Theorem 1

We prove the lower bound by induction on n. Clearly, we have $p(1) \geq 1$, $p(2) \geq 2$. Assume the inequality holds for all values smaller than n. Consider a set S_a of n points placed on a circle, and having alternating colors, white and black (if n is odd, there will be two adjacent points of the same color, say, white). We call this configuration *alternating*. Denote by $h(n) = p(S_a)$ the minimum number of parts necessary to partition an alternating configuration of n points. Clearly, we have $p(n) \geq h(n)$.

Assume first that n is even, and fix a partition of S_a. We may suppose without loss of generality that this partition has a monochromatic (say, white) part P of size $l \geq 2$, otherwise the number of parts is n. The set $S_a \setminus P$ falls into l contiguous alternating subsets, each consisting of an odd number, $n_1, n_2, \ldots, n_l$, of elements, such that

$$\sum_{i=1}^{i=l} n_i + l = n, \quad \text{or} \quad \sum_{i=1}^{i=l} (n_i + 1) = n.$$

Hence, by induction,

$$h(n) \geq 1 + \sum_{i=1}^{i=l} h(n_i) \geq 1 + \sum_{i=1}^{i=l} \frac{n_i + 1}{2} = 1 + \frac{n}{2} = \left\lceil \frac{n+1}{2} \right\rceil,$$

as required. If n is odd, let P denote one of the two adjacent white points of S_a. Then

$$h(n) = h(S_a) \geq h(S_a - \{P\}) = h(n-1) \geq \frac{n-1}{2} + 1 = \frac{n+1}{2} = \left\lceil \frac{n+1}{2} \right\rceil .$$

To prove the upper bound, we also use induction on n. Clearly, we have $p(1) \leq 1$, $p(2) \leq 2$. Assume the inequality holds for all values smaller than n. If n is even,

$$p(n) \leq p(1) + p(n-1) \leq 1 + \frac{(n-1)+1}{2} = 1 + \frac{n}{2} = \frac{n+2}{2} = \left\lceil \frac{n+1}{2} \right\rceil .$$

Let n be odd. Consider a set S of n points, $n = w + b$, where w and b denote the number of white and black points respectively. Suppose, without loss of generality, that $w \geq b + 1$. If $conv(S)$, the convex hull of S, has two adjacent vertices of the same color, we can take them as a monochromatic part and apply the induction hypothesis to the remaining $n-2$ points. Therefore, we may assume that $conv(S)$ has an even number of vertices, and they are colored white and black, alternately. Let x be a white vertex of $conv(S)$, and consider the points of $S - \{x\}$ in clockwise order of visibility from x. By writing a 0 for each white point that we encounter and a 1 for each black point, we construct a $\{0,1\}$-sequence of length $n-1$. Obviously, this sequence starts and ends with a 1. Removing the first and the last elements, we obtain a sequence $T = a_0 a_1 \dots a_{n-4}$ of (even) length $n-3$.

The sequence T consists of $k = w - 1$ zeroes and $l = b - 2$ ones, where $k - l = (w-1) - (b-2) \geq 2$, $k + l =$ even. According to Claim 1 below, T has a 00 contiguous subsequence (of two adjacent zeroes) starting at an even index $(0, 2, 4, ...)$. Let y and z denote the two white points corresponding to these two consecutive zeroes. Partition the set S into the white triple $\{x, y, z\}$ and two sets of *odd* sizes, n_1 and n_2 (with $n_1 + n_2 = n - 3$), consisting of all points preceding and following $\{y, z\}$ in the clockwise order, respectively. It follows by induction that

$$p(n) \leq 1 + \frac{n_1 + 1}{2} + \frac{n_2 + 1}{2} = \frac{n+1}{2} = \left\lceil \frac{n+1}{2} \right\rceil .$$

It remains to establish

Claim 1. *Let $T = a_0 a_1 \dots a_{k+l-1}$ be a $\{0,1\}$-sequence of even length, consisting of k zeroes and l ones, where $k - l \geq 2$. Then there is an even index $0 \leq i \leq k + l - 2$ such that $a_i = a_{i+1} = 0$.*

Proof. We prove the claim by induction on $k + l$, the length of the string. The base case $k + l = 2$ is clear, because then we have $T = 00$. For the induction step, distinguish four cases, according to what the first two elements of T are: $00, 01, 10$, or 11. In the first case we are done. In the other three cases, the assertion is true, because the inequality $k' - l' \geq 2$ holds for the sequence T' obtained from T by deleting its first two elements. (Here k' and l' denote the number of zeroes and ones in T', respectively.) □

3 An Algorithmic Proof of Theorem 2

The functions $p(\cdot)$ and $p(\cdot,\cdot)$ are monotone increasing: if $i \le j$ then $p(i) \le p(j)$, and if $i \le k$ and $j \le l$ then $p(i,j) \le p(k,l)$. Thus, we have the lower bound in (1):

$$p(w,b) \ge p(w,w) \ge h(2w) \ge \left\lceil \frac{2w+1}{2} \right\rceil = w+1.$$

Next we prove the upper bound in (1). For $w = 1$, the inequality holds, so assume $w \ge 2$. Take an edge xy of the convex hull of the w white points, and let l denote its supporting line. Take $\{x, y\}$ as a monochromatic (white) part of size 2, and another (black, possibly empty) monochromatic part by selecting all black points lying in the open half-plane bounded by l, which contains no white points. Continue this procedure in the other open half-plane bounded by l as long as it contains at least two white points. If there is one (resp., no) white point left, then the remaining points can be partitioned into at most three (resp., one) monochromatic parts, whose convex hulls are pairwise disjoint. It is easy to verify that the number of parts in the resulting partition does not exceed $2\left\lceil \frac{w}{2} \right\rceil + 1$. It is also true that the white parts of this partition form a perfect matching (of the white points) if w is even, and an *almost perfect* one (with one isolated point) if w is odd.

At the end of this section we present an $O(n \log n)$ time algorithm which computes such a partition.

Further, we verify the three cases highlighted in the theorem. For even w, the lower and upper bounds in (1) are both equal to $w+1$. The same holds in case (i), i.e., when $w \le b \le w+1$: $p(w,b) \le p(2w+1) = w+1$. For odd w, the gap between the lower and upper bounds is at most one.

It remains to discuss case (ii), when w is odd and $b \ge 2w$, for which we get a tight bound. By the monotonicity of $p(w,b)$, it is enough to show that $p(w,2w) \ge w+2$. Let Y be a regular w-gon. Let X and Z denote the *inner* and *outer* polygons obtained from Y by slightly shrinking it and blowing it up, respectively, around its center O. To bring $X \cup Y \cup Z$ into general position, slightly rotate Y around its center in the clockwise direction so that, if $x \in X$ and $z \in Z$ are the two vertices corresponding to $y \in Y$, then the points O, x, y, z are almost collinear. Place a white point at each vertex of Y, and a black point at each vertex of X and Z. (See Fig. 1.)

We say that a *triangle* is *white* (resp. *black*), if all of its vertices are white (resp. black). Notice that the white vertices must be partitioned into at least $(w+1)/2$ parts. Otherwise, there would be a white part of size (at least) 3, which is impossible, because every white triangle Δ contains a black point of X in its interior. (If the center O is inside Δ, there are three black points of X inside Δ. If O is outside Δ, the black point of X corresponding to the "middle" point of Δ lies in Δ.) Similarly, the vertices of Z require at least $(w+1)/2$ parts.

Assume, for contradiction, that S can be partitioned into at most $w+1$ parts. Then the monochromatic parts restricted to the vertices of Z form an almost perfect non-crossing matching using straight-line segments, with exactly

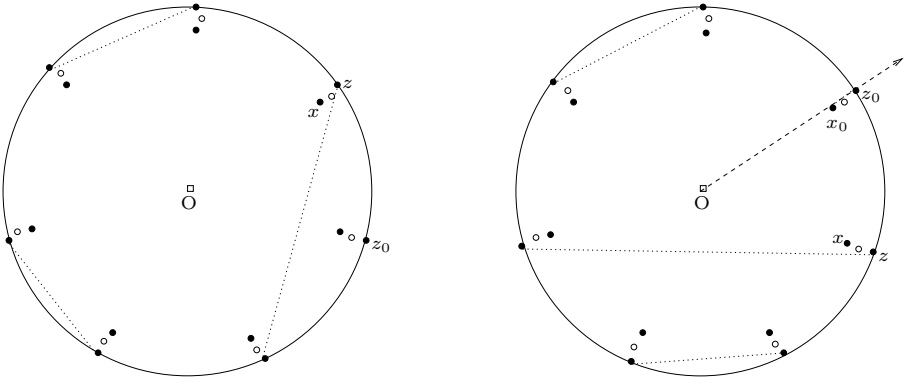


Fig. 1. A point set proving the lower bound in Theorem 2

one unmatched point, and the same holds for Y. Let $z_0 \in Z$ be the unmatched point of Z. We restrict our attention to the matching M formed by the vertices of Z. These segments partition the circumscribing circle C of Z into $(w+1)/2$ convex regions. The *boundary* $B(R)$ of a region R, is formed by some segments in M and arcs of C. Let Z_0 denote the region containing the center O. Each point of X must be assigned to a black part determined by M. We distinguish two cases.

Case 1: $z_0 \notin Z_0$ (see Figure 1, left). Pick any segment $s \in M \cap B(Z_0)$, and let z denote its first vertex in the clockwise order. Let x denote the vertex of X belonging to the ray Oz. It is easy to see that x cannot be assigned to any of the black parts determined by M, since the black triangle obtained in this way would contain a white point in its interior, contradiction.

Case 2: $z_0 \in Z_0$ (see Figure 1, right). Let x_0 be the vertex of X belonging to the ray Oz_0. The point x_0 can only be assigned to the part containing z_0. By adding the segment x_0z_0 to M, we obtain a non-crossing matching M'. Let z be the first vertex on $B(Z_0)$ following z_0 in the clockwise order. Let x denote the vertex of X belonging to the ray Oz. Similarly to Case 1, it is easy to see that x cannot be assigned to any of the black parts determined by M', since any black triangle obtained in this way would contain a white point, contradiction.

Thus, any partition of S requires at least $w+2$ parts, which completes the proof of the theorem.

Algorithm outline. We follow the general scheme described in the proof of the upper bound in Theorem 2, but allow some variations. Let W and B denote the set of white and black points of S, respectively.

1. This step ignores the black points. First, compute the convex layer decomposition of W. Based on this decomposition, find a perfect (or almost perfect) non-crossing matching M, using straight-line segments of W. At the same time, construct a planar subdivision D of the plane into at most $\lceil \frac{w}{2} \rceil + 1$ convex regions

by extending the segments of M along their supporting lines. (If w is odd, draw an arbitrary line through the point which is left unmatched by M, and consider this point a part of M.) See Claim 2 below for a more precise description of the partitioning procedure using the extension of segments.
2. Here the black points come into play. Preprocess D for point location and perform point location in D for each black point.
3. Output the partition consisting of the pairs of points in M (the unmatched point as a part, if w is odd) for the white points, and the groups of points in the same region for the black points.

The following statement is an easy consequence of Euler's polyhedral formula (see also [13], page 259).

Claim 2. *Let $S = \{s_1, s_2, \ldots, s_m\}$ be a family of m pairwise disjoint segments in the plane, whose endpoints are in general position. In the given order, extend each segment in both directions until it hits another segment or the extension of a segment, or to infinity. After completing this process, the plane will be partitioned into $m + 1$ convex regions.*

Algorithm details and analysis. The algorithm outputs at most $2\lceil \frac{w}{2} \rceil + 1$ monochromatic parts, $\lceil \frac{w}{2} \rceil$ of which are white and at most $\lceil \frac{w}{2} \rceil + 1$ are black. The decomposition into convex layers takes $O(n \log n)$ time [3]. The matching M is constructed starting with the outer layer and continues with the next layer, etc. If, for example, the size of the outer layer is even, every other segment in the layer is included in M. If the size of the outer layer is odd, we proceed in the same way, but at the end we compute the tangent from the last unmatched point to the convex polygon of the next layer, include this segment in M, and proceed to the next layer. The algorithm maintains at each step a current convex (possibly unbounded) polygonal region P containing all unmatched points. It also maintains the current planar subdivision D computed up to this step. The next segment s to be included in M lies inside P. Let s be oriented so that all unmatched (white) points are to the right of it. After extending s, P will fall into two convex polygonal regions, P' and P'', containing all the remaining unmatched points, and none of the points, respectively. At this point, P' becomes the new P and P'' is included in the polygonal subdivision D, which is thus refined. An example of step 1 is shown in Figure 2, with the segments numbered in their order of inclusion in M.

For each convex layer, we compute a balanced hierarchical representation (see Lemma 1 below). Since a tangent to a convex polygon of size at most n from an exterior point can be computed in $O(\log n)$ time, having such a representation, the time to compute all tangents is $O(n \log n)$. By dynamically maintaining the same (balanced hierarchical) representation for the current polygonal region P, we can compute the (at most two) intersection points between P and the supporting line of s in $O(\log n)$ time. The reader is referred to [11], pages 84–88 for this representation and the for two statements below.

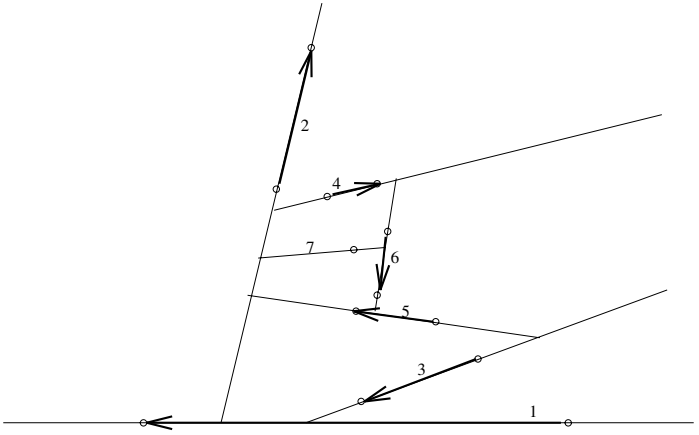


Fig. 2. Illustration of Step 1 of the algorithm

Lemma 1. ([11], page 85) *A balanced hierarchical representation of a convex polygon on n points can be computed in $O(n)$ time.*

Lemma 2. ([11], page 87) *Given a balanced hierarchical representation of a convex polygon on n points and a line l, $P \cap l$ can be computed in $O(\log n)$ time.*

One can obtain a balanced hierarchical representation of a convex polygon in a natural way, by storing the sequence of edges in the leaves of a balanced tree. Furthermore, selecting (2-3)-trees for this representation, allows us to get the balanced hierarchical representation of $P^{'}$ from that of P in $O(\log n)$ time by using the SPLIT operation on concatenable queues (see [2], pages 155–157). To get $P^{'}$ from P calls for at most two INSERT and at most two SPLIT operations, each taking $O(\log n)$ time. The update of D with $P^{''}$ takes $O(|P^{''}|)$ time. Since $\sum |P^{''}|$ over the execution of the algorithm is $O(n)$, the total time required by step 1 is $O(n \log n)$.

We recall the following well known fact (e.g. see [12], page 77).

Lemma 3. *For any planar graph with n vertices, one can build a point location data structure of $O(n)$ size in $O(n \log n)$ time, guaranteeing $O(\log n)$ query time.*

The point location is performed for $b \leq n$ points, so the total time of step 2 is also $O(n \log n)$. The complexity of the last step is linear, thus the total time complexity of the algorithm is $O(n \log n)$. The space requirement is $O(n)$.

Another approach. The next algorithm closely mimics the proof of the upper bound in Theorem 2. Using the method of [10] (or of [3]) for convex hull maintenance under deletions of points, a sequence of n deletions performed on an n-point set takes $O(n \log n)$ time. Specifically, maintain the convex hull of the white points and that of the black points. Take an edge e of the white hull. Repeatedly, query the black hull to find a black point in the halfplane determined

by e containing none of the white points. Such points (if any) are deleted one by one, until no other is found, and their collection is output as a black part. Then the two endpoints of e are deleted and this pair is output as a white part. The above step is repeated until the white points are exausted and the partition is complete. The $O(n)$ queries and deletions take $O(n \log n)$ time.

4 Coloring with More Colors – Proof of Theorem 3

We prove the lower bound by induction on n. Clearly, we have $p_k(1) \geq 1,\ p_k(2) \geq 2, \ldots,\ p_k(k) \geq k$. Assume the inequality holds for all values smaller than n. Consider a set S of n points placed on a circle, colored with k colors, $1, 2, \ldots, k$, in a periodic fashion:

$$1, 2, \ldots, k, 1, 2, \ldots, k, \ldots, 1, 2, \ldots, r,$$

where $n = mk + r,\ 0 \leq r \leq k-1$. We call such a configuration *periodic*. Let $h_k(n) := p_k(S)$, the smallest number of monochromatic parts a periodic configuration S can be partitioned into so that their convex hulls are pairwise disjoint. Clearly, we have $p_k(n) \geq h_k(n)$. Fix such a partition.

Case 1: $r = 0$. If every monochromatic part of the partition is a singleton, then we are done. Consider a monochromatic part P of size $l \geq 2$, and assume without loss of generality that the color of P is k. $S - P$ splits into smaller periodic subsets of sizes $n_1, n_2, \ldots, n_l$, with $n_i \equiv -1 \pmod{k}$ for every i, such that

$$\sum_{i=1}^{i=l} n_i + l = n, \quad \text{or} \quad \sum_{i=1}^{i=l} (n_i + 1) = n. \tag{2}$$

Hence, by induction,

$$h_k(n) \geq 1 + \sum_{i=1}^{i=l} h_k(n_i) \geq 1 + \sum_{i=1}^{i=l} \left\lceil \frac{(k-1)n_i + 1}{k} \right\rceil = 1 + \sum_{i=1}^{i=l} \frac{(k-1)n_i + (k-1)}{k}$$

$$= 1 + \sum_{i=1}^{i=l} \frac{(k-1)(n_i+1)}{k} = 1 + \frac{(k-1)n}{k} = \frac{(k-1)n + k}{k} = \left\lceil \frac{(k-1)n + 1}{k} \right\rceil.$$

Case 2: $r = 1$. Discard 1 point of color 1 from the last (incomplete) group, and get a periodic configuration with $n-1$ points, as in the previous case. Then, by induction,

$$h_k(n) \geq h_k(n-1) \geq \left\lceil \frac{(k-1)(n-1)+1}{k} \right\rceil$$

$$= \left\lceil \frac{(k-1)(n-1)+k}{k} \right\rceil = \left\lceil \frac{(k-1)n+1}{k} \right\rceil.$$

Case 3: $r \geq 2$. We distinguish two subcases:

Subcase 3.a: There exists a monochromatic part P of size $l \geq 2$ of color 1. The set $S - P$ splits into smaller periodic subsets of sizes $n_1, n_2, \ldots, n_l$ satisfying

(2), where $n_i \equiv -1 \pmod{k}$ for all $1 \le i \le l-1$, and $n_l \equiv r-1 \pmod{k}$. Then, using the induction hypothesis, we obtain

$$h_k(n) \ge 1 + \sum_{i=1}^{i=l} h_k(n_i) \ge 1 + \sum_{i=1}^{i=l} \left\lceil \frac{(k-1)n_i + 1}{k} \right\rceil$$

$$= 1 + \sum_{i=1}^{i=l-1} \frac{(k-1)n_i + (k-1)}{k} + \frac{(k-1)n_l + (r-1)}{k}$$

$$= 1 + \sum_{i=1}^{i=l} \frac{(k-1)(n_i+1)}{k} + \frac{r-k}{k} = \frac{(k-1)n + r}{k} = \left\lceil \frac{(k-1)n + r}{k} \right\rceil$$

$$= \left\lceil \frac{(k-1)n + r - (k-1)}{k} \right\rceil = \left\lceil \frac{(k-1)n + 1}{k} \right\rceil.$$

Subcase 3.b: There is no monochromatic part P of size $l \ge 2$ of color 1.

$$h_k(n) \ge \frac{n-r}{k} + h_{k-1}(n - \frac{n-r}{k}) \ge \frac{n-r}{k} + \left\lceil \frac{(k-2)(n - \frac{n-r}{k}) + 1}{k-1} \right\rceil$$

$$= m + \left\lceil \frac{(k-2)((k-1)m + r) + 1}{k-1} \right\rceil = (k-1)m + \left\lceil \frac{(k-2)r + 1}{k-1} \right\rceil.$$

The lower bound we want to prove can be written as

$$\frac{(k-1)n + 1}{k} = (k-1)m + \left\lceil \frac{(k-1)r + 1}{k} \right\rceil.$$

An easy calculation shows that

$$\left\lceil \frac{(k-2)r + 1}{k-1} \right\rceil = r + \left\lceil \frac{1-r}{k-1} \right\rceil = r + \left\lceil \frac{1-r}{k} \right\rceil = \left\lceil \frac{(k-1)r + 1}{k} \right\rceil.$$

The value of the above expression is 1 for $r = 0$, and r for $r \ge 1$ (only the latter case is of interest in this subcase). This concludes the proof of the lower bound.

Next, we establish the upper bound in Theorem 3.

Proposition 1. $p_k(n) \le n - g_k(n)$.

Proof. A set of n points contains a non-crossing matching of $g_k(n)$ pairs. Thus, using parts of size 2 for the points in the matching and singletons for the remaining ones, we obtain a partition, in which the number of parts is

$$g_k(n) + (n - 2g_k(n)) = n - g_k(n).$$

□

Proposition 2. [7] *For $k \geq 3$, we have $g_k(6k+1) = 6$.*

It follows immediately from Propositions 1 and 2 that

$$p_k(n) \leq n - g_k(n) = \left(1 - \frac{6}{6k+1}\right) n + O(1) = \left(1 - \frac{1}{k+1/6}\right) n + O(1),$$

as required.

In the end, we outline an algorithm. Its main part is the computation of the matching M, which appears in [7]. Let $k \geq 3$ be a fixed integer.

Algorithm. The n points are sorted according to their x-coordinate and divided into consecutive groups of size $6k+1$. In each group, with the possible exception of the last one, one can match at least 12 elements. This can be done by matching 12 out of 19 points in the three largest color classes. The time to process a group is $O(k)$, so the total time is $O(n \log n)$. The number of output parts is bounded as in the theorem.

5 Higher Dimensions – Proof of Theorem 4

The upper bound (i) can be shown by straightforward generalization of the proof of Theorem 2: if $w \leq b$, $n = w + b$, $|W| = w$, one can iteratively remove a facet of $conv(W)$. The easy details are left to the reader.

To verify part (ii), we need a simple property of the d-dimensional *moment curve*

$$M_d(\tau) = (\tau, \tau^2, \ldots, \tau^d), \ \tau \in \mathbb{R}.$$

For two points, $x = M_d(\tau_1)$ and $y = M_d(\tau_2)$, we say that x *precedes* y and write $x \prec y$ if $\tau_1 < \tau_2$. For simplicity, let $u = \lceil d/2 \rceil$ and $v = \lfloor d/2 \rfloor$.

Lemma 4. (see [5], [6]) *Let $x_1 \prec x_2 \prec \ldots \prec x_{u+1}$ and $y_1 \prec y_2 \prec \ldots \prec y_{v+1}$ be two sequences of distinct points on the d-dimensional moment curve, whose convex hulls are denoted by X and Y, respectively.*

Then X and Y cross each other if and only if the points x_i and y_j interleave, i.e., every arc $x_i x_{i+1}$ of $M_d(\tau)$ contains exactly one y_j.

Let S_a be a sequence of n points on the d-dimensional moment curve, $x_1 \prec x_2 \prec \ldots \prec x_n$, which are colored white and black, alternately. Consider a partition of S_a into m monochromatic parts, labeled by integers between 1 and m, and suppose that the convex hulls of these parts are pairwise disjoint. Replacing every element of the sequence S_a by the the label of the part containing it, we obtain a sequence T of length n, whose elements are integers between 1 and m. Obviously, no two consecutive elements of this sequence coincide, because adjacent points have different colors, and thus belong to different parts in the partition. It follows from the above lemma that T has no alternating subsequence of length $(u+1)+(v+1) = d+2$. That is, T is an (m, d)-Davenport-Schinzel sequence of length n. Therefore, we have $n \leq \lambda_d(m)$, or, equivalently, $m \geq \mu_d(n)$, as required.

Acknowledgments. The authors would like to thank the anonymous referee for suggesting the alternative algorithmic approach at the end of Section 3.

References

1. P. K. Agarwal, M. Sharir, and P. Shor, Sharp upper and lower bounds for the length of general Davenport-Schinzel sequences, *Journal of Combinatorial Theory, Ser. A*, **52** (1989), 228–274.
2. V. Aho, J. Hopcroft and J. Ullman, *The Design and analysis of Computer Algorithms*, Addison-Wesley, Reading, 1974.
3. B. Chazelle, On the convex layers of a planar set, *IEEE Transactions on Information Theory*, **31(4)** (1985), 509–517.
4. H. Davenport and A. Schinzel, A combinatorial problem connected with differential equations, *American Journal of Mathematics*, **87** (1965), 684–694.
5. T. K. Dey and H. Edelsbrunner, Counting triangle crossings and halving planes, *Discrete & Computational Geometry*, **12** (1994), 281–289.
6. T. K. Dey and J. Pach, Extremal problems for geometric hypergraphs, *Discrete & Computational Geometry*, **19** (1998), 473–484.
7. A. Dumitrescu and R. Kaye, Matching colored points in the plane: some new results, *Computational Geometry: Theory and Applications*, to appear.
8. A. Dumitrescu and W. Steiger, On a matching problem in the plane, *Workshop on Algorithms and Data Structures*, 1999 (WADS '99). Also in: *Discrete Mathematics*, **211** (2000), 183–195.
9. S. Hart and M. Sharir, Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes, *Combinatorica*, **6** (1986), 151–177.
10. J. Hershberger and S. Suri, Applications of a semi-dynamic convex hull algorithm, *BIT*, **32** (1992), 249–267.
11. K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional searching and Computational Geometry*, Springer Verlag, Berlin, 1984.
12. K. Mulmuley, *Computational Geometry – An Introduction through Randomized Algorithms*, Prentice Hall, Englewood Cliffs, 1994.
13. J. O'Rourke, *Art Gallery Theorems and Algorithms*, Oxford University Press, New York, 1987.
14. M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*, Cambridge University Press, Cambridge, 1995.

The Analysis of a Probabilistic Approach to Nearest Neighbor Searching*

Songrit Maneewongvatana and David M. Mount

Department of Computer Science
University of Maryland
College Park, Maryland
{songrit,mount}@cs.umd.edu

1 Introduction

Given a set S of n data points in some metric space. Given a query point q in this space, a *nearest neighbor query* asks for the nearest point of S to q. Throughout we will assume that the space is real d-dimensional space $\Re^d$, and the metric is Euclidean distance. The goal is to preprocess S into a data structure so that such queries can be answered efficiently. Nearest neighbor searching has applications in many areas, including data mining [7], pattern classification [5], data compression [10].

Because many applications involve large data sets, we are interested in data structures that use linear storage space. Naively, nearest neighbor queries can be answered in $O(dn)$ time through brute-force search. Although nearest neighbor searching can be performed efficiently in low-dimension spaces, for all known exact linear-space data structures, search times grow exponentially as a function of dimension. Thus for reasonably large dimensions, brute-force search is often the most efficient in practice. One approach to reducing the search time is through *approximate nearest neighbor search*. A number of data structures for approximate nearest neighbor searching have been proposed [1,3,11]. The phenomenon of concentration of distance would suggest that approximate nearest neighbor searching is meaningless. Fortunately, the distributions that arise in applications tend to be clustered in lower dimensional subspaces [6]. Good search algorithms take advantage of this low-dimensional clustering.

The fundamental problem that motivates this work is the lack of *predictability* in existing practical approaches to nearest neighbor searching. In high dimensions, exact search is no better than brute-force, and approximate search algorithms are acceptably fast only when the allowed approximation factors are set to unreasonably high values [1]. The goal of this work is to address this shortcoming by providing a practical data structure for nearest neighbor searching that is both *efficient* and guarantees *predictable performance*.

We measure performance in an aggregate sense, rather than for individual queries. We assume that queries are drawn from some known probability distribution. The user provides a desired *failure probability*, p_f, and the resulting

* The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 276–286, 2001.

search fails to return the true nearest neighbor with probability at most p_f. The query distribution is presented by a set of *training queries* as part of the preprocessing stage. The training data permits us to tailor the data structure to the underlying structure of the point distribution.

The idea of allowing failures in nearest neighbor searching was proposed by Ciaccia and Patella [2], but no performance guarantees were provided. We present a data structure called an *overlapped split tree*, or *os-tree* for short. The tree, which we first introduced in [12], is a generalization of the well known BSP-tree, in which each node of the tree is associated with a convex region of space. However, unlike the BSP-tree in which the child regions partition the parent's region, here we allow these regions to overlap one another. The degree of overlap is determined by the failure probability and the query distribution as represented by the training points.

Based on empirical experiments on both synthetic and real data sets, we have shown that compared to the popular kd-tree data structure, it is possible to achieve significantly better predictability of failure probabilities while achieving running times that are competitive, and sometimes superior to the kd-tree [12]. However that paper did not address the efficiency of the data structure except for experiments on synthetic data sets. In this paper we present a theoretical analysis of the os-tree's efficiency and performance and provide experimental evidence of its efficiency on real application data sets.

2 Overlapped Split Tree

The os-tree was introduced by the authors in an earlier paper [12]. We summarize the main elements of the data structure here. It is a generalization of the well-known *binary space partition (BSP) tree* (see, e.g., [4]). Consider a set S of points in d-dimensional space. A BSP-tree for this set of points is based on a hierarchical subdivision of space into convex polyhedral *cells*. Each node in the BSP-tree is associated with such a cell and the subset of points lying within this cell. The root node is associated with the entire space and the entire point set. A cell is split into two disjoint cells by a *separating hyperplane*, and these two cells are then associated with the children of the current node. Data points are then distributed among the children according to which side of the separating hyperplane they lie. The process is repeated until the number of points associated with a node falls below a given threshold, which we assume to be one throughout.

The os-tree generalizes the BSP-tree by associating each node with an additional convex polyhedral region called a *cover*. (In [12] the term *approximate cover* was used.) Consider a node δ in the tree, associated with the point subset S_δ. The cover C_δ is constructed to contain a significant fraction of the points of $\Re^d$ whose nearest neighbor is in S_δ, that is, C_δ covers a significant fraction of the union of the Voronoi cells [4] of the points of S_δ. Computing these Voronoi cells would be too expensive in high dimensional spaces, and so the covers are computed with the aid of a large set of *training points* T, which is assumed to be sampled from the same distribution at the data points, and where $|T| \gg |S|$.

For each point in T we compute its nearest neighbor in S. The cover C_δ contains all the points of T whose nearest neighbor is in S_δ. See Fig. 1. As the size of T increases, the probability that a point lies outside of C_δ but has its nearest neighbor in S_δ decreases.

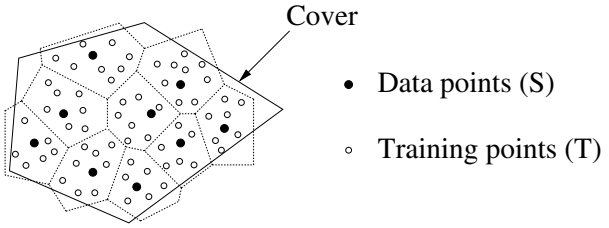


Fig. 1. The cover for a set of data points (filled circles) contains all the training points (shown as white circles) whose nearest neighbor is in this subset.

As with the BSP tree, covers are not stored explicitly, but rather are defined by a set of *boundary hyperplanes* stored at the nodes of each of the ancestors. In typical BSP fashion, the parent cell is split by a hyperplane, which partitions the point set and cell. We introduce two halfspaces, supported by parallel hyperplanes. One covers the Voronoi cells associated with the left side of the split and the other covers the Voronoi cells associated with the right side. These hyperplanes are stored with the parent cell. Thus, as we descend the tree, the intersection of the associated halfspaces implicitly defines the cover.

The construction algorithm works recursively, starting with the root. The initial point set consists of all the data points, and the initial cell and cover are $\Re^d$. Consider a node δ containing two or more data points. First, we compute a separating hyperplane H for the current point set (see Fig. 2). This hyperplane is chosen to be orthogonal to the largest eigenvector of the covariance matrix of the points of S_δ. This is the direction of maximum variance for the points of S_δ [9]. The position of the hyperplane is selected so that it bisects the points of S_δ. Let S_l and S_r denote the resulting partition, and define T_l and T_r analogously. (This partition is indicated using circles and squares in Fig. 2.)

To determine the orientation of the boundary hyperplanes, we invoke support-vector machines (SVM) to the subsets S_l and S_r. SVM was developed in learning theory [13] to find the hyperplane that best separates two classes of points in multidimensional space. SVM finds a splitting hyperplane with the highest margin, which is defined to be the sum of the nearest distance from the hyperplane to the points in S_l and the nearest distance to the points in S_r. After SVM determines orientation of the splitting hyperplane, the two boundary hyperplanes H_l and H_r are chosen to be parallel to this hyperplane, such that H_l bounds all the points of T_l and H_r bounds all the points of T_r. The hyperplanes H_l and H_r are stored in node δ, and the process continues recursively on each of the two subsets.

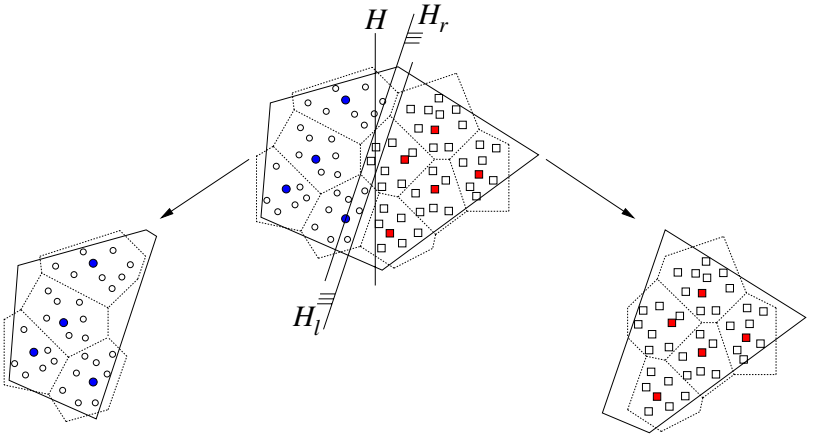


Fig. 2. The construction of the boundary hyperplanes (H_l and H_r) and the resulting partition of the data and training points.

To answer the nearest neighbor query q, the search starts at the root. For any leaf, the distance to the point in this node is returned. At any internal node δ, we first determine which cover(s) the query lies. This is done in $O(d)$ time by comparing q against each of the boundary hyperplanes. If q lies in only one cover, then we recursively search the corresponding subtree and return the result. Otherwise, the subtree closest to q (say the left) is searched first, and the distance d_l to the nearest neighbor in this subtree is returned. If the distance from q to the other (right) boundary hyperplane is less than d_l, then this subtree is also searched resulting in a distance d_r. The minimum of d_l and d_r is returned.

It is easy to see that this search will fail only if for some node δ, the query point q lies in the Voronoi cell of some point $p \in S_\delta$, but lies outside the associated cover. It is not hard to show that if T and S are independent and identically distributed, then the probability of such a failure is proportional to $|S|/|T|$. (The proof is omitted from this version.) By making the training set sufficiently large, we can reduce the failure probability below any desired threshold.

3 Theoretical Analysis

In this section we explore the expected search time of the os-tree. As the search visits each nonleaf node of the os-tree, there are three possibilities: visit the left child only, visit the right child only, or visit both children. Suppose that the probability that we visit both children at any given node is bounded above by b, $0 < b \leq 1$. Let $T(n)$ denote the expected running time of the search algorithm given a subtree with n data points. Then because we do $O(d)$ computations at each node and split the data points evenly at each step, it follows that up to constant factors, $T(n)$ is bounded by the following recurrence

$$T(n) \;\leq\; 2bT(n/2) + (1-b)T(n/2) + d \;=\; (1+b)T(n/2) + d.$$

Solving the recurrence yields

$$T(n) \in O\left(dn^{\lg(1+b)}\right).$$

Thus the analysis of the expected search reduces to bounding the value of b.

In general b is a function of the dimension d and the point distribution. It is not difficult to contrive distributions in which, on any given node, virtually all query points visit both subtrees. In such cases the above analysis suggests that the running time is no better than brute-force search. Our analysis is based on some assumptions on the data distribution, which we believe are reasonable models of typical real data sets.

An important aspect of real data sets in high dimensional spaces is that many of the coordinates are correlated and some exhibit very small variances. Hence, points tend to be clustered (at least locally) around low dimensional spaces. A common way of modeling this is through *principal component analysis*. Consider a fixed node δ in the tree and the subset S_δ of data points associated with this node, and let n_δ be the cardinality of this set. Let C_δ denote the corresponding cover. Because δ will be fixed for the rest of the analysis, we will drop these subscripts to simplify the notation. We assume that the data points are sampled from a d-dimensional multivariate distribution. Let $\boldsymbol{x} = \{x_i\}_{i=1}^d$ denote a random vector from this distribution. Let $\boldsymbol{\mu} \in \Re^d$ denote the mean of this distribution and let $\boldsymbol{\Sigma}$ denote the $d \times d$ *covariance matrix* for the distribution [9],

$$\boldsymbol{\Sigma} = E((\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T).$$

This is a symmetric positive definite matrix, and hence has d positive eigenvalues. We can express this as $\boldsymbol{\Sigma} = \boldsymbol{U}\boldsymbol{\Lambda}\boldsymbol{U}^T$, where $\boldsymbol{U}$ is a $d \times d$ orthogonal matrix whose columns $\{\boldsymbol{u}_i\}_{i=1}^d$ are the eigenvectors of $\boldsymbol{\Sigma}$ and $\boldsymbol{\Lambda}$ is a $d \times d$ diagonal matrix, whose entries $\{\lambda_i\}_{i=1}^d$ are the eigenvalues of $\boldsymbol{\Sigma}$. We may assume that the eigenvalues are sorted in decreasing order, so that λ_1 is the largest eigenvalue. By applying the transformation $\boldsymbol{y} = \boldsymbol{U}^T(\boldsymbol{x} - \boldsymbol{\mu})$, we map the points to a reference system in which the samples have mean $\boldsymbol{0}$, and a diagonal covariance matrix with the prescribed eigenvalues. The coordinates are now uncorrelated. We may assume henceforth that all point coordinates are given relative to this new reference system, the λ_i is the distribution variance for the ith coordinate. Henceforth, let $\sigma_i = \sqrt{\lambda_i}$.

Given a parameter $\gamma > 0$ define the *pseudo-dimension* $\hat{d}$, to be minimum integer such that $1 \le \hat{d} \le d$ and

$$\sum_{\hat{d} < i \le d} \frac{\sigma_i^2}{\sigma_1^2} \le \gamma^2.$$

Under the assumption that the points are clustered in a low-dimension subspace, we would expect that $\hat{d}$ is small relative d. Let $\widehat{H}$ be the $\hat{d}$-dimensional hyperplane spanned by the first $\hat{d}$ eigenvectors of $\boldsymbol{\Sigma}$. Given a query point $\boldsymbol{q} = (q_i)_{i=1}^d$

sampled from the same distribution as the data points, let $\hat{\boldsymbol{q}}$ denote its orthogonal projection onto $\widehat{H}$, and let $\epsilon(\boldsymbol{q}) = ||\boldsymbol{q} - \hat{\boldsymbol{q}}||$, where $||\cdot||$ denotes Euclidean distance. It is well known [9] that the expected value of $\epsilon^2(\boldsymbol{q})$ is

$$E(\epsilon^2(\boldsymbol{q})) = \sum_{\hat{d}<i\leq d} \sigma_i^2 \leq (\gamma\sigma_1)^2.$$

Our analysis is based on the following *distribution assumptions*. The first is that the pseudo-dimension $\hat{d}$ is significantly smaller that the dimension d of the space. The second is a type of Lipschitz condition on the distribution function, which states that point densities are bounded relative to the principal components. In particular, consider the restriction of the point distribution to the first $\hat{d}$ coordinates. We assume that there exist positive constants c_1 and c_2 such that given any point $\hat{\boldsymbol{q}} \in C \cap \widehat{H}$ and for all sufficiently small positive r, (1) the probability of a point lying within a ball of radius r centered at $\hat{\boldsymbol{q}}$ is at least $(c_1 r/\sigma_1)^{\hat{d}}$, and (2) for all positive x, the probability that $|q_1| \leq x$ is at most $c_2 x/\sigma_1$. Note that these conditions are satisfied for a uniform distribution assuming that the first $\hat{d}$ eigenvalues are bounded away from 0.

Recall that in the construction of the os-tree, we first partition the points into equal sizes using a separating hyperplane that is orthogonal to the largest eigenvector of the sample covariance matrix. We assume that n is large enough that the differences in the sample covariance matrix and the distribution covariance matrix are negligible. In the os-tree construction, we use SVM to compute the best orientation for the boundary hyperplanes. To simplify the analysis, let us assume that the boundary hyperplanes are chosen to be orthogonal to the largest eigenvector. Due to space limitations the proof has been omitted. It will be presented in the full version of the paper.

Theorem 1. *For any b, $0 < b \leq 1$ and for pseudo-dimension $\hat{d}$ defined by*

$$\gamma \leq \frac{b^{1.5}}{c_2 4\sqrt{3}}.$$

there is a value n_b (depending on b, c_1, c_2, and $\hat{d}$), such that for any node δ in the os-tree associated with at least n_b points and satisfying distribution assumptions stated earlier, the probability that either (1) a random query point $\boldsymbol{q}$ visits both children in the os-tree search or (2) the search returns an incorrect result from this node is at most b.

4 Experimental Results

We used both synthetic and real data sets. Because the os-trees require a relatively large number of training points, one advantage of synthetic data sets is that we can generate training sets of arbitrary size. To emulate real data sets, we choose a distribution that is clustered in subspaces having low intrinsic dimensionality, which we call *clustered rotated flats*. In this distribution, points

are distributed among clusters, whose centers are sampled uniformly from a hypercube $[-1,1]^d$. Each cluster contains a low dimensional flat. We denote *fat dimension* for each dimension on the flat, and *thin dimension* for others. The fat dimensions are randomly chosen among the coordinates of the full space. Points are distributed evenly to each cluster. In fat dimensions, point are drawn from uniform distribution in the range of $[-1,1]$. In thin dimensions, we use a Gaussian distribution with a standard deviation of σ_{thin}. Each flat is then rotated independently. This is done by repeatedly applying the rotation matrix A to all points in the cluster. A is an identity matrix with four elements $A_{ii} = A_{jj} = \cos(\theta), A_{ij} = -A_{ji} = \sin(\theta)$, where i and j are randomly chosen axes and θ is randomly chosen from $-\pi/2$ to $\pi/2$. In the experiments, the number of clusters is fixed at 5, the number of fat dimensions is $\lfloor 3d/10 \rfloor$.

Two real data sets are used in the experiments. The first set is *NASA MODIS satellite images*. MODIS is a sensor aboard a satellite to acquire spatial data in 36 spectral bands of which 26 were usable. The data from the sensor are archived into files according to region of the earth and particular time. Both data and query sets contained around 13K points. The second set is *NOAA World Ocean Atlas* This data set contains information about some basic attributes of the oceans of the world. Examples of attributes are temperature, salinity, and so on. We use 8 attributes in the data set. The data type of all attributes is floating point. Both data and query set contained around 11K points.

4.1 Eigenvalues of Point Sets

In our analysis, we assumed that if we sort the eigenvalues of the data set associated with each node in decreasing order, these values decrease rapidly. This assumption is generally valid for most of the synthetic data sets. It is also true in both real data sets we use. We recorded all eigenvalues of the point set in each node in os-trees during tree construction. These eigenvalues are sorted and normalized. We computed these relative eigenvalues averaged over the nodes at each level of the os-tree.

In Fig. 3 we show the first, second, third, fifth eigenvalues and every five eigenvalues after that. Note that the plot is log-scale along the y-axis. Level 0 represents the root node, level 1 represents the children of the root node and so on. Only the first five eigenvalues are consistently greater than 0.1 in several levels in the tree. The rest decrease rapidly. Some of the lower eigenvalues are left unplotted at near leaf levels because they are zero. This is because there are not enough points in such nodes to span all the dimensions. The results of the NOAA ocean data set are quite similar, relatively few (three) eigenvalues out of 8 eigenvalues are greater than 0.1.

The results of a synthetic data set are shown in Fig. 4. We generated a data set with 32K points in 30 dimensional space and set $\sigma_{\text{thin}} = 0.05$. The number of rotations is 30. The clusters in this data set are 9-dimensional flats. The results show that there is a significant gap between 9th and 10th eigenvalues. The first nine eigenvalues capture the major characteristics of the data set. Observe that in the first few levels of the tree each node spans multiple clusters, and hence

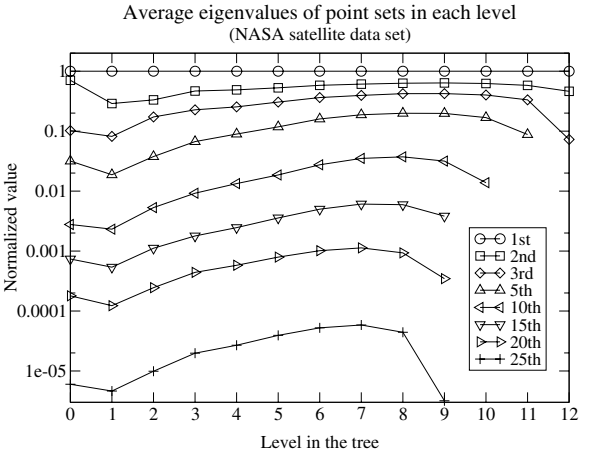


Fig. 3. Eigenvalues of point sets in each level in the tree. NASA satellite data set.

the rapid reduction in eigenvalues is not as evident as it is in latter levels, where clusters are better isolated.

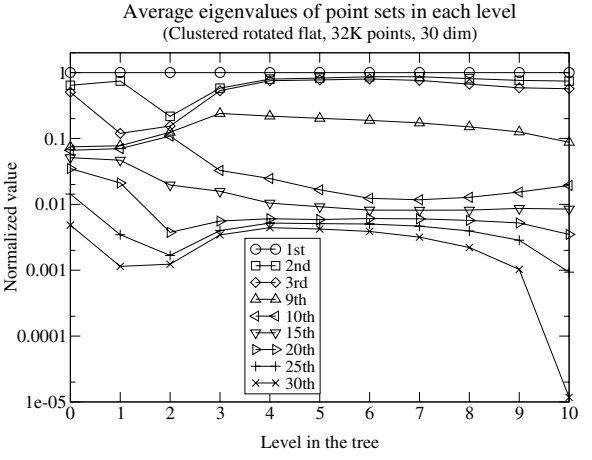


Fig. 4. Eigenvalues of point sets in each level in the tree. Synthetic data set

4.2 Number of Points in the Overlap Region

In the second set of experiments, we investigated the fraction of data points of each node that fall in the overlap region between the boundary hyperplanes. These are points for which the search may visit both children, and hence is related to the value of b described in Section 3.

Fig. 5 shows the fraction of points that are in the overlap region from our real data sets. Again, they show the average value for nodes at the same level, where level 0 is the root. The size of the overlap region depends on the value of desired failure probability. To achieve a small failure probability, a large training set is needed. Additional points in the training set may widen the overlap region. Consequently, the overlap region may contain more data points. The fraction usually falls between 0.3 and 0.7 in various levels in the tree. Note that near the leaf level this fraction drops significantly. This is because there are very few points in the node relative to the dimension of the space. Therefore the Voronoi diagram can be approximated much better by a single hyperplane. Similar behaviors are also observed for NASA satellite data and the synthetic data set that we tested.

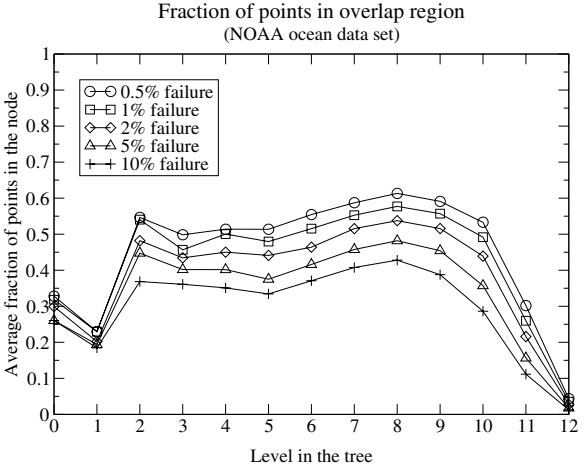


Fig. 5. Fraction of points in overlap region. NOAA ocean data sets.

4.3 Comparison with kd-Tree Search

We considered the search performance of the os-tree against an approximate version of the the well-known kd-tree data structure [8]. The difficulty in comparing these data structures is that the search models are different: probably-correct search in the os-tree and approximate nearest neighbor search in the kd-tree. To produce a realistic comparison, we adjusted the approximation factor (ϵ) of the kd-tree so that the resulting failure probability of the kd-tree matches that of the os-tree.

We show the results for the synthetic data sets only. The comparative results of the real data sets as well as other synthetic data sets were presented in [12]. We used different parameters from the other experiments. The number of points is varied from 2K to 32K, the number of rotations is equal to the dimension, $\sigma_{\text{thin}} = 0.01$.

Fig. 6 compares the performance of both trees as we vary the number of dimensions, and the number of points. The average query time is used as the metric. From the figure, we see that the os-tree is competitive with the kd-tree except for high dimensional instances. If the number of leaf nodes visited is used as the measure the search performance, the os-tree search visits fewer nodes than the kd-tree in almost all cases. The reason for the differences in CPU times is largely due to the additional overhead suffered by the os-tree. Through the use of incremental distance calculation [1], the processing time at each node of the kd-tree is independent of dimension, while it is $O(d)$ for the os-tree.

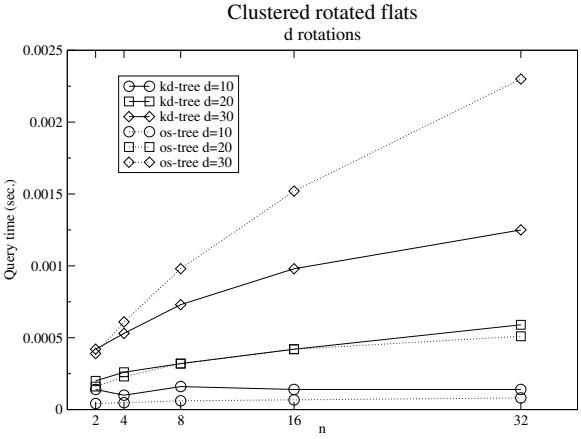


Fig. 6. Performance comparison. Using query time (sec) as the metric.

References

1. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.
2. P. Ciaccia and M. Patella. Using the distance distribution for approximate similarity queries in high-dimensional metric spaces. In *Proc. 10th Workshop Database and Expert Systems Applications*, pages 200–205, 1999.
3. K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.
4. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 1997.
5. R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, New York, NY, 1973.
6. C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, pages 4–13, 1994.

7. U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining*. AAAI Press/Mit Press, 1996.
8. J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Software*, 3(3):209–226, 1977.
9. K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, 2nd edition, 1990.
10. A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic, Boston, MA, 1992.
11. P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 604–613, 1998.
12. S. Maneewongvatana and D. Mount. An empirical study of a new approach to nearest nearbor searching. In *ALENEX*, 2001.
13. V. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, New York, NY, 1998.

I/O-Efficient Shortest Path Queries in Geometric Spanners

Anil Maheshwari[1,⋆], Michiel Smid[2], and Norbert Zeh[1,⋆]

[1] School of Computer Science, Carleton University, Ottawa, Canada
{maheshwa,nzeh}@scs.carleton.ca
[2] Fakultät für Informatik, Otto-von-Guericke-Universität, Magdeburg, Germany
michiel@isg.cs.uni-magdeburg.de

Abstract. We present I/O-efficient algorithms to construct planar Steiner spanners for point sets and sets of polygonal obstacles in the plane, and for constructing the "dumbbell" spanner of [6] for point sets in higher dimensions. As important ingredients to our algorithms, we present I/O-efficient algorithms to color the vertices of a graph of bounded degree, answer binary search queries on topology buffer trees, and preprocess a rooted tree for answering prioritized ancestor queries.

1 Introduction

Motivation: Geometric spanners are sparse subgraphs of the complete Euclidean graph over a set of points in $\mathbb{R}^d$. They play a key role in efficient algorithmic solutions for several fundamental geometric problems. Several efficient algorithms for constructing spanners in Euclidean space are known, including I/O-efficient algorithms [12], thereby enabling the processing of much bigger data sets that do not fit into internal memory. With respect to geometric shortest path problems, in internal memory, spanners are useful because they are sparse, so that approximate shortest path queries on the complete Euclidean graph, whose size is $\Theta(N^2)$, can be answered by solving the single-source shortest path (SSSP) problem on a graph of size $O(N)$. In external memory, sparseness is not sufficient to obtain I/O-efficient algorithms, as the best known single source shortest path algorithm takes $O(|V| + (|E|/B)\log_2 |E|)$ I/Os [14]. The focus of this paper is to construct spanners in such a way that spanner paths can be reported I/O-efficiently.

Computational Model and Previous Results: In the Parallel Disk Model (PDM) (see [17]), an external memory (EM) consisting of D disks is attached to a machine with an internal memory of size M. Each of the D disks is divided into blocks of B consecutive data items. Up to D blocks, at most one per disk, can be transferred between internal and external memory in a single I/O-operation. The complexity of an algorithm in this model is the number of I/O operations it performs. It has been shown that sorting an array of size N takes

⋆ Research supported by NSERC and NCE GEOIDE.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 287–299, 2001.

$\text{sort}(N) = \Theta((N/DB)\log_{(M/B)}(N/B))$ I/Os in the PDM (see [17]). Scanning an array of size N takes $\text{scan}(N) = \Theta(N/DB)$ I/Os. Due to the lack of space, we are forced to omit a discussion of previous related work, except for the most relevant results. However, we refer the reader to [15] for geometric spanners, [9] for the well-separated pair decomposition (WSPD), and [17] for external memory models and algorithms.

In [12] an algorithm to compute the WSPD and the corresponding spanner of a point set in $\mathbb{R}^d$ in $O(\text{sort}(N))$ I/Os using $O(N/B)$ blocks of external memory has been proposed. By carefully choosing spanner edges, the diameter of the spanner graph is shown to be at most $2\log_2 N$. Reporting a spanner path in this spanner takes $O(1)$ I/Os per edge in the path. Moreover, a lower bound of $\Omega(\min\{N, \text{sort}(N)\})$ I/Os for computing a t-spanner of a given point set is presented.

New Results: In Sec. 3, we present an algorithm to construct the dumbbell spanner of [6] for a set of N points in $\mathbb{R}^d$ in $O(\text{sort}(N))$ I/Os, show how to compute an augmented spanner of size $O(N)$ and spanner diameter $O(\log_2 N)$ (resp. $O(\alpha(N))$) and how to report spanner paths in these two spanners in $O(\log_2 N/(DB))$ (resp. $O(\alpha(N))$) I/Os. These spanners are induced by a constant number of rooted trees, called dumbbell trees, so that reporting a spanner path reduces to reporting a path in one of these trees. The latter can be done I/O-efficiently [13]. To construct dumbbell spanners, we have to solve several interesting subproblems, including vertex coloring of graphs of bounded degree, answering prioritized ancestor queries, and answering queries on topology buffer trees (see Sec. 2). In Sec. 4, we present an external version of the algorithm of [5] to construct in $O(\text{sort}(N))$ I/Os a planar Steiner spanner of size $O(N)$ and spanning ratio $1+\epsilon$ for a given set of N points, or polygonal obstacles with N vertices in the plane. Planarity is desirable, as planar graphs can be blocked [1], and an I/O-efficient single source shortest path algorithm for embedded planar graphs is known [3]. Also, planar graphs can be preprocessed for fast shortest path queries [13].

Preliminaries: A Euclidean graph $G = (V, E)$ is a *t-Steiner spanner* for the complete Euclidean graph $\mathcal{E}(S)$ defined on a set S of points in $\mathbb{R}^d$, if $S \subseteq V$ and for every pair of vertices $p, q \in S$, $\text{dist}_G(p, q) \leq t \cdot \text{dist}_2(p, q)$. The vertices in $V \setminus S$ are called *Steiner points*. G is a t-Steiner spanner for the visibility graph $\mathcal{V}(P)$ defined on a set P of polygonal obstacles with vertex set S, if $S \subseteq V$ and no edge in G crosses the interior of any obstacle in P, and for every pair of vertices $p, q \in S$, $\text{dist}_G(p, q) \leq t \cdot \text{dist}_{\mathcal{V}(P)}(p, q)$. We call graph G a *t-spanner* if $V = S$.

Given an axes-parallel box R and a point set S contained in R, a *fair split* of R is a partition of R into two boxes R_1 and R_2, each containing at least one point in S, using an axes-parallel hyperplane H; the distance of H from the two sides of R parallel to H has to be at least $\ell/3$, where ℓ is the shortest side of the bounding box of S. Given a point set S, let $R(S)$ be its bounding box, and $\hat{R}(S)$ be the smallest axes-parallel hypercube containing S and centered at the center of $R(S)$. The following recursive procedure defines a *fair split tree* $T(S)$ for S: If $|S| = 1$, then $T(S)$ consists of the single node S. Otherwise, we split $\hat{R}(S)$ into

two non-empty boxes $R'(S_1)$ and $R'(S_2)$, using a fair split. $\hat{R}(S_i)$ is defined as an axes-parallel box of aspect-ratio at most 3 which is contained in $R'(S_i)$, contains S_i, can be partitioned using a fair split, and such that every side e of $\hat{R}(S_i)$ either coincides with the corresponding side e' of $R'(S_i)$ or is at distance at least $l'/3$ from e', where l' is the length of the side of $R'(S_i)$ perpendicular to e'. Now recursively compute trees $T(S_1)$ and $T(S_2)$ and make their roots children of S. Note that every node in $T(S)$ corresponds to a unique subset of S. We identify this subset with the node.

A *well-separated pair decomposition* (WSPD) of S consists of a fair split tree $T(S)$ and a set of pairs $\{\{A_1, B_1\}, \{A_2, B_2\}, \ldots, \{A_m, B_m\}\}$ such that A_i, B_i are nodes in $T(S)$, for $1 \leq i \leq m$, for every pair of points $p, q \in S$, there is a unique pair $\{A_i, B_i\}$ such that $p \in A_i$ and $q \in B_i$, and for all $1 \leq i \leq m$, $\text{dist}_2(p, q) \geq s \cdot \text{dist}_2(x, y)$, for all $p \in A_i$, $q \in B_i$, and either $x, y \in A_i$ or $x, y \in B_i$. The real number $s > 0$ is called the *separation constant*.

2 Techniques

Coloring Graphs of Bounded Degree: Given a graph G with N vertices and maximal degree bounded by some constant Δ, the following algorithm colors the vertices of G with $\Delta + 1$ colors so that any two adjacent vertices in G have different colors: Number the vertices of G in their order of appearance in the given vertex list, and direct the edges of G from the vertices with smaller numbers to those with larger numbers. We now process the vertices by increasing numbers, one at a time. When processing a vertex we color it with the smallest color different from the colors of its in-neighbors. This technique can easily be realized in $O(\text{sort}(N))$ I/Os using the time-forward processing technique [10,2].

Prioritized Ancestor Queries: Given a rooted tree T and an assignment of priorities $\text{priority}(v) \in \{0, 1, \ldots, k\}$ to the vertices of T, we want to build a data structure $\mathcal{D}$ that allows answering queries of the following type in $O(1)$ I/Os: Given a vertex v and an ancestor u of v in T, find the highest vertex priority h on the path π from v to u, and report the first vertex $\text{first}(v, u)$ and the last vertex $\text{last}(v, u)$ on π with priority h. We call these queries *prioritized ancestor queries*. We show how to find $\text{first}(v, u)$. A slight modification of this procedure finds $\text{last}(v, u)$. We augment T with an artificial root r with $\text{priority}(r) = k + 1$, and make r the parent of the original root of T. Let $W_i = \{v \in T : \text{priority}(v) = i\}$. Then every node $v \in W_i$, $i \leq k$, has an ancestor of higher priority. For every node v, let $p'(v)$ be the lowest ancestor u of v in T such that $\text{priority}(u) > \text{priority}(v)$. Then we define a tree T' by making v the child of $p'(v)$.

Lemma 1. *Given a node $v \in T$ and an ancestor u of v in T, $\text{first}(v, u) = u$ or $\text{first}(v, u)$ is a child of the lowest common ancestor of u and v in T', denoted by $\text{LCA}_{T'}(v, u)$.*

We compute a binary tree $\mathcal{T}'$ with $|\mathcal{T}'| = O(|T|)$ from T' as follows: Replace every node v with children $w_1, \ldots, w_t$ by a path $v_1, \ldots, v_t$ of new nodes, such that v_{i+1} is the right child of v_i, for $1 \leq i < t$. We call v_1 the *representative*

rep(v) of v and v_t its *anchor* anchor(v). Let the children $w_1, \ldots, w_t$ be sorted by decreasing depth in T. Then we make rep(w_i) the left child of v_i, for $1 \leq i \leq t$ and give node v_i a label left(v_i) $= w_i$. The following result follows from Lemma 1 and gives an $O(1)$ I/O procedure to report first(v, u).

Lemma 2. *Given a node v and an ancestor u of v in T,* first$(v, u) \in \{u, z\}$, *where* $z = \text{left}(\text{LCA}_{\mathcal{T}'}(\text{anchor}(v), \text{anchor}(u)))$.

We compute the parents of the nodes in T' for every set W_i separately. Let $T_0 = T$. Then we mark every vertex $w \in T_0$ with priority$(w) > 0$. For every node, we compute its lowest marked ancestor in T_0. This produces all parents $p'(v)$, $v \in W_0$. We remove all vertices in W_0 from T_0 and make every vertex $w \notin W_0$ the child of its lowest marked ancestor in T_0. Let T_1 denote the resulting tree. We now recursively apply this procedure to T_1 to obtain all parents $p'(v)$, for $v \notin W_0$. Using time-forward processing, each recursive step takes $O(\text{sort}(|T_i|))$ I/Os. Once T' has been computed, it takes $O(\text{sort}(|T'|))$ I/Os to compute $\mathcal{T}'$ from T' and to preprocess $\mathcal{T}'$ for answering LCA-queries in $O(1)$ I/Os. If $|W_i| \leq c|W_{i-1}|$, for some constant $0 < c < 1$ and $1 \leq i \leq k$, we obtain the following result.

Theorem 1. *Given a rooted tree T with vertex priorities* priority$(v) \in \{0, 1, \ldots, k\}$, *let* $W_i = \{v \in T : \text{priority}(v) = i\}$, $0 \leq i \leq k$. *If* $|W_i| \leq c|W_{i-1}|$, *for some constant* $0 < c < 1$ *and* $1 \leq i \leq k$, *it takes* $O(\text{sort}(N))$ *I/Os and* $O(N/B)$ *blocks of external memory to construct a data structure $\mathcal{D}$ that allows answering prioritized ancestor queries on T in $O(1)$ I/Os.*

Querying Topology Buffer Trees: Given a binary tree T whose nodes store $O(1)$ information each, we call a binary search query q *strongly local* on T if the information stored at a node w and an ancestor v of w is sufficient to decide whether all, none, or some of the answers to q in $T(v)$ are stored in $T(w)$, and we are required to report *all* answers to q stored in T. We call q *weakly local* on T if the information stored at a node v is sufficient to decide whether $T(v)$ contains an answer to q, and we have to report *one* answer to q.

A *topology tree* $\mathcal{T}$ [11] is a balanced representation of a possibly unbalanced binary tree T. $\mathcal{T}$ has height $O(\log_2 N)$, where N is the number of nodes in T, and allows answering weakly local binary search queries on T in $O(\log_2 N)$ time. To construct $\mathcal{T}$, one starts with a tree $T_0 = T$, and recursively constructs a sequence $T_0, T_1, \ldots, T_k$ of binary trees, where T_{i+1} is obtained from T_i by contracting a carefully chosen set of edges in T_i. The vertex set of $\mathcal{T}$ is the disjoint union of the vertex sets of trees $T_0, T_1, \ldots, T_k$. A vertex v in T_{i+1} is the parent of a vertex w in T_i if v is the result of contracting an edge $\{u, w\}$, or v is a copy of w in T_{i+1} and no edge $\{u, w\}$ in T_i has been contracted.

If T is static, we can extend the idea of [7] to obtain a topology buffer tree. We construct a topology tree $\mathcal{T}$ for T and cut it into layers of height $\log_2(M/B)$. Each layer is a collection of rooted trees. We contract each such tree into a single node. The resulting tree $\mathcal{B}$ is the topology buffer tree corresponding to T. $\mathcal{B}$ has height $O(\log_{(M/B)} N)$; each node of $\mathcal{B}$ represents a subtree of $\mathcal{T}$ of size $O(M/B)$ and has at most M/B children. Thus, every node of $\mathcal{B}$ fits into internal memory.

Combining the ideas of topology B-trees [7] and buffer trees [2], we obtain the following result.

Theorem 2. *Given a topology buffer tree $\mathcal{B}$ representing an N node binary tree T and $O(N)$ (weakly or strongly) local binary search queries, it takes $O(\text{sort}(N+K))$ I/Os and $O((N+K)/B)$ blocks of external memory to answer all queries, where K is the size of the output, provided that $M \geq B^2$.*

3 Spanners of Low Diameter

Given a point set S in $\mathbb{R}^d$, we want to construct linear size spanner graphs of spanner diameters $O(\log_2 N)$ and $O(\alpha(N))$ that can be represented by data structures to report spanner paths in $O(\log_2 N/(DB))$ and $O(\alpha(N))$ I/Os, respectively.

3.1 Dumbbell Trees

Given a WSPD $\mathcal{D}$ for S, let $T(S)$ be the fair split tree of $\mathcal{D}$, and $C = \hat{R}(S)$. We refer to the well-separated pairs of $\mathcal{D}$ as *dumbbells* (as they look like dumbbells if we connect the two centers of the bounding boxes of each well-separated pair by a straight line segment). We define the *length* of a dumbbell to be the length of this line segment. Refer to the two bounding boxes as the *heads* of the dumbbell. Also, refer to C as a head (which does not belong to any dumbbell). We want to partition the set of dumbbells into a constant number of groups such that the lengths of two dumbbells in the same group differ by a factor of at most 2 or by a factor of at least $1/\delta$, for some $0 < \delta < \frac{1}{2}$ to be defined later; the heads of two dumbbells in the same group whose lengths differ by a factor of at most two are required to have distance at least $c\ell$ from each other, where c is a constant to be specified later, and ℓ is the length of the shorter dumbbell. We call the former the *length grouping* property; the latter the *separation* property.

For every such group $\mathcal{G}$ of dumbbells we define a *dumbbell tree* $T_{\mathcal{G}}$ as follows: $T_{\mathcal{G}}$ contains one *dumbbell node* per dumbbell in $\mathcal{G}$, one *head node* per dumbbell head of the dumbbells in $\mathcal{G}$, and one node per point in S. The points in S are the leaves of $T_{\mathcal{G}}$. The head node corresponding to the special head C is the root of $T_{\mathcal{G}}$. For every dumbbell $\{A, B\}$, heads A and B are the children of $\{A, B\}$. The parent of dumbbell $\{A, B\}$ is the smallest head node containing one of its heads. Thus, every node, except C, has a well-defined parent. Such a tree can be computed in $O(\text{sort}(N))$ I/Os per group by marking all nodes in the fair split tree corresponding to dumbbell heads in the group, making every leaf of the fair split tree a child of its lowest marked ancestor, and making every dumbbell node the child of the lowest marked ancestor of one of its heads. Leaves and dumbbell nodes without marked ancestors are children of the head node C.

In order to compute groups $\mathcal{G}$ with the above properties, we first compute $O(1)$ groups having the length-grouping property and then refine these groups to ensure the separation property. The length grouping property can be guaranteed

by simulating the algorithm of [6] in external memory, which takes $O(\text{sort}(N))$ I/Os. In particular, we compute a number of groups $\mathcal{G}_{i,j}$, where $0 \leq j \leq b$, for some constant b, such that each group $\mathcal{G}'_j = \bigcup_i \mathcal{G}_{i,j}$ has the length grouping property and the dumbbells in each group $\mathcal{G}_{i,j}$ differ by a factor of at most two in length. To guarantee the separation property, we partition each group $\mathcal{G}_{i,j}$ into $O(1)$ subgroups $\mathcal{G}_{i,j,k}$ such that the dumbbells in each subgroup satisfy the separation property. We merge groups $\mathcal{G}_{i,j,k}$ into $O(1)$ groups $\mathcal{G}'_{j,k} = \bigcup_i \mathcal{G}_{i,j,k}$, each having the length grouping and separation properties. Consider one particular group $\mathcal{G}_{i,j}$, and let ℓ be the length of the shortest dumbbell in $\mathcal{G}_{i,j}$.

In order to compute groups $\mathcal{G}_{i,j,k}$, we need to modify the dumbbells in $\mathcal{D}$ slightly. Consider a dumbbell $D = \{A, B\}$, and let $D' = \{A', B\}$ be its parent in the computation tree.[1] Then A' has been split into two boxes A_1 and A_2, where A is contained in A_1. In the following, we will consider $\{A, B\}$ to be dumbbell $\{A_1, B\}$. It follows from the properties of a fair split tree and its WSPD that the shortest side of head A_1 has length at least $\ell' = \frac{\ell}{(3s+12)\sqrt{d}}$.

The core of our algorithm is the construction of a *proximity graph* $\mathcal{P}$ containing one vertex per dumbbell in $\mathcal{G}_{i,j}$ and an edge between two vertices if the two corresponding dumbbells are too close. We do this as follows: For every dumbbell $\{A, B\} \in \mathcal{G}_{i,j}$, put a box $\mathcal{B}$ of side length $(c+8/s+4)\ell$ around the center of head A. Then every dumbbell $\{E, F\} \in \mathcal{G}_{i,j}$ that is too close to $\{A, B\}$ must have both its heads within this box. Partition $\mathcal{B}$ into $O(1)$ grid cells of side length $\ell'/2$. Then head E_1 must contain at least one of the grid vertices because it has side length at least ℓ'. Thus, if p is a grid point generated by dumbbell $\{A, B\}$, and $\{E, F\}$ is a dumbbell whose enlarged head E_1 contains p, we add an edge between the vertices corresponding to $\{A, B\}$ and $\{E, F\}$ to $\mathcal{P}$. Next we show how to find all dumbbell heads E_1 containing a grid point p.

The set of dumbbell heads containing a point p are stored along a path in the fair split tree T. Only a constant number of them can be heads of dumbbells in $\mathcal{G}_{i,j}$, as the minimal side length of a dumbbell head in $\mathcal{G}_{i,j}$ is at most $2\ell/s$, the minimal side length of the parent of a dumbbell head in $\mathcal{G}_{i,j}$ is at least ℓ', and the side lengths of the boxes along a root-to-leaf path in the fair split tree decrease by a factor of $2/3$ every d steps. For every grid point p, we report all these heads using strongly local binary search on T. The total number of heads reported for all grid points and all dumbbells is $O(N)$. It takes sorting and scanning to find the dumbbells in $\mathcal{G}_{i,j}$ having the reported heads and to add the corresponding edges to $\mathcal{P}$. It follows from standard packing arguments that $\mathcal{P}$ has bounded degree, so that we can compute a vertex coloring of $\mathcal{P}$ with a constant number of colors. The resulting color classes are the desired subgroups $\mathcal{G}_{i,j,k}$ of $\mathcal{G}_{j,k}$.

Lemma 3. *Given a point set S in $\mathbb{R}^d$ and a WSPD $\mathcal{D}$ for S, it takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to partition the dumbbells of $\mathcal{D}$ into $O(1)$ groups, each having the length grouping and separation properties. Each group can be represented by a dumbbell tree. The construction of all dumbbell trees takes $O(\text{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory.*

[1] See [9] for the definition of computation trees. Intuitively, A' is the parent of A in the fair split tree, $\{A', B\}$ is not well-separated, and A' is larger than B.

3.2 Spanners of Logarithmic Diameter

Let $T_1, \ldots, T_r$ be the dumbbell trees constructed in the previous section. Then we construct graphs $G_1, \ldots, G_r$, each having vertex set S, from those trees. We merge all these graphs $G_1, \ldots, G_r$ into a spanner G.

We construct G_i as follows: For every node $v \in T_i$, let $\omega(v) = |T_i(v)|$. We choose a representative point $r(v)$, for every node $v \in T_i$. If v is a leaf, then $r(v)$ is the point represented by v. Otherwise, let $w_1, \ldots, w_k$ be the children of v in T_i. Then $r(v) = r(w_j)$, where $\omega(w_j) = \max\{\omega(w_h) : 1 \leq h \leq k\}$. We add an edge $\{r(v), r(w)\}$ to G_i, for every edge $\{v, w\}$ in T_i with $r(v) \neq r(w)$.

For two points $p, q \in S$, let $\{A, B\}$ be the unique dumbbell such that $p \in A$ and $q \in B$. Let T_i be the dumbbell tree containing the dumbbell node corresponding to $\{A, B\}$, and let v and w be the two leaves of T_i such that $p = r(v)$ and $q = r(w)$. There is a unique path $\pi = (v = v_0, v_1, \ldots, v_k = w)$ from v to w in T_i. This path corresponds to a path $\tilde{\pi} = (p = r(v_0), r(v_1), \ldots, r(v_k) = q)$ in G_i. It is shown in [6] that $\tilde{\pi}$ has length at most $t \cdot \mathrm{dist}_2(p, q)$ if we choose $s = O(d/(t-1))$, $\delta = 1/s$, and $c = 2/\delta$ in the construction of the dumbbell trees. Thus, graph G is a t-spanner. Moreover, once we know tree T_i such that G_i contains the spanner path $\tilde{\pi}$ as constructed above, we can easily report $\tilde{\pi}$ by traversing the paths from v and w to their LCA in T_i; but π may be much longer than $\tilde{\pi}$ because many nodes along π may have the same representative. Observe, however, that all nodes in T_i with the same representative $r(v)$ form a path from the leaf ℓ with $r(\ell) = r(v)$ to some ancestor of ℓ in T_i. We construct a tree T_i' by compressing all non-leaf nodes on such a path into a single node. It follows from the choice of representatives in T_i that tree T_i' has height at most $\log_2 N + 1$, so that we can report $\tilde{\pi}$ in $O(\log_2 N/(DB))$ I/Os [13]. Unfortunately, we do not know which of the dumbbell trees contains the dumbbell node corresponding to $\{A, B\}$. However, as there are only $O(1)$ dumbbell trees, we can afford to query all dumbbell trees and report the shortest path found.

Theorem 3. *It takes $O(\mathrm{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to construct a t-spanner of spanner diameter $O(\log_2 N)$ and size $O(N)$ for a given set S of N points in $\mathbb{R}^d$, along with a data structure using $O(N/B)$ blocks of external memory that allows reporting a t-spanner path with $O(\log_2 N)$ edges between any two query points in $O(\log_2 N/(DB))$ I/Os.*

3.3 Spanners of Nearly Constant Diameter

Next we present an I/O-efficient algorithm to reduce the spanner diameter of all graphs G_i to $O(\alpha(N))$ and to construct a data structure that allows spanner paths with $O(\alpha(N))$ edges to be reported at a cost of $O(1)$ I/Os per edge. The construction is based on [16]. The idea is to augment every dumbbell tree T_i' with additional edges between nodes and subsets of their ancestors, so that the shortest monotone path from a node v to one of its ancestors contains $O(\alpha(N))$ edges, where a path is *monotone* if its nodes appear in the same order along a leaf-to-root path in T_i. Let T_i° be the resulting graph and G_i° be the supergraph

of G_i containing edges $\{r(v), r(w)\}$, $\{v, w\} \in T_i^\circ$. For a node v and an ancestor u of v, let π be the path from v to u in T_i', and π° be the shortest monotone path from v to u in T_i°. Let $\tilde{\pi}$ and $\tilde{\pi}^\circ$ be the corresponding spanner paths in G_i and G_i°. Then $\tilde{\pi}^\circ$ is no longer than $\tilde{\pi}$, by the triangle inequality.

We need some definitions: Given a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ such that $f(0) = 0$ and $f(x) < x$, for $x > 0$, we define $f^{(0)}(x) = x$ and $f^{(i)}(x) = f(f^{(i-1)}(x))$, for $i > 0$. Then $f^*(x) = \min\{k \geq 0 : f^{(k)}(x) \leq 1\}$. We define a series of functions ϕ_i, where $\phi_0(x) = \lfloor \sqrt{x} \rfloor$ and $\phi_i(x) = \phi_{i-1}^*(x)$, for $i > 0$. For a forest F and a set W of vertices in F, let $F \cap W$ be the forest obtained by contracting every maximal subtree whose non-root nodes are not in W to a single node. Let $F \star W$ be the set of edges containing edges between every vertex v in W and all its ancestors and descendants u in $F \setminus W$ so that the path from u to v does not contain a vertex in $W \setminus \{v\}$. Denote the irreflexive transitive closure of a DAG G by G^+. Given two parameters $0 \leq k < x$, let $V(x, k)$ be the set of vertices in F at levels $k, k + x, k + 2x, \ldots$. We call $V(x, 0), \ldots, V(x, x - 1)$ the *x-strided levels* of F.

The algorithm of [16] consists of two parts. The top-level procedure Shortcut computes the 13-strided level W of forest F which has minimum size, outputs edges $F \star W$ to be added to F, and then recursively shortcuts $F \cap W$, calling procedure RecShortcut with parameter $\beta = \min\{k \geq 0 : \phi_k(h(F)) \leq 4\}$, where $h(F)$ is the maximum height of any tree in F. Procedure RecShortcut computes two parameters $x_1 = \phi_{\beta-1}(h(F))$ and $x_2 = 3$, and the minimum x_1-strided level V_1 of F. If $\beta = 1$, it outputs the edges in $(F \cap V_1)^+ \cup (F \star V_1)$ to be added to F and recursively calls RecShortcut$(F \setminus V_1, 1)$. If $\beta > 1$, let $F_1 = F \cap V_1$, V_2 be the minimum x_2-strided level of F_1, and $F_2 = F_1 \cap V_2$; then RecShortCut returns the edges in $(F \star V_1) \cup F_1 \cup F_2$ to be added to F and recursively shortcuts $F \setminus V_1$ and F_2 by invoking RecShortcut$(F \setminus V_1, \beta)$ and RecShortcut$(F_2, \beta - 1)$.

It is shown in [16] that the graph T° produced by augmenting a rooted tree T with the output of Shortcut(T) has size $O(|T|)$. The shortest monotone path in T° from a node v to any of its ancestors u in T contains $O(\alpha(|T|))$ edges. A data structure to find the shortest monotone path in T° between two query vertices v and u can be derived quite naturally from the computation of algorithm Shortcut. In particular, denote the input forest to Shortcut by $F = F_{2\beta+1}$, and consider the tree $\mathcal{R}$ of recursive calls to RecShortcut triggered by Shortcut. Then for every β, the recursive invocations with parameter β form a set of paths in $\mathcal{R}$. For each such path p_i, let $\mathcal{I}_i$ be the topmost node in $\mathcal{R}$. Then we define a forest $F_{2\beta}$ as the union of the input forests to all such top-level invocations $\mathcal{I}_i$. We define another forest $F_{2\beta-1}$ as the union of forests F_1 in the description of RecShortcut for invocations with parameter β. If $\beta = 1$, then forest $F_{2\beta-1}$ does not exist. Thus, we obtain a sequence $F_{2\beta+1}, \ldots, F_2$ of forests.

Given a forest F which is the input to invocation $\mathcal{I}$, let $\mathcal{J}_1, \ldots, \mathcal{J}_k$ be the descendants of $\mathcal{I}$ in $\mathcal{R}$ that represent invocations of RecShortcut with parameter β. That is, $\mathcal{I} = \mathcal{J}_1$ represents RecShortcut(F, β), $\mathcal{J}_2$ represents RecShortcut$(F \setminus V_1, \beta)$, and so on. Then every node v of F appears in the set V_1 for at most one invocation $\mathcal{J}_i$. We give v priority i. For forests $F_{2\beta-1}$, we

give nodes in $F_1 \cap V_2$ priority 1. For forest $F_{2\beta+1}$, all nodes in W get priority 1. All nodes with no priority label are assumed to have infinite priority.

Given this labeling, a shortest monotone path from a node $v = v_{2\beta+1}$ to its ancestor $u = u_{2\beta+1}$ can be found as follows: Since $F = F_{2\beta+1}$, we start by examining $F_{2\beta+1}$. Given two nodes v_γ and u_γ whose shortest monotone path in $F_2 \cup F_3 \cup \cdots \cup F_\gamma$ we want to find, we find the minimum priority i such that there is a vertex on the path from v_γ to u_γ in F_γ with priority i, and report the lowest and highest such vertices $v_{\gamma-1}$ and $u_{\gamma-1}$ on this path. There have to be edges $\{v_\gamma, v_{\gamma-1}\}$ and $\{u_{\gamma-1}, u_\gamma\}$ in T°. If $\gamma = 2$, there is also an edge $\{v_{\gamma-1}, u_{\gamma-1}\}$ in T°. Otherwise, we recursively find the shortest monotone path from $v_{\gamma-1}$ to $u_{\gamma-1}$ in $F_2 \cup F_3 \cup \cdots \cup F_{\gamma-1}$. If there is no vertex with finite priority on the path from v_γ to u_γ in F_γ, this path must be short. We report this path by traversing F_γ and do not recurse.

Finding vertices $v_{\gamma-1}$ and $u_{\gamma-1}$ in forest F_γ for two query vertices v_γ and u_γ is a prioritized ancestor query; we just report minimum priority ancestors instead of maximum priority ancestors. Thus, given data structures $\mathcal{F}_{2\beta+1}, \ldots, \mathcal{F}_2$ to answer prioritized ancestor queries on $F_{2\beta+1}, \ldots, F_2$, the above shortest path procedure takes $O(1)$ I/Os per step along the shortest path from v to u in T°, $O(\alpha(N))$ I/Os in total. It remains to show that these data structures can be built I/O-efficiently. The following lemma is crucial for this.

Lemma 4. *Given a forest F_γ, $2 \le \gamma \le 2\beta+1$, and a partitioning of the vertices of F_γ into subsets $W_1, \ldots, W_k$ such that W_i contains all vertices in F_γ having priority i, $1 \le i \le k$, $|W_i| \le \frac{1}{2}|W_{i+1}|$, for $1 \le i < k$.*

Forests $F_{2\beta+1}, \ldots, F_2$ are easily obtained by simulating the computation of procedures ShortCut and RecShortCut. By Theorem 1 and Lemma 4, it takes $O(\mathrm{sort}(N))$ I/Os to compute data structures $\mathcal{F}_{2\beta+1}, \ldots, \mathcal{F}_2$ from forests $F_{2\beta+1}, \ldots, F_2$, as the total size of the forests $F_{2\beta+1}, \ldots, F_2$ is $O(N)$ [16].

Theorem 4. *It takes $O(\mathrm{sort}(N))$ I/Os and $O(N/B)$ blocks of external memory to construct a t-spanner of spanner diameter $O(\alpha(N))$ and size $O(N)$ for a given set S of N points in $\mathbb{R}^d$, along with a data structure using $O(N/B)$ blocks of external memory that allows reporting a t-spanner path with $O(\alpha(N))$ edges between any two query points in $O(\alpha(N))$ I/Os.*

4 Planar Steiner Spanners

Given a set P of simple polygonal obstacles with vertex set S in the plane, we want to construct a planar Steiner spanner G of size $O(|S|)$ and spanning ratio $1+\epsilon$ for the visibility graph $\mathcal{V}(P)$ of P. Our algorithm follows the framework of [5]. It constructs a planar subdivision based on the position of the vertices in S and then combines this subdivision with the subdivision defined by the obstacle edges to obtain an L_1-Steiner spanner. A planar Euclidean Steiner spanner is computed by superimposing a constant number of planar L_1-Steiner spanners.

4.1 Planar L_1-Steiner Spanners for Point Sets

We make frequent use of a procedure interval(s, r) that partitions the segment s into subsegments of length r each by adding Steiner vertices on s. The following planar subdivision D' of a minimal axes-parallel square C containing all points of S is the basis for our spanner construction. The cells of D' are of two types. Let a *box* be an axes-parallel rectangle of aspect ratio at most 3. A *box cell* is a box and contains exactly one point of S. A *donut cell* is the set-theoretic difference of two boxes B and B', does not contain any point of S, and for every side e of B, the distance to the corresponding side e' of B' is either zero or at least $\|e'\|/6$.

Given such a subdivision D', construct a planar L_1-Steiner spanner D'' for S as follows: Perform interval$(e, \gamma\ell)$, for every edge e of D', where ℓ is the length of the shortest edge of the box to which e belongs, and $0 < \gamma < 1$ is an appropriately chosen constant to be defined later. For every cell R and every boundary edge e of R shoot rays orthogonal to e from the endpoints of e and from the Steiner vertices on e toward the interior of R until they meet another edge. For every box cell R containing a point $p \in S$, we also shoot rays from p in all four axes-parallel directions until they meet the boundary of R. To preserve the planarity of the resulting graph, we introduce all intersection points between such rays as Steiner vertices. The following lemma now follows from [5] and [13].

Lemma 5. *Given a set S of N points in the plane and a linear size subdivision D' as above, it takes $O(\text{sort}(N) + \text{scan}(N/\gamma^2))$ I/Os to construct a planar L_1-Steiner spanner of size $O(N/\gamma^2)$ and with spanning ratio $1 + 6\gamma$ for S.*

Subdivision D' is quite naturally derived from a fair split tree T for S. The rectangles $\hat{R}(v)$ associated with the leaves of T are the box cells of D'. These box cells cover almost all the square C containing all points in S. The uncovered parts of C can be covered by regions $R'(v) \setminus \hat{R}(v)$, where $R'(v)$ was shrunk to $\hat{R}(v)$ before splitting $\hat{R}(v)$. We include these regions as the donut cells of D'. Using the fair split tree construction of [12], and choosing $\gamma = \epsilon/6$, we obtain the following result.

Theorem 5. *Given a set S of N points in the plane and a constant $\epsilon > 0$, it takes $O(\text{sort}(N) + \text{scan}(N/\epsilon^2))$ I/Os to construct a planar L_1-Steiner spanner of size $O(N/\epsilon^2)$ and with spanning ratio $1 + \epsilon$.*

4.2 Planar Steiner Spanners among Polygonal Obstacles

First we construct a planar L_1-Steiner spanner for a given set P of polygonal obstacles with vertex set S in the plane. We construct the subdivision D' w.r.t. set S and combine it in an appropriate manner with the graph defined by the obstacles in P to obtain a subdivision D_2. The spanner is then constructed from D_2 in a manner similar to the construction of D''. Our algorithm to construct D_2 is based on [5]. However, we use only one (a, b)-tree to represent the sweep line status instead of using two balanced binary trees. This simplification is crucial to allow an I/O-efficient implementation of this procedure.

Let D_1 be the superimposition of subdivision $D' = (S', E')$, viewed as a graph, and the graph $D = (S, E)$ defined by the set of obstacles in P. That is, the vertex set of D_1 contains all vertices in $S \cup S'$ and all intersection points between edges in E' and E. The edge set of D_1 is obtained by splitting the edges in $E' \cup E$ at their intersection points with other edges. D_1 may have size $\Omega(N^2)$. That is why we base the construction of an L_1-spanner for P on a linear size subgraph D_2 of D_1, which we construct without constructing D_1 first.

We divide the regions of D_1 into two classes: A *red* region is a quadrilateral none of whose vertices is in $S \cup S'$. The remaining regions are *blue* regions. Let the *red graph* of D_1 be the subgraph of the dual of D_1 containing a vertex for every red region of D_1 and an edge between two vertices if the two corresponding regions share an edge that is part of the boundary of a box or donut cell. The connected components of the red graph are paths. The red regions along such a path are bounded by the same two obstacle edges and a set of edges in E'. We call such a set of red regions a *ladder*. The two obstacle edges on their boundaries are the *sides* of the ladder; the edges from E' are its *rungs*. Call the topmost horizontal rung of a ladder its *top rung*; we define *left*, *right*, and *bottom rungs* in a similar manner. All of these four types of rungs are called *extremal rungs*. We call a ladder *trivial* if it consists only of a single red region. Otherwise, it is *non-trivial*. Subdivision D_2 is obtained from D_1 by replacing every ladder in D_1 by a single region. It is shown in [5] that D_2 has size $O(N)$. Using arguments from [5] and a construction similar to that of Sec. 4.1, we obtain the following result.

Lemma 6. *Given a subdivision D_2 as defined above, it takes $O(\mathrm{sort}(N) + \mathrm{scan}(N/\gamma^2))$ I/Os to construct a planar L_1-Steiner spanner of size $O(N/\gamma^2)$ and spanning ratio $1 + 6\gamma$ for a given set P of polygonal obstacles with N vertices.*

In order to construct D_2, we use four plane sweeps to compute potential top, bottom, left, and right rungs. The resulting subdivision D_3 may still contain non-trivial ladders. But its size is $O(N)$, so that we can afford to construct D_3 explicitly and remove all non-extremal rungs to obtain D_2. We describe the construction of potential top rungs.

Let E'_h be the set of horizontal edges in E'. We use a bottom-up sweep to compute all potential top rungs. During the sweep, we maintain a set of intervals defined by intersections between the sweep line ℓ and obstacle edges. In particular, let $e_1, \ldots, e_k$ be the edges in E intersected by the sweep line from left to right. Then the intervals currently stored for ℓ are $(e_1, e_2), (e_2, e_3), \ldots, (e_{k-1}, e_k)$. An interval $I = (l, r)$ in this list is a *ladder interval* if there is a ladder between l and r, and we have already found at least one horizontal rung of that ladder. Otherwise, it is a *non-ladder interval*.

We start the sweep with a single non-ladder interval defined by the left and right boundaries of the square C containing the whole vertex set S. Event points of the sweep are the y-coordinates of edges in E'_h and endpoints of obstacle edges. We perform the following updates, depending on the type of event point. Let $I_1, \ldots, I_k$ be the current set of intervals defined by the sweep line. When we encounter a horizontal edge $e \in E'_h$, we find intervals I_l and I_r containing

the left and right endpoints of e. Intervals $I_{l+1}, \ldots, I_{r-1}$ now become ladder intervals with their current top rungs set to e. Intervals I_l and I_r become non-ladder intervals. If I_l or I_r was classified as a ladder interval before ℓ passed e, we output the top rung for I_l or I_r, respectively. Similar procedures are applied to update the interval list when the sweep line passes an obstacle vertex.

We use a buffer tree [2] to represent the sweep line status. In particular, we store the current set of intervals sorted from left to right in this tree.[2] Every node in the buffer tree stores a time-stamped tag classifying all intervals stored in this subtree as ladder or non-ladder intervals and describing the top rung in the case of a ladder interval. These tags are chosen so that for every interval, at any time, the most recent tag along the path from the root of the tree to the leaf storing the interval represents the type and top rung of the interval correctly.

When the sweep passes a horizontal edge e, we search for the leaves l_l and l_r of T storing I_l and I_r. Denote the paths from the LCA of l_l and l_r in T to l_l and l_r by p_l and p_r. For all right siblings of nodes on p_l, we store that their descendants store ladder intervals with top rung e. We do the same for the left siblings of nodes on p_r. As this is the most recent information added to T, all intervals between I_l and I_r are now tagged as ladder intervals with top rung e. Intervals I_l and I_r are being tagged as non-ladder intervals. It is easy to find the most recent tags for I_l and I_r on the way down p_l and p_r, so that the top rungs for I_l and I_r are output if necessary. The procedures to update the tree when the sweep line passes an endpoint of an obstacle edge are similar.

Theorem 6. *Given a set of polygonal obstacles in the plane with N vertices in total, a planar L_1-Steiner spanner of spanning ratio $1+\epsilon$ and size $O(N/\epsilon^2)$ can be computed in $O(\mathrm{sort}(N) + \mathrm{scan}(N/\epsilon^2))$ I/Os using $O(N/(\epsilon^2 B))$ blocks of external memory.*

Combining the final step of the algorithm of [5] with the red-blue line intersection algorithm of [4], we obtain the following corollary.

Corollary 1. *Given a set of polygonal obstacles in the plane with N vertices in total, a planar Euclidean Steiner spanner of spanning ratio $1+\epsilon$ and size $O(N/\epsilon^4)$ can be computed in $O(\mathrm{sort}(N/\epsilon^3) + \mathrm{scan}(N/\epsilon^4))$ I/Os using $O(N/(\epsilon^4 B))$ blocks of external memory.*

References

1. P.K. Agarwal, L. Arge, M. Murali, K.R. Varadarajan, J.S. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. *SODA'98*, 1998.
2. L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. *WADS'95*, pp. 334–345, 1995.
3. L. Arge, G.S. Brodal, L. Toma. On external memory MST, SSSP, and multi-way planar separators. *SWAT'2000*, 2000.

[2] As buffer trees can only be used to represent sets drawn from a total order, we have to augment the partial left-to-right order of the obstacle segments to a total order. We show in the full paper how to do this.

4. L. Arge, D.E. Vengroff, J.S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *ESA'95*, pp. 295–310, 1995.
5. S. Arikati, D.Z. Chen, L.P. Chew, G. Das, M. Smid, C.D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. *ESA'96*, pp. 514–528, 1996.
6. S. Arya, G. Das, D.M. Mount, J.S. Salowe, M. Smid. Euclidean spanners: Short, thin, and lanky. *STOC'95*, pp. 489–498, 1995.
7. P.B. Callahan, M. Goodrich, K. Ramaiyer. Topology B-trees and their applications. *WADS'95*, pp. 381–392, 1995.
8. P.B. Callahan, S.R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest neighbors and n-body potential fields. *J. ACM*, 42:67–90, 1995.
9. P.B. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications.* PhD thesis, Johns Hopkins, Baltimore, 1995.
10. Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter. External-memory graph algorithms. *SODA'95*, 1995.
11. Greg N. Frederickson. A data structure for dynamically maintaining rooted trees. *J. of Algorithms*, 24:37–65, 1997.
12. S. Govindarajan, T. Lukovszki, A. Maheshwari, N. Zeh. I/O-efficient well-separated pair decomposition and its applications. *ESA'2000*, pp. 220–231, 2000.
13. D. Hutchinson, A. Maheshwari, N. Zeh. An external memory data structure for shortest path queries. *COCOON'99*, pp. 51–60, 1999 (to appear in Disc. App. Maths).
14. V. Kumar, E.J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. *SPDC'96*, 1996.
15. M. Smid. Closest-point problems in computational geometry. J.-R. Sack, J. Urrutia (eds.), *Handbook of Computational Geometry*, pp. 877–936. North-Holland, 2000.
16. M. Thorup. Parallel shortcutting of rooted trees. *J. of Algorithms.*, 23:123–159, 1997.
17. J.S. Vitter. *External memory algorithms and data structures.* J. Abello, J.S. Vitter (eds.), *External Memory Algorithms and Visualization*, AMS, 1999.

Higher-Dimensional Packing with Order Constraints

Sándor P. Fekete[1], Ekkehard Köhler[2], and Jürgen Teich[3]

[1] Department of Mathematical Optimization TU Braunschweig, Braunschweig, Germany, sandor.fekete@tu-bs.de
[2] Department of Mathematics, TU Berlin, Berlin, Germany, ekoehler@math.tu-berlin.de
[3] Computer Engineering Laboratory, University of Paderborn, Paderborn, Germany, teich@date.upb.de

Abstract. We present a first exact study on higher-dimensional packing problems with order constraints. Problems of this type occur naturally in applications such as logistics or computer architecture and can be interpreted as higher-dimensional generalizations of scheduling problems. Using graph-theoretic structures to describe feasible solutions, we develop a novel exact branch-and-bound algorithm. This extends previous work by Fekete and Schepers; a key tool is a new order-theoretic characterization of feasible extensions of a partial order to a given comparability graph that is tailor-made for use in a branch-and-bound environment. The usefulness of our approach is validated by computational results.

1 Introduction

Scheduling and Packing Problems. Scheduling is arguably one of the most important topics in combinatorial optimization. Typically, we are dealing with a one-dimensional set of objects ("jobs") that need to be assigned to a finite set of containers ("machines"). Problems of this type can also be interpreted as (one-dimensional) packing problems, and they are NP-hard in the strong sense, as problems like 3-Partition are special cases.

Starting from this basic scenario, there are different generalizations that have been studied. Many scheduling problems have *precedence constraints* on the sequence of jobs. On the other hand, a great deal of practical packing problems consider *higher-dimensional* instances, where objects are axis-aligned boxes instead of intervals. More-dimensional packing problems arise in many industries, where steel, glass, wood, or textile materials are cut. The three-dimensional problem is important for practical applications such as container loading.

In this paper, we give the first study of problems that comprise both generalizations: these are higher-dimensional packing problems with order constraints—or, from a slightly different point of view, higher-dimensional scheduling problems. In higher-dimensional packing, these problems arise when dealing with precedence constraints that are present in many container-loading problems. Another practical motivation to consider multi-dimensional scheduling problems

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 300–312, 2001.

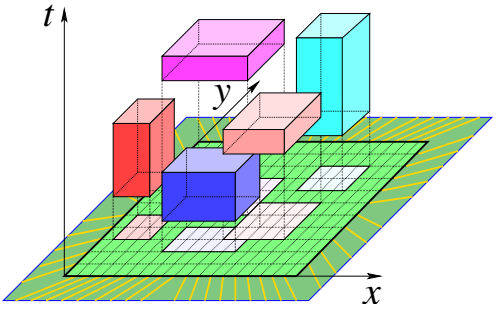


Fig. 1. *An FPGA and a set of five jobs, shown (as rectangles) in regular two-dimensional x,y-space and (as boxes) in three-dimensional space-time x, y, t. All jobs must be placed inside the chip and must not overlap if executed simultaneously on the chip.*

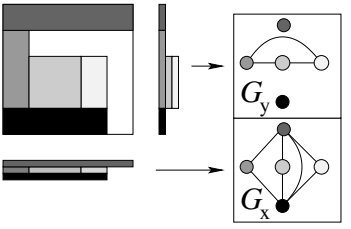


Fig. 2. *Projecting the boxes of a feasible packing onto the coordinate axes defines interval graphs (here in 2D: $\boldsymbol{G_x}$ and $\boldsymbol{G_y}$).*

arises from optimizing the reconfiguration of a particular type of computer chips called FPGA's—described below.

Field-Programmable Gate Arrays and More-Dimensional Scheduling. A particularly interesting class of instances of three-dimensional orthogonal packing arises from a new type of reconfigurable computer chips, called *field-programmable gate arrays* (FPGA's).

An FPGA typically consists of a regular rectangular grid of equal configurable cells (logic blocks) that allow the prototyping of simple logic functions together with simple registers and with special routing resources (see Figure 1). These chips (see e.g. [1,30]) may support several independent or interdependent jobs and designs at a time, and parts of the chip can be reconfigured quickly during run-time. (For more technical details on the underlying architecture, see our previous paper [28], and the more recent abstract [4].) Thus, we are faced with a general class of problems that can be seen as both scheduling and packing problems; two dimensions correspond to the space of the chip area, while the third dimension corresponds to time. In this paper, we develop a set of mathematical tools to deal with these *higher-dimensional scheduling problems*. We show that our methods are suitable for solving instances of interesting size to optimality.

Related Work. It is easy to see that any higher-dimensional packing problem (possibly with precedence constraints on the temporal order) can be relaxed to a resource-constrained scheduling problem. However, there are examples with as few as eight jobs showing that the converse is not true, even for small instances of two-dimensional packing problems without any precedence constraints: An optimal solution for the corresponding resource-constrained scheduling problem may not give rise to a feasible arrangement of rectangles for the original packing problem.

Higher-dimensional packing problems (without order constraints) have been considered by a great number of authors, but only few of them have dealt with the exact solution of general two-dimensional problems. See [6,8] for an overview. It should be stressed that unlike one-dimensional packing problems, higher-dimensional packing problems allow no straightforward formulation as integer programs: After placing one box in a container, the remaining feasible space will in general not be convex. Moreover, checking whether a given set of boxes fits into a particular container (the so-called *orthogonal packing problem*, OPP) is trivial in one-dimensional space, but NP-hard in higher dimensions.

Nevertheless, attempts have been made to use standard approaches of mathematical programming. Beasley [2] and Hadjiconstantinou and Christofides [15] have used a discretization of the available positions to an underlying grid to get a 0-1 program with a pseudopolynomial number of variables and constraints. Not surprisingly, this approach becomes impractical beyond instances of rather moderate size. More recently, Padberg [25] gave a *mixed integer programming* formulation for three-dimensional packing problems, similar to the one anticipated by Schepers [26] in his thesis. Padberg expressed the hope that using a number of techniques from branch-and-cut will be useful; however, he did not provide any practical results to support this hope.

In [6,8,9,10,28], a different approach to characterizing feasible packings and constructing optimal solutions is described. See Figure 2 for a visual description. A graph-theoretic characterization of the relative position of the boxes in a feasible packing (by so-called *packing classes*) is used, which represent d-dimensional packings by a tuple of d interval graphs (called *component graphs*). Any d-tuple of graphs G_i arising in this manner must satisfy the following conditions:

C1: G_i is an interval graph, $\forall i \in \{1, \cdots, d\}$.
C2: Any independent set S of G_i is i-admissible, $\forall i \in \{1, \cdots, d\}$, i.e., $w_i(S) = \sum_{v \in S} w_i(v) \leq h_i$, as all boxes in S must fit into the container in the ith dimension.
C3: $\cap_{i=1}^{d} E_i = \emptyset$. In other words, there must be at least one dimension in which the corresponding boxes do not overlap.

A d-tuple of component graphs satisfying these necessary conditions is called a *packing class*. The remarkable property (proven in [8,26]) is that these three conditions are also sufficient for the existence of a feasible packing.

Theorem 1
A d-tuple of graphs $G_i = (V, E_i)$ corresponds to a feasible packing, iff it is a packing class, i. e., if it satisfies the conditions **C1, C2, C3**.

This factors out a great deal of symmetries between different feasible packings, it allows to make use of a number of elegant graph-theoretic tools (like the characterizations reported in [12,13]), and it reduces the geometric problem to a purely combinatorial one without using brute-force methods like introducing an underlying coordinate grid. Combined with good heuristics for dismissing infeasible sets of boxes [7], a tree search for constructing feasible packings was

developed. This exact algorithm has been implemented; it outperforms previous methods by a clear margin. See our previous papers for details.

Graph Theory of Order Constraints. In the context of scheduling with precedence constraints, a natural problem is the following, called *transitive ordering with precedence constraints* (TOP): Consider a partial order $P = (V, \prec)$ of precedence constraints and a (temporal) comparability graph $G = (V, E)$, such that all relations in P are represented by edges in G. Is there a transitive orientation $D = (V, A)$ of G, such that P is contained in D?

Korte and Möhring [18] have given a linear-time algorithm for deciding TOP. However, their approach is only useful when the full set of edges in G is known. When running a branch-and-bound algorithm for solving a scheduling problem, these edges of G are only known partially, but they may already prohibit the existence of a feasible solution for a given partial order P. This makes it desirable to come up with structural characterizations that are already useful when only parts of G are known.

Results of this paper. In this paper, we give the first exact study of higher-dimensional packing with order constraints, which can also be interpreted as *higher-dimensional scheduling problems*. We develop a general framework for problems of this type by giving a pair of necessary and sufficient conditions for the existence of a solution for the problem TOP on graphs G in terms of forbidden substructures. Using the concept of packing classes described above, our conditions can be used quite effectively in the context of a branch-and-bound framework, since it can recognize infeasible subtrees at "high" branches of the search tree. In particular, we describe how to find an exact solution to the problem of minimizing the height of a container of given base area. If this third dimension represents time, this amounts to minimizing the makespan of a higher-dimensional scheduling problem. We validate the usefulness of these concepts and results by providing computational results. Other problem versions (like higher-dimensional knapsack or bin packing problems with order constraints) can be treated similarly.

The rest of this paper is organized as follows. In Section 2, we describe basic assumptions and some terminology. In Section 3, we introduce precedence constraints, describe the mathematical foundations for incorporating them into the search, and explain how to implement the resulting algorithms. Finally, we present computational results for a number of different benchmarks in Section 4.

2 Preliminaries

Problem instances. We assume that a problem instance is given by a *set of jobs* V. All our results can be applied to instances in arbitrary fixed dimension. For the purposes of this abstract, the reader may wish to focus on the scenario where each job has a *spatial requirement* in the x- and y-direction, denoted by $w_x(v)$ and $w_y(v)$, and possibly a *duration*, denoted by a size $w_t(v)$ along the

time axis. The available space H consists of an area of size $h_x \times h_y$. In addition, there may be an overall allowable time h_t for all jobs to be completed.

Graphs. Some of our descriptions make use of a number of different graph classes. An (undirected) graph $G = (V, E)$ is given by a set of vertices V, and a set of edges E; each edge describes the adjacency of a pair of vertices, and we write $\{u, w\}$ for an edge between vertices u and w. For a graph G, we obtain the *complement graph* $\overline{G}$ by exchanging the set E of edges with the set $\overline{E}$ of non-edges. In a directed graph $D = (V, A)$, edges are oriented, and we write (u, w) to denote an edge directed from u to w. A graph $G = (V, E)$ is a *comparability graph* if the edges E can be oriented to a set of directed arcs A, such that we get a (transitively closed) partial order, i.e., a cycle-free digraph for which the existence of edges $(u, v) \in A$ and $(v, w) \in A$ for any $u, v, w \in V$ implies the existence of $(u, w) \in A$.

Precedence constraints. Mathematically, a set of precedence constraints is given by a partial order $P = (V, \prec)$ on V. The relations in $\prec$ form a directed acyclic graph $D_P = (V, A_P)$, where A_P is the set of directed arcs. In the presence of such a partial order, a feasible schedule is assumed to satisfy the capacity constraints of the container, as well as these additional constraints.

Packing problems. In the following, we treat jobs as axis-aligned multi-dimensional boxes with given orientation, and feasible schedules as arrangements of boxes that satisfy all side constraints. This is implied by the term of a *feasible packing*. There may be different types of objective functions, corresponding to different types of packing problems. The *Orthogonal Packing Problem* (OPP) is to decide whether a given set of boxes can be placed within a given "container" of size $h_x \times h_y \times h_t$. For the Constrained OPP (COPP), we also have to satisfy a partial order $P = (V, \prec)$ of precedence constraints in the t-dimension. (To emphasize the motivation of temporal precedence constraints, we write t to suggest that the time coordinate is constrained, and x and y to imply that the space coordinates are unrestricted. Clearly, our approach works the same way when dealing with spatial restrictions.)

There are various optimization problems that have the OPP or COPP as their underlying decision problem. Since our main motivation arises from dynamic chip reconfigurations, where we want to minimize the overall running time, we focus on the *Constrained Strip Packing Problem* (CSPP), which is to minimize the size h_t for a given base size $h_x \times h_y$, such that all boxes fit into the container $h_x \times h_y \times h_t$. Clearly, we can use a similar approach for other objective functions.

3 Packing Problems with Precedence Constraints

As mentioned in the introduction, a key advantage of considering packing classes is that it allows to deal with packing problems independent of precise geometric placement, and that it allows arbitrary feasible interchanges of placement. However, for most practical instances, we have to satisfy additional constraints for the temporal placement, i.e., for the start times of jobs. For our approach, the nature of the data structures may simplify these problems from three-dimensional

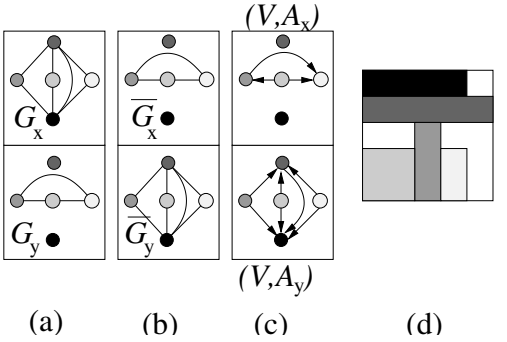


Fig. 3. *(a) A two-dimensional packing class, with G_x representing x-, G_y representing y-projections. (b) The corresponding comparability graphs. (c) A transitive orientation. (d) A feasible packing corresponding to the orientation.*

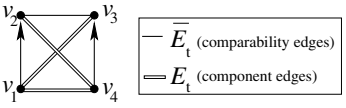


Fig. 4. *An example of a comparability graph $\overline{G_t} = (V, \overline{E_t})$ with a partial order P contained in $\overline{E_t}$, such that no transitive orientation of $\overline{G_t}$ extends P.*

to purely two-dimensional ones: If the whole schedule is given, all edges E_t in one of the graphs are determined, so we only need to construct the edge sets E_x and E_y of the other graphs. As worked out in detail in [27,28], this allows it to solve the resulting problems quite efficiently if the arrangement in time is already given.

A more realistic, but also more involved situation arises if only a set of precedence constraints is given, but not the full schedule. We describe in the following how further mathematical tools in addition to packing classes allow useful algorithms.

3.1 Packing Classes and Interval Orders

An important concept used frequently in the following is the concept of a *component graph*. Any edge $\{v_1, v_2\}$ in a component graph G_i corresponds to an overlap between the projections of boxes 1 and 2 onto the x_i-axis. This means that the complement graph $\overline{G_i}$ given by the complement $\overline{E_i}$ of the edge set E_i consists of all pairs of coordinate intervals that are "comparable": Either the first interval is "to the left" of the second, or vice versa. Any (undirected) graph of this type is a so-called *comparability graph* (see [14] for further details). By orienting edges to point from "left" to "right" intervals, we get a partial order of the set V of vertices, a so-called *interval order* [22]. Obviously, this order relation is transitive, i.e., $e \prec f$ and $f \prec g$ imply $e \prec g$, which is the reason why we also speak of a *transitive orientation* of the undirected comparability graph G_i. See Figure 3 for a (two-dimensional) example of a packing class, the corresponding comparability graph, a transitive orientation, and the packing corresponding to the transitive orientation.

Now consider a situation where we need to satisfy a partial order $P = (V, A_P)$ of precedence constraints in the time dimension. It follows that each arc $a = (u, w) \in A_P$ in this partial order forces the corresponding undirected edge $e = \{u, w\}$ to be excluded from E_t. Thus, we can simply initialize our algorithm for constructing packing classes by fixing all undirected edges corresponding to A_P to be contained in $\overline{E_t}$. After running the original algorithm, we may get additional comparability edges. As the example in Figure 4 shows, this causes an additional problem: Even if we know that the graph $\overline{G_t}$ has a transitive orientation, and all arcs $a = (u, w)$ of the precedence order (V, A_P) are contained in $\overline{E_t}$ as $e = \{u, w\}$, it is not clear that there is a transitive orientation that contains all arcs of A_P.

3.2 Finding Feasible Transitive Orientations

Consider a comparability graph $\overline{G}$ that is the complement of an interval graph G. The problem TOP of deciding whether $\overline{G}$ has a transitive orientation that extends a given partial order P has been studied in the context of scheduling. where $\overline{G}$ is the comparability graph of an interval order. For this scenario, Korte and Möhring [18] give a linear-time algorithm for determining a solution for TOP, or deciding that none exists. Their approach is based on a very special data structure called *modified PQ-trees*.

In principle, it is possible to solve higher-dimensional packing problems with precedence constraints by adding this algorithm as a black box to test the leaves of our search tree for packing classes: In case of failure, backtrack in the tree. However, the resulting method cannot be expected to be reasonably efficient: During the course of our tree search, we are not dealing with one fixed comparability graph, but only build it while exploring the search tree. This means that we have to expect spending a considerable amount of time testing similar leaves in the search tree, i.e., comparability graphs that share most of their graph structure. It may be that already a very small part of this structure that is fixed very "high" in the search tree constitutes an obstruction that prevents a feasible orientation of all graphs constructed below it. So a "deep" search may take a long time to get rid of this obstruction. This makes it desirable to use more structural properties of comparability graphs and their orientations to make use of obstructions already "high" in the search tree.

3.3 Implied Orientations

As in the basic packing class approach, we consider the component graphs G_i and their complements, the comparability graphs $\overline{G_i}$. This means that we continue to have three basic states for any edge: (1) edges that have been fixed to be in E_i, i.e., *component edges*; (2) edges that have been fixed to be in $\overline{E_i}$, i.e., *comparability edges*; (3) *unassigned edges*.

In order to deal with precedence constraints, we also consider orientations of the comparability edges. This means that during the course of our tree search, we can have three different possible states for each comparability edge: (2a)

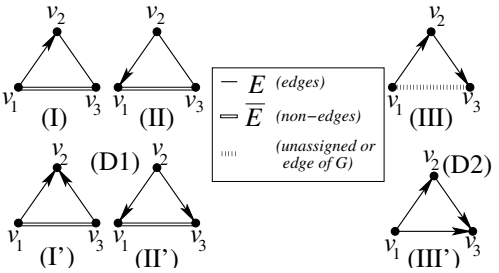


Fig. 5. *Two types of implications for edges and their orientations: Above are the arrangements that trigger path implications (D1, left) and transitivity implications (D2, right); below are the forced orientations of edges.*

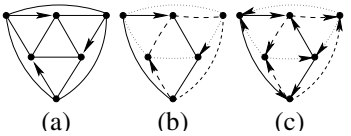


Fig. 6. *(a) A graph $\overline{G_t}$ with a partial order formed by three directed edges; (b) there are three path implication classes that each have one directed arc; (c) carrying out path implications creates directed cycles, i.e., transitivity conflicts.*

one possible orientation; (2b) the opposite possible orientation; (2c) no assigned orientation.

A stepping stone for this approach arises from considering the following two configurations – see Figure 5:

The first consists of the two comparability edges $\{v_1, v_2\}, \{v_2, v_3\} \in \overline{E_t}$, such that the third edge $\{v_1, v_3\}$ has been fixed to be an edge from the component graph E_t. Now any orientation of one of the comparability edges forces the orientation of the other comparability edge, as shown in the left part of the figure. We call this arrangement a *path implication*, since this configuration corresponds to an induced path in $\overline{G_i}$,

The second configuration consists of two directed comparability edges, say, the edges (v_1, v_2) and (v_2, v_3). In this case we know that the edge $\{v_1, v_3\}$ must also be a comparability edge, with an orientation of (v_1, v_3). Since this configuration arises directly from transitivity in $\overline{G_i}$, we call this arrangement a *transitivity implication.*

Clearly, any implication arising from one of the above configurations can induce further implications.

In particular, when considering only sequences of path implications, we get a partition of comparability edges into *path implication classes.* Two comparability edges are in the same implication class, iff there is a sequence of path implications, such that orienting one edge forces the orientation of the other edge. For an example, consider the arrangement in Figure 4. Here, all three comparability edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, and $\{v_3, v_4\}$ are in the same path implication class. Now the orientation of (v_1, v_2) implies the orientation (v_3, v_2), which in turn implies the orientation (v_3, v_4), contradicting the orientation of $\{v_3, v_4\}$ in the given par-

tial order P. It is not hard to see that the implication classes form a partition of the comparability edges, since we are dealing with an equivalence relation.

We call a violation of a path implication a *path conflict*.

As the example in Figure 6 shows, only excluding path conflicts when recursively carrying out path implications does not suffice to guarantee the existence of a feasible orientation: Working through the queue of path implications, we end up with a directed cycle, which violates a transitivity implication.

We call a violation of a transitivity implication a *transitivity conflict*.

Summarizing, we have the following necessary conditions for the existence of a transitive orientation that extends a given partial order P:

D1: Any path implication can be carried out without a conflict.
D2: Any transitivity implication can be carried out without a conflict.

These necessary conditions are also sufficient:

Theorem 2 (Fekete, Köhler, Teich)
Let $P = (V, A_P)$ be a partial order with arc set A_P that is contained in the edge set E of a given comparability graph $G = (V, E)$. A_P can be extended to a transitive orientation of G, iff all arising path implications and transitivity can be carried out without creating a path conflict or a transitivity conflict.

A proof and further mathematical details are described in our report [5]. The interested reader may take note that we are extending previous work by Gallai [11], who extensively studied implication classes of comparability graphs. In particular, we use the concept of *modular decomposition*. For more background on implication classes and comparability graphs, see Kelly [17], Möhring [22] for informative surveys on this topic, and Krämer [20] for an application in scheduling theory.

3.4 Solving Problems with Precedence Constraints

We start by fixing for all arcs $(u, v) \in A$ the edge $\{u, v\}$ as an edge in the comparability graph $\overline{G_t}$, and we also fix its orientation to be (u, v). In addition to the tests for enforcing the conditions for unoriented packing classes (C1, C2, C3), we employ the implications suggested by conditions D1 and D2. For this purpose, we check directed edges in $\overline{G_t}$ for being part of a triangle that gives rise to either implication. Any newly oriented edge in $\overline{G_t}$ gets added to a queue of unprocessed edges. Like for packing classes, we can again get cascades of fixed edge orientations. If we get an orientation conflict or a cycle conflict, we can abandon the search on this tree node. The correctness of the overall algorithm follows from Theorem 2; in particular, the theorem guarantees that we can carry out implications in an arbitrary order.

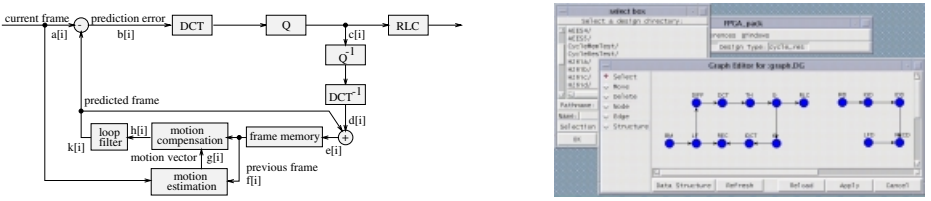


Fig. 7. *Block diagram of a video-codec (H.261) (left); precedence constraints for the video-codec (right).*

4 Computational Experiments

In the following we present our results for different types of instances: The video-codec benchmark described in Section 4.1 arises from an actual application to FPGA's. In Section 4.2 we give a number of results arising from different geometric packing problems.

Our code was implemented in C++ and tests were carried out on a SUN Ultra 2.

4.1 Video-Codec Benchmark

Figure 7 shows a block diagram of the operation of a hybrid image sequence coder/decoder that arises from the FPGA application. The purpose of the coder is to compress video images using the H.261 standard. In this device, transformative and predictive coding techniques are unified. The compression factor can be increased by a predictive method for motion estimates: blocks inside a frame are predicted from blocks of previous images. See our report [4] for more technical details. The result is shown in Table 1.

4.2 Geometric Instances

Here we describe computational results for two types of two-dimensional objects. See Table 2 for an overview. The first class of instances was constructed from a

Table 1. Optimizing reconfigurations for the Video-Codec

test	container sizes			CPU-time (s)
	h_t	h_x	h_y	
1	59	64	64	24.87 s

Table 2. Optimal packing with order constraints

instance	optimal h_t	h_x	upper bound	lower bound
okp17-0	169	100	7.29 s	179 s
okp17-1	172	100	6.73 s	1102 s
okp17-2	182	100	5.39 s	330 s
okp17-3	184	100	236 s	553 s
okp17-4	245	100	0.17 s	0.01 s
square21-no	112	112	84.28 s	0.01 s
square21-mat	117	112	15.12 s	277 s
square21-tri	125	112	107 s	571 s
square21-2mat	[118,120]	[118,120]	346 s	476 s

Table 3. The `okp17` problem instances.

`okp17`:	base width of container = 100, number of boxes = 17
sizes =	[(8,81),(5,76),(42,19),(6,80), (9,52),(41,48),(6,86),(58,20), (99,3),(100,14),(7,53),(24,54), (23,77),(42,32),(17,30),(11,90), (26,65)]
okp17-0:	no order constraints
okp17-1:	11→8, 11→16
okp17-2:	11→8, 11→16, 8→16
okp17-3:	11→8, 11→16, 8→16, 8→17, 11→7, 16→7
okp17-4:	11→8, 11→16, 8→16, 8→17, 11→7, 16→7, 17→16

Table 4. The `square21` problem instances.

`square21`:	base width of container = 112, number of boxes = 21
sizes =	[(50,50),(42,42),(37,37),(35,35), (33,33),(29,29),(27,27),(25,25), (24,24),(19,19),(18,18),(17,17), (16,16),(15,15),(11,11),(9,9), (8,8), (7,7),(6,6),(4,4),(2,2)]
square21-0:	no order constraints
square21-mat:	2→4, 6→7, 8→9, 11→15, 16→17, 18→19, 24→25, 27→29, 33→35, 37→42, 2→50, 50→4
square21-tri:	2→15, 15→17, 2→27, 4→16, 16→29, 4→29, 6→17, 17→33, 6→33, 7→18, 18→35, 7→35, 8→19, 19→37, 8→37, 9→24, 24→42, 9→42, 11→25, 25→50, 11→50
square21-2mat:	x-constraints: 2→19, 6→25, 8→29, 11→35, 16→42, 18→4, 24→7, 27→9, 33→15, 37→17, 50→4, 18→50 y-constraints: 2→4, 6→7, 8→9, 11→15, 16→17, 18→19, 24→25, 27→29, 33→35, 37→42, 2→50, 50→4

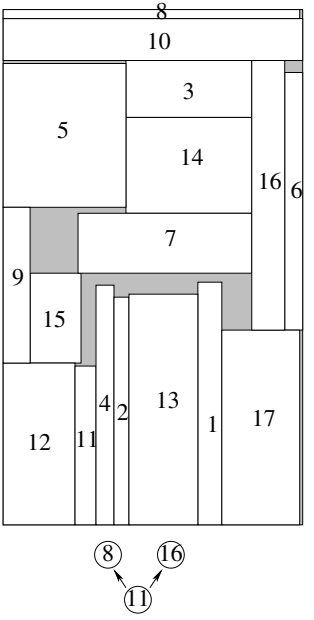


Fig. 8. *An optimal solution for the instance okp17-1.*

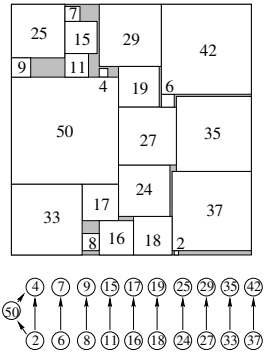


Fig. 9. *An optimal solution for the instance square21-mat.*

particularly difficult random instance of 2-dimensional knapsack (see [6]). Results are given for order constraints of increasing size. In order to give a better idea of the computational difficulty, we give separate running times for finding an optimal feasible solution, and for proving that this solution is best possible.

See Table 3 for the exact sizes of the 17 rectangles involved for the geometric layout of optimal packings. For easier reference, the boxes are labeled 1–17 in the order in which they are listed in the table.

The second class of instances arises from the well-known tiling of a 112x112 square by 21 squares of different sizes. Again, we have added order constraints of various sizes. (See Table 4 for details.) For the instance square21-2mat (with order constraints in two dimensions), we could not close the gap between upper and lower bound. For this instance, we report the running times for achieving the best known bounds.

Two examples of resulting packings are shown in Figures 8 and 9.

Acknowledgments. We are extremely grateful to Jörg Schepers for letting us continue the work with the packing code that he started as part of his thesis, and for several helpful hints, despite of his departure to industry. We also thank Marc Uetz for a useful discussion on resource-constrained scheduling.

References

1. Atmel. *AT6000 FPGA configuration guide*. Atmel Inc.
2. J. E. Beasley. An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, **33** (1985), pp. 49–64.
3. J. E. Beasley. OR-Library: distributing test problems by electronic mail. *Journal of the Operations Research Society*, **41** (1990), pp. 1069–1072.
4. S. P. Fekete, E. Köhler, and J. Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In: *Proc. DATE 2001, Design, Automation and Test in Europe*, pp. 658–665.
5. S. P. Fekete, E. Köhler, and J. Teich. Extending partial suborders and implication classes. Technical Report 697-2000, TU Berlin.
6. S. P. Fekete and J. Schepers. A new exact algorithm for general orthogonal d-dimensional knapsack problems. In *Algorithms – ESA '97*, volume 1284, pp. 144–156, Springer Lecture Notes in Computer Science, 1997.
7. S. P. Fekete and J. Schepers. New classes of lower bounds for bin packing problems. In *Proc. Integer Programming and Combinatorial Optimization (IPCO'98)*, volume 1412, pp. 257–270, Springer Lecture Notes in Computer Science, 1998.
8. S. P. Fekete and J. Schepers. On more-dimensional packing I: Modeling. Technical Report 97-288, Center for Applied Computer Science, Universität zu Köln, available at `http://www.zpr.uni-koeln.de/ABS/~papers`, 1997.
9. S. P. Fekete and J. Schepers. On more-dimensional packing II: Bounds. Technical Report 97-289, Universität zu Köln, 1997.
10. S. P. Fekete and J. Schepers. On more-dimensional packing III: Exact algorithms. Technical Report 97-290, Universität zu Köln, 1997.
11. T. Gallai. Transitiv orientierbare Graphen. *Acta Math. Acd. Sci. Hungar.*, **18** (1967), pp. 25–66.

12. A. Ghouilà-Houri. Caractérization des graphes non orientés dont on peut orienter les arrêtes de manière à obtenir le graphe d'une relation d'ordre. *C.R. Acad. Sci. Paris*, **254** (1962), pp. 1370–1371.
13. P.C. Gilmore and A.J. Hoffmann. A characterization of comparability graphs and of interval graphs. *Canadian Journal of Mathematics*, **16** (1964), pp. 539–548.
14. M. Golumbic. *Algorithmic graph theory and perfect graphs.* Academic Press, 1980.
15. E. Hadjiconstantinou and N. Christofides. An exact algorithm for general, orthogonal, two-dimensional knapsack problems. *European J. of Operations Research*, **83** (1995), 39–56.
16. C.-H. Huang and J.-Y. Juang. A partial compaction scheme for processor allocation in hypercube multiprocessors. In *Proc. of 1990 Int. Conf. on Parallel Proc.*, pp. 211–217, 1990.
17. D. Kelly. Comparability graphs. In I. Rival, editor, *Graphs and Order*, pp. 3–40. D. Reidel Publishing Company, Dordrecht, 1985.
18. N. Korte and R. Möhring. Transitive orientation of graphs with side constraints. In H. Noltemeier, editor, *Proceedings of WG'85*, pp. 143–160. Trauner Verlag, 1985.
19. N. Korte and R. H. Möhring. An incremental linear–time algorithm for recognizing interval graphs. *SIAM Journal of Computing*, **18** (1989), pp. 68-81.
20. A. Krämer. *Scheduling Multiprocessor Tasks on Dedicated Processors.* Doctoral thesis, Fachbereich Mathematik und Informatik, Universität Osnabrück, 1995.
21. E. L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and D. B. Shmoys. *Sequencing and Scheduling: Algorithms and Complexity.* in: S. C. Graves, A. H. G. Rinnooy Kan, and P. H. Zipkin. Logistics of Production and Inventory, vol. 4, Handbooks in Operations Research and Management, pp. 445–522. North–Holland, Amsterdam, 1993,
22. R. H. Möhring. Algorithmic aspects of comparability graphs and interval graphs. In I. Rival, editor, *Graphs and Order*, pages 41–101. D. Reidel Publishing Company, Dordrecht, 1985.
23. R. H. Möhring. Algorithmic aspects of the substitution decomposition in optimization over relations, set systems, and Boolean functions. *Annals of Oper. Res.*, **4** (1985), pp. 195–225.
24. R. H. Möhring, A. S. Schulz, F. Stork, and M. Uetz. Solving Project Scheduling Problems by Minimum Cut Computations. Technical Report 680-2000, TU Berlin.
25. M. Padberg. Packing small boxes into a big box. *Math. Meth. of Op. Res.*, **52** (2000), pp. 1–21.
26. J. Schepers. Exakte Algorithmen für orthogonale Packungsprobleme. Doctoral thesis, Universität Köln, 1997, available as Technical Report 97-302.
27. J. Teich, S. Fekete, and J. Schepers. Compile-time optimization of dynamic hardware reconfigurations. In *Proc. Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pp. 1097–1103, Las Vegas, U.S.A., June 1999.
28. J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *J. of Supercomputing*, **19** (2001), pp. 57–75.
29. J. Weglarz. *Project Scheduling. Recent Models, Algorithms and Applications.* Kluwers Academic Publishers, Norwell, MA, USA, 1999.
30. Xilinx. XC6200 field programmable gate arrays. Tech. report, Xilinx, Inc., October 1996.

Bin Packing with Item Fragmentation

Nir Menakerman and Raphael Rom

Department of Electrical Engineering
Technion - Israel Institute of Technology
Haifa 32000, Israel
mnir@techunix.technion.ac.il, rom@ee.technion.ac.il

Abstract. We investigate a variant of the bin packing problem in which items may be fragmented into smaller size pieces called fragments. While there are a few applications to bin packing with item fragmentation, our model of the problem is derived from a scheduling problem present in data over CATV networks. Fragmenting an item is associated with a cost which renders the problem NP-hard. We study two possible cost functions and as a result get two variants of bin packing with item fragmentation. In the first variant, called *bin packing with size-increasing fragmentation*, each item may be fragmented in which case overhead units are added to the size of every fragment. In the second variant each item has a size and a cost and fragmenting an item increases its cost but does not change its size. We call this variant *bin packing with size-preserving fragmentation.*
We develop several algorithms for the problem and investigate their performance. The algorithms we present are based on well known bin packing algorithms such as Next-Fit and First-Fit Decreasing, as well as of other algorithms. . . .

1 Introduction

Because of its applicability to a large number of applications and because of its theoretical interest bin packing has been widely researched and investigated (see, e.g., [4], [6], [9] and [2] for a comprehensive survey). In the classical one-dimensional bin packing problem, we are given a list of items $L = (a_1, a_2, ..., a_n)$, each with a size $s(a_i) \in (0, 1]$ and are asked to pack them into a minimum number of unit capacity bins. Since the problem, as many of its derivatives, is NP-hard many approximation algorithms have been developed for it (see, e.g., [3], [7], [8] and [1] for a survey). The common assumption in bin packing problems is that an item may not be fragmented into smaller pieces. There are several applications, however, in which this assumption does not hold. The subject of item fragmentation in bin packing problems received almost no attention so far. This paper concentrates on aspects that were heretofore never researched, such as developing algorithms for the problem and investigating their performance.

The variant of bin packing presented in this paper is derived from a scheduling problem present in data over CATV (community antenna television) networks. In particular we refer to Data-Over-Cable Service Interface Specification

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 313-324, 2001.

(DOCSIS), standard of the Multimedia Cable Network System (MCNS) standard committee, for a detailed description see [12]. When using CATV networks for data communication the data subscribers are connected via a cable modem to the headend. The headend is responsible for the scheduling of all transmissions in the upstream direction (from the cable modem to the headend). Scheduling is done by dividing the upstream, in time, into a stream of numbered mini-slots. The headend receives requests from the modems for allocation of datagram transmission. The length of each datagram can vary and may require a different number of mini-slots. From time to time, the headend publishes a *MAP* in which it allocates mini-slots to one modem or a group of modems. The scheduling problem is that of allocating the mini-slots to be published in the MAP, or in other words, how to order the datagrams transmission in the best possible way.

The headend must consider two kinds of datagram allocations:

1. *Fixed Location* - Allocations for connections with timing demands, such as a CBR (constant bit rate) connection. These connections must be scheduled so as to ensure delivering the guaranteed service. Fixed location datagrams are therefore scheduled in fixed, periodically located mini-slots.
2. *Free Location* - Allocations for connections without timing demands, such as a best effort connection. Free location datagrams can use any of the mini-slots.

The headend therefore performs the allocation in two stages: in the first stage it schedules, or allocates, all fixed location datagrams. We assume that after the fixed allocations have been made, a gap of U mini-slots is left between successive fixed allocations. In the second stage all free location datagrams are scheduled. The free allocations must fit into the gaps left by the fixed allocations.

The relation to the bin packing problem should now be clear. The items are the free location datagrams that should be scheduled, each of which may require a different number of mini-slots. The bins are defined by the gaps between every two successive fixed allocations in the MAP. The goal is to use the available mini-slots in the MAP in the best way. We point out that the practical scheduling problem may actually be somewhat more complicated since the bins (gaps between fixed allocation) may not be of uniform size. Dealing with variable size bins is beyond the scope of this paper (we present some results in [11]).

One of the capabilities of the system is the ability to break a datagram into smaller pieces called *fragments*. When a datagram is fragmented, i.e., transmitted in non successive mini-slots, extra bits are added to the original datagram to enable the reassembly of all the fragments at the headend. In a typical CATV network one mini-slot is added to every fragment.

To make the problem of bin packing with item fragmentation nontrivial a cost must be associated with fragmentation. We study two possible cost functions and as a result get two variants of bin packing with item fragmentation. In the first variant, called *bin packing with size-increasing fragmentation*, the cost function adds one (or more) overhead unit to the size of every fragment. In the second variant the cost function increases the cost of an item upon each fragmentation, but does not change its size. We call this variant, *bin packing*

with size preserving fragmentation. The scheduling problem, where the cost is due to the extra overhead bits that are added to each fragment, serves as a model to the first variant. The second variant is suitable for problems where the cost associated with fragmentation is a result of extra processing time or reassembly delay. The two cost functions we present here are not the only possible cost functions. In other applications, for example, the cost may be related to the size of the item, a combination of the two cost functions is also possible. It is interesting to note that when the cost associated with fragmentation is ignored the packing problem becomes trivial, and when the cost is very high it does not pay to fragment items and we face the classical problem. Hence, the problem is interesting with the middle-range costs. It has been shown in [10] that for non zero cost the ability to fragment items does not reduce the complexity of the problem, that is, the problem of bin packing with item fragmentation is NP-hard.

We present worst case analysis of both variants of bin packing with item fragmentation. We begin by showing that the two variants are NP-hard in the strong sense. We then devise approximation algorithms for the problem and investigate their performance. We restrict our attention to practical bin packing algorithms, i.e., of low order polynomial running time, and examine both *online* and *offline* algorithms. Online algorithms are applicable to cases where the items arrive in some order and must be assigned to the bins as soon as they arrive. Offline algorithms assume the entire list of items is known before the packing begins. We devise algorithms which are based on well known bin packing algorithms but include the capability of fragmenting items. We investigate the performance of algorithms such as Next-Fit (NF) and First-Fit Decreasing (FFD), as well as of other algorithms.

The reminder of the paper is organized as follows. In Section 2 we address the problem of bin packing with size-increasing fragmentation. Section 3 is devoted to the problem of bin packing with size-preserving fragmentation.

2 Bin Packing with Size-Increasing Fragmentation

In this section we study the variant of bin packing with item fragmentation where fragmentation increases the size of an item. We define the problem similar to the classical bin packing problem. The classical bin packing problem deals with equal-sized (unit capacity) bins and a list of items each of which can fit in every bin. To handle fragmentation we use a discrete version of the problem and add a fragmentation cost function that adds overhead units to each fragment. We proceed to formally define the problem.

Bin Packing with Size-Increasing Fragmentation (BP-SIF): We are given a list of items $L = (a_1, a_2, ..., a_n)$, each with a size $s(a_i) \in \{1, 2, ..., U\}$. The items must be packed into a minimum number of bins, which are all the size of U units. When packing a fragment of an item, one unit of overhead is added to the size of *every* fragment.

Performance Ratio: We use the same definition as is typically used in analyzing the classical problem. For a given list L and algorithm A, let $A(L)$ be the

number of bins used when algorithm A is applied to list L, let $OPT(L)$ denote the optimum number of bins for a packing of L, and let $R_A(L) \equiv A(L)/OPT(L)$. The ***asymptotic worst case performance ratio*** R_A^∞ is defined to be:

$$R_A^\infty \equiv \inf\{r \geq 1 : for some N > 0, R_A(L) \leq r for all L with OPT(L) \geq N\} \quad (1)$$

The bin packing problem is know to be NP-hard in the strong sense [13]. We show that the complexity of BP-SIF is the same.

Claim. BP-SIF is NP-hard in the strong sense.

Proof. We denote by D(BP-SIF) the decision version of BP-SIF and show that it is NP-complete in the strong sense. We do so by reducing the 3-PARTITION problem to a restricted instance of D(BP-SIF). The 3-PARTITION problem (defined formally below) is known to be NP-complete in the strong sense [5].

3-PARTITION: given a list L of $n = 3m$ integers: $w_1, w_2, ..., w_n$ and a bound $B \in Z^+$ such that $B/4 < w_j < B/2$ for $j = 1, ..., n$ and $\sum_{j=1}^{n} w_j = m\,B$, can L be partitioned into m disjoint subsets $S_1, ..., S_m$ such that $\sum_{j \in S_i} w_j = B$ for $i = 1, ..., m$?

We define D(BP-SIF) as follows: given a list of items L, a size $s(a) \in Z^+$ for each item $a \in L$, a positive integer bin capacity U and a positive integer K, is there a feasible packing of L in K bins of size U? Any instance I of 3-PARTITION can be polynomially transformed into an equivalent instance I' of D(BP-SIF) by setting $U = B$ and $K = m$. To realize the two decision problems are equivalent note that, since the total size of all items is equivalent to the total capacity of all bins, in any "yes" instance of D(BP-SIF) all n items are packed without fragmentation and the packing is therefore also valid for 3-PARTITION. Clearly the packing of any "yes" instance of 3-PARTITION is also valid for D(BP-SIF). It follows that the "yes" and "no" instances of the two problems are equivalent. □

When packing a list of n items into m bins, the maximum number of fragmentations possible is $n \cdot m$ (each item is fragmented over all bins). From the definition of the problem it is obvious that a good algorithm should try to perform the minimum number of fragmentations. Therefore we would only like to consider algorithms that do not fragment items unnecessarily.

Definition 1: An algorithm A is said to *prevent unnecessary fragmentation* if it follows the following two rules:

1. No unnecessary fragmentation: An item (or fragment of an item) is fragmented only if it is to be packed into a bin that cannot contain it. In case of fragmentation, the item (or fragment) is divided into two fragments. The first fragment must fill one of the bins. The second fragment is packed according to the packing rules of the algorithm.
2. No unnecessary bins: An item is packed in a new bin only if it cannot fit in any of the open bins used by A.

Algorithms that prevent unnecessary fragmentation have the following property.

Lemma 1. *For any algorithm A that prevents unnecessary fragmentations -* $R_A^\infty \leq \frac{U}{U-2}$, *for every* $U > 2$.

Proof. Assume that the number of bins used by the algorithm is $A(L) = m$. Since A prevents unnecessary fragmentation it can perform at most $m-1$ fragmentations while packing m bins (no fragmentation in the last bin), regardless of the size of the list. Each fragmentation adds two units of overhead at the most. Therefore, in the worst case, $2m-1$ units of overhead are added to the total size of all items. Note that in this case only the last bin may be left unfilled. Assuming the optimal packing does not fragment any item, the number of bins used by it satisfies: $OPT(L) \geq \frac{(m-1)U-2(m-1)+1}{U}$

The asymptotic performance ratio follows:

$$R_A(L) = \frac{A(L)}{OPT(L)} \leq \frac{mU}{(m-1)(U-2)+1} \overset{m\to\infty}{\longrightarrow} R_A^\infty \leq \frac{U-2}{U}$$

□

Remark: For the more general case, where r units of overhead (instead of one) are added to the size of every fragment, it can be shown by similar arguments, that: $R_A^\infty \leq \frac{U}{U-2r}, \quad U > 2r$.

We now have an upper bound on the performance ratio of any algorithm. In the remainder of this section we investigate specific algorithms to find their actual performance ratio. For a given algorithm A we define a version of the algorithm that allows item fragmentation and denote it by A_f. We investigate the worst case performance ratio of the following algorithms: NF_f, NFD_f (NFI_f) and FFD_f (BFD_f).

2.1 Next-Fit with Item Fragmentation - NF_f

The NF_f algorithm is defined similar to the NF algorithm.

Algorithm NF_f - In each stage there is only one open bin. The items are packed, according to their order in the list L, into the open bin. When an item does not fit in the open bin, it is fragmented into two parts. The first part fills the open bin and the bin is closed. The second part is packed into a new bin which becomes the open bin. Offline version of the algorithm sorts the items in decreasing (NFD_f) or increasing (NFI_f) order before packing the list.

The NF_f algorithm is very simple, can be implemented to run in linear time and requires only one open bin (bounded space). However, as we show next, similar to the classical problem, the performance ratio the algorithm achieves is the worst possible.

Theorem 1. *For algorithm* NF_f - $R_{NF_f}^\infty = \frac{U}{U-2}, \quad \forall\, U \geq 6$.

Proof. Lemma 1 provides an upper bound on the performance ratio of the algorithm. We present an example that proves the lower bound. Let us first consider the case where the bin size U is an even number. As a worst case

example we choose the following list L: The first item is of size $U/2$, the next $\left(\frac{U}{2}-2\right)$ items are of size 1. The rest of the list repeats this pattern kU times. The optimal packing avoids fragmentations by packing bins with two items of size $U/2$ or U items of size 1. The total number of bins used is: $OPT(L) = \frac{U}{2}k + \left(\frac{U}{2}-2\right)k = (U-2)\,k$. On the other hand, algorithm NF_f fragments each item of size $U/2$ (except for the first item). Overall $2(kU-1)$ units of overhead are added to the packing and therefore the number of bins used by NF_f is:

$$NF_f(L) = \left\lceil \frac{U}{2}k + 2\frac{kU-1}{U} + \left(\frac{U}{2}-2\right)k \right\rceil = \left\lceil Uk - \frac{2}{U} \right\rceil = Uk. \tag{2}$$

A worst case example for the case where U is an odd number is similar. The first item in L is of size $(U-1)/2$, the next $\left(\frac{U-1}{2}-1\right)$ items are of size 1. The rest of the list repeats this pattern kU times. It is easy to verify that, as in the previous example, $OPT(L) = (U-2)k$, while $NF_f(L) = kU$. □

When the bin size is very small the above proof does not hold. For the values $3 < U \leq 5$ we show in [11] that the worst case asymptotic performance ratio is: $R^{\infty}_{NF_f} = \frac{3}{2}$.

It is interesting to compare the NF_f algorithm to the classic NF algorithm for which the asymptotic worst case performance ratio is: $R^{\infty}_{NF} = \frac{2U}{U+1}, \quad \forall\, U \geq 1$. While the performance ratio of NF is increasing with U the performance ratio of NF_f is decreasing with U. This is intuitive since as the bin size gets larger the effective cost of fragmentation gets smaller.

***NFD_f* and *NFI_f* Algorithms.** Given the poor performance of the NF_f algorithm, one may ask whether sorting the items before applying the NF_f packing, would yield better results. It turns out that algorithms NFD_f and NFI_f are very similar to the NF_f algorithm in their worst case performance. The actual performance ratio of these algorithms depends on the bin size U, but in all cases it is not far from the ratio of NF_f. To avoid dealing with each value of U separately, we first present an example for a general value. Our list of items is made of k items of size $U-2$ and k items of size 2. The optimal packing uses k bins where each bin contains two items, one of each kind. The NFD_f algorithm first packs k items, of size U-2, into k bins. Then k-1 items of size 2 are packed into $\lceil 2\,(k-1)\,/U \rceil$ bins if U is even, or $\lceil 2\,(k-1)\,/\,(U-1) \rceil$ bins if it is odd. The NFI_f algorithm will use the same number of bins, since the only difference is that the items are packed in reverse order. This simple example gives us the following lower bounds:

$$R^{\infty}_{NFD_f} = R^{\infty}_{NFI_f} \geq \frac{U+2}{U}, \quad \text{for any even } U \geq 6 \tag{3}$$

$$R^{\infty}_{NFD_f} = R^{\infty}_{NFI_f} \geq \frac{U+1}{U-1}, \quad \text{for any even } U \geq 5 \tag{4}$$

The above example provides a lower bound. We now demonstrate that in some cases the NFD_f and NFI_f algorithms can perform just as bad as NF_f. The first example is for U=5 and a list $L = \{3, ..., 3, 2, ..., 2\}$, for which $R^{\infty}_{NFD_f} = R^{\infty}_{NFI_f} = R^{\infty}_{NF_f} = \frac{3}{2}$. To see that such examples are not restricted to low values of bin size, consider $U = 32$ and choose a list of $15k$ items of size 10 and $15k$ items of size 6. In the optimal packing the content of each bin is {10, 10, 6, 6}, therefore $OPT(L) = 7.5k$. The total number of bins used by the algorithms is: $NFD_f(L) = NFI_f(L) = 8k$, since all bins (save two) contain two units of overhead. The performance ratio of all three algorithms in this case is $\frac{32}{30}$ which is the worst possible ratio. On the other hand for some values of bin size, NFI_f and NFD_f have a better performance ratio than NF_f. For example when $U = 6$, $R^{\infty}_{NFD_f} = R^{\infty}_{NFI_f} = \frac{4}{3}$, while $R^{\infty}_{NF_f} = \frac{3}{2}$.

2.2 First-Fit Decreasing with Item Fragmentation - FFD_f

We now develop an algorithm based on the FFD heuristic. Let us first describe algorithm FFD_f which packs items from a list L, into a *fixed* number of m bins.
Algorithm FFD_f - First-Fit Decreasing with item fragmentation: The algorithm packs the items in decreasing order. An item is packed into the lowest indexed bin into which it fits. If an item does not fit into any bin it is fragmented. When fragmenting an item the first fragment fills the lowest indexed bin that is as yet not full. If the second fragment can be packed without fragmentation, it is packed into the lowest indexed bin into which it fits, otherwise another fragmentation is performed according to the above rule.
Remark: Other definitions are possible. For example, upon fragmentation we may choose to insert the second fragment back to the list. Another possibility is to first go over the whole list and pack items without fragmentation and only then pack the remaining items into the available free space.

Note that FFD_f may not be able to pack all the items in L. To ensure all items are packed we use the following iterative algorithm:
Algorithm FFD_f **Iterative** ($FFD_f - I$) - The $FFD_f - I$ algorithm tries to pack the list L into a fixed number of m bins. If it fails it increases m by one and tries again. Let $s(L)$ be the sum of all items in L, the first value of m is: $m_1 = \lceil s(L)/U \rceil$, which is the minimum number of bins possible.

The algorithm performs the following steps:

1. Set $m = m_1 = \lceil s\,(L)\,/U \rceil$.
2. Try to pack the list L into m bins using the FFD_f algorithm.
3. If all items were packed stop.
4. Otherwise set $m = m + 1$ and go to step 2.

It is interesting to see if $FFD_f - I$ improves the performance ratio of NF_f. As we shell see the improvement is significant for small values of bin size.

Theorem 2. *The asymptotic worst case performance ratio of the* $FFD_f - I$ *algorithm satisfies:*

(i) $R^{\infty}_{FFD_f-I} \leq \frac{U}{U-1}$ *when* $U \leq 15$

(ii) $\frac{U}{U-1} < R^{\infty}_{FFD_f-I} < \frac{U}{U-2}$ *when* $U \geq 16$

Proof. To prove the theorem we use a property which we call the *border bin property*. We assume $FFD_f - I$ has m bins to pack and number the bins $B_1, ..., B_m$ according to the order they are opened by the algorithm. Looking at the level (used space) of the bins at a certain time during algorithm execution, we say that bin B_j, $j < m$, is a *border bin* at that time if its level satisfies: $L(B_j) \neq L(B_{j+1})$.

Claim. At any time before the first item is fragmented, the packing of $FFD_f - I$ contains at most $2U$ border bins.

Proof. The $FFD_f - I$ algorithm packs the items in decreasing size order. We consider the number of border bins before the first item is fragmented. Denote by $BR(k)$ the number of border bins, after k different sizes of items have been packed by $FFD_f - I$. Clearly $BR(1) \leq 2$ since items of the same size are packed in a similar way. Whenever each additional size is packed the number of border bins may increase by at most two, therefore $BR(k) \leq BR(k-1)+2$. Since there are at most U different sizes, $BR(U) \leq 2U$, and the claim follows. □

We now proceed to prove the theorem for each range separately.

(i) $U \leq 15$: We establish $\frac{U}{U-1}$ as an upper bound on the asymptotic performance ratio. It is clear that as long as the number of bins with two units of overhead is small, i.e., O(1), the asymptotic performance ratio cannot exceed $\frac{U}{U-1}$ (the proof is similar to that of Lemma 1). We make the following observation:

Claim. The final packing of the $FFD_f - I$ algorithm **cannot** contain more than O(1) bins with 2 units of overhead, if one of the following conditions is met:

1. Before the first fragmentation occurs, the free space in the bins is 2 or less.
2. The fragmented items are of size 4 or less.

Proof. Omitted.

It is easy to verify that the conditions set by the above claim cannot be extended, since fragmenting items of size 5 over bins of size 3 results in one third of the bins containing 2 units of overhead. Therefore, in order to get a significant number of bins with 2 units of overhead, items of size 5 or more must be fragmented. This means that the list should contain items of size 6 or more. Note that if $FFD_f - I$ fragments items of sizes $s(a_i) \geq \frac{U}{2}$, they are also fragmented by the optimal packing. We conclude that in order to create a difference of more than one overhead unit, between the optimal packing and the packing of $FFD_f - I$, two items of size $s(a_i) \geq 6$ must be packed in a bin and leave a free space of at least 3. To do so, the bin size must satisfy $U \geq 15$. However, in the case of U=15, items of size 5 are packed without fragmentation

and therefore the value $\frac{U}{U-1}$ is an upper bound on the asymptotic performance ratio for $U \leq 15$.

(ii) $U \geq 16$: We first prove the lower bound and then the upper bound.

Claim. For the $FFD_f - I$ algorithm - $\frac{U}{U-1} < R^{\infty}_{FFD_f - I}$.

Proof. For each value $U \geq 16$, there is a list for which the performance ratio exceeds $\frac{U}{U-1}$. We present here an example only for the case where U is even, an example for odd values of U is similar. Assume $U = 2u$ and choose a list L with k items of size $u-2$, then k items of size $u-3$ and finally k items of size 5. The optimal packing fills k bins each with one item of each size, $OPT(L) = k$. When $FFD_f - I$ is applied to L the first k bins contain all items except for the last $k/4$ items of size 5. In the best case (U is a multiple of 5) no more overhead is produced and $\frac{5k}{4U}$ more bins are used. The total number of bins required by the algorithm is: $FFD_f - I(L) \geq (k + 5k/4U)$. The performance ratio in this case is:

$$R^{\infty}_{FFD_f - I} \geq \frac{4U+5}{4U} > \frac{U}{U-1}, \quad \forall\, even\ U \geq 16. \tag{5}$$

□

We now turn to the upper bound on the performance ratio.

Claim. For the $FFD_f - I$ algorithm - $R^{\infty}_{FFD_f - I} < \frac{U}{U-2}$.

Proof. In order to prove the claim we show that the algorithm cannot produce a packing where each bin (maybe except for a negligible number) contains two units of overhead. The *border bins property* implies that just before the first item is fragmented the bins are arranged in long sequences where the free space in all bins is equal. The items are also packed in long sequences. Let us assume the free space in a sequence is x and the size of the items packed is y. Obviously $x < y$ otherwise the items are not fragmented. Note that when an item is fragmented over more than two bins, only the first and last bins can contain two units of overhead. Therefore it is enough to consider only the case where an item is fragmented over two bins. Assume that item number k is fragmented over two bins such that bin number 1 contains a fragment of size α and bin number 2 contains a fragment of size $y - \alpha$. The next item (number $k+1$) is also fragmented and a fragment of size $x - y + \alpha - 2$ is packed in bin number 2. In order to create a repeated cycle of fragmentations the size of the first fragment of each item must be equal, that is $\alpha = x - y + \alpha - 2$. This is true for $y = x - 2$, but for that value the items are not fragmented in the first place. Since we can not create a repeated cycle of equal size fragmentations, the size of the first fragment packed in a bin increases, until it reaches the size of the bin, in which case only one unit of overhead is packed in the bin. This means that at least one of every y bins contains only one unit of overhead. Since $y \leq U/2$ we can establish that:

$$R^{\infty}_{FFD_f - I} \leq \frac{U}{U - 2 + \frac{2}{U}} < \frac{U}{U-2} \tag{6}$$

This concludes the proof of Theorem 2. □

The improvement of $FFD_f - I$ over NF_f is significant for low values of bin size ($U \leq 15$), which are the most meaningful values (since the performance ratio is decreasing with the bin size). Moreover, $FFD_f - I$ is superior for any value of U. We make the following remarks:

- For each of the following values: $U \in \{7, 9, 10, 11, 13, 14, 15\}$, it is possible to find an example where the ratio is $\frac{U}{U-1}$, hence $R^{\infty}_{FFD_f-I} = \frac{U}{U-1}$, [11].
- When $U \leq 5$, $R^{\infty}_{FFD_f-I} = 1$, however when $U \in \{6, 8\}$ $R^{\infty}_{FFD_f-I} > 1$. This is interesting, since for the classical problem the performance ratio of FFD is: $R^{\infty}_{FFD} = 1$.
- We may define algorithm $BFD_f - I$ in a similar way to $FFD_f - I$, only this algorithm is based on the Best-Fit Decreasing heuristic. We can show that the $BFD_f - I$ algorithm has the same performance ratio as $FFD_f - I$ [11].

3 Bin Packing with Size-Preserving Fragmentation

In this section we study a different fragmentation cost function in which fragmenting an item does not increase its size. Instead, we assume that packing an item is associated with a cost and fragmentation increases this cost. The cost of packing an item (or fragmenting it) depends on the application. As an example, take the scheduling problem but assume that fragmenting a datagram does not increase its size, because the format of the datagram already includes the fragmentation fields. On the other hand fragmentation requires additional resources from the system (CPU, memory) and takes longer to process. In other applications, such as in stock-cutting problems, it may simply cost to fragment an item (cut a piece of pipe for example) or put it back together. We proceed to formally define the problem.

Bin Packing with Size-Preserving Fragmentation (BP-SPF): We are given a list of n items $L = (a_1, a_2, ..., a_n)$, each with a size $s(a_i) \in \{1, 2, ..., U\}$ and a cost $c(a_i) \in Z^+$. The items must be packed into m identical bins, of size U. It is possible to fragment any item, in which case one unit is added to its cost but does not change its size. The goal is to minimize the total cost.

Denote by $s(L)$ and $c(L)$ the total size and cost of all items, respectively. To ensure all items can be packed, we assume $s(L) \leq mU$.

Performance: There are several ways to evaluate the performance of an algorithm for the problem. We observe that since the cost of fragmentation is not related to the size or cost of an item, the additional cost of an algorithm depends only on the number of fragmentations it performs. We therefore chose to evaluate the performance of an algorithm by its *overhead*. For a given list L and algorithm A, let $c(A, L)$ be the total cost of algorithm A, let $c(OPT, L)$ denote the optimal (minimal) cost and define the overhead of A as: $OH_A(L) \equiv c\,(A, L) - c\,(OPT, L)$. For the case of $OPT(L) = m$, we define the worst case overhead of algorithm A, OH^m_A, as:

$$OH^m_A \equiv \inf\{h : \ OH_A(L) \leq h \quad \text{for all } L \text{ with } OPT(L) = m\}. \tag{7}$$

We first show that the complexity of BP-SPF is similar to that of BP-SIF.

Claim. BP-SPF is NP-hard in the strong sense.

Proof. The proof is similar to that of BP-SIF and is omitted from this version.

We consider only algorithms that prevent unnecessary fragmentation (see Definition 1). Such algorithms have the following property.

Lemma 2. *For any algorithm A, that prevents unnecessary fragmentations -* $OH_A^m \leq m-1$.

Proof. Since A prevents unnecessary fragmentations, it may perform at most $m-1$ fragmentations when packing m bins. The maximum cost of algorithm A is therefore: $c(A,L) = c(L) + (m-1)$. Clearly $c(OPT,L) \geq c(L)$, which means that for any list L: $OH_A(L) \leq m-1$. □

We now examine the performance of the NF_f and FFD_f algorithms (defined in subsections 2.1 and 2.2, respectively). We show that the performance of the NF_f algorithm is the worst possible while FFD_f performs better.

Theorem 3. *The overhead of algorithm NF_f for every $m \geq 2$ is -* $OH^m_{NF_f} = m-1, \quad \forall U \geq 2$.

Proof. Lemma 2 provides an upper bound. As a worst case example choose a list of items with one item of size $U-1$ followed by $m-1$ items of size U. □

We now turn to the FFD_f algorithm. We expect FFD_f to perform better than NF_f and this is indeed true when the bin size U is small. However, we show that if the bin size is not bounded, the worst case overhead of FFD_f is the maximum possible, that is, there exist a list L for which, for any value of m, $c(FFD_f, L) - c(OPT, L) = m-1$.

Claim. For the FFD_f algorithm, for every $m \geq 2$ - $OH^m_{FFD_f} = m-1$.

Proof. We choose a bin size satisfying: $U > 2m+16$. The list, L, is made of k repetitions of the following set: $L' = \{U/2+2,\ U/2+1,\ U/4+2,\ U/4+1,\ U/4-3,\ U/4-3\}$. Two bins of size U are needed to pack L', therefore $m = 2k$ bins are needed to pack L. The optimal packing causes no fragmentations. The FFD_f algorithm packs the items in the following way: The first $5k$ items are packed without fragmentations. The free space in the first k bins is $U/4-4$, in the remaining k bins the free space is 1. Since the free space in all bins is smaller than the size of the items, the items are fragmented. Each item except the last is fragmented over two bins. When packing the last item, bins B_1,...B_{k-1} are full, bin B_k has free space of $\frac{U}{4}-k-3$ and each bin B_{k+1},...,B_{2k} has free space of 1. As a result, the last item is fragmented over the remaining free space, causing k more fragmentations. The total number of fragmentations is $m-1$ and the overhead is therefore $OH^m_{FFD_f} = m-1$. □

To hold for every value of m the above claim requires an unbounded bin size. Since we are mainly interested in asymptotic behavior, i.e., $U \ll m$, this is clearly not a practical assumption. For the more reasonable cases where $m > U$ the overhead of algorithm FFD_f is less than $m-1$.

Claim. For any bin size U there is $N > 0$ for which the overhead of FFD_f satisfies - $OH^m_{FFD_f} < m - 1, \quad \forall\, N < m$.

Proof. We show that if $U \ll m$ then $c\,(A, L) < c\,(L) + (m - 1)$. Recall that when analyzing the $FFD_f - I$ algorithm (subsection 2.2) we proved the border bin property. The property tells us that before the first item is fragmented the bins are ordered in long sequences of equal content bins. Note that by increasing m while keeping U constant, we can create sequences of any length. Consider the moment before FFD_f fragments the first item. Let us assume the free space in a sequence is x and the size of the items packed is y, where $x < y$. Clearly at least one out of every $x \cdot y$ bins is closed without fragmentation. The overhead is therefore always smaller than $m - 1$. □

References

1. E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin-packing: An updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, Algorithm Design for Computer System Design, pp. 49-106. Springer-Verlag, Wien, 1984.
2. E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In D. Hochbaum (ed), PSW publishing, Boston. Approximation Algorithms for NP-Hard Problems, pp. 46-93. 1996.
3. D. K. Friesen and F. S. Kuhl. Analysis of a hybrid algorithm for packing unequal bins. SIAM J. of Computing, vol. 17, pp. 23-40, 1988.
4. D. K. Friesen and M. A. Langston. Analysis of a compound bin packing algorithm. SIAM J. Disc. Math, vol. 4, pp.61-79, 1991.
5. M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Co., San Francisco, 1979.
6. D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. SIAM J. of Computing, 3:299-325, 1974.
7. D. S. Johnson. Fast algorithms for bin packing. Journal of computer and system Science, vol. 8, pp. 272-314, 1974.
8. N. Karmarkar and R. M. Karp. An efficient approximation scheme for the one-dimensional bin packing problem. In Proc. 23rd Ann. Symp. on Foundations of Computer Science, pp. 312-320, 1982.
9. C. C. Lee and D. T. Lee. A simple on-line packing algorithm. J. ACM, vol. 32, pp. 562-572, 1985.
10. Mandal-CA, Chakrabarti-PP and Ghose-S. Complexity of fragmentable object bin packing and an application. Computers and Mathematics with Applications, vol.35, no.11, pp. 91-7, 1998.
11. Nir Menakerman and Raphael Rom. Bin packing problems with item fragmentation. Technical report, EE publication CITT #342, April 2001.
12. Multimedia Cable Network System Ltd., "Data-Over-Cable Service Interface Specifications - Radio Frequency Interface Specification", July 2000.
13. J. D. Ullman. Complexity of sequencing Problems. Computer and Job-Shop Scheduling Theory. Wiley and Sons. 1976.

Practical Approximation Algorithms for Separable Packing Linear Programs*

Feodor F. Dragan[1], Andrew B. Kahng[2], Ion I. Măndoiu[3], Sudhakar Muddu[4], and Alexander Zelikovsky[5]

[1] Department of Computer Science, Kent State University, Kent, OH 44242
dragan@cs.kent.edu
[2] Departments of Computer Science and Engineering, and of Electrical and Computer Engineering, UC San Diego, La Jolla, CA 92093-0114
abk@cs.ucsd.edu
[3] Department of Computer Science, UC Los Angeles, Los Angeles, CA 90095-1596
mandoiu@cc.gatech.edu
[4] Sanera Systems, Inc., Santa Clara, CA
muddu@sanera.net
[5] Department of Computer Science, Georgia State University, Atlanta, GA 30303
alexz@cs.gsu.edu

Abstract. We describe fully polynomial time approximation schemes for generalized multicommodity flow problems arising in VLSI applications such as Global Routing via Buffer Blocks (GRBB). We extend Fleischer's improvement [7] of Garg and Könemann [8] fully polynomial time approximation scheme for edge capacitated multicommodity flows to multiterminal multicommodity flows in graphs with capacities on vertices and subsets of vertices. In addition, our problem formulations observe upper bounds and parity constraints on the number of vertices on any source-to-sink path. Unlike previous works on the GRBB problem [5,17], our algorithms can take into account (i) multiterminal nets, (ii) simultaneous buffered routing and compaction, and (iii) buffer libraries. Our method outperforms existing algorithms for the problem and has been validated on top-level layouts extracted from a recent high-end microprocessor design.

1 Introduction

In this paper, we address the problem of how to perform buffering of global nets *given an existing buffer block plan*. We give integer linear program (ILP) formulations of the basic Global Routing via Buffer Blocks (GRBB) problem and its extensions to (i) multiterminal nets, (ii) simultaneous buffered routing and compaction, and (iii) buffer libraries. The fractional relaxations of these ILP's are *separable packing LP's* (SP LP) which are multiterminal multicommodity flows in graphs with capacities on vertices and subsets of vertices.

* This work was partially supported by Cadence Design Systems, Inc., the MARCO Gigascale Silicon Research Center and NSF Grant CCR-9988331.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 325-337, 2001.

The main contribution of this paper is a practical algorithm for the GRBB problem and its extensions based on a fully polynomial time approximation scheme (FPTAS) for solving SP LPs. Prior to our work, heuristics based on solving fractional relaxations followed by randomized rounding have been applied to VLSI global routing [12,16,2,9,1]. As noted in [11], the applicability of this approach is limited to problem instances of relatively small size by the prohibitive cost of solving exactly the fractional relaxation. We avoid this limitation by giving an FPTAS for SP LP's based on results in [8,7]. Computational experience with industrial benchmarks shows that our approach is practical and outperforms existing algorithms.

The rest of the paper is organized as follows. In Section 3 we formulate the GRBB problem and its extensions as integer linear programs. The fractional relaxation of these ILPs is a special type of packing LP which we refer to as separable packing LP. In Sections 4 we give a practical approximation algorithm, obtained by extending the ideas of Fleischer [7] for separable packing LPs; the details of the key subroutine for finding minimum-weight feasible Steiner trees are given in Section 5; the details of randomized rounding algorithms are in Section 6. In Section 7 we describe implementations of several GRBB heuristics and give the results of an experimental comparison of these heuristics on industrial test cases.

2 Global Buffering via Buffer Blocks

Process scaling in VLSI leads to an increasingly dominant effect of interconnect on high-end chip performance. Each top-level global net must undergo repeater or buffer (inverter) insertion to maintain signal integrity and reasonable signal delay [4]. It is estimated that up to 10^6 repeaters will be needed for the next generation on-chip interconnect. To isolate repeaters from circuit block implementations, a buffer block methodology is becoming increasingly popular. Two recent works by Cong, Kong and Pan [5] and Tang and Wong [17] give algorithms to solve the *buffer block planning* problem. Their buffer block planning formulation is roughly stated as follows: Given a placement of circuit blocks, and a set of 2-pin connections with *feasible regions* for buffer insertion, plan the location of *buffer blocks* within the available free space so as to route a maximum number of connections.

In this paper we address the problem of maximizing the number of routed nets for given buffer block locations and capacities, informally defined as follows.

Given:

- a planar region with rectangular obstacles;
- a set of nets in the region, each net having:
  - a non-negative importance (criticality) coefficient;
  - a single source and multiple sinks;

- for each sink:
  - a parity requirement and an upper-bound on the number of buffers on the path connecting it to the source;
- a set of buffer blocks, each with given capacity; and
- an interval $[L, U]$ specifying lower and upper bounds on the distance between buffers.

Global Routing via Buffer Blocks (GRBB) Problem: route a subset of the given nets, with maximum total importance, such that:

- the distance between the source of a route and its first repeater, between any two consecutive repeaters, respectively between the last repeater on a route and the route's sink, are all between L and U;
- the number of routing trees passing through any given buffer block does not exceed the block's capacity;
- the number of buffers on each source-sink path does not exceed the given upper bound and has the required parity; to meet the parity constraint two buffers of the same block can be used.

We also address the following extensions of the basic GRBB problem:

- **GRBB with Set Capacity Constraints.** The basic GRBB problem assumes predetermined capacities for all buffer blocks. In practice buffer blocks are placed in the space available after placing circuit blocks, and some of the circuit blocks can still be moved within certain limits (Figure 1). The *GRBB problem with set capacity constraints* captures this freedom by allowing constraints on the total capacity of arbitrary *sets* of buffer blocks.

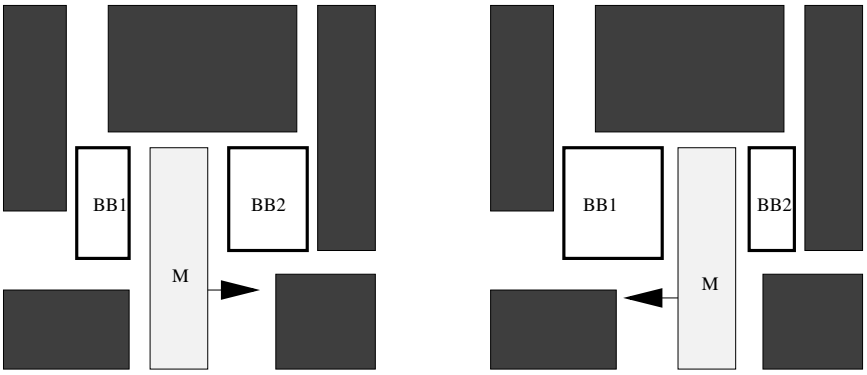


Fig. 1. Two buffer blocks BB1 and BB2 that share capacity: if the circuit block M moves to the right, then the capacity of buffer block BB1 is increasing while the capacity of buffer block BB2 is decreasing. In this example it is the sum of capacities of BB1 and BB2, rather than their individual capacities, that is bounded.

– **GRBB with Buffer Library.** To achieve better use of area and power resources, multiple buffer types can be used. The *GRBB problem with buffer library* optimally distributes the available buffer block capacity between given buffer types and simultaneously finds optimum buffered routings.

3 Integer Linear Program Formulations

Throughout this paper we let $N_k = (s_k; t_k^1, \ldots, t_k^{q_k})$, $k = 1, \ldots, K$, denote the nets to be routed; s_k is the *source*, and $t_k^1, \ldots, t_k^{q_k}$ are the *sinks* of net N_k. We denote by $g_k \geq 1$ the importance (criticality) coefficient of net N_k, and by $a_k^i \in \{\text{even, odd}\}$ and $l_k^i \geq 0$ the prescribed *parity*, respectively *upper bound*, on the number of buffers on the path between source s_k and sink t_k^i. We also let $S = \{s_1, \ldots, s_K\}$ and $S' = \{t_1^1, \ldots, t_1^{q_1}, \ldots, t_K^1, \ldots, t_K^{q_K}\}$ denote the set of sources, respectively of sinks, and $R = \{r_1, \ldots, r_n\}$ denote the given set of *buffer blocks*. For each buffer block r_i, we let $c(r_i)$ denote its *capacity*, i.e., the maximum number of buffers that can be inserted in r_i.

A *routing graph* for nets N_k, $k = 1, \ldots, K$, is an undirected graph $G = (V, E)$ such that $S \cup S' \subseteq V$. The set of vertices of G other than sources and sinks, $V \setminus (S \cup S')$, is denoted by V'. All vertices in a routing graph are associated to locations on the chip, including vertices of V' which are associated with buffer block locations. We require that the rectilinear distance with obstacles between two vertices connected by an edge in the routing graph be either between L and U or 0 (this last case corresponds to using two buffers in the same buffer block). Thus, inserting a buffer at each Steiner point ensures that every Steiner tree in the routing graph satisfies the given L/U bounds. A *feasible Steiner tree* for net N_k is a Steiner tree T_k connecting terminals $s_k, t_k^1, \ldots, t_k^{q_k}$ such that, for every $i = 1, \ldots, q_k$, the path of T_k connecting s_k to t_k^i has length at most l_k^i and parity a_k^i. We denote the set of all feasible Steiner trees for net N_k by $\mathcal{T}_k$, and let $\mathcal{T} = \bigcup_{k=1}^{K} \mathcal{T}_k$.

For the GRBB problem, the routing graph $G = (V, E)$ has $V = S \cup S' \cup \{r', r'' \mid r \in R\}$ (there are two vertices corresponding to each buffer block to allow for feasible Steiner trees that meet the parity constraints by using two buffers in the same buffer block) and $E = \{(r', r'') \mid r \in R\} \cup \{(x, y) \mid x, y \in V,\ L \leq d(x, y) \leq U\}$, where, $d(x, y)$ is the rectilinear distance with obstacles between points x and y. Given importance coefficients $g_k = g(N_k)$ for each net N_k, let $g(T) = g_k$ for each tree $T \in \mathcal{T}_k$, $k = 1, \ldots, K$. The GRBB problem is then equivalent to the following integer linear program:

$$
\begin{array}{lll}
\textbf{maximize} & \sum_{T \in \mathcal{T}} g(T) f_T & \text{(GRBB ILP)} \\
\textbf{subject to} & & \\
 & \sum_{T \in \mathcal{T}} \pi_T(v) f_T \leq 1, & \forall v \in S \cup S' \\
 & \sum_{T \in \mathcal{T}} (\pi_T(r') + \pi_T(r'')) f_T \leq c(r), & \forall r \in R \\
 & f_T \in \{0, 1\}, & \forall T \in \mathcal{T}
\end{array}
$$

where $\pi_T(v)$ is 1 if $v \in T$ and 0 otherwise.

The GRBB ILP, as well as the ILP formulations for GRBB with set constraints and buffer library (which we omit from this extended abstract) are captured by the following common generalization, referred to as the *separable packing ILP* (SP ILP):

$$\begin{array}{ll} \textbf{maximize} & \sum_{T\in\mathcal{T}} g(T) f_T \qquad \text{(SP ILP)} \\ \textbf{subject to} & \\ & \sum_{T\in\mathcal{T}} (\sum_{v\in X} \pi_T(v) s(v))\, f_T \le c(X), \quad \forall X \in \mathcal{V} \\ & f_T \in \{0,1\}, \quad \forall T \in \mathcal{T} \end{array}$$

for given

- arbitrary sets $\mathcal{T}_k$ of Steiner trees for each net N_k;
- family $\mathcal{V}$ of subsets of V such that $\{v\} \in \mathcal{V}$ for every $v \in S \cup S'$;
- "size" function $s : V \to R_+$ such that $s(v) = 1$ for every $v \in S \cup S'$; and
- "set-capacity" function $c : \mathcal{V} \to Z_+$ such that $c(\{v\}) = 1$ for every $v \in S \cup S'$.

Our two-step approach to the GRBB problem and its extensions is to first solve the fractional relaxations obtained by replacing integrality constraints $f_T \in \{0,1\}$ with $f_T \ge 0$, and then use randomized rounding to get integer solutions. In next section we give an algorithm for approximating the fractional relaxation of the SP ILP. The algorithm relies on a subroutine for finding minimum weight feasible Steiner trees, the details of this subroutine are given in Section 5.

4 Approximating the SP ILP Relaxation

The fractional relaxation of the SP ILP can be solved exactly in polynomial time using, e.g., the ellipsoid algorithm. However, exact algorithms are highly impractical. The SP LP can be efficiently approximated within any desired accuracy using Garg and Könemann's approximation scheme for packing LPs [8]. The main step of their algorithm is computing the minimum weight column of the LP. For the special case of edge-capacitated multicommodity flow LPs, Fleischer [7] gave a significantly faster algorithm by computing in each step the minimum weight column only among columns corresponding to a single commodity. Below we generalize Fleisher's idea to separable packing LPs by partitioning the columns into groups corresponding to the nets.

4.1 The Algorithm

Our algorithm simultaneously finds feasible solutions to the SP LP and its dual. The dual LP asks for an assignment of non-negative weights $w(X)$ to every $X \in \mathcal{V}$ such that the weight of every tree $T \in \mathcal{T}$ is at least 1, where the weight of T is defined by $weight(T) = \frac{1}{g(T)} \sum_{X\in\mathcal{V}} \pi_T(X) w(X)$ and $\pi_T(X) = \sum_{v\in X} \pi_T(v) s(v)$:

Input: Nets $N_1, \ldots, N_K$, coefficients $g_1, \ldots, g_K$, routing graph $G = (V, E)$, family $\mathcal{V}$ of subsets of V, capacities $c(X)$, $X \in \mathcal{V}$, and weights $s(v)$, $v \in V$
Output: SP LP solution f_T, $T \in \mathcal{T}$

For every $T \in \mathcal{T}$, $f_T \leftarrow 0$
For every $X \in \mathcal{V}$, $w(X) \leftarrow \delta$
$\bar{\alpha} \leftarrow \delta / \Gamma$
For $i = 1$ to $t = \left\lfloor \log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta} \right\rfloor$ do
 For $k = 1$ to K do
  Find a minimum weight feasible Steiner tree T in $\mathcal{T}_k$
  While $weight(T) < min\{1, (1+\epsilon)\bar{\alpha}\}$ do
   $f_T \leftarrow f_T + 1$
   For all $X \in \mathcal{V}$, $w(X) \leftarrow w(X)(1 + \epsilon \pi_T(X)/c(X))$
   Find a minimum weight feasible Steiner tree T in $\mathcal{T}_k$
  End while
 End for on k
 $\bar{\alpha} \leftarrow (1+\epsilon)\bar{\alpha}$
End for on i
For every $T \in \mathcal{T}$, $f_T \leftarrow \frac{f_T}{\log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta}}$
Output f_T, $T \in \mathcal{T}$

Fig. 2. The algorithm for finding approximate solutions to the SP LP.

$$\begin{array}{lll} \textbf{maximize} & \sum_{X \in \mathcal{V}} w(X)c(X) & \text{(SP LP Dual)} \\ \textbf{subject to} & & \\ & \frac{1}{g(T)} \sum_{X \in \mathcal{V}} \pi_T(X) w(X) \geq 1, & \forall T \in \mathcal{T} \\ & w(X) \geq 0, & \forall X \in \mathcal{V} \end{array}$$

In the following we assume that $\min\{g_k : k = 1, \ldots, K\} = 1$ (this can be easily achieved by scaling) and denote $\max\{g_k : k = 1, \ldots, K\}$ by Γ.

The algorithm (Figure 2) starts with weights $w(X) = \delta$ for every $X \in \mathcal{V}$, where δ is an appropriately chosen constant, and with a SP LP solution $f \equiv 0$. While there is a feasible tree whose weight is less than 1, the algorithm selects such a tree T and increments f_T by 1. This increase will likely violate the capacity constraints for some of the sets in $\mathcal{V}$; feasibility is achieved at the end of the algorithm by uniformly scaling down all f_T's. Whenever f_T is incremented, the algorithm also updates each weight $w(X)$ by multiplying it with $(1 + \epsilon \pi_T(X)/c(X))$, for a fixed ϵ.

According to the Garg and Könemann's approximation algorithm [8] each iteration must increment the variable f_T corresponding to a tree with minimum weight among all trees in $\mathcal{T}$. Finding this tree essentially requires K minimum-weight feasible Steiner tree computations, one for each net N_k. We reduce the total number of minimum-weight feasible Steiner tree computations during the

algorithm by extending a speed-up idea due to Fleischer [7]. Instead of always finding the minimum-weight tree in $\mathcal{T}$, the idea is to settle for trees with weight within a factor of $(1+\epsilon)$ of the minimum. As shown in next section, the faster algorithm still leads to an approximation guarantee similar to that of Garg and Könemann.

4.2 Runtime and Performance Analysis

In each iteration the algorithm cycles through all nets. For each net, the algorithm repeatedly computes minimum-weight feasible Steiner tree until the weight becomes larger than $(1+\epsilon)$ times a lower-bound $\bar{\alpha}$ on the overall minimum weight, $\min\{weight(T) : T \in \mathcal{T}\}$. The lower-bound $\bar{\alpha}$ is initially set to δ/Γ, and then multiplied by a factor of $(1+\epsilon)$ from one iteration to another (note that no tree in $\mathcal{T}$ has weight smaller than $(1+\epsilon)\bar{\alpha}$ at the end of an iteration, so $(1+\epsilon)\bar{\alpha}$ is a valid lower-bound for the next iteration).

The scheme used for updating $\bar{\alpha}$ fully determines the number of iterations in the outer loop of the algorithm. Since $\bar{\alpha} = \delta/\Gamma$ in the first iteration and at most $(1+\epsilon)$ in the last one, it follows that the number of iterations is $\left\lfloor \log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta} \right\rfloor$. The following lemma gives an upper-bound on the runtime of the algorithm.

Lemma 1. *Overall, the algorithm in Figure 2 requires* $O\left(K \log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta}\right)$ *minimum-weight feasible Steiner tree computations.*

Proof. First, note that the number of minimum-weight feasible Steiner tree computations that do not contribute to the final fractional solution is $K \left\lfloor \log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta} \right\rfloor$. Indeed, in each iteration, and for each net N_k, there is exactly one minimum-weight feasible Steiner tree computation revealing that $min_{T \in \mathcal{T}_k} weight(T) \geq (1+\epsilon)\bar{\alpha}$, all other computations trigger the incrementation of some f_T.

We claim that the number of minimum-weight Steiner trees that lead to variable incrementations is at most $K \log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta}$. To see this, note that the weight of the set $\{s_k\} \in \mathcal{V}$ is updated whenever a variable f_T, $T \in \mathcal{T}_k$, is incremented. Moreover, $w(\{s_k\})$ is last updated when incrementing f_T for a tree $T \in \mathcal{T}_k$ of weight less than one. Thus, before the last update, $w(\{s_k\}) \leq \Gamma \cdot weight(T) < \Gamma$. Since $\pi_T(\{s_k\}) = c(\{s_k\}) = 1$, the weight of $\{s_k\}$ is multiplied by a factor of $1+\epsilon$ in each update, including the last one. This implies that the final value of $w(\{s_k\})$ is at most $(1+\epsilon)\Gamma$. Recalling that $w(\{s_k\})$ is initially set to δ, this gives that the number of updates to $w(\{s_k\})$ is at most $\log_{1+\epsilon} \frac{(1+\epsilon)\Gamma}{\delta}$. The lemma follows by summing this upper-bound over all nets. □

We now show that, for an appropriate value of the parameter δ, the algorithm finds a feasible solution close to optimum.

Theorem 1. *For every $\epsilon < 0.15$, the algorithm in Figure 2 computes a feasible solution to the SP LP within a factor of $1/(1+4\epsilon)$ of optimum by choosing $\delta = (1+\epsilon)\Gamma((1+\epsilon)L\Gamma)^{-\frac{1}{\epsilon}}$; the runtime of the algorithm for this value of δ is $O\left(\frac{1}{\epsilon^2}K(\log L + \log\Gamma)T_{tree}\right)$. Here, L is the maximum number of vertices in a feasible tree, and T_{tree} is the time required to compute the minimum weight feasible Steiner tree for a net.*

Proof. Our proof is an adaptation of the proofs of Garg and Könemann [8] and Fleischer [7]. We omit the proof that the solution found by the algorithm is feasible. To establish the approximation guarantee, we show that the solution computed by the algorithm is within a factor of $1/(1+4\epsilon)$ of the optimum objective value, β, of the dual LP. Let $\alpha(w)$ be the weight of a minimum weight tree from $\mathcal{T}$ with respect to weight function $w : \mathcal{V} \to R_+$, and let $D(w) = \sum_{X\in\mathcal{V}} w(X)c(X)$. A standard scaling argument shows that the dual LP is equivalent to finding a weight function w such that $D(w)/\alpha(w)$ is minimum, and that $\beta = \min_w\{D(w)/\alpha(w)\}$.

For every $X \in \mathcal{V}$, let $w_i(X)$ be the weight of set X at the end of the ith iteration and $w_0(X) = \delta$ be the initial weight of set X. For brevity, we will denote $\alpha(w_i)$ and $D(w_i)$ by $\alpha(i)$ and $D(i)$, respectively. Furthermore, let f_T^i be the value of f_T at the end of ith iteration, and $h_i = \sum_{T\in\mathcal{T}} g(T)f_T^i$ be the objective value of the SP LP at the end of this iteration.

When the algorithm increments f_T by one unit, each weight $w(X)$ is increased by $(\epsilon\pi_T(X)w(X)/c(X)$. Thus, the incrementation of f_T increases $D(w)$ by

$$\epsilon \sum_{X\in\mathcal{V}} \pi_T(X)w(X) = \epsilon\ weight(T)g(T)$$

If this update takes place in the ith iteration, then $weight(T) \le (1+\epsilon)\alpha(i-1)$. Adding this over all f_T's incremented in ith iteration gives

$$D(i) - D(i-1) \le \epsilon(1+\epsilon)\alpha(i-1)(h_i - h_{i-1})$$

which implies that

$$D(i) - D(0) \le \epsilon(1+\epsilon)\sum_{j=1}^{i} \alpha(j-1)(h_j - h_{j-1})$$

Consider the weight function $w_i - w_0$, and notice that $D(w_i - w_0) = D(i) - D(0)$. Since the minimum weight tree w.r.t. weight function $w_i - w_0$ has a weight of at most $\alpha(w_i - w_0) + L\delta$ w.r.t. w_i, $\alpha(i) \le \alpha(w_i - w_0) + L\delta$. Hence, if $\alpha(i) - L\delta > 0$, then

$$\beta \le \frac{D(w_i - w_0)}{\alpha(w_i - w_0)} \le \frac{D(i) - D(0)}{\alpha(i) - L\delta} \le \frac{\epsilon(1+\epsilon)\sum_{j=1}^{i}\alpha(j-1)(h_j - h_{j-1})}{\alpha(i) - L\delta}$$

Thus, in any case (when $\alpha(i) - L\delta \le 0$ this follows trivially) we have

$$\alpha(i) \le L\delta + \frac{\epsilon(1+\epsilon)}{\beta}\sum_{j=1}^{i}\alpha(j-1)(h_j - h_{j-1})$$

Note that, for each fixed i, the right-hand side of last inequality is maximized by setting $\alpha(j)$ to its maximum possible value, say $\alpha'(j)$, for every $0 \leq j < i$. Then, the maximum value of $\alpha(i)$ is

$$\begin{aligned}\alpha'(i) &= L\delta \ + \ \frac{\epsilon(1+\epsilon)}{\beta}\sum_{j=1}^{i-1}\alpha'(j-1)(h_j - h_{j-1}) \ + \ \frac{\epsilon(1+\epsilon)}{\beta}\alpha'(i-1)(h_i - h_{i-1}) \\ &= \alpha'(i-1)\left(1 \ + \ \frac{\epsilon(1+\epsilon)}{\beta}(h_i - h_{i-1})\right) \\ &\leq \alpha'(i-1)e^{\frac{\epsilon(1+\epsilon)}{\beta}(h_i - h_{i-1})}\end{aligned}$$

where the last inequality uses that $1 + x \leq e^x$ for every $x \geq 0$. Using that $\alpha'(0) = L\delta$, this gives

$$\alpha(i) \leq L\delta e^{\frac{\epsilon(1+\epsilon)}{\beta}h_i}$$

Let t be the last iteration of the algorithm. Since $\alpha(t) \geq 1$,

$$1 \leq L\delta e^{\frac{\epsilon(1+\epsilon)}{\beta}h_t}$$

and thus

$$\frac{\beta}{h_t} \leq \frac{\epsilon(1+\epsilon)}{\ln(L\delta)^{-1}}$$

Let $\gamma = \frac{\beta}{h_t}\log_{1+\epsilon}\frac{(1+\epsilon)\Gamma}{\delta}$ be the ratio between the optimum dual objective value and the objective value of the SP LP solution produced by the algorithm. By substituting the previous bound on β/h_t we obtain

$$\gamma \leq \frac{\epsilon(1+\epsilon)\log_{1+\epsilon}\frac{(1+\epsilon)\Gamma}{\delta}}{\ln(L\delta)^{-1}} = \frac{\epsilon(1+\epsilon)\ln\frac{(1+\epsilon)\Gamma}{\delta}}{\ln(1+\epsilon)\ln(L\delta)^{-1}}$$

For $\delta = (1+\epsilon)\Gamma((1+\epsilon)L\Gamma)^{-\frac{1}{\epsilon}}$,

$$\frac{\ln\frac{(1+\epsilon)\Gamma}{\delta}}{\ln(L\delta)^{-1}} = \frac{\ln\left((1+\epsilon)L\Gamma\right)^{\frac{1}{\epsilon}}}{\ln\left((1+\epsilon)L\Gamma\right)^{-1+\frac{1}{\epsilon}}} = \frac{\frac{1}{\epsilon}\ln\left(1+\epsilon\right)L\Gamma)}{\frac{1-\epsilon}{\epsilon}\ln\left(1+\epsilon\right)L\Gamma)} = \frac{1}{1-\epsilon}$$

and thus

$$\gamma \leq \frac{\epsilon(1+\epsilon)}{(1-\epsilon)\ln(1+\epsilon)} \leq \frac{\epsilon(1+\epsilon)}{(1-\epsilon)(\epsilon-\epsilon^2/2)} \leq \frac{(1+\epsilon)}{(1-\epsilon)^2}$$

Here we use the fact that $\ln(1+\epsilon) \geq \epsilon - \epsilon^2/2$ (by Taylor series expansion of $\ln(1+\epsilon)$ around the origin). The proof of the approximation guarantee is completed by observing that $(1+\epsilon)/(1-\epsilon)^2 \leq (1+4\epsilon)$ for every $\epsilon < 0.15$. The runtime follows by substituting δ in the bound given by Lemma 1. □

5 Computing Minimum-Weight Feasible Steiner Trees

The key subroutine of the approximation algorithm given in the previous section is to compute, for a fixed k and given weights $w(X)$, $X \in \mathcal{V}$, a feasible tree $T \in \mathcal{T}_k$ minimizing $weight(T) = \frac{1}{g(T)} \sum_{X \in \mathcal{V}} \pi_T(X) w(X)$. Define a weight function w' on the vertices of the routing graph $G = (V, E)$ by setting $w'(v) = \frac{1}{g(T)} \sum_{v \in X \in \mathcal{V}} w(X)$, and let $w'(T) = \sum_{v \in V(T)} w'(v)$ be the total vertex weight w.r.t. w' of T. Then $weight(T) = w'(T)$, and the problem reduces to finding a tree $T \in \mathcal{T}_k$ with minimum total vertex weight w.r.t. w'.

Recall that for the GRBB problem and its extensions, $\mathcal{T}_k$ contains all Steiner trees connecting the source s_k with the sinks $t_k^1, \ldots, t_k^{q_k}$ such that the number of intermediate vertices on each tree path between s_k and t_k^i has the parity specified by a_k^i and does not exceed l_k^i. In this case we can further reduce the problem of finding the tree $T \in \mathcal{T}_k$ minimizing $w'(T)$ to the *minimum-cost directed rooted Steiner tree* (DRST) problem in a directed acyclic graph. Unfortunately, the minimum-cost DRST problem is NP-hard, and the fact that D_k is acyclic does not help since there is a simple reduction for this problem from arbitrary directed graphs to acyclic graphs. As far as we know, the best result for the DRST problem, due to Charikar et al. [3], gives $O(\log^2 q_k)$-approximate solutions in quasi-polynomial time $O(n^{3 \log q_k})$. Note, on the other hand, that the minimum-cost DRST can be found in polynomial time for small nets (e.g., in time $O(n^{M-1})$ for nets with at most M sinks, for $M = 2, 3, 4$); most of the nets in industrial VLSI designs fall into this category [10]. For nets of small size, Theorem 1 immediately gives:

Corollary 1. *If the maximum net size is $M \leq 4$, the algorithm in Figure 2 finds, for every $\epsilon < 0.15$, a feasible solution to the SP LP within a factor of $1/(1+4\epsilon)$ of optimum in time $O\left(\frac{1}{\epsilon^2} K n^{M-1} (\log n + \log \Gamma)\right)$.*

We have implemented both heuristics that use approximate DRSTs instead of optimum DRSTs and heuristics that decompose larger nets into nets with 2-4 pins before applying the algorithm in Figure 2; results of experiments comparing these approaches are reported in Section 7.

6 Rounding Fractional SP LP Solutions

In the previous two sections we presented an algorithm for computing near-optimal solutions to the SP LP. In this section we give two algorithms based on the randomized rounding technique of Raghavan and Thomson [14] (see also [11]) for converting these solutions to integer SP ILP solutions.

The first algorithm is to route net N_k with probability equal to $f_k = \sum_{T \in \mathcal{T}_k} f_T$ by picking, for selected nets, one of the trees $T \in \mathcal{T}_k$ with probability f_T / f_k. A drawback of this algorithm is that it requires the explicit representation of trees $T \in \mathcal{T}$ with $f(T) \neq 0$. Although the approximate SP LP algorithm produces a polynomial number of trees with non-zero f_T, storing all such trees

Input: Net- and edge-cumulated f_T values, $f_k = \sum_{T \in \mathcal{T}_k} f_T$ and $f_k(e) = \sum_{T \in \mathcal{T}_k:\ e \in E(T)} f_T$, $k = 1, \ldots, K$, $e \in E(D_k)$
Output: Routed trees $T_k \in \mathcal{T}_k$

For each $k = 1, \ldots, K$, select net N_k with probability f_k
Route each selected net N_k as follows:
  $T_k \leftarrow \{s_k\}$
  For each sink t_k^i in N_k do
    $P \leftarrow \emptyset$; $v \leftarrow t_k^i$
    While $v \notin T_k$ do
      Pick arc (u, v) with probability $\frac{f_k(u,v)}{\sum_{(w,v) \in E} f_k(w,v)}$
      $P \leftarrow P \cup \{(u, v)\}$; $v \leftarrow u$
    End while
    $T_k \leftarrow T_k \cup P$
  End for

Fig. 3. The random walk based rounding algorithm.

is infeasible for large problem instances. Our second rounding algorithm (Figure 3) takes as input the net- and edge-cumulated f_T values, $f_k = \sum_{T \in \mathcal{T}_k} f_T$, respectively $f_k(e) = \sum_{T \in \mathcal{T}_k:\ e \in E(T)} f_T$, thus using only $O(K|E|)$ space.

As the first rounding algorithm, the algorithm in Figure 3 routes each net N_k with a probability of $f_k = \sum_{T \in \mathcal{T}_k} f_T$. The difference is in how each chosen net is routed: to route net N_k, the algorithm performs *backward random walks* from each sink of N_k until reaching either the source of N_k or a vertex already connected to the source. The random walks are performed in the directed acyclic graphs used for DRST computation, with probabilities given by the normalized $f_k(e)$ values.

On the average, the total importance of the nets routed by each of the two algorithm is $\sum_{k=1}^{K} g_k f_k = \sum_{T \in \mathcal{T}} g(T) f_T$. By Theorem 1, this is within a factor of $1/(1 + 4\epsilon)$ of the optimum SP LP solution, which in turn is an upper-bound on the optimum SP ILP solution. Ensuring that no set capacity is exceeded can be accomplished in two ways. One approach is to solve the SP LP with set capacities scaled down by a small factor which guarantees that the rounded solution meets the *original* capacities with very high probability (see [11]). A more practical approach, extending the so-called *greedy-deletion algorithm* in [6] to multiterminal nets, is to repeatedly drop routed paths passing through over-used sets until feasibility is achieved.

7 Experimental Results

We have implemented four greedy algorithms for the GRBB problem; all four greedy algorithms route nets sequentially. For a given net, the algorithms start

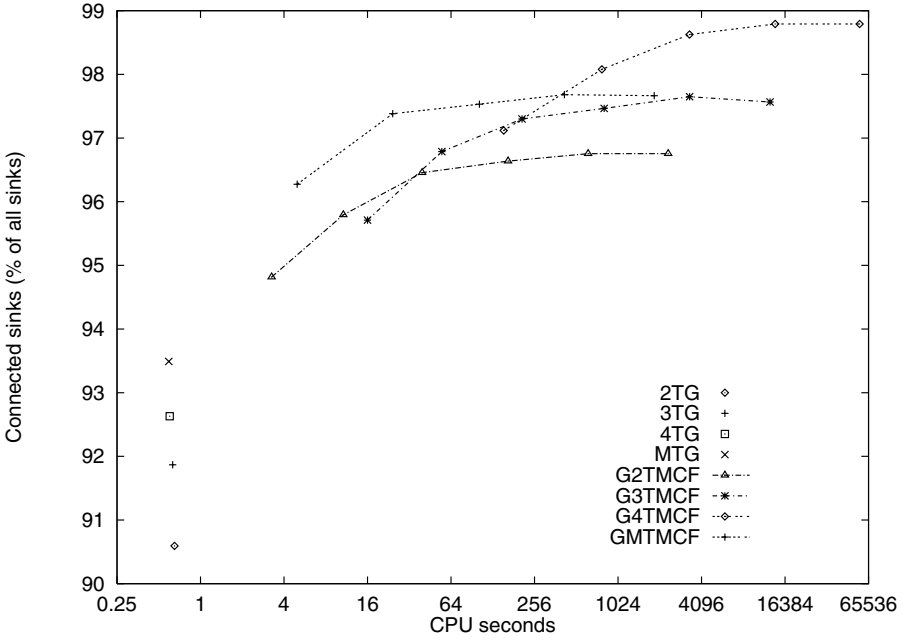


Fig. 4. Percent of sinks connected vs. CPU time.

with a tree containing only the net's source, then iteratively add shortest paths from each sink to the already constructed tree. The only difference is in whether or not net decomposition is used, and in the size of the decomposed nets. The first three algorithms—referred to as 2TG, 3TG, and 4TG, respectively—start by decomposing larger multiterminal nets into 2-, 3-, respectively 4-pin nets. The fourth algorithm, MTG, works on the original (undecomposed) nets.

We have also implemented four algorithms that approximate the fractional solution to the SP LP corresponding to GRBB problem (which generalizes the node-capacitated multiterminal multicommodity flow problem) and then apply randomized rounding. The first three algorithms (2TMCF, 3TMCF, and 4TMCF) decompose larger nets into 2-, 3-, respectively 4-pin nets then call the algorithm in Figure 2 with exact DRST computations. The fourth algorithm, MTMCF, works on the original (undecomposed) nets, using shortest-path trees as approximate DRSTs in the SP LP approximation algorithm.

Figure 4 plots the solution quality versus the CPU time (on a 195MHz SGI Origin 2000) of each implemented algorithm. The test cases used in our experiments were extracted from the next-generation (as of January 2000) microprocessor chip at SGI. The results clearly demonstrate the high quality of solutions obtained by rounding the approximate SP LP solutions. The MTMCF algorithm proves to be the best among all algorithms when the time budget is limited, providing significant improvements over greedy algorithms without undue runtime

penalty. However, the best convergence to the optimum is achieved by 4TMCF, which dominates all other algorithms when high time budgets are allowed.

References

1. C. Albrecht, "Provably good global routing by a new approximation algorithm for multicommodity flow", *Proc. ISPD*, 2000.
2. R.C. Carden and C.-K. Cheng, "A global router using an efficient approximate multicommodity multiterminal flow algorithm", *Proc. DAC*, 1991, pp. 316–321.
3. M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, and S. Cheung, "Approximation algorithms for directed Steiner problems", *J. Algorithms*, 33 (1999), pp. 73–91.
4. J. Cong, L. He, C.-K. Koh and P.H. Madden, "Performance optimization of VLSI interconnect layout", *Integration* 21 (1996), pp. 1–94.
5. J. Cong, T. Kong and D.Z. Pan, "Buffer block planning for interconnect-driven floorplanning", *Proc. ICCAD*, 1999, pp. 358–363.
6. F.F. Dragan, A.B. Kahng, I.I. Măndoiu, S. Muddu and A. Zelikovsky, "Provably good global buffering using an available buffer block plan", *Proc. ICCAD*, 2000, pp. 104–109.
7. L.K. Fleischer, "Approximating fractional multicommodity flow independent of the number of commodities", *Proc. 40th Annual Symposium on Foundations of Computer Science*, 1999, pp. 24–31.
8. N. Garg and J. Könemann, "Faster and simpler algorithms for multicommodity flow and other fractional packing problems", *Proc. 39th Annual Symposium on Foundations of Computer Science*, 1998, pp. 300–309.
9. J. Huang, X.-L. Hong, C.-K. Cheng and E.S. Kuh, "An efficient timing-driven global routing algorithm", *Proc. DAC*, 1993, pp. 596-600.
10. A.B. Kahng and G. Robins. *On Optimal Interconnections for VLSI*, Kluwer Academic Publishers, Norwell, Massachusetts, 1995.
11. R. Motwani, J. Naor, and P. Raghavan, "Randomized approximation algorithms in combinatorial optimization", In *Approximation algorithms for NP-hard problems* (Boston, MA, 1997), D. Hochbaum, Ed., PWS Publishing, pp. 144–191.
12. A.P.-C. Ng, P. Raghavan, and C.D. Thomson, "Experimental results for a linear program global router". *Computers and Artificial Intelligence*, 6 (1987), pp. 229–242.
13. C.A. Phillips, "The network inhibition problem", *Proc. 25th Annual ACM Symposium on Theory of Computing*, 1993, pp. 776–785.
14. P. Raghavan and C.D. Thomson, "Randomized rounding", *Combinatorica*, 7 (1987), pp. 365–374.
15. P. Raghavan and C.D. Thomson, "Multiterminal Global Routing: A Deterministic Approximation Scheme", *Algorithmica*, 6 (1991), pp. 73–82.
16. E. Shragowitz and S. Keel, "A global router based on a multicommodity flow model", *Integration*, 5 (1987), pp. 3–16.
17. X. Tang and D.F. Wong, "Planning buffer locations by network flows", *Proc. ISPD*, 2000.

The Challenges of Delivering Content on the Internet

F. Thomson Leighton

Massachusetts Institute of Technology
and
Akamai Technologies, Inc.
ftl@math.mit.edu

Abstract. In this talk, we will give an overview of how content is distributed on the internet, with an emphasis on the approach being used by Akamai. We will describe some of the technical challenges involved in operating a network of thousands of content servers across multiple geographies on behalf of thousands of customers. The talk will be introductory in nature and should be accessible to a broad audience.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, p. 338, 2001.

Upward Embeddings and Orientations of Undirected Planar Graphs

Walter Didimo[1] and Maurizio Pizzonia[1]

Dipartimento di Informatica e Automazione, Università di Roma Tre,
Via della Vasca Navale 79, 00146 Roma, Italy
{didimo,pizzonia}@dia.uniroma3.it

Abstract. An *upward embedding* of an embedded planar graph states, for each vertex v, which edges are incident to v "above" or "below" and, in turn, induces an *upward orientation* of the edges. In this paper we characterize the set of all upward embeddings and orientations of a plane graph by using a simple flow model. We take advantage of such a flow model to compute upward orientations with the minimum number of sources and sinks of 1-connected graphs. Our theoretical results allow us to easily compute visibility representations of 1-connected graphs while having a certain control over the width and the height of the computed drawings, and to deal with partial assignments of the upward embeddings "underlying" the visibility representations.

1 Introduction

Let G be an undirected planar graph with a given planar embedding. Loosely speaking, an *upward embedding* (also called an *upward representation*) of G is given by splitting, for each vertex v of G, the ordered circular list of the edges that are incident to v into two linear lists $L_{above}(v)$ and $L_{below}(v)$, in such a way that there exists a planar drawing Γ of G with the following properties: (i) all the edges are monotonically increasing in the vertical direction; (ii) for each vertex v the edges in $L_{above}(v)$ ($L_{below}(v)$) are incident to v above (below) the horizontal line through v. Drawing Γ is said to be an *upward drawing* of G. An orientation of all edges of Γ from bottom to top defines an orientation of all edges of G, which we call an *upward orientation* of G. Hence, any upward embedding of G induces an upward orientation of G. Figure 1 shows an upward embedding of a plane graph and the upward orientation induced by it.

Upward embeddings and orientations of undirected graphs have been widely studied within specific theoretical and application domains. For example, a deep investigation of the properties of upward embeddings and drawings for ordered sets and planar lattices can be found in [15,14,6,18]. Relations between the problem of finding bar layout drawings of weighted undirected graphs and the problem of computing upward orientations with specific properties are provided in [16,20]. An important class of upward orientations is represented by the so called *bipolar orientations* (or *st-orientations*). A bipolar orientation of an undirected planar graph G is an upward orientation of G with exactly one source s

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 339–351, 2001.

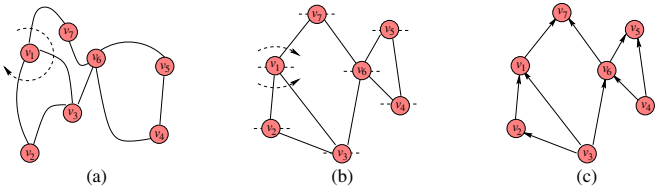


Fig. 1. (a) An embedded planar graph. (b) An upward embedding of the embedded planar graph. For each vertex v_i of the graph the edges in $L_{below}(v_i)$ and $L_{above}(v_i)$ are drawn incident below and above the horizontal line through v_i, respectively. (c) The upward orientation induced by the upward embedding.

(vertex without in-edges) and one sink t (vertex without out-edges). A bipolar orientation of G with source s and sink t exists if and only if $G \cup \{(s,t)\}$ is biconnected. Finding a bipolar orientation of a planar graph is often the first step of many algorithms in graph theory and graph drawing. The properties of bipolar orientations have been extensively studied in [7], and a characterization of bipolar orientations in terms of a network flow model is described in [5].

Many results on upward embeddings of digraphs have been also provided in the literature. In this case, the orientation of the edges of the graph is given, and a classical problem consists in finding a planar upward embedding within such an orientation. Clearly, a planar upward embedding of a digraph might not exist. In [3] a polynomial time algorithm for testing the existence of planar upward embeddings of a digraph within a given embedding is described. The algorithm is also able to construct an upward embedding if there exists one. In the variable embedding setting the upward planarity testing problem for digraphs is NP-complete [11], but it can be solved in polynomial time for digraphs with a single source [4].

In this paper we focus on upward embeddings and orientations of undirected planar graphs. The main contributions of our work are the following:

- Starting from the properties on upward planarity given in [3], we provide a full characterization of the set of all upward embeddings and orientations of any embedded planar graph (Section 3.1). It is based on a network flow model, which is related to the one used in [5] for characterizing bipolar orientations. In particular, if the graph is biconnected, our flow model also captures all bipolar orientations of the graph.
- We describe flow based polynomial time algorithms for computing upward embeddings of the input graph. Such algorithms allow us to deal with partial assignments of the upward embedding (Section 3.1). Further, we provide a polynomial time algorithm to compute upward orientations with the minimum number of sources and sinks (Section 3.2). An upward orientation with the minimum number of sources and sinks can be viewed as a natural extension of the concept of bipolar orientation to 1-connected graphs.
- We describe a simple technique to compute visibility representations of 1-connected planar graphs (Section 4), which can be of practical interest for

graph drawing applications. It is based on the computation of an upward embedding of the graph, and does not require running any augmentation algorithm to initially make the graph biconnected. Compared to a standard technique that uses the approximation algorithm in [10] to make the graph biconnected, the algorithm we propose is faster and achieves similar results in terms of area of the visibility representation. We also present (Section 4.1) a preliminary experimental study that shows how our technique can be used to have a certain control over the width and the height of the visibility representations.

In Section 2 we give some basic definitions and results on upward embeddings and orientations of undirected planar graphs. Due to space limitations, all the proofs of the theorems are omitted, and can be found in [9].

2 Basic Definitions and Results on Upward Embeddings

Let G be a graph. A *drawing* Γ of G maps each vertex u of G into a point p_u of the plane and each edge (u, v) of G into a Jordan curve between p_u and p_v. Γ is *planar* if two distinct edges never intersect except at common end-points. G is *planar* if it admits a planar drawing. A planar drawing Γ of G divides the plane into topologically connected regions called *faces*. Exactly one of these faces is unbounded, and it is said to be *external*; the others are called *internal* faces. Also, for each vertex v of G, Γ induces a circular clockwise ordering of the edges incident on v. The choice ϕ of such an ordering for each vertex of G and of an external face is called a *planar embedding* of G. A planar graph G with a given planar embedding ϕ is called an *embedded planar* graph and denoted by G_ϕ. A *drawing* of G_ϕ is a planar drawing of G that induces ϕ as the planar embedding.

Let G_ϕ be an (undirected) embedded planar graph. An *upward embedding* $\mathcal{E}_\phi$ of G_ϕ is a splitting of the adjacency lists of all vertices of G_ϕ such that: (**Pr.a**) for each vertex v of G_ϕ the circular clockwise list $L(v)$ of the edges incident on v is split into two linear lists, $L_{below}(v)$ and $L_{above}(v)$, so that the circular list obtained by concatenating $L_{above}(v)$ and the reverse of $L_{below}(v)$ is equal to $L(v)$; (**Pr.b**) there exists a planar drawing $\Gamma(\mathcal{E}_\phi)$ of G_ϕ such that all the edges are monotonically increasing in the vertical direction and for each vertex v of G_ϕ the edges of $L_{below}(v)$ and $L_{above}(v)$ are incident to v below and above the horizontal line through v, respectively. We say that $\Gamma(\mathcal{E}_\phi)$ is a *drawing* of $\mathcal{E}_\phi$ and an *upward drawing* of G_ϕ.

An upward embedding $\mathcal{E}_\phi$ of G_ϕ uniquely induces an *upward orientation* $\mathcal{O}_\phi$ of G_ϕ. Namely, for each edge $e = (u, v)$ such that $e \in L_{above}(u)$ and $e \in L_{below}(v)$, we orient e from u to v (see Figure 1). Conversely, an upward orientation defines in general a class of possible upward embeddings inducing that orientation. A *source* of $\mathcal{E}_\phi$ is a vertex v of G_ϕ such that $L_{below}(v)$ is empty. A source has only out-edges with respect to orientation $\mathcal{O}_\phi$. A *sink* of $\mathcal{E}_\phi$ is a vertex v of G_ϕ such that $L_{above}(v)$ is empty. A sink has only in-edges with respect to $\mathcal{O}_\phi$.

Given a vertex v of G_ϕ, we denote by $deg(v)$ the number of edges incident on v. An *angle* of G_ϕ at vertex v is a pair of clockwise consecutive edges incident

on v. In particular, if $deg(v) = 1$, and if we denote by e the edge incident on v, $\{e, e\}$ is an angle. Given a splitting of the adjacency lists of G_ϕ that verifies **Pr.a**, an angle $\{e_1, e_2\}$ at vertex v of G_ϕ can be of three types. *Large*: (i) both e_1 and e_2 belong to $L_{below}(v)$ ($L_{above}(v)$), and (ii) e_1 and e_2 are the first (last) edge and the last (first) edge of $L_{below}(v)$ ($L_{above}(v)$), respectively. We associate a label L with a large angle. *Flat*: (i) $e_1 \in L_{below}(v)$ and $e_2 \in L_{above}(v)$ or, (ii) $e_1 \in L_{above}(v)$ and $e_2 \in L_{below}(v)$. We associate a label F with a flat angle. *Small*: in all the other cases. We associate a label S with a small angle.

Figure 2 shows the labeling of the angles of a graph G_ϕ determined by an upward embedding $\mathcal{E}_\phi$. Each drawing of $\mathcal{E}_\phi$ maps the angles of G_ϕ to geometric angles such that large and small angles always correspond to geometric angles larger and smaller than 180 degrees, respectively. Both the two edges that form a large or a small angle at vertex v are incident to v either above or below the horizontal line through v. Instead, a flat angle at vertex v corresponds to a geometric angle that can be either larger or smaller than 180 degrees; in any case, an edge of the angle is incident to v above the horizontal line through v while the other edge is incident to v below the same line.

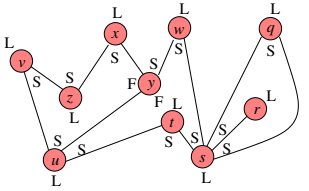


Fig. 2. Labeling of the angles of an embedded planar graph determined by an upward embedding of the graph.

Let f be a face of G_ϕ. We call *border* of f the alternating circular list of the vertices and edges that form the boundary of f. Note that, if the graph is not biconnected an edge or a vertex may appear more than once in the border of f. We say that an angle $\{e_1, e_2\}$ at vertex v *belongs* to face f if e_1, e_2, and v belong to the border of f. The *degree* of f, denoted by $deg(f)$, is the number of edges in the border of f. Observe that, $deg(f)$ is equal to the number of angles of f.

Consider now any labeling of the angles of G_ϕ with labels L, S, and F. For each face f of G_ϕ denote by $L(f)$, $S(f)$, and $F(f)$ the number of angles that belong to f with label L, S, and F, respectively. Also, for each vertex v of G_ϕ denote by $L(v)$, $S(v)$, and $F(v)$ the number of angles at vertex v with label L, S, and F, respectively. The following lemma is a restatement of a known result on upward planarity [3].

Lemma 1. *Let $\mathcal{E}_\phi$ be a splitting of the adjacency lists of G_ϕ that verifies* **Pr.a**, *and consider the labeling of the angles of G_ϕ determined by it. $\mathcal{E}_\phi$ is an upward embedding of G_ϕ if and only if the following properties hold:*

(a) $S(f) = L(f) + 2$, *for each internal face* f *of* G_ϕ.
(b) $S(f) = L(f) - 2$, *for the external face* f *of* G_ϕ.
(c) $F(v) = 2$, $S(v) = deg(v) - 2$, *and* $L(v) = 0$, *for each vertex* v *of* G_ϕ *such that both* $L_{above}(v)$ *and* $L_{below}(v)$ *are not empty.*
(d) $F(v) = 0$, $S(v) = deg(v) - 1$, *and* $L(v) = 1$, *for each vertex* v *of* G_ϕ *such that either* $L_{above}(v)$ *or* $L_{below}(v)$ *is empty.*

Properties (c) and (d) of Lemma 1 state that if $\mathcal{E}_\phi$ is an upward embedding of G_ϕ, each source or sink of $\mathcal{E}_\phi$ has exactly one large angle and no flat angles, while each vertex that is neither a source nor a sink has exactly two flat angles and no large angles. The next lemma provides a different formulation for properties (a) and (b).

Lemma 2. *Properties* (a) *and* (b) *of* Lemma 1 *are equivalent to the following properties:* (a') $deg(f) - 2 = 2L(f) + F(f)$, *for each internal face* f *of* G_ϕ. (b') $deg(f) + 2 = 2L(f) + F(f)$, *for the external face* f *of* G_ϕ.

3 Characterizing Upward Embeddings

In this section we give a full characterization of the set of all upward embeddings of a general embedded planar graph (Section 3.1). This also implies a characterization of all upward orientations of the given graph. Our characterization uses a flow model that is related to the one described in [5] for bipolar orientations. Also, we show how it is possible to add costs to our flow model in order to compute in polynomial time an upward orientation with the minimum number of sources and sinks (Section 3.2).

3.1 A Flow Model for Characterizing Upward Embeddings

The following theorem characterizes the class of labelings that are determined by all upward embeddings of an embedded planar graph. Observe that the characterization of such a class of labelings does not depend either on the choice of a splitting of the adjacency lists of the graph, in contrast to the result given in Lemma 1, or on the choice of an orientation of the graph.

Theorem 1. *Let* $\mathcal{L}$ *be any labeling of the angles of an embedded graph* G_ϕ *with labels* L, S, *and* F. $\mathcal{L}$ *is the labeling determined by an upward embedding of* G_ϕ *if and only if the following properties hold:*
(a') $deg(f) - 2 = 2L(f) + F(f)$, *for each internal face* f *of* G_ϕ.
(b') $deg(f) + 2 = 2L(f) + F(f)$, *for the external face* f *of* G_ϕ.
(c') *For each vertex* v *either* $F(v) = 2$ *and* $L(v) = 0$ *or* $F(v) = 0$ *and* $L(v) = 1$.

We call *upward labeling* of G_ϕ a labeling of the angles of G_ϕ that verifies properties (a'), (b'), and (c') of Theorem 1. The result of Theorem 1 allows us to describe all upward embeddings of G_ϕ by considering all upward labelings of G_ϕ. The proof of the theorem (see [9]) gives a method to construct the upward

embedding associated with an upward labeling. Actually, for each upward labeling, there are exactly two "symmetric" upward embeddings that determine it; they are obtained one from the other by simply exchanging list $L_{above}(v)$ with list $L_{below}(v)$ for each vertex v and then reversing such lists (see Figure 4 (b)).

We now provide a network flow model that characterizes all the upward labelings of G_ϕ. Because of the above considerations, this flow model provides a characterization of all upward embeddings of G_ϕ. We associate with G_ϕ a flow network $\mathcal{N}_\phi$, such that the integer feasible flows on $\mathcal{N}_\phi$ are in one-to-one correspondence with the upward labelings of G_ϕ. Flow network $\mathcal{N}_\phi$ is a directed graph defined as follows (see Figure 3): (i) The nodes of $\mathcal{N}_\phi$ are the vertices (*vertex-nodes*) and the faces (*face-nodes*) of G_ϕ. Each vertex-node supplies flow 2 and each face-node associated with face f of G_ϕ demands a flow equal to $deg(f) - 2$ if f is internal and $deg(f) + 2$ if f is external. (ii) For each angle of G_ϕ at vertex v in face f there is an associated arc (v, f) of $\mathcal{N}_\phi$ with lower capacity 0 and upper capacity 2.

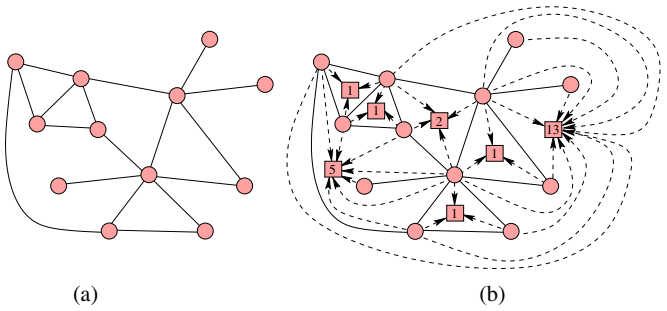


Fig. 3. (a) An embedded planar graph G_ϕ. (b) Flow network $\mathcal{N}_\phi$ associated with G_ϕ. The vertex-nodes are circles and the face-nodes are squares. Each face-node is marked with its demand. The arcs of the networks are dashed.

Observe that in $\mathcal{N}_\phi$ the total demand is equal to the total supply. In fact: $\sum_{f \in F} (deg(f) - 2) + 4 = \sum_{f \in F} deg(f) - 2|F| + 4 = 2|E| - 2|F| + 4 = 2|V|$. The intuitive interpretation of the flow model in terms of upward embedding is as follows: (i) Each unit of flow represents a flat angle, with the convention that a large angle counts as two flat angles; an arc a of $\mathcal{N}_\phi$ has flow $0, 1$, or 2, depending on the fact that its associated angle is small, flat, or large, respectively. (ii) The demand of each face-node and the supply of each vertex-node reflect the balancing properties (a'), (b') and (c'). Figure 4 shows a feasible flow on the network associated with an embedded planar graph, the corresponding upward labeling, and the two "symmetric" upward embeddings associated with the labeling. Theorem 2 formally proves the correctness of the intuitive interpretation above described.

The flow network used in [5] to characterize the bipolar orientations of an embedded planar graph is tailored for biconnected planar graphs and captures only bipolar orientations. The values of the flow are not able to represent large

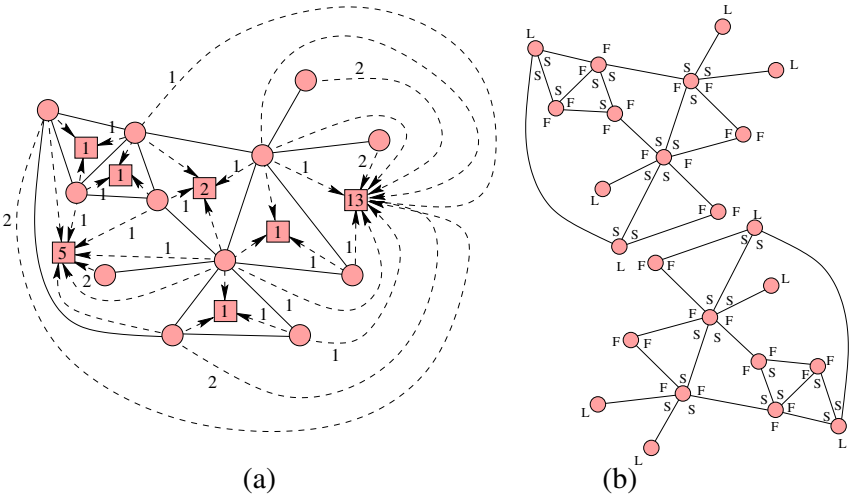


Fig. 4. (a) A feasible flow on the network associated with an embedded planar graph. Only the flow values different from zero are shown. (b) The upward labeling $\mathcal{L}$ corresponding to the flow and the "symmetric" upward embeddings associated with $\mathcal{L}$.

angles (the flow values are only 0 or 1), except for the source and the sink of the orientation. Our flow network generalizes this network to represent any kind of upward orientations and embeddings, including the bipolar orientations.

Theorem 2. *Let G_ϕ be an embedded planar graph and let $\mathcal{N}_\phi$ be the flow network associated with G_ϕ. There is a one-to-one correspondence between the set of the upward labelings of G_ϕ and the set of the integer feasible flows on $\mathcal{N}_\phi$.*

Theorem 1 and Theorem 2 allow us to compute an upward embedding of an embedded planar graph G_ϕ, by computing an integer feasible flow on network $\mathcal{N}_\phi$. Denote by n the number of vertices of G_ϕ. Since network $\mathcal{N}_\phi$ is planar and has $O(n)$ vertices, a feasible flow on $\mathcal{N}_\phi$ can be computed in $O(n \log n)$ time by applying a known maximum flow algorithm for planar networks [1]. Also, both $\mathcal{N}_\phi$ and an upward embedding associated with a feasible flow on $\mathcal{N}_\phi$ can be constructed in linear time. Therefore, an upward embedding of any embedded planar graph can be constructed in $O(n \log n)$ with the above technique.

We remark that there are two main advantages of computing upward embeddings of a general plane graph G_ϕ by using the flow model described so far: No augmentation algorithms have to be used to initially biconnected the graph (we just apply a standard flow algorithm); it is possible to fix the flow on some arcs of the network to constrain the upward embedding to have a partially specified "shape". For example, we can specify that an angle must be large in a certain face, or that some vertices must be neither sources nor sinks. In the next section we describe how to compute upward embeddings with the minimum number of sources and sinks, by adding costs to our network.

3.2 Minimizing Sources and Sinks

Computing an upward embedding of G_ϕ with the minimum number of sources and sinks (which we call *optimal upward embedding* for simplicity) is equivalent to computing an upward embedding with the minimum number of large angles. Clearly, if the graph is biconnected, the problem is reduced to the computation of a bipolar orientation. For this reason, we regard the concept of optimal upward orientation as the natural extension of the definition of bipolar orientation to the case of general connected graphs.

The flow model we use to compute an optimal upward orientation of G_ϕ is a variation of the one described for characterizing upward embeddings (see Section 3.1). We add a linear number of arcs to network $\mathcal{N}_\phi$ and we equip the arcs of the new network with costs. Each unit of cost represents a large angle. We also reduce the upper capacity of all the arcs of the network. More in detail, we define a network $\mathcal{N}'_\phi$ as follows: (i) The nodes of $\mathcal{N}'_\phi$ are again the vertices (vertex-node) and the faces (face-nodes) of G_ϕ. Each vertex-node again supplies flow 2 and each face-node associated with face f of G_ϕ again demands flow $deg(f) - 2$ if f is internal and $deg(f) + 2$ if f is external. (ii) For each angle of G_ϕ at vertex v in face f there is an associated pair of directed arcs $a_v = (v, f), a'_v = (v, f)$ in $\mathcal{N}'_\phi$. Both the arcs have lower capacity 0 and upper capacity 1. Also, arc a_v has cost 0 while arc a'_v has cost 1.

In $\mathcal{N}'_\phi$ we compute a minimum cost flow x. The interpretation of the flow in terms of upward labeling is similar to the one given for $\mathcal{N}_\phi$, with a slightly variation due to the additional arcs and costs. We first observe that for each pair of arcs a_v, a'_v it never happens $x(a_v) = 0$ and $x(a'_v) = 1$, due to the fact that the cost of a_v is 0 and the cost of a'_v is 1. In fact, if $x(a_v) = 0$ and $x(a'_v) = 1$, then there would exist a negative cost cycle represented by the two arcs a'_v, a_v, and it would be possible to derive a new flow x' from x by simply exchanging one unit of flow between a'_v and a_v (i.e., $x'(a_v) = 1$ and $x'(a'_v) = 0$). This would imply that x' has a cost smaller than the cost of x, in contrast to the assumption that x has the minimum cost. Hence, the only possibilities for the flow on arcs a_v, a'_v are: (i) $x(a_v) = x(a'_v) = 0$, the angle associated with arcs a_v, a'_v is small. (ii) $x(a_v) = 1$ and $x(a'_v) = 0$, the angle associated with arcs a_v, a'_v is flat. (iii) $x(a_v) = x(a'_v) = 1$, the angle associated with arcs a_v, a'_v is large.

Note that, only in the third case we have cost 1 on arcs a_v, a'_v, while in the other two cases we have cost 0. This implies that the total cost of flow x on $\mathcal{N}'_\phi$ represents the total number of large angles of the corresponding upward embedding of G_ϕ. Hence, since x has the minimum cost, the corresponding upward embedding has the minimum number of large angles.

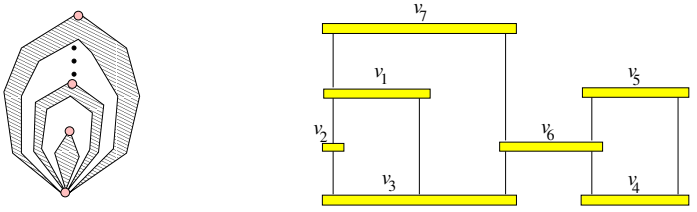
Let n be the number of vertices of G_ϕ. Since network $\mathcal{N}'_\phi$ is planar and has $O(n)$ vertices, and since its total demand (supply) is $O(n)$, a minimum cost flow on $\mathcal{N}'_\phi$ can be computed in $O(n^{\frac{7}{4}} \log n)$ time by the algorithm described in [12].

We conclude this section by giving an upper bound on the number of sources and sinks of an optimal upward embedding.

Lemma 3. *An optimal upward embedding of an embedded planar graph G_ϕ has at most $B+1$ sources and sinks, where B is the number of blocks of G_ϕ. Also, in*

the optimal upward embedding each block contains at most one source and one sink.

The bound of Lemma 3 is strict and a class of plane graphs whose upward embeddings have $B+1$ sources and sinks can be obtained by nesting each block into another, as shown by the example of Figure 5(a).



(a) (b)

Fig. 5. (a) A class of embedded planar graphs whose optimal upward embeddings have $B+1$ sources and sinks (circles). (b) A visibility representation of the upward embedded graph shown in Figure 1(b).

4 Algorithms for Visibility Representations

We use the above results on upward embeddings to compute drawings of general connected planar graphs. Namely, we focus on graph drawing algorithms which require the computation of a (*weak-*)*visibility representation* of the input graph as a preliminary step [8]. In a visibility representation (see Figure 5(b)), each vertex is mapped to a horizontal segment and each edge (u, v) is mapped to a vertical segment between the segments associated with u and v; horizontal segments do not overlap, and each vertical segment only intersect its extreme horizontal segments.

A standard technique [8] to compute a visibility representation of a plane graph G consists of calculating a bipolar orientation of G; if G is not biconnected it is augmented to a biconnected planar graph by adding a suitable number of dummy edges, which will be removed in the final drawing. However, this technique has several drawbacks: (i) Adding too many dummy edges may lead to a final drawing with area much bigger than necessary. On the other side, the problem of adding the minimum number of edges to make a planar graph biconnected and still planar is NP-hard [13]. (ii) Although an approximation algorithm for the above augmentation problem exists [10] (which reaches the optimal solution in many cases), implementing it efficiently is quite difficult, because it requires us to deal with the block tree of the graph and with an efficient incremental planarity testing algorithm. In fact, such an approximation

algorithm has $O(n^2T)$ running time, where T is the amortized time bound per query or insertion operation of the incremental planarity testing algorithm. (iii) The presence of dummy edges in the graph makes difficult to deal with partial assignments of the upward embedding.

In [19] it is sketched a strategy for computing visibility representations of general connected graphs; such a strategy does not explicitly detail how to perform some necessary topological and geometric operations. We propose the following algorithm for computing a visibility representation of a 1-connected embedded planar graph G_ϕ.

Algorithm *Visibility-Upward-Embedding*

1. Compute an upward embedding $\mathcal{E}_\phi$ of G_ϕ, by calculating a feasible flow on network $\mathcal{N}_\phi$.
2. Compute an upward embedded st-graph S_ϕ including G_ϕ and preserving $\mathcal{E}_\phi$ on G_ϕ, by using the linear time saturation procedure described in [3] (note that, S_ϕ is biconnected).
3. Compute a visibility representation of S_ϕ (within its upward embedding) by using any known linear time algorithm [8], and then remove the edges introduced by the saturation procedure.

Algorithm *Visibility-Upward-Embedding* has $O(n \log n)$ running time, because its time complexity is dominated by the cost of computing a feasible flow on $\mathcal{N}_\phi$. We experimentally observed that the area of the visibility representations produced by this algorithm can be dramatically improved by computing upward embeddings with the minimum number of sources and sinks. To do that we just apply a min-cost-flow algorithm in Step 1. Clearly, in this case, the running time of the whole algorithm grows to $O(n^{\frac{7}{4}} \log n)$. The following theorem summarizes the main contributions of this section.

Theorem 3. *There exists an $O(n^{\frac{7}{4}} \log n)$ time algorithm that computes an upward embedding of an embedded 1-connected planar graph with the minimum number of sources and sinks.*

4.1 Experimentation

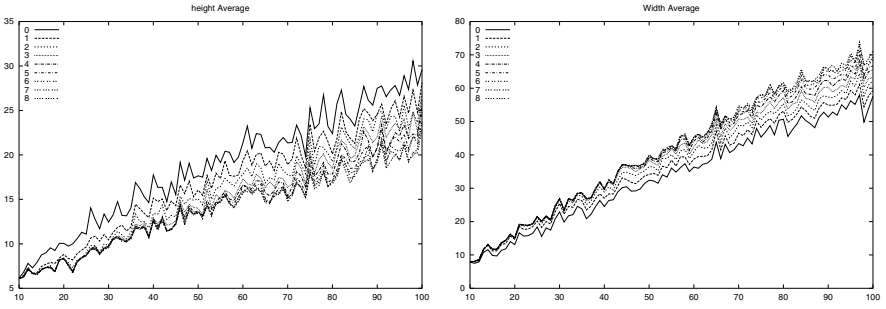
We present a preliminary study that shows how algorithm *Visibility Upward Embedding* can be slightly refined in order to get a certain control over the width and the height of visibility representations of 1-connected planar graphs. We start from the intuition that by re-arranging the blocks around the cutvertices in the upward embedding, it is possible to reduce the height or the width of the visibility representation. Namely, if v is a cutvertex and f a face which is doubly incident on v, placing all the blocks of v either above or below leads to a reduction of the height and to an increase in the width. Such re-arrangement is easily performed by exploiting the flow network associated with the plane graph. In such a network we can always move a unit of flow from an arc $a_1 = (v, f)$ to

another arc $a_2 = (v, f)$ keeping the feasibility of the flow. Clearly, arcs a_1 and a_2 exist only if v is a cutvertex.

The experimentation has been performed on a randomly generated test suite of 1820 graphs whose number n of vertices ranges from 10 to 100 (20 instances for each value of n). The graphs are planar and equipped with an embedding. Each graph of the test suite has a number of cutvertices between $n/10$ and $n/5$, the number of blocks attached to a cutvertex is between 2 and 5, the number of cutvertices of a block is between 1 and 5, and each biconnected component is generated by using the algorithm in [2]. A detailed description of the procedure can be found in [17].

For each graph of the test suite, we first compute an upward embedding having the minimum number of sources and sinks that keeps the given embedding unchanged. We proceed by redistributing the flow around at most a number k of cutvertices in such a way to place all their blocks above or below them. Note that, such a redistribution can be easily done in linear time. We average the width and the height on all the graphs having the same number of vertices. Charts in Figure 6 graphically show the results of the experimentation for k ranging from 0 to 8. It is interesting to observe how, for the same value of n and increasing k, the average of the width increases while the average of the height decreases.

Also, Figure 7 compares the area of the drawings computed with our strategy, where k is chosen equal to the total number of cutvertices of the graph, against the area of the drawings computed with a standard technique which uses the approximation algorithm in [10] to initially make the graph biconnected. In the two strategies we use the same algorithm for producing the visibility representation from the st-graph.



(a) height (b) width

Fig. 6. The charts show how re-arranging the blocks around cutvertices affects the width and the height of the visibility representation.

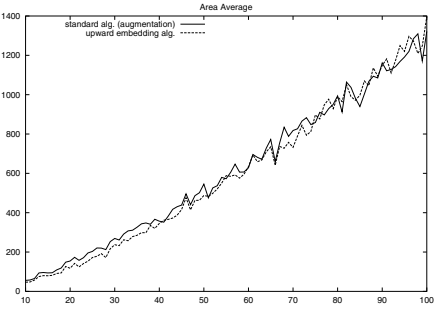


Fig. 7. Area of the drawings computed with our strategy against the area of the drawings computed with a standard technique based on a sophisticated augmentation algorithm (average values). The x-axis represents the number of vertices.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. In G. L. Nemhauser, A. H. G. Rinnooy Kan, and M. J. Todd, editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management*, pages 211–360. North-Holland, 1990.
2. P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum numbr of bends. *IEEE Transactions on Computers*, 49(8), 2000.
3. P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 6(12):476–497, 1994.
4. P. Bertolazzi, G. Di Battista, C. Mannino, and R. Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM J. Comput.*, 27(1):132–169, 1998.
5. M. Bousset. A flow model of low complexity for twisting a layout. In *Workshop of GD'93*, pages 43–44, Paris, 1993.
6. J. Czyzowicz, A. Pelc, and I. Rival. Drawing orders with few slopes. Technical Report TR-87-12, Department of Computer Science, University of Ottawa, 1987.
7. H. de Fraysseix, P. O. de Mendez, and P. Rosenstiehl. Bipolar orientations revisited. *Discrete Appl. Math.*, 56:157–179, 1995.
8. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.
9. W. Didimo and M. Pizzonia. Upward embeddings and orientations of undirected planar graphs. Technical Report RT-DIA-65-2001, University of Roma Tre, 2001.
10. S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Symposium on Discrete Algorithms (SODA '98)*, pages 260–269, 1998.
11. A. Garg and R. Tamassia. On the computational complexity of upward and rectilinear planarity testing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD '94)*, volume 894 of *Lecture Notes Comput. Sci.*, pages 286–297. Springer-Verlag, 1995.
12. A. Garg and R. Tamassia. A new minimum cost flow algorithm with applications to graph drawing. In S. C. North, editor, *Graph Drawing (Proc. GD '96)*, volume 1190 of *Lecture Notes Comput. Sci.*, pages 201–216. Springer-Verlag, 1997.
13. G. Kant and H. L. Bodlaender. Planar graph augmentation problems. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes Comput. Sci.*, pages 286–298. Springer-Verlag, 1991.

14. D. Kelly. Fundamentals of planar ordered sets. *Discrete Math.*, 63:197–216, 1987.
15. D. Kelly and I. Rival. Planar lattices. *Canad. J. Math.*, 27(3):636–665, 1975.
16. D. G. Kirkpatrick and S. K. Wismath. Weighted visibility graphs of bars and related flow problems. In *Proc. 1st Workshop Algorithms Data Struct.*, volume 382 of *Lecture Notes Comput. Sci.*, pages 325–334. Springer-Verlag, 1989.
17. M. Pizzonia. *Engineering of Graph Drawing Algorithms for Applications.* PhD thesis, Dipartimento di Informatica e Sistemistica, Università "La Sapienza" di Roma, 2001.
18. I. Rival. Reading, drawing, and order. In I. G. Rosenberg and G. Sabidussi, editors, *Algebras and Orders*, pages 359–404. Kluwer Academic Publishers, 1993.
19. R. Tamassia and I. G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete Comput. Geom.*, 1(4):321–341, 1986.
20. S. K. Wismath. *Bar-Representable Visibility Graphs and Related Flow Problems.* Ph.D. thesis, Dept. Comput. Sci., Univ. British Columbia, 1989.

An Approach for Mixed Upward Planarization*

Markus Eiglsperger and Michael Kaufmann

Universität Tübingen, Wilhelm-Schickard-Institut für Informatik
{eiglsper,mk}@informatik.uni-tuebingen.de

Abstract. In this paper, we consider the problem of finding a mixed upward planarization of a mixed graph, i.e., a graph with directed and undirected edges. The problem is a generalization of the planarization problem for undirected graphs and is motivated by several applications in graph drawing. We present a heuristical approach for this problem which provides good quality and reasonable running time in practice, even for large graphs. This planarization method combined with a graph drawing algorithm for upward planar graphs can be seen as a real alternative to the wellknown Sugiyama algorithm.

1 Introduction

Research for upward drawings of digraphs has been studied extensively in the last years. One reason is that such drawings have many applications in areas like workflow, project management and data flow.

An *upward drawing* of a digraph is a drawing such that all the edges are represented by curves monotonically increasing in the vertical direction. Note that such a drawing exists only if the digraph is acyclic.

A straightforward generalization of upward drawings are *mixed upward drawings*. In mixed upward drawings, only a part of the edges in the graph are directed und must point upward. Note that such a drawing exists only if the directed part of the graph is acyclic.

Mixed drawings arise in applications where the edges of the graph can be partioned into a set which denotes structural information and a set which does not carry structural information. An example is UML class diagrams [4] arising in software engineering. In these diagrams, the vertices of the graph represent classes in an object-oriented software system, and edges represent relations between these classes. There are two main types of relations: *generalizations* and *associations*. The generalization relations describe structural information and form a directed acyclic subgraph in the diagram. It is an often employed convention to draw generalizations upward, whereas associations can have arbitrary directions [18].

The most popular approach for creating upward drawings of digraphs is probably the Sugiyama algorithm [19]. The main idea of the Sugiyama algorithm is to assign layers to the vertices of the graph, such that edges point in ascending

* Partially supported by DFG-Grant Ka812/8-1

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 352-364, 2001.

layer order. In a next step, the number of crossings are minimized by ordering the nodes in the layer. For a fixed layer assignment, we call a graph *level planar* if it has a drawing which respects the layering and has no crossings. Several heuristics are proposed for this step and are used in practice, but there are also efficient algorithms to solve the level planarity problem [13] [14]. There have been several attempts to apply the Sugiyama algorithm also to mixed graphs, i.e., in [18] the approach is used for UML class diagrams.

The principal step of the Sugiyama algorithm, the layer assignment, is also its most severe drawback. The layer assignment restricts the freedom of choice for the crossing minimization algorithm drastically, and there may be large differences between the number of crossings for different layer assignments of one graph. Also, the generalizations of the Sugiyama algorithm for the mixed case have to assign layers to nodes with no directed adjacent edges. This only works when there is a low number of them, but if the directed part of the mixed graph is only small, the results are not satisfying and the layer assignment to the nodes seems artificial.

We propose in this work a different drawing strategy for upward drawings of directed graphs which is based on the concept of *upward planarity*. A directed graph is *upward planar* if it can be drawn upward without edge crossings. Our strategy consists of two phases. In the first phase, we make the input graph upward planar by replacing edge crossings by dummy nodes. We call the result of this phase *upward planarization*. In the second phase, an upward planar drawing of the upward planarization is generated and the dummy nodes are discarded.

A similar strategy has been applied very successfully in the area of drawing undirected graphs. The most popular algorithms based on this strategy are perhaps the graph drawing algorithms descending from the GIOTTO approach [2] [20] for orthogonal drawings. In [8] we showed recently how to extend the GIOTTO approach to mixed upward planar drawings of *mixed upward planar graphs*, i.e., mixed graphs which have a planar drawing in which the directed part of the graph is drawn upward. The above strategy can also be applied to this algorithm, the first phase then consisting of finding a mixed upward planarization.

In the remainder of this paper, we concentrate on the first phase of the strategy, see [7] for a survey on graph drawing algorithms for upward planar graphs. We give an efficient heuristics that computes a high quality upward planarization of a directed graph. We concentrate on a heuristical approach, since the upward planarity test problem is already NP-complete. This is the first time that this problem is studied; work on planarization has been restricted to the undirected case until now. We give also a generalization of our algorithm for mixed graphs.

We want to emphasize that the GIOTTO framework above is only one possible application of the new algorithm. It probably deserves also attention as a stand-alone product which might be applicable in other environments.

The rest of the paper is organized as follows. Section 2 gives the formal definitions of the upward and the mixed upward planarization problem. In Section 3,

we present an algorithm which solves the upward planarization problem. Finally, we show how the results of Section 3 can be generalized to the mixed case.

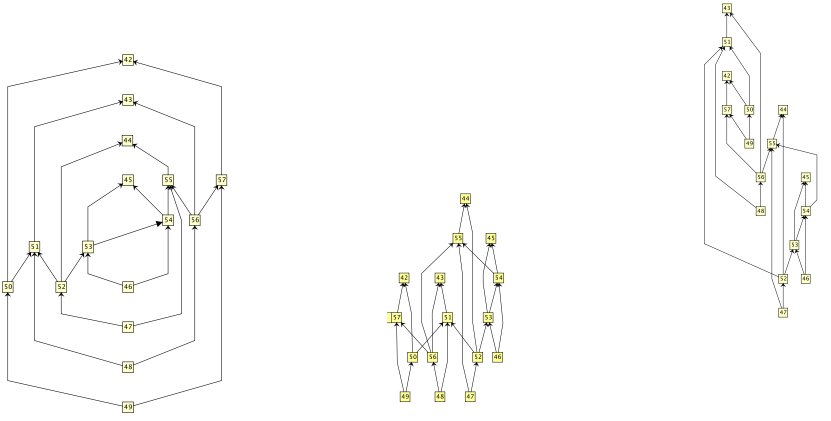


Fig. 1. A comparison of the Sugiyama-approach (seven crossings) and our new method (two crossings) applied to an upward planar graph (left figure).

2 Upward and Mixed Upward Planarization Problem

A drawing of a graph (digraph) is a mapping of its nodes to points in the plane and of its edges to open jordan curves. A graph (digraph) is *planar* if it has a drawing where no two edges have a common point. An *upward drawing* of a digraph is a drawing such that all the edges are represented by curves monotonically increasing in the vertical direction. A digraph is *upward planar* if it has a drawing which is upward and planar at the same time. Please note that there are graphs which have an upward drawing and also have a planar drawing, but do not have an upward planar drawing. An *embedding* of a graph is defined as a *cyclic ordering* of the adjacent edges of each vertex of the graph. An embedding is *planar* if there is a planar drawing of the graph which preserves this ordering. An *upward embedding* of a graph is a *linear ordering* of the adjacent edges of each vertex of the graph in which the incoming and outgoing edges form an interval. An upward embedding is *planar* if there is an upward planar drawing of the graph which preserves the corresponding ordering. Preserving the ordering means that the linear ordering is equivalent to the ordering that can be obtained by ordering the edges according to the angle they form with

a ray leaving the vertex in direction of the negative x-axis. We assume in the remainder of the paper that graphs have no multiple edges and selfloops.

Given a directed graph $G = (V, E)$, the graph $G' = (V \cup V', E')$ is an *upward planarization* of G with crossing number $|V'|$ if and only if

- G' is upward planar,
- $deg(v) = 4$ for all $v \in V'$, and
- $\forall e = (v, w) \in E$, there is a path $p(e) = (v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$ in G' with $v = v_0$, $w = v_n$ and $v_i \in V', 0 < i < n$. Every edge in E' is contained in such a path, and two paths have no edge in common.

A mixed graph is a three-tuple $G = (V, E_d, E_u) \subseteq (V, V \times V, V \times V)$, where V is the set of vertices, E_d is the set of directed edges and E_u is the set of undirected edges.

The mixed graph $G = (V, E_d, E_u)$ is *mixed upward planar* if there is a planar drawing of G where each edge in E_d is represented by a curve monotonically increasing in the vertical direction.

A mixed upward embedding is *planar* if there is a mixed upward planar drawing of the graph which preserves the corresponding ordering.

Determining for a (mixed) graph G a (mixed) upward planarized graph is called the *(mixed) upward planarization problem*. Determining for a (mixed) graph G a (mixed) upward planarized graph with minimal crossing number is called the *(mixed) upward crossing minimal problem*.

Because determining whether a graph has an upward embedding is a special case of the upward crossing minimization problem and the mixed case is a special case of the directed case, it follows that:

Corollary 1 ([10]). *The upward crossing minimization and the mixed upward crossing minimization problem are NP-hard.*

3 Upward Planarization

In this section, we propose an algorithmic framework for the upward crossing minimization problem. This framework is derived from techniques for the planarization of undirected graphs, see i.e. [7].

The framework consists of three parts:

1. Construct upward planar subgraph.
2. Determine upward embbeding of this subgraph.
3. Insert edges not contained in the subgraph, one by one.

In the first step, a subgraph of the input graph is calculated which is upward planar. For this subgraph, an upward embedding is determined in the second step. Of course, these two steps are only conceptually separated and can be combined to one step. Note that finding a maximum upward planar subgraph, i.e., finding an upward planar subgraph with the maximum number of edges, is NP-hard. In the third step, the edges which are not part of the upward planar

subgraph are inserted incrementally into the embedding. Additionally, we can perform some local optimizations on the resulting planarization to improve the quality of it.

3.1 Maximum Upward Planar Subgraph

The maximum upward planar subgraph problem can be stated as follows: Given a directed graph $G = (V, E)$. Find $E' \subseteq E$ such that the directed graph $G = (V, E)$ is upward planar with maximum number of edges.

The maximum planar subgraph problem is a related problem and a lot of algorithms have been proposed for its solution [11],[12],[16],[15],[17]. All of them, except [15], which can also compute the optimal solution when no time limit is specified, are heuristics, since the problem is NP-complete. Cimikowski [5] compared some of them empirically. In his comparison, the algorithm of Jünger and Mutzel(JM)[15] performed best in solution quality, followed by the algorithm of Goldschmidt and Takvorian (GT)[11]. The fastest algorithm was the one based on PQ-trees[16], but its performance in terms of the solution quality was significantly lower than JM and GT. Resende and Ribero give in [17] a randomized formulation of GT and show on the same test set as [5] that their formulation achieves better results with the same running time performance, except for one family of graphs where JM performs better.

However, the algorithm of GT is much easier to implement in contrast to the algorithm of JM. JM is a branch and cut algorithm and is, therefore, based on sophisticated algorithms for linear programming.

Because of its performance and its implementation issues, we use GT as a starting point. In the next section, we review the GT algorithm and show in the following section how it can be modified to calculate upward planar embeddings.

3.2 The Goldschmidt/Takvorian Planarization Algorithm

In this section, we review the main components of GT, the two-phase heuristic of Goldschmidt and Takvorian[11]. Our description follows the one in [17]. The first phase of GT consists in devising an ordering Π of the set of vertices of V of the input graph G. This ordering should possibly infer a Hamiltonian path. The vertices of G are placed on a vertical line according to the ordering Π obtained in the first phase, such that as many edges as possible between adjacent vertices can also be placed on the line. All other edges are drawn as arcs either right or left of the line.

The second phase of GT partitions the edge set E of G into subsets $\mathcal{L}$ (left of the line), $\mathcal{R}$ (right of the line), and $\mathcal{B}$ (the remainder) in such a way that $|\mathcal{L}+\mathcal{R}|$ is large (ideally maximum) and that no two edges both in $\mathcal{L}$ or both in $\mathcal{R}$ cross with respect to the sequence Π devised in the first phase.

Let $\pi(v)$ denote the relative position of vertex $v \in V$ within vertex sequence Π. Furthermore, let $e_1 = (a, b)$ and $e_2 = (c, d)$ be two edges of G, such that, without loss of generality, $\pi(a) < \pi(b)$ and $\pi(c) < \pi(d)$. These edges are said

to *cross* if, with respect to sequence Π, $\pi(a) < \pi(c) < \pi(b) < \pi(d)$ or $\pi(c) < \pi(a) < \pi(d) < \pi(b)$.

The *conflict graph* has a vertex for every edge in G and two vertices are adjacent if the corresponding edges cross with respect to Π. It follows directly from its definition that the conflict graph is an *overlap graph*, i.e. a graph whose vertices can be represented as intervals, and two vertices are adjacent if and only if the corresponding intervals intersect but none of the two is contained by the other.

A induced bipartite subgraph of the conflict graph represents a valid assignment of the edges in G to the sets $\mathcal{L}$,$\mathcal{R}$ and $\mathcal{B}$. Since finding a maximal induced bipartite subgraph is NP-complete, even for overlap graphs, GT uses a heuristics. This heuristic calculates two disjoint independent sets of the conflict graph which, together, are a bipartite subgraph of the conflict graph.

A maximum independent set of an overlap graph can be calculated in time $O(NM)$, where N is the number of different interval endpoints and M is the number of edges in the overlap graph by the algorithm of Asano, Imai and Mukaiyama[1]. In our setting, $N < n$ and $M < m^2$, which leads to a running time of $O(nm^2)$.

3.3 The Direct Version of the GT Algorithm

We now present our variant of the GT Algorithm for planar upward subgraph calculation. In order to change the GT algorithm to get a upward planar subgraph, we have to modify the first step of GT, the construction of the vertex order. The vertex order must ensure that no directed edge has a target vertex which is in the order before the source vertex. This is achieved by using algorithm *vertex order* as a first phase of GT. We call this variant directed GT or shorter DGT to distinguish it from the original formulation. The algorithm *vertex order* is a modification of the algorithm [11] and ensures this. It is a variation of a topological sorting algorithm, and constructs the ordering incrementally. Assume that vertex v is the vertex chosen in the last step. The algorithm chooses a vertex in the next step which is adjacent to v, but which is not the successor of a unchosen vertex. If this is not possible, it takes a vertex of minimal degree which, additionally, is not the successor of a unchosen vertex. As the first vertex, it chooses a vertex with no incoming edge with minimal degree.

Lemma 1. *Let G be a directed graph. If the vertex order Π in the first phase of the GT algorithm is a topological ordering of G, the result of GT is a upward planar subgraph of G.*

Lemma 2. *The vertex order calculated by algorithm* vertex order *is a topological ordering of G.*

From the sets $\mathcal{L}$ and $\mathcal{R}$ and the permutation Π, we can now easily obtain the upward planar embedding. The details are omitted because of space limitations. We conclude the section with the following theorem:

Algorithm 1: vertex order

Input: A directed graph $G = (V, E)$
Output: A permutation Π on the vertices
Select v_1 from G with minimal degree;
$\mathcal{V} = V \setminus \{v_1\}$;
G_1 = directed graph induced on G by $\mathcal{V}$;
for $k = 2, \ldots, |V|$ **do**
  $\mathcal{U} = \{v \in \mathcal{V} | v$ ist not target of an directed edge in $G_k\}$;
  if v_{k-1} *is connected to a vertex in* $\mathcal{U}$ **then**
    select v_k as vertex with minimal degree in $\mathcal{U}$
  else
    select v_k as vertex in $\mathcal{U}$ adjacent to v_{k-1} with minimal degree in G_{k-1}
  end
  $\mathcal{V} = \mathcal{V} \setminus v_k$;
  G_k = directed graph induced on G by $\mathcal{V}$;
end
return $\Pi = (v_1, v_2, \ldots, v_{|V|})$

Theorem 1. *Algorithm DGT computes an upward planar subgraph, together with an upward planar embedding of this subgraph, in time* $O(nm^2)$.

3.4 Edge Insertion

There is an interesting difference between the insertion of directed and undirected edges. In the undirected case, the edges which are not part of the planar subgraph in the first step can be inserted independently of each other. This is different in the directed case. Here, we cannot insert an edge into the drawing without looking at the remaining edges which have to be inserted later. The reason for this is that introducing dummy nodes in the graph introduces changes in the ordering of the vertices of the graph. This may introduce directed cycles if an edge is added later.

Assume that the dashed edges have to be inserted in Fig. 2(a), and we start by inserting edge $(5, 9)$. When we do not work carefully and insert edge $(5, 9)$ as in Fig. 2(b), we produce a crossing C with edge $(1, 3)$ and some new edges, where C is involved. Then, it is no longer possible to introduce edge $(3, 4)$ without destroying the upwardness property because of the new directed cycle $5 - C - 3 - 4 - 5$.

We call a vertex with indegree 0 a *source*, and a vertex with outdegree 0 a *sink*. A directed graph is called an *s-t graph* if it has exactly one sink and one source. We first restrict ourselves to s-t graphs. We show later how we can remove this restriction.

As shown above, we have to avoid cycles when we insert edges. We avoid this by *layering* the graph. A valid layering l of a directed graph $G = (V, E)$ is a mapping of V to integers such that $l(v) > l(u)$ for each edge $(u, v) \in E$. We then construct a routing graph R. The routing graph contains, for each face f and for each layer that f spans, a vertex. Two vertices laying in neighbored

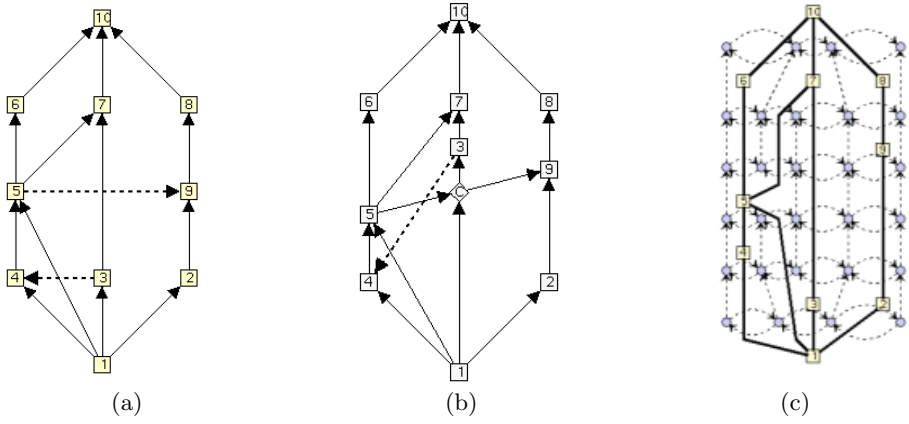


Fig. 2. Edge insertion: Critical configuration and the routing graph

layers and representing the same face are connected by a directed edge of weight 0 in increasing layer order. Additionaly, two vertices at the same layer i of adjacent faces are connected by an edge of weight 1 if the source vertex of an edge separting this two faces is less than or equal to i and the layer of the target node is greater than i.

In this graph, there are no edges in decreasing layer order. Edges of weight 1 represent one crossing. A shortest path in the routing graph represents, therefore, an insertion of an edge with minimal number of crossing with respect to the given layering. Figure 2(c) shows an example for a routing graph.

Let $s(f)$, resp. $t(f)$, denote the source, resp. sink, of a face f. Note that in a s-t graph, every face has exactly one source and one sink. Furthermore, $lf(e)$, resp. $rf(e)$, denotes the face on the left, resp., right side of e. We consider the outer face as two faces, the *left-outer face* and the *right-outer face*. The left-outer face denotes the left part of the outer face, the right-outer face the right part. The algorithm *edge insertion* summarizes the construction.

Lemma 3. *The graph G' calculated in* edge insertion *is an s-t graph and upward planar.*

Proof. In the edge insertion step, we do not decrease the indegree or the outdegree of any vertex existing already in the input graph. Therefore, we only have to show that none of the inserted vertices is a sink or a source. But this is true, since each of these vertices has indegree two and outdegree two. G' is upward planar, since there are no crossings and the layering is conserved.

Lemma 4. *The graph $(V', E' \cup F \setminus e)$ is acyclic.*

Proof. Assume that there is a directed cycle. Each inserted vertex w_i is induced by an edge of weight 1 in the routing graph which connects two face vertices laying in the same layer. Assign this layer to node w_i. To each vertex a layer is

Algorithm 2: edge insertion

Input: Embedded upw. planar s-t graph $G = (V, E)$, $F \subseteq V \times V$, $e = (a, b) \in F$
Output: Embedded upward planarized s-t graph G' of $G = (V, E \cup e)$
calculate faces from embedding;
determine valid layering l of $(V, E \cup F)$;
for *every face* f *of* G **do**
  create vertices $v(f, i)$ for $l(s(f)) \leq i < l(t(f))$;
  create edge of weight 0 from $v(f, i-1)$ to $v(f, i)$ for $l(s(f)) < i < l(t(f))$
end
for *every edge* $e' = (c, d)$ *of* E **do**
  create edge of weight 1 from $v(lf(e'), i)$ to $v(rf(e'), i)$ and the reverse for $l(c) \leq i < l(d)$;
end
create vertex $v(a)$ and $v(b)$ representing a resp. b;
Insert edge of weight 0 from $v(a)$ to $v(f, l(a))$ if f is adjacent to a and such a vertex exists;
Insert edge of weight 0 from $v(b)$ to $v(f, l(b)-1)$ if f is adjacent to b and such a vertex existst;
Calculate shortest path p from $v(a)$ to $v(b)$ in R;
$E' = E$,$V' = V$,$G' = (V', E')$; Let $e_0, \ldots, e_n$ be the edges of weight 1 in p;
Subdivide e_i with a vertex w_i, $0 < i < n$ in G';
Add an edge between a and w_0, w_n and b, and w_i and w_{i+1} in E';
return G'

assigned, and there are no edges which point in decreasing layer order. Thus, the cycle can only contain vertices in the same layer. These can only be vertices w_i by the construction of the layering. But, from this fact follows that the shortest path had a directed cycle which is a contradiction.

Lemma 5. *Algorithm* edge insertion *has time complexity* $O(|V|^2)$,

Proof. The faces of the graph can be computed in linear time from the embedding. A valid layering with a minimal number of layers can also be computed in linear time using a topological sorting. The maximum number of layers is linear, since a topological sorting is an upper bound for the number of layers. Hence, the number of vertices in the routing graph is $O(|V|^2)$ and, since each vertex has constant degree, the total size of the routing graph is $O(|V|^2)$. Because the maximum cost of an edge is 1, we can use Dials shortest path algorithm [6] which has linear running time in this case. The insertion of the edge can clearly be done in linear time.

The following theorem summarizes the lemmas above.

Theorem 2. *Algorithm* edge insertion *inserts one edge in an embedded upward planar s-t graph* $G = (V, E)$ *in time* $O(|V|^2)$ *without introducing cycles with a set of not yet inserted edges. The planarized graph is an s-t graph.*

3.5 The Complete Algorithm

Algorithm *upward planarization* contains a description of the algorithm. Note that if the input graph for the second phase is not an s-t graph, we augment it to a planar upward s-t graph, see [3] for a linear time algorithm. Edges in the routing graph representing an edge added in the augmenting step are assigned weight 0, because they do not introduce a real crossing. After the routing, the augmenting edges are removed. Note that the augmentation does not affect the worst-case running time of the algorithm, since the number of edges in the graph remains linear in the number of nodes.

Algorithm 3: upward-planarization

```
calculate embedded mixed upward planar subgraph with DGT;
augment subgraph to an s-t graph;
for Each directed edge not in subgraph do
    call algorithm directed edge insertion;
end
remove edges inserted in augmentation process;
for Each undirected edge not in subgraph do
    call algorithm undirected edge insertion;
end
```

From the discussion above, we derive the following theorem:

Theorem 3. *Let $G = (V, E)$ be a directed graph. Algorithm* upward-planarization *creates an embedded upward planarized graph of G in time $O(|V||E|^2 + (|V| + c)^2|E|)$, where c is the number of crossings of the planarized graph. When G is sparse, i.e. $|E| = O(|V|)$, the algorithm* upward-planarization *runs in time $O(|V|^3)$.*

However, the time bound in the theorem above is very pessimistic. In our experiments, the running time of the algorithm is reasonable, even for larger graphs.

3.6 Rerouting

In this section, we present a local optimization method for an upward planarization. This method removes a path representing an edge from the planarization. Then, we augment the resulting graph to an s-t-graph. For the removed edge, we try to find a better routing. If we succeed, we change the planarization according to the new routing. Otherwise, we do not change the planarization. In any case, the augmented edges are removed. We stop this local optimization when either no further improvements are made or a maximal number of iterations has been performed.

There is one advantage of rerouting with respect to routing: we do not have to observe that some edges will be inserted later and might cause cycles. We, therefore, do not need the layering concept here.

We now define the routing graph for the rerouting of an edge from a to b. Note that we assume that the outer face is split into two faces, one for each side. The routing graph contains a start node s and a end node t. Each face f, has for each adjacent edge e, a node $v(f, e)$. We call these nodes *edge nodes*. It has, additionaly, two nodes $l(f)$ and $r(f)$, except for the left outer face which has only node $r(f)$, and the right outer face which has only node $l(f)$. We call those nodes *face nodes*. For each edge e, the two corresponding edge nodes are connected by a directed edge of weight 1, except for edges introduced in the augmentation step. In this case, the weight is 0. If an edge e_1 is directly below an edge e_2 on a face f, the routing graph contains a directed edge $v(f, e_1)$ to $v(f, e_2)$ with weight 0. For a face f and a node v representing an edge on the left, resp. right, side of f, there is a directed edge of cost 0 from v to $r(f)$, resp. $l(f)$. There is a directed edge of weight 0 from the start node s to each node in the routing graph representing an outgoing edge of a. There is a directed edge of weight 0 from each node in the routing graph representing an incoming edge of b to the end node t.

Lemma 6. *The routing graph has linear size with respect to the planarization.*

Proof. The number of nodes in the routing graph is $2|E| + 2|F| + 2$, and it is therefore, linear in the number of nodes of the planarization by Euler's formula. Each edge node in the routing graph is adjacent to at most eight edges. Because face nodes are only connected to edge nodes in the routing graph but not to other face nodes, the number of edges is linear in the number of nodes in the routing graph.

The methods and observations on the subsection 'Edge Insertion' apply also to the rerouting, only the routing graph is different. Therefore, we can conclude with the following theorem:

Theorem 4. *Rerouting of an edge $e \in E$ of a graph $G = (V, E)$ in a planarization $G'(V', E')$ of G can be done in time $O(|V'|)$.*

4 Mixed Upward Planarization

In this section we show how the concepts in the previous sections can be extended to the mixed case, i.e., the input graph is a mixed graph.

For the mixed planar subgraph calculation, we also use the GT algorithm. As in the upward case, we only have to take care of the vertex ordering. We use a modified version of the *vertex-ordering* algorithm for this, which ignores the direction of the undirected edges. The changes are omitted here, because they add no major additional insight in the algorithm. They can be found in [9].

It is clear that the directed edges are more restrictive than the undirected. So, it is intuitive to prefer the directed edges when computing the planar subgraph. One variant of the planar subgraph algorithm takes this aspect into account. It extends the GT approach by assigning different weights to the directed and

undirected edges and then optimizes over the weighted sum of the edges [1]. The actual choice of the weights depends on the application, as well as on the class of graphs considered and is the subject of further research.

The upward edge insertion algorithm can be extended to the mixed case by directing the undirected edges in G temporarly according to the ordering in GT. We then insert the directed edges iteratively in the graph as described above. Next, we undirect the temporarily directed edges. Finally, we insert the undirected edges by an standard edge insertion algorithm for undirected graphs [7].

Theorem 5. *Let $G = (V, E_d, E_u)$ be a mixed graph. Algorithm* mixed-upward-planarization *creates an embedded mixed upward planarized graph of G in time $O(|V|(|E_d| + |E_u|)^2 + (|V| + c)^2|E_d| + (|V| + c)|E_u|)$, where c is the number of crossings in the planarized graph.*

References

1. T. Asano, H. Imai, and A. Mukaiyama, *Finding a Maximum Weight Independent Set of a Circle Graph.* IEICE Transactions, E74, 1991, pp.681-683.
2. C. Batini, E. Nardelli, and R. Tamassia, *A Layout Algorithm for Data-Flow Diagrams*, IEEE Trans. Softw. Eng., SE-12(4), 1986, pp. 538-546
3. P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino, *Upward drawings of triconnected digraphs*, Algorithmica, 6(12), 1996, p. 476-497.
4. G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language*, Addison-Weslay, 1999.
5. Cimikowski, R., *An analysis of heuristics for the maximum planar subgraph problem*, in Proceedings of the 6th ACM-SIAM Symposium of Discrete Algorithms, 1995, pp. 322-331.
6. R. Dial,*Algorithm 360: Shortest path forest with topological ordering*, Communications of ACM, 12, 1969, pp. 632-633
7. G. Di Battista and P. Eades and R. Tamassia and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999.
8. Eiglsperger, M., Foessmeier, U., Kaufmann, M., *Orthogonal Graph Drawing with Constraints*, Proc. 11th Symposium on Discrete Algorithms (SODA'00), pp. 3-11, ACM-SIAM, 2000.
9. F. Eppinger, *Kombinatorische Einbettung teilgerichteter Graphen*, Diplomarbeit, Universität Tübingen, 2001.
10. A. Garg and R. Tamassia. On the computational complexity of upward and rectiliniar planarity testing. In R. Tamassia and I. G. Tollis, editors, *Graph Drawing (Proc. GD'94)*, LLNCS 894, pages 286-297. Springer Verlag, 1995.
11. Goldschmidt, O., Takvorian, A., *An efficient graph planarization two-phase heuristic*, Networks, 24 (1994), pp. 69-73.
12. R. Jayakumar, K. Thulasiraman, and M.N.S. Swamy, $O(n^2)$ *Algorithms for graph planarization*, IEEE Trans. on CAD 8 (3),257-267, 1989
13. M. Jünger, S. Leipert and P. Mutzel, Level Planarity Testing in Linear Time, In S.H. Whiteside, editor, *Graph Drawing (Proc. GD'98)*, LLNCS 1547, pages 224-237. Springer Verlag, 1999.
14. M. Jünger and S. Leipert, Level Planar Embedding in Linear Time, In J. Kratochvil, editor, *Graph Drawing (Proc. GD'99)*, volume 1547 of Lecture Notes Comput. Sci., pages 72-81. Springer Verlag, 2000.

15. M. Jünger and P. Mutzel, *Solving the maximum weight planar subgraph problem by Branch & Cut*, In Proc. of the 3rd conference on integer programming and combinatorial optimization (IPCO), pp. 479–492, 1993
16. G. Kant, *An $O(n^2)$ maximal planarization algorithm based on PQ-trees.* Tech. Rep. RUU-CS-92-03, CS Dept., Univ. Utrecht, Netherlands, Jan., 1992
17. M. Resende and C. Ribeiro, *A GRASP for graph planarization*, Networks, 29 (1997), pp. 173-189
18. J. Seemann, *Extending the Sugiyama Algorithm for Drawing UML Class Diagrams*, In G. Di Battista, editor, *Graph Drawing (Proc. GD'97)*, volume 1353 of Lecture Notes Comput. Sci., pages 415-424. Springer Verlag, 1998.
19. Sugiyama, K., Tagawa, S. and Toda, M. 1981. *Methods for Visual Understanding of Hierarchical Systems*, IEEE Trans. Syst. Man Cybern., SMC-11, no.2, pp. 109-125.
20. R. Tamassia, *On Embedding a Graph in the Grid with the Minimum Number of Bends*, SIAM J. Comput., 16(3), 1987, pp. 421-444.

A Linear-Time Algorithm for Computing Inversion Distance between Signed Permutations with an Experimental Study

David A. Bader[1,⋆], Bernard M.E. Moret[2,⋆⋆], and Mi Yan[1,⋆⋆⋆]

[1] Department of Electrical and Computer Engineering, University of New Mexico, Albuquerque, NM 87131 USA. {dbader,miyan}@eece.unm.edu
[2] Department of Computer Science, University of New Mexico, Albuquerque, NM 87131 USA. moret@cs.unm.edu

Abstract. Hannenhalli and Pevzner gave the first polynomial-time algorithm for computing the inversion distance between two signed permutations, as part of the larger task of determining the shortest sequence of inversions needed to transform one permutation into the other. Their algorithm (restricted to distance calculation) proceeds in two stages: in the first stage, the overlap graph induced by the permutation is decomposed into connected components, then in the second stage certain graph structures (hurdles and others) are identified. Berman and Hannenhalli avoided the explicit computation of the overlap graph and gave an $O(n\alpha(n))$ algorithm, based on a Union-Find structure, to find its connected components, where α is the inverse Ackerman function. Since for all practical purposes $\alpha(n)$ is a constant no larger than four, this algorithm has been the fastest practical algorithm to date. In this paper, we present a new linear-time algorithm for computing the connected components, which is more efficient than that of Berman and Hannenhalli in both theory and practice. Our algorithm uses only a stack and is very easy to implement. We give the results of computational experiments over a large range of permutation pairs produced through simulated evolution; our experiments show a speed-up by a factor of 2 to 5 in the computation of the connected components and by a factor of 1.3 to 2 in the overall distance computation.

1 Introduction

Some organisms have a single chromosome or contain single-chromosome organelles (such as mitochondria or chloroplasts), the evolution of which is largely independent on the evolution of the nuclear genome. Given a particular strand from a single chromosome, whether linear or circular, we can infer the ordering and directionality of the genes, thus representing each chromosome

⋆ Supported in part by NSF Grants CAREER 00-93039, NSF ITR 00-81404 and NSF DEB 99-10123 and by DOE SUNAPP AX-3006 and DOE-CSRI-14968.
⋆⋆ Supported in part by NSF Grant ITR 00-81404.
⋆⋆⋆ Supported by the UNM Albuquerque High Performance Computing Center.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 365-376, 2001.

by an ordering of oriented genes. In many cases, the evolutionary process that operates on such single-chromosome organisms consists mostly of inversions of portions of the chromosome; this finding has led many biologists to reconstruct phylogenies based on gene orders, using as a measure of evolutionary distance between two genomes the inversion distance, i.e., the smallest number of inversions needed to transform one signed permutation into the other [18,19,20].

Both inversion distance and the closely related transposition distance are difficult computational problems that have been studied intensively over the last five years [1,3,4,5,7,8,11,12]. Finding the inversion distance between unsigned permutations is NP-hard [7], but with signed ones, it can be done in polynomial time [11]. The fastest published algorithm for the computation of inversion distance between two signed permutations has been that of Berman and Hannenhalli [5], which uses a Union-Find data structure and runs in $O(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function. (The later KST algorithm [12] reduces the time needed to compute the shortest sequence of inversions, but uses the same algorithm for computing the length of that sequence.) We have found only two implementations on the web, both designed to compute the shortest sequence of inversions as well as its length; one, due to Hannenhalli [10], implements his first algorithm [11], which runs in quadratic time when computing distances, while the other, a Java applet written by Mantin [13], a student of Shamir, implements the KST algorithm [12], but uses an explicit representation of the overlap graph and thus also takes quadratic time.

We present a simple and practical, worst-case linear-time algorithm to compute the connected components of the overlap graph, which results in a simple linear-time algorithm for computing the inversion distance between two signed permutations. We also provide ample experimental evidence that our linear-time algorithm is efficient in practice as well as in theory: we coded it as well as the algorithm of Berman and Hannenhalli, using the best principles of algorithm engineering [14,16] to ensure that both implementations would be as efficient as possible, and compared their running times on a large range of instances generated through simulated evolution. (The two implemenations on the web are naturally far slower.)

The paper is organized as follows. We begin by recalling some definitions, briefly review past work on sorting by reversals, then introduce the concepts that we will need in our algorithm, including the fundamental theorem that makes it possible. We then describe and analyze our algorithm, discuss our experimental setup, present and comment on our results, and briefly mention an application of our distance computation in a whole-genome phylogeny study.

2 Inversions on Signed Permutations

We assume a fixed set of genes $\{g_1, g_2, \ldots, g_n\}$. Each genome is then an ordering (circular or linear) of these genes, each gene given with an orientation that is either positive (g_i) or negative ($-g_i$). The ordering $g_1, g_2, \ldots, g_n$, whether linear or circular, is considered equivalent to that obtained by considering the complementary strand, i.e., the ordering $-g_n, -g_{n-1}, \ldots, -g_1$.

Let G be the genome with signed ordering (linear or circular) $g_1, g_2, \ldots, g_n$. An *inversion* between indices i and j, for $i \leq j$, produces the genome with linear ordering

$$g_1, g_2, \ldots, g_{i-1}, -g_j, -g_{j-1}, \ldots, -g_i, g_{j+1}, \ldots, g_n$$

If we have $j < i$, we can still apply an inversion to a circular (but not linear) genome by rotating the circular ordering until the two indices are in the proper relationship—recall that we consider all rotations of the complete circular ordering of a circular genome as equivalent.

The inversion distance between two genomes (two signed permutations of the same set) is then the minimum number of inversions that must be applied to one genome in order to produce the other. (This measure is easily seen to be a true metric.) Computing the shortest sequence of inversions that gives rise to this distance is also known as *sorting by reversals*—we shall shortly see why it can be regarded as a sorting problem.

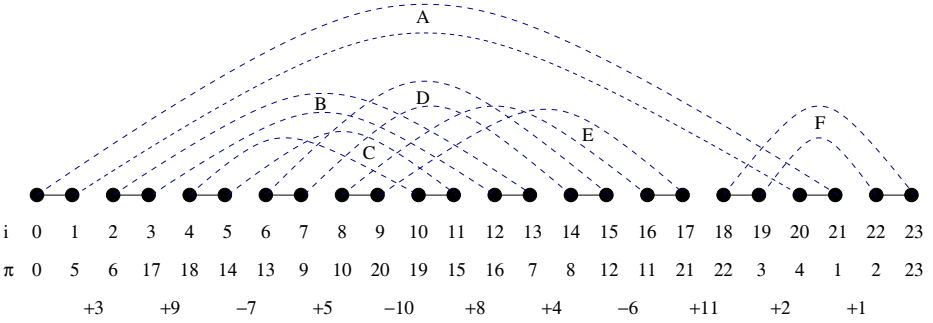
3 Previous Work

Bafna and Pevzner introduced the cycle graph of a permutation [2], thereby providing the basic data structure for inversion distance computations. Hannenhalli and Pevzner then developed the basic theory for expressing the inversion distance in easily computable terms (number of breakpoints minus number of cycles plus number of hurdles plus a correction factor for a fortress [2, 22]—hurdles and fortresses are easily detectable from a connected component analysis). They also gave the first polynomial-time algorithm for sorting signed permutations by reversals [11]; they also proposed a $O(n^4)$ implementation of their algorithm [10], which runs in quadratic time when restricted to distance computation. Their algorithm requires the computation of the connected components of the overlap graph, which is the bottleneck for the distance computation. Berman and Hannenhalli later exploited some combinatorial properties of the cycle graph to give a $O(n\alpha(n))$ algorithm to compute the connected components, leading to a $O(n^2\alpha(n))$ implementation of the sorting algorithm [5]. (We will refer to this approach as the UF approach.) Algorithms for finding the connected components of interval graphs (a class of graphs that include the more specialized overlap graphs used in sorting by reversals) that run in linear time are known, but they use range minima and lowest common ancestor data structures and algorithms so that, in addition to being complex and hard to implement, they suffer from high overhead—high enough, in fact, that the UF approach would remain the faster solution in practice.

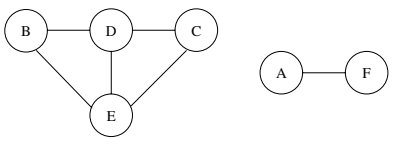
4 Overlap Graph and Forest

Given a signed permutation of $\{1, \ldots, n\}$, we transform it into an unsigned permutation π of $\{1, \ldots, 2n\}$ by substituting the ordered pair $(2x - 1, 2x)$ for the positive element x and the ordered pair $(2x, 2x - 1)$ for the negative elements

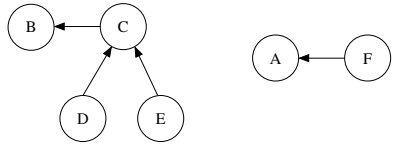
$-x$, then extend π to the set $\{0, 1, \ldots, 2n, 2n+1\}$ by setting $\pi(0) = 0$ and $\pi(2n+1) = 2n+1$. By convention, we assume that the two signed permutations for which we must compute a distance have been turned in this manner into unsigned permutations and then both permuted (by the same transformation) so that the first permutation becomes the linear ordering $(0, 1, \ldots, 2n, 2n+1)$; these manipulations do not affect the distance value. (This is the reason why transforming one permutation into the other can be viewed as sorting—we want to find out how many inversions are needed to produce the identity permutation from the given one.) We represent an extended unsigned permutation with an edge-colored graph, the *cycle graph* of the permutation. The graph has $2n+2$ vertices; for each i, $0 \leq i \leq n$, we join vertices $\pi(2i)$ and $\pi(2i+1)$ by a *gray* edge and vertices $2i$ and $2i+1$ by a *black* edge, as illustrated in Figure 1(a). The resulting graph consists of disjoint cycles in which edges alternate colors; we remove from it all 2-cycles (because these cycles correspond to portions of the permutation that are already sorted and cannot intersect with any other cycles). We say that



(a) the signed permutation, its unsigned extension, and its cycle graph (note that the gray edges appear as dashed arcs)



(b) the overlap graph



(c) the overlap forest

Fig. 1. The signed permutation $(+3, +9, -7, +5, -10, +8, +4, -6, +11, +2, +1)$ and its various representations

gray edges $(\pi(i), \pi(j))$ and $(\pi(k), \pi(t))$ *overlap* whenever the two intervals $[i, j]$ and $[k, t]$ overlap, but neither contains the other. Similarly, we say that cycles C_1 and C_2 *overlap* if there exist overlapping gray edges $e_1 \in C_1$ and $e_2 \in C_2$.

Definition 1. *The* overlap graph *of permutation π has one vertex for each cycle in the cycle graph and an edge between any two vertices that correspond to overlapping cycles.*

Figure 1 illustrates the concept. The *extent* of cycle C is the interval $[C.B, C.E]$, where we have $C.B = \min\{i \mid \pi(i) \in C\}$ and $C.E = \max\{i \mid \pi(i) \in C\}$. The extent of a set of cycles $\{C_1, \ldots, C_k\}$ is $[B, E]$, with $B = \min_{i=1}^{k} C_i.B$ and $E = \max_{i=1}^{k} C_i.E$. In Figure 1, the extent of cycle A is [0,21], that of cycle F is [18,23], and that of the set $\{A, F\}$ is [0,23].

No algorithm that actually builds the overlap graph can run in linear time, since that graph can be of quadratic size. Thus, our goal is to construct an *overlap forest* such that two vertices f and g belong to the same tree in the forest exactly when they belong to the same connected component in the overlap graph. An overlap forest (the composition of its trees is unique, but their structure is arbitrary) has exactly one tree per connected component of the overlap graph and is thus of linear size.

5 The Linear-Time Algorithm for Connected Components

Our algorithm for computing the connected components scans the permutation twice. The first scan sets up a trivial forest in which each node is its own tree, labelled with the beginning of its cycle. The second scan carries out an iterative refinement of this first forest, by adding edges and so merging trees in the forest; unlike a Union-Find, however, our algorithm does not attempt to maintain the trees within certain shape parameters.

Recall that a node in the overlap graph (or forest) corresponds to a cycle in the cycle graph. The extent $[f.B, f.E]$ of a node f of the overlap forest is the extent of the set of nodes in the subtree rooted at f. Let F_0 be the trivial forest set up in the first scan and assume that the algorithm has processed elements 0 through $j-1$ of the permutation, producing forest F_{j-1}. We construct F_j from F_{j-1} as follows. Let f be the cycle containing element j of the permutation. If j is the beginning of its own cycle f, then it must be the root of a single-node tree; otherwise, if f overlaps with another cycle g, then we add a new arc (g, f) and compute the combined extent of g and of the tree rooted at f. We say that a tree rooted at f is *active* at stage j whenever j lies properly within the extent of f; we shall store the extent of the active trees in a stack.

Figure 2 summarizes our algorithm for constructing the overlap forest; in the algorithm, *top* denotes the top element of the stack. The conversion of a forest of up-trees into connected component labels is accomplished in linear time by a simple sweep of the array, taking advantage of the fact that the parent of i must appear before i in the array.

Lemma 1. *At iteration i of Step (3) of the algorithm, if the tree rooted at* top *is active and i lies on cycle f and we have $f.B <$* top*$.B$, then there exists h in the tree rooted at* top *such that h overlaps with f.*

Proof. Since *top* is active, it must have been pushed onto the stack before the current iteration ($top.B < i$) and we must not have reached the end of *top*'s extent ($i < top.E$). Hence, i must be contained in *top*'s extent ($top.B < i < top.E$). Since i lies on the cycle f that begins before *top* ($f.B < top.B$), there must be an edge from cycle f that overlaps with *top*.

Input: permutation
Output: *parent*$[i]$, the parent of i in the overlap forest
Begin

1. scan the permutation, label each position i with $C[i].B$, and set up $[C[i].B, C[i].E]$
2. initialize empty stack
3. for $i \leftarrow 0$ to $2n+1$
   a) if $i = C[i].B$
      then push $C[i]$
   b) *extent* $\leftarrow C[i]$
      while $(top.B > C[i].B)$
         $extent.B \leftarrow \min\{extent.B, top.B\}$
         $extent.E \leftarrow \max\{extent.E, top.E\}$
         pop *top*
         $parent[top.B] \leftarrow C[i].B$
      endwhile
      $top.B \leftarrow \min\{extent.B, top.B\}$
      $top.E \leftarrow \max\{extent.E, top.E\}$
   c) if $i = top.E$
      then pop *top*
4. convert each tree into a labeling of its vertices

End

Fig. 2. Constructing the Interleaving Forest in Linear Time

Theorem 1. *The algorithm produces a forest in which each tree is composed of exactly those nodes that form a connected component.*

Proof. It suffices to show that, after each iteration, the trees in the forest correspond exactly to the connected components determined by the permutation values scanned up to that point. We prove this invariant by induction on the number of applications of Step (3) of the algorithm.

The base case is trivial: each tree of F_0 has a single node and no two nodes belong to the same connected component since we have not yet processed any element of the permutation.

Assume that the invariant holds after the $(i-1)$st iteration and let i lie on cycle f. We prove that the nodes of the tree containing i form the same set as the nodes of the connected component containing i—other trees and connected components are unaffected and so still obey the invariant.

– We prove that a node in the tree containing i must be in the same connected component as i.
  If we have $i = f.B$, then, as we remarked earlier, nothing changes in the overlap graph (and thus in the connected components); from Step (3), it is also clear that the forest remains unchanged, so that the invariant is preserved.
  On the other hand, if we have $i > f.B$, then at Step (3) the edge (top, f) will be added to the forest whenever $f.B < top.B$ holds. This edge will join the subtree rooted at f with that rooted at *top* into a single subtree. From

Lemma 1, we also know that, whenever $f.B < top.B$ holds, there must exist h in the tree rooted at top such that h and f overlap, so that edge (h, f) must belong to the overlap graph, thereby connecting the component containing f with that containing top and merging them into a single connected component, which maintains the invariant.
– We prove that a node in the same connected component as i must be in the tree containing i. Whenever (j, i) and (k, l), with $j < k < i < l$, are gray edges on cycles f and h respectively, then edge (f, h) must belong to the overlap graph built from the first i entries of the permutation. In such a case, our algorithm ensures that edge (h, f) belongs the overlap forest. Our conclusion follows.

Obviously, each step of the algorithm takes linear time, so the entire algorithm runs in worst-case linear time.

6 Experiments

Programs. The implementation due to Hannenhalli is very slow and implements the original method of Hannenhalli and Pevzner and not the faster one of Berman and Hannenhalli. The KST applet is very slow as well since it explicitly constructs the overlap graph; it is also written in Java which makes it difficult to compare with C code. For these reasons we wrote our own implementation of the Berman and Hannenhalli algorithm (just the part handling the distance computation) with a view to efficiency. Thus, we not only have an efficient implementation to compare to our linear-time algorithm, but also we have ensured that the two implementations are truly comparable because they share much of their code (hurdles, fortresses, breakpoints), were written by the same person, and used the same algorithmic engineering techniques.

Experimental Setup. We ran experiments on signed permutations of length 10, 20, 40, 80, 160, 320, and 640, in order to verify rate of growth as a function of the number of genes and also to cover the full range of biological applications. We generated groups of 3 signed permutations from the identity permutation using the evolutionary model of Nadeau and Taylor [17]; in this model, randomly chosen inversions are applied to the permutation at a node to generate the permutations labelling its children, repeating the process until all nodes have been assigned a permutation. The expected number of inversions per edge, r, is fixed in advance, reflecting assumptions about the evolutionary rate in the model. We use 5 evolutionary rates: 4, 16, 64, 256, and 1024 inversions per edge and generated 10 groups of 3-leaf trees—or 10 groups of 3 genomes each—at each of the 6 selected lengths. We also generated 10 groups of 3 random permutations (from a uniform distribution) at each length to provide an extreme test case. For each of these 36 test suites, we computed the 3 distances among the 3 genomes in each group 20,000 times in a tight loop, in order to provide accurate timing values for a single computation, then averaged the values over the 10 groups and computed the standard deviation. The computed inversion distances are

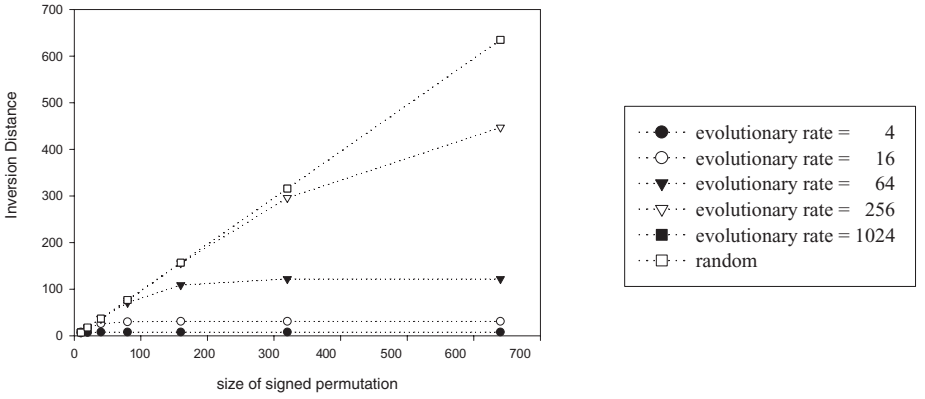


Fig. 3. The inversion distance as a function of the size of the signed permutation.

expected to be at most twice the evolutionary rate since there are two tree edges between each pair of genomes. Our linear algorithm exhibited very consistent behavior throughout, with standard deviations never exceeding 2% of the mean; the UF algorithm showed more variation for $r = 4$ and $r = 16$.

We ran all our tests on a 300MHz Pentium II with 16KB of L1 data cache, 16KB of L1 instruction cache, and 512KB of L2 cache running at half clockspeed; our codes were compiled under Linux with the GNU gcc compiler with options `-O3 -mpentiumpro`. Our code also runs on other systems and machines (e.g., Solaris and Microsoft), where we observed the same behavior.

Experimental Results. We present our results in four plots. The first two (Figure 4) show the actual running time of our linear-time algorithm for the computation of the inversion distance between two permutations as a function of the size of the permutation, with one plot for the computation of the connected components alone and the other for the complete distance computation. Each plot shows one curve each for the various evolutionary rates and one for the random permutations. We added a third plot showing the average inversion distance; note the very close correlation between the distance and the running time.

For small permutation sizes (10 or less), the L1 data cache holds all of the data without any cache misses, but, as the permutation size grows, the hit rate in the direct-mapped L1 cache steadily decreases until, for permutations of size 100 and larger, execution has slowed down to the speed of the L2 cache (a ratio of 2). From that point on, it is clear that the rate of growth is linear, as predicted. It is also clear that $r = 1024$ is as high a rate of evolution as we need to test, since the number of connected components and inversion distance are nearly indistinguishable from those of the random permutations (see the plot in Figure 3 that plots inversion distance as a function of the permutation size). The speed is remarkable: for a typical genome of 100 gene fragments (as found in chloroplast data, for instance [9]), well over 20,000 distance computations can be carried out every second on our rather slow workstation.

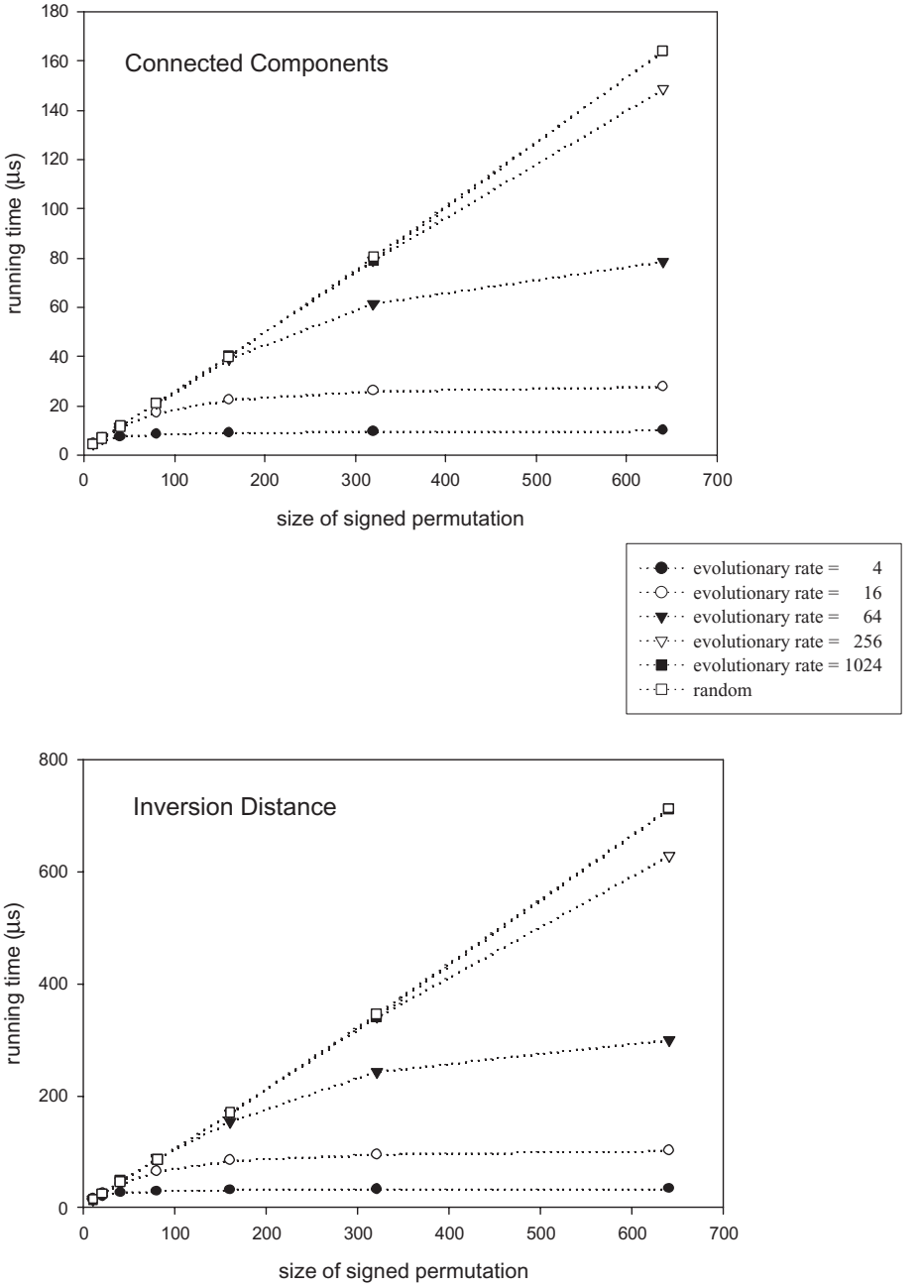


Fig. 4. The running time of our linear-time algorithms as a function of the size of the signed permutation.

Our second two plots (Figure 5) compare the speed of our linear-time algorithm and that of the UF approach. We plot speedup ratios, i.e., the ratio of the running time of the UF approach to that of our linear-time algorithm. Again, the first plot addresses only the connected components part of the computation, while the second captures the complete distance computation.

Since the two approaches use a different amount of memory and give rise to a very different pattern of addressing (and thus different cache conflicts), the ratios up to permutations of size 100 vary quite a bit as the size increases—reflecting a transition from the speed of the L1 cache to that of the L2 cache at different permutation sizes for the two algorithms. Beyond that point, however, the ratios stabilize and clearly demonstrate the gain of our algorithm, a gain that increases with increasing permutation size as well as with increasing evolutionary rate.

7 Concluding Remarks

We have presented a new, very simple, practical, linear-time algorithm for computing the inversion distance between two signed permutations, along with a detailed experimental study comparing the running time of our algorithm with that of Berman and Hannenhalli. Our code is available from the web page `www.cs.unm.edu/~moret/GRAPPA` under the terms of the GNU Public License (GPL); it has been tested under Linux (including the parallel version), FreeBSD, Solaris, and Windows NT. This code includes inversion distance as part of a much larger context, which provides means of reconstructing phylogenies based on gene order data. We found that using our inversion distance computation in lieu of the surrogate breakpoint distance (which was used by previous researchers in an attempt to speed up computation [6,21]) only slowed down the reconstruction algorithm by about 30%, enabling us to extend work on breakpoint analysis (as reported in [9,15]) to similar work on inversion phylogeny.

References

[1] V. Bafna and P. Pevzner. Sorting permutations by transpositions. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms*, pages 614–623, New York, January 1995. ACM Press.

[2] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science (FOCS93)*, pages 148–157. IEEE Press, 1993.

[3] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25:272–289, 1996.

[4] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11:224–240, 1998.

[5] P. Berman and S. Hannenhalli. Fast Sorting by Reversal. In D.S. Hirschberg and E.W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 168–185, Laguna Beach, CA, June 1996. Lecture Notes in Computer Science, **1075**, Springer-Verlag.

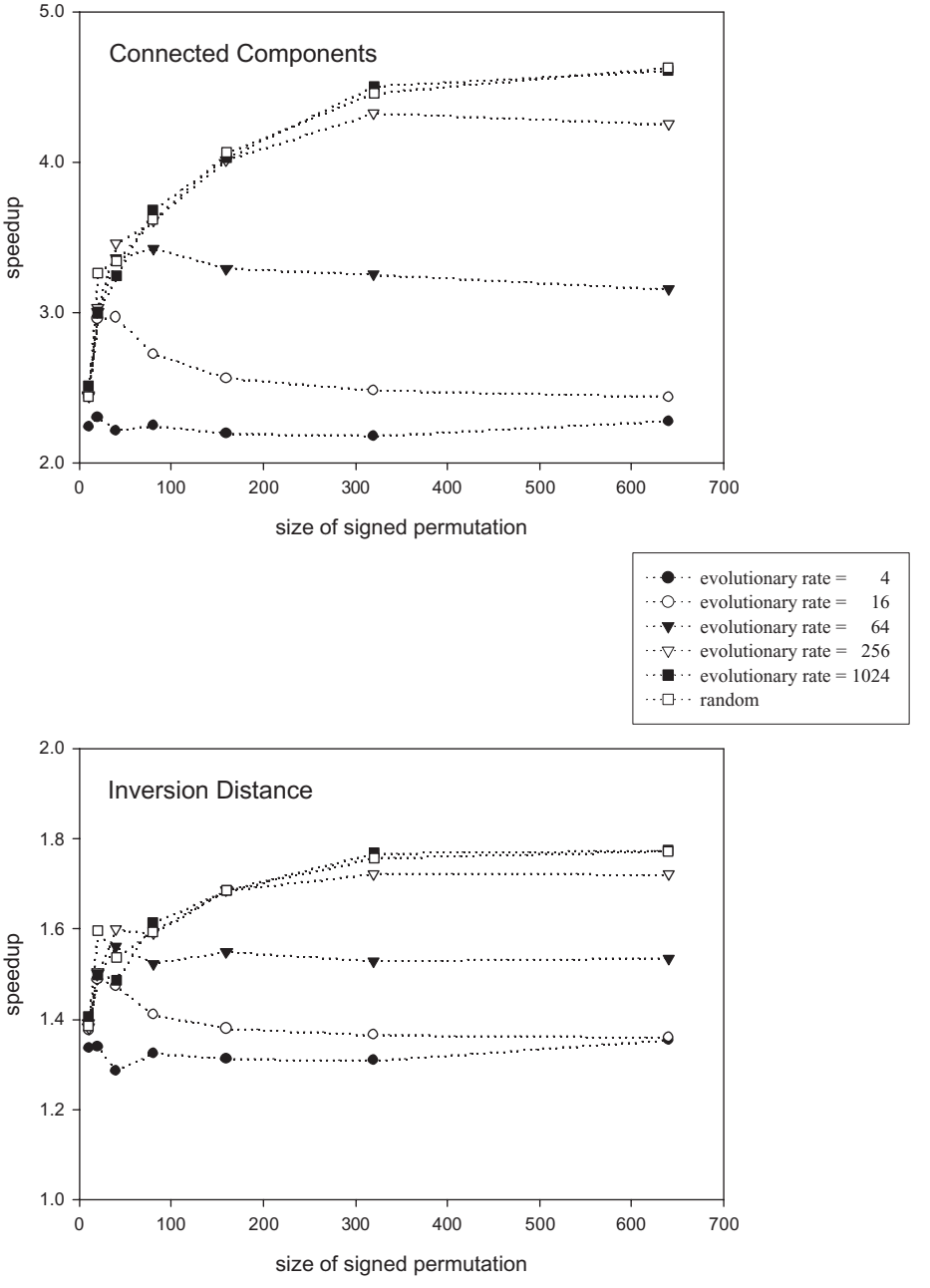


Fig. 5. The speedup of our linear-time algorithms over the UF approach as a function of the size of the signed permutation.

[6] M. Blanchette, G. Bourque, and D. Sankoff. Breakpoint phylogenies. In S. Miyano and T. Takagi, editors, *Genome Informatics*, pages 25–34. University Academy Press, Tokyo, Japan, 1997.

[7] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the 1st Conference on Computational Molecular Biology (RECOMB97)*, pages 75–83, Santa Fe, NM, 1997. ACM Press.

[8] A. Caprara. Sorting permutations by reversals and Eulerian cycle decompositions. *SIAM J. Discrete Math.*, 12(1):91–110, 1999.

[9] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.-S. Wang, T. Warnow, and S. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB00)*, pages 104–115, San Diego, CA, 2000.

[10] S. Hannenhalli. *Software for computing inversion distances between signed gene orders*. Department of Mathematics, University of Southern California, URL. `www-hto.usc.edu/plain/people/Hannenhalli.html`.

[11] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th Annual Symposium on Theory of Computing (STOC95)*, pages 178–189, Las Vegas, NV, 1995. ACM Press.

[12] H. Kaplan, R. Shamir, and R.E. Tarjan. A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal of Computing*, 29(3):880–892, 1999. First appeared in *Proceedings of the 8th Annual Symposium on Discrete Algorithms (SODA97)*, 344-351, New Orleans, LA. ACM Press.

[13] I. Mantin and R. Shamir. Genome Rearrangement Algorithm Applet: An algorithm for sorting signed permutations by reversals. `www.math.tau.ac.il/~rshamir/GR/`, 1999.

[14] C.C. McGeoch. Toward an experimental method for algorithm simulation. *INFORMS Journal of Computing*, 8:1–15, 1996.

[15] B. M.E. Moret, S. Wyman, D.A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proceedings of the 6th Pacific Symposium on Biocomputing (PSB2001)*, pages 583–594, Big Island, HI, January 2001.

[16] B.M.E. Moret. Towards a discipline of experimental algorithmics. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 2001. To appear. Available at `www.cs.unm.edu/~moret/dimacs.ps`.

[17] J.H. Nadeau and B.A. Taylor. Lengths of chromosome segments conserved since divergence of man and mouse. In *Proceedings of the National Academy of Sciences*, volume 81, pages 814–818, 1984.

[18] R.G. Olmstead and J.D. Palmer. Chloroplast DNA systematics: a review of methods and data analysis. *American Journal of Botany*, 81:1205–1224, 1994.

[19] J.D. Palmer. Chloroplast and mitochondrial genome evolution in land plants. In R. Herrmann, editor, *Cell Organelles*, pages 99–133. Springer Verlag, 1992.

[20] L.A. Raubeson and R.K. Jansen. Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants. *Science*, 255:1697–1699, 1992.

[21] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5:555–570, 1998.

[22] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS, Boston, MA, 1997.

Computing Phylogenetic Roots with Bounded Degrees and Errors

Extended Abstract

Zhi-Zhong Chen[1]*, Tao Jiang[2,3]**., and Guo-Hui Lin[3,4]***.

[1] Department of Mathematical Sciences, Tokyo Denki University, Hatoyama, Saitama 350-0394, Japan.
[2] Department of Computer Science, University of California, Riverside, CA 92521.
[3] Department of Computing and Software, McMaster University, Hamilton, Ontario L8S 4L7, Canada.
[4] Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

Abstract. Given a set of species and their similarity data, an important problem in evolutionary biology is how to reconstruct a phylogeny (also called evolutionary tree) so that species are close in the phylogeny if and only if they have high similarity. Assume that the similarity data are represented as a graph $G = (V, E)$ where each vertex represents a species and two vertices are adjacent if they represent species of high similarity. The phylogeny reconstruction problem can then be abstracted as the problem of finding a (phylogenetic) tree T from the given graph G such that (1) T has no degree-2 internal nodes, (2) the external nodes (*i.e.* leaves) of T are exactly the elements of V, and (3) $(u, v) \in E$ if and only if $d_T(u, v) \leq k$ for some fixed threshold k, where $d_T(u, v)$ denotes the distance between u and v in tree T. This is called the PHYLOGENETIC kTH ROOT PROBLEM (PRk), and such a tree T, if exists, is called a *phylogenetic* kth *root* of graph G. The computational complexity of PRk is open, except for $k \leq 4$. In this paper, we investigate PRk under a natural restriction that the maximum degree of the phylogenetic root is bounded from above by a constant. Our main contribution is a linear-time algorithm that determines if G has such a phylogenetic kth root, and if so, demonstrates one. On the other hand, as in practice the collected similarity data are usually not perfect and may contain errors, we propose to study a generalized version of PRk where the output phylogeny is only required to be an *approximate* root of the input graph. We show that this and other related problems are computationally intractable.

* Supported in part by the Grant-in-Aid for Scientific Research of the Ministry of Education, Science, Sports and Culture of Japan, under Grant No. 12780241. Email: chen@r.dendai.ac.jp. Work done while visiting at UC Riverside.

** Supported in part by a UCR startup grant and NSF Grants CCR-9988353 and ITR-0085910. Email: jiang@cs.ucr.edu

*** Supported in part by NSERC Research Grant OGP0046613 and a CITO grant. Email: ghlin@math.uwaterloo.ca

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 377-388, 2001.

1 Introduction

The reconstruction of evolutionary history for a set of species from quantitative biological data has long been a popular problem in computational biology. This evolutionary history is typically modeled by an evolutionary tree or *phylogeny*. A phylogeny is a tree where the leaves are labeled by species and each internal node represents a speciation event whereby an ancestral species gives rise to two or more child species. Both rooted and unrooted trees have been used to describe phylogenies in the literature, although they are practically equivalent. In this paper, we will consider only unrooted phylogenies for the convenience of presentation. [1] The internal nodes of a phylogeny have degrees (in the sense of unrooted trees, *i.e.* the number of incident edges) at least 3. Proximity within a phylogeny in general corresponds to similarity in evolutionary characteristics.

Many phylogenetic reconstruction algorithms have been proposed and studied in the literature [12]. In this paper we investigate the computational feasibility of a graph-theoretic approach for reconstructing phylogenies from similarity data. Specifically, interspecies similarity is represented by a graph where the vertices are the species and the adjacency relation represents evidence of evolutionary similarity. A phylogeny is then reconstructed from the graph such that the leaves of the phylogeny are labeled by vertices of the graph (*i.e.* species) and for any two vertices of the graph, they are adjacent in the graph if and only if their corresponding leaves in the phylogeny are connected by a path of length at most k, where k is a predetermined proximity threshold. To be clear, vertices in the graph are called *vertices* while those in the phylogeny *nodes*. Recall that the length of the (unique) path connecting two nodes u and v in phylogeny T is the number of edges on the path, which is denoted by $d_T(u, v)$. This approach gives rise to the following algorithmic problem [8]:

> Phylogenetic kth Root Problem (PRk):
> Given a graph $G = (V, E)$, find a phylogeny T with leaves labeled by the elements of V such that for each pair of vertices $u, v \in V$, $(u, v) \in E$ if and only if $d_T(u, v) \leq k$.

Such a phylogeny T (if exists) is called a *phylogenetic* kth *root*, or a kth *root phylogeny*, of graph G. Graph G is called the kth *phylogenetic power* of T. For convenience, we denote the kth phylogenetic power of any phylogeny T as T^k. Thus, PRk asks for a phylogeny T such that $G = T^k$.

1.1 Connection to Graph and Tree Roots, and Previous Results

Phylogenetic power might be thought of as a *Steiner* extension of the standard notion of *graph power*. A graph G is the kth *power* of a graph H (or equivalently, H is a kth *root* of G) if vertices u and v are adjacent in G if and only if the length of the shortest path from u to v in H is at most k. An important special case of graph power/root problems is the following:

[1] But some of our hardness proofs will also use rooted trees as intermediate data structures in the construction.

TREE kTH ROOT PROBLEM (TRk):
Given a graph $G = (V, E)$, find a tree $T = (V, E_T)$ such that $(u, v) \in E$ if and only if $d_T(u, v) \leq k$.

If T exists then it is called a *tree* kth *root*, or a kth *root tree*, of graph G.

The special case TR2 is also known as the TREE SQUARE ROOT PROBLEM [9]. Correspondingly, we call PR2 the PHYLOGENETIC SQUARE ROOT PROBLEM. There is rich literature on graph root and power (see [2, Section 10.6] for an overview), but few results on phylogenetic/tree roots/powers. It is NP-complete to recognize a graph power [10], nonetheless, it is possible to determine if a graph has a kth root tree, for any fixed k, in $O(n^3)$ time, where n is the number of vertices in the input graph [5]. In particular, determining if a graph has a tree square root can be done in $O(n + e)$ time [9], where e is the number of edges in the input graph. Recently, Nishimura, Ragde, and Thilikos [11] presented an $O(n^3)$-time algorithm for a variant of PRk, for $k \leq 4$, where internal nodes of the output phylogeny are allowed to have degree 2. More recently, Lin, Kearney, and Jiang [8] introduced a novel notion of *critical clique* and obtained an $O(n + e)$-time algorithm for PRk, for $k \leq 4$. Unfortunately, both algorithms cannot be generalized to $k \geq 5$.

1.2 Our Contribution

In the practice of phylogeny reconstruction, most phylogenies considered are trees of degree 3 [12] because speciation events are usually bifurcating events in the evolutionary process. In such *fully resolved* phylogenetic trees, each internal node has three neighbors and represents a speciation event that some ancestral species splits into two child species. Nodes of degrees higher than 3 are introduced only when the input biological (similarity) data is not sufficient to separate individual speciation events and hence several such events may be collapsed into a non-bifurcating (super) speciation event in the reconstructed phylogeny. Hence in this paper, we consider a restricted version of PRk where the output phylogeny is assumed to have degree at most Δ, for some fixed constant $\Delta \geq 3$. For simplicity, we call it the DEGREE-Δ PRk and denote it in short as ΔPRk. Since in the practice of computational biology the set of species under consideration are more or less related, we are mostly interested in connected graphs. The main contribution of this paper is a linear-time algorithm that determines, for any input connected graph G and constant $\Delta \geq 3$, if G has a kth root phylogeny with degree at most Δ, and if so, demonstrates one such phylogeny. The basic construction in our algorithm is a nontrivial application of bounded-width tree-decomposition of certain chordal graphs [2].

Notice that the input graph in PRk is derived from some similarity data, which is usually inexact in practice and may have erroneous (spurious or missing) edges. Such errors may result in graphs that has no phylogenetic roots. Hence, it is natural to consider a more relaxed problem where we look for phylogenetic trees whose powers are *close* to the input graphs. The precise formulation is as follows:

Closest Phylogenetic kth Root Problem (CPRk):
Given a graph $G = (V, E)$ and a nonnegative integer ℓ, find a phylogeny T with leaves labeled by V such that G and T^k differ by at most ℓ edges. That is,

$$\left|E(G) \oplus E(T^k)\right| = \left|\left(E(G) - E(T^k)\right) \cup \left(E(T^k) - E(G)\right)\right| \leq \ell.$$

A phylogeny T which minimizes the above edge discrepancy is called a *closest* kth *root phylogeny* of graph G.

The Closest Tree kth Root Problem (CTRk) is defined analogously. Notice that CTR1 is trivially solved by finding a spanning tree of the input graph. Kearney and Corneil [5] proved that CTRk is NP-complete when $k \geq 3$. The computational complexity for CTR2 had been open for a while and is recently shown to be intractable by Jiang, Lin, and Xu [4]. In this paper, we will show that CPRk is NP-complete, for any $k \geq 2$. Another closely related problem, the Steiner kth Root Problem (where $k \geq 1$), is also studied.

We introduce some notations and definitions, as well as some existing related results, in the next section. Our main result on bounded-degree PRk is presented in Section 3. The hardness of closest phylogenetic root and Steiner root problems is discussed in Section 4. We close the paper with some remarks in Section 5.

Due to the page limit, we will omit all proofs and most of the details of the constructions for the hardness results in this extended abstract. The proofs and detailed constructions can be found in the full version [3].

2 Preliminaries

We employ standard terminologies in graph theory. In particular, the subgraph of a graph G induced by a vertex set U of G is denoted by $G[U]$, the degree of a vertex v in G is denoted by $deg_G(v)$, and the maximum size of a clique in G is denoted by $\omega(G)$. First, it is obvious that if a graph has a kth root phylogeny, then it must be *chordal*, that is, it contains no induced subgraph which is a cycle of size greater than 3.

Definition 1. *A* tree-decomposition *of a graph $G = (V, E)$ is a pair $\mathcal{D} = (\mathcal{T}, \mathcal{B})$ consisting of a tree $\mathcal{T} = (U, F)$ and a collection $\mathcal{B} = \{B_\alpha \mid B_\alpha \subseteq V, \alpha \in U\}$ of* sets *(called* bags*) for which*

- $\bigcup_{\alpha \in U} B_\alpha = V$,
- *for each edge $(v_1, v_2) \in E$, there is a node $\alpha \in U$ such that $\{v_1, v_2\} \subseteq B_\alpha$, and*
- *if $\alpha_2 \in U$ is on the path connecting α_1 and α_3 in $\mathcal{T}$, then $B_{\alpha_1} \cap B_{\alpha_3} \subseteq B_{\alpha_2}$.*

The treewidth *associated with this tree-decomposition $\mathcal{D} = (\mathcal{T}, \mathcal{B})$ is $tw(G, \mathcal{D}) = \max_{\alpha \in U} |B_\alpha| - 1$.*

The treewidth *of graph G, denoted by $tw(G)$, is the minimum $tw(G, \mathcal{D})$ taken over all tree-decompositions $\mathcal{D}$ of G.*

A clique-tree-decomposition *of G is a tree-decomposition $(\mathcal{T}, \mathcal{B})$ of G such that each bag in $\mathcal{B}$ is a maximal clique of G.*

Lemma 1. [6] *Every chordal graph has a clique-tree-decomposition.*

From the proof of Lemma 1 given in [6], it is not difficult to see that a clique-tree-decomposition $\mathcal{D} = (\mathcal{T}, \mathcal{B})$ of a given chordal graph G can be computed in linear time if $\omega(G) = O(1)$. We can further modify $\mathcal{D}$ so that $deg_{\mathcal{T}}(\alpha) \leq 3$ for each node α of $\mathcal{T}$ [1]. This modification takes linear time too if $\omega(G) = O(1)$.

Hereafter, a tree-decomposition of a chordal graph G always means a clique-tree-decomposition $\mathcal{D} = (\mathcal{T}, \mathcal{B})$ of G such that $deg_{\mathcal{T}}(\alpha) \leq 3$ for all nodes of $\mathcal{T}$. Furthermore, in the sequel, we abuse the notations to use $\mathcal{D}$ to denote the tree $\mathcal{T}$ in it (since we will use T to denote the kth root phylogeny of graph G), and denote the bag associated with a node α of $\mathcal{D}$ by B_α.

3 Algorithm for Bounded-Degree PRk

This section presents a linear-time algorithm for solving 3PRk. The adaptation to ΔPRk where $\Delta \geq 4$ is straightforward and is hence omitted here.

We assume that the input graph $G = (V, E)$ is not complete but is chordal; otherwise the problem is trivially solved in linear time. Since every vertex $v \in V$ appears as an external node (*i.e.* leaf) in the kth root phylogeny, the maximum size $\omega(G)$ of a clique in G can be bounded from above by a constant $f(k)$, where

$$f(k) = \begin{cases} 3 \cdot 2^{\frac{k}{2}-1}, & \text{if } k \text{ is even,} \\ 2^{\frac{k+1}{2}} - 1, & \text{if } k \text{ is odd.} \end{cases}$$

So, we can construct a clique-tree-decomposition $\mathcal{D}$ of G in linear time. The basic idea behind our algorithm is to do a dynamic programming on a rooted version of the decomposition $\mathcal{D}$. The dynamic programming starts at the leaves of $\mathcal{D}$ and proceeds upwards. After processing the root, the algorithm will construct a kth root phylogeny of G if there is any. The processing of a node α of $\mathcal{D}$ can be sketched as follows. Let U_α be the union of the bags associated with α and its descendants in $\mathcal{D}$. While processing α, the algorithm computes a set of trees T such that (1) T may possibly be a subtree of a kth root phylogeny of G, (2) all vertices of U_α are leaves of T, and (3) each leaf of T not contained in U_α is not labeled. The unlabeled leaves of T serve as ports from which we can expand T so that it may eventually become a kth root phylogeny of G. The crucial point we will observe is that we only need those ports that are at distance at most k apart from vertices of B_α in T. This point implies that the number of necessary ports only depends on k and hence is a constant.

One more notation is in order. For two adjacent nodes α and β of $\mathcal{D}$, let $U(\alpha, \beta) = \bigcup_\gamma B_\gamma$ where γ ranges over all nodes of $\mathcal{D}$ with $d_{\mathcal{D}}(\gamma, \alpha) < d_{\mathcal{D}}(\gamma, \beta)$. In other words, if we root $\mathcal{D}$ at node β, then $U(\alpha, \beta)$ is the union of the bags associated with α and its descendants in $\mathcal{D}$. A useful property of $\mathcal{D}$ is that for every internal node β and every two neighbors α_1 and α_2 of β in $\mathcal{D}$, G has no edge between a vertex of $U(\alpha_1, \beta) - B_\beta$ and a vertex of $U(\alpha_2, \beta) - B_\beta$.

3.1 Ideas behind the Dynamic Programming Algorithm

Note that since $\Delta = 3$, every internal node in a kth root phylogeny T of G has degree exactly 3.

Definition 2. *Let U be a set of vertices of G. A* relaxed phylogeny *for U is a tree R satisfying the following conditions:*

- *The degree of each internal node in R is* 3.
- *Each vertex of U is a leaf in R and appears in R only once. For convenience, we call the leaves of R that are also vertices of U* final leaves *of R, and call the rest leaves of R* temporary leaves *of R.*
- *For every two vertices u and v of U, u and v are adjacent in G if and only if $d_R(u, v) \leq k$.*
- *Each temporary leaf v of R is assigned a pair (γ, t), where γ is a node of $\mathcal{D}$ and $0 \leq t \leq k$. We call γ the* color *of v and call t the* threshold *of v. For convenience, we denote the color of a temporary leaf v of R by $c_R(v)$, and denote the threshold of v by $t_R(v)$.*

Intuitively speaking, the temporary leaves of R serve as ports from which we can expand R so that it may eventually become a kth root phylogeny of G.

Recall that our algorithm processes the nodes of $\mathcal{D}$ one by one. While processing a node α of $\mathcal{D}$, the algorithm finds out all relaxed phylogenies for B_α that are subtrees of kth root phylogenies of graph G. The following lemma shows that such relaxed phylogenies for B_α have certain useful properties.

Lemma 2. *Let T be a kth root phylogeny of G. Let α be a node of $\mathcal{D}$. Root T at an arbitrary leaf that is in B_α. Define a* pure *node to be a node w of T such that α has a neighbor γ in $\mathcal{D}$ such that all leaf descendants of w in T are in $U(\gamma, \alpha) - B_\alpha$. Define a* critical *node to be a pure node of T whose parent is not pure. Let R be the relaxed phylogeny for B_α obtained from T by performing the following steps:*

1. *For every critical node w of T, perform the following:*
   a) *Compute the minimum distance from w to a leaf descendant of w in T; let i_w denote this distance. (*Comment: *$i_w \leq k$ or else the leaf descendants of w in tree T would be unreachable from the outside in graph G.)*
   b) *Find the neighbor γ of α such that all leaf descendants of w in T are contained in $U(\gamma, \alpha)$.*
   c) *Delete all descendants (excluding w, of course) of w, and assign the pair (γ, i_w) to w.*
2. *Unroot T.*

Then, R has the following properties:

- *For every temporary leaf v of R, $c_R(v)$ is a neighbor of α in $\mathcal{D}$.*
- *For every two temporary leaves u and v of R with different colors, it holds that $t_R(u) + t_R(v) + d_R(u, v) > k$.*

- *For every neighbor γ of α in $\mathcal{D}$, every temporary leaf v of R with $c_R(v) = \gamma$, and every final leaf w of R with $w \notin B_\gamma$, it holds that $d_R(v, w) + t_R(v) > k$.*
- *For every internal node v of R, either at least one descendant of v is a final leaf of R, or there is a final leaf u of R with $d_R(u, v) \leq k - 1$.*

Each relaxed phylogeny R for B_α having the four properties in Lemma 2 is called a *skeleton* of α. The following lemma shows that there can be only a constant number of skeletons of α.

Lemma 3. *For each node α of $\mathcal{D}$, the number of skeletons of α is bounded from above by a constant depending only on k and $|B_\alpha|$.*

By Lemma 3, while processing a node α of $\mathcal{D}$, our algorithm can find out all skeletons of α in constant time. For each skeleton S of α, if possible, the algorithm then extends S to a relaxed phylogeny for $U(\alpha, \beta)$ where β is the parent of α in rooted $\mathcal{D}$. The algorithm records these relaxed phylogenies of α in the dynamic programming table for later use when processing the parent β. The following definition aims at removing unnecessary relaxed phylogenies of α from the dynamic programming table.

Definition 3. *Let α and β be two adjacent nodes of $\mathcal{D}$. Let S be a skeleton of α. The* projection *of S to β is a relaxed phylogeny for $B_\alpha \cap B_\beta$ obtained from S by performing the following steps:*

1. *Change each final leaf $v \notin B_\beta$ to a temporary leaf; set the threshold of v to be 0 and set the color of v to be α.*
2. *Root S at an arbitrary vertex of $B_\alpha \cap B_\beta$.*
3. *Find those nodes v in S such that (i) every leaf descendant of v in S is a temporary leaf whose color is not β, but (ii) the parent of v in S does not have Property (i).*
4. *For each node v found in the last step, if v is a leaf in S then set the color of v to be α; otherwise, perform the following steps:*
   a) *Set $m_v = \min_u\{t_S(u) + d_S(u, v)\}$ where u ranges over all leaf descendants u of v in S.*
   b) *Delete all descendants of v in S.*
   c) *Set v to be a temporary leaf, set the color of v to be α, and set the threshold of v to be m_v.*
5. *Unroot S.*

Obviously, two different skeletons of α may have the same projection to β. For convenience, we say that these skeletons are *equivalent*. Among equivalent skeletons of α, our algorithm will extend only a hopeful one of them to a relaxed phylogeny for $U(\alpha, \beta)$ and record it in the dynamic programming table. This motivates the following definition:

Definition 4. *Let α and β be two adjacent nodes of $\mathcal{D}$. A* projection *of α to β is the projection of a skeleton of α to β. Let P be a projection of α to β. An* expansion *of P to $U(\alpha, \beta)$ is a relaxed phylogeny X for $U(\alpha, \beta)$ such that some subtree Y of X is isomorphic to P, and the bijection f from the node set of P to the node set of Y witnessing this isomorphism satisfies the following conditions:*

- *For every final leaf v of P, $f(v) = v$.*
- *For every temporary leaf v of P with $c_P(v) = \beta$, $f(v)$ is a temporary leaf of X with $c_X(f(v)) = c_P(v)$ and $t_X(f(v)) = t_P(v)$.*
- *Suppose that we root X at a vertex in $B_\alpha \cap B_\beta$. Then, for every temporary leaf v of P with $c_P(v) \neq \beta$ (hence $c_P(v) = \alpha$), all leaf descendants of $f(v)$ in X are final leaves and are contained in $U(\alpha, \beta) - B_\beta$, and the minimum distance between $f(v)$ and a leaf descendant of $f(v)$ in X equals to $t_P(v)$.*

Note that a projection of α to β may have no expansion to $U(\alpha, \beta)$. The following lemma shows that if G has a kth root phylogeny T, then some subtree of T is a projection of α to β and another subtree of T is its expansion to $U(\alpha, \beta)$.

Lemma 4. *Let α and β be two adjacent nodes in $\mathcal{D}$. Let T be a kth root phylogeny of G. Root T at an arbitrary leaf that is in B_α. Let R be the skeleton of α obtained from T as in Lemma 2. Let P be the projection of R to β. Define a* β-pure *node to be a node w of T such that all leaf descendants of w in T are contained in $U(\beta, \alpha) - B_\alpha$. Further define a* β-critical *node to be a β-pure node of T whose parent is not β-pure. Let X be the relaxed phylogeny for $U(\alpha, \beta)$ obtained from T by performing the following steps:*

1. *For every β-critical node w of T, perform the following:*
   a) *Compute the minimum distance from w to a leaf descendant of w in T; let i_w denote this distance. (*Comment: *$i_w \leq k$ or else the leaf descendants of w in tree T would be unreachable from the outside in graph G.)*
   b) *Delete all descendants (excluding w, of course) of w, and assign the pair (β, i_w) to w.*
2. *Unroot T.*

Then, X is an expansion of P to $U(\alpha, \beta)$.

By Lemma 4, whenever G has a kth root phylogeny, there is always a projection of α to β that has an expansion to $U(\alpha, \beta)$. While processing α, our algorithm will find out those projections that have expansions to $U(\alpha, \beta)$, and record the expansions in the dynamic programming table.

3.2 Details of Dynamic Programming for 3PRk

To solve the 3PRk problem for G, we perform a dynamic programming on the tree-decomposition $\mathcal{D}$ as follows. To simplify the description of the algorithm, we add a new node r to $\mathcal{D}$, connect r to an arbitrary leaf α of $\mathcal{D}$, and copy the bag at α to r (that is, $B_r = B_\alpha$). Clearly, the resultant $\mathcal{D}$ is still a required tree-decomposition of G. Root $\mathcal{D}$ at r.

The dynamic programming starts at the leaves of $\mathcal{D}$, and proceeds upwards; after the unique child of the root r of $\mathcal{D}$ is processed, we will know whether G has a kth root phylogeny or not. The invariant maintained during the dynamic programming is that after each non-root node α has been processed, for each

projection P of α to its parent β, we will have found out whether P has an expansion X to $U(\alpha, \beta)$, and will have found and recorded such an X if any.

Now consider how a non-root node α of $\mathcal{D}$ is processed. Let β be the parent of α in $\mathcal{D}$. First suppose that α is a leaf of $\mathcal{D}$. When processing α, we find and record all possible projections of α to β; moreover, for each projection P found, we also record a skeleton S of α such that P is the projection of S to β.

Next suppose that α is neither a leaf nor the root node of $\mathcal{D}$, and suppose that all descendants of α in $\mathcal{D}$ have been processed. To process α, we try all possible skeletons S of α. When trying S, for each child γ of α in $\mathcal{D}$, we first compute the projection P_γ of S to γ, and then check whether P_γ is also a projection of γ to α and additionally has an expansion to $U(\gamma, \alpha)$. If the checking fails for at least one child of α, we proceed to try the next possible skeleton of α. Otherwise, we can conclude that the projection P_β of S to β has an expansion to $U(\alpha, \beta)$ because such an expansion can be constructed as follows:

1. For each child γ of α in $\mathcal{D}$, search the dynamic programming table to find the expansion X_γ of P_γ to $U(\gamma, \alpha)$, and find the bijection f_γ (from the node set of P_γ to the node set of some subtree of X_γ) witnessing that X_γ is an expansion of P_γ to $U(\gamma, \alpha)$.
   (*Comment:* To speed up the algorithm, we may have recorded this bijection in the dynamic programming table when processing γ.)
2. For each child γ of α in $\mathcal{D}$, root X_γ at an arbitrary vertex of $B_\gamma \cap B_\alpha$.
3. Modify S as follows: For each temporary leaf v of S with $c_S(v) \neq \beta$, replace v by the subtree rooted at $f_\gamma(v)$ of X_γ, where $\gamma = c_S(v)$.
   (*Comment*: Recall that by Definition 3, each temporary leaf v of S with $c_S(v) = \gamma$ is also a temporary leaf of P_γ.)

One can verify that the above construction indeed gives us an expansion of P_β. Since P_β is a possible projection of α to β, we record this expansion for P_β in the dynamic programming table. After trying all possible skeletons of α, if we find no projection of α to β that has an expansion to $U(\alpha, \beta)$, then we can conclude that G has no kth root phylogeny; otherwise, we proceed to the processing of the next node of $\mathcal{D}$.

Finally, suppose that α is the unique child of the root r of $\mathcal{D}$. Further suppose that α has been successfully processed; otherwise we already knew that G has no kth root phylogeny. Then, by searching the dynamic programming table, we try to find a projection P of α to r such that (i) P has no temporary leaf whose color is r, and (ii) an expansion X of P to $U(\alpha, r)$ has been recorded in the dynamic programming table. If P is found, we can conclude that X is a kth root phylogeny for G; otherwise, we can conclude that G has no kth root phylogeny.

The above discussion justifies the following theorem:

Theorem 1. *Let k be a constant integer larger than or equal to 2. There is a linear-time algorithm determining if a given graph has a kth root phylogeny in which every internal node has degree 3, and if so, demonstrating one such phylogeny.*

We can easily generalize the above discussion to prove the following:

Corollary 1. *Let Δ and k be constant integers such that $\Delta \geq 3$ and $k \geq 2$. There is a linear-time algorithm determining if a given graph has a kth root phylogeny in which every internal node has degree in the range $[3, \Delta]$, and if so, demonstrating one such phylogeny.*

4 The Hardness of Closest Phylogenetic Root Problems

Due to the page limit, we will only introduce some basic concepts that are useful in proving our hardness results, and refer the reader to the full version [3] for a detailed discussion of the hardness results and their proofs.

Consider a set $S = \{s_1, s_2, \ldots, s_n\}$. Let M be a symmetric matrix with rows and columns indexed by the elements of S. M is a *binary dissimilarity matrix* on set S if $M(s_i, s_j) \in \{1, 2\}$ for every pair (s_i, s_j) of distinct elements of S and $M(s_i, s_i) = 0$ for every element $s_i \in S$.

A tree T is a 2-*ultrametric* on set S if T is a *rooted* tree whose leaves are labeled by the elements of S and each leaf-to-root path contains exactly two edges. Call a node in T that is neither a leaf nor the root a *middle* node, to avoid ambiguity. The *half-distance* between two leaves s_i and s_j, denoted by $h_T(s_i, s_j)$, is one half of the number of edges on the unique path in T connecting s_i and s_j. Clearly, $h_T(s_i, s_j) \in \{1, 2\}$ if $i \neq j$, and $h_T(s_i, s_i) = 0$ for every i.

Given a binary dissimilarity matrix M and a 2-ultrametric T on set S, define

$$D(T, M) = \sum_{i<j} |h_T(s_i, s_j) - M(s_i, s_j)|,$$

which measures how well T matches the inter-leaf (half-)distances specified by M. [2] The following Fitting Ultrametric Trees (FUT) problem has been shown to be NP-complete by Křivánek and Morávek [7].

Fitting Ultrametric Trees (FUT):
Given a binary dissimilarity matrix M on set S and a nonnegative integer ℓ, decide if there is a 2-ultrametric T on S such that $D(T, M) \leq \ell$.

Kearney and Corneil [5] proved that CTRk is NP-hard when $k \geq 3$ by a (polynomial-time) reduction from FUT (to CTR3). Using a more dextrous reduction, Jiang, Lin, and Xu [4] have recently shown that CTR2 is intractable too. To prove the intractability of CPRk, we need one more definition. A *critical clique* [8] of a graph is a maximal subset of vertices that are adjacent to each other and have a common neighborhood. As for constructing a phylogenetic root, the vertices in a critical clique can be identified because they are interchangable in every phylogenetic root.

Using the above concepts and some very careful and involved construction, we are able to prove the following hardness result:

[2] So, here the entries in M are supposed to represent the half-distances between species instead of full distances.

Theorem 2. CPRk *is NP-complete, for any* $k \geq 2$.

We next study another problem closely related to PRk and TRk, which is the Steiner kth Root Problem [8]. Recall that given a graph $G = (V, E)$, TRk asks for a tree whose node set is exactly V and PRk asks for a tree whose leaf-set is exactly V. A more general problem is to ask for a tree T whose node set is a superset of V and whose leaf-set is a subset of V, and such that for every pair of vertices u and v in V, $d_T(u, v) \leq k$ if and only if $(u, v) \in E$. We call a tree T, whose node set is a superset of V and whose leaf-set is a subset of V, a *Steiner tree* on V. The nodes of T that are not in V are called the *Steiner nodes*.

Steiner kth Root Problem (SRk):
Given a graph $G = (V, E)$, find a Steiner tree T on V such that for each pair of vertices $u, v \in V$, $(u, v) \in E$ if and only if $d_T(u, v) \leq k$.

Such a Steiner tree T (if exists) is called a *Steiner* kth *root* or a kth *root Steiner tree* of G. G is called the kth *Steiner power* of T. We also abuse T^k to denote the kth *Steiner power* of T, when there is no confusion from the context.

Notice that we do not require here a non-leaf node in a Steiner tree to have degree at least 3. This requirement is not necessary from the tree root point of view. But, one may do so as this requirement is natural from the phylogenetic root point of view. Steiner trees satisfying this additional requirements are called *restricted Steiner trees*. Graphs having a restricted Steiner kth root, for $k = 1, 2$, can be recognized in linear time [8]. The recognition algorithm can be extended to find an ordinary Steiner kth root, for $k = 1$ and $k = 2$. However, when $k \geq 3$, no polynomial-time recognition algorithm has been reported yet to find either a Steiner kth root or a restricted Steiner kth root. In the following, we will only consider ordinary Steiner roots and show that the closest Steiner kth root problem (CSRk), defined in a straightforward way, is NP-complete when $k \geq 2$.

For CSR1, we notice that deleting all Steiner nodes from an (approximate) 1st root Steiner tree T results in a collection of subtrees such that vertices in different subtrees are not adjacent in T^1. Therefore, for any input graph G, the best way to build the closest 1st root Steiner tree is to construct a spanning tree for each connected component in G and then connect these spanning trees together via a Steiner node. That is, a closest 1st root Steiner tree can be computed in $O(n)$ time, where n is the number of vertices in the input graph. The complexity changes when k marches from 1 to 2.

Theorem 3. CSRk *is NP-complete, for any* $k \geq 2$.

5 Closing Remarks

Since CPRk is NP-complete for all $k \geq 2$, it would be interesting to know how well we can approximate the closest phylogenetic kth root.

Acknowledgments. GL would like to thank Paul Kearney for bringing CPRk to his attention, Bin Ma and Jinbo Xu for many helpful discussions.

References

1. H.L. Bodlaender. Nc-algorithms for graphs with small treewidth. In *The 14th International Workshop on Graph-Theoretic Concepts in Computer Science (WG 1988)*, LNCS 344, pages 1–10, 1989.
2. A. Brandstädt, V.B. Le, and J.P. Spinrad. *Graph Classes: a Survey*. SIAM Monographs on Discrete Mathematics and Applications. SIAM, Philadelphia, 1999.
3. Z.-Z. Chen, T. Jiang, and G.-H. Lin. Computing phylogenetic roots with bounded degrees and errors. Manuscript, Department of Computer Science, University of Waterloo, April 2001.
4. T. Jiang, G.-H. Lin, and J. Xu. On the closest tree kth root problem. Manuscript, Department of Computer Science, University of Waterloo, November 2000.
5. P.E. Kearney and D.G. Corneil. Tree powers. *Journal of Algorithms*, 29:111–131, 1998.
6. T. Kloks. *Treewidth*. LNCS 842. Springer-Verlag, Berlin, 1994.
7. M. Křivánek and J. Morávek. NP-hard problems in hierarchical-tree clustering. *Acta Informatica*, 23:311–323, 1986.
8. G.-H. Lin, P.E. Kearney, and T. Jiang. Phylogenetic k-root and Steiner k-root. In *The 11th Annual International Symposium on Algorithms and Computation (ISAAC 2000)*, LNCS 1969, pages 539–551, 2000.
9. Y.-L. Lin and S.S. Skiena. Algorithms for square roots of graphs. *SIAM Journal on Discrete Mathematics*, 8:99–118, 1995.
10. R. Motwani and M. Sudan. Computing roots of graphs is hard. *Discrete Applied Mathematics*, 54:81–88, 1994.
11. N. Nishimura, P. Ragde, and D.M. Thilikos. On graph powers for leaf-labeled trees. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory (SWAT 2000)*, LNCS 1851, pages 125–138, 2000.
12. D.L. Swofford, G.J. Olsen, P.J. Waddell, and D.M. Hillis. Phylogenetic inference. In D.M. Hillis, C. Moritz, and B.K. Mable, editors, *Molecular Systematics (2nd Edition)*, pages 407–514. Sinauer Associates, Sunderland, Massachusetts, 1996.

A Decomposition-Based Approach to Layered Manufacturing*

Ivaylo Ilinkin[1], Ravi Janardan[1], Jayanth Majhi[2], Jörg Schwerdt[3], Michiel Smid[3], and Ram Sriram[4]

[1] Dept. of Computer Science & Engineering, University of Minnesota, Minneapolis, MN 55455, U.S.A.
{ilinkin,janardan}@cs.umn.edu
[2] Synopsys Corporation, 700 East Middlefield Road, Mountain View, CA 94043
majhi@synopsys.com
[3] Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, D-39106 Magdeburg, Germany.
{schwerdt,michiel}@isg.cs.uni-magdeburg.de
[4] National Institute of Standards and Technology, Gaithersburg, MD 20899.
sriram@cme.nist.gov

Abstract. *Layered Manufacturing* allows physical prototypes of 3D parts to be built directly from their computer models, as a stack of 2D layers. This paper proposes a new approach, which decomposes the model into a small number of pieces, builds each separately, and glues them together to generate the prototype. This allows large models to be built in parallel and also reduces the need for so-called support structures. Decomposition algorithms that minimize support requirements are given for convex and non-convex polyhedra. Experiments, on convex polyhedra, show that the approach can reduce support requirements substantially.

1 Introduction

Layered Manufacturing (LM) is an emerging technology that allows the construction of physical prototypes of 3D parts directly from their CAD models using a "3D printer" attached to a personal computer. LM provides the designer with an additional level of physical verification that makes it possible to detect and correct design flaws that may have, otherwise, gone unnoticed in the virtual model. It is used extensively in the automotive, aerospace, and medical industries.

The basic principle underlying LM is simple: The CAD model, assumed to a surface triangulation in the industry-standard STL format, is oriented suitably and sliced into thin layers by horizontal planes. The layers are then sent over a network to a fabrication device which "prints" them one by one, each on top of the previous one, with the first layer resting on the platform of the fabrication

* Research of II and RJ supported, in part, by NSF grant CCR–9712226 and by NIST grant 60NANB8D0002. Portions of this work were done when RJ visited the University of Magdeburg and JS and MS visited the University of Minnesota under a joint grant for international research from NSF and DAAD.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 389–400, 2001.

device. (The mechanics of this step depend on the specific LM process [6].) Since portions of a layer can overhang previous layers, *support structures* are generated automatically during the process to prop up the overhangs; these are removed subsequently via postprocessing.

The performance of LM depends, in part, on certain geometric factors: For instance, the orientation of the part (or the *build direction*) determines the number of layers, the quantity (i.e., volume) of supports, the location of supports on the part, the extent to which supports "stick" to the part (i.e., area of contact), and the surface finish and accuracy. The orientation also determines the shape of each layer (a polygon), which affects the choice of tool-paths during the printing stage. Currently, these issues are resolved by a human operator, based on experience. The problem of automating these process-planning decisions has been addressed recently by researchers in computational geometry and CAD (details and references can be found in the full version [5] of this paper).

Current process-planning algorithms for LM view the CAD model as a single, monolithic unit. We propose an approach which decomposes the model into a small number of pieces, builds each separately, and then glues them together to generate the prototype. (The number of pieces generated can be controlled by the user.) Specifically, given a decomposition direction, we decompose the model by intersecting it with a suitable plane perpendicular to this direction; we then build the pieces that lie in the same halfspace in the direction given by the normal to the plane that is contained in the halfspace. (Figure 1.)

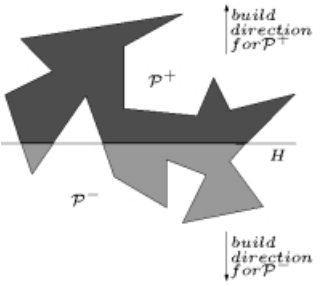


Fig. 1. The decomposition-based approach, shown in 2D for convenience. The polyhedron $\mathcal{P}$ is decomposed by a plane H into polyhedra $\mathcal{P}^+$ and $\mathcal{P}^-$, which are then built in the indicated directions.

This approach has several advantages: It allows the construction of large models that cannot be accommodated in the workspace as a single piece. Moreover, the model can be built very quickly by building the pieces in parallel. The method also lends itself naturally to applications where LM is used to make mold-halves for the model, which are then used to mass-produce the model via casting or injection molding.

A less obvious, but nevertheless crucial, advantage is that it can also reduce support requirements substantially. For instance, if a hollow sphere is built the conventional way, supports will be needed in the interior void and below the lower hemisphere; however, if it is built as two hemispheric shells, in opposite directions, then supports will be needed only in the regions previously occupied by the void, resulting in a reduction in both support volume and contact-area. Reducing support requirements is important because this translates into lower material costs and faster build times.

1.1 Contributions

We give efficient geometric algorithms to decompose polyhedral (i.e., STL) models, w.r.t. a given decomposition direction, so that the support contact-area and, independently, the support volume is minimized. In Sections 3 and 4, we give plane-sweep-based algorithms for convex polyhedra that work by generating and optimizing expressions for the support volume and contact-area as a function of the height of the sweep plane. We also give experimental results that show substantial reductions in support requirements. In Section 5, we consider non-convex polyhedra, which are considerably more difficult because of the complex structure of the supports (Figure 2). We handle these by first identifying certain critical facets (or parts thereof) using cylindrical decomposition [7], and then applying the algorithm for convex polyhedra to these critical facets. We also give a method to keep the size of the decomposition within a user-specified limit.

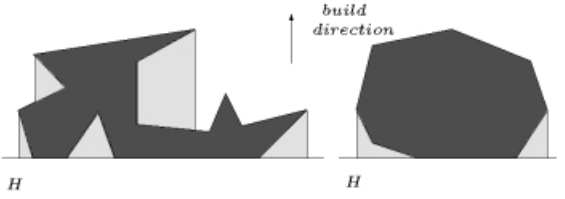


Fig. 2. Support structures (light shading), shown in 2D. Supports in the non-convex case (left) exhibit complexities not seen in the convex case (right): (i) They can rest partially on other parts of the polyhedron; (ii) only a fraction of a facet may be in contact with supports; (iii) parallel facets can also be in contact with supports.

To our knowledge, the only related prior work is due to Fekete and Mitchell [4]. They consider decomposing a polyhedron into special polyhedra called histograms that can be built without supports. They prove that deciding if a polyhedron of genus zero (or a polygon with holes) can be decomposed into k histograms is NP-complete. Our work is different in that we allow supports (thus expanding the class of polyhedra buildable by LM), and we seek a decomposition into two polyhedra (w.r.t. a given direction) such that the total support requirement is minimized, but not necessarily zero.

Due to space limitations, we only sketch the main ideas behind our results and omit many details and proofs. These can be found in the full paper [5].

2 Formalizing the Problem

Let $\mathcal{P}$ be the polyhedron of interest. The facets of $\mathcal{P}$ are triangles and its boundary is assumed to be represented in some standard form, say, as a doubly-connected edge list [3]. (This can be computed easily from the standard STL representation of $\mathcal{P}$.) Let $\mathbf{d}$ be a given *decomposition direction*, which, w.l.o.g., is the positive z direction. Let H be any plane perpendicular to $\mathbf{d}$ and intersecting $\mathcal{P}$; H is the *decomposition plane*. Let $\mathcal{P}^+$ be the closed polyhedron bounded by the facets of $\mathcal{P}$ (or portions thereof) that are above H, and by the facet $\mathcal{P} \cap H$; $\mathcal{P}^+$ may consist of multiple connected components. Define $\mathcal{P}^-$ symmetrically w.r.t. the part of $\mathcal{P}$ below H. Define the *build direction* for $\mathcal{P}^+$ to be $\mathbf{d}$ and for $\mathcal{P}^-$ to be $-\mathbf{d}$, and let H be the *platform* for both polyhedra. (Figure 1.)

We classify any facet $f \in \mathcal{P}$, w.r.t. the given decomposition direction $\mathbf{d}$, as a *front facet*, a *back facet*, or a *parallel facet* of $\mathcal{P}$ depending on whether the angle between the decomposition direction $\mathbf{d}$ and the outward unit-normal, $\mathbf{n}_f$, of f is less than, greater than, or equal to 90°, respectively.

We say that a facet of a polyhedron *needs to be supported* iff the angle between its outer normal and the build direction of the polyhedron is greater than 90°. Thus back facets of $\mathcal{P}^+$ and front facets of $\mathcal{P}^-$ need to be supported. The *support polyhedron* for a back facet $f \in \mathcal{P}^+$ is the closure of the set of all points $p \in \mathbb{R}^3$ such that p is not in the interior of $\mathcal{P}^+$ and the ray shot from p in direction $\mathbf{d}$ first enters $\mathcal{P}^+$ through f. Informally, it is bounded from above by f, on the sides by vertical facets that "drop down" from the edges of f, and from below by the platform and/or portions of front facets of $\mathcal{P}^+$. If $\mathcal{P}^+$ is convex, then it is bounded from below by only the platform. (Figure 2.)

The *support contact-area* for $\mathcal{P}^+$ is the total surface area of $\mathcal{P}^+$ that is in contact with supports. It includes the area of all the back facets of $\mathcal{P}^+$, except $\mathcal{P} \cap H$, and the areas of those portions of front facets and parallel facets that are in contact with supports. (Facet $\mathcal{P} \cap H$ rests on the platform and hence needs no supports. Note that while back facets are completely in contact with supports, front and parallel facets may be only partially in contact.) The *support volume* for $\mathcal{P}^+$ is the total volume of the support polyhedra for all back facets f of $\mathcal{P}^+$ (excluding $\mathcal{P} \cap H$). Symmetrically for $\mathcal{P}^-$. We can now state our problem:

Problem 1. Given a polyhedron $\mathcal{P}$, with n vertices, and a decomposition direction $\mathbf{d}$, compute a plane H perpendicular to $\mathbf{d}$ which decomposes $\mathcal{P}$ into polyhedra $\mathcal{P}^+$ and $\mathcal{P}^-$ such that the total support contact-area or, independently, the support volume is minimized when $\mathcal{P}^+$ and $\mathcal{P}^-$ are built in directions $\mathbf{d}$ and $-\mathbf{d}$, respectively. Additionally, if the user specifies an integer K, then the plane H should be optimal over all planes that generate no more than a total of K connected components of $\mathcal{P}^+$ and $\mathcal{P}^-$.

3 Decomposing Convex Polyhedra: Contact-Area

We sweep the plane $H : z = h$ upwards over $\mathcal{P}$. A facet $f \in \mathcal{P}$ is an *active facet w.r.t. H* if $H \cap f \neq \emptyset$ and at least one vertex of f is strictly above H. Otherwise, f is an *inactive facet w.r.t. H*. Intuitively, an active facet is contained partially in both $\mathcal{P}^+$ and $\mathcal{P}^-$, and small movements of H affect its contribution to the contact-area, whereas an inactive facet is completely contained in $\mathcal{P}^+$ or $\mathcal{P}^-$ and its contribution to the contact-area is not affected by small movements of H.

It is clear that if f is an inactive front facet then its contribution to the total contact-area is $area(f)$ if it is in $\mathcal{P}^-$, and zero if it is in $\mathcal{P}^+$, where $area(f)$ is the area of f. If f is an active front facet, then only the part, f^-, of f that lies below H is in $\mathcal{P}^-$, so f contributes $area(f^-)$ to the total contact-area. Symmetrically, if f is an inactive/active back facet. (In the convex case, parallel facets are never in contact with supports, and are hence ignored.)

The expression for the total contact-area of $\mathcal{P}^+$ and $\mathcal{P}^-$ consists of the *inactive-area* term and the *active-area* term, which are, respectively, the contact-area contributed by the inactive and active facets. If we move H up or down, without crossing a vertex, then the inactive-area does not change, so the inactive-area term is simply a real number. However, the active-area changes because the fraction of an active facet that contributes to the total contact-area changes as H is moved. Lemma 1 shows that the active-area term is quadratic in h.

Lemma 1. *Let the current sweep plane be $H : z = h$. The total contact-area contributed by the active facets (i.e., the active-area) is of the form Ah^2+Bh+C, where the coefficients A, B, and C depend only on the coordinates of the vertices of the active facets.*

Proof. (Sketch) Let f be any active facet, with vertices $v_i = (x_i, y_i, z_i)$, $v_j = (x_j, y_j, z_j)$, and $v_k = (x_k, y_k, z_k)$. (Figure 3) We will prove that the contact-area contributed by f is of the form $a_f h^2+b_f h+c_f$, where the coefficients a_f, b_f, and c_f depend only on the coordinates of the vertices of f. This implies the result.

Let H intersect edge $\overline{v_i v_j}$ at $v_{ij} = (x_{ij}, y_{ij}, z_{ij} = h)$ and edge $\overline{v_i v_k}$ at $v_{ik} = (x_{ik}, y_{ik}, z_{ik} = h)$. It is easy to verify that $x_{ij} = x_i+\alpha_j(h-z_i)$, $y_{ij} = y_i+\beta_j(h-z_i)$, $x_{ik} = x_i+\alpha_k(h-z_i)$, and $y_{ik} = y_i+\beta_k(h-z_i)$, where $\alpha_j = (x_j-x_i)/(z_j-z_i)$, $\alpha_k = (x_k-x_i)/(z_k-z_i)$, $\beta_j = (y_j-y_i)/(z_j-z_i)$, and $\beta_k = (y_k-y_i)/(z_k-z_i)$.

The part of f that is in contact with supports is the triangle f^-, with vertices v_i, v_{ij}, and v_{ik}. We have $area(f^-) = \frac{1}{2}|(\mathbf{v}_{ij}-\mathbf{v}_i)\times(\mathbf{v}_{ik}-\mathbf{v}_i)|$, where $\mathbf{v}_{ij}-\mathbf{v}_i = \alpha_j(h-z_i)\mathbf{i}+\beta_j(h-z_i)\mathbf{j}+(h-z_i)\mathbf{k}$ and $\mathbf{v}_{ik}-\mathbf{v}_i = \alpha_k(h-z_i)\mathbf{i}+\beta_k(h-z_i)\mathbf{j}+(h-z_i)\mathbf{k}$. It follows that $area(f^-) = \frac{1}{2}(h-z_i)^2((\beta_j-\beta_k)^2+(\alpha_j-\alpha_k)^2+(\alpha_j\beta_k-\alpha_k\beta_j)^2)^{1/2}$.

The coefficient of $(h-z_i)^2$ above is a constant which depends only on the coordinates of the vertices of f. (In fact, it is easy to verify that this constant coefficient is equal to $area(f)/((z_j-z_i)(z_k-z_i))$.) □

We sort the vertices of $\mathcal{P}$ by non-decreasing z-coordinates, as $v_1, v_2, \ldots, v_n$, classify each facet $f \in \mathcal{P}$ as a front, back, or parallel facet, and compute $area(f)$. We set the active-area term to zero, the inactive-area term to the total area of the back facets, and the current estimate of the minimum contact-area to their

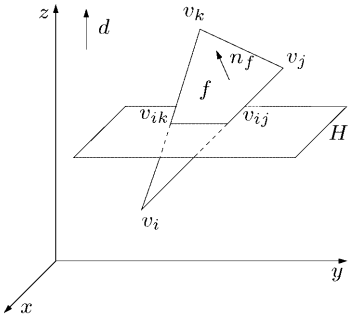


Fig. 3. *Intersection of H and active front facet f.*

sum. We then scan the vertices in their sorted order. Let v_ℓ be the current vertex, $1 \leq \ell < n$. For each facet f incident to v_ℓ, we do the following:

Case 1: *v_ℓ is the lowest vertex of f.* (Thus, f changes from inactive to active at v_ℓ.) If f is a back facet, then we subtract $area(f)$ from the inactive-area term. Using Lemma 1, we compute and add the expression $a_f h^2 + b_f h + c_f$ to the active-area term, thereby including the contribution of f^+ to the total contact-area when H is between z_ℓ and $z_{\ell+1}$. If f is a front facet, then we update only the active-area term to include the contribution of f^- to the total contact-area.

Case 2: *v_ℓ is the highest vertex of f.* This is symmetric to Case 1.

Case 3: *v_ℓ is the middle vertex of f.* Here f continues to be active, but the active-area term must be updated since H intersects a different edge of f above v_ℓ than it did below. We perform this update using Lemma 1.

After all facets incident to v_ℓ have been processed, we have a new active-area term $Ah^2 + Bh + C$, valid for h in the interval $[z_\ell, z_{\ell+1}]$. We minimize this using standard techniques from calculus and update the current estimate of the minimum contact-area. The running time is dominated by the time to sort; the time to process any vertex is proportional to its degree, hence $O(n)$ in total.

Theorem 1. *The contact-area version of Problem 1 can be solved in $O(n \log n)$ time for a convex polyhedron $\mathcal{P}$ with n vertices ($O(n)$ time if the vertices are given in sorted order in the decomposition direction $\mathbf{d}$).*

3.1 Experimental Results

We have implemented the above algorithm in C++, using floating point computations in double-precision, and tested it on randomly-generated convex polyhedra. The tests were done on a SUN Ultra 10 Sparc machine with 256 MB of main memory and a 440 MHz processor.

For our test polyhedra, we used `qhull` [1] to generate $n+1$ points at random on a cone whose major axis was along the z-axis, for n in the range 20,000 to 200,000 (the additional point was the apex of the cone). We then rotated the cone by a randomly-chosen angle to make it non-symmetric about the origin.

For each n, we generated two non-symmetric test polyhedra in this fashion. Table 1 shows a subset of our results. (All the results are not shown due to space constraints.) For each input size, the running times on the two polyhedra were nearly the same; therefore, we averaged the times. This closeness in run times is to be expected since the complexity of the algorithm depends primarily on the graph structure of $\mathcal{P}$, which is the same regardless of orientation.

Table 1. Minimum support contact-area and support volume for convex polyhedra generated from random points on a cone; each polyhedron has been rotated by the indicated angle to make it non-symmetric about the origin. Here "non-decomp area" and "non-decomp volume" refer to the contact-area and volume when the polyhedra are built without decomposition; observe the significant reductions achieved via decomposition.

Support Contact-area						**Support Volume**			
#verts n	angle	min contact-area	h_{min}	mean time (s)	non-decomp contact-area	min volume	h_{min}	mean time (s)	non-decomp volume
20001	200	7752.6	-3.2	.6	55300.5	10999.5	-21.1	4.2	965039.4
	37	11705.0	2.3		57111.7	26389.1	21.5		1107383.9
40001	40	12193.1	2.7	1.3	58304.3	24003.9	21.2	8.6	947591.6
	75	3973.7	.7		52005.6	3631.2	-4.4		994452.0
60001	240	7037.8	-.5	2.0	55082.1	12900.6	-4.1	12.7	815518.8
	112	5315.5	-.5		55068.2	8118.6	2.4		865432.8
80001	80	2714.5	0	2.8	52272.1	1359.2	-4.2	17.3	1014386.0
	150	10108.5	-1.5		51825.6	20249.8	-24.6		887435.1
100001	280	2649.1	0	3.6	52160.8	1349.3	-4.2	21.5	1014432.7
	187	3447.8	-3.6		59820.0	1068.5	-10.7		1033711.3

4 Decomposing Convex Polyhedra: Support Volume

For any given decomposition plane $H : z = h$, the expression for the support volume consists of the *inactive-volume* term and the *active-volume* term, which are, respectively, the total volumes of the support polyhedra contributed by the inactive and the active facets. If we move H up or down without crossing a vertex, then the active-volume and the inactive-volume both change (unlike contact-area, where only the active-area changes). Lemma 2 and Lemma 3 (proofs omitted here) establish the dependence of these terms on h.

Lemma 2. *For the decomposition plane $H : z = h$, the active-volume term is of the form $Ah^3 + Bh^2 + Ch + D$, where the coefficients A, B, C, and D depend only on the coordinates of the vertices of the active facets.*

Lemma 3. *For the decomposition plane $H : z = h$, the inactive-volume term is of the form $Ch+D$, where the coefficients C and D depend only on the coordinates of the vertices of the inactive facets.*

The volume-minimization algorithm is similar to the one in Section 3, except that we minimize the sum of the active-volume and inactive-volume terms.

Theorem 2. *The support volume version of Problem 1 can be solved in* $O(n \log n)$ *time for a convex polyhedron* $\mathcal{P}$ *with* n *vertices* ($O(n)$ *time if the vertices are given in sorted order in the decomposition direction* $\mathbf{d}$*).*

This algorithm has also been implemented; Table 1 shows some of the results.

5 Decomposing Non-convex Polyhedra

Despite the challenges posed by non-convex polyhedra (Figure 2), we show that they can be handled in essentially the same way as convex polyhedra after doing some initial processing. We focus on the contact-area version of Problem 1. (The volume problem is easier, because parallel facets do not contribute to support volume—as they do to contact-area—and so can be ignored.) We partition each front or back facet of $\mathcal{P}$ into two classes of triangles, called black and gray. (One of these classes may be empty.) A black triangle t will always be completely in contact with supports, regardless of the position of H; thus, it always contributes $area(t)$ to the total contact-area and can be ignored for minimization purposes. A gray triangle t will contribute an amount that is a quadratic function of the height of H, and so needs to be accounted for. A parallel facet of $\mathcal{P}$ is partitioned into three classes of triangles, called black, gray, and white. (Up to two of these classes may be empty.) Black and gray triangles are as above; a white triangle will never be in contact with supports, regardless of the height of H, and so can be ignored. Thus, only gray facets are relevant to the minimization problem. We will see that these can be handled as in the convex case.

5.1 Black, Gray, and White Triangles

Imagine that $\mathcal{P}$ is built, without decomposition, in direction $\mathbf{d}$. Consider the supports (if any) that are in contact with a front facet f (due to a back facet "above" it in direction $\mathbf{d}$). Their *footprint* on f, i.e., their intersection with f, is a collection of polygons, called *black polygons* and their triangulation yields the set, B_f, of *black triangles* associated with f. For any point p in a black triangle, the ray emanating from p in this direction must intersect $\mathcal{P}$. If p' is the first intersection of the ray and $\mathcal{P}$—not counting p—then the segment $\overline{p'p}$ is the support for p'. The complement of the black polygons on f is a collection of *gray polygons*, and their triangulation yields the set G_f, of *gray triangles* associated with f. No point in a gray triangle is in contact with supports for build direction $\mathbf{d}$. If f is a back facet of $\mathcal{P}$, then a symmetric discussion applies w.r.t. building $\mathcal{P}$ without decomposition in direction $-\mathbf{d}$.

Next, consider a parallel facet f. Imagine that we build $\mathcal{P}$ without decomposition in direction $\mathbf{d}$, and, independently, in direction $-\mathbf{d}$. The *black* (resp. *gray*, *white*) polygons consist of all points on f that are in contact with supports for both (resp. exactly one, neither) direction. The triangulations of these polygons yield the sets B_f, G_f, and W_f of *black*, *gray*, and *white* triangles on f.

Lemma 4. *Let $\mathcal{P}$ be built via decomposition and let $H : z = h$ be any decomposition plane. Consider a black or gray triangle from a front, back, or parallel facet, or a white triangle from a parallel facet. The contribution of the black or white triangle to the total contact-area is a constant independent of h. The contribution of the gray triangle is a quadratic function of h if the triangle is active, and a constant otherwise.*

Proof. (Sketch) W.l.o.g., let t be a black triangle from a front facet. (The discussion for back and parallel facets is similar.) Assume that t is active, and let $t^+ = t \cap \mathcal{P}^+$ and $t^- = t \cap \mathcal{P}^-$. Thus, all of t^- will require support. $\mathcal{P}^+$ is built in direction $\mathbf{d}$, and the part of $\mathcal{P}^+$ that is directly above t^+ is the same as the part of $\mathcal{P}$ that would be directly above t^+ if $\mathcal{P}$ were built without decomposition in direction $\mathbf{d}$. Since t^+ is in contact with supports in the latter case, it is also in contact with supports in the case with decomposition. Thus, all of t is in contact with supports and it always contributes $area(t)$ to the total contact-area. A similar argument applies if t is inactive.

Next, let t be an active gray triangle. As above, $t^- \in \mathcal{P}^-$ will be in contact with supports. However, no part of t^+ will be in contact with supports, since in the case where $\mathcal{P}$ is built in direction $\mathbf{d}$, without decomposition, no part of t is in contact with supports. Therefore, t's contribution to the total contact-area depends on the height of H and a proof similar to that of Lemma 1 shows that this is quadratic in h. If t is inactive, then it is either not in contact with supports at all or is completely in contact, so the contribution is either zero or $area(t)$.

If t is a white triangle from a parallel facet, then clearly no part of it is in contact with supports in $\mathcal{P}^+$ or in $\mathcal{P}^-$, so its contribution is always zero. □

5.2 Computing Black, Gray, and White Triangles

We use *cylindrical decomposition* [7]. From each edge, e, of each back facet, b, we erect a strip, $V_{e,b}$, which passes exactly through e and extends vertically downwards, in direction $-\mathbf{d}$. As soon as a part of $V_{e,b}$ intersects another facet of $\mathcal{P}$ (which must be a front facet), we stop propagating that part below the intersected facet; however, we continue propagating the remaining parts of $V_{e,b}$. Each such intersection of $V_{e,b}$ with a front facet will be a part of the footprints that we are trying to compute.

To perform this step efficiently, we compute the intersection of each front facet with $V_{e,b}$ to get a set, L, of line segments. We do a trapezoidal decomposition [3] of $L \cup \{e\}$ in the plane of $V_{e,b}$. We identify each trapezoid in this decomposition that is adjacent to e at the top and to a line segment $\ell \in L$ at the bottom. The bottom edge of this trapezoid is one of the sought intersections of $V_{e,b}$ with a front facet. We store this edge with the front facet that generated ℓ.

However, not all footprints on front facets will be discovered by the above process. For instance, if the projection of b completely covers a front facet, f, below it, then none of the strips $V_{e,b}$ erected from b will intersect f, and, yet, supports for b will rest on f. To handle such situations, we also erect from each

edge, e, of each front facet, f, a strip $V_{e,f}$ vertically upwards, stopping the propagation of any part of the strip as soon as it intersects a (back) facet above it, while continuing to propagate other parts. To compute these intersections we do a trapezoidal decomposition of the set $L' \cup \{e\}$, where L' contains the intersections of all the back facets with $V_{e,f}$. For each trapezoid in this decomposition that is incident to e at the bottom and to a segment $\ell' \in L'$ at the top, we take the top edge of the trapezoid as the intersection of $V_{e,f}$ with the back facet that generated ℓ', and store it with that back facet.

After we have done this for all front and back facets, we have associated with each facet a list of line segments corresponding to intersections of the different vertical strips with the facet. Since a strip is not propagated below an intersected facet, it is easy to see that the line segments associated with a facet are non-crossing (but may be touching). For each facet, we compute the arrangement [3] of the set consisting of the associated segments and the edges bounding the facet.

Let f be any front facet and let c be any cell of the arrangement computed on f. Then c is the footprint of a support on f for build direction $\mathbf{d}$, hence a black polygon, iff there is a cell c' on a back facet b above f, such that c' projects to c. (The cells c and c' form the bottom and top facets of a support cylinder; the other facets of this cylinder are vertical and bounded below and above by edges of c and c'.) Any other cell of f is a gray polygon.

We can identify the black triangles of f directly (instead of first computing the black polygons) using an approach given in [2]. We first triangulate the cells of the arrangements on all the front facets. We make a list, F, of these triangles along with their centroids. We sort F lexicographically on the x-, y-, and z-coordinates of the centroids, taken in that order. We make a similar sorted list B for the back facets. Then a simultaneous scan of the two lists suffices to identify matching pairs of triangles, i.e., pairs where one triangle is from B and the other is from F such that the former is above the latter and projects to it. For each matching pair, the triangle from F is a black triangle; all unmatched triangles of F are gray triangles.

A symmetric approach yields the black and gray triangles for the back facets.

To compute the relevant triangles for a parallel facet f, we take the strip V_f which is in the plane of f and exactly contains it, intersect it with each back facet that is above f to get a set, A, of line segments, and do a trapezoidal decomposition of A. Let T_f be the set of trapezoids in this decomposition that are bounded from above by some segment of A, unbounded below, and intersect f. Compute a symmetric set T'_f w.r.t. the front facets below f. The black polygons on f are intersections of pairs from T_f and T'_f. Any part of a trapezoid in T_f or T'_f that is not a black polygon is a gray polygon. The complement of the union of the gray and black polygons on f is the set of white polygons. These sets can be found by a simple sort-and-scan step and the corresponding triangles can then be identified by triangulation.

Lemma 5. *The set of black, gray, and white triangles for all facets of an n-vertex polyhedron $\mathcal{P}$ can be computed in $O(n^2 \log n)$ time.*

5.3 The Algorithm for Contact-Area

We compute for each front, back and parallel facet of $\mathcal{P}$, a list of the black, gray, and white polygons. As discussed in Section 5.1, only the gray triangles are relevant when decomposing $\mathcal{P}$ to minimize the contact-area of supports. We store the subdivision defined by the union of the set of gray triangles in a doubly-connected edge list and perform a sweep over them as in Section 3 to compute the optimum decomposition plane H. There are $O(n^2)$ gray triangles and the sweep therefore takes time $O(n^2 \log n)$. (Even though the algorithm in Section 3 is for convex polyhedra, the sweep does not depend on convexity *per se*, and so it works for our collection of gray triangles as well.)

As noted earlier, essentially the same approach works for volume minimization also. We will see in Section 5.4 that the size of the decomposition can be controlled in $O(n \log n)$ time. We conclude:

Theorem 3. *The contact-area and support volume versions of Problem 1 can be solved in $O(n^2 \log n)$ time for a non-convex polyhedron $\mathcal{P}$ with n vertices.*

5.4 Controlling the Size of the Decomposition

The optimal plane computed by the algorithm in Section 5.3 could decompose a non-convex polyhedron $\mathcal{P}$ into as many as $\Theta(n)$ polyhedra, which is undesirable since it increases the cost of re-assembling $\mathcal{P}$. Ideally, the designer should be able to specify an integer K, and the algorithm should compute, among along all possible planes that generate no more than K polyhedra, a plane which is optimal w.r.t. support contact-area or volume. We show how this can be done via a preprocessing step. The idea is to partition the z-axis into $O(n)$ intervals, I_j, such that all planes whose heights are in I_j decompose $\mathcal{P}$ into the same number, k_j, of polyhedra. We then run the sweep algorithm of the previous section but do the minimization step only in those intervals I_j for which $k_j \leq K$.

Let $z_1 < z_2 < \cdots < z_t$, $t \leq n$, be the distinct z-coordinates of the n vertices of $\mathcal{P}$. The preprocessing involves two sweeps. The first sweep is in the positive z direction and it computes a set of intervals on the z-axis and associates with each interval an integer which is the number of connected components of $\mathcal{P}^-$ generated by any plane whose height is in the interval. Observe that the number of connected components of $\mathcal{P}^-$ w.r.t. a plane of height z_j is the same as the number w.r.t. a plane whose height is anywhere in the interval (z_j, z_{j+1}); let this number be k_j^-. Thus, the first sweep computes intervals of the form $[z_j, z_{j+1})$ and associates with each the integer k_j^-. Symmetrically, the second sweep is in the negative z direction and it computes intervals of the form $(z_j, z_{j+1}]$ and associates with each an integer k_j^+, which is the number of connected components of $\mathcal{P}^+$ w.r.t. any plane whose height is in $(z_j, z_{j+1}]$. Once these two sets of intervals have been computed, a single scan of them suffices to compute the desired intervals I_j and the corresponding integers k_j. Specifically, each interval I_j is either of the form $[z_j, z_j]$, with $k_j = k_j^- + k_{j-1}^+$, or of the form (z_j, z_{j+1}), with $k_j = k_j^- + k_j^+$.

Consider the first sweep. At any time, the vertices of the different connected components of $\mathcal{P}^-$ form a collection of disjoint sets. We maintain these using

a Union-Find-Makeset data structure. We initialize the structure to empty and set the current number, c, of connected components of $\mathcal{P}^-$ to zero. Let z_j be the current z-coordinate in the sweep and let V_j be the set of vertices of $\mathcal{P}$ at this z-coordinate. We consider each vertex $v \in V_j$ in turn and process it as follows: We create a new set containing just v and increment c by one. Then for each neighbor, u, of v such that u is already in the Union-Find-Makeset data structure we do the following: If u and v are in different connected components, then we union the sets containing u and v, and decrement c by one. Notice that in this sweep, we only "add" edges to $\mathcal{P}^-$, so the connected components of $\mathcal{P}^-$ always merge, never split. Thus, a Union-Find-Makeset structure suffices to maintain the connected components of $\mathcal{P}^-$. After all vertices of V_j have been processed, we set k_j^- to c. At the end of the sweep, all the intervals $[z_j, z_{j+1})$, and their associated integers k_j^- will have been computed. The running time is dominated by the $O(n \log n)$ time for the sorting.

6 Conclusion

We have presented a new decomposition-based approach to LM, which reduces substantially the support requirements of the process, while also realizing other benefits as discussed in Section 1. Throughout, we have assumed a fixed decomposition direction $\mathbf{d}$ and found an optimum plane that is normal to $\mathbf{d}$. A more challenging problem is to compute over all directions $\mathbf{d}$ an optimum decomposition plane (while also limiting the number of pieces). We are pursuing this problem currently and plan to report on it in a future paper.

References

1. C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, December 1996. (See also `http://www.geom.umn.edu/software/qhull/`).
2. B. Chazelle and L. Palios. Triangulating a nonconvex polytope. *Discrete & Computational Geometry*, 5:505–526, 1990.
3. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.
4. S. Fekete and J. Mitchell. Histogram decomposition and stereolithography, 1997. Manuscript. `http://www.zpr.uni-koeln.de/~paper/paper.php3?paper=280`.
5. I. Ilinkin, R. Janardan, J. Majhi, J. Schwerdt, M. Smid, and R. Sriram. A decomposition-based approach to layered manufacturing, 2001. CS-TR-041, Dept. of Computer Science & Engineering, Univ. of Minnesota, 2000. `http://www.cs.umn.edu/~janardan/decomp.pdf`.
6. C. Kai and L. Fai *Rapid Prototyping: Principles and applications in manufacturing*. John-Wiley & Sons, 1997.
7. K. Mulmuley. *Computational Geometry: An introduction through randomized algorithms*. Prentice–Hall, 1993.

When Can You Fold a Map?

Esther M. Arkin[1], Michael A. Bender[2], Erik D. Demaine[3], Martin L. Demaine[3], Joseph S.B. Mitchell[1], Saurabh Sethia[2], and Steven S. Skiena[2]

[1] Department of Applied Mathematics and Statistics, State University of New York, Stony Brook, NY 11794-3600, USA, {estie, jsbm}@ams.sunysb.edu.
[2] Department of Computer Science, State University of New York, Stony Brook, NY 11794-4400, USA, {bender, saurabh, skiena}@cs.sunysb.edu.
[3] Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, {eddemaine, mldemaine}@uwaterloo.ca.

Abstract. We explore the following problem: given a collection of creases on a piece of paper, each assigned a folding direction of mountain or valley, is there a flat folding by a sequence of simple folds? There are several models of simple folds; the simplest *one-layer simple fold* rotates a portion of paper about a crease in the paper by $\pm 180°$. We first consider the analogous questions in one dimension lower—bending a segment into a flat object—which lead to interesting problems on strings. We develop efficient algorithms for the recognition of simply foldable 1-D crease patterns, and reconstruction of a sequence of simple folds. Indeed, we prove that a 1-D crease pattern is flat-foldable by any means precisely if it is by a sequence of one-layer simple folds.

Next we explore simple foldability in two dimensions, and find a surprising contrast: "map" folding and variants are polynomial, but slight generalizations are NP-complete. Specifically, we develop a linear-time algorithm for deciding foldability of an orthogonal crease pattern on a rectangular piece of paper, and prove that it is (weakly) NP-complete to decide foldability of (1) an orthogonal crease pattern on a orthogonal piece of paper, (2) a crease pattern of axis-parallel and diagonal (45-degree) creases on a square piece of paper, and (3) crease patterns without a mountain/valley assignment.

1 Introduction

> The easiest way to refold a road map is differently.
> — Jones's Rule of the Road (M. Gardner [6])

Perhaps the best-studied problem in origami mathematics is the characterization of flat-foldable crease patterns. A crease pattern is a straight-edge embedding of a graph on a polygonal piece of paper; a flat folding must fold along all of the edges of the graph, but no more. For example, two crease patterns are shown in Figure 1. The first one folds flat into a classic origami crane, whereas the second one cannot be folded flat (unless the paper is allowed to pass through itself), even though every vertex can be "locally" flat folded.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 401–413, 2001.

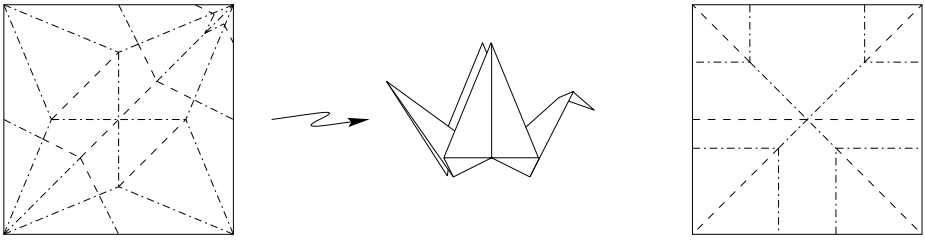


Fig. 1. Sample crease patterns. Left: the classic crane. Right: pattern of Hull [9], which cannot be folded flat, for any mountain-valley assignment.

The algorithmic version of this problem is to determine whether a given crease pattern is flat-foldable. The crease pattern may also have a direction of "mountain" or "valley" assigned to each crease, which restricts the way in which the crease can be folded. (Our figures adhere to the standard origami convention that valleys are drawn as dashed lines and mountains are drawn as dot-dashed lines.)

It is known that the general problem of deciding flat foldability of a crease pattern is NP-hard (Bern and Hayes [2]). In this paper, we consider the important and very natural case of recognizing crease patterns that arise as the result of flat foldings using *simple foldings*. In this model, a flat folding is made by a sequence of *simple folds*, each of which folds one or more layers of paper along a single line segment. As we define in Section 2, there are different types of simple folds (termed "one-layer," "some-layers," and "all-layers"), depending on how many layers of paper are required or allowed to be folded along a crease.

Note that not every flat folding can be achieved by a simple folding. For example, the crane in Figure 1 (top) cannot be made by a simple folding. Also, the hardness gadgets of [2] require nonsimple folds.

The problem we study in this paper is that of determining whether a given crease pattern (usually with specified mountain and valley assignments) can be folded flat by a sequence of simple folds, and if so, to construct such a sequence of folds.

Several of our results are based on the special case in which the creases in the piece of paper are all parallel to one another. This case can be seen to be equivalent to a *one-dimensional* folding problem of folding a line segment ("paper") according to a set of prescribed crease points (possibly labeled "mountain" or "valley"). We will therefore refer to this special case, which has a rich structure of its own, as the "1-D" case to distinguish it from the general 2-D problem. In contrast to the 2-D problem, we show that 1-D flat foldability is equivalent to 1-D simple foldability.

Motivation. In addition to its inherent interest in the mathematics of origami, our study is motivated by applications in sheet metal and paper product manufacturing, where one is interested in determining if a given structure can be manufactured using a given machine. (See references cited below.) While origamists

can develop particular skill in performing nonsimple folds to make beautiful artwork, practical problems of manufacturing with sheet goods require simple and constrained folding operations. Our goal is to develop a first suite of results that may be helpful towards a fuller algorithmic understanding of the several manufacturing problems that arise, e.g., in making three-dimensional cardboard and sheet-metal structures.

Related Work. Our problem is related to the classic combinatorics question of *map folding* [15]. This question asks for the *number* of foldings of a particular crease pattern, namely an $m \times n$ rectangular grid, by a sequence of simple folds. See also the discussion in Gardner's book [6]. In contrast with this combinatorics question, we study the algorithmic complexity of the decision problem, also in some more general instances of crease patterns.

The mathematical and algorithmic problems arising in the study of flat origami have been examined by several researchers, e.g., Hull [9], Justin [10], Kawasaki [12], and Lang [13]. Of particular relevance to our work is the paper of Bern and Hayes [2], in which the general problem of deciding flat foldability of a crease pattern is shown to be strongly NP-hard. Demaine et al. [4] used computational geometry techniques to show that any polygonal (connected) silhouette can be obtained by simple folds from a rectangular piece of paper.

There has been quite a bit of work on the related problems of manufacturability of sheet metal parts (see, e.g., [21]) and folding cartons (see, e.g., [14]). Existing CAD/CAM techniques (including BendCad and PART-S) rely on worst-case exponential-time state space searches (using the A* algorithm). In general, the problem of bend sequence generation is a challenging (and provably hard [1]) coordinated motion planning problem. For example, Lu and Akella [14] utilize a novel configuration-space formulation of the folding sequence problem for folding cartons using fixtures; their search, however, is still worst-case exponential time. Our work differs from the prior work on sheet metal and cardboard bending in that the structures we are folding are ultimately "flat" in their folded states (all bend angles in the input crease pattern are $\pm 180°$, according to a mountain-valley assignment that is part of the input crease pattern). Also, we are concerned only with the feasibility of the motion of the (stiff) material that is being folded – does it collide with itself during the folding motion? We are not addressing here the issues of reachability by the tools that perform the folding. As we show, even with the restrictions that come with the problems we study, there is a rich mathematical and algorithmic theory of foldability.

Summary of Our Results.[1]

(1) We analyze the 1-D one-layer and some-layers cases, giving a full characterization of flat-foldability and an $O(n)$ algorithm for deciding foldability and producing a folding sequence, if one exists.

(2) We analyze the 1-D all-layers case as a "string folding" problem. In addition to a simple $O(n^2)$ algorithm, we give an algorithm utilizing suffix trees that

[1] Due to space limitations, many of the proofs and details are omitted from this extended abstract. The reader is referred to the full paper.

requires time linear in the bit complexity of the input, and a randomized algorithm with expected $O(n)$ running time.

(3) We give a linear-time algorithm for deciding foldability of orthogonal crease patterns on a rectangular paper (the "map folding problem"), in the one-, some-, and all-layers cases, based on our 1-D results.

(4) We prove that it is (weakly) NP-complete to decide foldability of an orthogonal crease pattern on a piece of paper that is more general than a rectangle: a simple orthogonal polygon.

(5) We also prove that it is (weakly) NP-complete to decide foldability of a rectangular piece of paper with a crease pattern that includes *diagonal* creases (at 45-degrees), in addition to axis-parallel creases.

(6) We show that it is (weakly) NP-complete to decide foldability of a orthogonal piece of paper having a crease pattern for which no mountain-valley assignment is given.

2 Definitions

We are concerned with foldings in one and two dimensions. A one-dimensional piece of paper is a (line) *segment* in $\mathbf{R}^1$. A two-dimensional piece of paper is a (connected) *polygon* in $\mathbf{R}^2$, possibly with holes. In both cases, the paper is folded through one dimension higher than the object; thus, segments are folded through $\mathbf{R}^2$ and polygons are folded through $\mathbf{R}^3$. *Creases* have one less dimension; thus, a crease is a point on a segment and a line segment on a polygon.

A *crease pattern* is a collection of creases on the piece of paper, no two of which intersect except at a common endpoint. A *folding* of a crease pattern is an isometric embedding of the piece of paper, bent along every crease in the crease pattern (and not bent along any segment that is not a crease). In particular, each facet of paper must be mapped to a congruent copy, the connectivity between facets must be preserved, and the paper cannot pass through itself. See Figure 2.

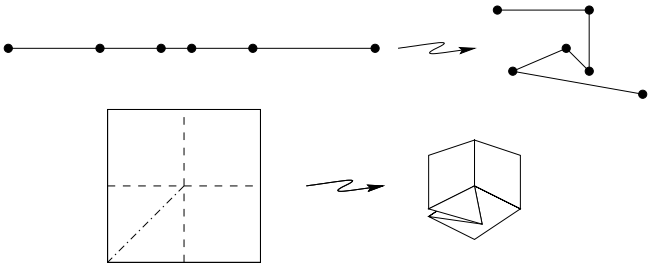


Fig. 2. Sample nonflat foldings in one and two dimensions.

A *flat folding* has the additional property that it lies in the same space as the unfolded piece of paper. That is, a flat folding of a segment lies in $\mathbf{R}^1$, and a flat

folding of a polygon lies in $\mathbf{R}^2$. In reality, there can be multiple layers of paper at a point, so the folding really occupies a finite number of infinitesimally close copies of $\mathbf{R}^1$ or $\mathbf{R}^2$. More formally, a flat folding can be specified by a function mapping the vertices to their folded positions, together with a partial order of the facets of paper that specifies their overlap order [2,9,13].

If we orient the piece of paper to have a top and bottom side, we can talk about the *direction* of a crease in a flat folding. A *mountain* brings together the bottom sides of the two adjacent facets of paper, and a *valley* brings together the top sides. A *mountain-valley assignment* is a function from the creases in a crease pattern to $\{M, V\}$. Together, a crease pattern and a mountain-valley assignment form a *mountain-valley pattern*.

This paper is concerned with the following generic question.

Problem 1. Simple Folding: Given a mountain-valley pattern, is there a simple folding satisfying the specified mountains and valleys? If so, construct such a simple folding.

There are three natural versions of this problem, depending on the type of "simple folds" allowed. In general, a *simple folding* is a sequence of simple folds. Each simple fold takes a flat-folded piece of paper, and folds it into another flat folding using additional creases. There are three types of simple folds: one-layer, all-layers, and some-layers.

A *one-layer simple fold* f is a crease on the folded piece of paper, together with a direction. If we look at the unfolded piece of paper, then f partitions it into two parts, call them A and B. Performing f corresponds to rotating A about f by $\pm 180°$, where $\pm$ depends on the fold direction, if this does not cause the paper to cross itself. This makes just a single crease, which is what we mean by folding one layer of paper.

An *all-layers simple fold* f is also a crease together with a direction. Now we consider the partition of the flat folding (instead of the unfolded piece of paper) by f into two parts, call them A and B again. Performing f corresponds to rotating (all layers of) A about f by $\pm 180°$. This makes a crease through all of the layers of paper at f. Note that this type of fold can never cause the paper to cross itself.

Finally, a *some-layers simple fold* f is the most general. It takes some of the top [bottom] layers of A, and rotates them about f by $180°$ $[-180°]$, provided the paper does not cross itself.

3 1-D: One-Layer and Some-Layers

This section is concerned with the 1-D one-layer simple-fold problem. We will prove the surprising result that we only need to search for one of two local operations to perform. The two operations are called *crimps* and *end folds*, and are shown in Figure 3.

More formally, let $c_1, \ldots, c_n$ denote the creases on the segment, oriented so that c_i is left of c_j for $i < j$. Let c_0 $[c_{n+1}]$ denote the left [right] end of the

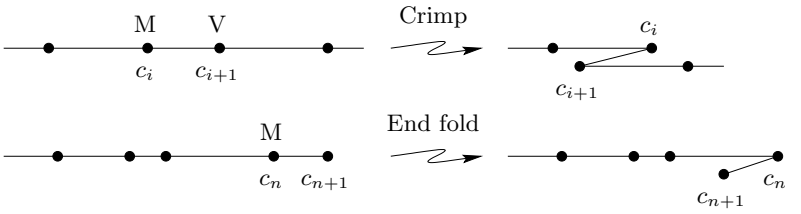


Fig. 3. The two local operations for one-dimensional one-layer folds.

segment. Despite the "c" notation (which is used for convenience), c_0 and c_{n+1} are not considered *creases*; instead they are called the *ends*.

First, a pair (c_i, c_{i+1}) of consecutive creases is *crimpable* if c_i and c_{i+1} have opposite directions and $|c_{i-1} - c_i| \geq |c_i - c_{i+1}| \leq |c_{i+1} - c_{i+2}|$. *Crimping* such a pair corresponds to folding c_i and then folding c_{i+1}, using one-layer simple folds.

Second, c_0 is a *foldable end* if $|c_0 - c_1| \leq |c_1 - c_2|$, and c_{n+1} is a *foldable end* if $|c_{n-1} - c_n| \geq |c_n - c_{n+1}|$. *Folding* such an end corresponds to performing a one-layer simple fold at the nearest crease (crease c_1 for end c_0, and crease c_n for end c_{n+1}).

We claim that one of the two local operations exists in any flat-foldable 1-D mountain-valley pattern. We claim further that an operation exists for any pattern satisfying a certain "mingling property." Specifically, a 1-D mountain-valley pattern is called *mingling* if for every sequence $c_i, c_{i+1}, \ldots, c_j$ of consecutive creases with the same direction, either (1) $|c_{i-1} - c_i| \leq |c_i - c_{i+1}|$; or (2) $|c_{j-1} - c_j| \geq |c_j - c_{j+1}|$. We call this the mingling property because, for maximal sequences of consecutive creases with the same direction, it says that there are folds of the opposite direction nearby. In this sense, the mountain-valley pattern is "crowded" (mountains and valleys must "mingle" together).

First we show (in the full paper) that mingling mountain-valley patterns include flat-foldable patterns:

Lemma 1. *Every flat-foldable 1-D mountain-valley pattern is mingling.*

Next we show (again, see the full paper) that having the mingling property suffices to imply the existence of a single crimpable pair or foldable end.

Lemma 2. *Any mingling 1-D mountain-valley pattern has either a crimpable pair or a foldable end.*

Ideally, we could show at this point that performing either of the two local operations preserves the mingling property, and hence a mountain-valley pattern is mingling precisely if it is flat-foldable. Unfortunately this is false; see the full paper. Instead, we must prove that flat foldability is preserved by each of the two local operations; i.e., if we treat the folded object from a single crimp as a new segment, it is flat-foldable.

Lemma 3. *Folding a foldable end preserves foldability.*

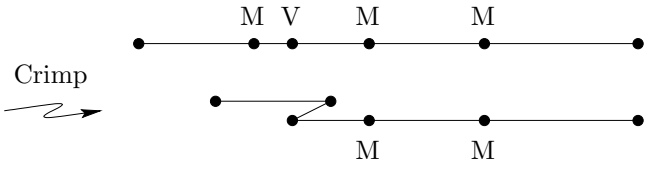


Fig. 4. A mingling mountain-valley pattern that when crimped is no longer mingling and hence not flat-foldable. Indeed, the original mountain-valley pattern is not flat-foldable.

Lemma 4. *Crimping a crimpable pair preserves flat foldability.*

Combining all of the previous results, we have the following:

Theorem 1. *Any flat-foldable 1-D mountain-valley pattern can be folded by a sequence of crimps and end folds.*

A particularly interesting consequence of this theorem is the following:

Corollary 1. *The following are equivalent for a 1-D mountain-valley pattern P:*

1. *P has a flat folding.*
2. *P has a some-layers simple folding.*
3. *P has a one-layer simple folding.*

Finally, let us show (in the full paper) that Theorem 1 leads to a simple linear-time algorithm:

Theorem 2. *The 1-D one-layer and some-layers simple-fold problems can be solved in $O(n)$ worst-case time on a machine supporting arithmetic on the input lengths.*

4 1-D: All-Layers Simple Folds

The 1-D all-layers simple-fold problem can be cast as an interesting "string folding" problem. (This folding problem is not to be confused with the well-known protein/string folding problem in biology [3].) The input mountain-valley pattern can be thought of as a string of lengths interspersed with mountain and valley creases. Specifically, we will assume that the input lengths are specified as integers or equivalently rational numbers. (Irrational numbers can be replaced by close rational approximations, provided the sorted order of the lengths is preserved.)

Thus, an input string is of the form $d_0\ c_1\ d_1\ c_2 \cdots c_{n-1}\ d_{n-1}\ c_n\ d_n$, where each $c_i \in \{M, V\}$ and each d_i is a positive rational number. We call each c_i and d_i a *symbol* of the string. It will be helpful to introduce some more uniform

notation for symbols. For a string S of length n, we denote the ith symbol by $S[i]$, where $1 \leq i \leq n$.

When we make an all-layers simple fold, we cannot "cover up" a crease except with a matching crease (which when unfolded is in fact the other direction), because otherwise this crease will be impossible to fold later. To formalize this condition, we define the *complement* of symbols in the string: $\text{comp}(d_i) = d_i$, $\text{comp}(M) = V$, and $\text{comp}(V) = M$. Finally, we call a crease (M or V symbol) $S[i]$ *allowable* if $S[i-x] = \text{comp}(S[i+x])$ for all $1 \leq x \leq \min(i-1, n-i)$, except that $S[1]$ and $S[n]$ (the ends) are allowed to be shorter than their complements.

Lemma 5. *A mountain-valley pattern can be folded by a sequence of all-layers simple folds precisely if there is an allowable fold, and the result after performing that fold has an allowable fold, and so on, until all creases of the segment have been folded.*

By Lemma 5, the problem of testing foldability reduces to repeatedly finding allowable folds in the string. Testing whether a fold at position i is allowable can clearly be done in $O(1 + \min\{i-1, n-i\})$ time, by testing the boundary conditions and whether $S[i-x] = \text{comp}(S[i+x])$ for $1 \leq x \leq \min(i-1, n-i)$. Explicitly testing all creases in this manner would yield an $O(n^2)$-time algorithm for finding an allowable fold (if one exists). Repeating this $O(n)$ times results in a naive $O(n^3)$ algorithm for testing foldability.

This cubic bound can be improved by being a bit more careful. In $O(n^2)$ time, we can determine for each crease $S[i]$ the largest value of k for which $S[i-x] = \text{comp}(S[i+x])$ for all $1 \leq x \leq k$. Using this information it is easy to test whether a crease $S[i]$ is allowable. After making one of these allowable folds, we can in $O(n)$ time update the value of x for each crease, and hence maintain the collection of allowable folds in linear time. This gives an overall $O(n^2)$ bound, which we now proceed to improve further.

We present two efficient algorithms for folding strings. The algorithm in Section 4.1 is based on suffix trees and runs in time linear in the bit complexity of the input. In Section 4.2, we use randomization to obtain a simpler algorithm that runs in $O(n)$ time.

4.1 Suffix-Tree Algorithm

In the full paper we prove the following:

Theorem 3. *A string S of length n can be tested for all-layers simple foldability, in time that is dominated by that to construct a suffix tree on S.*

The difficulty with the time bound is that sorting the alphabet seems to be required. Other than the time to sort the alphabet, it is possible to construct a suffix tree in $O(n)$ time [5]. To sort the alphabet in the comparison model, $O(n \log n')$ time suffices, where n' is the number of distinct input lengths. In particular, if the input lengths are encoded in binary, then the algorithm is linear in this bit complexity. On a RAM, the current state-of-the-art algorithm for integer sorting [20] uses $O(n(\log\log n)^2)$ time and linear space.

4.2 Randomized Algorithm

In this section we describe a simple randomized algorithm that solves the 1-D all-layers simple-fold problem in $O(n)$ time. There are two parts to the algorithm:

1. assigning labels to the input lengths so that two lengths are equal precisely if they have the same label; and
2. finding and making allowable folds.

The first part is essentially element uniqueness, and can be solved in linear expected time using hashing. For example, the dynamic hashing method described by Motwani and Raghavan [16] supports insertions and existence queries in $O(1)$ expected time. We can use this data structure as follows. For each input length, check whether it is already in the hash table. If it is not, we assign it a new unique identifier, and add it to the hash table. If it is, we use the existing unique identifier for that value (stored in the hash table). Let n' denote the number of distinct labels found in this process (or 2, whichever is larger).

For the second part, we will show that each performed fold can be found in $O(1 + r)$ time, where r is the number of creases removed by the discovered fold (in other words, the minimum length to an end of the segment to be folded). However, it is possible that the algorithm makes a mistake, and that some of the reported folds are not actually possible. Fortunately, mistakes can be detected quickly, and after $O(1)$ expected iterations the pattern will be folded. (Unless of course the pattern is not flat-foldable, in which case the algorithm reports this fact correctly.)

In the full paper we give details of the algorithm, and conclude:

Theorem 4. *The 1-D all-layers simple-fold problem can be solved in $O(n)$ expected time on a machine supporting random numbers and hashing of the input lengths.*

5 Orthogonal Simple Folds in 2-D

In this section, we generalize our results for 1-D simple folds to *orthogonal* 2-D crease patterns, which consist only of horizontal and vertical folds on a rectangular piece of paper, where horizontal and vertical are defined by the sides of the rectangular paper. In such a pattern, the creases must go all the way through the paper, because every vertex of a flat-foldable crease pattern has degree at least four [2,9]. Hence, the crease pattern is a *map* or grid of creases. Recall from Section 3 that the opposite holds in 1-D: one-layer and some-layers folds are equivalent to general flat-foldability.

In this section we handle all three cases of simple folds: one-, some-, and all-layers folds. To know what time bounds we desire, we must first discuss encoding the input. A natural encoding of maps specifies the height of each row and the width of each column, thereby using $n_1 + n_2$ space for an $n_1 \times n_2$ grid. The mountain-valley assignment, however, requires $\Theta(n_1 n_2)$ space to specify the

direction for each edge of the grid. Hence, our goal of linear time amounts to being linear in $n = n_1 n_2$.

It is easy to see that in exactly one of the two orientations, vertical or horizontal, there must be at least one crease line, all the way across the paper, that is entirely valley or mountain. (If there is no such crease, the pattern is unfoldable. And it cannot be that there is both a horizontal crease and a vertical crease all the way through the paper, since their intersection would be a vertex that is locally unfoldable.) Without loss of generality, assume it is horizontal. Let the set of these horizontal fold lines be $\mathcal{H}$.

We claim that all fold lines in $\mathcal{H}$ must be folded before any other fold. This is so because (1) folding along any vertical fold line v will lead to a mismatch of creases at the intersection of v with any unfolded elements of $\mathcal{H}$ and (2) horizontal folds not in $\mathcal{H}$ are not entirely mountain or valley and hence cannot be folded before some vertical fold is made. Thus we have a corresponding 1-D problem (one-, some- or all-layer folds) to solve with added necessary condition that the non-$\mathcal{H}$ folds must match up appropriately after all the folds in $\mathcal{H}$ are made. (The time spent of checking for this necessary condition can be attributed to the non-$\mathcal{H}$ folds that vanish after every fold.) Since $\mathcal{H}$ contains at least one fold, performing the $\mathcal{H}$ folds (strictly) reduces the size of the problem, and we continue. The base case consists of just horizontal or vertical folds, which corresponds to a 1-D problem. In summary we have:

Lemma 6. *If a crease pattern is foldable, it remains foldable after the folds in $\mathcal{H}$ have been made in any feasible way considering $\mathcal{H}$ to be a 1-D problem and ignoring other creases.*

To find $\mathcal{H}$ quickly we maintain the number of mountain and valley creases for each row and column of creases. We maintain these numbers as we make folds in $\mathcal{H}$. To do this we traverse all the creases that will vanish after a fold and decrement the corresponding numbers. The cost of this traversal is attributed to the vanishing creases. Every time the number of mountain or valley creases hits zero in a column or a row, we add the row or column to a list to be used as the new $\mathcal{H}$ in the next step. Thus,

Theorem 5. *The problem of deciding simple foldability of an orthogonal crease pattern on a rectangular piece of paper can be solved in linear time.*

6 Hardness of Simple Folds in 2-D

In this section we prove that the problem of deciding whether a 2-D axis-parallel mountain-valley pattern can be simply folded is (weakly) NP-hard, if we allow the initial paper to be an arbitrary orthogonal polygon. We also show that it is (weakly) NP-hard to decide whether a mountain-valley pattern on a square piece of paper can be folded by some-layers simple folds, if the creases are allowed to be axis-parallel *plus* at a 45-degree angle.

Both hardness proofs are based on a reduction from an instance of PARTITION: given a set X of n integers $a_1, a_2, \ldots, a_n$ whose sum is A, does there exist a set $S \subset X$ such that $\sum_{a \in S} a = A/2$? For convenience we define the set $\bar{S} = X \setminus S$. Also, without loss of generality, we assume that $a_1 \in S$.

The PARTITION problem is known to be (weakly) NP-hard [7]. We transform an instance of the PARTITION problem into an orthogonal 2-D crease pattern on a orthogonal polygon, as shown in Figure 5.

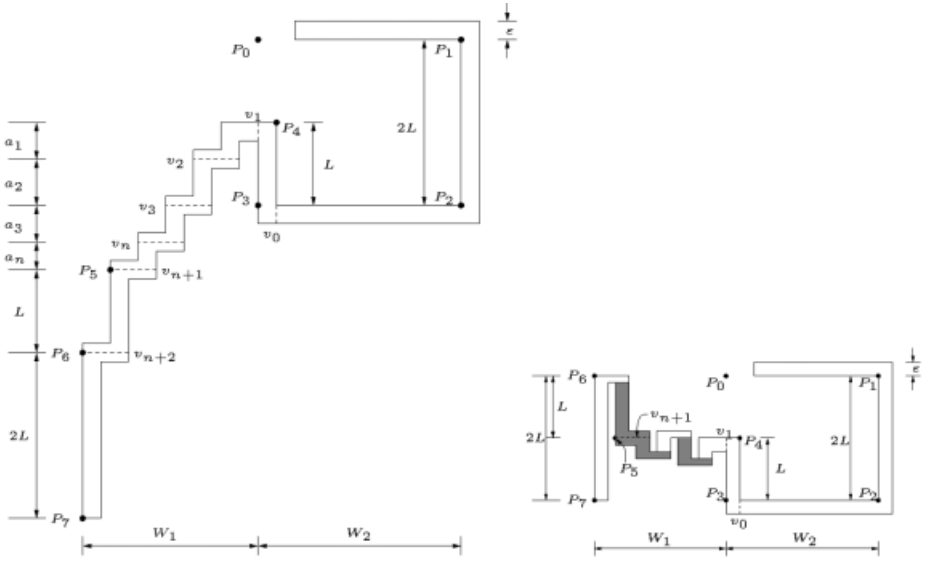


Fig. 5. Top: Hardness reduction from PARTITION problem. Bottom: Semi-folded staircase confined between y coordinates of P_1 and P_2. The top side of the paper is drawn white and the other side is drawn gray.

In the figure, all creases are valleys. There is a staircase of width ε corresponding to $a_1, \ldots, a_n$, where $0 < \varepsilon < 2/3$. There is a step in the staircase of length a_i corresponding to each element a_i in X. L is a constant greater than $A/2$. Also $W_2 > W_1$. Also let there be a coordinate system with horizontal x-axis and vertical y-axis.

Lemma 7. *If the* PARTITION *instance has a solution, then the crease pattern in Figure 5 is simply foldable.*

Lemma 8. *If the crease pattern in Figure 5 is simply foldable, there exists a solution to the* PARTITION *instance.*

Lemmas 7 and 8 imply the following theorem.

Theorem 6. *The problem of deciding simple foldability of a orthogonal paper with an orthogonal crease pattern is (weakly) NP-complete.*

In the full paper, we prove the following theorem, which shows that even on a rectangular piece of paper it is hard to decide foldability if, besides axis-parallel, there are creases in diagonal directions (45 degrees with respect to the axes):

Theorem 7. *It is (weakly) NP-complete to decide the foldability of an (axis-parallel) square sheet of paper with a crease pattern having axis-parallel creases and creases at the diagonal angles of 45 degrees with respect to the axes, for both all-layers and some-layers simple folds.*

The problem is open for the one-layer case.

7 No Mountain-Valley Assignments

An interesting case to consider is when all creases do not have mountain-valley assignment: Any crease can be folded in either direction. Even with this flexibility, we are able to show that the problem is hard (see the full paper for the proof):

Theorem 8. *The problem of deciding the foldability of a orthogonal paper with a crease pattern that does not have mountain-valley assignment is (weakly) NP-complete, for both the all-layers and some-layers cases.*

The problem is open for the one-layer case.

Acknowledgments. We thank Jack Edmonds for helpful discussions which inspired this research. E. Arkin acknowledges support from the NSF (CCR-9732221) and HRL Labs. M. Bender acknowledges support from HRL Labs. J. Mitchell acknowledges support from HRL Labs, the NSF (CCR-9732221), NASA (NAG2-1325), Northrop-Grumman, Sandia, Seagull Technology, and Sun Microsystems.

References

1. E. M. Arkin, S. P. Fekete, J. S. B. Mitchell, and S. S. Skiena. On the manufacturability of paperclips and sheet metal structures. In *Proc. of the 17th European Workshop on Computational Geometry*, pages 187–190, 2001.
2. M. Bern and B. Hayes. The complexity of flat origami. In *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 175–183, 1996.
3. P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. *J. of Computational Biology*, 5(3), 1998.
4. E. D. Demaine, M. L. Demaine, and J. S. B. Mitchell. Folding flat silhouettes and wrapping polyhedral packages: New results in computational origami. In *Proc. of the 15th ACM Symposium on Computational Geometry*, 1999.

5. M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. of the 38th Symp. on Foundations of Computer Science*, pages 137–143, 1997.
6. M. Gardner. The combinatorics of paper folding. In *Wheels, Life and Other Mathematical Amusements*, Chapter 7, pp. 60–73. W. H. Freeman and Company, 1983.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.
8. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. on Computing*, 13(2):338–355, 1984.
9. T. Hull. On the mathematics of flat origamis. *Congressum Numerantium*, 100:215–224, 1994.
10. J. Justin. Towards a mathematical theory of origami. In Koryo Miura, editor, *Proc. of the 2nd International Meeting of Origami Science and Scientific Origami*, pages 15–29, 1994.
11. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
12. T. Kawasaki. On the relation between mountain-creases and valley-creases of a flat origami. In H. Huzita, editor, *Proc. of the 1st International Meeting of Origami Science and Technology*, pages 229–237, Ferrara, Italy, December 1989. An unabridged Japanese version appeared in *Sasebo College of Technology Report*, 27:153–157, 1990.
13. R. J. Lang. A computational algorithm for origami design. In *Proc. of the 12th ACM Symposium on Computational Geometry*, pages 98–105, 1996.
14. L. Lu and S. Akella. Folding cartons with fixtures: A motion planning approach. *IEEE Trans. on Robotics and Automation*, 16(4):346–356, 2000.
15. W. F. Lunnon. Multi-dimensional map-folding. *The Computer Journal*, 14(1):75–80, 1971.
16. R. Motwani and P. Raghavan. *Randomized Algorithms*, Chapter 8.4, pages 213–221. Cambridge University Press, 1995.
17. B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. on Computing*, 17(6):1253–1262, 1988.
18. J. S. Smith. Origami profiles. *British Origami*, 58, 1976.
19. J. S. Smith. *Pureland Origami 1, 2, and 3*. British Origami Society. Booklets 14, 29, and 43, 1980, 1988, and 1993.
20. M. Thorup. Faster deterministic sorting and priority queues in linear space. In *Proc. of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 550–555, 1998.
21. C-H. Wang. *Manufacturability-driven decomposition of sheet metal*. PhD thesis, Carnegie Mellon University 1997. Technical report CMU-RI-TR-97-35.

Search Trees with Relaxed Balance and Near-Optimal Height

Rolf Fagerberg[1], Rune E. Jensen[2], and Kim S. Larsen[3]

[1] BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, building 540, DK-8000 Århus C, Denmark, rolf@brics.dk.
[2] ALOC Bonnier A/S, Buchwaldsgade 35, DK-5000 Odense C, Denmark, runej@aloc.dk.
[3] Department of Mathematics and Computer Science, University of Southern Denmark, Main Campus: Odense University, Campusvej 55, DK-5230 Odense M, Denmark, kslarsen@imada.sdu.dk.

Abstract. We introduce the relaxed k-tree, a search tree with relaxed balance and a height bound, when in balance, of $(1+\varepsilon)\log_2 n + 1$, for any $\varepsilon > 0$. The rebalancing work is amortized $O(1/\varepsilon)$ per update. This is the first binary search tree with relaxed balance having a height bound better than $c \cdot \log_2 n$ for a fixed constant c. In all previous proposals, the constant is at least $1/\log_2 \phi > 1.44$, where ϕ is the golden ratio.
As a consequence, we can also define a standard (non-relaxed) k-tree with amortized constant rebalancing per update, which is an improvement over the original definition.
Search engines based on main-memory databases with strongly fluctuating workloads are possible applications for this line of work.

1 Introduction

The k-trees [7] differ from other binary search trees in that the height can be maintained arbitrarily close to the optimal $\lceil \log_2 n \rceil$ while the number of rebalancing operations carried out in response to an update remains $O(\log n)$. The price to be paid is that the size of each rebalancing operation (the number of nodes which must be inspected) grows as we approach $\lceil \log_2 n \rceil$. More precisely, one can show that to obtain height less than $(1+\varepsilon)\log_2 n + 1$, rebalancing operations have size $\Theta(1/\varepsilon)$ in [7].

Thus, using k-trees, a trade-off between search time and rebalancing time becomes an option; the more interesting direction being the scenario where updates are infrequent compared with searches. Under such circumstances, it may be beneficial to spend more time on the occasional updates in order to obtain shorter search paths.

Search engines pose a particular search/update problem. Searching is dominant, but when updates are made, they often come in bursts because keywords originate from large sites. Equipping search trees with relaxed balance has been proposed as a way of being able to adapt smoothly to this ever-changing scenario.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 414-425, 2001.

Relaxed balance is the term used for search trees where updating and rebalancing have been uncoupled, most often through a generalization of the basic structure. The uncoupling is achieved by allowing rebalancing after different updates to be postponed, broken down into small steps, and interleaved.

The challenge for the designers of these structures is to ensure and be able to prove efficient rebalancing in this more general structure. If that problem is overcome, then the benefit is the extra flexibility. During periods with heavy updating, rebalancing can be decreased or even turned completely off to allow a higher throughput of updates as well as searches. When the update burst is over, the structure can gradually be rebalanced again. Since a search engine is in constant use, it is important that this rebalancing is also carried out efficiently, i.e., using as few rebalancing operations as possible.

Besides search engines, the flexibility provided by relaxed balance may be an attractive option for any database application with strongly fluctuating work loads.

Relaxed balance has been studied in the context of AVL-trees [1] starting in [10] and with complexities matching the ones from the standard case in [6]. The height bound for AVL-trees in balance is $\frac{1}{\log_2 \phi} \log_2 n > 1.44 \log_2 n$. In the context of red-black trees [4], relaxed balance has been studied starting in [8, 9] with results gradually matching the standard case [11] in [3,2,5]. The height bound for red-black trees in balance is $2 \log_2 n$. A more thorough introduction to relaxed balance as well as a comprehensive list of references can be found in [5].

In this paper, we first develop an alternative definition of standard k-trees. The purpose of this is both to cut down on the number of special cases, and to pave the way for an improved complexity result. Based on this, a relaxed proposal is given, and complexity results are shown. The complexity results are in the form of upper bounds on the number of rebalancing operations which must be carried out in response to an update.

It is worth noting that the alternative definition of k-trees, which is the starting point for the relaxed definition, also gives rise to an improved complexity result for the standard case: in addition to the logarithmic worst-case bound, rebalancing can now be shown to use amortized constant time as well.

2 *K*-Trees

The k-trees of [7] are search trees where all external nodes have the same depth and all internal nodes are either unary or binary. The trees are leaf-oriented, meaning that the external nodes contain the keys of the elements stored, and the internal nodes contain *routers*, which are keys guiding the search. Binary internal nodes contain one key, unary internal nodes contain no keys. To avoid arbitrarily deep trees, restrictions are imposed on the number of unary nodes: on any level of the tree, the first k nodes to the right of a unary node should (if present) be binary. As this does not preclude a string of unary nodes starting at the root, it is also a requirement in [7] that the rightmost node on each level is binary.

It is intuitively clear that a larger value of k gives a lower density of unary nodes, which implies a smaller height for a given number n of stored keys. The price to be paid is an increased amount of rebalancing work per update—more precisely, one can show that with the proposal of [7] a height bound of $(1 + \Theta(1/k)) \log_2 n + 1$ can be maintained with $O(k \log n)$ work per update. Thus, k-trees offer a tradeoff between the height bound and the rebalancing time, and furthermore allow for height bounds of the form $c \cdot \log_2 n + 1$, where the constant c can be arbitrarily close to one.

While the possibility of such a tradeoff is a very interesting property, the k-trees of [7] also have some disadvantages. One is that, unlike red-black trees, for example, they do not have an amortized constant bound on the amount of rebalancing work per update. As a counterexample, consider a series of alternating deletions and insertions of the smallest key in a complete binary tree of height h, having $2^h - 1$ binary nodes. Since the tree does not contain any unary nodes, it is a valid k-tree for any k, and it is easy to verify that the rebalancing operations described in [7] will propagate all the way to the root after each update. Another disadvantage is the lack of left-right symmetry in the definition of k-trees in [7], forcing operations at the rightmost path in the tree to be special cases. This approximately doubles the number of operations compared with the number of essentially different operations.

We therefore propose an alternative definition of k-trees. This definition will allow us to add relaxed balance using a relatively simple set of rebalancing operations, for which we can prove an amortized complexity of $O(1)$ per update. Additionally, this enables us to define a new non-relaxed k-tree with the same complexity, simply by deciding to rebalance completely after each update. This is an improvement of the result from [7].

Our basic change is in the way the density of unary nodes is kept low. On each level in the tree, except the topmost, we divide the nodes into *groups* of $\Theta(k)$ neighboring nodes. Thus, a group is simply a contiguous segment of a given level. The groups are implemented by marking the leftmost node in each group, using one bit. Furthermore, in each group, we allow *two* unary nodes, contrary to the original proposal [7] which considers unary nodes one by one. Intuitively, this is what gives the amortized constant rebalancing per update. The top of the tree is managed differently, as the levels are too small to contain a group.

Definition 1. *For any integer $k \geq 2$, a* symmetric k-tree *is a tree containing unary and binary nodes, where all external nodes have the same depth. The topmost $1 + \lceil \log k \rceil$ levels consist of binary nodes only. In level number $2 + \lceil \log k \rceil$ from the top, there is at least one binary node. On the rest of the levels in the tree, the internal nodes are divided into groups of neighboring nodes. Each group contains at least $2k$ nodes and at most $4k$ nodes. In each group, at most two of the nodes are unary.*

We call level number $2 + \lceil \log k \rceil$ the *buffer level*. For the number S of nodes in the buffer level, we have $2k \leq S = 2^{\lceil \log k \rceil + 1} < 4k$.

The tree is turned into a search tree by storing elements in the external nodes and routers in the binary internal nodes in accordance with the usual in-order

ordering. Searching proceeds as in any binary search tree, except that unary nodes (which contain no keys) are just passed through.

To add relaxed balance, we must allow insertions and deletions to proceed without immediate rebalancing, and therefore must relax the structural constraints of Definition 1. To achieve this, we allow nodes of degree larger than two, and allow an arbitrary number of unary nodes in a group. To keep the actual trees binary, we use the standard method [4] of representing i-ary nodes by binary subtrees with $i-1$ nodes, indicating the root of each subtree by one bit of information, traditionally termed a red/black color.

We define the *black depth* of a node as the number of black nodes (including itself, if black) on the path to the root. The *black level* number i consists of all black nodes having the black depth i. Note that for any node, the number of black nodes below it on a path to an external node is the same for all such paths. We call this number the *black height* of the node.

Definition 2. *For any integer $k \geq 2$, a* relaxed k-tree *is a tree containing unary and binary nodes, where nodes are colored either black or red. The root, the unary nodes and the external nodes are always black. All external nodes in the tree have the same black depth. In the topmost $1 + \lceil \log k \rceil$ black levels there are no unary nodes, and no node has a red child. In black level number $2 + \lceil \log k \rceil$, there is at least one binary node. On the rest of the black levels in the tree, the internal nodes are divided into groups of neighboring nodes, with each group containing at least $2k$ nodes and at most $4k$ nodes.*

A relaxed k-tree is a standard (symmetric) k-tree if all nodes are black, and no group contains more than two unary nodes. It turns out that in our relaxed search trees, we also need to allow *empty* external nodes, i.e., external nodes with no elements. Later in this paper, we give a set of rebalancing operations which can turn a relaxed k-tree with empty external nodes into a standard k-tree without empty external nodes, and we give bounds on the number of operations needed for this.

3 Height Bound

By the *height* of a tree we mean the maximal number of edges on any path from the root to an external node. We now show that the height of symmetric k-trees is just as good as that of the original version in [7].

Theorem 1. *The height of a symmetric k-tree with n external nodes is bounded by $\log_\alpha n + 1$, where $\alpha = 2 - 1/k$.*

Proof. On any level, except the buffer level, at most two out of each $2k$ nodes are unary. It follows that the number of nodes for each new level, except the buffer level, increases at least by a factor of $2(1-1/k)+1/k = 2-1/k = \alpha$. For the buffer level, the number of nodes does not decrease. Hence, a tree of height h contains at least α^{h-1} external nodes. □

Using the identity $\log_\alpha(x) = \log_2(x)/\log_2(\alpha)$, this height bound may be stated in a more standard way as $c \cdot \log_2 n + 1$. Examples of values of c attainable by varying k are shown in Table 1.

Table 1. Corresponding values of k and c.

k	2	3	4	5	6	7	8	9	10	20	50	100
c	1.71	1.36	1.24	1.18	1.14	1.12	1.10	1.09	1.08	1.038	1.015	1.007

The asymptotic relationship between k and c is as follows:

Corollary 1. *In symmetric k-trees, the height is bounded by*

$$(1 + \Theta(1/k)) \log_2 n + 1.$$

Proof. This follows from Theorem 1 by the identity $\log_\alpha(x) = \log_2(x)/\log_2(\alpha)$ and the first order approximations

$$\log_2(1+\varepsilon) = 0 + \varepsilon/\ln 2 + O(\varepsilon^2),$$
$$1/(1-\varepsilon) = 1 + \varepsilon + O(\varepsilon^2).$$

□

4 Operations

As mentioned above, a *search* operation proceeds as in any binary search tree, except that unary nodes are just passed through.

An *insert* operation starts by a search which ends in an external node v. If v is empty, the new element is placed there. Otherwise, a new external node containing the new element is made. In that case, if the parent of v is unary, it is made binary, and the new external node becomes a child of the binary node. The key of the new element is inserted as router in the binary node. If the parent of v is binary, a new red binary node is inserted below it, having v and the new external node as children and the key of the new element as router.

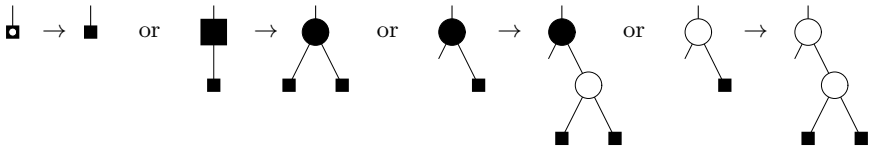


Fig. 1. The insert operation.

A *delete* operation first searches for the external node v, containing the element to be deleted. If the parent of v is unary, the leaf becomes empty. Otherwise,

v is removed, and the binary parent is made unary (discarding the router in it), in case it is black, and is removed completely, in case it is red.

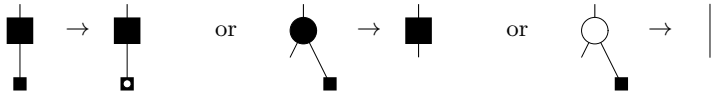


Fig. 2. The delete operation.

Fundamental to the *rebalancing operations* on k-trees is the observation [7] that the position of unary nodes may be moved horizontally in the tree by shifting subtrees. In Fig. 3, the position of the unary node on the left is moved six nodes to the right. Letters denote subtrees.

Fig. 3. The slide operation.

We call this operation a *slide operation*, and we use it to move the positions of unary nodes horizontally among the black nodes of the same black depth.

Note that for a slide involving i neighboring nodes, it is necessary to redistribute some keys to keep the in-ordering of the keys in the tree. The keys in question are contained in the binary nodes among the nodes involved in the slide, as well as in the least common ancestors of each of the consecutive pair of nodes involved in the slide. This is at most $2i - 1$ keys in total. Excluding the time for locating these least common ancestors, the slide can be performed in $O(i)$ time. We address the question of the time for locating these common ancestors later. In [7], this question is not considered at all.

A relaxed k-tree may contain two kinds of structural problems which keep it from being a standard (symmetric) k-tree: *red binary nodes* and *groups containing more than two unary nodes*. Additionally, it may contain *empty external nodes*. We now describe the set of rebalancing operations which we use to remove these three problem types.

We only deal with red binary nodes having a black parent. If the parent of the red node is unary, we use a *contract* operation, which merges the node and the parent into a black binary node.

If the parent is binary, we first check if there is a unary node in its group. If so, we use the slide operation to make the parent unary, and then perform a contract operation.

If the parent is binary, and there is no unary node in its group, we apply the following operation, which makes the parent red and the node itself black.

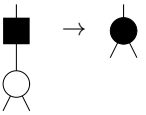


Fig. 4. The contract operation.

Furthermore, if the sibling is red, the operation makes it black. Otherwise, the operation inserts a unary node above the sibling:

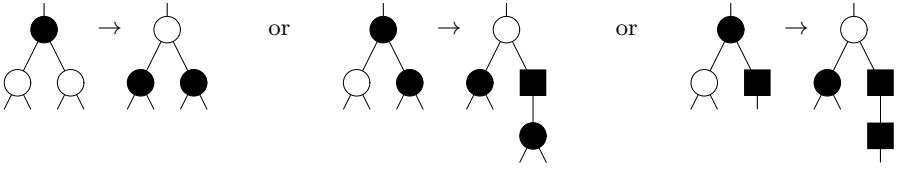


Fig. 5. The split operation.

We call this operation a *split* operation, since it corresponds to the splitting of an i-ary node ($i \geq 3$) in a formulation with multi-way nodes instead of red/black colors.

For a group containing more than two unary nodes, we use the *merge* operation from Fig. 6 which merges two unary siblings into a black binary node. If the parent is black, it is converted to a unary node. If it is red, it is removed.

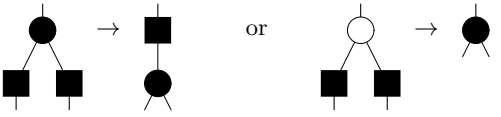


Fig. 6. The merge operation.

Note that by using the slide operation within a group, we can decide freely which nodes within the group should be the unary ones. Note also that since the group contains at least four nodes (as $k \geq 2$), there will be at least two neighboring nodes which have parents belonging to the same group on the level above. Using the slide operation within that group, we can ensure that if it contains any binary node at all, it will be the parent of the two neighboring nodes. Thus, only if this group on the level above does not contain any binary nodes will we not be able to perform a merge operation on a group containing more than two unary nodes.

We note that split and merge operations will make the number of nodes in the affected group increase, respectively decrease, by one. To keep the group

sizes within the required bounds, we use a policy similar to that of B-trees, i.e., a group which reaches a size of $4k + 1$ nodes is split evenly in two, and a group which reaches a size of $2k - 1$ nodes will either borrow a node from a neighboring group, or, if this is not possible because no neighboring group of more than $2k$ nodes exist, will be merged with a neighboring group. This entails simply setting, removing, or moving a group border, i.e., a bit in a node. When borrowing a node from a neighboring group containing fewer than two unary nodes, we ensure that the borrowed node is binary, by first using a slide operation, if necessary. The maintenance of group sizes is performed as part of the split and merge operations.

Regarding empty external nodes, we note that these will always be children of black nodes; they are created that way, and no operation changes this. If the parent of the empty external node is binary, we remove the external node and make the parent unary. If the parent is unary, but a binary node exist in its group, we use the slide operation to make the parent binary, and then proceed as above. Only if the parent's group does not contain any binary nodes will we not be able to remove the empty external node.

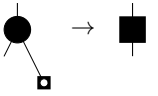


Fig. 7. The removal of an empty external node.

For problems immediately below the buffer level, special *root operations* apply. If the problem is a red node, we use the contract operation, and as usual use a slide to make the parent of the red node unary, if necessary. However, if no unary node exists in the buffer level, this is not possible. In that case, a new buffer level consisting entirely of unary nodes is inserted above the previous buffer level. We then use the split operation to move the red node past the previous buffer level, and then use a contract operation on the new buffer level to remove the red node. Note that this maintains the invariant that the buffer level should contain at least one binary node.

Conversely, if a merge operation removes the last binary node of the buffer level, we first check if any of the unary nodes in the buffer level has a red child. If so, we perform a contract operation on that node. If this is not possible, we remove the nodes in the current buffer level (these are all unary), and let the black level below be the new buffer level. As the merge operation introduced a binary node on this level, the invariant that the buffer level should contain at least one binary node is maintained.

Note that the black height of the tree can only change via a root operation.

It is clear from inspection that the update and rebalancing operations do not violate the invariant that all external nodes have the same black depth. The set of rebalancing operations is also complete in the following sense:

Lemma 1. *On any relaxed k-tree which is not a standard symmetric k-tree without empty external nodes, at least one of the rebalancing operations can be applied on the tree.*

Proof. A rebalancing operation can always be performed at the topmost red node and at the topmost group which contains more than two unary nodes. If no group with more than two unary nodes exists, an empty external node can always be removed. □

5 Complexity Analysis

The number of rebalancing operations per update is amortized constant:

Theorem 2. *During a sequence of i insertions and d deletions performed on an initially empty relaxed k-tree, at most $6i + 4d$ rebalancing operations can be performed before the tree is in balance.*

Proof. The number of removals of empty external nodes clearly cannot exceed d. To bound the rest of the operations, we define a suitable potential function on any relaxed k-tree T. Let the unary potential of a group be $|u - 1|$, where u denotes the number of unary nodes in the group. Denote by $\Phi_1(T)$ the sum of the unary potential of all groups in T (the buffer level does not constitute at group, and neither do the levels above it). Denote by $\Phi_2(T)$ the number of red nodes in T, by $\Phi_3(T)$ the number of groups in T containing $2k$ nodes, and by $\Phi_4(T)$ the number of groups in T containing $4k$ nodes. Define the potential function $\Phi(T)$ by

$$\Phi(T) = 3 \cdot \Phi_1(T) + 6 \cdot \Phi_2(T) + 1 \cdot \Phi_3(T) + 2 \cdot \Phi_4(T).$$

By a lengthy inspection it can be verified that all rebalancing operations, including any necessary group splitting, merging or sharing, will decrease $\Phi(T)$ by at least one. We analyze one case to give the flavor of the argument, and leave the rest to the full paper. Consider the case of the split operation depicted in the middle of Fig. 5. As the group of the top node in the operation does not contain any unary nodes before the operation, the added unary node after the operation reduces $\Phi_1(T)$ by one. The number of red nodes do not change, so neither does $\Phi_3(T)$. The group size increases by one, hence $\Phi_4(T)$ may grow by one, or the group may have to be split (if the size of the group raises to $4k+1$). In the latter case, the sizes of the new groups will be $2k$ and $2k+1$, which makes $\Phi_3(T)$ grow by one while reducing $\Phi_4(T)$ by one, and the new groups will contain zero and one unary node, respectively, which will make $\Phi_1(T)$ grow by one (for a total change of zero). By the weights of $\Phi_1(T), \ldots, \Phi_4(T)$ in $\Phi(T)$, this gives a reduction of $\Phi(T)$ of at least one in all cases.

By inspection, it can also be seen that each insert operation increases $\Phi(T)$ by at most six, and that each delete operation either increases $\Phi(T)$ by at most three, or does not change $\Phi(T)$, but introduces an empty external node, the removal of which later increases $\Phi(T)$ by at most three. As $\Phi(T)$ is zero for the empty tree and is never negative, the result follows. □

The proof above may be refined to give the following:

Theorem 3. *During a sequence of i insertions and d deletions performed on an initially empty relaxed k-tree, at most $O((i+d)(6/7)^h)$ rebalancing operations can be performed at black height h before the tree is in balance.*

Proof. The idea of the proof is to define a potential functions $\Phi^h(T)$ for $h = 0, 1, 2, 3, \ldots$, where $\Phi^h(T)$ is defined as $\Phi(T)$ in the proof above, except that it only counts potential residing at black height h. By inspection, it can be verified that a rebalancing operation at black height h always decreases $\Phi^h(T)$ by some amount $\Delta \geq 1$, and that, while it may increase the value of $\Phi^{h+1}(T)$, it will never do so by more than $6\Delta/7$. As the $\Phi^h(T)$'s are initially zero and are never negative, this implies the statement in the theorem. The details will appear in the full paper. □

Theorem 4. *If n updates are made on a balanced relaxed k-tree containing N keys, then at most $O(n \log(N+n))$ rebalancing operations can be made before the tree is again balanced.*

Proof. The problems in a non-balanced tree consist of red nodes and excess unary nodes in groups. Assigning to each such problem a height equal to the black height of its corresponding node, it can be verified that each rebalancing operation which does not reduce the number of problems will increase the height of some problem by one, and that no rebalancing operation will decrease the height of any problem.

Problems arise during updates at black height zero, and each update introduce at most one problem. Thus, if n updates are performed on an initially balanced tree T, the number of update operations cannot exceed n times the maximum black height of T since the start of the sequence of updates.

To bound this maximum height, we recall that the black height of the tree can only increase during root operations. Specifically, if the black height of the tree reaches some value h, then there has been a root operation at black height $h-1$. It is easily verified that the value of $\Phi(T)$ for a balanced tree T is linear in the number of keys N in the tree. During the n updates, this value may only grow by $O(n)$, by the analysis in the proof of Theorem 2. By an argument similar to that in the proof of Theorem 3, the maximum black height since the start of the sequence of updates is $O(\log_{7/6}(N+n))$, which proves the theorem. The details will appear in the full paper. □

6 Comments on Implementation

In the previous section, we have been concerned with the number of operations which have to be carried out, and we have discussed configurations in the tree at a fairly abstract level. In order to carry out each operation efficiently, it is necessary to be able to find other nodes at the same level, to find least common ancestors, and to locate problems in the tree.

First, we note that by maintaining parent pointers and level links between the black nodes sharing the same black height, all rebalancing operation can be performed in $O(k)$ time, when not counting the time to find the necessary least common ancestors during a slide operation. The same is true for a group resizing operation.

We now discuss how to find the necessary least common ancestor (LCA) of a neighboring pair of black nodes participating in a slide operation.

One approach is *heuristic*: simply search upwards for each of these LCAs. The worst case time for this is poor (the search may take time proportional to the height of the tree for each LCA), but should be good on average in the following sense: If on some level i in a complete binary tree we consider k neighboring nodes, then the LCAs of these k nodes will reside in at most two subtrees with roots at most $\lceil \log(k) \rceil$ levels above i, *except* that one of the LCAs may reside higher (it could be the root of the entire tree). If by δ we denote the difference between $i - \lceil \log(k) \rceil$ and the level of this singular LCA, then it is easily verified that the expected value of δ over all possible start positions of the k neighboring nodes on level i is $O(1)$. As the parts of the two subtrees residing above level i may be traversed in $O(k)$ time, the time for a randomly placed slide involving k nodes is expected $O(k)$ in a binary tree. As a k-tree is structurally close to a binary tree (especially for large k), we therefore believe that the time for finding the LCAs during a slide is not likely to be a problem, unless during the use of a relaxed k-tree we allow the tree to become very unbalanced before rebalancing again.

Another approach is to *maintain explicit links* from every black node to the two LCAs between itself and its two black neighbors, allowing each LCA to be found in constant time during a slide operation. These links then have to be updated for the black nodes on the leftmost and rightmost path on the subtrees exchanged between neighboring black nodes during a slide. Assuming that the black nodes on such a left- or rightmost path can be accessed in constant time per node, this gives a time for a slide involving k neighboring nodes which is proportional to k times the black height at which the slide takes place. However, as $\sum_{h=1}^{\infty} h(6/7)^h = O(1)$, Theorem 3 implies that the amortized rebalancing work is still $O(k)$ per update.

So, we must be able to traverse only the black nodes on the left- and rightmost paths mentioned above. For every black node, we keep all its immediate red descendants (those forming a single node in a formulation with multi-way nodes instead of red/black colors) in a doubly linked list. The list is ordered (the list may be seen as adding in-order links to all red connected components of the tree), and the front and rear of the list is pointed to from the black node rooting the red connected component. Using these front and rear pointers, it is now possible to jump from black to black level during a traversal of right- and leftmost paths, as assumed above. Furthermore, it can be verified that these list can be maintained during update and rebalancing operations, including slides.

Finally, locating problems in the tree is complicated by the main feature of relaxed balance, namely that the rebalancing is uncoupled from the updating. Hence, an update operation simply ignores any problem which arises as a con-

sequence of the update. To be able to return to these problems later, a *problem queue* can be maintained. For problems discovered during an update, a pointer is stored in the queue. One pointer per group suffices. Since update and rebalancing operations may remove other problems, it is convenient to be able to remove problems from the queue. Thus, each group must have a back-pointer into the problem queue. Rebalancing operations start by dequeuing a pointer, which is then followed, and the appropriate rebalancing operation is performed. Rebalancing operations should also insert new pointers when they move problems upwards in the tree, unless the receiving group is already in the queue. Note that when a problem is added to the tree, it can be verified in $O(k)$ time whether the affected group has a problem already.

Acknowledgment. The first and third author were partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). The third author was partially supported by the Danish Natural Science Research Council (SNF).

References

1. G. M. Adel'son-Vel'skiĭ and E. M. Landis. An Algorithm for the Organisation of Information. *Doklady Akadamii Nauk SSSR*, 146:263–266, 1962. In Russian. English translation in *Soviet Math. Doklady*, 3:1259-1263, 1962.
2. Joan Boyar, Rolf Fagerberg, and Kim S. Larsen. Amortization Results for Chromatic Search Trees, with an Application to Priority Queues. *Journal of Computer and System Sciences*, 55(3):504–521, 1997.
3. Joan F. Boyar and Kim S. Larsen. Efficient Rebalancing of Chromatic Search Trees. *Journal of Computer and System Sciences*, 49(3):667–682, 1994.
4. Leo J. Guibas and Robert Sedgewick. A Dichromatic Framework for Balanced Trees. In *Proceedings of the 19th Annual IEEE Symposium on the Foundations of Computer Science*, pages 8–21, 1978.
5. Kim S. Larsen. Amortized Constant Relaxed Rebalancing using Standard Rotations. *Acta Informatica*, 35(10):859–874, 1998.
6. Kim S. Larsen. AVL Trees with Relaxed Balance. *Journal of Computer and System Sciences*, 61(3):508–522, 2000.
7. H. A. Maurer, Th. Ottmann, and H.-W. Six. Implementing Dictionaries using Binary Trees of Very Small Height. *Information Processing Letters*, 5(1):11–14, 1976.
8. Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 192–198, 1991.
9. Otto Nurmi and Eljas Soisalon-Soininen. Chromatic Binary Search Trees—A Structure for Concurrent Rebalancing. *Acta Informatica*, 33(6):547–557, 1996.
10. Otto Nurmi, Eljas Soisalon-Soininen, and Derick Wood. Relaxed AVL Trees, Main-Memory Databases and Concurrency. *International Journal of Computer Mathematics*, 62:23–44, 1996.
11. Neil Sarnak and Robert E. Tarjan. Planar Point Location Using Persistent Search Trees. *Communications of the ACM*, 29:669–679, 1986.

Succinct Dynamic Data Structures*

Rajeev Raman[1], Venkatesh Raman[2], and S. Srinivasa Rao[2]

[1] Department of Mathematics and Computer Science
University of Leicester, Leicester LE1 7RH, UK.
r.raman@mcs.le.ac.uk.
[2] Institute of Mathematical Sciences, Chennai, India 600 113,
{vraman,ssrao}@imsc.ernet.in

Abstract. We develop succinct data structures to represent (i) a sequence of values to support *partial sum* and *select* queries and *update* (changing values) and (ii) a dynamic array consisting of a sequence of elements which supports *insertion*, *deletion* and *access* of an element at any given index.

For the partial sums problem on n non-negative integers of k bits each, we support *update* operations in $O(b)$ time and *sum* in $O(\log_b n)$ time, for any parameter b, $\lg n/\lg\lg n \leq b \leq n^\epsilon$ for any fixed positive $\epsilon < 1$. The space used is $kn+o(kn)$ bits and the time bounds are optimal. When $b = \lg n/\lg\lg n$ or $k = 1$ (i.e., when we are dealing with a bit-vector), we can also support the *select* operation in the same time as the *sum* operation, but the update time becomes amortised.

For the dynamic array problem, we give two structures both using $o(n)$ bits of extra space where n is the number of elements in the array: one supports lookup in constant worst case time and updates in $O(n^\epsilon)$ worst case time, and the other supports all operations in $O(\lg n/\lg\lg n)$ amortized time. The time bound of both these structures are optimal.

1 Introduction

Recently there has been a surge of interest in the study of *succinct* data structures [1,2,3,9,10,11,12,13]. The aim is to design data structures that are asymptotically optimal with respect to operation times, but whose space usage is optimal to within lower-order additive terms. Barring a few exceptions [2,13], most of these are static structures. In this paper we look at succinct solutions to two classical interrelated dynamic data structuring problems, namely maintaining *partial sums* and *dynamic arrays*. We assume a RAM model with word size $\Theta(\lg n)$ bits, where n is the input size. In this model, reading and writing $O(\lg n)$ consecutively stored bits, arithmetic and bit-wise boolean operations on $O(\lg n)$-bit operands can be performed in constant time. In more detail, the problems considered are:

* Research supported in part by UK-India Science and Technology Research Fund project number 2001.04/IT and in part by UK EPSRC grant GR L/92150.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 426–437, 2001.

Partial Sums. This problem has two positive integer parameters, the *item size* $k = O(\lg n)$, and the *maximum increment* $\delta_{max} = \lg^{O(1)} n$. The problem consists in maintaining a sequence of n numbers $A[1], \ldots, A[n]$, such that $0 \leq A[i] \leq 2^k - 1$ under the operations:

- *sum*(i): return the value $\sum_{j=1}^{i} A[j]$.
- *update*(i, δ): set $A[i] \leftarrow A[i] + \delta$, for some integer δ such that $0 \leq A[i] + \delta \leq 2^k - 1$ and $|\delta| \leq \delta_{max}$.

We also consider adding the following operation:

- *select*(j): find the smallest i such that *sum*$(i) \geq j$.

In what follows, we refer to the partial sums problem with *select* as the *searchable* partial sums problem.

Dietz [4] has given a structure for the partial sums problem that supports *sum* and *update* in $O(\lg n / \lg \lg n)$ worst-case time using $\Theta(n \lg n)$ bits of extra space, for the case $k = \Theta(\lg n)$. As the information-theoretic space lower bound is kn bits, Dietz's data structure uses a constant factor extra space even when $k = \Theta(\lg n)$, and is worse for smaller k. We modify Dietz's structure to obtain a data structure that solves the searchable partial sums problem in $O(\lg n / \lg \lg n)$ worst case time using $kn + o(kn)$ bits of space. Thus, we improve the space utilisation and add the *select* operation as well.

For the partial sums problem we can trade off query and update times as follows: for any parameter $b \geq \lg n / \lg \lg n$ we can support *sum* in $O(\lg_b n)$ time and *update* in $O(b)$ time[1]. The space used is the minimum possible to within a lower-order term.

Our time bounds are optimal in the following sense. Fredman and Saks [5] gave lower bounds for this problem in the *cell probe* model with logarithmic word size, a much stronger model than ours. For the partial sums problem, they show that an intermixed sequence of n updates and queries requires $\Omega(\lg n / \lg \lg n)$ amortized time per operation. Furthermore, they give a more general trade-off [5, Proof of Thm 3′] between the number of memory locations that must be written and read by an intermixed sequence of updates and queries. Our data structure achieves the optimal trade-off between reads and writes, for the above range of parameter values. If we require that queries be performed using read-only access to the data structure—a requirement satisfied by our query algorithms—then the query and update times we achieve are also optimal.

Next, we consider a special case of the searchable partial sums problem that is of particular interest.

Dynamic Bit Vector. Given a bit vector of length n, support the following operations:

[1] In the partial sum and bit-vector results, other trade-offs that allow expensive queries and cheap updates are possible. These are mentioned in the appropriate sections.

– $rank(i)$: find the number of 1's occurring before and including the ith bit
– $select(j)$: find the position of jth one in the bit vector and
– $flip(i)$: flip the bit at position i in the bit vector.

A bit vector supporting *rank* and *select* is a fundamental building block for succinct static tree and set representations [2,11]. Given a (static) bit vector, we can support the *rank* and *select* operations in $O(1)$ time using $o(n)$ bits of extra space [3,10].

As the dynamic bit vector problem is simply the searchable partial sums problem with $k = 1$, we immediately obtain a data structure that supports *rank*, *select* and *flip* operations in $O(\lg n/\lg\lg n)$ worst case time using $o(n)$ bits of extra space. For the bit vector, however, we are able to give a trade-off for all three operations. Namely, for any parameter $b \geq \lg n/\lg\lg n$ we can support *rank* and *select* in $O(\lg_b n)$ time and *update* in amortised $O(b)$ time. In particular, we can support *rank* and *select* in constant time if we allow updates to take $O(n^\epsilon)$ amortised time for any constant $\epsilon > 0$.

If we remove the *select* operation from the dynamic bit vector problem, we obtain the *subset rank* problem considered by Fredman and Saks [5]. From their lower bound on the subset rank problem, we conclude that our time bounds are optimal, in the sense described above.

Next we consider another fundamental problem addressed by Fredman and Saks [5].

Dynamic Array. Given an initially empty sequence of records, support the following operations:

– $insert(x, i)$: insert a new record x at position i in the sequence
– $delete(i)$: delete the record at position i in the sequence and
– $index(i)$: return the ith record in the sequence.

Dynamic arrays are useful data structures in efficiently implementing the data types such as the Vector class in Java and C++. The dynamic array problem was called the *List Representation* problem by Fredman and Saks, who gave a cell probe lower bound of $\Omega(\lg n/\lg\lg n)$ time for this problem, and also showed that $n^{\Omega(1)}$ update time is needed to support constant-time queries. For this problem, Goodrich and Kloss [8] obtained a structure that supports *insert* and *delete* operations in $O(n^\epsilon)$ amortized time while supporting the *index* operation in $O(1/\epsilon)$ worst case time. This structure uses $O(n^{1-\epsilon})$ words of extra space, (besides the space required to store the n elements of the array) for any fixed ϵ, $0 < \epsilon < 1$. Here n is the size of the current sequence.

We first observe that the structure of Goodrich and Kloss can be viewed as a version of the well-known implicit data structure: the 'rotated list' [7]. Using this connection, we observe that the structure of Goodrich and Kloss can be made to take $O(n^\epsilon)$ worst case time for updates while maintaining the same storage ($o(n)$ additional words) and $O(1)$ worst case time for the *index* operation. Then using this structure in small blocks, we obtain a dynamic array structure that

supports *insert*, *delete* and *index* operations in $O(\lg n/\lg\lg n)$ amortized time using $o(n)$ bits of extra space. Due to the lower bound result of Fredman and Saks, both our results above are on optimal points of the query time-update time trade-off while using optimal (within lower order term) amount of extra space.

We should also point out that the resizable arrays of Brodnik et al. [1] can be used to support inserting and deleting elements at either ends of an array and accessing the ith element in the array, all in constant time. The data structure uses $O(\sqrt{n})$ words of extra space if n is the current size of the array. Brodnik et al. do not support insertion into the middle of the array.

For the dynamic array problem, we assume a memory model in which the system returns a pointer to the beginning of a block of requested size. I.e. any element in a block of memory can be accessed in constant time given the block pointer and an integer index into the block. This is the same model used by Brodnik et al. [1]. We count the time as the number of word operations, and space as the number of bits used to store the data structure. To simplify notation, we ignore rounding as it does not affect our asymptotic analysis.

In Section 2, we describe our space efficient structures for the partial sum problem. In Section 3, we look at the special case of the partial sum problem when the given elements are bits, and give the details of a structure that supports full tradeoff between queries (*select* and *rank*) and update (*flip*). Section 4 addresses the problem of supporting the dynamic array operations. We conclude with some open problems in Section 5.

2 Partial Sums

2.1 Searchable Partial Sums

In this section we describe a structure for the searchable partial sums problem that supports all operations in $O(\lg n/\lg\lg n)$ time. The space used is $kn+o(kn)$ bits. We begin by solving the problem on a small set of integers in $O(1)$ time, by adapting an idea of Dietz [4].

Lemma 1. *On a RAM with a word size of w bits, we can solve the searchable partial sum problem on a sequence of $m = w^\epsilon$ numbers, for any fixed $0 \leq \epsilon < 1$, with item size $k \leq w$, in $O(1)$ worst-case time and using $O(mw)$ bits of space. The data structure requires a precomputed table of size $O(2^{\epsilon' w})$ for any fixed $\epsilon' > 0$.*

Proof. Let $A[1], \ldots, A[m]$ denote the sequence of elements for which the partial sums are to be calculated. We store another array $B[1], \ldots, B[m]$ which contains the partial sums of A, i.e. $B[i] = \sum_{j=1}^{i} A[i]$. As we cannot hope to maintain B under the *update* operation, we use Dietz's idea of letting B get slightly 'out of date'. More precisely, B is not changed after each *update*; instead, after every m *updates* B will be *refreshed*, or brought up to date. Since the cost of refreshing is $O(m)$, the amortized cost is $O(1)$ per *update*.

To answer queries, we maintain an array $C[1], \ldots, C[m]$ in addition to A and B. C is set to all zeros when B is refreshed. Otherwise, when an *update* changes $A[i]$ by δ, we set $C[i] \leftarrow C[i]+\delta$. Since $|C[i]| \leq m\delta_{max} = w^{O(1)}$ always, the entire array C occupies $O(m \lg w)$ bits, which is less than $\epsilon' w$ bits for sufficiently large w. As observed by Dietz, $sum(i)$ can be computed by adding to $B[i]$ a corrective term obtained by using C to index into a pre-computed table.

We now show how to perform *select* in $O(1)$ time. For this, we use the *Q-heap* structure given by Fredman and Willard [6], which solves the following dynamic predecessor problem:

Theorem 1. *[6] For any $0 < M < 2^w$, given a set of at most $(\lg M)^{1/4}$ integers of $O(w)$ bits each, one can support the operations insert, delete, predecessor and successor operations in constant time where predecessor(x) (successor(x)) returns the largest (smallest) element y in the set such that $y < x$ ($y \geq x$). The data structure requires a precomputed table of size $O(M)$.*

By choosing $M = 2^{\epsilon' w}$, we can do predecessor queries on sets of size $m' = w^{\epsilon'/4}$ in $O(1)$ time, using a table of size $O(M)$. By using this data structure in a tree with branching factor m', we can support $O(1)$-time operations on sets of size m as well. We store the elements of B in the Q-heap data structure. Note that changes to the Q-heap caused by refreshing have $O(1)$ amortised cost.

If the array B were up-to-date, then we can answer *select* queries in $O(1)$ time by finding the successor of j in the Q-heap. However, again we face the problem that B may not be up-to-date. To overcome this, we let $D[1], \ldots, D[m]$ be an array where $D[i] = \min\{A[i], m\delta_{max}\}$. As with C, D is also stored in a single word, and is changed in $O(1)$ time (using either table lookup or bitwise operations) whenever A is changed by an *update*. To implement $select(j)$, we first consult the Q-heap data structure to determine an index t such that $B[t-1] < j \leq B[t]$. By calculating $sum(t-1)$ and $sum(t)$ in $O(1)$ time we determine whether t is the correct answer. In general, the correct answer would be an index $t' \neq t$; assume for specificity that $t' > t$. Note that t' is the smallest integer such that $A[t+1]+\cdots+A[t'] \geq j-sum(t)$. Since $j \leq B[t]$ and $B[t]-sum(t) \leq m\delta_{max}$, it follows that $j - sum(t) \leq m\delta_{max}$. By the definition of D, it also follows that t' is the smallest integer such that $D[t+1] + \ldots + D[t'] \geq j - sum(t)$. Given D, $j - sum(t)$ and t, one can look up a table to calculate t' in $O(1)$ time. A similar procedure is followed if $t' < t$.

Finally, we note that amortization can be eliminated by Dietz's incremental refreshing approach [4]. □

For larger inputs, we choose a parameter $m = (\lg n)^\epsilon$, for some positive constant $\epsilon < 1$. We create a complete m-ary tree, the leaves of which correspond to the entries of the input sequence A. We define the *weight* of a node (leaf or internal) as the sum of the sequence elements under it. At each internal node we store the weights of its children in the data structure of Lemma 1. The tree has $O(n/m)$ internal nodes, each occupying $O(m)$ words and supporting all operations in constant time. From this we get:

Lemma 2. *There is a data structure for the searchable partial sums problem that supports all operations in $O(\lg n/\lg\lg n)$, and requires $O(n)$ words of space.*

We now modify this structure reducing the space complexity of the structure to $kn + o(kn)$ bits. For this, we need the following:

Lemma 3. *For any parameter $b \geq 4$, there is a data structure for the searchable partial sums problem that supports update in $O(\log_b n)$ time and sum and search in $O(b \log_b n)$ time. The space used is $kn + O(((k + \lg b) \cdot n)/b)$ bits.*

Proof. We construct a complete b-ary tree over the elements of the input sequence A. At each internal node we store the sum of the elements of A under it. Clearly *update* takes time proportional to the height of the tree, and *sum* and *select* can be implemented within the claimed bounds by traversing the tree from the root to a leaf, looking at all the children of a node at each level. The space bounds follow after a straightforward calculation. □

We take the input and divide it into groups of numbers of size $(\lg n)^2$ each. The groups are represented internally using Lemma 3, with $b = (\lg n)^{1/2}$. This requires $kn+o(kn)$ bits, and all operations within a group take $O((\lg n)^{1/2})$ time, which is negligible. The $n/(\lg n)^2$ group sums are stored in the data structure of Lemma 2, which requires $o(n)$ bits now. The precomputed tables (required in Lemma 1) also require $o(n)$ bits. Thus we have:

Theorem 2. *There is a data structure for the searchable partial sums problem that supports all operations in $O(\lg n/\lg\lg n)$ worst-case time and uses $kn+o(kn)$ bits of space.*

2.2 Trade-Offs for Partial Sums

We now observe that one can trade off query and update times for the partial sums problem, and show that for any parameter $2 \leq b \leq n$, we can support *sum* in $O(\log_b n)$ and *update* in $O(b \log_b n)$ time, while still ensuring that the data structure is space-efficient. As these bounds are subsumed by Theorem 2 for $b \leq (\lg n)^2$, we will assume that $b > (\lg n)^2$ in what follows.

We construct a complete tree with branching factor b, with the given sequence of n elements at the leaves. Clearly this tree has height $h = \log_b n$. At each internal node, we store the *weight* of that node, i.e. the sum of the leaves descended from it, and also store an array containing the partial sums of the weights of all its children. By using the obvious $O(b)$ time algorithm, the partial sum array at an internal node is kept up-to-date after each *update*. This gives running times of $O(b \lg_b n)$ and $O(\lg_b n)$ for *update* and *sum* respectively. Unfortunately, the space used to store this 'simple' structure is $O(kn)$ bits.

To get around this, we use one of two methods, depending on the value of k. If $k \geq (\lg n)^{1/2}$ then we divide the input values into groups of size $\lg n$. Within a group, we do not store the $A[i]$'s explicitly, but store only their partial sums. The sums of elements in each of the $n/\lg n$ groups are stored in the

simple structure above, but the space required by that data structure is now $O((k + \lg\lg n)n/\lg n)$ (as the size of each sum could be $k + \lg\lg n$) which is $o(kn)$ bits. The space required by each group is $\lg n(k + \lg\lg n)$ bits; this sums up to $kn + n\lg\lg n = kn + o(kn)$ bits overall. Clearly the asymptotic complexity of *update* and *sum* are not affected by this change.

If $k < (\lg n)^{1/2}$ then we divide the given sequence of elements into groups of $\epsilon \lg n/k$ each. Again, group sums are stored in the simple structure, which requires $O(kn(k+\lg\lg n)/\lg n) = o(kn)$ bits. Noting that an entire group requires $\epsilon \lg n$ bits, we answer *sum* queries within a group by table lookup.

Finally, noting that given any parameter $b \geq (\lg n)^2$, we can reduce the branching factor from b to $b/\lg n$ without affecting the complexity of *sum*; however, *update* would now take $O(b)$ steps. Combining this with Theorem 2 we have:

Theorem 3. *For any parameter* $\lg n/\lg\lg n \leq b < n$*, there is a data structure for the partial sums problem that supports sum in* $O(\lg_b n)$ *time and update in* $O(b)$ *time, and uses* $kn + o(kn)$ *bits of space.*

Remark 1. Note that Lemma 3 combined with Theorem 2 also gives a trade-off whereby *update* takes $O(\log_b n)$ and *sum* takes $O(b)$ time, for any $b \geq \lg n/\lg\lg n$.

3 Dynamic Bit Vector

The dynamic bit vector problem operation is a special case of the searchable partial sum problem. The following corollary follows from Theorem 2.

Corollary 1. *Given a bit vector of length* n*, we can support the rank, select and flip operations in* $O(\lg n/\lg\lg n)$ *time using* $o(n)$ *bits of space in addition to the bit vector.*

Similarly, Theorem 3 immediately implies the following result (the only thing to observe is that of the two cases in Theorem 3, we apply the one that stores the input sequence explicitly):

Corollary 2. *For any parameter* $\lg n/\lg\lg n \leq b < n$*, there is a data structure for the dynamic bit vector problem that supports rank in* $O(\lg_b n)$ *time and flip in* $O(b)$ *time, using* $o(n)$ *bits of space in addition to the bit vector.*

3.1 Trade-Off between *query* and *update* Times

In this section we show that the trade-off's between *sum* and *update* for partial sums established in Section 2.2, also hold between *select* and *update* for the special case of the dynamic bit vector problem. We first note the following proposition.

Lemma 4. *The operations select and flip can be supported in $O(1)$ time on a bit-vector of size $N = (\lg n)^{O(1)}$ on a RAM with word size $O(\lg n)$, using a fixed pre-computed table of size $O(n^\epsilon)$ bits for some constant $\epsilon < 1$. The space required is $o(N)$ bits in addition to the pre-computed table and the bit-vector itself.*

Proof. Simply store the values in a balanced tree with branching factor $\sqrt{\lg n}$, and stop the tree at the level when the number of leaves at the subtree rooted at the nodes is about $(\lg n)/2$. With each internal node, we keep the searchable structure of Lemma 1. At the leaf level, we will use a precomputed table to support flip and select in constant time.

Since the height of the tree is a constant, select and flip can be supported in constant time. The space used is $o(N)$ besides the $O(n^\epsilon)$ bits required for the precomputed table. □

We now show how to support *select* in $O(\lg_b n)$ time if *flip* takes $O(b)$ time, for any parameter $(\lg n)^4 \leq b \leq n$. We divide the bit vector into *superblocks* of size $(\lg n)^4$. With each superblock we store the number of ones in it. The sequence of superblock counts is stored in the data structure of Theorem 3 with the same value of b. This enables us, $O(\lg_b n)$ time, to look up the number of ones to the left of any given superblock. We store each of the superblocks using the structure of Lemma 4. The space required is $o(n)$ bits.

In addition, we divide the ones in the bit vector into groups of $\Theta((\lg n)^2)$ successive ones each. A group's size varies between $0.5(\lg n)^2$ and $2(\lg n)^2$. We construct a weight-balanced B-tree (WBB tree) in the sense of Dietz [4], each leaf of which corresponds to a group leader. Roughly speaking, the branching of this tree is b. Some small modifications need to be made to Dietz's balance conditions: for example, the weight of an internal node needs to be redefined to be the sum of the sizes of the groups under it (we omit details of the WBB tree in this abstract). Given an integer j, using Dietz's ideas we can locate the group in which the j-th one lies in $O(\lg_b n)$ time, and support changes due to *flip*s in $O(b)$ amortised time.

With each group we store the index of the superblock in which the group's leader lies. Equally, with each superblock, we store all group leaders which lie in that superblock.

The *span* of a group is the index of the superblock in which the next group's leader lies minus the index of the superblock in which its group leader lies. If the span of a group is ≥ 2 we say it is *sparse*. With each sparse group, we store an array which gives the location of each 1 in the group. Since the maximum size of this array is $O((\lg n)^2)$, using either the implementation of Goodrich and Kloss or the implementation of Theorem 5, we get a dynamic array which allows insertions and deletions in $O(\lg n)$ time and accesses in $O(1)$ time. This requires $O((\lg n)^3)$ bits per sparse group, but there can only be $O(n/(\lg n)^4)$ sparse groups, so the total space used here is $O(n/\lg n)$.

To execute *select*(j) we first locate the group in which the j-th one lies in $O(\lg_b n)$ time, spending $O(1)$ time at each level of the tree. If this group is sparse then we look up the array associated with the group in $O(1)$ time. Otherwise,

we look in the superblock in which the group leader lies, as well as the adjacent superblock, where we can answer the query in $O(1)$ time each by Lemma 4.

To set bit j to 1, we first locate the group to which position j would belong. This is easy given the space bounds: recall that via *rank* and *select* one can move from the $i+1$st one bit to the ith one bit in $O(1)$ time. As we move along this "virtual linked list" of 1 bits, we check to see if we have reached a group leader (by looking to see if it is listed among the group leaders in the current superblock). Having thus located the group leader in poly-log (negligible) time, we then either insert into the appropriate array (if the group is sparse) and also into the data structure associated with the superblock. Group splits and merges are handled straightforwardly. Again for $b \geq \lg^4 n$, we can actually make the branching factor to be $b/\lg n$. For other values of b, using Corollaries 1 and 2, we have:

Theorem 4. *Given a bit vector of length n, we can support the rank and select operations in $O(\lg_b n)$ time and flip in $O(b)$ amortised time for any parameter b, $b \geq \lg n/\lg\lg n$ using $o(n)$ bits of extra space.*

Remark 2. Note that one can also get a trade-off similar to that of Remark 1 whereby *flip* takes $O(\log_b n)$ and *rank*/*select* takes $O(b)$ time, for any $b \geq \lg n/\lg\lg n$, using $o(n)$ bits of extra space.

4 Dynamic Arrays

We look at the problem of maintaining an array structure under the operations of insertion, deletion and indexing. Goodrich and Kloss [8] have given a structure that supports (arbitrary) *insert* and *delete* operations in $O(n^\epsilon)$ amortized time and *index* operation in $O(1)$ worst case time using $o(n)$ bits of extra space to store a sequence of n elements. Here, first we describe a structure that essentially achieves the same bounds above (except that we can now support updates in $O(n^\epsilon)$ worst case time) using a well known implicit data structure called *recursively rotated list* [7]. Using this as a basic block, we will give a structure that supports all the dynamic array operations in $O(\lg n/\lg\lg n)$ amortized time using $o(n)$ bits of extra space.

We assume a memory model in which the system returns a pointer to the beginning of a block of requested size and hence any element in a block of memory can be accessed in constant time given its index within the block and the block pointer. This is the same model used in the resizable array of Brodnik et al. [1].

Rotated lists were discovered to support dictionary operations implicitly, on a totally ordered set. A (1-level) rotated list is an arbitrary cyclic shift of the sorted order of the given list. We can search for an element in a rotated list on n elements in $O(\lg n)$ time by a modified binary search, though updates (replacing one value with another) can take $O(n)$ time. However, replacing the largest (smallest) element with an element smaller (larger) than the smallest (largest) can be done in $O(1)$ time if we know the position of the smallest element in the

list. A 2-level rotated list consists of elements stored in an array divided into blocks where the i-th block is a rotated list of i elements. It is easy to see that such a structure containing n elements has $r = O(\sqrt{n})$ blocks. In addition, all the elements of block i are less than every element of block $i+1$, for $1 \leq i < r$.

This structure supports searches in $O(\lg n)$ time and updates in $O(\sqrt{n})$ time, if we also explicitly store the position of the smallest element in each block (otherwise the updates take $O(\sqrt{n} \lg n)$ time). This is easily generalized to an l-level rotated list where searches take $O(2^l \lg n)$ time and updates take $O(2^l n^{1/l})$ time. See [7] for details.

To use this structure to implement a dynamic array, we do the following. We simply store the elements of the array in a rotated list based on their order of insertions. We also keep the position of the first element in each recursive block. Since we know the size of each block, $index(i)$ operation just takes $O(l)$ time in an l-level rotated list implementation of a dynamic array. Similarly inserting/deleting at position i can be done in a similar fashion as in a rotated list taking $O(2^l n^{1/l})$ time. Thus we have,

Theorem 5. *A dynamic array having n elements can be implemented using an l-level rotated list such that queries can be supported in $O(l)$ time and updates in $O(2^l n^{1/l})$ time using an extra space of $O(n^{1-1/l})$ pointers.*

Choosing l to be a small constant, we get

Corollary 3. *A dynamic array containing n elements can be implemented to support queries in constant time, and updates in $O(n^\epsilon)$ time using $O(n^{1-\epsilon})$ pointers, where ϵ is any fixed positive constant.*

Using this structure, we now describe a structure that supports all the dynamic array operations in $O(\lg n / \lg \lg n)$ amortized time using $o(n)$ bits of extra space.

We divide the given list of length n into sub lists of length $\Theta(\lg^4 n)$. In particular, each sub-list will be of length between $\frac{1}{2} \lg^4 n$ and $2 \lg^4 n$. (We implement these leaves using the dynamic array structure of Theorem 5.) We construct a weight-balanced B-tree (WBB tree) in the sense of Dietz [4], each leaf of which corresponds to a sub-list. Some small modifications need to be made to Dietz's balance conditions: for example, the weight of an internal node needs to be redefined to be the sum of the sizes of the sub-lists under it. The space required to store this tree is $o(n)$ bits. Supporting insert and delete in $O(\lg n / \lg \lg n)$ amortized time is done as in [4]. To find the jth element of the list, we first find the block in which the jth element occurs, using the $select(j)$ operation on the WBB tree and then find the required element in that block. Thus we have:

Theorem 6. *A dynamic array can be implemented using $o(n)$ bits of extra space besides the space used to store the n records, in which all the operations can be supported in $O(\lg n / \lg \lg n)$ amortized time, where n is the current size of the array.*

5 Conclusions

We have given a succinct searchable partial sum data structure where *sum*, *select* and *update* can be supported in optimal $O(\lg n/\lg\lg n)$ time. We have also given structures in which *sum* can be supported in $(\lg_b n)$ time and *update* in $O(b)$ time for any $b \geq \lg n/\lg\lg n$. These tradeoffs also hold between *select*/*rank* and *update* (flip) for the dynamic bit vector problem. These structures use at most $o(n)$ extra words than necessary.

For the dynamic array, we have given two structures, both using $o(n)$ bits of extra space where n is the number of elements in the array: one supports lookup in constant worst case time and updates in $O(n^\epsilon)$ worst-case time, and the other supports all operations in $O(\lg n/\lg\lg n)$ amortized time.

The following problems remain open:

1. In the searchable partial sums problem, we were able to support select in $O(\lg_b n)$ time and update in $O(b)$ time using $o(kn)$ bits for the special case of $k = 1$. When is this trade-off achievable in general?
2. For the dynamic array problem, are there tradeoffs (both upper and lower bounds) similar to those in the partial sum problem between query and update operations? In particular is there a structure where updates can be made in $O(1)$ time and access in $O(n^\epsilon)$ time?
3. Another related problem looked at by Dietz, and Fredman and Saks is the List indexing problem which is like the dynamic array problem, but adds the operation *position*(x), which gives the position of item x in the sequence, and also modifies *insert* to insert a new element after an existing one. Dietz has given a structure for this problem that takes $O(n)$ extra words and supports all the operations in the optimal $O(\lg n/\lg\lg n)$ time. It is not clear that one can reduce the space requirement to just $o(n)$ extra words, and still support the operations in optimal time.

References

1. A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro and R. Sedgewick, "Resizable Arrays in Optimal Time and Space", *Proceedings of Workshop on Algorithms and Data Structures*, **LNCS 1663**, 37-48 (1999).
2. A. Brodnik and J. I. Munro, "Membership in Constant Time and Almost Minimum Space", *SIAM Journal on Computing*, **28(5)**, 1628-1640 (1999).
3. D. R. Clark, "Compact Pat Trees", Ph.D. Thesis, University of Waterloo, 1996.
4. Paul F. Dietz, "Optimal Algorithms for List Indexing and Subset Rank", *Proceedings of Workshop on Algorithms and Data Structures*, **LNCS 382**, 39-46 (1989).
5. M. L. Fredman and M. Saks, "The Cell Probe Complexity of Dynamic Data Structures", *Proceedings of the* 21^{st} *ACM Symposium on Theory of Computing*, 345-354 (1989).
6. M. L. Fredman and D. E. Willard, "Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths", *Journal of Computer Systems Science*, **48**, 533-551 (1994).

7. G. N. Fredrickson, "Implicit Data Structures for the Dictionary Problem", *Journal of Association of the Computing Machinery*, **30**, 80-94 (1983).
8. M. T. Goodrich and J. G. Kloss II, "Tiered Vectors: Efficient Dynamic Array for JDSL", *Proceedings of Workshop on Algorithms and Data Structures*, **LNCS 1663**, 205-216 (1999).
9. R. Grossi and J. S. Vitter, "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching", *Proceedings of Symposium on Theory of Computing*, 397-406 (2000).
10. G. Jacobson, "Space Efficient Static Trees and Graphs", *Proceedings of the IEEE Symposium on Foundations of Computer Science*, 549-554 (1989).
11. J. I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs", *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1997) 118-126.
12. J. I. Munro, V. Raman and S. S. Rao, "Space Efficient Suffix trees", *Proceedings of the conference on Foundations of Software Technology and Theoretical Computer Science*, **LNCS 1530**, 186-196 (1998).
13. J. I. Munro, V. Raman and A. Storm, "Representing Dynamic Binary Trees Succinctly", *Proceedings of Symposium on Discrete Algorithms*, 529-536 (2001).

Optimal Algorithms for Two-Guard Walkability of Simple Polygons

Binay Bhattacharya[1], Asish Mukhopadhyay[2], and Giri Narasimhan[3]

[1] School of Comput. Science, Simon Fraser Univ., Burnaby, B.C., Canada, V5A 1S6.
binay@cs.sfu.ca
[2] Department of Computer Science, University of Windsor, ON, Canada N9B 3P4.
asishm@cs.uwindsor.ca
[3] Math. Sciences Deptartment, University of Memphis, Memphis TN 38152, USA.
Giri.Narasimhan@memphis.edu

Abstract. A polygon P admits a *walk* from a boundary point s to another boundary point t if two guards can simultaneously walk along the two boundary chains of P from s to t such that they are always visible to each other. A walk is called a *straight* walk if no backtracking is required during the walk. A straight walk is *discrete* if only one guard is allowed to move at a time, while the other guard waits at a vertex. We present simple, optimal $O(n)$ time algorithms to determine all pairs of points of P which admit walks, straight walks and discrete straight walks. The chief merits of the algorithms are that these require simple data structures and do not assume a triangulation of P. Furthermore, the previous algorithms for the straight walk and the discrete straight walk versions ran in $O(n \log n)$ time even after assuming a triangulation.

1 Introduction

As a topological entity a simple polygon is merely a closed non-intersecting curve; as a metric entity its boundary structure can be extremely complex. Various computational problems related to simple polygons are the result of efforts to classify polygons with respect to their boundary structures. Among these are visibilty problems in their different incarnations, art-gallery or guard problems [10], which are fundamental to the understanding of polygon visibility. Many of these problems can be solved efficiently, if one assumes that a triangulation of the polygon is available. However, any algorithm that requires the notoriously complex linear-time triangulation algorithm of Chazelle [3] is simply impractical. In this paper, we show that it is possible to find efficient algorithms that avoid this requirement. In particular, we show how to solve various versions of the 2-guard walkability problem of Icking and Klein [8] in optimal linear time.

The 2-guard problem involves determining whether two guards can move along the boundary of a simple polygonal room from a common starting point s to a common destination point t, in opposite directions, while remaining in sight of each other at all times. If they can, then P is said to be 2-guard walkable (or, simply, *walkable*) with respect to (s, t). A polygon is said to be walkable if

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 438–449, 2001.

such a walk exists for some pair (s,t). Icking and Klein [8] introduced this class of problems, and showed that it is possible to test in $O(n \log n)$ time whether P is walkable with respect to a pair of points (s,t). Subsequently, a linear time solution was presented by Heffernan [7], assuming that a triangulation of the polygon is available. If a triangulation is not assumed the bound is $O(n \log n)$.

A stronger notion than walkability is that of straight walkability. A walkable polygon is said to be *straight walkable* if the guards are able to complete the walk, again under the condition of constant mutual visibility, without having to *backtrack* during their walk to ensure this. Tseng et al.[11] presented an $O(n \log n)$ algorithm to determine whether P is straight walkable, and within the same time bound showed how to generate all such pairs. An even stronger notion is that of discrete straight walkability [1]. A straight walkable polygon is said to be *discretely straight walkable* if only one of the guards is allowed to walk at a time, while the other remains stationary at a vertex. Testing discrete straight walkability and computing all discrete straight walkable pairs of points can be performed in $O(n \log n)$ time [9],

The **main results** of this paper are surprisingly simple, optimal linear-time algorithms that determine all pairs of points of P with respect to which the polygon P is: (i) walkable, (ii) straight walkable, and (iii) discretely straight walkable. While we improve the best known results for (discrete) straight walkability, the surprising element is that all our algorithms use only simple sweeps and elementary data structures.

2 Preliminaries

Let P be a simple polygon with n vertices. The open (closed) clockwise chain of P from u to v is denoted by $P(u,v)$ ($P[u,v]$). Half-open chains will be denoted by $P[u,v)$ or $P(u,v]$, depending on which end is open. For points p and q that belong to a boundary chain of P with distinct end-points, if p precedes q in counterclockwise(clockwise) order, we denote this by $p \leq_{ccw} q$ ($p \leq_{cw} q$). Every reflex vertex u (of P) determines two components, a clockwise component $P_{cw}(u)$, consisting of the chain $P[u,u_{cw}]$ and the bounding chord $\overline{uu_{cw}}$; and a counterclockwise component $P_{ccw}(u)$ a counterclockwise component $P_{ccw}(u)$, consisting of the chain $P[u_{ccw},u]$ and the bounding chord $\overline{uu_{ccw}}$ (see Fig. 1). The bounding chords $\overline{uu_{cw}}$ and $\overline{uu_{ccw}}$ are called respectively the clockwise and the counterclockwise chords of u. A component is called *non-redundant* if it does not contain any other component.

A walk is said to be in an *s-deadlock* (*t-deadlock*) if two guards, both starting from s (t), have reached points from where neither can move forward without losing sight of each other. Points u and v illustrate this situation in Fig. 1 for two guards starting from s. Various cases of *s-deadlock* and *t-deadlock* are discussed in [11]. A walk with respect to a pair of points, (s,t), is deadlock free iff it is both s-deadlock free and t-deadlock free. Equivalently, a walk with respect to the pair (s,t) is s-deadlock (t-deadlock) free iff, for a walk originating at s (t),

the moment a guard enters a component, say Q, the other guard must be already inside Q. We will use this latter formulation in designing our algorithms.

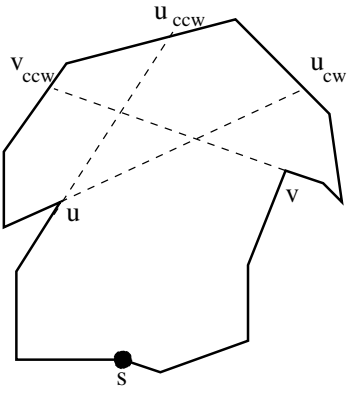


Fig. 1. Components $P_{cw}(u)$ and $P_{ccw}(v)$ creating s-deadlock

A polygon P is LR-visible [5] if there exist a pair of points s and t on P such that $P(s,t)$ and $P(t,s)$ are weakly-visible from each other. It was shown [8] that P is 2-walkable iff it is *LR-visible* and has a deadlock-free region. Bhattacharya and Ghosh [2] have given a simple linear time algorithm to determine if P is LR-visible. The algorithm also allows one to compute the shortest path tree of P from an arbitrary vertex. Thus if P is LR-visible, all its non-redundant components can be determined in linear time [4,5]. Using these components, it is easy to compute in $O(n)$ time all possible pairs of boundary chains $(A_i, B_i), i = 0, \ldots, m$, such that for any $s \in A_i$ and any $t \in B_i$, P is LR-visible with respect to the pair of boundary points (s,t). The above pairs of chains can be determined with either all A_i's disjoint or with all the B_i's disjoint. The disjoint ones are reported in counterclockwise order. This latter output is the input to our algorithms in this paper.

Additional conditions are necessary for a 2-walkable polygon to be (discrete) straight walkable. A polygon is straight walkable (discrete straight walkable) if there exists a pair of points s and t on P such that P is 2-walkable with respect to s and t, and the chains $P(s,t)$ and $P(t,s)$ do not have a configuration called a "wedge" ("semi-wedge").

3 Two-Walkability

3.1 Overview

From the pairs of chains $(A_i, B_i), i = 0, \ldots, m$ that describe the LR-visible pairs of points of P, we choose a set of point pairs $(s_i, t_i) \in A_i \times B_i, i = 0, \ldots, m$. The points satisfy the property that $s_0 \leq_{ccw} s_1 \leq_{ccw} \ldots \leq_{ccw} s_m \leq_{ccw} t_0 \leq_{ccw}$

$t_1 \leq_{ccw} \ldots \leq_{ccw} t_m \leq_{ccw} s_0$. We assume that the s_i's and the t_i's are not reflex vertices. We shall show in a later section how to choose such a set of point-pairs.

The main idea is to find a maximal deadlock-free region around each s_i and t_i by using a clockwise sweep by a clockwise guard and a counterclockwise sweep by a counterclockwise guard. This is achieved by a pair of "cooperating processes" termed as a *walk* where the two guards traverse the boundary of the polygon without losing sight of each other. In the i-th iteration, $walk_i$ can be described as follows: the counterclockwise guard, starting from s_i, is targeted to reach s_{i+1}, while the clockwise guard, also starting at s_i, is targeted to go as far as possible without crossing t_i. In particular, the clockwise guard begins to walk towards t_i, unless it encounters a reflex vertex v_l whose clockwise component $P_{cw}(v_l)$ does not contain the current position of the counterclockwise guard. Since any advancement of the clockwise guard would result in the loss of co-visibility, control passes to the counterclockwise guard who begins to walk from its current position in an attempt to enter this component. This walk of the counterclockwise guard can terminate in one of three different ways.

- It reaches its preassigned destination (s_{i+1}), in which case $walk_i$ is terminated after stacking some related parameters;
- It succeeds in entering $P_{cw}(v_l)$, in which case it waits, while the clockwise guard resumes advancing towards t_i;
- It encounters a reflex vertex v_r and $P_{ccw}(v_r)$ does not contain v_l in which case we report a deadlock. We now let the counterclockwise guard reach its target s_{i+1}, and terminate $walk_i$.

After we have generated all the walks, $walk_i$, $i = 0, \ldots, m$, the walk parameters left on the stack correspond to (s_i, t_i) pairs that permit s-deadlock free walks. We shall prove this claim in the next section.

3.2 Details of the Algorithm

To implement $walk_i$ described above, we need to be able to answer the following queries efficiently. Let the l and r be the current positions of the clockwise and the counterclockwise guards respectively. If l (r) is a reflex vertex, let l_{cw} (r_{ccw}) be the other endpoint of the bounding chords of $P_{cw}(l_i)$ ($P_{ccw}(r_i)$).

Visibility: Are $l, r \in P$ co-visible?
Clockwise containment: Is r in $P_{cw}(l)$?
Counterclockwise containment: Is l in $P_{ccw}(r)$?
Crossing-over event: Is $r \leq_{ccw} l_{cw}$?

Since (s_i, t_i) is an LR-visible pair, every component of P must contain one of s_i or t_i. Below, we will often appeal to this fact. In our description below, we differentiate between the cases: $i = 0$ and $i > 0$.

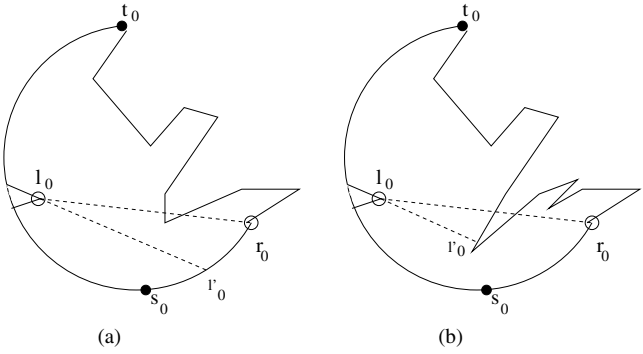


Fig. 2. l_{cw} can lie in $P(r_0, s_0]$ or on $P[t_0, r_0)$.

Implementing the Queries for $walk_0$.

Visibility Query: As seen below, when the guards are at location l_0 and r_0 during the walk process, we need to answer the visibility query. It may be noted that this is not a general visibility query since the two guards are monotonically traversing from point s_0 towards their destinations (s_1 and t_0). In order to answer this query efficiently (in constant amortized time), we preprocess P with respect to the pair (s_0, t_0). The preprocessing involves computing the "restricted shortest paths" from s_0 and t_0 to every vertex on P. A restricted shortest path from s_0 to vertex $u \in P(s_0, t_0)$ (or, $v \in P(t_0, s_0)$) is the shortest path between s_0 and u (v) ignoring the effect of $P(v, s_0)$ ($P(u, s_0)$). This is a standard procedure that has been used in several papers (refer to [4,2]) for exactly the same reason (i.e., answer visibility queries).

Clockwise Containment Query: Let $\overline{l_0 l_{cw}}$ denote the bounding chord of the component $P_{cw}(l_0)$. We determine, in *constant time*, if $l_{cw} \in P(t_0, s_0)$. Note that this is easily determined by inspecting the last edges on the shortest paths from s_0 and t_0 to l_0, as these paths do not cross. If *not*, then $P_{cw}(l_0)$ contains r_0. If *yes*, as $P(r_0, l_0)$ does not intersect $\overline{l_0 r_0}$, we test if $P(t_0, r_0)$ intersects this segment. If it *does not*, we easily determine if $l_{cw} \in P(r_0, s_0)$. If it *does*, we resolve between the two cases shown in Fig. 2 as follows. We traverse the boundary chain from r_0 to t_0 to find the first vertex x from which l_0 is visible. Clearly, x is reflex. Since P is LR-visible with respect to the pair (s_0, t_0), this vertex also has the property that all counterclockwise components of the reflex vertices of $P[x, r_0)$ contain l_0.

To find x, the counterclockwise guard walks towards t_0, stopping at s_1 (= *CCW.limit*) or x, whichever comes earlier. This stop becomes the new r_0. In either case, the property of nonintersection of the segment $\overline{l_0 r_0}$ and the chain $P(r_0, l_0)$ is preserved.

Counterclockwise Containment Query: Let $\overline{r_0 r_{ccw}}$ denote the counterclockwise chord of the component $P_{ccw}(r_0)$. We determine, in *constant time*, if

$r_{ccw} \in P(s_0, t_0)$. If *not*, it follows from the LR-visibility of P that the component $P_{ccw}(r_0)$ contains s_0 and, *a fortiori,* l_0. If *yes* we check, in constant time again, if the ray $\overrightarrow{r_0 r_{ccw}}$ is directed into the pocket determined by the chain $P[r_0, l_0]$ and the chord $\overline{r_0 l_0}$. If *yes* then the component $P_{ccw}(r_0)$ contains l_0, otherwise not.

Crossing-over Event Query: We must determine if the clockwise ray shot from l_0 hits r_0. In other words, the counterclockwise guard must find l_{cw} to effect the crossover. We know that $P[r_0, l_0]$ does not intersect the interior of $\overline{l_0 r_0}$. Also, $P[l_0, t_0]$ does not intersect the interior of $\overline{l_0 r_0}$. Hence $r_0 = l_{cw}$ iff a ray from r_0 in the direction of l_0 hits the chain $P[s_0, t_0]$.

We summarize the above discussions with the following claim:

Lemma 1. *We can check in (amortized) constant time if* $r_0 \in P_{cw}(l_0)$, $l_0 \in P_{ccw}(r_0)$, *and if the crossing-over event has occurred, i.e., when the counterclockwise guard reaches* l_{cw}.

Implementing the Queries for *walk$_i$*. We will now discuss how to answer the same queries for an arbitrary LR-pair (s_i, t_i), where $s_i \in P(t_0, s_0)$ and $t_i \in P(s_0, t_0)$. Let l_i and r_i denote the positions of the clockwise and counterclockwise guards respectively.

Clockwise Containment Query: The following cases arise.
Case 1: $l_i \in P[s_0, t_i]$ and $r_i \in P[t_0, s_i]$. This problem is the same as the clockwise containment problem for (s_0, t_0).
Case 2: $l_i \in P[s_0, t_i]$ and $r_i \in P[t_i, t_0]$. In this case $P_{cw}(l_i)$ contains r_i since l_{cw} cannot lie on $P[l_i, t_0]$.
Case 3: $l_i \in P[s_i, s_0]$ and $r_i \in P[t_i, t_0]$. Again, this is very similar to Case 1.
Case 4: $l_i \in P[s_i, s_0]$ and $r_i \in P[t_0, s_i]$. Clearly, if $l_{cw} \in P[s_0, t_0]$, $P_{cw}(l_i)$ does not contain r_i. Suppose this is not the case. As P is LR-visible with respect to (s_0, t_0), $l_{cw} \notin P[l_i, s_0]$. We know that $P(r_i, l_i)$ does not intersect $\overline{l_i r_i}$. Again, since P is LR-visible with respect to (s_0, t_0), $P[t_0, r_i)$ cannot intersect $\overline{l_i r_i}$. Hence, we can easily determine whether $l_{cw} \in P[r_i, l_i]$ or $l_{cw} \in P[t_0, r_i]$.

Counterclockwise Containment Query: Again, this query is resolved by examining the following cases, given that $P_{cw}(l_i)$ does not contain r_i.
Case 1: $l_i \in P[s_0, t_i]$ and $r_i \in P[t_0, s_i]$. This is similar to the case for (s_0, t_0).
Case 2: $l_i \in P[s_0, t_i]$ and $r_i \in P[t_i, t_0]$. In this case r_{ccw} must lie on $P[t_0, s_0]$, since P is LR-visible with respect to (s_0, t_0). Therefore, $P_{ccw}(r_i)$ contains l_i.
Case 3: $l_i \in P[s_i, s_0]$ and $r_i \in P[t_i, t_0]$. Again, this is very similar to Case 1.
Case 4: $l_i \in P[s_i, s_0]$ and $r_i \in P[t_0, s_i]$. Clearly, if $r_{ccw} \in P[s_0, t_0]$ then $P_{ccw}(r_i)$ does not contain l_i. Suppose $r_{ccw} \in P[t_0, s_0]$. Suppose the ray from r_i towards r_{ccw} is directed towards the interior of the pocket determined by $P[r_i, l_i]$ and $\overline{l_i r_i}$. In this case, $r_{ccw} \notin P[l_i, s_0]$ as shown in Fig. 3, otherwise l_i is not visible from $P[s_0, t_0]$, contradicting the LR-visibility of P with respect to (s_0, t_0). If the ray is directed away from the pocket, then $r_{ccw} \in P[l_i, s_0]$ and so $P_{ccw}(r_i)$ does not contain l_i.

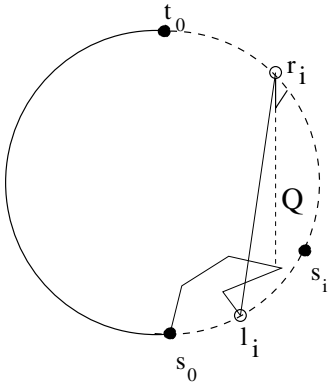


Fig. 3. r_{ccw} cannot lie on $P[l_i, s_0]$

Crossing-over Event Query: We detetmine if the ray shot from l_i hits r_i.
Case 1: $l_i \in P[s_0, t_i]$ and $r_i \in P[t_0, s_i]$. This problem is the same as the crossing-over event problem for (s_0, t_0).
Case 2: $l_i \in P[s_0, t_i]$ and $r_i \in P[t_i, t_0]$. This case cannot arise as $l_{cw} \in P[t_0, s_0]$.
Case 3: $l_i \in P[s_i, s_0]$ and $r_i \in P[t_i, t_0]$. This problem is the same as the crossing-over event problem for (s_0, t_0).
Case 4: $l_i \in P[s_i, s_0]$ and $r_i \in P[t_0, s_i]$. First we check if $l_{cw} \in P[s_0, t_0]$. If so, $r_i \neq l_{cw}$. Suppose $l_{cw} \in P[t_0, s_0]$. Since P is LR-visible with respect to (s_0, t_0), the first intersection of $P[t_0, l_i]$ in counterclockwise order with $\boldsymbol{l_i l_{cw}}$ is l_{cw}. Since $P(r_i, l_i)$ does not intersect $\overline{l_i r_i}$ in the interior, in our case $r_i = l_{cw}$. We summarize the above discussions as follows:

Lemma 2. *We can check in (amortized) constant time if $r_i \in P_{cw}(l_i)$, $l_i \in P_{ccw}(r_i)$, and if the crossing-over event has occurred, i.e., when $r_i = l_{cw}$.*

Deadlock-Test: The process $walk_i$ is described a slightly more general form in the procedure DEADLOCK-TEST as shown in Fig. 4. It accepts four arguments: the start points *CW.start* and *CCW.start* of the clockwise and counterclockwise guards, and their respective destinations, *CW.limit* and *CCW.limit*. The algorithm returns the maximal s-deadlock free chain $P[r, l]$ in $P[CCW.limit, CW.limit]$.

Several observations are in order. We note that if r_i and l_i are the current positions of the two guards, the boundary chain $P[r_i, l_i]$ does not obstruct their mutual visibility. This is easily seen from the invariance of the following two assertions during the movement of the guards from their start positions to the current positions.

- All the clockwise components generated by the reflex vertices on $P[s_i, l_i]$, with the possible exception of l_i, contain r_i.
- All the counterclockwise components generated by the reflex vertices on $P[r_i, s_i]$, with the possible exception of r_i, contain l_i.

Algorithm DEADLOCK-TEST($CW.start, CCW.start, CW.limit, CCW.limit$)
Output A pair of points (l, r). If $r \neq CCW.limit$, $P[r, l]$ has an s-deadlock.
begin
$l \leftarrow CW.start$; $r \leftarrow CCW.start$;
{ l (r) represents current position of clockwise (counterclockwise) guard }
{The clockwise guard moves}
Loop1: while (l is not a reflex vertex and $l <_{cw} CW.limit$) do
$l \leftarrow next(l)$;
if $CW.limit <=_{cw} l$ then
Output (CW.limit,CCW.limit)
{ $P[CCW.limit, CW.limit]$ *has no s-deadlock.*};
Return
else {l is a reflex vertex.}
if $P_{cw}(l)$ contains r then
$l \leftarrow next(l)$;
go to Loop1;
else go to Loop2
{ l is a reflex vertex and r is not contained in $P_{cw}(l)$}
{The counterclockwise guard moves}
Loop2: while (r is not a reflex vertex and $CCW.limit <_{cw} r$ and $l_{cw} <_{cw} r$) do
$r \leftarrow next(r)$;
if $r <=_{cw} l_{cw}$ then
{ counterclockwise guard has entered the clockwise component at l;}
{ hand over control to clockwise guard }
$r = l_{cw}$
$l \leftarrow next(l)$;
go to Loop1;
else if {r = CCW.limit }
Output (l, CCW.limit)
Return
else {r is reflex }
if $P_{ccw}(r)$ contains l then
{ counterclockwise guard continues in Loop 2}
$r \leftarrow next(r)$;
go to Loop2;
else {deadlock situation}
Output (l, r)
Return
end

Fig. 4. Algorithm DEADLOCK-TEST

In summary, we conclude with this theorem:

Theorem 1. *Given a pair of points (s_i, t_i) with respect to which P is LR-visible,* DEADLOCK-TEST(s_i, s_i, t_i, s_{i+1}) *returns a pair of points (l_i, r_i); if $r_i = s_{i+1}$, then the chain $P(r_i, l_i)$ is deadlock-free, else $P(r_i, l_i)$ is a deadlock region with respect*

to a walk starting at s_i. The cost is linear in the number of vertices contained in the chain $P(r_i, l_i)$.

3.3 Reporting All Deadlock-Free Pairs

We now show how to combine the results of the calls to procedure DEADLOCK-TEST from consecutive iterations. Suppose that after iterations i and $i+1$, DEADLOCK-TEST indicate that $P[r_i, l_i]$ and $P[r_{i+1}, l_{i+1}]$ are deadlock free. Since $r_i = s_{i+1}$ and $l_{i+1} \leq_{ccw} s_{i+1}$, the two regions overlap and we can combine the results to claim that $P[r_{i+1}, l_i]$ is deadlock-free. Furthermore, in order to avoid either the clockwise guard or the counterclockwise guard from repeatedly going over any overlapping portions in different iterations (which will happen if $l_{i+1} \leq_{ccw} s_i$), the walks make use of information stored on a special *snapshot-stack* and skip over these regions during their walks.

Whenever an iteration reports a deadlock region, then the corresponding boundary chain of P is marked as *ineligible*. Future iterations will skip over this portion of the polygon, traversed by the clockwise guard.

The algorithm ALL-POINTS-PAIRS implements the walks for all the iterations by making calls to DEADLOCK-TEST to determine all deadlock free regions.

Algorithm ALL-POINTS-PAIRS
Output Boundary chains that admit s-deadlock.
begin
Set $i \leftarrow 0$; $s_{m+1} \leftarrow t_m$;
while ($i \leq m$) do
CW.start $\leftarrow$ CCW.start $\leftarrow s_i$; CW.limit $\leftarrow t_i$; CCW.limit $\leftarrow s_{i+1}$;
Loop: $(l, r) \leftarrow$ DEADLOCK-TEST(CW.start, CCW.start, CW.limit, CCW.limit)
If r = CCW.limit then
PUSH(snapshot-stack,l, r,CW.limit);
$i \leftarrow$ index(CCW.limit);
else
Label the chain $P(r, l)$ as ineligible.
if (snapshot-stack $\neq$ empty) then
(a,b,c) $\leftarrow$ POP(snapshot-stack);
if $P(b, a)$ does not contain l then
CW.start $\leftarrow l$; CCW.start $\leftarrow r$; Go to Loop;
else
CW.start $\leftarrow$ a; CCW.start $\leftarrow r$; CW.limit $\leftarrow c$; Go to Loop;
else $i \leftarrow$ index(CCW.limit);
Traverse the ineligible boundary chains and remove all pairs whose associated s points lie on these parts. Report the non-deleted pairs as the pairs that allow s-deadlock free walk.
end

ALL-POINTS-PAIRS needs to maintain lists of boundary objects not traversed by the clockwise guard yet, as well as a list of boundary objects labeled as "ineligible". Both lists are easily maintained as stacks.

Since the boundary chains are traversed at most once in clockwise order and at most once in counterclockwise order and since the subroutine DEADLOCK-TEST outputs chain in time proportional to the size of the chain, we claim therefore:

Lemma 3. *All the pairs of* $(s_i, t_i), i = 0, 1, \ldots, m$ *that allow* s*-deadlock free and* t*-deadlock free walks can be correctly determined in* $O(n)$ *time.*

Given the lists produced in Lemmas 3 it is easy determine the pairs of points with respect to which the polygon is 2-walkable. Therefore, we have:

Theorem 2. *Given* $(s_i, t_i), i = 0, 1, \ldots, m$ *and* P*, it is possible to determine in optimal time the pairs of points among* $(s_i, t_i), i = 0, 1, \ldots, m$ *with respect to which* P *is 2-walkable.*

3.4 Reporting All Pairs of Bounding Chains with Respect to which P has no s-Deadlock

[4,5,11] gave methods to determine the pairs of bounding chains $(A_i, B_i), i = 0, 1 \ldots m$ of P such that for any arbitrary $(s, t) \in (A_i, B_i)$, the polygon P is LR-visible. The pairs can be determined with either A_i's all disjoint or B_i's all disjoint. We assume all A_i's to be disjoint when they are tested for s-deadlock and assume all B_i's to be disjoint when they are tested for t-deadlock.

We describe our method for s-deadlock only. Our objective is to identify the parts A'_i and B'_i of A_i and B_i respectively such that for any arbitrary $(s, t) \in (A'_i, B'_i)$, the polygon P is s-deadlock free. Let us consider the pair (A_i, B_i). In order to test (A_i, B_i) for s-deadlock, we replace this pair (A_i, B_i) by a set of points $(s_j, t_j), j = 0, 1, \ldots$ where $s_0, s_1, \ldots$ belong to A_i and $t_0, t_1, \ldots$ belong to B_i such that s-deadlock free chains corresponding to (A_i, B_i) can be determined once we know the status of $(s_j, t_j)'$s for s-deadlock.

Let $v_1, v_2, \ldots, v_k$ be the reflex vertices on $A_i = p_{cw}(a_i, a'_i)$. Let $v_0 = a_i$ and $v_{k+1} = a'_i$. Let $A_{i_j} = P(v_{j-1}, v_j), j = 1, 2, \ldots, k+1$. Let (s_j, t_j) be any arbitrary pair in (A_{i_j}, B_i). We can show that

Lemma 4. P *is* s*-deadlock free for chain* (A_{i_j}, B_i) *if and only if* P *is* s*-deadlock free for the pair* (s_j, t_j)*.*

From the above lemma we see that we can replace each chain pair by point pairs whose number is one more than than the number of reflex vertices in A_i. Since A'_is are disjoint, the total number of points generated is at most n. Also it should be noted that s's are all non-reflex points. Thus we have the two theorems below.

Theorem 3. *Given LR-visible chain pairs* $(A_i, B_i), i = 1, \ldots, m$*, where all* A_i*'s are disjoint, all* s*-deadlock-free chain pairs can be determined in* $O(n)$ *time.*

Theorem 4. *All 2-walkability pairs of an arbitrary polygon* P *can be determined in optimal linear time.*

4 Straight Walkability

Our input is a set chain-pairs (A_i, B_i) such that each chain-pair (A_i, B_i) admits 2-walkability. We outline an algorithm to determine if P is straight walkable and, if so, determine the chain-pairs which admit such walkability.

The concept of a wedge is important for determining whether P is straight walkable [8]. A wedge is a chain $P[v, u]$, defined by two reflex vertices u and v such that $P_{ccw}(u)(P_{cw}(v))$ contain $v(u)$ and the clockwise chord $\overline{uu}_{cw}$ and the counterclockwise chord $\overline{vv}_{ccw}$ intersect. A wedge is called non-redundant if it does not contain any other wedge or a component. Otherwise, it is called redundant.

Let W be any wedge in P. Tseng at al [11] showed that

Lemma 5. *If P is straight walkable with respect to (s, t) then either $s \in W$ or $t \in W$.*

Therefore P is not straight walkable if it has at least three or more disjoint wedges. To determine if P is straight walkable we proceed along the following lines. Let P be 2-walkable with respect to the point pair (s, t). We compute all the non-redundant wedges in stages. First, all the non-redundant wedges containing s or t are computed. Then we compute all the non-redundant wedges contained in $P(s, t)$ and $P(t, s)$, keeping track of the number of disjoint wedges computed so far. We stop if three disjoint wedges are detected. We can implement the above ideas using some of the techniques developed in Section 3. Hence

Theorem 5. *We can determine all pairs of chains with respect to which P is straight walkable in linear time.*

5 Discrete Straight Walkability

The problem of determining the chain-pairs with respect to which P is discretely straight walkable can also be determined in a similar manner.

The concept of a semi-wedge [9] is important here. A semi-wedge is a chain $P[v, u]$, defined by two reflex vertices u and v such that $P_{ccw}(u)(P_{cw}(v))$ contain $v(u)$ and the clockwise chord $\overline{uu}_{cw}$ (counterclockwise chord $\overline{vv}_{ccw}$), while u_{ccw} and v_w belong to the relative interior of some edge. A semi-wedge is called non-redundant if it does not contain any other semi-wedge or wedge. Otherwise, it is called redundant.

Analogous to the result by [11], if W is a semi-wedge of P, it was shown in [9] that

Lemma 6. *If P is discretely straight walkable with respect to (s, t) then either $s \in W$ or $t \in W$.*

This implies that if P is not discretely straight walkable with respect to any (s, t) pair if it contains three disjoint non-redundant semi-wedges.

Our algorithm proceeds in a similar manner: determine three disjoint semi-wedges with respect to an arbitrary (s, t) pair. Details are provided in Appendix C. The claims are summarized in the following theorem:

Theorem 6. *We can determine all pairs of chains with respect to which P is discretely straight walkable in linear time.*

6 Conclusions

In this paper, we present surprisingly simple and efficient (optimal, linear-time) algorithms for many 2-guard problems. We believe that this work will help to bridge the gap between impractical linear-time algorithms for visibility problems and practical efficient implementations.

References

1. E. M. Arkin, M. Held, J. S. B. Mitchell, S. S. Skiena. Hamiltonian Triangulations for Fast Rendering. *Visual Computer*, 12:429-444, 1996.
2. B. K. Bhattacharya and S. K. Ghosh. Characterizing LR-visible polygons and related problems. *Proc. of CCCG 1998; (accepted to CGTA, 2000).*
3. B. Chazelle. Triangulating a simple polygon in linear time. *Proc. of the 31-st Annual IEEE FOCS* 220-229, 1990.
4. G. Das, P. J. Heffernan and G. Narasimhan. Finding all weakly-visible chords of a polygon in linear time. *Nordic J. of Comput.* 1:433-457, 1994.
5. G. Das, P. J. Heffernan and G. Narasimhan. LR-visibility in polygons. *Computational Geometry: Theory and Applications* 7:37-57, 1997.
6. L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear time algorithms for visibility and shortest path problems inside a triangulated simple polygon. *Algorithmica*, 2:209–233, 1987.
7. P. Heffernan. An optimal algorithm for the two-guard problem. *Intl. Journal on Comp. Geometry and Applications* 6:15-44, 1996.
8. C. Icking and R. Klein. The two guards problem. *Intl. Journal on Comp. Geometry and Applications* 2:257-285, 1992.
9. G. Narasimhan. On Hamiltonian Triangulations of Simple Polygons. *Intl. Journal on Comp. Geometry and Applications* 9(3):261-275, 1999.
10. J. O'Rourke. Art Gallery Theorems and Algorithms. Oxford University Press, New York, NY, 1987.
11. L.H. Tseng, P. Heffernan and D.T. Lee. Two-guard walkability of simple polygons. *Intl. Journal on Comp. Geometry and Applications* 8(1):85-116, 1998.

Movement Planning in the Presence of Flows*

John Reif[1] and Zheng Sun[1]

Duke University, Durham, NC 27708, USA

Abstract. This paper investigates the problem of time-optimum movement planning in $d = 2, 3$ dimensions for a point robot which has bounded control velocity through a set of n polygonal regions of given translational flow velocities. This intriguing geometric problem has immediate applications to macro-scale motion planning for ships, submarines and airplanes in the presence of significant flows of water or air. Also, it is a central motion planning problem for many of the meso-scale and micro-scale robots that recently have been constructed, that have environments with significant flows that affect their movement. In spite of these applications, there is very little literature on this problem, and prior work provided neither an upper bound on its computational complexity nor even a decision algorithm. It can easily be seen that optimum path for the $d = 2$ dimensional version of this problem can consist of at least an exponential number of distinct segments through flow regions. We provide the first known computational complexity hardness result for the $d = 3$ dimensional version of this problem; we show the problem is PSPACE hard. We give the first known decision algorithm for the $d = 2$ dimensional problem, but this decision algorithm has very high complexity. We also give the first known efficient approximation algorithms with bounded error.

1 Introduction

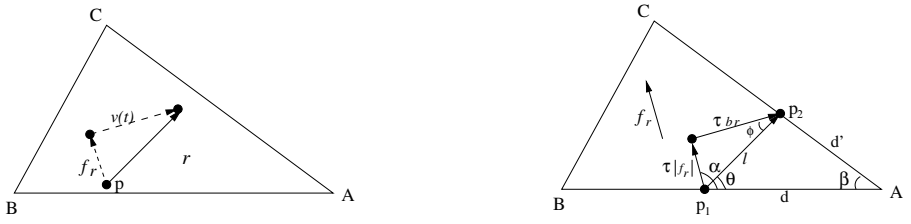
1.1 Formulation of the Problem and Motivation

We assume the problem is given as a polyhedral decomposition of d-space, where each region r defined by the polyhedral decomposition has an assigned *translational flow* defined by a vector f_r. There is also associated with each region r a non-negative real number b_r giving the maximum Euclidean norm of the control velocity that the robot can apply within region r. The robot is considered to be a point with a given initial position and also a given final position to be reached by the robot.

At time $t = 0$, the point robot is at some given initial position point. Within each region r, the robot can apply, at each time $t \geq 0$ and in any direction, a translational *control velocity vector* $v(t)$ of bounded Euclidean norm $|v(t)| \leq b_r$. However, the *actual velocity* of the robot at time t is given by the sum $v(t) + f_r$ of it's control velocity vector $v(t)$ and the translational flow velocity f_r of region r, as shown by Figure 1.a.

* Supported by NSF ITR EIA-0086015, NSF-IRI-9619647, NSF CCR-9725021, SEGR Award NSF-11S-01-94604, Office of Naval Research Contract N00014-99-1-0406.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 450-461, 2001.



(a) Composite Velocity

(b) Optimum Path is Straight Line Segment

Fig. 1.

The *flow path optimization problem* is to find an *optimum path* of movement of the point robot from the initial position to the final position with minimum time duration. The *flow path decision problem* is given a rational number $\tau > 0$, determine if the flow path optimization problem has time τ.

In an extreme example, where $b_r = 0$ at each region r, then the movement of the robot is simply a sequence of translations provided by each region's flow, and the problem reduces to the prediction of the path of the robot in the presence of overwhelming flow velocities where the robot has no control of movement. In an another extreme example, where $|f_r| = 0$ for each region r, then this problem reduces to the usual weighted shortest path problem through regions of given translational velocities.

In our investigation of the computational complexity of this problem, we assume that the polygonal decomposition of input problem has n regions, and that the input problem is specified with a total of $n^{O(1)}$ bits: in particular, for some constant $c \geq 1$, we assume that we are given the following within cn bits:

- the positions of the boundaries of these regions,
- the initial and final positions of the robot,
- the flow velocity and bounding velocity of the robot within each region.

This flow path problem has a number of macro-scale movement planning applications:

- The problem of moving a ship on the surface of an ocean or river though regions where the surface currents have known flow velocity.
- The problem of moving a submarine though regions where the underwater currents have known flow velocity.
- The problem of moving a aircraft though regions where the air currents have known flow velocity.

The flow path problem becomes particularly relevant to these practical problems in the case where the object to be moved is under autonomous control, and where the flow velocities are significant to require careful motion planning. This is an increasing occurrence as new robotic devices are developed that are of a rapidly decreasing size, and hence these meso and micro-scale robots can be strongly influenced by the local flows in their environment.

1.2 Previous Work and Our Results

Papadakis and Perakis [10,11] previously gave heuristic algorithms for related problems such as for minimal time vessel routing in ocean currents. Sellen [18] studied the optimal route problem for a sailboat in a single region with multiple obstacles, where the velocity is a continuous function of sailing direction. There seems not to have been much other previous research on this problem, but there is considerable previous research on related movement problems.

Reif [12] provided the first PSPACE hardness result for a robotic motion planning problem, and Schwartz and Sharir [17] gave motion planning algorithms using the theory of real closed fields (Canny [3]). Reif and Sharir [13] gave algorithms and computational complexity lower bound results for robotic motion with moving obstacles (also see Wilfong [19]).

Canny and Reif [4] showed the $3\mathcal{D}$ minimal cost path problem with polygonal obstacles is NP hard, and Reif and Storer [14] applied the theory of real closed fields to give a decision algorithm for this problem. The reference [8] surveys work on the $2\mathcal{D}$ and $3\mathcal{D}$ minimal cost path problem, and approximation algorithms for the weighted region minimal cost path problem include the continuous Dijkstra method of Mitchell and Papadimitriou [9], as well as a variety of other discretization algorithms given by Mata and Mitchell [7], Lanthier *et al* [6], Aleksandrov *et al* [1]. More recent works include Reif and Sun [16,15], and Aleksandrov *et al* [1,2].

In Section 1, we have defined and motivated the flow path problem, and stated our results. In Section 2, we provide some preliminary results on the geometry of optimum paths for flow path problems, including a simple example where the $2\mathcal{D}$ version of the for flow path problem consists of an exponential number of distinct segments through flow regions. In that section, we also provide a proof that the $3\mathcal{D}$ version of the flow path problem is PSPACE hard, which is the first known hardness result for the computational complexity of this problem. In Section 3, we provide the first known decision algorithm for the $2\mathcal{D}$ flow path problem. This decision algorithm is of theoretical interest only, but is proved by a rather interesting and unique inductive argument that repeatedly makes use of root separation bounds derived from the theory of real closed fields. In Section 4, we provide the first known approximation algorithm for the $2\mathcal{D}$ flow path problem, which is efficient for any given bounded error. In Section 5, we conclude the paper with some open problems.

2 Preliminary and Lower Bound Results

We first state some relevant properties of optimum paths for flow path problems within regions of translational flow:

Proposition 1. *The optimum path is a simple path: it does not self-intersect.*

Proposition 2. *To move between any two points within the same flow region r, the optimum path is a straight line path with a control velocity of fixed direction and fixed maximum modulus b_r.*

Next, we provide some lower bound results for optimum paths. First we give a simple construction of a family of flow problems where the optimum path contains an exponential number of distinct straight-line segments.

Theorem 1. *In the $d = 2$ dimensional version of the flow path problem with $O(1)$ regions and with flow rates specified by n bits, the optimum path can consist of at least $2^{\Omega(n)}$ distinct straight-line segments through flow regions.*

Proof: We construct a flow path problem where there are four unit square flow regions on the plane, each bordering the origin, and with unit magnitude flows that force a point robot with control velocity magnitude 2^{-n} starting at the origin to spiral an exponential times around the origin before reaching the destination point $(1, 0)$.

In particular, for $i = 0, 1, 2, 3$, define angle $\theta_i = i\pi/2 + \pi/4$ and let the ith flow region be a unit square centered at point $(\frac{1}{2}\cos(\theta_i), \frac{1}{2}\sin(\theta_i))$, with unit flow direction $\theta_i + \pi/2$. Thus the concatenation of the flows of the four squares run in counterclockwise fashion around the origin. Starting at the origin, the goal point $(1, 0)$ can only be reached by an exponential number of cycles by the point robot around the origin, where on each cycle the point can get an additional distance $c2^{-n}$ further from the origin, for a fixed constant $c > 0$. □

Next we prove that the $d = 3$ dimensional version of this problem is PSPACE hard. We use a construction given by Canny and Reif [4] in their proof that the problem of finding a minimal path in $3\mathcal{D}$ between two points and avoiding a set of polygonal obstacles is NP hard. Given a Boolean formula with a list of N variables X, they construct a polynomial number of obstacles of polynomial size description, with a start point, an end point and an intended path length L such that the formula is satisfiable iff there is a path of distance $\leq L$ between the start and end points avoiding these obstacles. The construction represents each possible variable assignment A of X by a (binary) encoding as a point $p(A)$ on a particular segment of an obstacle; each point $p(A)$ of the segment is reachable from the start point by path of distance at most L iff A is a satisfying assignment.

We use their construction as follows:

We begin with a linear space Turing Machine M with given binary input of length n. We can assume without loss of generality that M accepts the input only on computations of 2^{cn} steps, for some constant $c \geq 1$. Let X, X' each be lists of N Boolean variables, where $N = cn + O(1)$. We define a Boolean formula $NEXT(X, X')$ which is true iff X, X' each encode valid configurations of M and the configuration associated with X' is reachable from the configuration associated with X by one step of the machine M. Then we apply the Canny and Reif [4] construction to form a set of obstacles in $3\mathcal{D}$ of polynomial size with distinguished obstacle segments e, e' associated with variable lists X, X' respectively. Again, for any Boolean assignment A of X there is a point $p(A)$ of e encoding A, and for any Boolean assignment A' of X' there is a point $p'(A')$ of e' encoding A'. The Canny and Reif [4] construction ensures that point $p'(A')$ of the segment e' is reachable from the point $p(A)$ of the segment e in distance at most L' iff $NEXT(X, X')$ holds for these Boolean variable assignments A, A' of

Boolean variable lists X, X' respectively. We define the modulus of the control velocity of the point robot to be unit within the free space contained within the set of obstacles, and the the modulus of the control velocity of the point robot to be 0 within the obstacles.

We now provide for a translational "ribbon" of flow providing time paths of time duration $\leq 2^{-c'n}$, for a sufficiently large constant $c' > 1$ from points of e' back again to corresponding points of e. This 'ribbon" has a cross-section that is a rectangle of unit length and $\leq 2^{-\Omega(n)}$ width. The purpose of this "ribbon" is to allow each of the single step transitions of the Turing machine to be repeated without changing the encoding of configurations. In particular, choosing a sufficiently large constant $c'' > 1$, we define an additional polynomial sequence of connected rectangular flow strips (comprising the "ribbon" and external to the previously defined obstacles) each with a translational rate of flow $2^{c''n}$ along the "ribbon". Then the time of the robot to move via this "ribbon" from a point $p'(A')$ of e' to a $p(A')$ of e is at most $2^{-c'n}$.

Furthermore, we define the start point of the robot to be a point $p(A_0)$ of e such that A_0 encodes the initial configuration of machine M, and we define the destination point of the robot to be a point $p(A_f)$ of e' such that A_f encodes the accepting configuration of machine M. Then it follows by this construction that M has an accepting computation of 2^{cn} steps iff the flow path problem has an optimum path of time duration $L'(1+2^{-c'n})$ for the point robot to move between the start and destination points. This construction can easily be shown to require $O(\log n)$ deterministic space to determine the specification of the resulting flow path problem. Hence we have shown:

Theorem 2. *The flow path problem in three dimensions is PSPACE hard with respect to log-space reductions.*

3 A Decision Algorithm for the 2$\mathcal{D}$ Flow Path Problem

Here we develop a decision algorithm for the two dimensional flow path problem. We can assume, with out loss of generality, that all flow regions are triangles.

We first state the following two lemmas (refer to Figure 1.b) for two dimensional flow path problems. We will apply them to develop a decision algorithm.

Lemma 1. *Let $r = \triangle ABC$ be a region with flow f_r. Let p be a point on $\overline{AB}$ with distance d to A and p' be a point on $\overline{AC}$ with distance d' to A. Let α be the angle between $\overline{AB}$ and f_r and let θ be the angle between $\overline{pp'}$ and $\overline{AB}$. Let ρ_r be the ratio of b_r and $|f_r|$. Then the optimum path from p to p' can be achieved by the point robot adopting a velocity with maximum magnitude and an angle of $\Phi = \arcsin(\frac{\sin(\alpha-\theta)}{\rho_r})$ from $\overline{pp'}$.*

Lemma 2. *Let β be the angle between $\overline{AB}$ and $\overline{AC}$. Then the optimum path from p to p' inside r has a cost of $\tau = \frac{l^2}{b_r(\sqrt{l^2-T_1^2}+T_2)}$, where $l = |pp'| = \sqrt{d^2 + d'^2 - 2dd'\cos\beta}$, $T_1 = (d\sin\alpha - d'\sin(\alpha+\beta))/\rho_r$ and $T_2 = (d\cos\alpha - d'\cos(\alpha+\beta))/\rho_r$.*

The next theorem directly follows Lemma 2.

Lemma 3. *Let $s_r(d, d')$ be an optimum path segment for the robot within a convex flow region r that begins at a distance d along a boundary edge e_1 of r, and ends at a distance d' along a boundary edge e_2 of r. Then there is a formula $F(t, x, x')$ of the existential theory of real closed fields (with free variables t, x, x' and involving only a constant number of other variables; furthermore with a constant number of terms and with constant rational coefficients of most $O(n)$ size), such that $F(\tau, d, d')$ is true iff $s_r(d, d')$ has time duration τ.*

The *existential theory of real closed fields* is the logical system consisting of existentially quantified formulas, whose variables range over the real numbers, and whose formulas are constructed of inequalities of rational forms (these rational forms are arithmetic expressions involving these real variables and fixed rational constants which may be added and multiplied together) and the usual Boolean logical connectives AND, OR, NOT. Collins [5] gave a decision procedure for the existential theory of real closed fields that was improved by Canny [3] to run in polynomial space:

Lemma 4. *Given a formula of the existential theory of real closed fields of length n, the formula can be decided in $n^{O(1)}$ space and $2^{O(n)}$ time, and the existentially quantified variables can be determined, up to exponential bit precision, within this computational complexity.*

Collins [5] proved a useful Lemma as a byproduct of his decision procedure:

Lemma 5. *If the solution of a formula of length n in the existential theory of real closed fields is not the zero vector $\mathbf{0}$, then it is of modulus at least $2^{-2^{cn}}$, for some constant $c > 0$.*

Now consider an optimum path $S(d, d')$ for the robot that begins at a distance d along a boundary edge e of a flow region r, and ends at a distance d' along the same boundary edge e of r, and does not visit any points of e between these, but may pass through f (where f counts the repetitions) other flow regions. Then $S(d, d')$ consists of at most $O(f)$ straight-line segments, though flow regions. By Lemma 3, we have:

Lemma 6. *There is a formula $F'(t, x, x')$ of the existential theory of real closed fields (with free variables t, x, x' and involving only $O(f)$ other variables, and with a quadratic number of terms, and with constant rational coefficients of most $O(n)$ size), such that $F'(\tau, d, d')$ is true iff $S(d, d')$ has time duration τ.*

We now define a hierarchy of paths of increasing *complexity* (this term should not be confused with the usual notion of computational complexity). Let a path have *complexity* 0 if it visits no flow region boundary edge more than once, except possibly at the start and final points of the path. Let a path p have *complexity* k if it does not have complexity $< k$ and $p = q_0 p_1 q_1 \ldots, p_h q_h$ where each p_i is a path that passes through only one flow region, and each q_i is a path of complexity $< k$ that begins and ends at the same flow region boundary edge. The following can be proved by induction on the number of distinct flow edges:

Proposition 3. *The maximum complexity number of an optimum path in any 2D flow path problem with n triangular flow regions is at most* $n^{O(1)}$.

We will derive, for any optimum path of complexity 0, a finite bound on the number of straight-line segments through flow regions. Consider an optimum path $S(d, d')$ of complexity 0 that begins and ends at a distance d, d' respectively along a flow region boundary edge e of a flow region r. Applying Lemma 5 to Lemmas 3 and 6, we have: either $d = d'$ or $|d' - d| \geq 2^{-2^{cn}}$, for some constant $c > 0$. This implies:

Lemma 7. *Any optimum path of complexity* 0 *between two points consists of at most* $2^{2^{O(n)}}$ *straight-line segments through flow regions.*

In the following, let $E_0(n) = n$ and for $k > 0$, let $E_k(n) = 2^{E_{k-1}(n)}$.

Now suppose as an inductive assumption that, for some $k \geq 0$, that any optimum path of complexity $k' < k$ between two points consists of at most $E_{2k'}(O(n))$ straight-line segments through flow regions. Applying Lemma 6, we can construct an existentially quantified formula of the theory of real closed fields with $E_{2k'}(O(n))$ variables, which is true iff there is a optimum path of complexity k' and of length τ between the initial and final points.

Consider an optimum path $S(d, d')$ of complexity k that begins and ends at a distance d, d' respectively along a flow region boundary edge e of a flow region r. Recall that since $S(d, d')$ has complexity k, it can be decomposed as $q_0 p_1 q_1 \ldots, p_h q_h$ where each p_i is a path that passes through only one flow region, and each q_i is a path of complexity $< k$ that begins and ends at the same flow region boundary edge. Applying again Lemma 5 to Lemmas 3 and 6, we now have: either $d = d'$ or $|d' - d| \geq 2^{-2^{O(E_{2(k-1)}(O(n)))}} \geq 1/E_{2k}(O(n))$. Hence we have that:

Lemma 8. *Any optimum path of complexity k between two points consists of at most* $E_{2k}(O(n))$ *straight-line segments through flow regions.*

Then applying the Canny [3] decision procedure (Lemma 4), we have that there is an algorithm that decides, within $E_{2k}(O(n))$ space and $2^{O(E_{2k}(O(n)))}$time, the two dimensional flow path problem for optimum paths of complexity k. By Proposition 3 (which bounds the complexity number of an optimum path to be $n^{O(1)}$), we have one of our main results:

Theorem 3. *There is an algorithm, with* $E_{2k}(O(n))$ *space and* $2^{O(E_{2k}(O(n)))}$ *time cost, for the two dimensional flow path decision problem of size n, where* $k \leq n^{O(1)}$ *is the minimum complexity of an optimum path.*

4 An Efficient ϵ-Short Approximation Algorithm

4.1 Approximation Algorithm Based on Discretization

The significance of the above algorithm is that the problem is decidable, but its complexity is far too high for practical implementation. A natural strategy to

approximately solve the $2\mathcal{D}$ flow problem is to discretize the polyhedral decomposition of the $2\mathcal{D}$ space by inserting Steiner points on edges. A directed discrete graph G is then constructed by interconnecting each pair of Steiner points or vertices on the boundary of the same region. Each edge (p_1, p_2) connecting Steiner point (or vertex) p_1 to p_2 is assigned a weight that is equal to $\tau(p_1, p_2)$.

Dijkstra's algorithm can be used to find the shortest path from the given source point s to the destination point t in G. This shortest path found in G can be used to approximate the "true" optimum path in the original continuous space. This approach has been used in approximating the geometric shortest path in the $2\mathcal{D}$ or $3\mathcal{D}$ space, or the optimum path in weighted regions in $2\mathcal{D}$ space.

The basic discretization scheme is to place m Steiner points uniformly on each edge, for some positive integer m. In G there will be $O(nm)$ points and $O(nm^2)$ edges and thus the time complexity of Dijkstra's algorithm is $O(nm^2 + nm\log(nm))$. Using a discretization algorithm done by Sun and Reif ([15]) named BUSHWHACK, the optimum path in G can be computed in $O(nm \log(nm))$ time without visiting all edges. This scheme is easy to implement, although no error bound can be guaranteed for the approximate optimum path.

4.2 Non-uniform Discretization

The above approximation algorithm can be improved by adopting a non-uniform discretization. This discretization scheme is similar to the one introduced by Aleksandrov *et al* for the weighted region optimum path problem [1,2]. Instead of placing Steiner points with even spacing on each edge, we place Steiner points with higher density in the portions of the edge that are closer to the two end points. The discretization scheme assumes that $\rho_r > 1$ for any region r.

Before we describe the new discretization scheme, we first introduce some notations that will be used. These notations are, collectively, $\tau_{min}(p), p_e, b_e$ and R_v.

Let p be an arbitrary point on the boundary of some region r. We define the *vicinity* of p, V_p, to be the union of all boundary edges each of which is not incident to p yet is on the same region boundary as p. We define $\tau_{min}(p) = \inf\{\min\{\tau(p, v), \tau(v, p)\} \mid v \in V_p\}\}$. In the full version of the paper we show that $\tau_{min}(p)$ can be computed in constant time for each p. Observe that $\tau_{min}(p)$ represents the minimum cost of traveling in a straight line path between p and any point on the boundary of the union of all regions incident to p.

For each edge e, we let $\tau_{max}(e) = \sup\{\tau_{min}(p) \mid p \in e\}$. $\tau_{max}(e)$ is a constant that is decided by the geometric properties as well as f_r and b_r of each region r incident to e. We let p_e denote a point on e such that $\tau_{min}(p_e) = \tau_{max}(e)$. $\tau_{min}(e)$ and p_e can be determined in constant time.

Let r_1 and r_2 be the two regions incident to e, and let α_1, α_2 be the angles between $\overline{v_2v_1}$ and f_{r_1}, f_{r_2} respectively. We define $b_{v_1,v_2} = \max\{\sqrt{b_{r_1}^2 - \sin^2\alpha_1 \cdot |f_{r_1}|^2} + |f_{r_1}| \cdot \cos\alpha_1, \sqrt{b_{r_2}^2 - \sin^2\alpha_2 \cdot |f_{r_2}|^2} + |f_{r_2}| \cdot \cos\alpha_2\}$. b_{v_1,v_2} is the "effective velocity" of a robot traveling on edge e from v_1 to v_2 as

it takes time $d/b_{v_1,v_2}$ to travel distance d on e following Lemma 2. Similarly, we can define b_{v_2,v_1} and let $b_e = \min\{b_{v_1,v_2}, b_{v_2,v_1}\}$. Note that b_e is the minimum effective velocity of the robot traveling on edge e in any of the two directions.

For any vertex v and any triangular region r incident to v, we define $d_{min}(v,r)$ to be the minimum Euclidean distance from v to the edge of r that is not incident to v. We define the *radius* of v to be the following: $R_v = \min\{\frac{d_{min}(v,r)}{b_r+|f_r|} \mid r \text{ is incident to } v\}$. Observe that while $\tau_{min}(v)$ is the minimum cost of one-segment path between v and any point on the boundary of the union of regions incident to v, R_v provides a lower bound on the minimum cost of any path between v and any point on the boundary of the union.

For any edge $e = \overline{v_1v_2}$, p_e divides edge e into two segments, $\overline{v_1p_e}$ and $\overline{v_2p_e}$. In the following we describe the placement of Steiner points $p_{i,1}, p_{i,2}, \cdots, p_{i,k}$ on each segment $\overline{v_ip_e}$ for $i = 1, 2$. The first Steiner point $p_{i,1}$ is placed on segment $\overline{v_ip_e}$ with distance $b_eR_{v_i}\epsilon$ to vertex v_i. The subsequent Steiner point $p_{i,j}$ is placed between $p_{i,j-1}$ and p_e with distance $\epsilon b_e\tau_{min}(p_{i,j-1})$ to $p_{i,j-1}$. We continue adding Steiner points until no more Steiner point can be added on $\overline{v_ip_e}$. That is, if $p_{i,j}$ is the last Steiner point added, no more Steiner point is inserted on segment $\overline{v_ip_e}$ if $|\overline{Ap_{i,j}}| + \epsilon b_e\tau_{min}(p_{i,j}) \geq |\overline{v_ip_e}|$. Finally, we add p_e as a Steiner point on e.

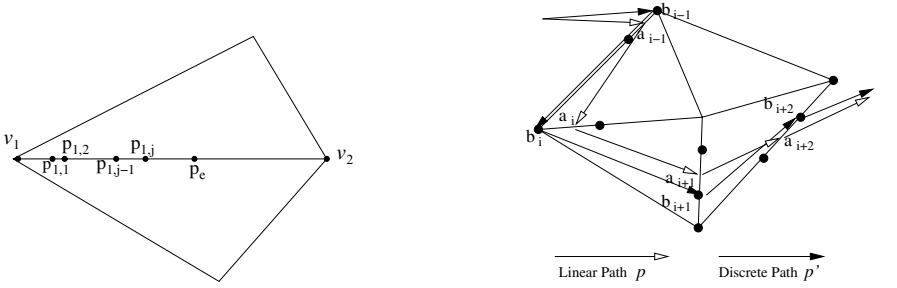
To give an upper bound for the number of Steiner points on each edge e, we introduce three parameters θ_{min}, ρ_{min} and λ. We let θ_{min} be the smallest angle of any triangular region and let ρ_{min} be the minimum b_r among all regions. We define λ to be $\max\{b_r/b_{r'} | \text{regions } r \text{ and } r' \text{ are adjacent}\}$. λ indicates how drastic the velocity bound of the robot can change from one region to another. In practice, since the triangular decomposition is usually a result of discretization, λ is not very large in many cases. Further, we let $C_{skew} = \frac{\lambda\cdot(\rho_{min}+1)}{\sin\theta_{min}\cdot(\rho_{min}-1)}$ and call it the "skewness" parameter of the space. We have the following theorem (we include the proof in the full version of the paper):

Theorem 4. *For the non-uniform discretization scheme, the total number of Steiner points added into the triangular decomposition is* $O\left(\frac{C_{skew}\cdot n}{\epsilon}\log\frac{C_{skew}}{\epsilon}\right)$, *where* n *is the number of triangular regions in the decomposition.*

The dependence of C_{skew} on λ implies that the variations of the velocity bound of the robot in different regions will affect the complexity of this approximation algorithm. If the velocity bound is uniform in all regions, less Steiner points are needed to guarantee an ϵ-approximation. On the contrary, if the velocity bound changes drastically in adjacent regions, it will take much more Steiner points to achieve the same approximation bound.

4.3 Discrete Path

The discrete graph G constructed from the discretization described above has $O(\frac{C_{skew}\cdot n}{\epsilon}\log\frac{C_{skew}}{\epsilon})$ points and $O(n(\frac{C_{skew}}{\epsilon}\log\frac{C_{skew}}{\epsilon})^2)$ edges. The time complexity of computing the optimum path in G is $O(\frac{C_{skew}\cdot n}{\epsilon}(\frac{C_{skew}}{\epsilon}\log\frac{C_{skew}}{\epsilon} + \log n)\log\frac{C_{skew}}{\epsilon})$ by Dijkstra's algorithm, or $O(\frac{C_{skew}\cdot n}{\epsilon}(\log\frac{C_{skew}}{\epsilon} + \log n)\log\frac{C_{skew}}{\epsilon})$ by the BUSHWHACK algorithm. In this



(a) Adding Steiner Points on Edge

(b) Discrete Path that Neighbors a Linear Path

Fig. 2.

subsection we analyze how well a discrete path can approximate an optimum path in the original continuous space.

As we have mentioned earlier, an optimum path p_{opt} from a given source point s to a given destination point t in the original continuous space is piecewise linear. It only bends on the boundary of regions. We establish the following theorem that provides a bound on the error of using a discrete path to approximate an optimum path.

Theorem 5. *For any such linear path p, there exists a discrete path p' such that* $\|p'\| \leq (1 + 9\epsilon)\|p\|$.

Here $\|p\|$ denotes the cost of path p. To prove this theorem, we first describe how to construct a discrete path p' that "neighbors" a given linear path p, and then we show that the cost of such a path p' is no more than $(1 + 9\epsilon)\|p\|$ by using several lemmas.

Let $s = a_1, a_2, \cdots, a_{k-1}, a_k = t$ be the points where the linear path p bends, as shown in Figure 2b. We need to construct a discrete path with bending points $s = b_1, b_2, \cdots, b_{k-1}, b_k = t$ such that each b_i is either a Steiner point or a vertex. As path p only bends on the boundary of regions, each bending point a_i is either between two Steiner points on the same edge, or between a vertex and its neighboring Steiner point.

Suppose that a_i is between two Steiner points p_j, p_{j+1} on edge $e = \overline{v_{i_1} v_{i_2}}$. Without loss of generality, suppose that p_j is between p_{j+1} and v_{i_1}. Then we choose b_i to be p_j if p_j is between v_{i_1} and p_e; if otherwise, we choose b_i to be p_{j+1}. If a_i is between vertex v_{i_1} and Steiner point p_1 on edge $e = \overline{v_{i_1} v_{i_2}}$, we choose b_i to be v_{i_1}.

To evaluate $\|p'\|$, we need to estimate the cost of each segment $\overline{b_j b_{j+1}}$. We can compare $\|\overline{b_j b_{j+1}}\|$ against $\|\overline{a_j a_{j+1}}\|$ in various situations by using this lemma:

Lemma 9. *Assuming $\epsilon \leq \frac{1}{3}$, if both b_j and b_{j+1} are Steiner points, then* $\|\overline{b_j b_{j+1}}\| \leq (1 + 3\epsilon)\|\overline{a_j a_{j+1}}\|$. *Also, if one of b_j and b_{j+1} is a vertex v of*

the triangular decomposition, then $\|\overline{b_j b_{j+1}}\| \leq (1+\frac{3}{2}\epsilon)\|\overline{b_j b_{j+1}}\| + \epsilon R_v$. *Further, if both* b_j *and* b_{j+1} *are vertices of the triangular decomposition, then* $\|\overline{b_j b_{j+1}}\| \leq \|\overline{a_j a_{j+1}}\| + \epsilon R_{v'} + \epsilon R_{v''}$, *where* $v' = b_j$ *and* $v'' = b_{j+1}$.

With this lemma, we are ready to prove Theorem 5.

Proof of Theorem 5 Among the bending points $b_1, b_2, \cdots, b_{k-1}, b_k$ of p', let $b_{i_1}, b_{i_2}, \cdots, b_{i_d}$ be vertices of the triangular decomposition, where $1 = i_1 < i_2 < \cdots < i_{d-1} < i_d = k$. Let $b_{i_j} = v_j$ for $1 \leq j \leq d$. We have $\|p'\| = \sum_{j=1}^{k} \|\overline{b_j b_{j+1}}\| \leq (1+3\epsilon)\sum_{j=1}^{k} \|\overline{a_j a_{j+1}}\| + \epsilon \cdot (R_{v_1} + 2\sum_{j=2}^{d-1} R_{v_j} + R_{v_d}) = (1+3\epsilon)\|p\| + \epsilon \cdot (R_{v_1} + 2\sum_{j=2}^{d-1} R_{v_j} + R_{v_d})$. For each j, $1 \leq j < d$, let $C_j = \sum_{i=i_j}^{i_{j+1}-1} \overline{a_j a_{j+1}}$. That is, C_j is the sub-path of p between a_{i_j} and $a_{i_{j+1}}$. Let r be the region incident to b_{i_j}, a_{i_j} and a_{i_j+1} and let r' be the region incident to $b_{i_{j+1}}$, $a_{i_{j+1}-1}$ and $a_{i_{j+1}}$. As $\overline{b_{i_j} a_{i_j}} + C_j + \overline{a_{i_{j+1}} b_{i_{j+1}}}$ is a path from vertex v_j to v_{j+1}, we have $\|\overline{b_{i_j} a_{i_j}}\|_r + \|C_j\| + \|\overline{a_{i_{j+1}} b_{i_{j+1}}}\|_{r'} \geq R_{v_j}$ as well as $\|\overline{b_{i_j} a_{i_j}}\|_r + \|C_j\| + \|\overline{a_{i_{j+1}} b_{i_{j+1}}}\|_{r'} \geq R_{v_{j+1}}$. Here $\|p\|_r$ is the cost of path p measured by the cost function of r. As $\|\overline{b_{i_j} a_{i_j}}\|_r \leq \epsilon R_{v_j}$ and $\|\overline{a_{i_{j+1}} b_{i_{j+1}}}\|_{r'} \leq \epsilon R_{v_{j+1}}$, we can get $\|C_j\| \geq (1-2\epsilon)(R_{v_j} + R_{v_{j+1}})/2$ and therefore $R_{v_1} + 2\sum_{j=2}^{d-1} R_{v_j} + R_{v_d} = \sum_{j=1}^{d-1}(R_{v_j} + R_{v_{j+1}}) \leq \sum_{j=1}^{d-1} \frac{2}{1-2\epsilon} C_j = \frac{2}{1-2\epsilon}\|p\| \leq 6\|p\|$. Hence $\|p'\| \leq (1+3\epsilon)\|p\| + 6\epsilon\|p\| = (1+9\epsilon)\|p\|$. □

Combining Theorem 4 and 5 we have the second main result of this paper:

Theorem 6. *An* ϵ*-short approximation of the optimum path in regions with flows can be computed in* $O(\frac{C_{skew} \cdot n}{\epsilon}(\log \frac{C_{skew}}{\epsilon} + \log n) \log \frac{C_{skew}}{\epsilon})$ *time, where* n *is the number of regions.*

5 Conclusion

While we have provided the first decision procedure for the two dimension flow path problem, the complexity of our algorithm appears far too high to have practical use. There is no known lower bound for this two dimensional version of the flow path problem. Furthermore, there is no decision algorithm for the three dimensional version of the flow path problem. It remains an open problem to determine a more exact complexity bound for the two and three dimensional flow problems.

References

1. L. Aleksandrov, M. Lanthier, A. Maheshwari, and J.-R. Sack. An ϵ-approximation algorithm for weighted shortest paths on polyhedral surfaces. *Lecture Notes in Computer Science*, 1432:11–22, 1998.
2. L. Aleksandrov, A. Maheshwari, and J.-R. Sack. Approximation algorithms for geometric shortest path problems. In *Proceedings of the 32nd ACM Symposium on Theory of Computing*, 2000.
3. J. Canny. Some algebraic and geometric computations in PSPACE. In *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing*, 1988.

4. J. Canny and J. Reif. New lower bound techniques for robot motion planning problems. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science*, 1987.
5. G. E. Collins. Quantifier elimination for real closed fields by cylindric algebraic decomposition. In *Proc. Second GI Conference on Automata Theory and Formal Languages*, volume 33 of *Lecture Notes in Computer Science*, 1975.
6. M. Lanthier, A. Maheshwari, and J. Sack. Approximating weighted shortest paths on polyhedral surfaces. In *6th Annual Video Review of Computational Geometry, Proc. 13th ACM Symp. Computational Geometry*, 1997.
7. C. Mata and J. Mitchell. A new algorithm for computing shortest paths in weighted planar subdivisions. In *Proceedings of the 13th ACM International Annual Symposium on Computational Geometry (SCG-97)*, 1997.
8. J. S. B. Mitchell. Geometric shortest paths and network optimization. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1998.
9. J. S. B. Mitchell and C. H. Papadimitriou. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of the ACM*, 38(1):18–73, 1991.
10. N. Papadakis and A. Perakis. Minimal time vessel routing in a time-dependent environment. *Transportation Science*, 23(4):266–276, 1989.
11. N. Papadakis and A. Perakis. Deterministic minimal time vessel routing. *Operations Research*, 38(3):426–438, 1990.
12. J. Reif. Complexity of the mover's problem and generalizations. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, 1979.
13. J. Reif and M. Sharir. Motion planning in the presence of moving obstacles. In *26th Annual Symposium on Foundations of Computer Science*, 1985.
14. J. H. Reif and J. A. Storer. A single-exponential upper bound for finding shortest paths in three dimensions. *Journal of the ACM*, 41(5):1013–1019, Sept. 1994.
15. J. Reif and Z. Sun. BUSHWHACK: An Approximation Algorithm for Minimal Paths Through Pseudo-Euclidean Spaces. Submitted for publication.
16. J. Reif and Z. Sun. An efficient approximation algorithm for weighted region shortest path problem. In *Proceedings of the 4th Workshop on Algorithmic Foundations of Robotics*, 2000.
17. J. T. Schwartz and M. Sharir. On the piano movers problem: II. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied mathematics*, 4:298–351, 1983.
18. J. Sellen. Direction weighted shortest path plannin. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA-95)*, 1995.
19. G. Wilfong. Motion planning in the presence of movable obstacles. In *Proceedings of the Fourth ACM Annual Symposium on Computational Geometry*, 1988.

Small Maximal Independent Sets and Faster Exact Graph Coloring

David Eppstein

Dept. of Information and Computer Science, UC Irvine
Irvine, CA 92697-3425, USA
eppstein@ics.uci.edu
http://www.ics.uci.edu/~eppstein/

Abstract. We show that, for any n-vertex graph G and integer parameter k, there are at most $3^{4k-n}4^{n-3k}$ maximal independent sets $I \subset G$ with $|I| \leq k$, and that all such sets can be listed in time $\mathcal{O}(3^{4k-n}4^{n-3k})$. These bounds are tight when $n/4 \leq k \leq n/3$. As a consequence, we show how to compute the exact chromatic number of a graph in time $\mathcal{O}((4/3 + 3^{4/3}/4)^n) \approx 2.4150^n$, improving a previous $\mathcal{O}((1 + 3^{1/3})^n) \approx 2.4422^n$ algorithm of Lawler (1976).

1 Introduction

One of the earliest works in the area of worst-case analysis of NP-hard problems is a 1976 paper by Lawler [5] on graph coloring. It contains two results: an algorithm for finding a 3-coloring of a graph (if the graph is 3-chromatic) in time $\mathcal{O}(3^{n/3}) \approx 1.4422^n$, and an algorithm for finding the chromatic number of an arbitrary graph in time $\mathcal{O}((1+3^{1/3})^n) \approx 2.4422^n$. Since then, the area has grown, and there has been a sequence of papers improving Lawler's 3-coloring algorithm [1,2,4,7], with the most recent algorithm taking time $\approx 1.3289^n$. However, there has been no improvement to Lawler's chromatic number algorithm.

Lawler's algorithm follows a simple dynamic programming approach, in which we compute the chromatic number not just of G but of all its induced subgraphs. For each subgraph S, the chromatic number is found by listing all maximal independent subsets $I \subset S$, adding one to the chromatic number of $S \setminus I$, and taking the minimum of these values. The $\mathcal{O}((1+3^{1/3})^n)$ running time of this technique follows from an upper bound of $3^{n/3}$ on the number of maximal independent sets in any n-vertex graph, due to Moon and Moser [6]. This bound is tight in graphs formed by a disjoint union of triangles.

In this paper, we provide the first improvement to Lawler's algorithm, using the following ideas. First, instead of removing a maximal independent set from each induced subgraph S, and computing the chromatic number of S from that of the resulting subset, we add a maximal independent set of $G \setminus S$ and compute the chromatic number of the resulting superset from that of S. This reversal does not itself affect the running time of the dynamic programming algorithm, but it allows us to constrain the size of the maximal independent sets we consider

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 462–470, 2001.

to at most $|S|/3$. We show that, with such a constraint, we can improve the Moon-Moser bound: for any n-vertex graph G and integer parameter k, there are at most $3^{4k-n}4^{n-3k}$ maximal independent sets $I \subset G$ with $|I| \leq k$. This bound then leads to a corresponding improvement in the running time of our chromatic number algorithm.

2 Preliminaries

We assume as given a graph G with vertex set $V(G)$ and edge set $E(G)$. We let $n = |V(G)|$ and $m = |E(G)|$. A *proper coloring* of G is an assignment of colors to vertices such that no two endpoints of any edge share the same color. We denote the chromatic number of G (the minimum number of colors in any proper coloring) by $\chi(G)$.

If $V(G) = \{v_0, v_1, \ldots v_{n-1}\}$, then we can place subsets $S \subseteq V(G)$ in one-to-one correspondence with the integers $0, 1, \ldots 2^n - 1$:

$$S \leftrightarrow \sum_{v_i \in S} 2^i.$$

Subsets of vertices also correspond to induced subgraphs of G, in which we include all edges between vertices in the subset. We make no distinction between these three equivalent views of a vertex subset, so e.g. we will write $\chi(S)$ to indicate the chromatic number of the subgraph induced by set S, and $X[S]$ to indicate a reference to an array element indexed by the number $\sum_{v_i \in S} 2^i$. We write $S < T$ to indicate the usual arithmetic comparison between two numbers, and $S \subset T$ to indicate the usual (proper) subset relation between two sets. Note that, if $S \subset T$, then also $S < T$, although the reverse implication does not hold.

A set S is a *maximal k-chromatic subset* of T if $S \subseteq T$, $\chi(S) = k$, and $\chi(S') > k$ for every $S \subset S' \subseteq T$. In particular, if $k = 1$, S is a *maximal independent subset* of T.

For any vertex $v \in V(G)$, we let $N(v)$ denote the set of neighbors of v, including v itself. If S and T are sets, $S \setminus T$ denotes the set-theoretic difference, consisting of elements of S that are not also in T. K_i denotes the complete graph on i vertices. We write $\deg(v, S)$ to denote the degree of vertex v in the subgraph induced by S.

We express our pseudocode in a syntax similar to that of C, C++, or Java. In particular this implies that array indexing is zero-based. We assume the usual RAM model of computation, in which a single cell is capable of storing an integer large enough to index the memory requirements of the program (thus, in our case, n-bit values are machine integers), and in which arithmetic and array indexing operations on these values are assumed to take constant time.

3 Small Maximal Independent Sets

Theorem 1. *Let G be an n-vertex graph, and k be a nonnegative number. Then the number of maximal independent sets $I \subset V(G)$ for which $|I| \leq k$ is at most $3^{4k-n}4^{n-3k}$.*

Proof. We use induction on n; in the base case $n = 0$, there is one (empty) maximal independent set, and for any $k \geq 0$, $1 \leq 3^{4k}4^{-3k} = (81/64)^k$. Otherwise, we divide into cases according to the degrees of the vertices in G, as follows:

- If G contains a vertex v of degree three or more, then each maximal independent set I either contains v (in which case $I \setminus \{v\}$ is a maximal independent set of $G \setminus N(v)$) or it does not contain v (in which case I itself is a maximal independent set of $G \setminus \{v\}$). Thus, by induction, the number of maximal independent sets of cardinality at most k is at most

$$3^{4k-(n-1)}4^{(n-1)-3k} + 3^{4(k-1)-(n-4)}4^{(n-4)-3(k-1)}$$
$$= (\frac{3}{4} + \frac{1}{4})3^{4k-n}4^{n-3k} = 3^{4k-n}4^{n-3k}$$

  as was to be proved.
- If G contains a degree-one vertex v, let its neighbor be u. Then each maximal independent set contains exactly one of u or v, and removing this vertex from the set produces a maximal independent set of either $G \setminus N(u)$ or $G \setminus N(v)$. If the degree of u is d, this gives us by induction a bound of

$$3^{4(k-1)-(n-2)}4^{(n-2)-3(k-1)} + 3^{4(k-1)-(n-d-1)}4^{(n-d-1)-3(k-1)}$$
$$\leq \frac{8}{9}\,3^{4k-n}4^{n-3k}$$

  on the number of maximal independent sets of cardinality at most k.
- If G contains an isolated vertex v, then each maximal independent set contains v, and the number of maximal independent sets of cardinality at most k is at most

$$3^{4(k-1)-(n-1)}4^{(n-1)-3(k-1)} = \frac{16}{27}\,3^{4k-n}4^{n-3k}.$$

- If G contains a chain u-v-w-x of degree two vertices, then each maximal independent set contains u, contains v, or does not contain u and contains w. Thus in this case the number of maximal independent sets of cardinality at most k is at most

$$2{\cdot}3^{4(k-1)-(n-3)}4^{(n-3)-3(k-1)} + 3^{4(k-1)-(n-4)}4^{(n-4)-3(k-1)} = \frac{11}{12}\,3^{4k-n}4^{n-3k}.$$

- In the remaining case, G consists of a disjoint union of triangles, all maximal independent sets have exactly $n/3$ vertices, and there are exactly $3^{n/3}$ maximal independent sets. If $k \geq n/3$, then $3^{n/3} \leq 3^{4k-n}4^{n-3k}$. If $k < n/3$, there are no maximal independent sets of cardinality at most k.

```
// List maximal independent subsets of S smaller than a given parameter.
//      S is a set of vertices forming an induced subgraph in G,
//      I is a set of vertices to be included in the MIS (initially zero), and
//      k bounds the number of vertices of S to add to I.
// We call processMIS(I) on each generated set. Some non-maximal sets may be
// generated along with the maximal ones, but all generated sets are independent.

void smallMIS (set S, set I, int k)
{
    if (S = 0 or k = 0) processMIS(I);
    else if (there exists v ∈ S with deg(v, S) ≥ 3)
    {
        smallMIS (S \ {v}, I, k);
        smallMIS (S \ N(v), I ∪ {v}, k − 1);
    }
    else if (there exists v ∈ S with deg(v, S) = 1)
    {
        let u be the neighbor of v;
        smallMIS (S \ N(u), I ∪ {u}, k − 1);
        smallMIS (S \ N(v), I ∪ {v}, k − 1);
    }
    else if (there exists v ∈ S with deg(v, S) = 0)
        smallMIS (S \ {v}, I ∪ {v}, k − 1);
    else if (some cycle in S is not a triangle or k ≥ |S|/3)
    {
        let u, v, and w be adjacent degree-two vertices, such that (if possible) u and w are nonadjacent;
        smallMIS (S \ N(u), I ∪ {u}, k − 1);
        smallMIS (S \ N(v), I ∪ {v}, k − 1);
        smallMIS (S \ ({u} ∪ N(w)), I ∪ {w}, k − 1);
    }
}
```

Fig. 1. Algorithm for listing all small maximal independent sets.

Thus in all cases the number of maximal independent sets is within the claimed bound. □

Croitoru [3] proved a similar bound with the stronger assumption that all maximal independent sets have $|I| \leq k$. When $n/4 \leq k \leq n/3$, our result is tight, as can be seen for a graph formed by the disjoint union of $4k - n$ triangles and $n - 3k$ K_4's.

Theorem 2. *There is an algorithm for listing all maximal independent sets smaller than k in an n-vertex graph G, in time $\mathcal{O}(3^{4k-n}4^{n-3k})$.*

Proof. We use a recursive backtracking search, following the case analysis of Theorem 1: if there is a high-degree vertex, we try including it or not including it; if there is a degree-one vertex, we try including it or its neighbor; if there is a degree-zero vertex, we include it; and if all vertices form chains of degree-two vertices, we test whether the parameter k allows any small maximal independent sets, and if so we try including each of a chain of three adjacent vertices. The same case analysis shows that this algorithm performs $\mathcal{O}(3^{4k-n}4^{n-3k})$ recursive calls.

```
int chromaticNumber (graph G)
{
    int X[2^n];
    for (S = 0; S ≤ 2^n; S++)
    {
        if (χ(S) ≤ 3) X[S] = χ[S];
        else X[S] = ∞;
    }
    for (S = 0; S ≤ 2^n; S++)
    {
        if (3 ≤ X[S] < ∞)
        {
            for (each maximal independent set I of G \ S with |I| ≤ |S| / X[S])
                X[S ∪ I] = min(X[S ∪ I], X[S] + 1);
        }
    }
    return X[V(G)];
}
```

Fig. 2. Algorithm for computing the chromatic number of a graph.

Each recursive call can easily be implemented in time polynomial in the size of the graph passed to the recursive call. Since our $3^{4k-n}4^{n-3k}$ bound is exponential in n, even when $k = 0$, this polynomial overhead at the higher levels of the recursion is swamped by the time spent at lower levels of the recursion, and does not appear in our overall time bound. □

A more detailed pseudocode description of the algorithm is shown in Figure 1. The given pseudocode may generate non-maximal as well as maximal independent sets, because (when we try not including a high degree vertex) we do not make sure that a neighbor is later included. This will not cause problems for our chromatic number algorithm, but if only maximal independent sets are desired one can easily test the generated sets and eliminate the non-maximal ones. The pseudocode also omits the data structures necessary to implement each recursive call in time polynomial in $|S|$ instead of polynomial in the number of vertices of the original graph.

4 Chromatic Number

We are now ready to describe our algorithm for computing the chromatic number of graph G. We use an array X, indexed by the 2^n subsets of G, which will (eventually) hold the chromatic numbers of certain of the subsets including $V(G)$ itself. We initialize this array by testing, for each subset S, whether $\chi(S) \leq 3$; if so, we set $X[S]$ to $\chi(S)$, but otherwise we set $X[S]$ to ∞.

Next, we loop through the subsets S of $V(G)$, in numerical order (or any other order such that all proper subsets of each set S are visited before we visit

S itself). When we visit S, we first test whether $X[S] \geq 3$. If not, we skip over S without doing anything. But if $X[S] \geq 3$, we loop through the small independent sets of $G \setminus S$, limiting the size of each such set to $|S|/X[S]$, using the algorithm of the previous section. For each independent set I, we set $X[S \cup I]$ to the minimum of its previous value and $X[S] + 1$.

Finally, after looping through all subsets, we return the value in $X[V(G)]$ as the chromatic number of G. Pseudocode for this algorithm is shown in Figure 2.

Lemma 1. *Throughout the course of the algorithm, for any set S, $X[S] \geq \chi(S)$.*

Proof. Clearly this is true of the initial values of X. Then for any S and any independent set I, we can color $S \cup I$ by using a coloring of S and another color for each vertex in I, so $\chi(S \cup I) \leq \chi(S) + 1 \leq X[S] + 1$, and each step of our algorithm preserves the invariant. □

Lemma 2. *Let M be a maximal $k+1$-chromatic subset of G, and let (S, I) be a partition of M into a k-chromatic subset S and an independent subset I, maximizing the cardinality of S among all such partitions. Then I is a maximal independent subset of $G \setminus S$ with $|I| \leq |S|/k$, and S is a maximal k-chromatic subset of G.*

Proof. If we have any $(k+1)$-coloring of G, then the partition formed by separating the largest k color classes from the smallest color class satisfies the inequality $|I| \leq |S|/k$, so clearly this also is true when (S, I) is the partition maximizing $|S|$. If I were not maximal, due to the existence of another independent set $I \subset I' \subset G \setminus S$, then $S \cup I'$ would be a larger $(k+1)$-chromatic graph, violating the assumption of maximality of M.

Similarly, suppose there were another k-chromatic set $S \subset S' \subset G$. Then if $S' \cap I$ were empty, $S' \cup I$ would be a $(k+1)$-chromatic superset of M, violating the assumption of M's maximality. But if $S' \cap I$ were nonempty, $(S', I \setminus S')$ would be a better partition than (S, I), so in either case we get a contradiction. □

Lemma 3. *Let M be a maximal $k+1$-chromatic subset of G. Then, when the outer loop of our algorithm reaches M, it will be the case that $X[M] = \chi(M)$.*

Proof. Clearly, the initialization phase of the algorithm causes this to be true when $\chi(M) \leq 3$. Otherwise, let (S, I) be as in Lemma 2. By induction on $|M|$, $X[S] = \chi(S)$ at the time we visit S. Then $X[S] \geq 3$, and $|I| \leq |S|/X[S]$, so the inner loop for S will visit I and set $X[M]$ to $X[S] + 1 = \chi(M)$. □

Theorem 3. *We can compute the chromatic number of a graph G in time $\mathcal{O}((4/3 + 3^{4/3}/4)^n)$ and space $\mathcal{O}(2^n)$.*

Proof. $V(G)$ is itself a maximal $\chi(G)$-chromatic subset of G, so Lemma 3 shows that the algorithm correctly computes $\chi(G) = X[V(G)]$. Clearly, the space is bounded by $\mathcal{O}(2^n)$. It remains to analyze the algorithm's time complexity.

```
void color (graph G)
{
    compute array X as in Figure 2;
    S = V(G);
    for (T = 2^n − 1; T ≥ 0; T−−)
    {
        if (T ⊂ S and X[S \ T] = 1 and X[T] = X[S] − 1)
        {
            color all vertices in S \ T with the same new color;
            S = T;
        }
    }
}
```

Fig. 3. Algorithm for optimally coloring a graph.

First, we consider the time spent initializing X. Since we perform a 3-coloring algorithm on each subset of G, this time is

$$\sum_{S\subset V(G)} \mathcal{O}(1.3289^{|S|}) = \mathcal{O}\Big(\sum_{i=0}^{n} \binom{n}{i} 1.3289^i\Big) = \mathcal{O}(2.3289^i).$$

Finally, we bound the time in the main loop of the algorithm. We may possibly apply the algorithm of Theorem 2 to generate small independent subsets of each set $G \setminus S$. In the worst case, $X[S] = 3$ and we can only limit the size of the generated independent sets to $|S|/3$. We spend constant time adjusting the value of $X[S \cup I]$ for each generated set. Thus, the total time can be bounded as

$$\sum_{S\subset V(G)} \mathcal{O}(3^{4^{\frac{|S|}{3}} - |G\setminus S|} 4^{|G\setminus S| - 3^{\frac{|S|}{3}}}) = \mathcal{O}\Big(\sum_{i=0}^{n} \binom{n}{i} 3^{\frac{7i}{3}-n} 4^{n-2i}\Big) = \mathcal{O}\Big((\frac{4}{3}+\frac{3^{4/3}}{4})^n\Big).$$

This final term dominates the overall time bound. □

5 Finding a Coloring

Although the algorithm of the previous section finds the chromatic number of G, it is likely that an explicit coloring is desired, rather than just this number. The usual method of performing this sort of construction task in a dynamic programming algorithm is to augment the dynamic programming array with back pointers indicating the origin of each value computed in the array, but since storing 2^n chromatic numbers is likely to be the limiting factor in determining how large a graph this algorithm can be applied to, it is likely that also storing 2^n set indices will severely reduce its applicability.

Instead, we can simply search backwards from $V(G)$ until we find a subset S that can be augmented by an independent set to form $V(G)$, and that has

chromatic number $\chi(S) = \chi(G) - 1$ as indicated by the value of $X[S]$. We assign the first color to $G \setminus S$. Then, we continue searching for a similar subset $T \subset S$, etc., until we reach the empty set. Although not every set S may necessarily have $X[S] = \chi(S)$, it is guaranteed that for any S we can find $T \subset S$ with $S \setminus T$ independent and $X[T] = X[S] - 1$, so this search procedure always finds a correct coloring.

Theorem 4. *After computing the array X as in Theorem 3, we can compute an optimal coloring of G in additional time $\mathcal{O}(2^n)$ and no additional space.*

We omit the details of the correctness proof and analysis. A pseudocode description of the coloring algorithm is shown in Figure 3.

6 Conclusions

We have shown a bound on the number of small independent sets in a graph, shown how to list all small independent sets in time proportional to our bound, and used this algorithm in a new dynamic programming algorithm for computing the chromatic number of a graph as well as an optimal coloring of the graph.

Our bound on the number of small independent sets is tight for $n/4 \leq k \leq n/3$, and an examination of the analysis of Theorem 3 shows that independent set sizes in this range are also the ones leading to the worst case time bound for our coloring algorithm. Nevertheless, it would be of interest to find tight bounds on the number of small independent sets for all ranges of k. It would also be of interest to find an algorithm for listing all small maximal independent sets in time proportional to the number of generated sets rather than simply proportional to the worst case bound on this number.

Our worst case analysis of the chromatic number algorithm assumes that, every time we call the procedure for listing small maximal independent sets, this procedure achieves its worst case time bound. But is it really possible for all sets $G \setminus S$ to be worst case instances for this procedure? If not, perhaps the analysis of our coloring algorithm can be improved.

Can we prove a bound smaller than $\binom{n}{i}$ on the number of i-vertex maximal k-chromatic induced subgraphs of a graph G? If such a bound could be proven, even for $k = 3$, we could likely improve the algorithm presented here by only looping through the independent subgraphs of $G \setminus S$ when S is maximal.

An alternative possibility for improving the present algorithm would be to find an algorithm for testing whether $\chi(G) \leq 4$ in time $o(1.415^n)$. Then we could test the four-colorability of all subsets of G before applying the rest of our algorithm, and avoid looping over maximal independent subsets of $G \setminus S$ unless $X[S] \geq 4$. This would produce tighter limits on the independent set sizes and therefore reduce the number of independent sets examined. However such a time bound would be significantly better than what is currently known for four-coloring algorithms.

References

1. R. Beigel and D. Eppstein. 3-coloring in time $\mathcal{O}(1.3446^n)$: a no-MIS algorithm. *Proc. 36th Symp. Foundations of Computer Science*, pp. 444–453. IEEE, October 1995, ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/1995/TR95-033/index.html.
2. R. Beigel and D. Eppstein. 3-coloring in time $\mathcal{O}(1.3289^n)$. ACM Computing Research Repository, June 2000, cs.DS/0006046.
3. C. Croitoru. On stables in graphs. *Proc. 3rd Coll. Operations Research*, pp. 55–60. Babes-Bolyai Univ., Cluj-Napoca, Romania, 1979.
4. D. Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. *Proc. 12th Symp. Discrete Algorithms*, pp. 329–337. ACM and SIAM, January 2001, cs.DS/0009006.
5. E. L. Lawler. A note on the complexity of the chromatic number problem. *Inf. Proc. Lett.* 5(3):66–67, August 1976.
6. J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.* 3:23–28, 1965.
7. I. Schiermeyer. Deciding 3-colourability in less than $\mathcal{O}(1.415^n)$ steps. *Proc. 19th Int. Worksh. Graph-Theoretic Concepts in Computer Science*, pp. 177–182. Springer-Verlag, Lecture Notes in Comp. Sci. 790, 1994.

On External-Memory Planar Depth First Search

Lars Arge[1,⋆], Ulrich Meyer[2,⋆⋆], Laura Toma[1,⋆⋆⋆], and Norbert Zeh[3,†]

[1] Department of Computer Science, Duke University, Durham, NC 27708, USA
{large,laura}@cs.duke.edu
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
umeyer@mpi-sb.mpg.de
[3] School of Computer Science, Carleton University, Ottawa, Canada
nzeh@scs.carleton.ca

Abstract. Even though a large number of I/O-efficient graph algorithms have been developed, a number of fundamental problems still remain open. For example, no space- and I/O-efficient algorithms are known for depth-first search or breadth-first search in sparse graphs. In this paper we present two new results on I/O-efficient depth-first search in an important class of sparse graphs, namely undirected embedded planar graphs. We develop a new efficient depth-first search algorithm and show how planar depth-first search in general can be reduced to planar breadth-first search. As part of the first result we develop the first I/O-efficient algorithm for finding a simple cycle separator of a biconnected planar graph. Together with other recent reducibility results, the second result provides further evidence that external memory breadth-first search is among the hardest problems on planar graphs.

1 Introduction

External memory graph algorithms have received considerable attention lately because massive graphs arise naturally in many applications. Recent web crawls, for example, produce graphs with on the order of 200 million nodes and 2 billion edges, and recent work in web modeling uses depth-first search, breadth-first search, shortest path computation and connected component computation as primitive routines for investigating the structure of the web [5]. Massive graphs are also often manipulated in Geographic Information Systems (GIS), where many common problems can be formulated as basic graph problems. Yet another example of a massive graph is AT&T's 20TB phone-call data graph [7].

⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879 and CAREER grant EIA–9984099.

⋆⋆ Supported in part by the IST Programme of the EU under contract number IST–1999–14186 (ALCOM–FT). Part of this work was done while visiting Duke University.

⋆⋆⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734 and CAREER grant EIA–9984099.

† Supported in part by NSERC and NCE GEOIDE research grants. Part of this work was done while visiting Duke University.

F. Dehne, J.-R. Sack, and R. Tamassia (Eds.): WADS 2001, LNCS 2125, pp. 471–482, 2001.

When working with such massive graphs the I/O-communication, and not the internal memory computation time, is often the bottleneck. Designing I/O-efficient algorithms can thus lead to considerable runtime improvements.

Breadth-first search (BFS) and depth-first search (DFS) are the two most fundamental graph searching strategies. They are extensively used in many graph algorithms. The reason is that both strategies can be implemented in linear time in internal memory; still they reveal important information about the structure of the input graph. Unfortunately, no I/O-efficient BFS or DFS algorithms are known for arbitrary sparse graphs, while known algorithms perform reasonably well on dense graphs. In this paper we consider an important class of sparse graphs, namely *undirected embedded planar graphs*. This class is restricted enough to hope for more efficient algorithms than for arbitrary sparse graphs. Several such algorithms have indeed been obtained recently [3,15]. We develop an improved DFS algorithm for planar graphs and show how planar DFS can be reduced to planar BFS. Since several other problems on planar graphs have also been shown to be reducible to BFS, this provides further evidence that in external memory BFS is among the hardest problems on planar graphs.

1.1 I/O-Model and Previous Results

We work in the standard two-level I/O model with one (logical) disk [1] (our results work in a D-disk model; but for brevity we only consider one disk in this extended abstract). The model defines the following parameters:

$$\begin{aligned} N &= \text{number of vertices and edges } (N = |V| + |E|), \\ M &= \text{number of vertices/edges that can fit into internal memory}, \\ B &= \text{number of vertices/edges per disk block}, \end{aligned}$$

where $2B < M < N$. An *Input/Output* operation (or simply *I/O*) involves reading (or writing) a block from (to) disk into (from) internal memory. Our measure of performance of an algorithm is the number of I/Os it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \Theta\left(\frac{N}{B}\right)$ (the *linear* or *scanning* bound). The number of I/Os required to sort N items is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ (the *sorting* bound) [1]. For all realistic values of N, B, and M, $\text{scan}(N) < \text{sort}(N) \ll N$. Therefore the difference between the running times of an algorithm performing N I/Os and one performing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os can be very significant [8,4].

I/O-efficient graph algorithms have been considered by a number of authors. For a review see [19] and the references therein. We review the previous results most relevant to our work. The best previously known general DFS algorithms on undirected graphs use $O\big((|V| + (|E|/B)) \log_2 |V|\big)$ I/Os [12] or $O\big(|V| + (|V|/M) \cdot (|E|/B)\big)$ I/Os [8]. Since the best known general BFS algorithm uses only $O(|V| + (|E|/|V|)\text{sort}(|V|)) = O(|V| + \text{sort}(|E|))$ I/Os [17], this suggests that on undirected graphs DFS might be harder than BFS. For directed graphs the best known algorithms for BFS and DFS both

use $O((|V| + |E|/B) \cdot \log(|V|/B) + \text{sort}(|E|))$ I/Os [6]. In general we cannot hope to design algorithms that perform less than $\Omega(\min(|V|, \text{sort}(|V|)))$ I/Os for either of the two problems [2,8,17]. As mentioned above, in practice $O(\min(|V|, \text{sort}(|V|))) = O(\text{sort}(|V|))$. Still, all of the above algorithms use $\Omega(|V|)$ I/Os. For planar graphs this bound is matched by the standard internal memory algorithms.

Recently, the first $o(N)$ DFS and BFS algorithms for undirected planar graphs were developed [15]. These algorithms use $O(\frac{N}{\gamma \log B} + \text{sort}(NB^\gamma))$ I/Os and $O(NB^\gamma)$ space, for any $0 < \gamma \leq 1/2$. Further improved algorithms have been developed for special classes of planar graphs. For trees, $O(\text{sort}(N))$ I/O algorithms are known for both BFS and DFS—as well as for Euler tour computation, expression tree evaluation, topological sorting, and several other problems [6,8,3]. BFS and DFS can also be solved in $O(\text{sort}(N))$ I/Os on outerplanar graphs [13] and on k-outerplanar graphs [14]. Developing $O(\text{sort}(N))$ I/O DFS and BFS algorithms for arbitrary planar graphs is a challenging open problem.

1.2 Our Results

The contribution of this paper is two-fold. In Sec. 2 we present a new DFS algorithm for undirected embedded planar graphs that uses $O(\text{sort}(N) \log N)$ I/Os and linear space. For most practical values of B, M and N this algorithm uses $o(N)$ I/Os and is the first such algorithm using linear space. The algorithm is based on a divide-and-conquer approach first proposed in [18]. It utilizes a new $O(\text{sort}(N))$ I/O algorithm for finding a simple cycle in a biconnected planar graph such that neither the subgraph inside nor the one outside the cycle contains more than a constant fraction of the edges of the graph. Previously, no such algorithm was known.

In Sec. 3 we use ideas similar to the ones utilized in [9] to obtain an $O(\text{sort}(N))$ I/O reduction from DFS to BFS on undirected embedded planar graphs. Contrary to what has been conjectured for general graphs, this shows that for planar graphs BFS is as hard as DFS. A recent paper shows that given a BFS-tree of a planar graph, the single source shortest path problem as well as the multi-way separation problems can be solved in $O(\text{sort}(N))$ I/Os [3]. Together, these results suggest that BFS may indeed be a universally hard problem for planar graphs. That is, if planar BFS can be performed I/O-efficiently, most other problems on planar graphs can also be solved I/O-efficiently.

2 DFS Using Simple Cycle Separators

2.1 Outline of the Algorithm

Our $O(\text{sort}(N) \log N)$ I/O and linear space algorithm for computing a DFS tree of a planar graph is based on a divide-and-conquer approach proposed in [18].

A *cutpoint* of a graph G is a vertex whose removal disconnects G. We first consider the case where G is *biconnected*, i.e., does not contain any cutpoints. In

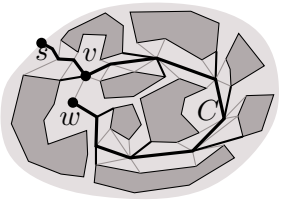


Fig. 1. The path P' is shown in bold. Components are shaded dark grey. Medium edges are edges $\{u_i, v_i\}$. Light edges are non-tree edges.

Sec. 2.2 we show that for a biconnected planar graph G we can compute a *simple cycle α-separator* in $O(\text{sort}(N))$ I/Os (Thm. 2). A simple cycle α-separator C of G is a simple cycle such that neither the subgraph inside nor outside the cycle contains more than $\alpha|E|$ edges. The main idea of our algorithm is to partition G using a simple cycle α-separator, for some constant $0 < \alpha < 1$, recursively compute DFS-trees for the connected components of $G \setminus C$, and combine them to obtain a DFS-tree for G. Given that each recursive step can be realized in $O(\text{sort}(N))$ I/Os, the whole algorithm takes $O(\text{sort}(N) \log N)$ I/Os.

In more detail, we construct a DFS tree T of a biconnected embedded planar graph G, rooted at some vertex s as follows (see Fig. 1): First we compute a simple cycle α-separator C of G in $O(\text{sort}(N))$ I/Os. Then we find a path P from s to some vertex v in C by computing a spanning tree T' of G and finding the closest vertex to s in C along T'. This also takes $O(\text{sort}(N))$ I/Os [8]. Next we extend P to a path P' containing all vertices in P and C. To do so we identify the counterclockwise neighbor $w \in C$ of v, relative to the last edge on P, remove edge $\{v, w\}$ from C, rank the resulting path to obtain the clockwise order of the vertices in C, and finally concatenate P with the resulting path. All these steps can be performed in $O(\text{sort}(N))$ I/Os [8]. We compute the connected components $H_1, \ldots, H_k$ of $G \setminus P'$ in $O(\text{sort}(N))$ I/Os [8]. For each component H_i, we find the vertex $v_i \in P'$ furthest away from s along P' such that there is an edge $\{u_i, v_i\}$, $u_i \in H_i$. This can easily be done in $O(\text{sort}(N))$ I/Os. Next we recursively compute DFS trees $T_1, \ldots, T_k$ for components $H_1, \ldots, H_k$ and obtain a DFS tree T for G as the union of trees $T_1, \ldots, T_k$, path P', and edges $\{u_i, v_i\}$, $1 \leq i \leq k$. Note that components $H_1, \ldots, H_k$ are not necessarily biconnected. Below we show how to deal with this case.

To see that T is indeed a DFS tree, first note that there are no edges between components $H_1, \ldots, H_k$. For every non-tree edge $\{v, w\}$ connecting a vertex v in a component H_i with a vertex w in P', v is a descendant of u_i and, by the choice of v_i, w is an ancestor of v_i. Thus all non-tree edges in G are back-edges, and T is a DFS tree.

We handle the case where G is not biconnected by finding the *biconnected components* or *bicomps* (i.e., the maximal biconnected subgraphs) of G, computing a DFS tree for each bicomp and joining them at the cutpoints. More precisely, we compute the *bicomp-cutpoint-tree* T_G of G containing all cutpoints of G and one vertex $v(C)$ per bicomp C. There is an edge between a cutpoint v

and a bicomp vertex $v(C)$ if v is contained in C. We choose a bicomp C_r containing vertex s as the root of T_G. The *parent cutpoint* of a bicomp C is the parent $p(v(C))$ of $v(C)$ in T_G. The *parent bicomp* of C is the bicomp C' corresponding to $v(C') = p(p(v(C)))$. T_G can be constructed in $O(\text{sort}(N))$ I/Os [8]. We compute a DFS tree of C_r rooted at vertex s. In all other bicomps C, we compute a DFS tree rooted at the parent cutpoint of C. The union of the resulting DFS trees is a DFS tree for G rooted at s, as there are no edges between different bicomps. Thus, we obtain our first main result.

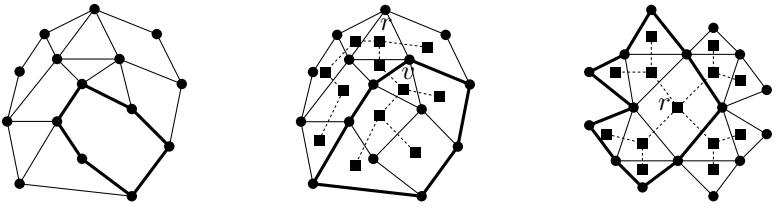
Theorem 1. *A DFS tree of an embedded planar graph can be computed in* $O(\mathit{sort}(N) \log N)$ *I/O operations and linear space.*

2.2 Finding a Simple Cycle Separator

Utilizing ideas similar to the ones used in [11,16] we now show how to compute a simple cycle $\frac{5}{6}$-separator for a planar biconnected graph.

Given an embedded planar graph G, the *faces* of G are the connected regions of $\mathbb{R}^2 \setminus G$. We use F to denote the set of faces of G. The *boundary* of a face f is the set of edges contained in the closure of f. For a set F' of faces of G, let $G_{F'}$ be the subgraph of G defined as the union of the boundaries of the faces in F'. The *complement* $\overline{G}_{F'}$ of $G_{F'}$ is the graph obtained as the union of boundaries of all faces in $F \setminus F'$. The *boundary* of $G_{F'}$ is the intersection between $G_{F'}$ and its complement $\overline{G}_{F'}$. The *dual* G^* of G is the graph containing one vertex f^* per face $f \in F$, and an edge between two vertices f_1^* and f_2^* if faces f_1 and f_2 share an edge. We use v^*, e^*, and f^* to refer to the face, edge, and vertex which is dual to vertex v, edge e, and face f, respectively. The dual G^* of a planar graph G is planar and can be computed in $O(\text{sort}(N))$ I/Os [10].

The main idea in our algorithm is to find a set of faces $F' \subset F$ such that the boundary of $G_{F'}$ is a simple cycle $\frac{5}{6}$-separator. The main difficulty is to ensure that the boundary of $G_{F'}$ is a simple cycle. We compute F' as follows: First we check whether there is a single face whose boundary has size at least $\frac{|E|}{6}$ (Fig. 2a). If we find such a face, we report its boundary as the separator C. Otherwise, we compute a spanning tree T^* of the dual G^* of G rooted at an arbitrary node r. Every node $v \in T^*$ defines a maximal subtree $T^*(v)$ of T^* rooted at v. The nodes in this subtree correspond to a set of faces in G whose boundaries define a graph $G(v)$. Below we show that the boundary of $G(v)$ is a simple cycle in G. We try to find a node v such that $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$, where $|G(v)|$ is the number of edges in $G(v)$ (Fig. 2b). If we succeed, we report the boundary of $G(v)$. Otherwise, we are left in a situation where for every leaf $l \in T^*$ (face in G^*) we have $|G(l)| < \frac{1}{6}|E|$, for the root r of T^* we have $|G(r)| = |E|$, and for every other vertex $v \in T^*$ either $|G(v)| < \frac{1}{6}|E|$ or $|G(v)| > \frac{5}{6}|E|$. Thus, there has to be a node v with $|G(v)| > \frac{5}{6}|E|$ and $|G(w_i)| < \frac{1}{6}|E|$, for all children $w_1, \ldots, w_k$ of v. We show how to compute a subgraph G' of $G(v)$ consisting of the boundary of the face v^* and a subset of the graphs $G(w_1), \ldots, G(w_k)$ such that $\frac{1}{6}|E| \leq |G'| \leq \frac{5}{6}|E|$, and the boundary of G' is a simple cycle (Fig. 2c). Below we describe our algorithm in detail and show that all of the above steps can be performed in $O(\text{sort}(N))$ I/Os. This proves the following theorem.



(a) (b) (c)

Fig. 2. (a) A heavy face. (b) A heavy subtree. (c) Splitting a heavy subtree.

Theorem 2. *A simple cycle $\frac{5}{6}$-separator of an embedded biconnected planar graph can be computed in $O(sort(N))$ I/O operations and linear space.*

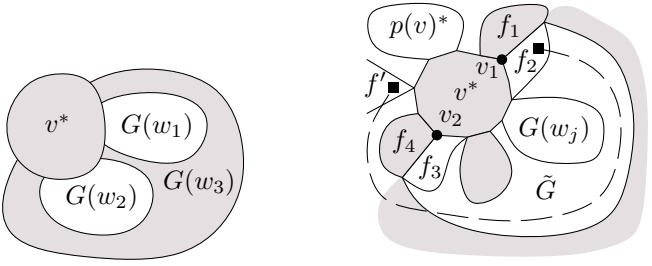
Checking for heavy faces. In order to check if there exists a face f in G with a boundary of size at least $\frac{1}{6}|E|$, we represent each face of G as a list of vertices along its boundary. Computing such a representation takes $O(\text{sort}(N))$ I/Os [10]. Then we scan these lists to see whether any of them has length at least $\frac{1}{6}|E|$.

Checking for heavy subtrees. First we prove that the boundary of $G(v)$ defined by the nodes in $T^*(v)$ is a simple cycle. A planar graph is *uniform* if its dual is connected. Since for every $v \in T^*$, $T^*(v)$ and $T^* \setminus T^*(v)$ are both connected, $G(v)$ and its complement $\overline{G(v)}$ are both uniform. Using the following lemma, this implies that the boundary of $G(v)$ is a simple cycle.

Lemma 1 (Smith [18]). *Let G' be a subgraph of a biconnected planar graph G. The boundary of G' is a simple cycle if and only if G' and its complement are both uniform.*

Next we show how to find a node $v \in T^*$ such that $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$. G^* and T^* can both be computed in $O(\text{sort}(N))$ I/Os [10,8]. For every node $v \in T^*$, let $|v^*|$ be the number of edges on the boundary of face v^*. Let the *weight* $\omega(G(v))$ of subgraph $G(v)$ be defined as $\omega(G(v)) = \sum_{w \in T^*(v)} |w^*|$. As $\omega(G(v)) = |v^*| + \sum_{i=1}^{k} \omega(G(w_i))$, where $w_1, \ldots, w_k$ are the children of v in T^*, we can process T^* bottom-up to compute the weights of all subgraphs $G(v)$. Using time-forward processing [8], this takes $O(\text{sort}(N))$ I/Os. For a node v in T^* every boundary edge of $G(v)$ is counted once in $\omega(G(v))$; every interior edge is counted twice. This implies that if $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$, then $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$. Thus, we can find a node $v \in T^*$ with $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$ in $O(\text{scan}(N))$ I/Os by scanning through the list of nodes T^* and finding a node v such that $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$, if such a node exists.[1]

[1] Note that even if a node v with $\frac{1}{6}|E| \leq |G(v)| \leq \frac{5}{6}|E|$ exists in T^*, the algorithm might not find it since it does not follow that $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$. This is not



(a) (b)

Fig. 3. (a) The boundary of $v^* \cup G(w_3)$ is not a simple cycle. (b) Grey regions are in $H_\sigma(i)$.

Splitting a heavy subtree. We are now in a situation where no vertex $v \in T^*$ satisfies $\frac{1}{3}|E| \leq \omega(G(v)) \leq \frac{2}{3}|E|$. Thus, there must be a vertex $v \in T^*$ with children $w_1, \ldots, w_k$ such that $\omega(G(v)) > \frac{2}{3}|E|$ and $\omega(G(w_i)) < \frac{1}{3}|E|$, for $1 \leq i \leq k$. Our goal is to compute a subgraph of $G(v)$ consisting of the boundary v^* and a subset of the graphs $G(w_i)$ whose weight is between $\frac{1}{3}|E|$ and $\frac{2}{3}|E|$ and whose boundary is a simple cycle C.

In [11] it is claimed that the boundary of the graph defined by v^* and any subset of graphs $G(w_i)$ is a simple cycle. Unfortunately this is not true in general, as illustrated in Fig. 3(a). However, as we show below, we can compute a permutation $\sigma : \{1 \ldots k\} \rightarrow \{1 \ldots k\}$ such that if we start with v^* and incrementally "glue" $G(w_{\sigma(1)})$, $G(w_{\sigma(2)})$, $\ldots$, $G(w_{\sigma(k)})$ onto face v^*, the boundary of each of the obtained graphs is a simple cycle. More formally, we show that if we define $H_\sigma(i) = v^* \cup \bigcup_{j=1}^{i} G(w_{\sigma(j)})$ then $H_\sigma(i)$ and $\overline{H_\sigma(i)}$ are both uniform for all $1 \leq i \leq k$. This implies that the boundary of $H_\sigma(i)$ is a simple cycle by Lemma 1. Since we have already computed the sizes $|v^*|$ of faces v^* and the weights $\omega(G(v))$ of all graphs $G(v)$, it takes $O(\text{scan}(N))$ I/Os to compute weights $\omega(H_\sigma(i))$, $1 \leq i \leq k$, and find index i such that $\frac{1}{3}|E| \leq \omega(H_\sigma(i)) \leq \frac{2}{3}|E|$. It remains to show how to compute the permutation σ I/O-efficiently.

To construct σ, we extract $G(v)$ from G, label v^* with 0, and label every face in $G(w_i)$ with i. Next we label every edge in $G(v)$ with the labels of the two faces on each side of it. We perform the labeling in $O(\text{sort}(N))$ I/Os using the previously computed representations of G and G^* and a post-order traversal of T^*. Details will appear in the full paper. Now consider the vertices $v_1, \ldots, v_t$ on the boundary of v^* in the order they appear clockwise around v^*, starting at the common endpoint of an edge shared by v^* and the face corresponding to v's parent $p(v)$ in T^*. As in Sec. 2, we can compute this order in $O(\text{sort}(N))$ I/Os using list ranking. For each v_i we construct a list L_i of edges around v_i in

a problem, however, since in this case a simple cycle $\frac{5}{6}$-separator will still be found in the final phase or our algorithm. In the full paper we discuss how to modify the algorithm in order to compute $|G(v)|$ exactly for each node $v \in T^*$. This allows us to find even a simple cycle $\frac{2}{3}$-separator.

clockwise order, starting with edge $\{v_{i-1}, v_i\}$ and ending with edge $\{v_i, v_{i+1}\}$. These lists are easily computed in $O(\text{sort}(N))$ I/Os from the embedding of G. Let L be the concatenation of lists $L_1, L_2, \ldots, L_t$. For an edge e in L incident to a vertex v_i, let f_1 and f_2 be the two faces incident to e, where f_1 precedes f_2 in clockwise order around v_i. We construct a list F of face labels from L in $O(\text{scan}(N))$ I/Os by considering the edges in L in order and appending the labels of f_1 and f_2 in this order to F. List F consists of integers between 1 and k. Some integers may appear more than once, and the occurrences of some integer i are not necessarily consecutive. (This happens if the union of v^* with a subgraph $G(w_i)$ encloses another subgraph $G(w_j)$; see Fig. 3(a).) We construct a final list S by removing all but the last occurrence of each integer from F (Intuitively, this ensures that if the union of v^* and $G(w_i)$ encloses another subgraph $G(w_j)$, then j appears before i in S). This takes $O(\text{sort}(N))$ I/Os by sorting and scanning F twice. Again details will appear in the full paper. S contains each of the integers 1 through k exactly once and thus defines a permutation σ. All that remains is to prove the following lemma.

Lemma 2. *For all $1 \leq i \leq k$, $H_\sigma(i)$ and $\overline{H_\sigma(i)}$ are both uniform.*

Proof. Every graph $H_\sigma(i)$ is uniform because every subgraph $G(w_j)$ is uniform and w_j is connected to v by an edge in G^*. Next we show that every $\overline{H_\sigma(i)}$ is uniform. To do this we must show that every $\overline{H_\sigma(i)}^*$ is connected. Note that $\overline{G(v)} \subseteq \overline{H_\sigma(i)}$, $\overline{G(v)}$ is uniform, and each graph $G(w_j)$ is uniform. Hence, in order to prove that $\overline{H_\sigma(i)}^*$ is connected, it suffices to show that for all $i < j \leq k$, there is a path in $\overline{H_\sigma(i)}^*$ connecting a vertex in $G(w_j)^*$ to a vertex in $\overline{G(v)}^*$. So assume for the sake of contradiction that there is a graph $G(w_j)$, $i < j \leq k$, such that there is no such path from a vertex in $G(w_j)^*$ to a vertex in $\overline{G(v)}^*$ in $\overline{H_\sigma(i)}^*$ (Fig. 3(b)). Let $\tilde{G}$ be the uniform component of $\overline{H_\sigma(i)}$ containing $G(w_j)$, and C be the boundary cycle of $\tilde{G}$. Let P be the path obtained by removing the edges shared by v^* and $p(v)^*$ from the boundary cycle of v^*. Let v_1 be the first vertex of C encountered during a clockwise walk along P; let v_2 be the last such vertex. We define P' to be the path obtained by walking clockwise around $\tilde{G}$ starting at v_1 and ending at v_2. Let e_1 be the first and e_2 be the last edge on P'. Edge e_1 separates two faces $f_1 \in H_\sigma(i)$ and $f_2 \in \overline{H_\sigma(i)}$. Similarly, edge e_2 separates two faces $f_3 \in \overline{H_\sigma(i)}$ and $f_4 \in H_\sigma(i)$. Let j_1, j_2, j_3 and j_4 be the labels of these faces. We show that label j_2 appears before label j_4 in S: Assume that label j_2 appears after label j_4 in S. Then there has to be a face f' with label j_2 occurring after face f_4 clockwise around v^*. In particular, face f' is outside cycle C, while face f_2 is inside. As $G^*(w_{j_2})$ is connected there has to be a path from f'^* to f_2^* in $G^*(w_{j_2})$. But this is not possible since every path from f_2^* to f'^* must contain an edge e^*, for some edge $e \in C$, and edge e^* cannot be in $G^*(w_{j_2})$ because one of its endpoints is in $H_\sigma^*(i)$. Therefore it follows that label j_2 appears before label j_4 in S. But this means means that f_2 is being added to $H_\sigma(i)$ before f_4, contradicting the assumption that $f_2 \in \overline{H_\sigma(i)}$ and $f_4 \in H_\sigma(i)$. □

(a) (b)

Fig. 4. (a) A graph G with its faces colored according to their levels; level 0 white, level 1 light grey, level 2 dark grey. (b) H_0 (solid), H_1 (dotted), H_2 (dashed).

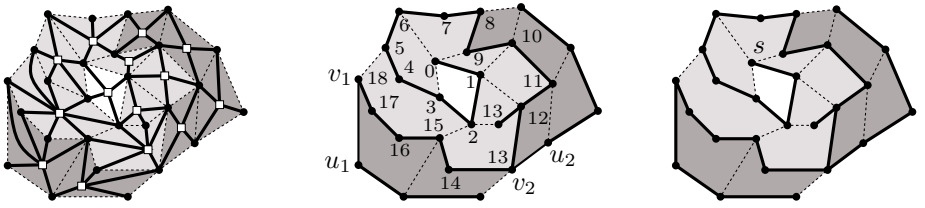
3 Reducing DFS to BFS

This section gives an I/O-efficient reduction from DFS in an embedded planar graph G to BFS in its *vertex-on-face graph*, using ideas from [9]. The vertex-on-face graph $G^\dagger$ of G is defined as follows: The vertex set of $G^\dagger$ is $V \cup V^*$; there is an edge (v, f^*) in $G^\dagger$ if v is on the boundary of face f. The graph $G^\dagger$ can be computed from G in $O(\text{sort}(N))$ I/Os in a way similar to the computation of the dual G^* of G. We use the vertex-on-face graph instead of the graph used in [9], because the vertex-on-face graph of an embedded planar graph G is planar. This could be important in case planar BFS turns out to be easier than general BFS.

The basic idea in our algorithm is to partition the faces of G into levels around a source face with the source s of the DFS tree on its boundary, and then grow a DFS tree level-by-level; Let the source face be at level 0. We partition the remaining faces of G into levels so that all faces at level 1 share a vertex with the level-0 face, all faces at level 2 share a vertex with some level-1 face but not with the level-0 face, and so on (Fig. 4a). Let G_i be the subgraph of G defined by the union of the boundaries of faces at level at most i, and let $H_i = G_i \setminus G_{i-1}$ (Fig. 4b). We call the vertices of H_i *level-i vertices*. To grow the DFS tree we start by walking clockwise[2] around the level-0 face G_0 until we reach the counterclockwise neighbor of s on G_0. The resulting path is a DFS tree T_0 for G_0. Next we build a DFS tree for H_1 and attach it to T_0 in a way that does not introduce cross-edges, thereby obtaining a DFS tree T_1 for G_1. We repeat this process until we have processed all layers H_i. The key to the efficiency of the algorithm lies in the simple structure of the graphs H_i. Below we give the details of our algorithm and prove the following theorem.

Theorem 3. *Let G be an undirected embedded planar graph, $G^\dagger$ its vertex-on-face graph, and f a face of G^* containing the source vertex s. Given a BFS tree of $G^\dagger$ rooted at f^*, a DFS tree of G rooted at s can be computed in $O(sort(N))$ I/Os.*

[2] A *clockwise* walk on the boundary of a face means walking so that the face is to our right.



(a) (b) (c)

Fig. 5. (a) $G^\dagger$ shown in bold. (b) T_1, H_2 and attachment edges $\{u_i, v_i\}$.Vertices in T_1 are labeled with their DFS-depths. (c) The DFS tree.

Corollary 1. *If there is an algorithm that computes a BFS tree of a planar graph in $\mathcal{I}(N)$ I/Os using $S(N)$ space, then DFS on planar graphs takes $O(\mathcal{I}(N))$ I/Os and $O(S(N))$ space.*

First consider the computation of graphs G_i and H_i. The level of all faces can be obtained from a BFS tree for the vertex-on-face graph $G^\dagger$ rooted at a face containing s (Fig. 5(a)). Every vertex of G is at an odd level in the BFS tree; every dual vertex corresponding to a face of G is at an even level. The *level of a face* is the level of the corresponding vertex in the BFS tree divided by two. Given the levels of all faces, the graphs G_i and H_i can be computed in $O(\text{sort}(N))$ I/Os using standard techniques similar to the ones used in computing G^* from G.

Now assume that we have computed a DFS tree T_{i-1} for G_{i-1}. Our goal is to compute a DFS forest for H_i and link it to T_{i-1} without introducing cross-edges. If we can do so in $O(\text{sort}(|H_i|))$ I/Os we obtain an $O(\text{sort}(N))$ I/O reduction from planar DFS to planar BFS. Note that the entire graph H_i lies "outside" the boundary of G_{i-1}, i.e., in $\overline{G_{i-1}}$. The boundary of G_{i-1} is in H_{i-1} and consists of cycles, called the *boundary cycles* of G_{i-1}. The graph G_{i-1} is uniform; but $\overline{G_{i-1}}$ may not be uniform. Graph H_i may consist of several connected components. The following lemma shows that H_i has a simple structure, which allows us to compute its DFS tree efficiently.

Lemma 3. *The non-trivial bicomps of H_i are the boundary cycles of G_i.*

Proof. Consider a cycle C in H_i. All faces incident to C are at level i or greater. Since G_{i-1} is uniform, all its faces are either inside or outside C. Assume w.l.o.g. that G_{i-1} is inside C. Then none of the faces outside C shares a vertex with a level-$(i-1)$ face. That is, all faces outside C must be at level at least $i+1$, which means that C is a boundary cycle of G_i. Thus any cycle in H_i is a boundary cycle of G_i. Every bicomp that is not a cycle consists of at least two cycles sharing at least two vertices; but the cycles must be boundary cycles, and two boundary cycles of a uniform graph cannot share two vertices. Hence every bicomp is a cycle and thus a boundary cycle. □

Assume for the sake of simplicity that the boundary of G_{i-1} is a simple cycle, so that $\overline{G_{i-1}}$ is uniform. During the construction of the DFS tree for G we

maintain the following invariant used to prove the correctness of the algorithm: For every boundary cycle C of G_{i-1}, there is a vertex v on C such that the path traversed by walking clockwise along C is a path in T_{i-1}, and v is an ancestor of all vertices in C (Figure 5b). The *depth* of a vertex in G_{i-1} is its distance from s in T_{i-1}. Let $H'_1, \ldots, H'_k$ be the connected components of H_i. They can be found in $O(\text{sort}(|H_i|))$ I/Os [8]. For every component H'_j, we find the deepest vertex v_j on the boundary of G_{i-1} such that there is an edge $\{u_j, v_j\} \in G$ with $u_j \in H'_j$. We find these vertices using a procedure similar to the one used in Sec. 2. Below we show how to compute DFS trees T'_j for components H'_j rooted at nodes u_j in $O(\text{sort}(|H'_j|))$ I/Os. Let T_i be the spanning tree of G_i obtained by adding these DFS trees and all edges $\{u_j, v_j\}$ to T_{i-1}. T_i is a DFS tree for G_i: Let $\{v, w\}$ be a non tree edge with $v \in H'_j$. Then either $w \in H'_j$, or w is a boundary vertex of G_{i-1} because $H_i \subseteq G \setminus G_{i-1}$. In the former case, $\{v, w\}$ is a back-edge, as T'_j is a DFS tree for H'_j. In the latter case, $\{v, w\}$ is a back-edge because v is a descendant of u_j, and w is an ancestor of v_j, by the choice of v_j and by our invariant.

All that remains to show is how to compute the DFS tree rooted at u_j for each connected component H'_j of H_i. If we can compute DFS trees for the biconnected components of H'_j, we obtain a DFS tree for H'_j using the bicomp-cutpoint tree as in Sec. 2. By Lemma 3 the non-trivial biconnected components of H_i are cycles. Let C be such a cycle in H'_j, and v be the chosen root for the DFS tree of C. The path obtained after removing the edge between v and its counterclockwise neighbor w along C is a DFS tree for C. We find w using techniques similar to those applied in Sec. 2. In total we compute the DFS tree for H'_j in $O(\text{sort}(|H'_j|))$ I/Os. As this adds simple paths along the boundary cycles of G_i to T_i, the above invariant is preserved.

For the sake of simplicity all the previous arguments were based on the assumption that the boundary of G_{i-1} is a simple cycle. In the general case we compute the boundary cycles $C_1, \ldots, C_k$ of G_{i-1} and apply the above algorithm to every C_j. Each cycle C_j is the boundary of a uniform component G'_j of $\overline{G_{i-1}}$. Thus, cycles $C_1, \ldots, C_k$ separate subgraphs $H_{i,j} = H_i \cap G'_j$ from each other. Details will appear in the full paper. This concludes the proof of Thm. 3.

4 Conclusions

We developed the first $o(N)$ and linear space algorithm for DFS in planar graphs. We also designed an $O(\text{sort}(N))$ reduction from planar DFS to planar BFS, proving that in external memory DFS is not harder than BFS and thus providing further evidence that BFS is among the hardest problems for planar graphs.

Adding the single source shortest path algorithm of [4] as an intermediate reduction step, we can modify our reduction algorithm in order to reduce planar DFS to BFS on either a planar triangulated graph or a planar 3-regular graph. Developing an efficient BFS algorithm for one of these classes of graphs remains an open problem.

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995.
3. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.
4. L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2000.
5. A. Broder, R. Kumar, F. Manghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web: experiments and models. In *Proc. WWW Conference*, 2000.
6. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 859–860, 2000.
7. A. L. Buchsbaum and J. R. Westbrook. Maintaining hierarchical graph views. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 566–575, 2000.
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. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.
9. T. Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, August 1990.
10. D. Hutchinson, A. Maheshwari, and N. Zeh. An external-memory data structure for shortest path queries. In *Proc. Annual Combinatorics and Computing Conference, LNCS 1627*, pages 51–60, 1999.
11. J. JáJá and R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304–311, March 1988.
12. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.
13. A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1741*, pages 307–316, 1999.
14. A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 89–90, 2001.
15. U. Meyer. External memory bfs on undirected graphs with bounded degree. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 87–88, 2001.
16. G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.
17. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.
18. J. R. Smith. Parallel algorithms for depth-first searches I. Planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.
19. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 1–38. DIMACS series in Discrete Mathematics and Theoretical Computer Science, 1999.

Author Index