Frank Dehne
John Iacono
Jörg-Rüdiger Sack (Eds.)

Algorithms and Data Structures

12th International Symposium, WADS 2011
New York, NY, USA, August 2011
Proceedings

Springer

Lecture Notes in Computer Science 6844

Frank Dehne   John Iacono
Jörg-Rüdiger Sack (Eds.)

Algorithms and Data Structures

12th International Symposium, WADS 2011
New York, NY, USA, August 15-17, 2011
Proceedings

Springer

Volume Editors

Frank Dehne
Carleton University, School of Computer Science
Parallel Computing and Bioinformatics Laboratory
VISM Building, Room 6210, 1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
E-mail: frank@dehne.net

John Iacono
Polytechnic Institute of New York University
Department of Computer Science and Engineering
5 MetroTech Center, New York, NY 11201, USA
E-mail: jiacono@poly.edu

Jörg-Rüdiger Sack
Carleton University, School of Computer Science
Herzberg Laboratories
Herzberg Building, Room 5350, 1125 Colonel By Drive
Ottawa, ON K1S 5B6, Canada
E-mail: sack@scs.carleton.ca

Preface

This volume contains the papers presented at the 2011 Algorithms and Data Structures Symposium (WADS 2011), formerly Workshop on Algorithms and Data Structures, held during August 15-17, 2011 at the Polytechnic Institute of New York University in Brooklyn, New York. WADS alternates with the Scandinavian Workshop on Algorithms Theory (SWAT), continuing the tradition of SWAT and WADS starting with SWAT 1988 and WADS 1989.

In response to the call for papers, 141 papers were submitted. From these submissions, the Program Committee selected 59 papers for presentation at WADS 2011. In addition, invited lectures were given by the following distinguished researchers: Janos Pach, Mihai Pătraşcu, and Robert E. Tarjan. The proceedings do not contain an abstract of Mihai Pătraşcu's invited lecture "Modern Data Structures", because it was unavailable at the time of printing for health reasons.

We would like to express our appreciation to the Program Committee members, invited speakers, reviewers, and all authors who submitted papers.

May 2011   Frank Dehne
                                                             John Iacono
                                                        Jörg-Rüdiger Sack

Organization

Program Committee

Oswin Aichholzer	Graz University of Technology, Austria
David Bader	Georgia Institute of Technology, USA
Piotr Berman	Penn State University, USA
Prosenjit Bose	Carleton University, Canada
Timothy Chan	University of Waterloo, Canada
Otfried Cheong	KAIST, Korea
Frank Dehne	Carleton University, Canada
Marina Gavrilova	University of Calgary, Canada
Roberto Grossi	University of Pisa, Italy
John Iacono	Polytechnic Institute of New York University, USA
Giuseppe Italiano	University of Rome "Tor Vergata", Italy
Naoki Katoh	Kyoto University, Japan
Rolf Klein	University of Bonn, Germany
Eduardo Sany Laber	PUC Rio, Brazil
Mike Langston	University of Tennessee, USA
Moshe Lewenstein	Bar-Ilan University, Israel
M. Müller-Hannemann	University of Halle-Wittenberg, Germany
Joerg-Ruediger Sack	Carleton University, Canada
Peter Sanders	University of Karlsruhe, Germany
Paul Spirakis	University of Patras, Greece
Subhash Suri	University of California, Santa Barbara, USA
Monique Teillaud	INRIA Sophia Antipolis, France
Jan Arne Telle	University of Bergen, Norway
Marc Van Kreveld	Utrecht University, The Netherlands

Additional Reviewers

Aloupis, Greg	Bauer, Reinhard
Alt, Helmut	Bereg, Sergey
Alves Pessoa, Artur	Berger, Florian
Amir, Amihood	Bodlaender, Hans L.
Aumüller, Martin	Bornstein, Claudson
Bae, Sang Won	Cabello, Sergio
Battaglia, Giovanni	Calka, Pierre
Batz, G. Veit	Caprara, Alberto

Castelli Aleardi, Luca

Cesati, Marco

Chen, Danny Z.

Cicalese, Ferdinando

Cohen-Steiner, David

Collette, Sebastien

Cozzens, Midge

Damaschke, Peter

Deberg, Mark

Devillers, Olivier

Didimo, Walter

Disser, Yann

Driemel, Anne

Durocher, Stephane

Dyer, Ramsay

Eppstein, David

Erdos, Peter L.

Erickson, Jeff

Erlebach, Thomas

Eyraud-Dubois, Lionel

Fagerberg, Rolf

Fekete, Sandor

Ferreira, Rui

Foschini, Luca

Foschni, Luca

Fotakis, Dimitris

Fournier, Hervé

Fukunaga, Takuro

Georgiadis, Loukas

Ghosh, Arijit

Giannopoulos, Panos

Gibson, Matt

Gilbers, Alexander

Goaoc, Xavier

Goldstein, Isaac

Golin, Mordecai

Grandoni, Fabrizio

Gudmundsson, Joachim

Görke, Robert

Ha, Jae-Soon

Halldorsson, Magnus M.

Har-Peled, Sariel

Haverkort, Herman

Henriques Carvalho, Marcelo

Hermelin, Danny

Hershberger, John

Hoffmann, Michael

Hong, Seok-Hee

Howat, John

Hüffner, Falk

Jacobs, Tobias

Jørgensen, Allan

Kaminski, Marcin

Kamiyama, Naoyuki

Kaporis, Alexis

Kawahara, Jun

Keil, Mark

Klein, Philip

Kobayashi, Yusuke

Kobourov, Stephen

Kopelowitz, Tsvi

Korman, Matias

Koutsoupias, Elias

Kraschewski, Daniel

Kratochvil, Jan

Kása, Zoltán

Kärkkäinen, Juha

Laber, Eduardo

Lancichinetti, Andrea

Landau, Gad

Langerman, Stefan

Langetepe, Elmar

Laura, Luigi

Lazard, Sylvain

Lecroq, Thierry

Lee, Mira

Lenchner, Jonathan

Leveque, Benjamin

Liotta, Giuseppe

Lopez-Ortiz, Alejandro

Luccio, Fabrizio

Löffler, Maarten

M.M. De Castro, Pedro

Manthey, Bodo

Manzini, Giovanni

Meyerhenke, Henning

Michail, Othon

Molinaro, Marco

Morin, Pat

Moruz, Gabriel

Mulzer, Wolfgang
Mumford, Elena
Mäkinen, Veli
Naswa, Sudhir
Navarro, Gonzalo
Nekrich, Yakov
Nikoletseas, Sotiris
Nöllenburg, Martin
Okamoto, Yoshio
Orlandi, Alessio
Osipov, Vitaly
Otachi, Yota
Ottaviano, Giuseppe
Pal, Sudebkumar
Panagopoulou, Panagiota
Papadopoulos, Fragkiskos
Park, Jeong-Hyeon
Park, Kunsoo
Penninger, Rainer
Phillips, Charles
Porat, Ely
Rabinovich, Yuri
Reinbacher, Iris
Riedy, Jason
Roditty, Liam
Romani, Francesco
Röglin, Heiko
Saitoh, Toshiki
Sauerwald, Thomas
Schlipf, Lena
Schulz, André
Schuman, Catherine
Serna, Maria

Silveira, Rodrigo
Slingsby, Adrian
Smorodinsky, Shakhar
Sotelo, David
Speckmann, Bettina
Stamatiou, Ioannis
Stehn, Fabian
Sviridenko, Maxim
Takaoka, Tadao
Takazawa, Kenjiro
Tanigawa, Shin-Ichi
Tazari, Siamak
Thilikos, Dimitrios
Tischler, German
Toma, Laura
Täubig, Hanjo
Uno, Takeaki
Uno, Yushi
Vigneron, Antoine
Villanger, Yngve
Wang, Kai
Weihe, Karsten
Wiese, Andreas
Wilkinson, Bryan
Wismath, Steve
Wolff, Alexander
Wood, David R.
Wulff-Nilsen, Christian
Xin, Qin
Yang, Jungwoo
Yildiz, Hakan
Yvinec, Mariette
Zarrabi-Zadeh, Hamid

Table of Contents

Piecewise-Linear Approximations of Uncertain Functions

Mohammad Ali Abam[1,*], Mark de Berg[2], and Amirali Khosravi[2,**]

[1] Department of Computer Engineering, Sharif University of Technology,
Tehran, Iran
abam@sharif.ir
[2] Department of Mathematics and Computing Science, TU Eindhoven,
Eindhoven, The Netherlands
{mdberg,akhosrav}@win.tue.nl

Abstract. We study the problem of approximating a function $F : \mathbb{R} \to \mathbb{R}$ by a piecewise-linear function $\overline{F}$ when the values of F at $\{x_1, \ldots, x_n\}$ are given by a discrete probability distribution. Thus, for each x_i we are given a discrete set $y_{i,1}, \ldots, y_{i,m_i}$ of possible function values with associated probabilities $p_{i,j}$ such that $\mathbf{Pr}[F(x_i) = y_{i,j}] = p_{i,j}$. We define the error of $\overline{F}$ as $error(F, \overline{F}) = \max_{i=1}^{n} \mathbf{E}[|F(x_i) - \overline{F}(x_i)|]$. Let $m = \sum_{i=1}^{n} m_i$ be the total number of potential values over all $F(x_i)$. We obtain the following two results: (i) an $O(m)$ algorithm that, given F and a maximum error ε, computes a function $\overline{F}$ with the minimum number of links such that $error(F, \overline{F}) \leqslant \varepsilon$; (ii) an $O(n^{4/3+\delta} + m \log n)$ algorithm that, given F, an integer value $1 \leqslant k \leqslant n$ and any $\delta > 0$, computes a function $\overline{F}$ of at most k links that minimizes $error(F, \overline{F})$.

1 Introduction

Motivation and problem statement. Fitting a function to a given finite set of points sampled from an unknown function $F : \mathbb{R} \to \mathbb{R}$ is a basic problem in mathematics. Typically one is given a class of "simple" functions—linear functions, piecewise linear functions, quadratic functions, etcetera—and the goal is to find a function $\overline{F}$ from that class that fits the sample points best. One way to measure how well $\overline{F}$ fits the sample points is the *uniform metric*, defined as follows. Suppose that F is sampled at $x_1, \ldots, x_n$, with $x_1 < \cdots < x_n$. Then the error of $\overline{F}$ according to the uniform metric is $\sum_{i=1}^{n} |F(x_i) - \overline{F}(x_i)|$. This measure is also known as the l_∞ or the Chebychev error measure.

The problem of finding the best approximation $\overline{F}$ under the uniform metric has been studied from an algorithmic point of view, in particular for the case where the allowed functions are piecewise linear. There are then two optimization problems that can be considered: the min-k and the min-ε problem. In the min-k

---

problem one is given a maximum error $\varepsilon \geqslant 0$ and the goal is to find piecewise-linear function $\overline{\mathrm{F}}$ with error at most ε that minimizes the number of links. In the min-ε problem one is given a number $k \geqslant 1$ and the goal is to find a piecewise-linear function with at most k links that minimizes the error.

The min-k problem was solved in $O(n)$ time by Hakimi and Schmeichel [10]. They also gave an $O(n^2 \log n)$ algorithm for solving the min-ε problem. This was later improved to $O(n^2)$ by Wang *et al.* [17]. Goodrich [8] then managed to obtain an $O(n \log n)$ algorithm.

In this paper we also study the problem of approximating a sampled function by a piecewise-linear function, but we do this in the setting where the function values $\mathrm{F}(x_i)$ at the sample points are not known exactly. Instead we have a discrete probability distribution for each $\mathrm{F}(x_i)$, that is, we have a discrete set $y_{i,1}, \ldots, y_{i,m_i}$ of possible values with associated probabilities $p_{i,j}$ such that $\mathbf{Pr}[\mathrm{F}(x_i) = y_{i,j}] = p_{i,j}$. We call such a function an *uncertain function*. The goal is now to find a piecewise-linear function $\overline{\mathrm{F}}$ with at most k links that minimizes the expected error (the min-ε problem) or a piecewise-linear function $\overline{\mathrm{F}}$ with error at most ε that minimizes the number of links (the min-k problem).

There are several possibilities to define the expected error. We use the uniform metric and define our error measure in the following natural way.
$$error(\mathrm{F}, \overline{\mathrm{F}}) = \max \left\{ \; \mathbf{E}[|\mathrm{F}(x_i) - \overline{\mathrm{F}}(x_i)|] : 1 \leqslant i \leqslant n \; \right\}.$$
This error is not equal to $error_2(\mathrm{F}, \overline{\mathrm{F}}) = \max\{|\mathbf{E}[\mathrm{F}(x_i)] - \overline{\mathrm{F}}(x_i)| : 1 \leqslant i \leqslant n\}$. Indeed, to minimize $|\mathbf{E}[\mathrm{F}(x_i)] - \overline{\mathrm{F}}(x_i)|$ one should take $\overline{\mathrm{F}}(x_i) = \mathbf{E}[\mathrm{F}(x_i)]$ leading to an error of zero at x_i. Hence, we feel that $error(\mathrm{F}, \overline{\mathrm{F}})$ is more appropriate than $error_2(\mathrm{F}, \overline{\mathrm{F}})$. (Note that approximating F under error measure $error_2$ boils down to approximating the function $G : \mathbb{R} \to \mathbb{R}$ with $G(x_i) = \mathbf{E}[\mathrm{F}(x_i)]$.)

Related work. The problem of approximating a sampled function can be seen as a special case of line simplification. The line-simplification problem is to approximate a given polygonal curve $P = p_1, p_2, \ldots, p_n$ by a simpler polygonal curve $Q = q_1, q_2, \ldots, q_k$. The problem comes in many flavors, depending on the restrictions that are put on the approximation Q, and on the error measure $error(P, Q)$ that defines the quality of the approximation. A typical restriction is that the sequence of vertices of Q be a subsequence of the vertices of P, with $q_1 = p_1$ and $q_k = p_n$; the unrestricted version, where the vertices $q_1, q_2, \ldots, q_k$ can be chosen arbitrarily, has been studied as well. (In this paper we do not restrict the locations of the breakpoints of our piecewise-linear function.) Typical error measures are the Hausdorff distance and the Fréchet distance [4].

The line-simplification has been studied extensively. The oldest and probably best-known algorithm for line simplification is the so-called Douglas-Peucker algorithm [7], dating back to 1973. This algorithm achieves good results in practice, but it is not guaranteed to give optimal results. Over the past 20 years or so, algorithms giving optimal results have been developed for many line-simplification variants [1,2,3,6,8,10,11,?,16]. Although both function approximation and line-simplification are well-studied problems, and there has been ample research on uncertain data in other contexts, the problem we study has, to the best of our knowledge, not been studied so far.
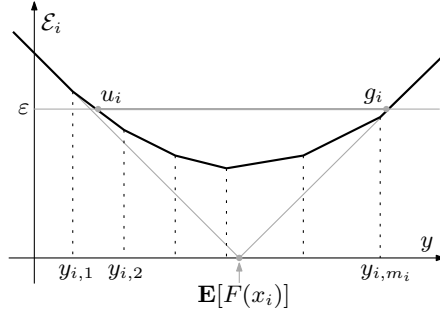
Fig. 1. The function $\mathcal{E}_i(y)$

Our results. We start by studying the min-k problem. As it turns out, this problem is fairly easily reduced to the problem of computing a minimum-link path that stabs a set of vertical segments. The latter problem can be solved in linear time [10], leading to an algorithm for the min-k problem running in $O(m)$ time, where $m = \sum_{i=1}^{n} m_i$. We then turn our attention to the much more challenging min-ε problem, where we present an algorithm that, for any fixed $\delta > 0$, runs in $O(n^{4/3+\delta} + m \log n)$ time. Our algorithm uses similar ideas as the algorithm from Goodrich [8], but it requires several new ingredients to adapt it to the case of uncertain functions. For the important special case $k = 1$—here we wish to find the best linear function to approximate F—we obtain a faster algorithm for the min-ε problem, running in $O(m \log m)$ time.

2 The min-k Problem

We start by studying the properties of $error(F, \overline{F})$. First we define an error function $\mathcal{E}_i(y)$ for every value x_i:

$$\mathcal{E}_i(y) = \mathbf{E}[|F(x_i) - y|].$$

Observe that $\mathcal{E}_i(\overline{F}(x_i))$ is the expected error of $\overline{F}$ at x_i, and $error(F, \overline{F}) = \max_{i=1}^{n} \mathcal{E}_i(\overline{F}(x_i))$. The following lemma shows what $\mathcal{E}_i(y)$ looks like. For simplicity we assume $y_{i,1} \leqslant \cdots \leqslant y_{i,m_i}$ for $1 \leqslant i \leqslant n$.

Lemma 1. *For any i, the function $\mathcal{E}_i(y)$ is a convex piecewise-linear function with $m_i + 1$ links.*

Proof. To simplify the presentation, define $y_{i,0} = -\infty$ and $y_{i,m_i+1} = +\infty$, and we set $p_{i,0} = p_{i,m_i+1} = 0$. Now fix some j with $0 \leqslant j \leqslant m_i$, and consider the y-interval $[y_{i,j}, y_{i,j+1}]$. Within this interval we have

$$\mathcal{E}_i(y) = \mathbf{E}[|F(x_i) - y|]$$
$$= \sum_{\ell=1}^{j} p_{i,\ell}(y - y_{i,\ell}) + \sum_{\ell=j+1}^{m_i} p_{i,\ell}(y_{i,\ell} - y)$$
$$= \left(\sum_{\ell=1}^{j} p_{i,\ell} - \sum_{\ell=j+1}^{m_i} p_{i,\ell} \right) \cdot y - \left(\sum_{\ell=1}^{j} p_{i,\ell} y_{i,\ell} - \sum_{\ell=j+1}^{m_i} p_{i,\ell} y_{i,\ell} \right)$$
$$= a_j y + b_j,$$

where

$$a_j = \sum_{\ell=1}^{j} p_{i,\ell} - \sum_{\ell=j+1}^{m_i} p_{i,\ell} \quad \text{and} \quad b_j = -\left(\sum_{\ell=1}^{j} p_{i,\ell} y_{i,\ell} - \sum_{\ell=j+1}^{m_i} p_{i,\ell} y_{i,\ell}\right).$$

Hence, $\mathcal{E}_i(y)$ is a piecewise-linear function with $m_i + 1$ links. Moreover, since $a_0 = -1 \leqslant a_1 \leqslant \cdots \leqslant a_{m_i} = 1$, it indeed is convex. It is not difficult to see that the first and last links of $\mathcal{E}_i$, when extended, meet exactly in the point $(\mathbf{E}[\mathrm{F}(x_i)], 0)$; see Fig. 1. (Actually these two links, when extended, form the graph of the function $g(y) = |\mathbf{E}[\mathrm{F}(x_i)] - y|$, so they correspond to the error at x_i as given by $error_2$.)   □

Now consider the min-k problem, and let ε be the given bound on the maximum error. Because $\mathcal{E}_i(y)$ is a convex function, there exists an interval $[u_i, g_i]$ such that $\mathcal{E}_i(y) \leqslant \varepsilon$ if and only if $y \in [u_i, g_i]$; see Fig. 1. The interval $[u_i, g_i]$ can be computed in $O(m_i)$ time. When there exists an i such that $[u_i, g_i]$ is empty, we report that there is no $\overline{\mathrm{F}}$ approximating F within error ε. Otherwise, the problem is reduced to finding a function $\overline{\mathrm{F}}$ with a minimum number of links stabbing every vertical segment $x_i \times [u_i, g_i]$. This problem can be solved in linear time [10], leading to the following theorem.

Theorem 1. *Let* $\mathrm{F} : \mathbb{R} \to \mathbb{R}$ *be an uncertain function whose values are given at n points $\{x_1, \ldots, x_n\}$ and let m be the total number of possible values at these points. For a given ε, a piecewise-linear function $\overline{\mathrm{F}}$ with a minimum number of links such that* $error(\mathrm{F}, \overline{\mathrm{F}}) \leqslant \varepsilon$ *can be computed in $O(m)$ time.*

Remark 1. Above we computed the intervals $[u_i, g_i]$ in $O(m_i)$ time in a brute-force manner. However, we can also compute the values u_i and g_i in $O(\log m_i)$ time using binary search. Then, after computing the functions $\mathcal{E}_i$ in $O(m)$ time in total, we can construct the segments $x_i \times [u_i, g_i]$ in $O(\sum_i \log m_i) = O(n \log m)$ time. Thus, after $O(m)$ preprocessing, the min-k problem can be solved in $O(n \log m)$ time. This can be used to speed up the algorithm from the next section when $n = o(m/\log m)$. For simplicity we do not consider this improvement in the next section.

3   The min-ε Problem

We now turn our attention to the min-ε problem. Let k be the maximum number of links we are allowed to use in our approximation. For any $\varepsilon > 0$, define $\mathcal{K}(\varepsilon)$ to be the minimum number of links of any approximation $\overline{\mathrm{F}}$ such that $error(\mathrm{F}, \overline{\mathrm{F}}) \leqslant \varepsilon$. Note that $\mathcal{K}(\varepsilon)$ can be computed with the algorithm from the previous section. Clearly, if $\varepsilon_1 < \varepsilon_2$ then $\mathcal{K}(\varepsilon_1) \geqslant \mathcal{K}(\varepsilon_2)$. Hence, $\mathcal{K}(\varepsilon)$ is a non-increasing function of ε and $\mathcal{K}(\varepsilon) \leqslant n$. Our goal is now to find the smallest ε such that $\mathcal{K}(\varepsilon) \leqslant k$. Let's call this value ε^*. Because $\mathcal{K}(\varepsilon)$ is non-increasing, the idea is to use a binary search to find ε^*, with the algorithm from the previous section as decision procedure. Doing this in an efficient manner is not so easy,

however. We will proceed in several phases, zooming in further and further to the value ε^*, as explained next.

Our algorithm maintains an *active interval* $\mathcal{I}$ containing ε^*. Initially $\mathcal{I} = [0, \infty)$. A basic subroutine is to refine $\mathcal{I}$ on the basis of a set $\mathcal{S}$ of ε-values, whose output is the smallest interval containing ε^* whose endpoints come from the set $\mathcal{S} \cup \{\text{endpoint of } \mathcal{I}\}$. The subroutine *ShrinkActiveInterval* runs in $O(|\mathcal{S}| \log |\mathcal{S}| + m \log |\mathcal{S}|)$ time.

> *ShrinkActiveInterval*$(\mathcal{S}, \mathcal{I})$
> Sort $\mathcal{S}$ to get a sorted list $\varepsilon_1 < \varepsilon_2 < \cdots < \varepsilon_h$. Add values $\varepsilon_0 = -\infty$ and $\varepsilon_{h+1} = \infty$. Do a binary search over $\varepsilon_0, \ldots, \varepsilon_{h+1}$ to find an interval $[\varepsilon_j, \varepsilon_{j+1}]$ containing ε^*; here the basic test during the binary search— namely whether $\varepsilon^* \leqslant \varepsilon_l$, for some ε_l—is equivalent to testing whether $\mathcal{K}(\varepsilon_l) \leqslant k$; this can be done in $O(m)$ time with the algorithm from the previous section. Finally, return $\mathcal{I} \cap [\varepsilon_j, \varepsilon_{j+1}]$.

The first phase. In the previous section we have seen that each error function $\mathcal{E}_i$ is a convex piecewise-linear function with m_i breakpoints. Let $E_i = \{\mathcal{E}_i(y_{i,j}) : 1 \leqslant j \leqslant m_i\}$ denote the set of error-values of the breakpoints of $\mathcal{E}_i$, and let $E = E_1 \cup \cdots \cup E_n$. The first phase of our algorithm is to call *ShrinkActiveInterval*$(E, [0, \infty))$ to find two consecutive values $\varepsilon_j, \varepsilon_{j+1} \in E$ such that $\varepsilon_j \leqslant \varepsilon^* \leqslant \varepsilon_{j+1}$. Since $|E| = m$, this takes $O(m \log m)$ time.

Recall that for a given ε, the approximation $\overline{\mathbf{F}}$ has to intersect the segment $x_i \times [u_i, g_i]$ in order for the error to be at most ε at x_i. Now imagine increasing ε from ε_j to ε_{j+1}. Then the values u_i and g_i change continuously. In fact, since $\mathcal{E}_i$ is a convex piecewise-linear function and ε_j and ε_{j+1} are consecutive values from E, the values u_i and g_i change linearly, that is, we have

$$u_i(\varepsilon) = a_i \varepsilon + b_i \quad \text{and} \quad g_i(\varepsilon) = c_i \varepsilon + d_i$$

for constants a_i, b_i, c_i, d_i that can be computed from $\mathcal{E}_i$. As ε increases, u_i decreases and g_i increases—thus $a_i < 0$ and $c_i > 0$—and so the vertical segment $x_i \times [u_i, g_i]$ is growing. After the first phase we have $\mathcal{I} = [\varepsilon_j, \varepsilon_{j+1}]$ and the task is to find the smallest $\varepsilon \in [\varepsilon_j, \varepsilon_{j+1}]$ such that there exists a k-link path stabbing all the segments.

Intermezzo: the case $k = 1$. We first consider the second phase for the special but important case where $k = 1$. This case can be considered as the problem of finding a regression line for uncertain points except that our error is not the squared distance. Thus we want to approximate the uncertain function F by a single line $\ell : y = ax + b$ that minimizes the error. The line ℓ stabs a segment $x_i \times [u_i, g_i]$ if ℓ is above (x_i, u_i) and below (x_i, g_i). In other words, we need $ax_i + b \geqslant a_i \varepsilon + b_i$ and $ax_i + b \leqslant c_i \varepsilon + d_i$. Hence, the case $k = 1$ can be handled by solving the following linear program with variables a, b, ε:

$$
\begin{aligned}
\text{Minimize} \quad & \varepsilon \\
\text{Subject to} \quad & x_i a + b - a_i \varepsilon \geqslant b_i && \text{for all } 1 \leqslant i \leqslant n \\
& x_i a + b - c_i \varepsilon \leqslant d_i && \text{for all } 1 \leqslant i \leqslant n
\end{aligned}
$$
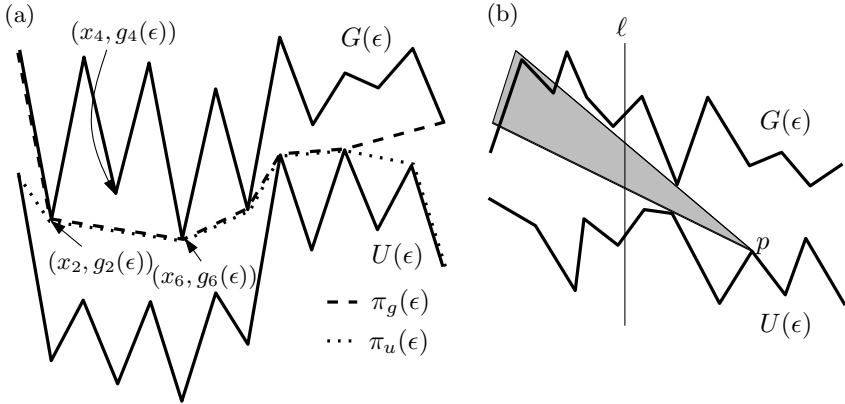
Fig. 2. (a) The paths $\pi_u(\varepsilon)$ and $\pi_g(\varepsilon)$. (b) The visibility cone of p.

Since a 3-dimensional linear program can be solved in linear expected time [5], we get the following theorem.

Theorem 2. *The line ℓ minimizing* error(F, ℓ) *can be computed in $O(m \log m)$ expected time.*

The second phase. As mentioned earlier, our algorithm uses ideas from the algorithm that Goodrich [8] developed for the case of certain[1] functions. Next we sketch his algorithm and explain the difficulties in applying it to our problem.

For a certain function F, the error functions $\mathcal{E}_i(y)$ are cones whose bounding lines have slope -1 and $+1$, respectively. This implies that, using the notation from above, we have $u_i(0) = g_i(0)$, and $a_i = -1$, and $c_i = 1$ for all i. For a given ε, we define $U(\varepsilon)$ to be the polygonal chain $(x_1, u_1(\varepsilon)), \ldots, (x_n, u_n(\varepsilon))$ and $G(\varepsilon)$ to be the polygonal chain $(x_1, g_1(\varepsilon)), \ldots, (x_n, g_n(\varepsilon))$. (NB: The minimum-link path of error at most ε stabbing all segments $x_i \times [u_i, g_i]$ does not have to stay within the region bounded by $U(\varepsilon)$ and $G(\varepsilon)$.) Let $\pi_u(\varepsilon)$ be the Euclidean shortest path from $(x_1, u_1(\varepsilon))$ to $(x_n, u_n(\varepsilon))$ that is below $G(\varepsilon)$ and above $U(\varepsilon)$—see Fig. 2(a) for an illustration. Similarly, let $\pi_g(\varepsilon)$ be the Euclidean shortest path from $(x_1, g_1(\varepsilon))$ to $(x_n, g_n(\varepsilon))$ that is below $G(\varepsilon)$ and above $U(\varepsilon)$. The paths $\pi_u(\varepsilon)$ and $\pi_g(\varepsilon)$ together form a so-called *hourglass*, which we denote by $H(\varepsilon)$. An edge $e \in H(\varepsilon)$ is called an *inflection edge* if one of its endpoints lies on $U(\varepsilon)$ and the other one lies on $G(\varepsilon)$. Goodrich [8] showed that there is a minimum-link function $\overline{\mathrm{F}}$ with error ε such that each inflection edge is contained in a link of $\overline{\mathrm{F}}$ and in between two links containing inflection edges the function is convex or concave. (In other words, the "zig-zags" of $\overline{\mathrm{F}}$ occur exactly at the inflection edges.)

Goodrich then proceeds by computing all values of ε at which the set of inflection edges changes; these are called *geodesic-critical* values of ε. Note that

---

[1] We use the term *certain function* for a function with exactly one possible value per sample point, that is, when no uncertainty is involved.

the moments at which an inflection edge changes are exactly the moments at which the hourglass $H(\varepsilon)$ changes. The number of geodesic-critical values is $O(n)$ and they can be found in $O(n \log n)$ time [8]. After finding these geodesic-critical values, a binary search is applied to find two consecutive critical values $\varepsilon_j, \varepsilon_{j+1}$ such that $\varepsilon_j \leqslant \varepsilon^* \leqslant \varepsilon_{j+1}$. Then the inflection edges that $\overline{\mathrm{F}}$ must contain are known, and from this F can be computed using parametric search.

The main difference between our setting and the setting of Goodrich is that the points $(x_i, u_i(\varepsilon))$—and, similarly, the points $(x_i, g_i(\varepsilon))$—do not move at the same speed. As a result the basic lemma underlying the efficiency of the approach, namely that there are only $O(n)$ geodesic-critical values of ε, no longer holds. The following lemma shows that the number of such values can actually be quadratic. The proof of the lemma can be found in the full version of the paper.

Lemma 2. *There is an instance of n vertical segments $x_i \times [u_i(\varepsilon), g_i(\varepsilon)]$ where the $(x_i, u_i(\varepsilon))$'s and $(x_i, g_i(\varepsilon))$'s are moving at constant (but different) velocities such that the number of geodesic-critical events is $\Omega(n^2)$.*

The fact that the number of geodesic-critical values of ε is $\Omega(n^2)$ is not the only problem we face. The other problem is that detecting these events becomes more difficult in our setting. When all points on $U(\varepsilon)$ and on $G(\varepsilon)$ move with the same speed, then these events occur only when two consecutive edges of $\pi_u(\varepsilon)$ become collinear or when two consecutive edges of $\pi_g(\varepsilon)$ become collinear. When the points have different speeds, however, this is no longer the case. In Fig. 2, for example, the hourglass $H(\varepsilon)$ can change when $(x_2, g_2(\varepsilon))$, $(x_4, g_4(\varepsilon))$ and $(x_6, g_6(\varepsilon))$ become collinear (which could happen when $(x_4, g_4(\varepsilon))$ moves up relatively slowly).

Below we show how to overcome these two hurdles and obtain an algorithm with subquadratic running time.

We start with a useful observation. Let $\Psi(\varepsilon)$ be the simple polygon whose boundary consists of the chains $G(\varepsilon)$ and $U(\varepsilon)$, and the vertical segments $x_1 \times [u_1(\varepsilon), g_1(\varepsilon)]$ and $x_n \times [u_n(\varepsilon), g_n(\varepsilon)]$. Let $\mathcal{G}(\varepsilon)$ be the visibility graph of $\Psi(\varepsilon)$, that is, $\mathcal{G}(\varepsilon)$ is the graph whose nodes are the vertices of $\Psi(\varepsilon)$ and where two nodes are connected by an edge if the corresponding vertices can see each other inside $\Psi(\varepsilon)$. As ε increases and the vertices of $\Psi(\varepsilon)$ move, $\mathcal{G}(\varepsilon)$ can change combinatorially, that is, edges may appear or disappear.

Lemma 3. *The visibility graph $\mathcal{G}(\varepsilon)$ changes $O(n^2)$ times as ε increases from 0 to ∞. Moreover, if $\mathcal{G}(\varepsilon)$ does not change when ε is restricted to some interval $[\varepsilon_1, \varepsilon_2]$ then $H(\varepsilon)$ does not change either when ε is in this interval.*

Proof. First we observe that any three vertices of $\Psi(\varepsilon)$ become collinear at most once, because each vertex moves with constant velocity and has constant x-coordinate. Indeed, three points p, q, r are collinear when $(p_y - q_y)/(p_x - q_x) = (p_y - r_y)/(p_x - r_x)$, and when p_x, q_x, r_x are constant and p_y, q_y, r_y are linear functions of ε, then this equation has one solution (or possibly infinitely many solutions, which means the points are always collinear).

Next we show that every edge e of $\mathcal{G}$, once it disappears, cannot re-appear. Define $\tilde{u}_i = (x_i, u_i)$ and $\tilde{g}_i = (x_i, g_i)$. Assume that $e = (\tilde{u}_i, \tilde{u}_j)$ for some i, j;

the case where one or both of the endpoints of e are on $G(\varepsilon)$ is similar. None of the vertices of $G(\varepsilon)$ can stop $\tilde{u}_i$ and $\tilde{u}_j$ from seeing each other, since all $\tilde{u}_i$'s move down and all $\tilde{g}_i$'s move up. Hence, the only reason for $\tilde{u}_i$ and $\tilde{u}_j$ to become invisible to each other is that some vertex $\tilde{u}_l$, with $i < l < j$, crosses e. For $\tilde{u}_i$ and $\tilde{u}_j$ to become visible again, $\tilde{u}_l$ would have to cross e again, but this is impossible since $\tilde{u}_i, \tilde{u}_j, \tilde{u}_j$ can become collinear at most once. It follows that each edge can appear and disappear at most once, and since there are $O(n^2)$ edges in total, $\mathcal{G}$ changes $O(n^2)$ times.

The second part of the lemma immediately follows from the fact that the shortest paths $\pi_u(\varepsilon)$ and $\pi_g(\varepsilon)$ cannot "jump" as ε changes continuously, because shortest paths in a simple polygon are unique. Hence, these shortest paths—and, consequently, $H(\varepsilon)$—can only change when $\mathcal{G}(\varepsilon)$ changes. $\square$

Computing all combinatorial changes of $\mathcal{G}$ still results in an algorithm with running time $\Omega(n^2)$. Next we show that it suffices to compute $O(n^{4/3+\delta})$ combinatorial changes of $\mathcal{G}$ in order to find an interval $[\varepsilon_1, \varepsilon_2]$ such that $\varepsilon^* \in [\varepsilon_1, \varepsilon_2]$ and $\mathcal{G}$ does not change in this interval. (Recall that ε^* denotes the minimum error that can be achieved with k links, and that we thus wish to find).

Obtaining stable visibility cones. Let $\mathcal{I}$ be the active interval resulting from the first phase of our algorithm. We now describe an approach to quickly find a subset of the events where $\mathcal{G}$ changes, which we can use to further shrink $\mathcal{I}$.

Let ℓ be a vertical line splitting the set of vertices of Ψ into two (roughly) equal-sized subsets. We will concentrate on the visibility edges whose endpoints lie on the different sides of ℓ; the approach will be applied recursively to deal with the visibility edges lying completely to the right or completely to the left of ℓ. For a vertex p of $\Psi(\varepsilon)$ we define $\sigma(p, \varepsilon)$, the *visibility cone* of p in $\Psi(\varepsilon)$, as the cone with apex p that contains all rays emanating from p that cross a point on ℓ that is visible from p (within $\Psi(\varepsilon)$). A crucial observation is that for a vertex p to the left of ℓ and a vertex q to the right of ℓ, we have that (p, q) is an edge of $\mathcal{G}(\varepsilon)$ if and only if $p \in \sigma(q, \varepsilon)$ and $q \in \sigma(p, \varepsilon)$.

As the vertices of Ψ move, $\sigma(p, \varepsilon)$ changes continuously but its combinatorial description (the vertices defining its sides) changes at discrete times. Notice that the bottom side of $\sigma(p, \varepsilon)$ passes through a vertex of the lower boundary of $\Psi(\varepsilon)$ lying to the same side of ℓ as p. More precisely, if $U(p, \ell)$ denotes the set of vertices on the lower boundary of $\Psi(\varepsilon)$ that lie between ℓ and the vertical line through p, then the lower side of $\sigma(p, \varepsilon)$ is tangent to the upper hull of $U(p, \ell)$—see Fig. 2(b). Similarly, if $G(p, \ell)$ denotes the set of vertices on the upper boundary of $\Psi(\varepsilon)$ that lie between ℓ and the vertical line through p, then the upper side of $\sigma(p, \varepsilon)$ is tangent to the lower hull of $G(p, \ell)$. The following lemma shows how many times the lower hull changes of a set of points that all move vertically upwards; by symmetry, the lemma also applies to the number of changes to the upper hull of points moving downwards.

Lemma 4. *Suppose n points in the plane move vertically upward, each with its own constant velocity. Then the number of combinatorial changes to the lower hull is $O(n)$. Furthermore, all event times at which the lower hull changes can be computed in $O(n \log^3 n)$ time.*

Proof. Let $\{p_1, \ldots, p_n\}$ be the set of points vertically moving upwards with constant velocities, ordered from left to right. Since the points move with constant velocities, any three points become collinear at most once. As in the proof of Lemma 3, this implies that once a point disappears from the lower hull, it cannot re-appear. Hence, the number of changes to the lower hull is $O(n)$.

To compute all event times, we construct a balanced binary tree $\mathcal{T}$ storing the points $\{p_1, \ldots, p_n\}$ in its leaves in an ordered manner based on their x-coordinates. At each node ν of $\mathcal{T}$, we maintain a kinetic data structure to track $\mathrm{LH}(\nu)$, the lower hull of the points stored in the subtree rooted at ν. Let ν_r and ν_l be the right and left child of node ν in $\mathcal{T}$. Then $\mathrm{LH}(\nu)$ is formed by portions of $\mathrm{LH}(\nu_r)$ and $\mathrm{LH}(\nu_l)$ and the common tangent of $\mathrm{LH}(\nu_r)$ and $\mathrm{LH}(\nu_l)$. This implies that in order to track all changes to the lower hull of the whole point set, it suffices to track for each node ν the changes to common tangents of $\mathrm{LH}(\nu_r)$ and $\mathrm{LH}(\nu_l)$. We thus maintain for each node ν a certificate that can be used to find out when the tangent changes. This certificate involves at most six points: the two points determining the current tangent and the at most four points (two on $\mathrm{LH}(\nu_r)$ and two on $\mathrm{LH}(\nu_l)$) adjacent to these points. The failure times of the $O(n)$ certificates are put into an event queue. When a certificate fails we update the corresponding tangent, and we update the failure time of the certificate (which means updating the event queue). A change at ν may propagate upwards in the tree—that is, it may trigger at most $O(\log n)$ changes in ascendants of ν. Hence, handling a certificate failure takes $O(\log^2 n)$ time. Since the number of changes at each node ν is the number of points stored at the subtree rooted at ν, we handle at most $O(n \log n)$ events in total. Each event takes $O(\log^2 n)$ time, and so we can compute all events in $O(n \log^3 n)$ time.   □

Our goal is to shrink the active interval $\mathcal{I}$ to a smaller interval such that the visibility cones of the points are stable, that is, do not change combinatorially. Doing this for each p individually will still be too slow, however. We therefore proceed as follows.

Recall that we split Ψ into two with a vertical line ℓ. Let $\mathcal{R}$ be the set of vertices to the right of ℓ. We show how to shrink $\mathcal{I}$ so that the cones of the points $p \in \mathcal{R}$ are stable. Below we only consider the top sides of the cones, which pass through a vertex of $G(\varepsilon)$; the bottom sides can be handled similarly.

We construct a binary tree $\mathcal{T}_{G,\mathcal{R}}$ on the points in $G(\varepsilon) \cap \mathcal{R}$, based on their x-coordinates. For a node ν, let $P(\nu)$ denote its canonical subset, that is, $P(\nu)$ denotes the set of points in the subtree of ν. Using Lemma 4 we compute all event times—that is, values of ε—at which the lower hull $\mathrm{LH}(P(\nu))$ changes, for each node ν. This takes $O(n \log^4 n)$ time in total and gives us a set $\mathcal{S}$ of $O(n \log n)$ event times. We then call procedure *ShrinkActiveInterval*$(\mathcal{S}, \mathcal{I})$ to further shrink $\mathcal{I}$, taking $O(n \log^2 n + m \log n)$ time. In the new active interval, none of the maintained lower hulls changes combinatorially. This does not mean, however, that the top sides of the cones are stable. For that we need some more work.

Recall that the top side of $\sigma(p, \varepsilon)$ is given by the tangent of p to $\mathrm{LH}(G(p, \ell))$, where $G(p, \ell)$ is the set of points on the $G(\varepsilon)$ in between p and ℓ (with respect to their x-coordinates). The set $G(p, \ell)$ can be formed from $O(\log n)$ canonical

subsets in $\mathcal{T}_{G,\mathcal{R}}$. Each canonical subset $P(\nu)$ gives a candidate tangent for p, namely the tangent from p to $\text{LH}(P(\nu))$. Even though the lower hulls, $\text{LH}(P(\nu))$, are stable, the tangents from p to the lower hulls are not. Next we describe how to shrink the active interval, so that these tangents become stable, and we have $O(\log n)$ stable candidates.

Consider a canonical subset $P(\nu)$ and let $p_1, \ldots, p_h$ be the vertices of $\text{LH}(P(\nu))$, ordered from left to right. An important observation is that, as ε increases and the points move, the tangent from p to $\text{LH}(P(\nu))$ steps from vertex to vertex along $p_1, \ldots, p_h$, either always going forward or always going backwards. (This is true because p can become collinear with any lower-hull edge at most once.) We can therefore proceed as follows. For each point p and each of its canonical subsets, we compute in constant time at what time p and the middle edge of the lower hull of the canonical subset become collinear. Finding the middle edge can be done using binary search, if we store the lower hulls $\text{LH}(P(\nu))$ as a balanced tree. Since we have n points and each of them is associated with $O(\log n)$ canonical subsets, in total we have $O(n \log n)$ event times. We put these into a set $\mathcal{S}$ and call $ShrinkActiveInterval(\mathcal{S}, \mathcal{I})$. In the new active interval the number of vertices of each lower hull to which p can be tangent has halved. We keep on shrinking $\mathcal{I}$ recursively, until we are left with an interval $\mathcal{I}$ such that, for each p and any canonical subset relevant for p, the tangent from p to the lower hull is stable. In total this takes $O(n \log^2 n + m \log n)$ time.

Note that within $\mathcal{I}$ we now have $O(\log n)$ stable candidate tangent lines for each p. We then compute all $O(\log^2 n)$ times at which the candidate tangent lines swap (in their circular order around p), collect all $O(n \log^2 n)$ event times, and call $ShrinkActiveInterval$ once more, taking $O(n \log^3 n + m \log n)$ time.

After this, we are left with an interval $\mathcal{I}$ such that the top side of the cone of each $p \in \mathcal{R}$ is stable. In a similar way we can make sure that the bottom sides are stable, and that the top and bottom sides of the points to the left of ℓ are stable. We get the following lemma.

Lemma 5. *In $O(n \log^3 n + m \log n)$ time we can shrink the active interval $\mathcal{I}$ so that in the new active interval all the cones defined with respect to ℓ are stable.*

We denote the set of edges of $\mathcal{G}$ crossing ℓ by $E(\ell)$. Next we describe a randomization algorithm to shrink the active interval $\mathcal{I}$ such that in the new active interval, the edges of $E(\ell)$ are stable. After this, we recurse on the part of Ψ to the left of ℓ and on the part to the right, so that the whole visibility graph becomes stable.

As already mentioned, (p, q) is an edge of $E(\ell)$ if and only if $p \in \sigma(q, \varepsilon)$ and $q \in \sigma(p, \varepsilon)$. Thus $E(\ell)$ changes when a point p becomes collinear with a side of $\sigma(q, \varepsilon)$ for some q. Without loss of generality we assume $p \in \mathcal{R}$ and $q \in \mathcal{L}$. Thus we have a set $\mathcal{H}$ of at most n half-lines originating from points in $\mathcal{L}$, where each half-line is specified by two points of $\mathcal{L}$, and a set of $n/2$ points from $\mathcal{R}$, and we want to compute the event times at which a point of $\mathcal{R}$ and a half-line of $\mathcal{H}$ become collinear. Again, explicitly enumerating all these event times takes $\Omega(n^2)$ time, so we have to proceed more carefully. To this end we use a variant

of *random halving* [13], which can be made deterministic using the expander approach [12]. We start with a primitive tool which is used in our algorithm.

Lemma 6. *For any ε_1 and ε_2, and any $\delta > 0$, we can preprocess $\mathcal{H}$ and $\mathcal{R}$ in $O(n^{4/3+\delta})$ time into a data structure that allows the following: count in $O(n^{4/3+\delta})$ time all events (that is, all the times at which a half-line in $\mathcal{H}$ and a point in $\mathcal{R}$ become collinear) lying in $[\varepsilon_1, \varepsilon_2]$, and select in $O(\log n)$ time one of these events uniformly at random.*

Proof. (Sketch) Consider a half-line $l \in \mathcal{H}$ and a point $p \in \mathcal{R}$. They become collinear at a time in $[\varepsilon_1, \varepsilon_2]$ if and only if either $p(\varepsilon_1)$ is below $l(\varepsilon_1)$ and $p(\varepsilon_2)$ is above $l(\varepsilon_2)$, or $p(\varepsilon_1)$ is above $l(\varepsilon_1)$ and $p(\varepsilon_2)$ is below $l(\varepsilon_2)$. Therefore we need a data structure to report for all $l \in \mathcal{H}$ the points of $\mathcal{R}$ lying to a given side of $l(\varepsilon_1)$ or $l(\varepsilon_2)$. We construct a multilevel partition tree over points of $\mathcal{R}$ at times ε_1 and ε_2 (one level dealing with ε_1, the other dealing with ε_2), each of whose nodes is associated with a canonical subset of $\mathcal{R}$. The total size of all canonical subsets is $O(n^{4/3+\delta})$. For a query line l, the query procedure selects $O(n^{1/3+\delta})$ pairwise disjoint canonical subsets whose union consists of exactly those points of $\mathcal{R}$ at different sides of l at times ε_1 and ε_2. Based on this we create a set of pairs $\{(A_i, B_i)\}$ with $\sum(|A_i| + |B_i|) = O(n^{4/3+\delta})$ where A_i is a subset of $\mathcal{R}$ and B_i is the subset of $\mathcal{H}$ and each $p \in A_i$ and $l \in B_i$ become collinear at some time in $[\varepsilon_1, \varepsilon_2]$. First we select a pair (A_i, B_i) at random, where the probability of selecting (A_i, B_i) is proportional to $|A_i| * |B_i|$. Then we select an element uniformly at random from A_i and an element uniformly at random from B_i. □

Based on Lemma 6 we proceed as follows. Let $\mathcal{I} = [\varepsilon_1, \varepsilon_2]$ be the current active interval. Let N be the number of event times in $\mathcal{I}$; we can determine N using Lemma 6. Then select an event time ε_3 uniformly at random from the event times in $\mathcal{I}$, again using Lemma 6, and we either shrink $\mathcal{I}$ to $[\varepsilon_1, \varepsilon_3]$ or to $[\varepsilon_3, \varepsilon_2]$ in $O(m)$ time. We recursively continue shrinking $\mathcal{I}$ until the set of events inside $\mathcal{I}$ is $O(n^{4/3})$; the expected number of rounds is $O(\log n)$. Once $N = O(n^{4/3})$ we list all event times, and do a regular binary search.

After this we are left with an active interval $\mathcal{I}$ such that the set $A(\ell)$ of visibility edges crossing ℓ is stable. We recurse on both halves of Ψ to get the whole visibility graph stable. Putting everything together we get the following result.

Lemma 7. *For any $\delta > 0$ in $O(n^{4/3+\delta} + m \log m)$ expected time we can compute an active interval $\mathcal{I}$ containing ε^* where the visibility graph $\mathcal{G}(\varepsilon)$ is stable.*

As observed before, the fact that the visibility graph is stable during the active interval $\mathcal{I} = [\varepsilon_1, \varepsilon_2]$ implies that the set of inflection edges is stable. This means we can compute all inflection edges during this interval by computing in $O(n)$ time the shortest paths $\pi_u(\varepsilon_1)$ and $\pi_g(\varepsilon_1)$. Once this has been done, we can proceed in exactly the same way as Goodrich [8] to find ε^*. Given ε^* we can find a k-link path of error ε^* by solving the min-k problem for ε^*. This leads to our main result.

Theorem 3. *Let* $F : \mathbb{R} \to \mathbb{R}$ *be an uncertain function whose values are given at* n *points* $\{x_1, \ldots, x_n\}$ *and let* m *be the total number of possible values at these points. For a given* k, *and any* $\delta > 0$, *we can compute in* $O(n^{4/3+\delta} + m \log n)$ *time a piecewise-linear function* $\overline{F}$ *with at most* k *links that minimizes* $\mathrm{error}(F, \overline{F}) \leqslant \varepsilon$.

References

1. Abam, M.A., de Berg, M., Hachenberger, P., Zarei, A.: Streaming Algorithms for Line Simplification. Discrete & Computational Geometry 43, 497–515 (2010)
2. Agarwal, P.K., Varadarajan, K.R.: Efficient Algorithms for Approximating Polygonal Chains. In: Chazelle, B., Goodman, J., Pollack, R. (eds.) Discrete & Computational Geometry, vol. 23, pp. 273–291 (2000)
3. Agarwal, P.K., Har-Peled, S., Mustafa, N.H., Wang, Y.: Near-linear Time Approximation Algorithms for Curve Simplification. Algorithmica 42, 203–219 (2005)
4. Alt, H., Godau, M.: Computing the Freéchet Distance between Two Polygonal Curves. International Journal on Computational Geometry and Applications 5, 75–91 (1995)
5. De Berg, M., Cheong, O., Van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Heidelberg (2008)
6. Chan, W.S., Chin, F.: Approximation of Polygonal Curves with Minimum Number of Line Segments. In: Chazelle, B., Goodman, J., Pollack, R. (eds.) Proc. 3rd Annual Symp. on Alg. and Comp., vol. 650, pp. 378–387 (1992)
7. Douglas, D.H., Peucker, T.K.: Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. Canadian Cartographer 10, 112–122 (1973)
8. Goodrich, M.T.: Efficient Piecewise-linear Function Approximation Using the Uniform Metric. Discrete & Computational Geometry 14, 445–462 (1995)
9. Guibas, L.J., Hershberger, J.E., Mitchell, J.S.B., Snoeyink, J.S.: Approximating Polygons and Subdivisions with Minimum Link Paths. International Journal of Computational Geometry and Applications 3, 383–415 (1993)
10. Hakimi, S.L., Schmeichel, E.F.: Fitting Polygonal Functions to a Set of Points in the Plane. Graph. Models Image Process. 52, 132–136 (1991)
11. Imai, H., Iri, M.: Polygonal Approximations of a Curve-formulations and Algorithms. In: Toussaint, G.T. (ed.) Computational Morphology, pp. 71–86 (1988)
12. Katz, M.J., Sharir, M.: An Expander-based Approach to Geometric Optimization. SIAM J. Comput. 26, 1384–1408 (1997)
13. Matousek, J.: Randomized Optimal Algorithm for Slope Selection. Inform. Process. Lett. 36, 183–187 (1991)
14. Melkman, A., O'Rourke, J.: On Polygonal Chain Approximation. In: Toussaint, G.T. (ed.) Computational Morphology, pp. 87–95 (1998)
15. Suri, S.: A Linear Time Algorithm for Minimum Link Paths inside a Simple Polygon. Comput. Vision Graph. Image Process. 35, 99–110 (1986)
16. Toussaint, G.T.: On the Complexity of Approximating Polygonal Curves in the Plane. In: Proc. Int. Symp. on Robotics and Automation (1985)
17. Wang, D.P., Huang, D.P., Chao, H.S., Lee, R.C.T.: Plane Sweep Algorithms for Polygonal Approximation Problems with Applications. In: Ng, K.W., Balasubramanian, N.V., Raghavan, P., Chin, F.Y.L. (eds.) ISAAC 1993. LNCS, vol. 762, pp. 515–522. Springer, Heidelberg (1993)

A Constant Factor Approximation Algorithm for Boxicity of Circular Arc Graphs

Abhijin Adiga, Jasine Babu[*], and L. Sunil Chandran

Department of Computer Science and Automation,
Indian Institute of Science, Bangalore 560012, India
{abhijin,jasine,sunil}@csa.iisc.ernet.in

Abstract. Boxicity of a graph $G(V, E)$ is the minimum integer k such that G can be represented as the intersection graph of k-dimensional axis parallel boxes in $\mathbf{R}^k$. Equivalently, it is the minimum number of interval graphs on the vertex set V such that the intersection of their edge sets is E. It is known that boxicity cannot be approximated even for graph classes like bipartite, co-bipartite and split graphs below $O(n^{0.5-\epsilon})$-factor, for any $\epsilon > 0$ in polynomial time unless $NP = ZPP$. Till date, there is no well known graph class of unbounded boxicity for which even an n^ϵ-factor approximation algorithm for computing boxicity is known, for any $\epsilon < 1$. In this paper, we study the boxicity problem on Circular Arc graphs - intersection graphs of arcs of a circle. We give a $(2+\frac{1}{k})$-factor polynomial time approximation algorithm for computing the boxicity of any circular arc graph along with a corresponding box representation, where $k \geq 1$ is its boxicity. For Normal Circular Arc(NCA) graphs, with an NCA model given, this can be improved to an additive 2-factor approximation algorithm. The time complexity of the algorithms to approximately compute the boxicity is $O(mn+n^2)$ in both these cases and in $O(mn+kn^2)$ which is at most $O(n^3)$ time we also get their corresponding box representations, where n is the number of vertices of the graph and m is its number of edges. The additive 2-factor algorithm directly works for any Proper Circular Arc graph, since computing an NCA model for it can be done in polynomial time.

Keywords: Boxicity, Circular Arc Graphs, Approximation Algorithm.

1 Introduction

Boxicity: Let $G(V, E)$ be a graph. If I_1, I_2, $\cdots$, I_k are interval graphs on the vertex set V with $E(G) = E(I_1) \cap E(I_2) \cap \cdots \cap E(I_k)$, then $\{I_1, I_2, \cdots, I_k\}$ is called a box representation of G of dimension k. Boxicity of G is defined as the minimum number k such that G has a box representation of dimension k. Equivalently, boxicity is the minimum integer k such that G can be represented as the intersection graph of k-dimensional axis parallel boxes in $\mathbf{R}^k$. For dense graphs, a box representation of low dimension requires lesser memory compared

---

to an adjacency list or matrix representation. Availability of a low dimensional box representation makes some well known NP-hard problems like the max-clique problem polynomial time solvable [17].

Introduced by Roberts [16] in 1968, boxicity is combinatorially well studied and many bounds are known in terms of parameters like maximum degree, minimum vertex cover size and tree-width [5]. Boxicity of any graph is upper bounded by $\lfloor \frac{n}{2} \rfloor$ where n is the number of vertices of the graph. It was shown by Scheinerman [18] in 1984 that the boxicity of outer planar graphs is at most two. In 1986, Thomassen [21] proved that the boxicity of planar graphs is at most 3. This parameter is also studied in relation with other dimensional parameters of graphs like partial order dimension and threshold dimension [3,24].

However, computing boxicity is a notoriously hard algorithmic problem. In 1981, Cozzens[6] showed that computing Boxicity is NP-Hard. Later Yannakakis [24] proved that determining whether boxicity of a graph is at most three is NP-Complete and Kratochvil[12] strengthened this by showing that determining whether boxicity of a graph is at most two itself is NP-Complete. Recently, Adiga et.al [3] proved that no polynomial time algorithm for approximating boxicity of bipartite graphs with approximation factor less than $O(n^{0.5-\epsilon})$ is possible unless $NP = ZPP$. Same non-approximability holds in the case of split graphs and co-bipartite graphs too. Even an n^ϵ-factor approximation algorithm, with $\epsilon < 1$ for boxicity is not known till now, for any well known graph class of unbounded boxicity. In this paper, we present a polynomial time $(2 + \frac{1}{k})$-factor approximation algorithm for finding the boxicity of circular arc graphs along with the corresponding box representation, where $k \geq 1$ is the boxicity of the graph. There exist circular arc graphs of arbitrarily high boxicity including the well known Robert's graph (the complement of a perfect matching on n vertices, with n even) which achieves boxicity $\frac{n}{2}$. For normal circular arc graphs, with an NCA model given, we give an additive 2-factor polynomial time approximation algorithm for the same problem. Note that, proper circular arc graphs form a subclass of NCA graphs and computing an NCA model for them can be done in polynomial time. We also give efficient ways of implementing all these algorithms.

Circular Arc Graphs: Circular Arc (CA) graphs are intersection graphs of arcs on a circle. That is, an arc of the circle is associated with each vertex and two vertices are adjacent if and only if their corresponding arcs overlap. It is sometimes thought of as a generalization of interval graphs which are intersection graphs of intervals on the real line. CA graphs became popular in 1970's with a series of papers from Tucker, wherein he proved matrix characterizations for CA graphs [22] and structure theorems for some of its important subclasses[22]. For a detailed description, refer to the survey paper by Lin et.al [13]. Like in the case of interval graphs, linear time recognition algorithms exist for circular arc graphs too [15]. Some of the well known NP-complete problems like tree-width, path-width are known to be polynomial time solvable in the case of CA graphs[19,20]. However, unlike interval graphs, problems like minimum vertex coloring [9] and branchwidth [14] remain NP-Complete for CA graphs. We believe that boxicity belong to the second category.

A family $\mathcal{F}$ of subsets of a set X has the Helly property if for every subfamily $\mathcal{F}'$ of $\mathcal{F}$, with every two sets in $\mathcal{F}'$ pairwise intersecting, we also have $\bigcap_{A \in \mathcal{F}'} A \neq \emptyset$. Similarly, a family $\mathcal{A}$ of arcs satisfy Helly property if every subfamily $\mathcal{A}' \subseteq \mathcal{A}$ of pairwise intersecting arcs have a common intersection point. The fundamental difficulty while dealing with CA graphs in comparison with interval graphs is the absence of Helly property for a family of circular arcs arising out of their circular adjacencies.

A Proper Circular Arc (PCA) graph is a graph which has some CA representation in which no arc is properly contained in another. A Unit Circular Arc (UCA) graph is one which has a CA representation in which all arcs are of the same length. A Helly Circular Arc (HCA) graph is one which has a representation satisfying the Helly property. In a CA representation M, a pair of arcs are said to be circle cover arcs if they together cover the circumference of the circle. A Normal Circular Arc (NCA) graph is one which has a CA representation in which there are no pairs of circle cover arcs. It is known that UCA $\subsetneq$ PCA $\subsetneq$ NCA and UCA $\nsubseteq$ HCA $\nsubseteq$ NCA.

Our main results in this paper are:

(a) Boxicity of any circular arc graph can be approximated within a $(2 + \frac{1}{k})$-factor in polynomial time where $k \geq 1$ is the boxicity of the graph.
(b) The boxicity of any normal circular arc graph can be approximated within an additive 2-factor in polynomial time, given a normal circular arc model of the graph.
(c) The time complexity of the algorithms to approximately compute the boxicity is $O(mn + n^2)$ in both the above cases and in $O(mn + kn^2)$ which is at most $O(n^3)$ time we also get their corresponding box representations, where n is the number of vertices of the graph, m its number of edges and k its boxicity.

A structural result we obtained in this paper may be of independent interest. The following way of constructing an auxiliary graph H^* of a given graph H is from [1].

Definition 1. *Given a graph $H = (V, E)$, consider the graph H^* constructed as follows: $V(H^*) = E(H)$, and edges wx and yz of H are adjacent in H^* if and only if $\{w, x, y, z\}$ induces a $2K_2$ in H. Notice that H^* is the complement of $[L(H)]^2$, the square of the line graph of H.*

The structural properties of H^* and its complement $[L(H)]^2$ had been extensively investigated for various graph classes in the context of important problems like largest induced matching and minimum chain cover. The initial results were obtained by Golumbic et.al [10]. Cameron et.al [4] came up with some further results. A consolidation of the related results can be found in [4].

The following intermediate structural result in our paper becomes interesting in this context:

(d) In Lemma 4, we observe that if H is a bipartite graph whose complement is a CA graph, then H^* is a comparability graph.

This is a generalization of similar results for convex bipartite graphs and interval bigraphs already known in literature [1,25]. This observation helps us in reducing the complexity of our polynomial time algorithms.

The proofs and detailed description of the algorithms omitted from this paper are included in the full version [2].

2   Preliminaries

2.1   Notations

We denote the vertex set of a given graph G by $V(G)$ and edge set by $E(G)$, with $|V(G)| = n$ and $|E(G)| = m$. We use e to denote $\min(m, nC_2 - m)$. We denote the complement of G by $\overline{G}$. We call a graph G the union of graphs $G_1, G_2, \cdots, G_k$ if they are graphs on the same vertex set and $E(G) = E(G_1) \cup E(G_2) \cup \cdots \cup E(G_k)$. Similarly, a graph G is the intersection of graphs $G_1, G_2, \cdots, G_k$ if they are graphs on the same vertex set and $E(G) = E(G_1) \cap E(G_2) \cap \cdots \cap E(G_k)$. We use $box(G)$ to denote boxicity of G and $\chi(G)$ to denote chromatic number of G.

A circular-arc (CA) model $M = (C, \mathcal{A})$ consists of a circle C, together with a family $\mathcal{A}$ of arcs of C. It is assumed that C is always traversed in the clockwise direction, unless stated otherwise. The arc A_v corresponding to a vertex v is denoted by $[s(v), t(v)]$, where $s(v)$ and $t(v)$ are the extreme points of A_v on C with $s(v)$ its start point and $t(v)$ its end point respectively, in the clockwise direction. Without loss of generality, we assume that no single arc of $\mathcal{A}$ covers C and no arc is empty or a single point.

An interval model I consists of a family of intervals on real line. An interval I_v corresponding to a vertex v is denoted by a pair $\left[l_v(I), r_v(I)\right]$, where $l_v(I)$ and $r_v(I)$ are the left and right end points of the interval I_v. Without loss of generality, we assume that an interval is always non-empty and is not a single point. We may use I to represent both an interval graph and its interval model, when the meaning is clear from the context.

Definition 2 (Bi-Consecutive Adjacency Property). *Let the vertex set $V(G)$ of a graph G be partitioned into two sets A and B with $|A| = n_1$ and $|B| = n_2$. A numbering scheme where vertices of A are numbered as $1, 2, \cdots, n_1$ and vertices of B are numbered as $1', 2', \cdots, n'_2$ satisfy Bi-Consecutive Adjacency Property if the following condition holds:*
For any $i \in A$ and $j' \in B$, if i is adjacent to j', then either
(a) j' is adjacent to all k such that $1 \le k \le i$ or
(b) i is adjacent to all k' such that $1 \le k' \le j'$.

2.2   A Vertex Numbering Scheme for Circular Arc Graphs

Let G be a CA graph. Assume a CA model $M = (C, \mathcal{A})$ of G is given. Let p be any point on the circle C. We define a numbering scheme for the vertices of G denoted by $NS(M, p)$ which will be helpful for us in explaining further results.
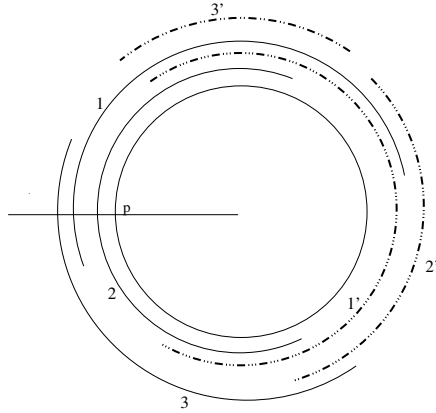
Fig. 1. Example for Numbering of vertices of a CA graph

Let A be the clique corresponding to the arcs passing through p and let $B = V \setminus A$. Let $|A| = n_1$ and $|B| = n_2$. Number the vertices in A as 1, 2, $\cdots$, n_1 such that the vertex v with its $t(v)$ farthest (in the clockwise direction) from p gets number 1 and so on. Similarly, number the vertices in B as $1'$, $2'$, $\cdots$, n_2' such that the vertex v' with its $t(v')$ farthest (in the clockwise direction) from p gets number $1'$ and so on. In both cases, break ties (if any) between vertices arbitrarily, while assigning numbers. See Figure 1 for an illustration of the numbering scheme. Now, observe that in G, if a vertex $i \in A$ is adjacent to a vertex $j' \in B$, then at least one of the following is true: (a) the point $t(i)$ is contained in the arc $[s(j'), t(j')]$ or (b) the point $t(j')$ is contained in the arc $[s(i), t(i)]$. This implies that if $i \in A$ is adjacent to $j' \in B$, then either (a) j' is adjacent to all k such that $1 \le k \le i$ or (b) i is adjacent to all k' such that $1 \le k' \le j'$. Thus we have the following lemma.

Lemma 1. *Given a circular arc graph G and a CA model $M(C, \mathcal{A})$ of G, together with a point p on the circle C, let A and B be as described above.*

1. *The numbering scheme $NS(M, p)$ of G defined above satisfy the Bi-Consecutive Adjacency Property.*
2. *$NS(M, p)$ can be computed in $O(n^2)$ time.*

Using Lemma 1, we can prove the following in the case of co-bipartite CA graphs.

Lemma 2. *If $G(V, E)$ is a co-bipartite CA graph, then we can find a partition $A \cup B$ of V where A and B induce cliques, having a numbering scheme of the vertices of A and B with $A = \{1, 2, \cdots, n_1\}$ and $B = \{1', 2', \cdots, n_2'\}$ such that it satisfies Bi-Consecutive Adjacency Property. Moreover, the numbering scheme can be found in $O(n^2)$ time.*

The following lemma is applicable in the case of co-bipartite graphs:

Lemma 3. *Let G be a co-bipartite graph with a partitioning of vertex set into cliques A and B with $|A| = n_1$ and $|B| = n_2$. Suppose there exist a numbering scheme of vertices of G which satisfies the Bi-Consecutive Adjacency Property. Then G is a CA graph.*

This can be proved by constructing a CA model for G.

3   Computing the Boxicity of Co-bipartite CA Graphs in Polynomial Time

Using some theorems in the literature, in this section we infer that computing boxicity of co-bipartite CA graphs can be done in polynomial time. A bipartite graph is chordal bipartite if it does not contain any induced cycle of length ≥ 6.

Theorem 1 (Feder, Hell and Huang 1999 [7]). *A graph G is a co-bipartite CA graph if and only if its complement is chordal bipartite and contains no edge-asteroids.*

A bipartite graph is called a chain graph if it does not contain any induced $2K_2$. The minimum chain cover number of G, denoted by $ch(G)$, is the minimum number of chain subgraphs of G such that the union of their edge sets is $E(G)$.

Recall Definition 1 of H^* from Section 1.

Theorem 2 (Abueida, Busch and Sritharan 2010 [1]). *If H is a bipartite graph with no induced cycles on exactly 6 vertices, then*

1. *$ch(H) = \chi(H^*)$.*
2. *Every maximal independent set of H^* corresponds to the edge-set of a chain subgraph of H. Moreover, the family of maximal independent sets obtained by extending the the color classes of the optimum coloring of H^* corresponds to a minimum chain cover of H.*
3. *In the more restricted case where H is chordal bipartite, H^* is a perfect graph and therefore, $ch(H)$ and a chain cover of H of minimum cardinality can be computed in polynomial time, in view of 1 and 2 above.*

Theorem 3 (Yannakakis 1982 [24]). *Let G be the complement of a bipartite graph H. Then, $box(G) = ch(H)$. Further, if $H_1, H_2, \cdots, H_k$ are chain subgraphs whose union is H, their respective complements $G_1, G_2, \cdots, G_k$ are interval supergraphs of G whose intersection is G.*

By Theorem 1, if $G = \overline{H}$ is a co-bipartite CA graph, then H is chordal bipartite. Hence by Theorem 2, a chain cover of H of minimum cardinality can be computed in polynomial time and $ch(H) = \chi(H^*)$. Combining with Theorem 3, we get :

Theorem 4. *If G is a co-bipartite CA graph, then $box(G) = \chi(H^*)$ and the family of maximal independent sets obtained by extending the color classes of an optimum coloring of H^* corresponds to the complements of interval supergraphs in an optimal box representation of G. Moreover, $box(G)$ and an optimal box representation of G are computable in polynomial time.*

4 Reducing the Time Complexity of Computing the Boxicity of Co-bipartite CA Graphs

Let t be the number edges of H or equivalently, the number of vertices in H^*. By Theorem 2, when H is a chordal bipartite graph, H^* is a perfect graph. Using the standard perfect graph coloring methods, an $O(t^3)$ algorithm is given in [1] to compute $\chi(H^*)$. In $O(t^3)$ time, they also compute a chain cover of minimum cardinality. However, $O(t^3)$ can be as bad as $O(n^6)$ in the worst case, where n is the number of vertices of G. In [1], for the restricted case when H is an interval bigraph, they succeeded in reducing the complexity to $O(tn)$, using the zero partitioning property of the adjacency matrix of interval bigraphs. Unfortunately, zero partitioning property being the defining property of interval bigraphs, we cannot use the method used in [1] in our case because of the following result by Hell and Huang [11]: A graph H is an interval bigraph if and only if its complement is a co-bipartite CA graph admitting a normal CA model. Since there are co-bipartite CA graphs which do not permit a normal CA model, the complements of CA co-bipartite graphs form a strict super class of interval bigraphs. Hence to bring down the complexity of the algorithm from $O(t^3)$, we have to go for a new method. The key ingredient of our method is the following generalization of the results in [1,25].

Lemma 4. *If the complement of graph H is a co-bipartite CA graph, then H^* is a comparability graph.*

Proof. Let $\overline{H} = G(V, E)$. Let $A \cup B$ be a partitioning of the vertex set V as described in Lemma 2, where A and B are cliques. Let $A = \{1, 2, \cdots, n_1\}$ and $B = \{1', 2', \cdots, n_2'\}$ be the associated numbering scheme.

Consider two adjacent vertices of H^* corresponding to the edges wx' and yz' of H. Since they are adjacent, $\{w, x', y, z'\}$ induces a $2K_2$ in H. Equivalently, these vertices induce a 4-cycle in G with edges wy, yx', $x'z'$ and $z'w$. We claim that $w < y$ if and only if $x' < z'$. To see this, assume that $w < y$. Since $yx' \in E(G)$, by the Bi-Consecutive property of the numbering scheme (Lemma 1), if $z' < x'$, $yz' \in E(G)$ or $wx' \in E(G)$, a contradiction. Hence, $x' < z'$.

Now, to show that H^* is a comparability graph, we define a relation $\prec$ as $ab' \prec cd'$ if and only if $a, c \in A$, $b', d' \in B$ with $a < c$ and $b' < d'$ and $\{a, b', c, d'\}$ induces a $2K_2$ in H. In view of the claim proved in the paragraph above, if ab' and cd' are adjacent vertices of H^*, they are comparable with respect to the relation $\prec$.

Let $ab' \prec cd'$ and $cd' \prec ef'$. We have $\{a, b', c, d'\}$ inducing a 4-cycle in G with edges ac, cb', $b'd'$ and $d'a$. Similarly, $\{c, d', e, f'\}$ induces a 4-cycle in G with edges ce, ed', $d'f'$ and $f'c$. We also have $a < c < e$ and $b' < d' < f'$, by the definition of the relation $\prec$. By the Bi-Consecutive property of the numbering scheme (Lemma 1), $cf' \in E(G)$ and $cd' \notin E(G)$ implies that $af' \in E(G)$. Similarly, $ed' \in E(G)$ and $cd' \notin E(G)$ implies that $eb' \in E(G)$. Edges ae and $b'f'$ are parts of cliques A and B. Hence, we have an induced 4-cycle in G with edges ae, eb', $b'f'$ and $f'a$. We can conclude that $ab' \prec ef'$. Thus the relation $\prec$ is transitive and hence, H^* is a comparability graph. $\qquad\square$

Improved Complexities

Lemma 4 serves as the key ingredient in improving the time complexities of our algorithms. By the definition of H^*, a proper coloring of the vertices of H^* is same as coloring the edges of H such that no two edges get the same color if their end points induce a $2K_2$ in H or equivalently a 4 cycle in G. Since the number of edges in H^* may be of $O(t^2)$, where $t = |E(H)|$, time for computing $\chi(H^*)$ might go up to $O(t^2) = O(n^4)$, if we use the standard algorithm for the vertex coloring of comparability graphs. Let $m_{AB} = n_1 n_2 - t$, the number of edges between A and B in G. We show that by utilizing the structure of G along with the underlying comparability relation on the set of non-edges of G defined in the proof of Lemma 4, computing the boxicity of G can be done in $O(en + n^2)$, where e is $\min(m_{AB}, t)$. Each color class can be extended to a maximal independent set and thus get an optimum box representation of G in $O(en + kn^2)$, where $k = box(G)$. The complexities claimed here are obtained by a suitable implementation of the greedy algorithm for the vertex coloring of comparability graphs, fine tuned for this special case and its careful amortized analysis. Due to the structural differences with interval bigraphs as explained before, this turned out to be much different from the method used in [1]. For a detailed description of the algorithm and its analysis, refer to the full version of our paper [2].

5   Constant Factor Approximation for the Boxicity of CA Graphs

First we give a lemma which is an adaptation of a similar one given in [3].

Lemma 5. *Let $G(V, E)$ be a graph with a partition (A, B) of its vertex set V with $A = \{1, 2, \cdots, n_1\}$ and $B = \{1', 2', \cdots, n_2'\}$. Let $G_1(V, E_1)$ be its supergraph such that $E_1 = E \cup \{(a', b') : a', b' \in B\}$. Then, $box(G_1) \leq 2 \cdot box(G)$.*

Definition 3. *Let $G(V, E)$ be an interval graph and I be an interval representation of G. Let $l = \min\limits_{u \in V} l_u(I)$ and $r = \max\limits_{u \in V} r_u(I)$. Consider a graph $G'(V', E')$ such that $V' \supseteq V$ and $E' = E \cup \{(a, b): a \in V' \setminus V \text{ and } b \in V'\}$. An interval representation I' of G' obtained by assigning interval $[l, r], \forall u \in V' \setminus V$ and intervals $[l_u(I), r_u(I)], \forall u \in V$ is called an extension of I on V'.*

Approximation Algorithm

A method for computing a box representation of a given CA graph G within a $(2 + \frac{1}{k})$-factor where $k \geq 1$ is the boxicity of G is given in Algorithm 1. We use the $O(en)$ algorithm for computing boxicity of co-bipartite CA graphs given in Section 3 as a subroutine here. Let $n = |V(G)|$ and $m = |E(G)|$. We can show that a near optimal box representation of G can be obtained in $O(mn + kn^2)$. If we just want to compute the approximate boxicity of G, it is enough to output $box(G') + 1$, as proved below. This can be done in $O(mn + n^2)$.

Proof of correctness: Let us analyze the non-trivial case when G is not an interval graph. Otherwise, the correctness is obvious.

Algorithm 1. Find a near optimal box representation of given CA graph

---

   Input: A circular arc graph $G(V, E)$
   Output: A box representation of G of dimension at most $2k + 1$ where
        $k = box(G)$
1 **if** G *is an interval graph* **then** Output an interval representation I_G of G, Exit
2 Compute a CA model $M(C, \mathcal{A})$ of G
3 Choose any point p on the circle C
4 Let A be the clique corresponding to p; $B = V \setminus A$
5 Construct $G'(V, E')$ with $E' = E \cup \{(u', v') : u', v' \in B\}$
            `/* `G'` is a co-bipartite CA graph by Lemma 6 */`
6 Find an optimum box representation $\mathcal{B}' = \{I'_1, I'_2, \cdots, I'_b\}$ of G'
             `/* Using the method described in Section 3 */`
7 Construct an interval representation I for the subgraph induced on B
          `/* Induced subgraph on B is clearly an interval graph */`
8 Construct I', the extension of I on V
9 Output $\mathcal{B} = \{I'_1, I'_2, \cdots, I'_b, I'\}$ as the box representation of G

Lemma 6. *G' constructed in Line 5 of Algorithm 1 is a co-bipartite CA graph.*

Proof. It can be easily seen that G' is a co-bipartite graph on the same vertex set as that of G with cliques A and B and $V = A \cup B$. Consider a numbering scheme $NS(M, p)$ of G as described in Section 2.2 such that $A = \{1, 2, \cdots, n_1\}$ and $B = \{1', 2', \cdots, n'_2\}$, based on the CA model $M(C, \mathcal{A})$ and the point p as chosen in Algorithm 1. Notice that by construction of G', for any pair of vertices $i \in A$ and $j' \in B$, $(u, v') \in E$ if and only if $(u, v') \in E'$. Recall that the numbering scheme $NS(M, p)$ satisfies Bi-Consecutive Adjacency Property for G by Lemma 1. Clearly, the same will apply to G' also. Hence by Lemma 3, we can infer that G' is a co-bipartite CA graph.   □

Lemma 7. *The box representation $\mathcal{B} = \{I'_1, I'_2, \cdots, I'_b, I'\}$, obtained in Line 9 of Algorithm 1 is a valid box representation of G with $|\mathcal{B}| \leq 2 \cdot box(G) + 1$.*

Proof. It is easy to see that I' constructed in Line 8 of Algorithm 1 is a super-graph of G, since I is an interval representation of the induced subgraph of G on B and I' is an extension of I on V. Since $\mathcal{B}'$ is a box representation of G', each $I'_i \in \mathcal{B}'$, for $1 \leq i \leq b$ is a supergraph of G' and in turn of G too. A is a clique in G by definition. Consider any $(u, v') \notin E$ with $u \in A$ and $v' \in B$. Clearly, $(u, v') \notin E'$ as well and since $\mathcal{B}'$ is a box representation of G', $\exists i$ such that $(u, v) \notin E(I'_i)$ for some $1 \leq i \leq b$. For any $(u', v') \notin E$ with $u', v' \in B$, we have $(u', v') \notin E(I')$. Thus, $G = I' \cap \bigcap_{1 \leq i \leq b} I'_i$.

   Thus, $\mathcal{B} = \{I'_1, I'_2, \cdots, I'_{b'}, I'\}$ is a valid box representation for G of size $box(G') + 1$. By Lemma 5, $box(G') \leq 2 \cdot box(G)$, implying that $\mathcal{B}$ is of size at most $2 \cdot box(G) + 1$.   □

Lemma 7 implies that $\mathcal{B}$ is a $(2 + \frac{1}{k})$-factor approximate box representation where $k \geq 1$ is the boxicity of G.

6 Additive 2-Factor Approximation for the Boxicity of Normal CA Graphs

We assume that a normal CA model $M(C, \mathcal{A})$ of G is given. An additive two factor approximation algorithm for computing a box representation of normal CA graphs is given in Algorithm 2. We can show that in $O(mn + kn^2)$ time, the algorithm outputs a near optimal box representation of G where $n = |V(G)|$, $m = |E(G)|$ and $k = box(G)$. If we just want to compute the approximate boxicity of G, it is enough to output $box(H) + 2$, as proved below. This can be done in $O(mn + n^2)$.

Algorithm 2. Find a additive 2-optimal box representation of given normal CA graph

---

Input: A normal CA graph $G(V, E)$, with an NCA model $M(C, \mathcal{A})$ of G
Output: A box representation of G of dimension at most $k + 2$ where
$\quad\quad k = box(G)$
1 **if** *G is an interval graph* **then** Output an interval representation I_G of G, Exit
2 Choose any point p on the circle C; Let A be the clique corresponding to p
3 Let p_1 be the farthest clockwise end point of any arc passing through p
4 Let p_2 be the farthest anticlockwise end point of any arc passing through p
5 Let q be a point on the arc $[p_1, p_2]$ with $q \neq p_1, p_2$
6 Let B be the clique corresponding to q
7 Let H be the induced subgraph on $A \cup B$
$\quad\quad\quad\quad$ /* Clearly, H is a co-bipartite CA graph */
8 Find an optimum box representation $\mathcal{B}' = \{I_1, I_2, \cdots, I_h\}$ of H
$\quad\quad\quad\quad$ /* Using the method described in Section 3 */
9 **for** $i = 1$ *to h* **do** Construct I'_i, the extension of I_i on V
10 Construct an interval representation I_A for the induced subgraph on $V \setminus A$
$\quad\quad\quad\quad$ /* Induced subgraph on $V \setminus A$ is an interval graph */
11 Construct I'_A, the extension of I_A on V
12 Construct an interval representation I_B for the induced subgraph on $V \setminus B$
$\quad\quad\quad\quad$ /* Induced subgraph on $V \setminus B$ is an interval graph */
13 Construct I'_B, the extension of I_B on V
14 Output $\mathcal{B} = \{I'_1, I'_2, \cdots, I'_h, I'_A, I'_B\}$ as the box representation of G

Proof of correctness: Since G is a normal CA graph, the set of arcs passing through p does not contain any circle cover pair of arcs. Therefore, $[p, p_1] \cup [p_2, p]$ does not cover the entire circle C. So, any point in the arc (p_1, p_2), in particular the point q defined in Line 5 of Algorithm 2, is not contained in any arc passing through p. It follows that $A \cap B = \emptyset$. Since A and B are cliques, H, the induced subgraph on $A \cup B$ is a co-bipartite CA subgraph of G. We can compute an optimum box representation $\mathcal{B}'$ of H in polynomial time using the method described in Section 3.

I_A and I_B are interval graphs because they are obtained by removing vertices corresponding to arcs in $\mathcal{A}$ passing through points p and q respectively. Since I_A is a supergraph of G on $V \setminus A$ and I'_A is the extension of I_A on V, we can conclude that I'_A is a super graph of G. Similarly, I'_B is also a super graph of G.

Since $\mathcal{B}'$ is a box representation of H, each $I_i \in \mathcal{B}'$ is a supergraph of induced subgraph H. Since I_i' is the extension of I_i on V, I_i' is a super graph of G.

Consider $(u,\ v) \notin E$. **Case (i)** If $u,\ v \in V \setminus A$, by construction of I_A', $(u,v) \notin E(I_A')$. **Case (ii)** If $u,\ v \in V \setminus B$, by construction of I_B', $(u,v) \notin E(I_B')$. Remember that A and B are cliques. If both (i) and (ii) are false, then one of $\{u,\ v\}$ is in A and the other is in B. Since $\mathcal{B}'$ is a box representation of H, $(u,v) \notin E(I_i)$ for some $1 \leq i \leq h = |\mathcal{B}'|$. By construction of I_i', $(u,\ v) \notin E(I_i')$ too. Hence, $G = I_A' \cap I_B' \cap \bigcap\limits_{1 \leq i \leq h} I_i'$. Thus we get $\mathcal{B} = \{I_A',\ I_B',\ I_1',\ I_2',\ \cdots,\ I_h'\}$ is a valid box representation of G of size $box(H) + 2$ which is at most $box(G) + 2$, since H is an induced subgraph of G.

In Algorithm 2, we assumed that an NCA model of the graph is given. This was required because recognizing NCA graphs in polynomial time is still an open problem. We can observe that though the algorithm of this section is given for normal CA graphs, it can be used for a wider class as stated below.

Theorem 5. *If we are given a circular arc model $M(C,\ \mathcal{A})$ of G with a point p' on the circle C such that the set of arcs passing through p' does not contain a circle cover pair, then we can approximate the boxicity of G within an additive 2-factor in polynomial time using Algorithm 2.*

Proof. In Line 2 of Algorithm 2, select p' (guaranteed by the assumption of the theorem) as the point p. Such a point can be found in $O(n^2)$ time, if it exists. The rest of the algorithm is similar. □

Though such a representation need not exist in general, it does exist for many important subclasses of of CA graphs and can be constructed in polynomial time; for example, for proper CA graphs or normal helly CA graphs. In fact, for these classes, construction of a normal CA (NCA) model itself from their adjacency matrices can be done in polynomial time.

Corollary 1. *Boxicity of any proper circular arc graph can be approximated within an additive 2-factor in polynomial time.*

References

1. Abueida, A.A., Busch, A.H., Sritharan, R.: A min-max property of chordal bipartite graphs with applications. Graphs and Combinatorics 26(3), 301–313 (2010)
2. Adiga, A., Babu, J., Chandran, L.S.: A constant factor approximation algorithm for boxicity of circular arc graphs. In: CoRR, abs/1102.1544 (February 2011), http://arxiv.org/abs/1102.1544
3. Adiga, A., Bhowmick, D., Sunil Chandran, L.: The hardness of approximating the boxicity, cubicity and threshold dimension of a graph. Discrete Appl. Math. 158, 1719–1726 (2010)
4. Cameron, K., Sritharan, R., Tang, Y.: Finding a maximum induced matching in weakly chordal graphs. Discrete Math. 266, 133–142 (2003)
5. Chandran, L.S., Das, A., Shah, C.D.: Cubicity, boxicity, and vertex cover. Discrete Mathematics 309(8), 2488–2496 (2009)

6. Cozzens, M.B.: Higher and multi-dimensional analogues of interval graphs. Ph.D. thesis, Department of Mathematics. Rutgers University, New Brunswick, NJ (1981)
7. Feder, T., Hell, P., Huang, J.: List homomorphisms and circular arc graphs. Combinatorica 19, 487–505 (1999)
8. Gallai, T.: On directed paths and circuits. In: Erdös, P., Katona, G. (eds.) Theory of Graphs, pp. 115–118. Academic Press, New York (1968)
9. Garey, M.R., Johnson, D.S., Miller, G.L., Papadimitriou, C.H.: The complexity of coloring circular arcs and chords. SIAM J. Alg. Disc. Meth. 1(2), 216–227 (1980)
10. Golumbic, M.C., Lewenstein, M.: New results on induced matchings. Discrete Appl. Math. 101, 157–165 (2000)
11. Hell, P., Huang, J.: Interval bigraphs and circular arc graphs. J. Graph Theory 46, 313–327 (2004)
12. Kratochvíl, J.: A special planar satisfiability problem and a consequence of its NP-completeness. Discrete Appl. Math. 52(3), 233–252 (1994)
13. Lin, M.C., Szwarcfiter, J.L.: Characterizations and recognition of circular-arc graphs and subclasses: A survey. Discrete Mathematics 309(18), 5618–5635 (2009)
14. Mazoit, F.: The branch-width of circular-arc graphs. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006. LNCS, vol. 3887, pp. 727–736. Springer, Heidelberg (2006)
15. McConnell, R.M.: Linear-time recognition of circular-arc graphs. Algorithmica 37(2), 93–147 (2003)
16. Roberts, F.S.: On the boxicity and cubicity of a graph. In: Recent Progresses in Combinatorics, pp. 301–310. Academic Press, New York (1969)
17. Rosgen, B., Stewart, L.: Complexity results on graphs with few cliques. Discrete Mathematics and Theoretical Computer Science 9, 127–136 (2007)
18. Scheinerman, E.R.: Intersection classes and multiple intersection parameters of graphs. Ph.D. thesis. Princeton University (1984)
19. Suchan, K., Todinca, I.: Pathwidth of circular-arc graphs. In: Brandstädt, A., Kratsch, D., Müller, H. (eds.) WG 2007. LNCS, vol. 4769, pp. 258–269. Springer, Heidelberg (2007)
20. Sundaram, R., Singh, K.S., Rangan, C.P.: Treewidth of circular-arc graphs. SIAM J. Discret. Math. 7, 647–655 (1994)
21. Thomassen, C.: Interval representations of planar graphs. J. Comb. Theory Ser. B 40, 9–20 (1986)
22. Tucker, A.C.: Matrix characterizations of circular-arc graphs. Pacific J. of Mathematics 19, 535–545 (1971)
23. Tucker, A.C.: Structure theorems for some circular-arc graphs. Discrete Mathematics 7(1,2), 167–195 (1974)
24. Yannakakis, M.: The complexity of the partial order dimension problem. SIAM J. Alg. Disc. Meth. 3(3), 351–358 (1982)
25. Yu, C.W., Chen, G.H., Ma, T.H.: On the complexity of the -chain subgraph cover problem. Theor. Comput. Sci. 205(1-2), 85–98 (1998)

On the Area Requirements of
Euclidean Minimum Spanning Trees[*]

Patrizio Angelini[1], Till Bruckdorfer[2], Marco Chiesa[1],
Fabrizio Frati[1,3], Michael Kaufmann[2], and Claudio Squarcella[1]

[1] Dipartimento di Informatica e Automazione, Università Roma Tre
{angelini,chiesa,frati,squarcel}@dia.uniroma3.it
[2] Wilhelm-Schickard-Institut für Informatik - Universität Tübingen, Germany
{bruckdor,mk}@informatik.uni-tuebingen.de
[3] School of Basic Sciences - École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. In their seminal paper on Euclidean minimum spanning trees *[Discrete & Computational Geometry, 1992]*, Monma and Suri proved that any tree of maximum degree 5 admits a planar embedding as a Euclidean minimum spanning tree. Their algorithm constructs embeddings with exponential area; however, the authors conjectured that $c^n \times c^n$ area is sometimes required to embed an n-vertex tree of maximum degree 5 as a Euclidean minimum spanning tree, for some constant $c > 1$. In this paper, we prove the first exponential lower bound on the area requirements for embedding trees as Euclidean minimum spanning trees.

1   Introduction

A *Euclidean minimum spanning tree* (MST) of a set P of points in the plane is a tree with a vertex in each point of P and with minimum total edge length. Euclidean minimum spanning trees have several applications in computer science and hence they have been deeply investigated from a theoretical point of view. To cite a few major results, optimal $\Theta(n \log n)$-time algorithms are known to compute an MST of a set of points and it is $\mathcal{NP}$-hard to compute an MST with maximum degree bounded by 2, 3, or 4 [7,9,17], while polynomial-time algorithms exist [2,4,11,15] to compute spanning trees with maximum degree bounded by 2, 3, or 4 and total edge length within a constant factor from the optimal one.

An *MST embedding* of a tree T is a plane embedding of T such that the MST of the points where the vertices of T are drawn coincides with T. In this paper we consider the problem of constructing MST embeddings of trees. Several results are known related to such a problem. No tree having a vertex of degree at least 7 admits an MST embedding. Further, deciding whether a tree with degree 6 admits an MST embedding is $\mathcal{NP}$-hard [6]. However, restricting the attention to trees of degree 5 is not a limitation since:

(i) every planar point set has an MST with maximum degree 5 [16], and (ii) every tree of maximum degree 5 admits an MST embedding in the plane [16].

MST embeddings have also been considered as a subtopic of proximity drawings (see, e.g., [3,13]), where adjacent vertices have to be placed "close" to each other. From the various applications, different measures for closeness have been introduced and evaluated, different graph classes that can be realized under the specific closeness measure have been studied together with the area/volume that is needed for their realization. Prominent examples are Delaunay triangulations, Gabriel drawings, nearest-neighbor drawings, β-drawings, and many more. The MST constraints can be formulated as closeness conditions with respect to pairs of vertices, either adjacent or non-adjacent.

Monma and Suri's proof [16] that every tree of maximum degree 5 admits an MST embedding in the plane is a strong combinatorial result; on the other hand, their algorithm for constructing MST embeddings seems to be useless in practice, since the constructed embeddings have $2^{\Theta(k^2)}$ area for trees of height k (hence, in the worst case the area requirement of such drawings is $2^{\Theta(n^2)}$). However, Monma and Suri conjectured that there exist trees of maximum degree 5 that require $c^n \times c^n$ area in *any* MST embedding, for some constant $c > 1$. The problem of determining whether or not the area upper bound for MST embeddings of trees can be improved to polynomial is reported also in [5,6,10,14]. Recently, MST embeddings in polynomial area have been proven to exist for trees with maximum degree 4 [8,12].

In this paper, we prove that there exist n-vertex trees of maximum degree 5 requiring $2^{\Omega(n)}$ area in any MST embedding. Our lower bound is achieved by considering an n-vertex tree T^*, shown in Fig. 1, composed of a degree-5 *complete tree* T_c with a constant number of vertices and of a set of degree-5 *caterpillars*, each one attached to a distinct leaf of T_c. The argument goes in two steps: For the first step, we walk down the tree T^*, starting from the root. The route is chosen so that the angles adjacent to the edges are narrowing at each step. The key observation here is a lemma relating the size of two consecutive angles adjacent to an edge. At the leaves of the complete tree T_c, where the caterpillars start, the angles incident to an end-vertex of the backbone of at least one of the caterpillars must be very small, that is, between $60°$ and $61°$. Using this as a starting point, we prove that each angle incident to a vertex of the caterpillar is either very small, that is, between $60°$ and $61°$, or is very large, that is, between $89.5°$ and $90.5°$. As a consequence, we show that when walking down along the backbone of the caterpillar, the lengths of the edges decrease exponentially along the caterpillar. Since the backbone has a linear number of edges, we obtain the claimed area bound.

The paper is organized as follows. In Sect. 2 we give definitions and preliminaries; in Sect. 3 we give some geometric lemmata; in Sect. 4 we argue about angles and edge lengths of the MST embeddings of T^*; in Sect. 5 we prove the area lower bound; in Sect. 6, we conclude with some remarks and a conjecture. Some proofs have been omitted for space limitations and can be found in the extended version of this paper [1].

2   Preliminaries

A *rooted tree* is a tree with one distinguished vertex, called *root*. The *depth* of a vertex in a rooted tree is its distance from the root, that is, the number of edges in the path
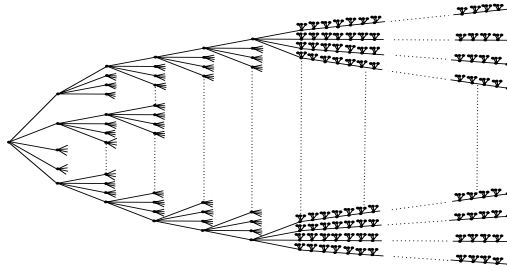
Fig. 1. A tree T^* requiring $2^{\Omega(n)}$ area in any MST embedding

from the root to the vertex. The *height* of a rooted tree is the maximum depth of one of its vertices. A *complete tree* is such that every path from the root to a leaf has the same number of vertices and every vertex has the same degree. A *caterpillar* is a tree such that removing the leaves yields a path, called the *backbone* of the caterpillar.

A *minimum spanning tree* MST of a planar point set P is a tree spanning P with minimum total edge length. Given a tree T, an *MST embedding* of T is a straight-line drawing of T such that the MST of the points where the vertices of T are drawn is isomorphic to T. The *area* of an MST embedding is the area of a rectangle enclosing such an embedding. The concept of area of an MST embedding only makes sense once a *resolution rule* is fixed, i.e., a rule that does not allow vertices to be arbitrarily close (*vertex resolution rule*), or edges to be arbitrarily short (*edge resolution rule*). Without any of such rules, one could just construct MST embeddings with arbitrarily small area. In the following we will hence suppose that any two vertices have distance at least one unit. Then, in order to prove that an n-vertex tree T requires $\Omega(f(n))$ area in any MST embedding, it suffices to prove that the ratio between the longest and the shortest edge of any MST embedding is $\Omega(f(n))$, and that both dimensions have at least $\Omega(1)$ size.

Consider any MST embedding of a tree T rooted at a vertex r. The *clockwise path* $Cl(u)$ of a vertex $u \neq r$ of T is the path $v_0, \ldots, v_k$ such that $v_0 = u$, (v_i, v_{i+1}) is the edge following the edge from v_i to its parent in the clockwise order of the edges incident to v_i, for $i = 0, \ldots, k-1$, and v_k is a leaf. The *counterclockwise path* $Ccl(u)$ of a vertex $u \neq r$ of T is defined analogously. Denote by $d(a, b)$ the Euclidean distance between two vertices a and b (or between two points a and b) and by $|e|$ the Euclidean length of an edge e. Further, $k(c, r)$ denotes the circle centered at a point c with radius r.

Next, we define an n-vertex tree T^* that requires $\Omega(2^n)$ area in any MST embedding. Let T_c be a complete tree of height six and degree five. Let r be the root of T_c. Augment T_c by inserting a degree-five caterpillar at each leaf of T_c. That is, for each leaf l of T_c, insert a caterpillar C_l whose every non-leaf vertex has degree five, such that l is an end-vertex of the backbone of C_l, the parent of l in T_c is a leaf of C_l, and C_l and T_c do not share any other vertex. The resulting tree, shown in Fig. 1, is denoted by T^*.

3   Geometric Lemmata

We give some properties for MST embeddings. The first lemma is easy to prove.

Lemma 1. *Consider an MST embedding Γ of a tree T. Then, the following statements hold: (i) for each pair of vertices u and v of T, $d(u,v) \geq |e|$, for each edge e in the path connecting u and v in T; (ii) any angle between two adjacent segments in Γ is at least $60°$; (iii) for any subtree T' of T, Γ restricted to the vertices and edges of T' is an MST embedding of T'; and (iv) Γ is planar.*

The next lemma bounds the length of an edge in an MST embedding in terms of the length of an adjacent edge and of the size of the angle between them.

Lemma 2. *Let e_1 and e_2 be two edges consecutively incident to the same vertex and let $\alpha \leq 90°$ be the angle they form. Then, $2|e_1| \cos(\alpha) \leq |e_2| \leq \frac{|e_1|}{2\cos(\alpha)}$.*

Consider an edge $e = (u,v)$ in an MST embedding of a tree T. Let $e_1 = (u,p)$ be the edge following e in the counterclockwise order of the edges incident to u and $e_1' = (v,q)$ be the edge following e in the clockwise order of the edges incident to v. Let α (β) be the angle defined by a counterclockwise (resp. clockwise) rotation of e around u (resp. around v) bringing e to coincide with e_1 (resp. with e_1'). Let $\phi = \frac{|e|}{|e_1|}$. The next lemma, that establishes a strong lower bound on β provided that α is sufficiently small, is one of our main tools for the remainder of the paper.

Lemma 3. *Suppose that $\alpha \leq 80°$. Then, $\beta \geq 120° - \alpha/2$.*

Proof: First, we determine restrictions on the region where q lies, once the drawings of e and e_1 are fixed. Refer to Fig. 2(a). Suppose, w.l.o.g., that e is horizontal, that u is at point $(0,0)$, that v is to the right of u, and that both p and q are above the horizontal line through u and v. We can suppose that q is to the left of the vertical line l_v through v, since otherwise $\beta \geq 90° \geq 120° - \alpha/2$, where the last inequality holds by Lemma 1(ii), and there is nothing to prove. By Lemma 1(i), $d(q,u) \geq d(u,v)$ holds. Then, q is outside $k(u, |e|)$. Still by Lemma 1(i), $d(p,q) \geq d(p,u)$ and $d(p,q) \geq d(u,v)$ hold. Then, q is outside $k(p,m)$, where $m = \max\{|e|, |e_1|\}$. Again by Lemma 1(i), $d(p,q) \geq d(v,q)$ holds. Denote by $l_{pv}^|$ the line orthogonal to $\overline{pv}$ passing through the midpoint of $\overline{pv}$; then, q is in the half-plane delimited by $l_{pv}^|$ and not containing p. Denote by t and n the intersections of $l_{pv}^|$ with $k(p,m)$ and l_v, respectively. Denote by l_β the line passing through v and creating with $\overline{uv}$ the angle $120° - \frac{\alpha}{2}$.

Second, we discuss about the intersections of $k(p,m)$ with l_v. The distance from p to l_v is less than $|e|$, because p is to the right of the vertical line through u, given that $\alpha \leq 80°$. It follows that $k(p,m)$ has exactly two intersections with l_v, given that $m \geq |e|$. Moreover both of such intersections lie not below v as the distance between p and v is at least m, by Lemma 1(i), and hence the distance between p and any point of l_v below v is strictly greater than m, while $k(p,m)$ has radius exactly m. Denote by h and b the highest and the lowest of such two intersection points, respectively.

Third, we prove that for any $\alpha \leq 80°$ the region R_2, bounded (when existing) by l_v from the right, by $k(p,m)$ from the left and by $l_{pv}^|$ from above, either does not exist or falls on the right of the line l_β. We distinguish two cases, based on whether $\phi \leq 1$, that is, $|e_1| \geq |e|$, or not.

In the former case, consider a segment $\overline{vw}$ parallel to e_1 such that $|e_1| = |\overline{vw}|$. See Fig. 2(b). Observe that $|\overline{pw}| = |e|$. Consider triangle $\Delta(puv)$. As $|e_1| \geq |e|$, we have
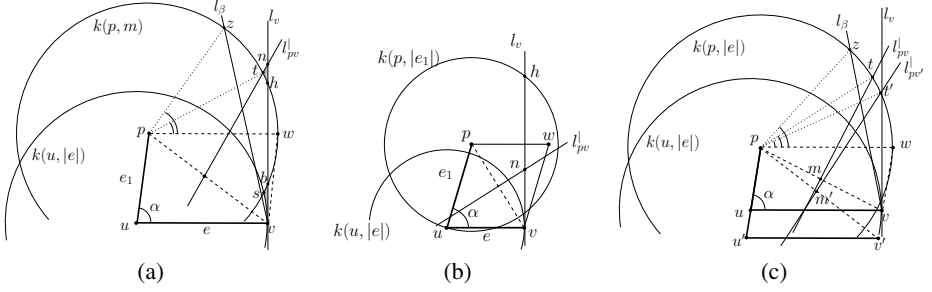
Fig. 2. (a) Illustration for the proof of Lemma 3. The lower and upper shaded regions are respectively R_1 and R_2 where q can lie. (b) When $\phi \leq 1$, region R_2 does not exist. (c) When $\phi > 1$, region R_2 is always avoided.

that $\widehat{pvu} \geq \widehat{vpu}$. Hence, the line $l_{pv}^{|}$ orthogonal to $\overline{pv}$ passing through its midpoint crosses segment $\overline{up}$. Analogously, in triangle $\triangle(pwv)$, line $l_{pv}^{|}$ crosses segment $\overline{vw}$. Hence, $l_{pv}^{|}$ cuts polygon (u, v, w, p) by crossing segments $\overline{up}$ and $\overline{vw}$, which implies that point n lies inside (u, v, w, p). As segment $\overline{pw}$ cuts l_v below h, we have that $|\overline{vn}| \leq |\overline{vh}|$, therefore region R_2 does not exist.

In the latter case, that is, when $\phi > 1$, we first prove the statement when ϕ assumes its maximum possible value, that is, by Lemma 2, $\phi = \frac{1}{2\cos\alpha}$; then, we extend the same statement to all the other values of ϕ greater than 1. Observe that, if any point of R_2 is to the left of l_β, then t is to the left of l_β. Hence, in order to prove that R_2 is entirely to the right of l_β, it suffices to prove that $\widehat{uvt} \geq 120° - \alpha/2$.

Suppose that $\phi = \frac{1}{2\cos\alpha}$. Then, by Lemma 2, triangle $\triangle(uvp)$ is isosceles, with the two equal-length sides being $\overline{uv}$ and $\overline{pv}$. Hence, triangle $\triangle(pvt)$ is equilateral, as $\overline{pt}$ is a radius of $k(p, |e|)$ and t is a point of $l_{pv}^{|}$. Therefore, $\widehat{uvp} = 180° - 2\alpha$ and $\widehat{pvt} = 60°$. Since $\widehat{uvt} = \widehat{uvp} + \widehat{pvt}$, we have that $\widehat{uvt} = 180° - 2\alpha + 60° = 240° - 2\alpha$. Since $240° - 2\alpha \geq 120° - \frac{\alpha}{2}$ holds for any $\alpha \leq 80°$, region R_2 is entirely to the right of l_β, provided that $\phi = \frac{1}{2\cos\alpha}$.

Now we extend the proof to the general case, in which $1 < \phi < \frac{1}{2\cos\alpha}$. See Fig. 2(c). Let w be the right-most point of $K(p, |e|)$. Note that points $p,w,v,$and u form a parallelogram. Note also that line $l_{pv}^{|}$ crosses segment $\overline{pw}$, since $|e| > |e_1|$; moreover the slope of $l_{pv}^{|}$ is positive, which guarantees that $0° \leq \widehat{tpw} \leq 90°$.

Let z be the intersection point between l_β and $k(p, |e|)$. Observe that, since t lies in the first quadrant of $k(p, |e|)$, since line l_β has a negative slope, and since line $l_{pv}^{|}$ has a positive slope, we have that t is to the right of l_β, that is, $\widehat{uvt} \geq 120° - \frac{\alpha}{2}$, if and only if $\widehat{zpw} \geq \widehat{tpw}$. Note that for the already studied case $\phi = \frac{1}{2\cos\alpha}$, indeed we have $\widehat{zpw} \geq \widehat{tpw}$. Hence, it suffices to show that, for any α, given two values ϕ and ϕ' such that $\phi' < \phi$ (or equivalently $|\overline{pu'}| > |\overline{pu}|$), we have $\widehat{tpw} > \widehat{t'p'w'}$.

Suppose, w.l.o.g., that $|\overline{u'v'}| = |\overline{uv}|$, that point p and p' coincide, and that segments $\overline{pu'}$ and $\overline{pu}$ lie on the same line. Note that m and m', respectively midpoints of segments $\overline{pv}$ and $\overline{p'v'}$ lie on a line which is parallel to $\overline{up}$ by construction. Moreover, m' lies below

m and the slope of $l^|_{pv'}$ is smaller than the one of $l^|_{pv}$. This implies that t' lies to the right of t and therefore $\widehat{tpw} > \widehat{t'p'w'}$. Thus, R_2 lies entirely to the right of l_β.

Finally, we prove the claimed lower bound for β by defining the remaining region R_1 in which q can lie, and showing that it always falls on the right of l_β. Region R_1 is bounded by l_v from the right, by $k(u, |e|)$ from the left, and either by $k(p, m)$ or by $l^|_{pv}$ from above (depending on whether n is higher or lower than b). Hence, such a region is a subset of the region bounded by l_v from the right, by $k(u, |e|)$ from the left, and by $k(p, m)$ from above. Then, denoting by s the intersection point between $k(p, m)$ and $k(u, |e|)$, we have $\beta \geq \widehat{uvs}$. Then, it suffices to show that $\widehat{uvs} \geq 120° - \alpha/2$. Denote by γ the angle $\widehat{vus}$. Then, we have $s \equiv (|e|\cos\gamma, |e|\sin\gamma)$ and $\widehat{uvs} = \frac{180° - \gamma}{2}$, where the last equality uses the fact that $|\overline{us}| = |\overline{uv}|$. Observe also that $p \equiv (|e_1|\cos\alpha, |e_1|\sin\alpha)$. We further distinguish two cases, namely the one in which $|e| \geq |e_1|$ (Case 1) and the one in which $|e_1| \geq |e|$ (Case 2).

Suppose that we are in Case 1. Then, there are two isosceles triangles, $\Delta(suv)$ and $\Delta(sup)$. Consider the triangle $\Delta(sup)$: its two equal-length sides, $\overline{us}$ and $\overline{ps}$, have length $|e|$ which is larger than the third side, which has length $|e_1|$. Thus we have $\widehat{sup} \geq 60°$. Since $\widehat{sup} = \alpha - \gamma$ we have $\gamma \leq \alpha - 60°$. At the same time $\widehat{uvs}$, $\widehat{usv} = \widehat{uvs}$, and γ are the angles in $\Delta(suv)$ and therefore sum up to $180°$. This shows that $\widehat{uvs} \geq 120° - \alpha/2$.

Case 2 is analogous to Case 1. The side lengths on the triangle $\Delta(sup)$ change: it remains an isosceles triangle, but now the two equal sized segments are $\overline{pu}$ and $\overline{ps}$, both with length $|e_1|$. The third side is shorter ($|e|$) and hence the angle $\widehat{sup}$ is again larger than $60°$. So we can argue as in Case 1. Hence, Lemma 3 holds. $\qquad\square$

4   Angles and Edge Lengths in MST Embeddings

In this section we argue about the angles and the edge lengths in each MST embedding of T^*. We start by providing a lemma about the complete tree T_c.

Lemma 4. *A vertex u of T_c with depth five exists such that there are two angles α_0 and α'_0 consecutively incident to u and not adjacent to the edge from u to its parent with $\alpha_0 + \alpha'_0 \leq 121°$.*

Consider any MST embedding of T^*; by Lemma 4, there exists a caterpillar C^* such that one of the end-vertices u_0 of the backbone of C^* is incident to an edge of T_c that is adjacent to two angles α_0 and α'_0 summing up to at most $121°$. Denote by $u_0, u_1, \ldots, u_k$ the vertices of the backbone of C^* and by e_i the backbone edge (u_i, u_{i+1}), for $i = 0, \ldots, k-1$. We call *outgoing angles α_i and α'_i* the angles adjacent to e_i and incident to u_i; we call *incoming angles β_{i+1} and β'_{i+1}* the angles adjacent to e_i and incident to u_{i+1}. An edge e incident to u_i and different from e_{i-1} is *in position $j \in \{1, 2, 3, 4\}$* if e is the j-th edge in the clockwise order of the edges incident to u_i starting at e_{i-1}. Note that, if e_i is in position 1 (resp. 4), the incoming angle β_i and the outgoing angle α_i (resp. the incoming angle β'_i and the outgoing angle α'_i) coincide. See Fig. 3. We prove that the outgoing and the incoming angles incident to a vertex of the backbone of C^* are either *small angles*, that is, between $60°$ and $61°$, or *large angles*, that is
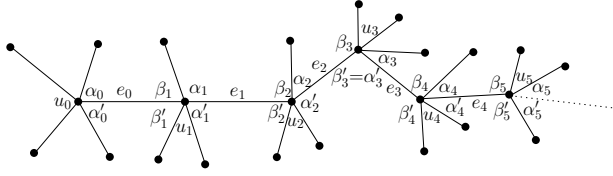
Fig. 3. An embedding of C^*

between $89.5^°$ and $90.5^°$. More precisely, the incoming angles are always large, while the outgoing angles are either both small or one large and one small. Indeed, observe that the outgoing angles of u_0 are both small by Lemma 4.

Suppose that a backbone edge e_i is in position 2 or 3 and that the incoming angles of u_i are at least $89.5^°$. By Lemma 1(ii), each of the outgoing angles of u_i is at most $61^°$. Then, by Lemma 3, the incoming angles of u_{i+1} are at least $89.5^°$. Hence, if e_i is in position 2 or 3 and the incoming angles of u_i are at least $89.5^°$, the incoming angles of u_{i+1} are also at least $89.5^°$.

If e_i is in position 1 or 4, then one outgoing angle of u_i, say α_i, coincides with one incoming angle of u_i, say β_i. Hence, $\alpha_i = \beta_i$ is large and no lower bound for β_{i+1} can obtained by Lemma 3. However, we can prove that, even if α_i is large, angle β_{i+1} is large, provided that the following condition is satisfied: The clockwise path $Cl(u_i)$ of u_i lies in a bounded region R_i that is a subset of a wedge W_i with angle $1^°$ centered at u_i. We will later prove (in Lemma 10) that, if such a condition is satisfied by a node u_i incident to a large outgoing angle α_i, then β_{i+1} is large and $Cl(u_{i+1})$ lies in a bounded region R_{i+1} that is a subset of a wedge W_{i+1} with angle $1^°$ centered at u_{i+1}. However, before that, we prove that such a condition is satisfied by a node u_i if α_{i-1} is small.

Suppose, w.l.o.g., that e_{i-1} is horizontal, with u_i to the right of u_{i-1}, and that e_i is in position 1. Denote by $e = (u_{i-1}, v)$ (by $e^* = (u_{i+1}, w)$) the edge following e_{i-1} (resp. e_i) in the counterclockwise (resp. clockwise) order of the edges incident to u_{i-1} (resp to u_{i+1}). Denote by $l(\underline{\alpha_i})$ (by $l(\overline{\alpha_i})$) the half-line with slope $90.5^°$ (resp. with slope $89.5^°$) starting at u_i. Denote by W_i the closed wedge with angle $1^°$ delimited by $l(\underline{\alpha_i})$ and $l(\overline{\alpha_i})$. See Fig. 4.

We will bound the region in which $Cl(u_i)$ lies from the right, from the left, and from above. Let $m = \max\{|e|, |e_{i-1}|\}$. Concerning the bound from the left, we can prove that the intersection point s of the circles $k(v, m)$ and $k(u_{i-1}, |e_{i-1}|)$ is not to the left of $l(\underline{\alpha_i})$, as stated in the following.

Lemma 5. *Suppose that $\alpha_{i-1} \leq 61^°$. Then, s is not to the left of $l(\underline{\alpha_i})$.*

We continue with the bound from the right.

Lemma 6. *Suppose that $\beta_i' \geq 89.5^°$. Then vertex u_{i+1} is not to the right of $l(\overline{\alpha_i})$.*

To derive the bound from above, we prove that $k(v, m)$ intersects $l(\overline{\alpha_i})$ twice and we argue about the distance between u_i and the highest intersection point $h_{\overline{\alpha_i}}$ of $k(v, m)$ with $l(\overline{\alpha_i})$.

Lemma 7. *Suppose that $\alpha_{i-1} \leq 61^°$. Then, $k(v, m)$ intersects $l(\overline{\alpha_i})$ twice.*
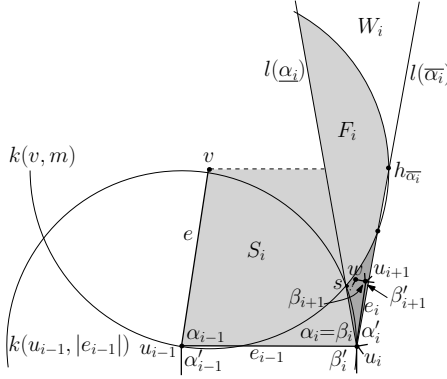
Fig. 4. The setting for Lemmata 5–9. The dark-shaded region is R_i. To improve the readability, angles and edge lengths in the illustration do not correspond to actual angles and edge lengths.

Lemma 8. *The distance between u_i and $h_{\overline{\alpha_i}}$ is at least $1.604|e_{i-1}|$.*

We are now ready to state the following:

Lemma 9. *Suppose that $\alpha_{i-1} \leq 61°$, that $\beta'_i, \beta'_{i+1} \geq 89.5°$, and that $|e_i| \leq \frac{|e_{i-1}|}{10}$. Then, $Cl(u_i)$ is inside a bounded region R_i that is a subset of W_i.*

Proof: Let R_i be the bounded region delimited by $l(\underline{\alpha_i})$ from the left, by $l(\overline{\alpha_i})$ from the right, and by $k(v,m)$ from above. We prove that $Cl(u_i)$ is inside R_i.

First, we prove that u_{i+1} is in R_i. By the assumption that $\alpha_{i-1} \leq 61°$ and by Lemma 3, u_{i+1} is not to the left of $l(\underline{\alpha_i})$. By the assumption that $\beta'_i \geq 89.5°$ and by Lemma 6, u_{i+1} is not to the right of $l(\overline{\alpha_i})$. Hence, u_{i+1} is in W_i. By the assumption that $\alpha_{i-1} \leq 61°$ and by Lemma 7, $k(v,m)$ intersects $l(\overline{\alpha_i})$. We now show that v is to the left of $l(\underline{\alpha_i})$. Namely, $v \equiv (|e|\cos\alpha_{i-1}, |e|\sin\alpha_{i-1})$. Further, if $y = |e|\sin\alpha_{i-1}$, then the x-coordinate of $l(\underline{\alpha_i})$ is $x = |e_{i-1}| - (|e|\sin\alpha_{i-1})/\tan 89.5°$. Since $|e_{i-1}| \geq 2|e|\cos\alpha_{i-1}$ (by Lemma 2) and $60° \leq \alpha_{i-1} \leq 61°$ (by assumption and by Lemma 1(ii)), we have $|e_{i-1}| - |e|\sin\alpha_{i-1}/\tan 89.5° \geq 2\cos 61°|e| - |e|\sin 61°/\tan 89.5° \geq 0.96|e| > |e|\cos 60° \geq |e|\cos\alpha_{i-1}$. Since v is to the left of $l(\underline{\alpha_i})$ and since $k(v,m)$ intersects $l(\overline{\alpha_i})$, there exists a bounded region F_i of W_i, delimited by $k(v,m)$ from above and from below, by $l(\underline{\alpha_i})$ from the left, and by $l(\overline{\alpha_i})$ from the right, in which u_{i+1} can not lie, as otherwise Lemma 1(i) would be violated. By Lemma 8, the distance between u_i and every point above F_i is at least $1.604|e_{i-1}|\cos 0.5° > 1.4|e_{i-1}|$. Hence, by the assumption that $|e_i| \leq |e_{i-1}|/10$, u_{i+1} is not above F_i. It follows that u_{i+1} is in R_i.

Next, we prove that w is in R_i. Observe that $\beta_{i+1} \leq 90.5°$, by the assumption that $\beta'_{i+1} \geq 89.5°$ and since the three angles incident to u_{i+1} and different from β_{i+1} and β'_{i+1} sum up to at least $180°$ (by Lemma 1(ii)). Hence, e^* can not cross $l(\overline{\alpha_i})$. Since $\beta_i, \beta_{i+1} \leq 90.5°$, the angle defined by a clockwise rotation bringing a horizontal line to coincide with e^* is at most $1°$. Since the x-coordinate of u_{i+1} is at most $|e_{i-1}| + \frac{|e_{i-1}|\sin 0.5}{10}$, the y-coordinate of the line through e^* if $x = |e|\cos\alpha_{i-1}$ is at
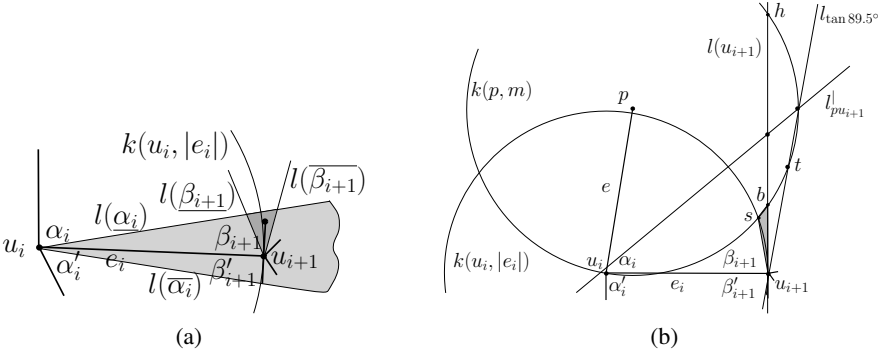
Fig. 5. (a) Illustration for Lemma 10. The dark-shaded region is R_{i+1}. (b) Illustration for Lemma 11. The dark-shaded region is R_1. To improve the readability, angles and edge lengths in the illustrations do not correspond to actual angles and edge lengths.

most $\frac{|e_{i-1}|}{10}+\tan 1^\circ(|e_{i-1}|+\frac{|e_{i-1}|\sin 0.5}{10}-|e|\cos\alpha_{i-1}) \leq \frac{|e|}{20\cos 61^\circ}+\tan 1^\circ(\frac{|e|}{2\cos 61^\circ}+\frac{|e|\sin 0.5}{20\cos 61^\circ}-|e|\cos 61^\circ) < 0.112|e| < |e|\sin\alpha_{i-1}$, since $\alpha_{i-1} \leq 61^\circ$, by assumption, and $2|e_{i-1}|\cos\alpha_{i-1} \leq |e|$, by Lemma 2. Then, the line through e^* crosses the vertical line through v below v. Since the y-coordinate of every point above F_i is at least $1.4|e_{i-1}|$, by Lemma 8, e^* can not cross $k(v,m)$. Further, the region S_i bounded by e from the left, by e_{i-1} from below, by $l(\alpha_i)$ from the right, and by the horizontal line through v from above entirely belongs to $\overline{k(v,m)} \cup k(u_{i-1},|e_{i-1}|)$, by Lemma 5; since the y-coordinate of w is at most $0.112|e| < |e|\sin\alpha_{i-1}$, if e^* crosses $l(\alpha_i)$, then either w is in S_i, thus violating Lemma 1(i), or e^* crosses an edge of T^*, thus violating Lemma 1(iv). Hence, w is in R_i.

Finally, consider the rest of $Cl(u_i)$. The angle defined by a clockwise rotation bringing an edge g_1 of $Cl(u_i)$ to overlap with the next edge g_2 of $Cl(u_i)$ is at most 120°, since the four other angles incident to the vertex shared by g_1 and g_2 sum up to at least 240° (by Lemma 1(ii)). Hence, no edge g_x of $Cl(u_i)$ crosses $l(\overline{\alpha_i})$ or $k(v,m)$, as otherwise g_x crosses an edge of T^*, thus violating Lemma 1(iv). Moreover, no edge g_x of $Cl(u_i)$ crosses $l(\underline{\alpha_i})$, as otherwise either one end-vertex of g_x is in S_i, thus violating Lemma 1(i), or g_x crosses an edge of T^*, thus violating Lemma 1(iv). □

Lemma 9 assumes that $|e_i| \leq \frac{|e_{i-1}|}{10}$. The reason why such a ratio can be assumed will be made clear at the end of the section.

We now prove that the condition that the clockwise path of each vertex is inside a bounded region propagates along the vertices of the backbone. Refer to Fig. 5(a).

Lemma 10. *Suppose that* $\alpha_i \geq 89.5^\circ$, *that* $\beta'_{i+1} \geq 89.5^\circ$, *and that* $Cl(u_i)$ *is in a bounded region* R_i *that is a subset of a wedge* W_i *centered at* u_i *with angle* 1°. *Then,* $\beta_{i+1} \geq 89.5^\circ$. *Moreover,* $Cl(u_{i+1})$ *is in a bounded region* R_{i+1} *that is a subset of a wedge* W_{i+1} *centered at* u_{i+1} *with angle* 1°.

Proof: Since $Cl(u_i)$ is in R_i, it follows that u_{i+1} is in R_i. Then, w is not inside $k(u_i,|e_i|)$, as otherwise Lemma 1(i) would be violated. Hence, the minimum value of $\widehat{u_i u_{i+1} w} = \beta_{i+1}$ is achieved if w is on $k(u_i,|e_i|)$, inside R_i, and hence inside W_i.

If w is on $k(u_i, |e_i|)$, then triangle $\Delta(u_i u_{i+1} w)$ is isosceles. Since $\widehat{u_{i+1} u_i w} \leq 1°$, then $\beta_{i+1} \geq 89.5$, thus proving the first part of the lemma.

Next, let $l(\beta_{i+1})$ $(l(\overline{\beta_{i+1}}))$ be the half-line starting at u_{i+1} such that a $89.5°$ (resp. $90.5°$) clockwise rotation around u_{i+1} brings e_i to overlap with $l(\beta_{i+1})$ (resp. with $l(\overline{\beta_{i+1}})$). Define R_{i+1} as the intersection of R_i and the wedge delimited by $l(\beta_{i+1})$ and $l(\overline{\beta_{i+1}})$. Then R_{i+1} is bounded as R_i is; further, R_{i+1} is a subset of a wedge W_{i+1} centered at u_{i+1} with angle $1°$. We prove that $Cl(u_{i+1})$ lies inside R_{i+1}. Since $\beta'_{i+1} \geq 89.5°$ and the three angles incident to u_{i+1} and different from β_{i+1} and β'_{i+1} sum up to at least $180°$, it holds $\beta_{i+1} \leq 90.5°$. Since $Cl(u_i)$ is in R_i and the angle defined by a clockwise rotation bringing an edge g_1 of $Cl(u_i)$ to overlap with the next edge g_2 of $Cl(u_i)$ is at most $120°$, as the four other angles incident to the vertex shared by g_1 and g_2 sum up to at least $240°$ (by Lemma 1(ii)), then every vertex of $Cl(u_{i+1})$ is not to the right of $l(\overline{\beta_{i+1}})$, as otherwise an edge of such a path crosses e_i or (u_{i+1}, w), thus contradicting Lemma 1(iv). The region delimited by e_i from below, by $l(\beta_{i+1})$ from the right, and by $l(\alpha_i)$ from above is a subset of $k(u_i, |e_i|)$ since the line through u_{i+1} and through the intersection point of $k(u_i, |e_i|)$ and $l(\alpha_i)$ forms with e_i an angle which is at least $89.5°$. Hence, if an edge of $Cl(u_{i+1})$ crosses $l(\beta_{i+1})$, then either a vertex of $Cl(u_{i+1})$ is in $k(u_i, |e_i|)$, thus violating Lemma 1(i), or an edge of $Cl(u_{i+1})$ crosses e_i or (u_{i+1}, w), thus violating Lemma 1(iv). It follows that $Cl(u_{i+1})$ is in R_{i+1}. $\qquad\square$

We now deal with the edge lengths in any MST embedding of T^*. Consider a backbone edge $e_i = (u_i, u_{i+1})$ such that the outgoing angle α_i is small. Let $e^* = (u_{i+1}, q)$ $(e = (u_i, p))$ be the edge following e_i in the clockwise (resp. counterclockwise) order of the edges incident to u_{i+1} (resp. to u_i). Let α_i and β_{i+1} be the angles delimited by e_i and e and by e_i and e^*, respectively. See Fig. 5(b). The following lemma asserts that is α_i is small and β_{i+1} is not too large then e^* is much shorter than e_i.

Lemma 11. If $\alpha_i \leq 61°$ and $\beta_{i+1} \leq 90.5°$, then $\frac{|e^*|}{|e_i|} \leq 0.073$.

The next lemma asserts that if β_{i+1} and β'_{i+1} are large, then all the edges incident to u_{i+1} have about the same length. Denote by $e_i, e^1_{i+1}, e^2_{i+1}, e^3_{i+1}$, and e^4_{i+1} the clockwise order of the edges incident to u_{i+1}, where β_{i+1} and β'_{i+1} are both incident to e_i.

Lemma 12. If $\beta_{i+1}, \beta'_{i+1} \geq 89.5°$, then $\max\{e^2_{i+1}, e^3_{i+1}, e^4_{i+1}\} \leq 1.032|e^1_{i+1}|$.

The previous two lemmata, together with Lemma 3, imply the following.

Corollary 1. If $\alpha_i \leq 61°$ and $\beta'_{i+1} \geq 89.5°$, then all the edges incident to u_{i+1} and different from e_i have length at most $0.1|e_i|$.

5   The Proof of the Area Bound

We prove that any MST embedding of T^* is such that, for each backbone vertex u_i of C^*, the outgoing angles of u_i are either both small or one small and one large. We derive a $2^{\Omega(n)}$ lower bound on the area requirements of any MST embedding of T^*. Refer to the notation of Sect. 4. Let k be the number of backbone vertices of C^*.

Lemma 13. *For each $0 \leq i \leq k - 2$, one of the following holds: (Condition 1): $\alpha_i, \alpha_i' \leq 61°$; (Condition 2): $\alpha_i \geq 89.5°$, $\alpha_i' \leq 61°$, and $Cl(u_i)$ is in a bounded region R_i that is a subset of a wedge W_i with angle $1°$ centered at u_i; (Condition 3): $\alpha_i' \geq 89.5°$, $\alpha_i \leq 61°$, and $Ccl(u_i)$ is in a bounded region R_i that is a subset of a wedge W_i with angle $1°$ centered at u_i.*

Proof: The proof is by induction on i. In the base case $i = 0$ and, by Lemma 4, $\alpha_0, \alpha_0' \leq 61°$, thus Condition 1 holds. Next we discuss the inductive case.

Suppose that Condition 1 holds for i. By Lemma 3, we have $\beta_{i+1}, \beta_{i+1}' \geq 89.5°$. By Corollary 1, all the edges incident to u_{i+1} and different from e_i have length at most $|e_i|/10$. By Lemma 1(ii), each of the angles incident to u_{i+1} and different from β_{i+1} and β_{i+1}' is at most $61°$. Hence, if e_{i+1} is in position 2 or 3, then Condition 1 holds for $i + 1$. If e_{i+1} is in position 1 (that is $\alpha_{i+1} = \beta_{i+1}$), then $\alpha_{i+1}' \leq 61°$. Moreover, by Lemma 3, $\beta_{i+2}' \geq 89.5°$. Then, all the conditions of Lemma 9 are satisfied, namely $\alpha_i \leq 61°$, $\beta_{i+1}', \beta_{i+2}' \geq 89.5°$, and $|e_{i+1}| \leq |e_i|/10$. Hence, $Cl(u_{i+1})$ is in a bounded region R_{i+1} that is a subset of W_{i+1} and thus Condition 2 holds for $i + 1$. If e_{i+1} is in position 4, then an analogous proof shows that Condition 3 holds for $i + 1$.

Suppose that Condition 2 holds for i (the case in which Condition 3 holds for i can be discussed symmetrically). By Lemma 3, $\beta_{i+1}' \geq 89.5°$. Hence, all the conditions of Lemma 10 are satisfied, namely $\alpha_i \geq 89.5°$, $\beta_{i+1}' \geq 89.5°$, and $Cl(u_i)$ is in a bounded region R_i that is a subset of a wedge W_i with angle $1°$ centered at u_i. It follows that $\beta_{i+1} \geq 89.5°$ and $Cl(u_{i+1})$ is in a bounded region R_{i+1} that is a subset of a wedge W_{i+1} with angle $1°$ centered at u_{i+1}. By Lemma 1(ii), each angle incident to u_{i+1} and different from β_{i+1} and β_{i+1}' is at most $61°$. Thus, if e_{i+1} is in position 2 or 3, then Condition 1 holds for $i + 1$, and if e_{i+1} is in position 1, then Condition 2 holds for $i+1$. Suppose that e_{i+1} is in position 4. Since each angle incident to u_{i+1} and different from β_{i+1} and β_{i+1}' is at most $61°$, it holds $\alpha_{i+1} \leq 61°$ and then, by Lemma 3, $\beta_{i+2} \geq 89.5°$. Since $\beta_{i+1}, \beta_{i+1}' \geq 89.5°$, by Corollary 1 all the edges incident to u_{i+1} and different from e_i have length at most $|e_i|/10$. Then, all the conditions of the symmetric of Lemma 9 are satisfied, namely $\alpha_i' \leq 61°$, $\beta_{i+1}, \beta_{i+2} \geq 89.5°$, and $|e_{i+1}| \leq |e_i|/10$. Hence, $Ccl(u_{i+1})$ is in a bounded region R_{i+1} that is a subset of W_{i+1} and thus Condition 3 holds for $i + 1$. $\square$

Theorem 1. *Any MST embedding of T^* has $2^{\Omega(n)}$ area.*

Proof: Since the complete tree T_c has constant degree and constant height, then each caterpillar, and in particular C^*, has $k = \Omega(n)$ backbone vertices. By Lemmata 3, 10, and 13, the incoming angles β_i and β_i' are both larger than $89.5°$, for each $1 \leq i \leq k-1$. By Corollary 1, $|e_{i+1}| \leq \frac{|e_i|}{10}$, for each $1 \leq i \leq k - 1$. Hence $\frac{|e_1|}{|e_k|} \geq 10^{k-1} = 2^{\Omega(n)}$. The theorem follows by observing that, in any MST embedding of the root of T_c and of its children, both dimensions have size at least $\sin 30° = 0.5$. $\square$

6   Conclusions

In this paper we have shown trees requiring exponential area in any MST embedding, thus settling a 20-years-old problem proposed by Monma and Suri [16]. Observe that

the area requirements of the MST embeddings constructed by the algorithm presented by Monma and Suri is $2^{\Omega(n^2)}$, while no $2^{O(n)}$-area MST embeddings are known to exist for all n-vertex degree-5 trees. We believe that such a gap can be closed by further improving our exponential lower bound, as in the following.

Conjecture 1. Every MST embedding of T^* has $2^{\Omega(n^2)}$ area.

References

1. Angelini, P., Bruckdorfer, T., Chiesa, M., Frati, F., Kaufmann, M., Squarcella, C.: On the area requirements of Euclidean minimum spanning trees. Technical Report RT-DIA-183-2011, Dept. Comp. Sci. Autom., Roma Tre University (2011)
2. Arora, S.: Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. J. ACM 45(5), 753–782 (1998)
3. Bose, P., Lenhart, W., Liotta, G.: Characterizing proximity trees. Algorithmica 16(1), 83–110 (1996)
4. Chan, T.M.: Euclidean bounded-degree spanning tree ratios. Discr. Comp. Geom. 32(2), 177–194 (2004)
5. Di Giacomo, E., Didimo, W., Liotta, G., Meijer, H.: Drawing a tree as a minimum spanning tree approximation. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 61–72. Springer, Heidelberg (2010)
6. Eades, P., Whitesides, S.: The realization problem for Euclidean minimum spanning trees is NP-hard. Algorithmica 16(1), 60–82 (1996)
7. Francke, A., Hoffmann, M.: The Euclidean degree-4 minimum spanning tree problem is NP-hard. In: SoCG 2009, pp. 179–188 (2009)
8. Frati, F., Kaufmann, M.: Polynomial area bounds for MST embeddings of trees. Tech. Report RT-DIA-122-2008, Dept. of Comp. Sci. Autom., Roma Tre University (2008)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)
10. Hurtado, F., Liotta, G., Wood, D.R.: Proximity drawings of high-degree trees. In: CoRR, abs/1008.3193 (2010)
11. Jothi, R., Raghavachari, B.: Degree-bounded minimum spanning trees. Discr. Appl. Math. 157(5), 960–970 (2009)
12. Kaufmann, M.: Polynomial area bounds for MST embeddings of trees. In: Hong, S.-H., Nishizeki, T., Quan, W. (eds.) GD 2007. LNCS, vol. 4875, pp. 88–100. Springer, Heidelberg (2008)
13. Liotta, G.: Handbook of Graph Drawing, ch.4. In: Tamassia, R. (ed.). CRC Press, Boca Raton (2011)
14. Liotta, G., Tamassia, R., Tollis, I.G., Vocca, P.: Area requirement of Gabriel drawings. In: Bongiovanni, G., Bovet, D.P., Di Battista, G. (eds.) CIAC 1997. LNCS, vol. 1203, pp. 135–146. Springer, Heidelberg (1997)
15. Mitchell, J.S.B.: Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. SIAM J. Comput. 28(4), 1298–1309 (1999)
16. Monma, C.L., Suri, S.: Transitions in geometric minimum spanning trees. Discr. Comput. Geom. 8, 265–293 (1992)
17. Papadimitriou, C.H., Vazirani, U.V.: On two geometric problems related to the traveling salesman problem. J. Algorithms 5, 231–246 (1984)

Multi-target Ray Searching Problems

Spyros Angelopoulos[1], Alejandro López-Ortiz[2], and Konstantinos Panagiotou[3]

[1] CNRS-LIP6, Pierre and Marie Curie University, Paris, France
[2] David R. Cheriton School of Computer Science, University of Waterloo, Canada
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. We consider the problem of exploring m concurrent rays using a single searcher. The rays are disjoint with the exception of a single common point, and in each ray a potential target may be located. The objective is to design efficient search strategies for locating t targets (with $t \leq m$). This setting generalizes the extensively studied *ray search* (or *star search*) problem, in which the searcher seeks a single target. In addition, it is motivated by applications such as the interleaved execution of heuristic algorithms, when it is required that a certain number of heuristics have to successfully terminate.

We apply two different measures for evaluating the efficiency of the search strategy. The first measure is the standard metric in the context of ray-search problems, and compares the total search cost to the cost of an optimal algorithm that has full information on the targets. We present a strategy that achieves optimal competitive ratio under this metric. The second measure is based on a weakening of the optimal cost as proposed by Kirkpatrick [ESA 2009] and McGregor *et al.* [ESA 2009]. For this model, we present an asymptotically optimal strategy which is within a multiplicative factor of $\Theta(\log(m-t))$ from the optimal search cost. Interestingly, our strategy incorporates three fundamental search paradigms, namely uniform search, doubling and hyperbolic dovetailing. Moreover, for both measures, our results demonstrate that the problem of locating t targets in m rays is essentially as difficult as the problem of locating a single target in $m - (t - 1)$ rays.

1   Introduction

Searching for a target is a common task in everyday life, and, unsurprisingly, an important computational problem with numerous applications in various contexts. This class of problems involves a *searcher* that must locate a *target* which lies at some unknown point in the environment. The natural objective is to devise efficient strategies that allow the searcher to locate the target as quickly as possible. One of the earliest examples of such problems is the *linear search problem*, proposed by Bellman [4] and independently by Beck [2]. Here, the environment consists of an infinite line, with the searcher initially at some point designated as the origin, and the target located at an unknown point on the line, at distance d from the origin. The objective is to minimize the worst-case ratio of the distance traveled by the searcher over d.

A natural generalization of the linear-search problem is the *star search* or *ray search* problem, which is also known, informally, as the *m-lane cow-path problem*. In this setting, we are given a set of m semi-infinite rays (lanes), all with a common origin O, and a searcher (cow) initially placed at the origin O. The target (pasture) is located at distance d from O, however the searcher is oblivious of the ray on which the target lies. A *strategy* is an algorithm that specifies how the searcher traverses the rays, and the objective is to minimize the worst-case distance traveled, again normalized by the optimal distance d.

This deceivably simple problem has important applications to robot navigation, artificial intelligence, and operations research (see e.g. [5,12,16,19,1,10,6,11] for some illustrative examples). This is due to the fact that it can be applied in settings in which we seek efficient allocation of resources to multiple tasks. For instance, consider the setting in which m different randomized heuristics (of the *Las Vegas* type) can be employed to solve a problem. However, we do not know in advance which of the heuristics will terminate successfully on a given input. How should we distribute the processing time to the different heuristics (assuming that we can interleave the execution of the heuristics)? This is an example of a setting that is often encountered in the construction of (deterministic or randomized) *hybrid* algorithms [12]. A different application is the design of efficient *interruptible* algorithms, namely algorithms that can return meaningful solutions even if interrupted during their execution. This is a fundamental problem in artificial intelligence, with surprising connections to the ray-search problem [5].

To our knowledge, with the exception of [18], all previous work on ray-search (and related) problems has focused on the case in which the searcher must locate *a single* target. However, a natural generalization of the problem involves the setting in which multiple targets may exist, and the searcher's objective is to locate t different targets. For instance, consider the setting in which a hybrid algorithm can execute m different heuristics, as described earlier. It may be the case that we require that not just a single, but rather several heuristic algorithms successfully terminate and return a solution. This is often desired in situations in which we do not have strong guarantees on the quality of the solution that each heuristic returns. A typical example is SAT-solvers that invoke such hybrid algorithms (also known as *algorithm portfolios* [9]): here, we do not know in advance which heuristic is the most appropriate for any given input. The objective of this paper is thus to initiate the study of ray-search problems in the setting where the strategy must guarantee that a certain number of targets are located as efficiently as possible. We also expect that the generalization to multiple targets will prove an interesting topic of study in other contexts of search and exploration problems, which so far have focused almost exclusively on the case of a single target.

Models and performance measures. Throughout the paper we consider the setting in which up to m potential targets are placed in the m rays, with at most one target per ray. More specifically, we will denote by $\lambda_1 \geq \ldots \geq \lambda_m$ the distances of the targets, in non-increasing order. We allow the possibility that $\lambda_i = \infty$ for

some values $i \in [1, m]$. In other words, there may be rays with no target located on them. We will denote by $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$ the multiset of all target distances, and we make the standard assumption $\lambda_m \geq 1$. Note that without this assumption, i.e., if the distances can become arbitrarily small, no search algorithm can be competitive (see, e.g., [6]). We seek efficient strategies for locating $t \leq m$ targets, where we assume that t is provided as input to the algorithm, and is thus known. We emphasize that the searcher can move from ray to ray only by passing over the origin, and that the search terminates when the t-th target is reached.

In order to evaluate the performance of a strategy, one needs to compare the *search cost* incurred by the algorithm (that has no information on the set Λ) to the cost of an *ideal* algorithm that has a certain amount of information concerning the targets. The worst-case ratio of these costs gives rise to the so-called *competitive ratio*, since search problems are often considered online problems in the literature of search and exploration.

We distinguish between two concrete models. First, we consider the classical model in which the ideal algorithm has *complete information* on the placement of targets, that is, the algorithm knows not only Λ, but also the specific ray on which the target at distance λ_i can be found. In this model, the cost of the optimal algorithm is easy to evaluate, and equals $2 \sum_{i=m-t+2}^{m} \lambda_i + \lambda_{m-t+1}$. In other words, the optimal strategy locates the t targets closest to the origin (the factor of 2 is due to the searcher returning back to the origin). On the other hand, more recently, a different approach in defining a less powerful (and in a sense, more realistic) ideal algorithm was proposed independently by Kirkpatrick [14] and McGregor, Onak and Panigrahy [18]. In their setting, the ideal algorithm has only *partial information* about the locations of the targets. More specifically, we allow the ideal algorithm knowledge of the set Λ, but not of the exact mapping of distances λ_i to the rays. The implication is that the ideal algorithm itself will be associated with an *intrinsic* search cost, which is the worst-case cost of a search strategy that has knowledge of Λ. Following the notation of Kirkpatrick, we will denote by $\xi_t(\Lambda)$ the intrinsic (i.e., "optimal") cost for locating t targets. We shall omit the subscript "t", whenever it is clear from the context.

Contribution of this paper. In this work we provide optimal strategies for locating $t \leq m$ targets in the m-ray setting. In Section 2 we focus on the complete information model for the ideal algorithm. We show that the worst-case competitive ratio for locating t targets is the same as the worst-case competitive ratio for locating a single target in $m - (t - 1)$ rays, and we obtain a tight analytical expression for the optimal competitive ratio (c.f. Theorem 1).

Section 3 addresses the problem under the partial information model, and contains the main results of this work. First, we provide an analytic expression of the intrinsic cost, given the set of target distances Λ (c.f. Theorem 2). Next, we present a strategy based on combination of doubling and hyperbolic search that yields a $\Theta(\log m)$-competitive algorithm (c.f. Theorem 3). Last, we use the previous strategy as a subroutine so as to obtain an optimal algorithm of competitive ratio $\Theta(\log(m - t))$ (Theorem 4 and Theorem 5). Interestingly,

this optimal strategy incorporates three fundamental search paradigms, namely uniform search, doubling and hyperbolic dovetailing. Similar to our results concerning the full-information model, we can interpret this result as a reduction between the problems of locating a single and multiple targets. Namely, the optimal competitive ratio for finding t targets in m rays is determined by the optimal strategy for locating *one* target in $m - (t - 1)$ rays. Observe that, as discussed above, while the competitive ratio is the same the strategy itself is quite different from the $m - (t - 1)$ ray case.

Related work. Ray-search problems have a long and exciting history of research. We review some representative results, with the observation that the vast majority apply to the complete information model for the ideal algorithm. For the linear search problem, Beck and Newman [3] first showed an optimal competitive ratio of 9. The generalization to the m-ray ray-search problem was first studied by Gal [8] and later by Baeza-Yates *et al.* [1]. Both works proposed a round-robin strategy of exponentially increasing lengths that achieves optimal competitive ratio (see also the discussion of Jaillet and Stafford [10]). The above results are obtained by means of deterministic strategies; however, it is known that randomization can help improve the competitive ratio. In particular, Kao *et al.* [13] gave an optimal randomized algorithm for linear search, a result that was extended by Kao *et al.* [12] to the m-ray problem (under the restrictive assumption of round robin strategies). Other variants include the setting in which the searchers incur some turn cost when they switch direction (studied by Demaine *et al.* [6]), the case of multiple searchers (López-Ortiz and Schuierer [17]) and the average-case analysis of linear search (due to Kao and Littman [11]). Typically, round-robin strategies based on iterative deepening yield optimal or near-optimal algorithms, and similar ideas lead to efficient search algorithms in more general settings and environments (see the results of Koutsoupias *et al.* [15] and Fleischer *et al.* [7]).

In contrast, the study of the partial information model is much more recent. Kirkpatrick [14] addressed both deterministic and randomized algorithms under this framework. For both cases he presented optimal strategies based on a searching technique named *hyperbolic dovetailing*, since in each round a ray is searched to distance inversely proportional to its rank. The (optimal) competitive ratio of both deterministic and randomized strategies based on hyperbolic search is shown to be $\Theta(\log m)$. Independently, McGregor *et al.* [18] studied the setting in which there is a target in each ray, and the objective is to locate as many as possible at a cost close to the intrinsic cost. Their results provide randomized algorithms for locating $k - \tilde{O}(k^{5/6})$ targets at a cost no more than $(1 + o(1))$ times the intrinsic cost for locating k of them.

2   Ray Search in the Full-Information Model

For the case of a single target and m concurrent rays, it is known that optimal strategies can be found in the class of the so-called *geometric* or *exponential*

strategies (see, e.g., [8]). In this class of strategies, the searcher performs a round-robin exploration of rays with distances forming a geometric sequence (i.e., of the form $b^0, b^1, b^2, \ldots$, for some appropriate choice of the base $b > 1$). We show that similar geometric strategies lead to optimal multi-target search algorithms. Proofs are omitted for space reasons.

Lemma 1. *There is a geometric strategy for searching t targets in m rays with competitive ratio at most $1 + 2\frac{b^{m-(t-1)}}{b-1}$, where b is the base of the strategy.*

Theorem 1. *The competitive ratio of the geometric strategy of Lemma 1 is minimized for $b = \frac{m-(t-1)}{m-t}$, for which it is equal to $1 + 2\frac{(m-(t-1))^{m-(t-1)}}{(m-t)^{m-t}}$. Moreover, this is optimal, i.e., there is no algorithm with a smaller competitive ratio.*

We emphasize that the competitive ratio of Theorem 1 is the same as the competitive ratio of searching a single target in $m - t + 1$ rays [8].

3   Ray-Search in the Partial-Information Model

In this section we study deterministic algorithms for ray-search in the partial-information model for the ideal algorithm (as discussed in the introduction). Recall that the m rays are associated with a (multi)set Λ of target distances $\Lambda = \{\lambda_1, \ldots \lambda_m\}$ (with $\lambda_1 \geq \lambda_2 \geq \ldots \lambda_m$), and the objective is to locate t targets. In order to facilitate the exposition of our results, we focus on a slightly different cost formulation; namely, we assume that the searcher incurs cost only the first time it traverses a previously unexplored segment of a certain ray (for instance, we do not charge the searcher for returning to the origin). In other words, the total search cost is the sum of the maximum distances traversed on each ray. For the sake of completeness, we note that, our results can be extended to the m-ray search problem under the "standard" cost formulation.

As mentioned in Section 1, we assume that the multiset Λ is not known to the (online) search strategy, but is known, in contrast, to the ideal algorithm. A *presentation* of Λ is a specific assignment of distances in Λ to target locations in the rays, which is unknown both to the online strategy *and* to the ideal algorithm. Given Λ, we denote by $\xi_t(\Lambda)$ the *intrinsic cost* of the ideal algorithm for locating at least t targets, namely the minimum worst-case search cost of a strategy that knows Λ. The special case of $t = 1$ was treated in [14,18].

3.1   Intrinsic Cost of Multi-target Search

We begin by evaluating the intrinsic cost in the case where we search for $t \geq 1$ targets.

Theorem 2. *The intrinsic cost for locating t targets in a presentation with associated distance set Λ is*

$$\xi_t(\Lambda) = \min_{1 \leq i_1 < \ldots < i_t \leq m} \sum_{j=1}^{t} i_j \cdot \mu_{i_j}, \tag{1}$$

where $\mu_{i_j} = \lambda_{i_j} - \lambda_{i_{j+1}}$, for $j < t$, and $\mu_{i_t} = \lambda_{i_t}$.

Proof. First, we will lower-bound the intrinsic cost of any search strategy A which succeeds for all possible presentations of Λ. At each point in time, the strategy has explored each ray to some distance: in particular, suppose without loss of generality, that A has searched rays $1, \ldots, m$ to distances $d_1, \ldots d_m$, in non-increasing order, i.e., $d_1 \geq d_2 \ldots \geq d_m$. We then can claim the following property concerning A and the set of distances $\{d_i : i \in [1, m]\}$: there must exist indices $1 \leq i_1 < \ldots < i_t \leq m$, such that $d_{i_j} \geq \lambda_{i_j}$ $(1 \leq j \leq t)$, otherwise strategy A will have not located t targets for at least one presentation of Λ. It follows then that the overall search cost of strategy A has to be at least $i_1 \lambda_{i_1} + (i_2 - i_1)\lambda_{i_2} + \ldots + (i_t - i_{t-1})\lambda_{i_t}$. Note that this is equal to $\sum_{j=1}^{t} i_j \cdot \mu_{i_j}$, where the μ_i's are as in the statement of the lemma.

On the other hand, we can upper-bound the intrinsic cost by considering the following strategy that works in t phases. Fix indices $1 \leq i_1 < i_2 \ldots < i_t \leq m$. In phase t, the strategy searches rays $1, \ldots i_t$ up to depth $\lambda_{i_t} = \mu_{i_t}$. Let N_t denote the set of rays on which no target was located in phase t. In phase $t - 1$, the strategy will search all rays in N_t up to an additional length of $\lambda_{i_{t-1}} - \lambda_{i_t} = \mu_{i_{t-1}}$. More general, if N_j denotes the set of rays for which no target was located during phase j, then in phase $j - 1$ the strategy will search all rays in N_j up to an additional distance of $\lambda_{i_{j-1}} - \lambda_{i_j} = \mu_{i_{j-1}}$. We terminate when t targets have been located (which may happen before we reach the end of phase 1). Note that this strategy will always locate at least one target per phase, since in phase j it searches i_j rays up to distances λ_{i_j}, hence its cost is upper bounded by $\sum_{j=1}^{t} i_j \cdot \mu_{i_j}$. $\square$

Given two sets of target distances Λ and Λ', we say that Λ *dominates* Λ' (denoted by $\Lambda \succ \Lambda'$) if $\lambda_i \geq \lambda_i'$. The following is an immediate corollary of Theorem 2.

Corollary 1. *If $\Lambda \succ \Lambda'$, then $\xi_t(\Lambda) \geq \xi_t(\Lambda')$, for any t.*

3.2   A $O(\log m)$-Competitive Algorithm

In this section we present a search strategy for locating t targets that achieves competitive ratio $O(\log m)$. This strategy, which we call Adaptive Hyperbolic Search is based on a combination of hyperbolic search and doubling, and will be used as subroutine in the construction of an optimal algorithm in Section 3.3.

Before we present our algorithm, let us describe briefly the hyperbolic dovetailing algorithm in [14,18] for locating a single target. The algorithm begins with assigning unique ranks to the rays, which are integers in $[1, m]$, and by initializing a counter c to the value 1. It then proceeds in iterations, where the ray with rank i is searched up to distance c/i. If no target was found this way, then c is increased by 1 at the beginning of the next iteration.

There are (at least) two natural ways one could attempt to extend this algorithm to the case where we are interested in finding $t > 1$ targets. On the one hand, we could simply choose to never change the rank of rays, even after a target is located on some ray. On the other hand, we could behave "aggressively", and update the ranks immediately after a target was located (according to some

chosen rule). However, it turns out that both ways lead to extremely ineffective algorithms of competitive ratio $\Omega(t)$.

Our algorithm (see the pseudocode below) strikes a balance between the above two extremes. Initially, as in the classical hyperbolic search, it begins by assigning unique ranks to the rays, and by initializing a counter c to 1. However, the execution of our algorithm is divided into *epochs*, where each epoch in turn consists of two *phases* (the boolean variable $firstphase$ in the statement of the algorithm determines whether we are in the first phase or not). During the first phase of each given epoch, the algorithm searches, for all i, the ray of rank i (denoted by r_i in the pseudocode) to a distance of c/i, i.e., it performs a hyperbolic search according to rank. The phase terminates when a target is discovered, at which point the second phase begins; this phase proceeds until iteration $\bar{c} \leftarrow 2c$, and again consists of hyperbolic search according to rank (in what follows we call *iteration j* the execution of lines 3–33 when c has value j). Targets found during this phase do not affect the rank. However, at iteration $\bar{c}$ the ranks of the rays are updated (lines 27–33), by removing rays on which targets are found.

Algorithm 1. Adaptive hyperbolic search

```
 1  T ← 0 , c ← 1, c̄ ← 0 , firstphase ← true
 2  for i = 1 to m do
 3      rᵢ ← i
 4      foundᵢ ← false
 5  end
 6  repeat
 7      while firstphase=true or (firstphase=false and c < c̄) do
 8          for i = 1 to m do
 9              if foundᵢ = false then
10                  search ray rᵢ up to distance c/i
11                  if target found at ray rᵢ then
12                      foundᵢ ← true
13                      T ← T + 1
14                      if T = t then break
15                      if firstphase=true then
16                          firstphase ← false
17                          c̄ ← 2c
18
19
20
21              end
22              c ← c + 1
23          end
24          firstphase ← true , count ← 1
25          for i = 1 to m do
26              if foundᵢ = false then
27                  rᵢ ← count
28                  count ← count+1
29
30          end
31  until T = t
```

Analysis. Denote by R_i the set of rays that acquire rank equal to i during the execution of the algorithm. Note that if a ray is assigned rank i during some epoch, and rank $j \neq i$ in some subsequent epoch, it cannot be assigned rank i again in the future. This observation allows us to define the search cost on a ray l for the

interval in which the rank of l is i, which we denote by $C^i(l)$. We also denote by $C(R_i) = \sum_{l \in R_i} C^i(l)$ the overall search cost for rays of rank i. With this notation, the cost of our algorithm can be written as

$$ALG = \sum_{i=1}^{m} C(R_i). \tag{2}$$

Moreover, we will use the notation c^* to denote the value of c when the algorithm terminates, i.e., the last iteration. The next lemma bounds the value of $C(R_i)$.

Lemma 2. *For any* $1 \le i \le m$, $C(R_i) \le \frac{3c^*}{i}$.

Proof. Note that every time the algorithm performs a search on a ray $l \in R_i$, it contributes to the cost $C(R_i)$ in two possible ways: i) By searching the ray the *first* time after it is assigned rank i (in other words, when line 12 is executed immediately after l acquires rank equal to i), and ii) in all remaining cases, i.e., subsequent iterations in the same epoch in which line 12 is executed for l. Let $C_1^i(l)$ and $C_2^i(l)$ denote the above two contributions to $C^i(l)$. Clearly,

$$C(R_i) = \sum_{l \in R_i} (C_1^i(l) + C_2^i(l)). \tag{3}$$

We first bound the cost incurred by $C_2^i(l)$. Let e denote the total number of epochs in the execution of the algorithm, and let $c_1, \ldots, c_e$ denote the value of c at the end of the corresponding epoch. (In particular, we have $c_e = c^*$). Let $1 \le j < e$ be any epoch, and let us denote by l the ray of rank i the jth epoch, and by l' the ray of rank i in the $(j+1)$th epoch. At the end of the jth epoch, l is searched to distance c_j/i. Moreover, note that the *second* time l' is searched in the $(j+1)$st epoch, it had been already searched down to distance $\frac{c_j}{i}$. Therefore, the accumulated cost for searching the rays of rank i over all epochs is bounded by $\frac{c^*}{i}$, i.e., $\sum_{l \in R_i} C_2^i(l) \le \frac{c^*}{i}$.

It remains to bound the cost $C_1^i = \sum_{l \in R_i} C_1^i(l)$. To this end, let $l_1, l_2, \ldots, l_e$ denote the rays in R_i, where l_k had rank i in epoch k. Note that there is a unique ray that had rank i in a particular epoch. Moreover, ray l_k contributes a cost of at most $\frac{c_{k-1}}{i}$ to C_1^i (where we use the convention $c_0 = 0$). Note also that the definition of the algorithm implies that $c_{k+1} \ge 2c_k$, as the first phase in the epoch ends when c attains the value $2c_k$, due to line 19 in the statement of the algorithm. We conclude that $c_k \le \frac{c^*}{2^{k-1}}$, and thus, $C_1^i = \sum_{k=1}^{j} C_1^i(l_k) \le \sum_{k=1}^{j} \frac{1}{2^{k-1}} \frac{c^*}{i} \le \frac{2c^*}{i}$. $\square$

The next lemma relates the intrinsic complexity with the cost of the algorithm.

Lemma 3. *For any* $t \ge 2$, $\xi_t(\Lambda) \ge \frac{c^*}{5}$.

Proof. Set $c' = \lfloor c^*/3 \rfloor$. We can assume, without loss of generality, that $c^* \ge 5$. In order to prove the statement, let us first assume that the iterations c' and c^* occurred in the same epoch of the execution of the algorithm. Then no target is found during iterations up to c' which also belong in the epoch of c', as otherwise it would be that $c^* \le 2c' < c^*$, a contradiction.

Observe that at the end of an epoch, i.e., at lines 28–33, each possible rank between 1 and $m - T$ is assigned to a unique ray (recall that T is the number of targets that were discovered up to the current iteration). Consequently, if in the first iteration of the succeeding epoch no additional target is found, then for each $1 \leq i \leq m - T$ there is a ray that has been searched up to distance c/i. In our case in particular, since at most $t - 1$ targets were found at the beginning of the last epoch (i.e., the epoch of c^*), for every j with $1 \leq j \leq m - (t - 1)$, there exists a ray that has been searched unsuccessfully up to distance c'/j. This implies that if we set

$$\Lambda' = \{\lambda'_1, \ldots, \lambda'_m\} = \left\{ c', \frac{c'}{2}, \frac{c'}{3}, \ldots, \frac{c'}{m - (t - 1)}, 0, \ldots, 0 \right\},$$

then $\Lambda \succ \Lambda'$. Corollary 1 implies that $\xi_t(\Lambda) \geq \xi_t(\Lambda') = \xi_1\left(\{\lambda'_1, \ldots, \lambda'_{m-(t-1)}\}\right)$, where the equality follows easily from Theorem 2. We conclude the proof in this case by observing that $\xi_1\left(\{\lambda'_1, \ldots, \lambda'_{m-(t-1)}\}\right) = \min_{1 \leq i \leq m-(t-1)} i\lambda'_i = c'$.

It remains to consider the case where iterations c' and c^* occurred in different epochs. We may further assume that in the epoch of iteration c', at least one target was discovered at some iteration smaller than or equal to c' which also belongs in the epoch of c', as otherwise the same argument as in the previous case would apply. Let $c'' \geq c'$ be the first iteration of the epoch that succeeds the epoch of iteration c'. We shall denote, in the remainder, this epoch as the *current* epoch. Note that $c'' \leq 2c' < c^*$. Suppose that $\ell \geq 0$ targets are discovered during iteration c'', and let us denote by $i_1 < i_2 < \cdots < i_\ell$ the ranks of the rays on which they were discovered, and by $d_1, \ldots, d_\ell$ their corresponding distances. We claim that

$$d_j \geq \frac{c''}{i_j + (t - \ell - 1)} \quad \text{for all} \quad 1 \leq j \leq \ell. \tag{4}$$

To show this, note that the number of targets that were found in all previous epochs (i.e., before the current epoch) is at most $t - \ell - 1$. Therefore, the ray with rank i_j in the current epoch had rank at most $i_j + (t - \ell - 1)$ in the previous epoch; this follows immediately from the rank update in lines 28–33 of the algorithm.

The definition of the i_j's implies that no target is found on all other rays that are searched in iteration c''. Thus, for all $i \in [1, m - (t - \ell - 1)] \setminus \{i_1, \ldots, i_\ell\}$ there is a ray that is searched to distance c''/i (this occurs on the ray of rank i in the current epoch). By putting this together with (4), we infer that for all i as above, there is a distinct target at distance at least c''/i, and that there are ℓ targets at distances at least $c''/(i_j + (t - \ell - 1))$. In what follows we shall exploit this to construct a lower bound on the intrinsic cost of Λ. Define

$$a_j = \begin{cases} j + (t - \ell - 1), & \text{if } j \in \{i_1, \ldots, i_\ell\} \\ j, & \text{otherwise} \end{cases}.$$

The previous discussion implies that for all $1 \leq j \leq m - (t - \ell - 1)$, there is a distinct ray with a target at distance at least c''/a_j. Let b_j denote the value of the jth element in the increasingly sorted sequence of $\{a_i\}_{1 \leq i \leq m-(t-\ell-1)}$. Note that

$b_1 \leq \ell + 1$, since the pigeonhole principle implies that one of the values $[1, \ell + 1]$ is not contained in $\{i_1, \ldots, i_\ell\}$. Similarly we can argue that $b_j \leq \ell + j$. We now define

$$\Lambda'' = \{\lambda_1'', \ldots, \lambda_m''\} = \left\{ \frac{c''}{\ell + 1}, \frac{c''}{\ell + 2}, \ldots, \frac{c''}{m - t + 1 + 2\ell}, 0, \ldots, 0 \right\},$$

where the number of 0's is $t - \ell - 1$. Then, $\Lambda \succ \Lambda''$, and thus $\xi_t(\Lambda) \geq \xi_{\ell+1}\left(\{\lambda_1'', \ldots, \lambda_{m-(t-\ell-1)}''\}\right)$. By applying Theorem 2 we infer that

$$\xi_t(\Lambda) \geq \min_{i_1 < \cdots < i_{\ell+1}} \left\{ \sum_{j=1}^{\ell} i_j(\lambda_{i_j}'' - \lambda_{i_{j+1}}'') + i_{\ell+1} \lambda_{i_{\ell+1}}'' \right\}. \tag{5}$$

Let $(i_j^*)_{1 \leq j \leq \ell+1}$ denote any choice of the i_j's that minimizes the above expression. Suppose that for all $1 \leq j \leq \ell + 1$ we have that $i_j^* \leq \ell + 1$. Then (5) simplifies to $\sum_{i=1}^{\ell+1} c''/(\ell + i) \geq c''/2$. On the other hand, suppose that there is a $1 \leq k \leq \ell + 1$ such that $i_k^* \geq \ell + 1$. Then, the bound in (5) is due to the monotonicity of the i_j^*'s at least

$$i_{\ell+1}^* \lambda_{i_{\ell+1}^*}'' = i_{\ell+1}^* \frac{c''}{\ell + i_{\ell+1}^*} \stackrel{(i_{\ell+1}^* \geq \ell)}{\geq} \frac{c''}{2}.$$

Since $c'' = \lfloor c/3 \rfloor$, the proof is completed. $\square$

Lemma 2, Lemma 3 and (3) imply the main result of this section.

Theorem 3. *The adaptive hyperbolic-search algorithm locates t targets in m rays with associated distance set Λ at a search cost of at most $15 \log m \cdot \xi_t(\Lambda)$.*

3.3   An Asymptotically Optimal Multi-target Search Algorithm

We now describe an optimal algorithm for locating t targets. We begin with a lower bound which demonstrates that the problem is at least as hard as searching a single target in $m - t + 1$ rays. The proof is omitted for space reasons.

Theorem 4. *For the m ray problem in the partial information model, there exists a distance set Λ such that every deterministic algorithm that successfully locates t targets incurs a cost at least $\Omega(\log(m-t)) \cdot \xi_t(\Lambda)$, for at least one presentation of Λ.*

Let s be such that $t = m - s$. Note that if $s \geq m/2$, then $t \leq m/2$, which in turn implies that the adaptive hyperbolic search of Section 3.2 is asymptotically optimal, as follows from Theorem 3 and Theorem 4. Therefore, we shall focus only on the case $s \leq m/2$.

The optimal algorithm, which we call *Hybrid*, consists of two phases. In the first phase we perform a *uniform* search, i.e., we search all rays in the same fixed order, increasing by a unit the distance up to which we search rays in each iteration. Once a target is located in one of the rays, we effectively discard that ray, without

affecting the ordering of the remaining rays. This phase continues until $m - 2s$ targets have been located, at which point the algorithm switches to the adaptive hyperbolic search algorithm (Algorithm 1). In this second phase, we search for s more targets in the remaining $2s$ rays on which the uniform search did not locate any targets.

Theorem 5. *Let $s \leq m/2$. Algorithm Hybrid locates $t = m - s$ targets at a total cost $O(\log s) \cdot \xi_t(\Lambda)$, for any presentation with associated distance set Λ.*

Proof. Let C_1, C_2 denote the search costs incurred by the first and second phase, respectively. We will show that $C_1 \leq 2 \cdot \xi_t(\Lambda)$, and $C_2 = O(\log s) \cdot \xi_t(\Lambda)$, which will be sufficient to prove the theorem.

Consider first the uniform-search phase. From construction, the first target will be located after the uniform search incurs a cost of at most $m\lambda_m$; the second target will be located at an overall cost of at most $m\lambda_m + (m-1)(\lambda_{m-1} - \lambda_m)$; and more generally, by the time the l-th target is discovered, the uniform-search phase has not incurred cost more than $m\lambda_m + \sum_{i=1}^{l-1}(m-i)(\lambda_{m-i} - \lambda_{m-i+1})$. Therefore, the overall cost of Phase 1 is at most

$$C_1 \leq m\lambda_m + \sum_{j=1}^{m-2s-1} (m-j)(\lambda_{m-j} - \lambda_{m-j+1}). \tag{6}$$

On the other hand, from Theorem 2, we know that there exist $1 \leq i_1 < \ldots < i_t \leq m$ such that $\xi_t(\Lambda) \geq i_t\lambda_{i_t} + \sum_{j=1}^{t-1} i_j(\lambda_{i_j} - \lambda_{i_{j+1}})$. Since $i_t > i_{t-1}$, and $\lambda_{i_t} \geq \lambda_m$ we deduce that $\xi_t(\Lambda) \geq i_t\lambda_m + i_{t-1}(\lambda_{i_{t-1}} - \lambda_m) + \sum_{j=2}^{t-1} i_{t-j}(\lambda_{i_{t-j}} - \lambda_{i_{t-j+1}})$, from which it follows that

$$\xi_t(\Lambda) \geq i_t\lambda_m + \sum_{j=1}^{t-1} i_{t-j}(\lambda_{m-j} - \lambda_{m-j+1}). \tag{7}$$

Since the indices i_j assume different values, we know that $i_t \geq t = m - s$. More generally, we have that $i_{t-j} \geq t - j = m - s - j$, for all $j \in [0, m - 2s - 1]$. Thus,

$$\frac{m-j}{i_{t-j}} \leq \frac{m-j}{m-s-j} \leq 2 \qquad \text{for all } j \leq m - 2s. \tag{8}$$

Combining (6), (7) and (8) we conclude that $C_1 \leq 2 \cdot \xi_t(\Lambda)$.

It remains to argue that $C_2 = O(\log s) \cdot \xi_t(\Lambda)$. Let M denote the subset of rays that are searched in phase 2, and let Λ_M denote the subset of Λ induced by the set M. By applying Theorem 3, we infer that $C_2 = O(\log(2s)) \cdot \xi_s(\Lambda_M)$. However, $\xi_s(\Lambda_M) \leq \xi_t(\Lambda)$. This is because even if the distances of the targets for the $m - 2s$ rays involved in Phase 1 are revealed to the optimal (i.e., ideal) algorithm, such an algorithm would still have to locate s more targets among the rays in M. Thus, it follows that $C_2 = O(\log s) \cdot \xi_t(\Lambda)$, which is also the overall complexity of the search algorithm. $\qquad \square$

4   Conclusions

Several problems remain to be studied in this setting, among them the optimal
expected search cost of t targets as well as the case where only an arbitrary subset
of size $s \leq t$ of the targets is being sought.

References

1. Baeza-Yates, R., Culberson, J., Rawlins, G.: Searching in the plane. Information and
   Computation 106, 234–244 (1993)
2. Beck, A.: On the linear search problem. Naval Research Logistics 2, 221–228 (1964)
3. Beck, A., Newman, D.J.: Yet more on the linear search problem. Israel J. of Math. 8,
   419–429 (1970)
4. Bellman, R.: An optimal search problem. SIAM Review 5, 274 (1963)
5. Bernstein, D.S., Finkelstein, L., Zilberstein, S.: Contract algorithms and robots on
   rays: unifying two scheduling problems. In: Proceedings of the 18th International
   Joint Conference on Artificial Intelligence (IJCAI), pp. 1211–1217 (2003)
6. Demaine, E.D., Fekete, S.P., Gal, S.: Online searching with turn cost. Theoretical
   Computer Science 361, 342–355 (2006)
7. Fleischer, R., Kamphans, T., Klein, R., Langetepe, E., Trippen, G.: Competitive on-
   line approximation of the optimal search ratio. SIAM Journal on Computing 38(3),
   881–898 (2008)
8. Gal, S.: Minimax solutions for linear search problems. SIAM J. on Applied Math. 27,
   17–30 (1974)
9. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–
   62 (2001)
10. Jaillet, P., Stafford, M.: Online searching. Opes. Res. 49, 234–244 (1993)
11. Kao, M.-Y., Littman, M.L.: Algorithms for informed cows. In: Proceedings of the
    AAAI 1997 Workshop on Online Search (1997)
12. Kao, M.-Y., Ma, Y., Sipser, M., Yin, Y.L.: Optimal constructions of hybrid algo-
    rithms. Journal of Algorithms 29(1), 142–164 (1998)
13. Kao, M.-Y., Reif, J.H., Tate, S.R.: Searching in an unknown environment:an opti-
    mal randomized algorithm for the cow-path problem. Information and Computa-
    tion 131(1), 63–80 (1996)
14. Kirkpatrick, D.: Hyperbolic dovetailing. In: Fiat, A., Sanders, P. (eds.) ESA 2009.
    LNCS, vol. 5757, pp. 516–527. Springer, Heidelberg (2009)
15. Koutsoupias, E., Papadimitriou, C.H., Yannakakis, M.: Searching a fixed graph. In:
    Proc. of the 23rd Int. Colloq. on Automata, Languages and Programming (ICALP),
    pp. 280–289 (1996)
16. López-Ortiz, A., Schuierer, S.: The ultimate strategy to search on m rays. Theoret-
    ical Computer Science 261(2), 267–295 (2001)
17. López-Ortiz, A., Schuierer, S.: On-line parallel heuristics, processor scheduling
    and robot searching under the competitive framework. Theoretical Computer Sci-
    ence 310(1-3), 527–537 (2004)
18. McGregor, A., Onak, K., Panigrahy, R.: The oil searching problem. In: Fiat, A.,
    Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 504–515. Springer, Heidelberg
    (2009)
19. Schuierer, S.: Lower bounds in online geometric searching. Computational Geome-
    try: Theory and Applications 18(1), 37–53 (2001)

Convex Transversals

Esther M. Arkin[1], Claudia Dieckmann[2], Christian Knauer[3],
Joseph S.B. Mitchell[1], Valentin Polishchuk[4], Lena Schlipf[2], and Shang Yang[5]

[1] Department of Applied Mathematics and Statistics, Stony Brook University, USA
{estie,jsbm}@ams.stonybrook.edu
[2] Institute of Computer Science, Freie Universität Berlin, Germany
{dieck,schlipf}@mi.fu-berlin.de
[3] Institute of Computer Science, Universität Bayreuth, Germany
christian.knauer@uni-bayreuth.de
[4] Helsinki Institute for Information Technology, CS Dept,
University of Helsinki, Finland
polishch@helsinki.fi
[5] Department of Computer Science, Stony Brook University, USA
syang@cs.stonybrook.edu

Abstract. We answer the question initially posed by Arik Tamir at
the Fourth NYU Computational Geometry Day (March, 1987): "Given
a collection of compact sets, can one decide in polynomial time whether
there exists a convex body whose boundary intersects every set in the
collection?"

We prove that when the sets are segments in the plane, deciding exis-
tence of the convex stabber is NP-hard. The problem remains NP-hard
if the sets are regular polygons. We also show that in 3D the stabbing
problem is hard when the sets are balls. On the positive side, we give a
polynomial-time algorithm to find a convex transversal of a maximum
number of pairwise-disjoint segments in 2D if the vertices of the transver-
sal are restricted to a given set of points. Our algorithm also finds a
convex stabber of the maximum number of a set of convex pseudodisks
in the plane.

The stabbing problem is related to "convexity" of point sets measured
as the minimum distance by which the points must be shifted in order
to arrive in convex position; we give a PTAS to find the minimum shift
in 2D, and a 2-approximation in any dimension. We also consider stab-
bing with vertices of a regular polygon – a problem closely related to
approximate symmetry detection.

1 Introduction

Let S be a finite set of line segments in the plane. We say that S is *stabbable* if
there exists a convex polygon whose boundary $\mathcal{C}$ intersects every segment in S;
the closed convex chain $\mathcal{C}$ is then called a (convex) *transversal* or *stabber* of S.

Research on transversals is an old and rich area. Most of the work, however,
focused on *line* transversals, i.e., on determining properties of families of *lines*

that stab sets of various types of geometric objects. Stabbing has attracted interest from various perspectives: purely combinatorial (complexity of the set of transversals, orders induced by stabbers), algorithmic (computing the stabbers), and applied (using transversal in curve reconstruction, line simplification, graphics, motion planning) – see [6] and references thereof. In some of these applications it could be of interest to use convex transversals instead of lines.

The problem of computing a convex transversal was posed in 1987 [8]. For the case of stabbing *vertical* line segments, an optimal algorithm for the problem was presented by Goodrich and Snoeyink in [4].

Contributions

We prove that finding a convex transversal for a set of segments in the plane is NP-hard; the problem remains NP-hard for a set of regular polygons. We also show that in 3D, it is NP-hard to decide stabbability of a set of balls.

We then turn to positive results: Section 3 presents a dynamic program (DP) to decide if a set of *pairwise-disjoint* segments is stabbable by a stabber whose vertices are a subset of a given candidate set of points; if the segments are not stabbable, we can output a convex stabber that intersects the maximum number of segments. The DP can be modified to decide stabbability (or to maximize the number of stabbed objects) also for a set of (possibly overlapping) convex pseudodisks. In particular, the DP can be used to give a PTAS for the following problem (related to convex hull of "imprecise" points [7]): Given a set of points in the plane, find the minimum δ^* such that shifting each point by at most δ^* brings the points into convex position. (In an earlier version of the paper (see, e.g., [1]) we erroneously claimed that there always exists a stabber with edges supported by bitangents between elements of S, leading us to claim that, even without a given candidate set of stabber vertices, we can decide existence of convex stabbers and that we can compute δ^* exactly.)

We also consider the *approximate symmetry detection* problem: Given a set of disks in the plane and an integer k, is it possible to find a point per disk such that the points form a set invariant under rotations by $2\pi/k$? For general k, the problem is NP-hard [5]; in Section 4 we give a polynomial-time algorithm for the case $k = n$. That is we answer the question: is it possible to find a point per disk such that the points are vertices of a regular polygon? We also consider an optimization variant of the problem: Given a set of points in the plane, find the minimum δ^* such that shifting each point by at most δ^* brings the points in a symmetric position.

Closed stabbers vs. Terrains. The stabbing problem formulation is isotropic in the sense that it does not single out any specific direction in the space. In function approximation and statistics applications (unlike in surface reconstruction), it is often the case that the transversal represents the graph of a function. That is, the stabber is a *terrain* – a surface that intersects every vertical line in at most one point. A *convex* terrain is a part of the boundary of a convex polygon (polytope in 3D).

Finding a convex terrain stabber is a special case of finding a convex stabber – to see this, just place one point far below the input. Our results, both positive and negative are as strong as possible w.r.t. the distinction between convex terrain and convex stabbers: Our DP allows one to find even a convex stabber (and hence also to find a convex terrain); our negative results show that it is hard already to find a convex terrain (and hence it is also hard to find a closed convex stabber).

2 Hardness Results

2.1 Stabbing Segments in the Plane is NP-Hard

We reduce from 3SAT. Our reduction is very similar to the one used to show hardness of finding the largest area convex hull of a set of points that are restricted to lie on line segments [7]. The reduction is shown in Fig. 1. We use n, m to denote the number of the 3SAT variables and clauses, respectively.

Variable gadget. For each variable we have a gadget that consists of three points (segments of 0 length) and one segment. There are two ways to traverse the gadget corresponding to setting the variable True or False.

"Squashing". We make the variable gadget "thin" by moving all three points close to the supporting line of the segment, and in addition by moving the non-middle points far apart.

Variable chain. Variable gadgets are placed along a convex chain, called the *variable chain*. The chain is almost vertical, bending to the right only slightly. The variable gadgets are "clenched" onto the chain, and the distance between consecutive gadgets is large. Thus the only way to traverse the gadgets with a convex terrain is to visit them one by one, in the order as they appear along the chain, assigning truth values to the variables in turn in each gadget.

Clause gadgets. The clause gadgets are similarly arranged, one after one, on another almost vertical convex chain, slightly bending to the left; this clause chain is placed to the right of the variable chain. Each clause gadget consists of 2 points and a segment.

Connectors. We now place $3m$ more segments, connecting a variable gadget to a clause gadget whenever the variable appears in the clause. The placement of the segments' endpoints within variable gadgets is as follows: if the variable appears unnegated, the segment touches the True path through the gadget and does not intersect the False subpath; on the contrary, if the variable appears negated, the segment touches only the False subpath. In every clause gadget, segments' endpoints look the same – see Fig. 1; as can be easily checked, a convex terrain can intersect any two of the segments, but not all three. This finishes the construction.
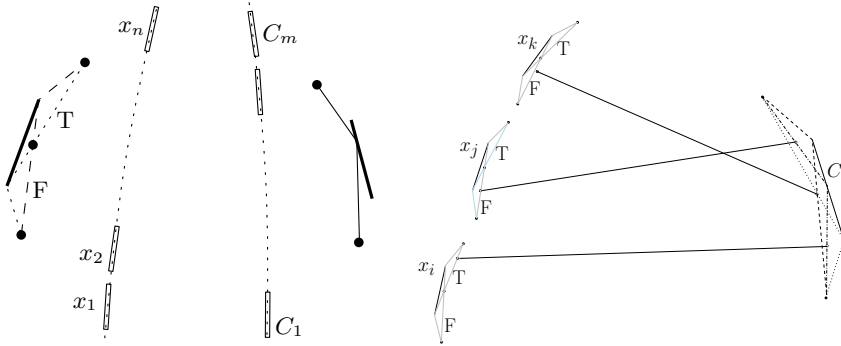
Fig. 1. From left to right: The variable gadget and the 2 ways to traverse it. The variable gadgets are threaded onto a convex chain; similarly, the clause gadgets are threaded. The chains (dotted) are not parts of the construction and are shown only for reference. The clause gadget can be traversed in only one way. A clause $C = x_i \vee \bar{x}_j \vee \bar{x}_k$: three paths are shown that pick different subsets of the three connecting segments. The gadgets and their locations are not to scale: the gadgets are thinner, so that the points are very close to the supporting line of the segment – this makes the turn angles of the paths close to π; also, consecutive gadgets along each chain are separated so that a convex terrain can make independent choices in each of them.

The reduction. If the 3SAT instance is feasible, the stabber may traverse the variables gadgets according to the satisfying truth assignment. In each of the clauses, at least one of the connecting segments (the one connecting to the satisfying variable) may be omitted; the other two are picked up by one of the three paths.

Conversely, if there exists a stabber, it must omit (at least) one connecting segment per clause. Set the variable True or False depending on whether the omitted segment connects from a True or False part of the variable gadget; this satisfies all the clauses. The True/False setting is consistent because any segment omitted by the stabber in the clause gadget must have been stabbed in the variable gadget, and there either only the True-subpath or only the False-subpath segments could have been stabbed, but not both.

2.2 Extensions

Our proof can be modified to show hardness of stabbing regular k-gons, and hardness of stabbing balls in 3D.

3 Stabbing Disjoint Segments and Convex Pseudodisks

We present a dynamic program (DP) to decide stabbability of a set S of *pairwise-disjoint* segments in the plane by a convex stabber whose vertices are restricted to come from a given discrete set $C \subset \mathbb{R}^2$ of candidate points. A subproblem in

the DP is specified by a pair of potential stabber edges together with a constant-complexity "bridge" between the edges (the bridge is either a single segment or a segment—visibility-edge—segment chain). The disjointness of the segments allows us to determine which segments must be stabbed within the subproblem. We show that a segment-free triangle can be found that separates a subproblem into smaller subproblems, which allows the DP to recurse. In the end of the section we also describe how the DP extends to the case of stabbing the maximum cardinality subset of arbitrary convex pseudodisks.

Nodes and arcs, chords and bridges. Let $\mathcal{P}'$ denote the set of intersections between two types of segments: (1) segments that have both endpoints in C (i.e., potential stabber edges), and (2) segments in S. Let $\mathcal{P} = \mathcal{P}' \cup C$; call points in $\mathcal{P}$ *nodes*. A straight-line segment between two nodes is an *arc*. Two arcs pq, rt are *compatible* if either they have a common endpoint or the supporting lines of the arcs intersect outside each of pq, rt. In other words, the points p, q, r, t are in convex position, and the pair pq, rt have the potential to be sides of a convex polygon – the stabber. Refer to Fig. 2, left.

A *chord* is an arc that does not intersect a segment in its interior. A *bridge* is a polygonal path whose both endpoints are nodes, that consists of at most 3 links, and that has the following properties: (i) if it has 1 link, then the link is either a chord or a part of a segment from S – in the latter case, the bridge is *chordless*; (ii) if it has 2 links, then one of the links is a chord, and the other is a part of a segment; (iii) if it has 3 links, then they are a part of a segment, a chord, and a part of another segment.
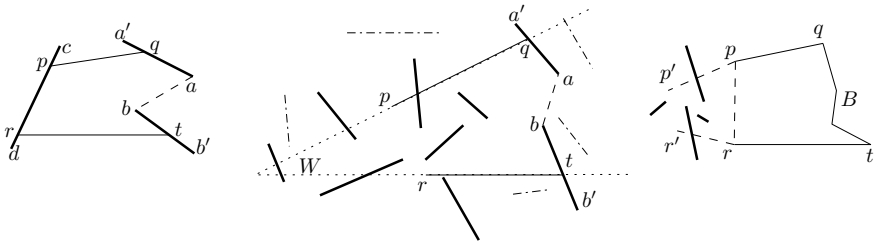
Fig. 2. Left: p, q are nodes, aa', bb', cd are segments from S. pq, rt are compatible. ba, bq, bp, br, ap, qr are chords. $pr, qabt$ are bridges; pr is chordless. Middle: The wedge W (boundaries dashed) and the bridge $B = tbaq$. The segments in $S_{pq,rt,B}$ that have to be stabbed to the left of B are bold; the segments in $S \setminus S_{pq,rt,B}$ are dash-dotted. Right: An empty subproblem (pq, rt, B) and an induced subproblem $(p'p, r'r, rp)$.

Subproblems. A subproblem in our DP is specified by two compatible arcs and a bridge. More specifically, let p, q, r, t be four nodes forming compatible arcs pq, rt. Without loss of generality let rt be below the line pq, and let q, p, r, t be the order in which the nodes appear counterclockwise on the convex hull of the arcs. We define the wedge W to be the region that is below the line supporting pq and above the line supporting rt. In addition to the two arcs, the subproblem

has in the input a bridge B that connects an endpoint of pq to an endpoint of rt. Refer to Fig. 2, middle.

Subproblem's responsibility. The crucial observation that allows us to run the DP is the following: Assuming that the arcs pq, rt are part of the stabber, we know for each segment $s \in S$ whether it should be stabbed to the left or to the right of the bridge. Indeed, only those segments that have non-empty intersection with the wedge W can be stabbed. On the other hand, no segment can have points on both sides of the bridge – for that it would have to cross the bridge, and this is impossible: the chord is not crossed by definition, and no segment is crossed by another segment due to the assumption of pairwise-disjointness of segments in S.

Let $S_{pq,rt,B}$ denote the segments that must be stabbed to the left of the bridge B; i.e., the segments that intersect W in the part of the wedge that lies to the left of B.

The function Stab($\cdot$). Define a Boolean function $\text{Stab}(pq, rt, B)$ to be True if the segments $S_{pq,rt,B}$ can be stabbed (assuming pq, rt is a part of the stabber), and to be False otherwise; for an incompatible pair of arcs pq, rt define $\text{Stab}(pq, rt, \cdot)$ to be always False. The function shows whether the stabber can be "completed" having pq, rt as its part.

In the remainder of this section we show how to evaluate the function on a subproblem given its values at other subproblems, i.e., how to solve the DP.

Empty subproblems. A subproblem is *empty* (Fig. 2, right) if no segment from S penetrates the region of W that is to the left of the bridge but to the right of rp (this includes the possibility that the bridge is the segment rp itself). An empty subproblem is *closed* if $p = r$. Closed subproblems are at the lowest level of our DP: clearly, $\text{Stab}(\sigma) = \text{True}$ for a closed subproblem σ.

Let (pq, rt, B) be an empty subproblem. We say that a subproblem $(p'p, r'r, rp)$ is an *induced* subproblem of (pq, rt, B) if pp' is below (the supporting line of) pq, and rr' is above rt. That is, the angles qpp' and trr' are convex, and thus both qpp' and trr' may potentially be parts of a convex chain – the stabber-to-be. Empty subproblems are easy to reduce to induced subproblems: $\text{Stab}(pq, rt, B) = \text{True}$ for an empty subproblem (pq, rt, B) iff $\text{Stab}(p'p, r'r, B)$ is True for at least one subproblem induced by (pq, rt, B).

General subproblems. Let $\mathcal{C}$ be the sought stabber that has pq, rt as two of the sides (Fig. 3, left). (Of course, we do not know $\mathcal{C}$, but we will not use its existence in the algorithm, we will only use $\mathcal{C}$ to argue that we can split the subproblem into smaller ones.) Let $\mathcal{C}'$ be the (convex) region bounded by $\mathcal{C}$, and let P be the part of $\mathcal{C}'$ to the left of the bridge B (i.e., P is what is chopped off $\mathcal{C}'$ by B). Consider the set $P' = P \setminus \bigcup_{s \in S_{pq,rt,B}} s$. That is, P' is P "pierced" by the segments $S_{pq,rt,B}$ that are stabbed in the subproblem (pq, rt, B).

Because $\mathcal{C}$ is a stabber, every segment in $S_{pq,rt,B}$ intersects the boundary of P. This means that P' is a (weakly) simple polygon (i.e., no segment makes
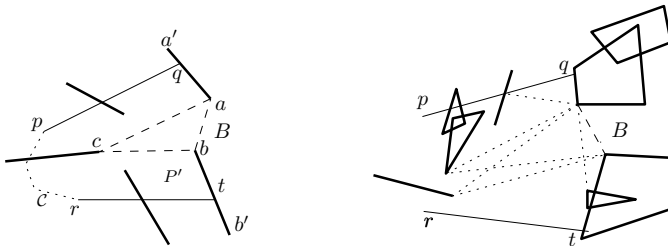
Fig. 3. Left: The (unknown) part of the stabber $\mathcal{C}$ is dotted. P' is the simple polygon bounded by the unknown part of $\mathcal{C}$, by pq, rt, by the bridge $B = tbaq$, and by the piercing segments. abc is a separating, i.e., segment-free triangle inside P'. Right: A subproblem in a collection of convex pseudodisks. Because of the pseudodisks property, no element lives on both sides of the bridge. Some potential boundaries between subproblems are shown dotted.

a hole in P' by being fully contained in the interior of P'). Each vertex of P' belongs to one of the following 5 (overlapping) sets: P_1 – vertices of the bridge; P_2 – nodes that reside on the arcs pq, rt; P_3 – nodes that belong to $\mathcal{C}$ except those in P_2; P_4 – endpoints of segments from $S_{pq,rt,B}$ that are stabbed by pq or rt; P_5 – endpoints of segments from $S_{pq,rt,B}$ that are stabbed by $\mathcal{C} \setminus pq, rt$. Note that only P_3 is not known to us (because we do not know $\mathcal{C}$); all the other sets are known as soon as the subproblem (pq, rt, B) is specified.

We define the *important* link ba of the bridge B as follows: if B is chordless, then $ba = B$; otherwise ba is the chord of B. We assume that a is closer to q, and b is closer to t along B. Our algorithm will search for a separating, i.e., segment-free triangle abc within P' where c is a vertex of P' *and* $c \notin P_3$. We first argue that such a triangle exists, and next describe what to do depending on the set, among P_1, P_2, P_4, P_5, to which c belongs.

Lemma 1. *There exists a vertex c of P' such that $c \notin P_3$ and no segment intersects the interior of abc.*

Proof. The link ba is a side of P'; thus, any triangulation of P' has a triangle abc, with c being a vertex of P'. If there exists a triangulation such that $c \notin P_3$, we are done. Otherwise, let xy be the segment that contains c; i.e., $c = xy \cap \mathcal{C}$ (Fig. 4). Move c along xy inside P'. Either c reaches the endpoint of the segment (in which case we are done because $c \in P_5$) or one of the sides of abc, say, bc hits an endpoint z of a segment from $S_{pq,rt,B}$; let c' be the position of c on xy when this happens. The convex quadrilateral $cc'ba$ has no segments in the interior, and abz is the sought triangle. $\square$

We emphasize that even though we used $\mathcal{C}$ in arguing the existence of the vertex as in the above lemma, we can find such a vertex without knowing $\mathcal{C}$ (e.g., just by trying all vertices in P_1, P_2, P_4, P_5).

We now show how our DP recurses into subproblems defined by the sides of the triangle abc (Fig. 5):
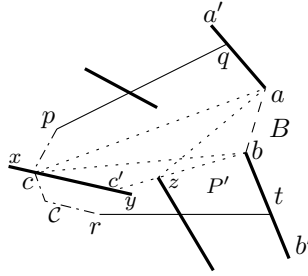
Fig. 4. abc is a triangle in a triangulation of P'; move c inside P'. abz is the sought triangle.

Case I: c is a vertex of the bridge; $c \in P_1$. Then the bridge has one fewer links, and $\mathrm{Stab}(pq, rt, B) = \mathrm{Stab}(pq, rt, B')$ where B' is the new bridge.

Case II: c is on pq, rt; $c \in P_2$. Without loss of generality suppose that $c \in rt$. If there exists a segment $s \in S_{pq,rt,B}$ that lies in the interior of the triangle tbc (i.e., s is not stabbed by tc), then s cannot be stabbed in the subproblem, and hence $\mathrm{Stab}(pq, rt, B) = \mathrm{False}$. Otherwise (i.e., if no segment intersects tbc or any segment that intersects tbc is already stabbed by rt), $\mathrm{Stab}(pq, rt, B) = \mathrm{Stab}(pq, rc, caq)$.

Case III: c is an endpoint of a segment from $S_{pq,rt,B}$ stabbed by pq, rt; $c \in P_4$. Without loss of generality suppose that c is the endpoint of a segment that is stabbed by rt; let z be the point of the stabbing. If there exists a segment $s \in S_{pq,rt,B}$ that lies in the interior of the quadrilateral $tbcz$ (i.e., s is not stabbed by tz), then s cannot be stabbed in the subproblem, and hence $\mathrm{Stab}(pq, rt, B) = \mathrm{False}$. Otherwise, $\mathrm{Stab}(pq, rt, B) = \mathrm{Stab}(pq, rz, zcaq)$.

Case IV: c is an endpoint of a segment from $S_{pq,rt,B}$ stabbed by $\mathcal{C} \setminus pq, rt$; $c \in P_5$. Let d be the other endpoint of the segment touched by the triangle abc. Then $\mathrm{Stab}(pq, rt, B) = \mathrm{True}$ iff there exists an arc uv that intersects dc (say, at a point z) such that both $\mathrm{Stab}(pq, uz, zcaq)$ and $\mathrm{Stab}(vz, rt, tbcz)$ are true. Formally,

$$\mathrm{Stab}(pq, rt, B) = \bigvee_{\substack{\mathrm{arc}\ uv:\\ dc \cap uv = z \neq \emptyset}} \left(\mathrm{Stab}(pq, uz, zcaq) \wedge \mathrm{Stab}(vz, rt, tbcz) \right)$$

Putting things together. If S is stabbable, then at least one subproblem will have $\mathrm{Stab}(pq, rt, B) = \mathrm{Stab}(rt, pq, \mathcal{B}) = \mathrm{True}$, where $\mathcal{B}$ is the bridge B traversed in the opposite direction. Indeed, if there exists a segment that intersects more than one edge of the stabber, the segment serves as the 1-link bridge B. Otherwise, let $xy \in S$ be a segment that has endpoint y inside (the convex hull of) the stabber; let $z = xy \cap \mathcal{C}$ be the point of the stabbing. Rotate yz around y, moving z along $\mathcal{C}$, until it hits another stabbing point z' or the endpoint w of another segment ww' (let $z'' = ww' \cap \mathcal{C}$ in this case). Then zyz' or $zywz''$ is a 2- or 3-link bridge in the subproblem.
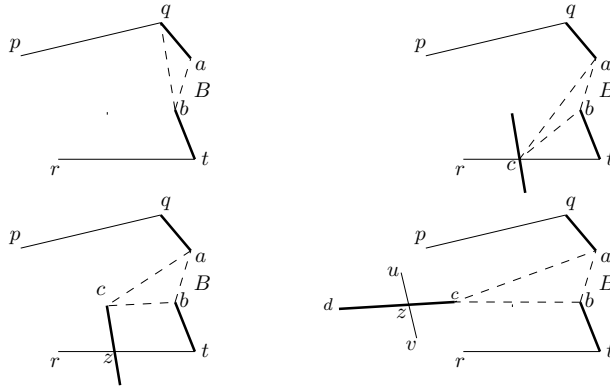
Fig. 5. The DP recursion. Top: $c \in P_1, c \in P_2$. Bottom: $c \in P_4, c \in P_5$.

Conversely, if a subproblem has $\text{Stab}(pq, rt, B) = \text{Stab}(rt, pq, \mathcal{A}) = \text{True}$, then S is stabbable. Hence, after computing the value of the function Stab() on all subproblems, we can determine if S is stabbable by checking for the existence of a subproblem with $\text{Stab}(pq, rt, B) = \text{Stab}(rt, pq, \mathcal{A}) = \text{True}$.

Stabbing convex pseudodisks. Our DP works also in the more general case of S being a collection of arbitrary convex pseudodisks. A subproblem is again specified by two potential edges of the stabber and a bridge between them; as before, the bridge is either a single element of S or two elements connected by a visibility edge (Fig. 3, right). Also as before, as soon as a subproblem is specified, we know which elements must be stabbed in the subproblem (this is where the pseudodisk property is used: no member of S may have points on both sides of the bridge.) Again, the free space P' bounded by the bridge and the (unknown, potential) stabber is simply connected, and hence can be triangulated (pseudotriangulated if the objects in S are not polygonal). The DP recurses on both sides of a triangle that has (part of) the bridge as a side.

Maximum stabbing. Our DP can be modified straightforwardly to find a stabber that stabs as many pseudodisks as possible. For that, we let the function $\text{Stab}(pq, rt, B)$ denote the number of elements of S stabbed by pq, rt plus the maximum number of other segments that can be stabbed in the subproblem (pq, rt, B). The recursions for the function change to reflect that $\text{Stab}(pr, qt, B)$ is the sum of the values of the function on the subproblems.

3.1   Convexification

Consider the following problem: Given a set P of points in the plane, find the minimum δ^* such that shifting each point by at most δ^* brings the points into convex position. A 2-approximation can be computed as follows: Let $\bar{P}$ be the convex hull of P, and let δ_m be the maximum distance from a point in P to the

boundary of $\bar{P}$; then $\delta_m \leq 2\delta^*$ (this 2-approximation algorithm works in any dimension). To obtain a PTAS, lay out $O(\frac{1}{\varepsilon}) \times O(\frac{1}{\varepsilon})$ grids in the δ_m-neighborhoods of points of P; let G be the grid points. Then do a binary search to find a minimum $k \in \{0, 1, \ldots, \lceil \frac{1}{2\varepsilon} \rceil\}$ such that there exists a convex polygon with vertices in G stabbing radius-$(\delta_m/2 + k\varepsilon\delta_m)$ disks centered on points of P.

4   Stabbing with Vertices of a Regular Polygon

The approximate symmetry detection problem is: Given a set of n disks in the plane and an integer k, is it possible to find a point per disk such that the points form a set invariant under rotations by $2\pi/k$? While the problem is NP-hard for general k [5], we solve the case $k = n$; i.e., we determine whether it is possible to find a point per disk so that the points are vertices of a regular n-gon.

4.1   Polynomial-Time Algorithm

Let $D = \{d_1, \ldots, d_n\}$ be the given disks. For points $p, c \in \mathbb{R}^2$ and integer k let $\rho_c^k(p)$ denote the image of p after rotation around c by the angle $k2\pi/n$. For two disks $d_i, d_j \in D$, let $A_{ij}^k = \{(p, c)|c \in \mathbb{R}^2, p \in d_i, \rho_c^k(p) \in d_j\} \subset \mathbb{R}^4$ be the set of all pairs (p, c) of points $p \in d_i, c \in \mathbb{R}^2$ such that p moves to d_j after rotating by $k2\pi/n$ around c; we call A_{ij}^k the *apex region*.

Fix a disk d_1. A regular n-gon with a vertex per disk of D exists iff there exist $p \in d_1$ and $c \in \mathbb{R}^2$ (the center of the n-gon) such that $\rho_c^j(p) \in d_{j+1}$ for $j = 1, \ldots, n-1$, or in other words, iff the intersection of $n-1$ apex regions A_{1j+1}^j is non-empty (here the vertices of the regular n-gon stab the disks in the order $d_1, d_2, \ldots$; of course this order is not known in advance). This prompts us to go through "all possible" intersections between the apex regions, checking for each of the intersections whether an n-gon exists.

Specifically, consider the $(n-1)^2$ apex regions $A_{1,j}^k, j = 2, \ldots, n, k = 1, \ldots, n-1$. Call a point $(p, c) \in \mathbb{R}^4$ *feasible* if it belongs to some $n-1$ of the regions, with each region being from a different disk with a different angle. Our problem has a feasible solution iff there exists a feasible point in $\mathbb{R}^4$.

There are $O(n^2)$ apex regions, and each is defined by 2 polynomials of constant degree; thus the arrangement of the regions has polynomial complexity. The feasibility of a point in $\mathbb{R}^4$ does not change as the point moves inside the cell of the arrangement; hence, in order to determine existence of a feasible point, it is enough to check the feasibility of an arbitrary representative point $r = (p, c)$ inside every cell. By [2], a representative for each cell can be obtained in $O(n^2)$ time.

To check if $r = (p, c)$ is feasible, build the bipartite graph G_r; the $n-1$ nodes on one part correspond to the disks $D \setminus d_1$, the $n-1$ nodes on the other part correspond to the angles $\{\pi/n, 4\pi/n, 6\pi/n, \ldots, (n-1)2\pi/n\}$. There is an edge between a disk node d_j and an angle node $k2\pi/n$ if p rotated around c by the angle $k2\pi/n$ lands in d_j, i.e., $\rho_c^k(p) \in d_j$. There is a perfect matching in G_r iff c is the center of a regular n-gon with vertices in the disks from D.

The above algorithm can be used for objects other than disks, only the running time will change depending on the complexity of the apex regions.

4.2   Optimization Problem: Symmetry with Imprecision

Consider the following problem (a sister of the convexification): Given a set $P = \{p_1, \ldots, p_n\}$ of n points, find minimum δ^* such that shifting each point by at most δ^* brings the points in symmetric position (which means they are vertices of a regular n-gon).

It is immediate that in the optimal solution, some J points of P are shifted by exactly δ^*; we argue that $J \leq 3$. Renumber the points in P so that the points shifted by δ^* are $p_1, \ldots, p_J$, and let $q_1, \ldots, q_J$ be the shifted points. Suppose we know that q_j is the k_j-th vertex of the optimal n-gon, where $k_1, \ldots, k_J$ are some distinct integers between 1 and n. We can then write 3 equations for each $j = 1 \ldots J$:

$$|p_j q_j| = \delta^*$$
$$q_j - c = R^{k_j 2\pi/n} (q_1 - c)$$

where c is the center of symmetry of the n-gon and $R^{k_j 2\pi/n}$ is the rotation matrix with rotation angle $k_j 2\pi/n$. Overall, we have $3J$ equations in $3 + 2J$ variables (3 variables for c and δ^* + 2 variables per q_j). The system has a solution with an isolated δ^* when $3J = 3 + 2J$.

The above observations lead to a (high) polynomial-time algorithm for the problem: Guess 3 points of P and 3 numbers $k_1 \ldots k_3$. For each guess, solve the above described system of 9 equations in 9 unknowns to get (a constant number of) candidate values for δ^*; for each candidate run the symmetry detection algorithm from Section 4.1 with radius-δ^* disks centered on points of P in the input.

A linear-time 4-approximation. Let g be the centroid of P. Take any point $p \in P$ and compute, in $O(n)$ time, the regular n-gon Q that has p as a vertex and g as center. It is known [5] that restricting the center of the n-gon to lie at g does not hurt by more than a factor of 2, and that neither does insisting, in addition, that p is a vertex of the n-gon. Interestingly, constructing Q alone does not yield a 4-approximation of the *value* of δ^* (even though we know that Q is a 4-approximation); this is because (other than for p) we do not know which point of P moves to which vertex of Q. If that is of interest, one can compute a bottleneck matching between P and vertices of Q in additional $O(n^{1.5} \log n)$ time [3].

A PTAS. We compute the 4-approximation, determine $\delta \leq 4\delta^*$, and lay out $O(\frac{1}{\varepsilon}) \times O(\frac{1}{\varepsilon})$ grids G_g, G_p in the δ-neighborhood of g and of some point $p \in P$ resp. Then, for each pair $(g_\varepsilon, p_\varepsilon)$ of grid points from $G_g \times G_p$ we compute the regular polygon $Q_{g_\varepsilon, p_\varepsilon}$ centered at g_ε and having a vertex at p_ε, and find the value $\delta_{q_\varepsilon, p_\varepsilon}$ of the bottleneck matching between P and the vertices of $Q_{g_\varepsilon, p_\varepsilon}$. The best $\delta_{g_\varepsilon, p_\varepsilon}$ is a $(1 + \varepsilon)$-approximation of δ^*, and the running time is $O(\frac{1}{\varepsilon^4} n^{1.5} \log n)$.

Acknowledgments

References

1. Arkin, E.M., Mitchell, J.S.B., Polishchuk, V., Yang, S.: Convex transversals. In: Fall Workshop on Computational Geometry (2010)
2. Basu, S., Pollack, R., Roy, M.-F.: On computing a set of points meeting every cell defined by a family of polynomials on a variety. J. Complex. 13(1), 28–37 (1997)
3. Efrat, A., Itai, A., Katz, M.J.: Geometry helps in bottleneck matching and related problems. Algorithmica 31(1), 1–28 (2001)
4. Goodrich, M.T., Snoeyink, J.: Stabbing parallel segments with a convex polygon. Comput. Vision Graph. Image Process. 49(2), 152–170 (1990)
5. Iwanowski, S.: Testing approximate symmetry in the plane is NP-hard. Theor. Comput. Sci. 80(2), 227–262 (1991)
6. Kaplan, H., Rubin, N., Sharir, M.: Line transversals of convex polyhedra in $\mathbb{R}^3$. In: SODA 2009, pp. 170–179 (2009)
7. Löffler, M., van Kreveld, M.J.: Largest and smallest convex hulls for imprecise points. Algorithmica 56(2), 235–269 (2010)
8. Tamir, A.: Problem 4-2 (New York University, Dept. of Statistics and Operations Research). Problems Presented at the Fourth NYU Computational Geometry Day (March 13, 1987)

How to Cover a Point Set with a V-Shape of Minimum Width

Boris Aronov and Muriel Dulieu[⋆]

Department of Computer Science and Engineering, Polytechnic Institute of NYU,
Brooklyn, NY 11201-3840, USA
aronov@poly.edu, mdulieu@gmail.com

Abstract. A balanced V-shape is a polygonal region in the plane contained in the union of two crossing equal-width strips. It is delimited by two pairs of parallel rays that emanate from two points x, y, are contained in the strip boundaries, and are mirror-symmetric with respect to the line xy. The width of a balanced V-shape is the width of the strips.

We first present an $O(n^2 \log n)$ time algorithm to compute, given a set of n points P, a minimum-width balanced V-shape covering P. We then describe a PTAS for computing a $(1 + \varepsilon)$-approximation of this V-shape in time $O((n/\varepsilon) \log n + (n/\varepsilon^{3/2}) \log^2(1/\varepsilon))$.

1 Introduction

Motivation. The problem we consider in this paper was motivated by the following curve reconstruction question: One is given a set of points sampled from a curve in the plane. The sample is noisy in the sense that the points lie near the curve, but not exactly on it. One would like to reconstruct the original curve from this data. Clearly one has to make some assumptions about the point set and the curve: If the curve is "too wiggly" or the noise is too large, little can be done. One approach is to assume that the curve is smooth and the sample points lie not too far from it; see [10,11] and references therein.[1] Roughly speaking, one can then approximate a stretch of a curve by an elongated rectangle (or strip) whose width is determined both by the curvature of the curve and the amount of noise. Refining this approximation allows one to reconstruct the location of the curve and its normal vector.

Complications arise when a curve makes a sharp turn, as it does not have a well-defined direction near the point of turn. It has been suggested [10, 15] that one approach to handle this situation is to replace fitting the set of points corresponding to a smooth arc of a curve with a strip by fitting with a wedge-like

---

[1] See [6] for a detailed survey of different notions of measuring similarity between geometric objects; is there a sensible (and relevant for our purposes) notion of closeness between a discrete unordered point set and a curve?

---

shape that we call a "balanced V-shape;" perhaps one might incorporate it in an algorithm such as that of [13]. It is meant to model one thickened turn in a piecewise-linear curve; refer to the figure and precise definitions below.

In this paper, we construct a slower exact algorithm for identifying a V-shape that best fits a given set of points in the plane, then a faster constant-factor approximation algorithm, and finally a considerably more involved algorithm that produces a $(1 + \varepsilon)$-approximation.

The problem we solve is a new representative of a widely studied class of problems, namely *geometric optimization* or *fitting* questions; see [1, 3, 4, 5] and references therein. Generally, the problem is to find a shape from a given class that best fits a given set of points. Classical examples of such problems are linear regression in statistics, the computation of the width of a point set (which constructs a minimum-width strip covering the set), computing a minimum enclosing ball, cylinder, or ellipsoid, a minimum-width spherical or cylindrical shell, or a small number of strips of minimum width, covering the point set; see [8, 1].

Previous work most closely related to our problem is that of Glozman, Kedem, and Shpitalnik [14]. They compute a double-ray center for a planar point set S. A *double-ray center* is a pair of rays emanating from one apex, minimizing the Hausdorff distance between S and the double ray. While the shape they consider is not exactly a V-shape, it is similar enough to be used for the same purpose. The exact algorithm they present runs in $O(n^3 \alpha(n) \log^2 n)$ time, however, in contrast to our near-quadratic-time algorithm.

Another paper closely related to our problem is that of Agarwal, Procopiuc, and Varadarajan [2]. It concerns the 2-line-center problem studied extensively in the past; see the references in [2]. The goal is to cover a given set of points by two strips of minimum possible width. A possible application is fitting *two* lines to a point set. There had been several previously known near-quadratic time exact algorithms for the problem. An $O(n \log n)$-time 6-approximation algorithm, and an $O(n \log n + n\varepsilon^{-2} \log(1/\varepsilon) + \varepsilon^{-7/2} \log(1/\varepsilon))$-time $(1 + \varepsilon)$-approximation algorithm were presented in [2]. A V-shape covering a point set is a special case of covering a point set by two strips, so some of the tools from [2] apply to our problem as well.

Problem statement and results. In this paper, we focus on the class of polygonal regions in the plane that we call balanced V-shapes. A *balanced V-shape* has two *vertices* x and y and is delimited by two pairs of parallel rays. One pair of parallel rays emanate from x and y on one side of the line xy and the other pair of rays emanate from x and y on the

other side of xy, symmetrically with respect to xy (see the above figure). In particular, a balanced V-shape is completely contained in the union of two crossing strips of equal width. Its *width* is the width of the strips.
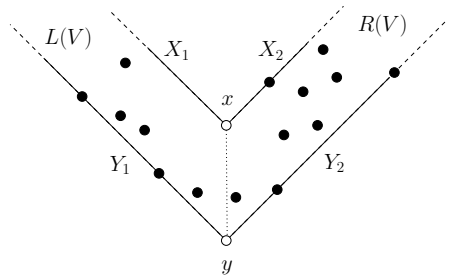
Consider a point set P of n points in the plane. We describe, in Section 3, an $O(n^2 \log n)$ time algorithm that computes a balanced V-shape with minimum width covering P.

Our algorithm actually identifies a particular type of V-shapes that we call "canonical" (see below for definitions) and enumerates all minimum-width canonical V-shapes covering P; as some degenerate n point sets have $\Theta(n^2)$ such V-shapes (see the full version of the paper), this approach will probably not yield a subquadratic algorithm. This leaves open the problem of how quickly one can identify just *one* minimum-width V-shape covering P.

In Section 4, we present an $O(n \log n)$ algorithm that constructs a V-shape covering P with width at most 13 times the minimum possible width. In Section 5, we show how to construct a $(1+\varepsilon)$-approximation in time $O((n/\varepsilon) \log n + (n/\varepsilon^{3/2}) \log^2(1/\varepsilon))$, starting with the 13-approximation obtained earlier.

2 Reduction to Canonical V-Shapes

In the remainder of this paper, unless noted otherwise, we assume that the points of P are in general position: no three points are collinear and no two pairs of points define parallel lines.

We will find it convenient to consider a larger class of objects, namely V-shapes. A (*not necessarily balanced*) *V-shape* (refer to the figure below) is a polygonal region similar to a balanced V-shape except that the widths except

that the widths of its two arms need not be the same. More formally, a *V-shape V* is a polygonal region bounded by two pairs of parallel rays emanating from its two *vertices* x and y. One pair of parallel rays (*left rays* X_1 and Y_1) lies on the left side of the directed line xy, while the other pair (*right rays* X_2 and Y_2) lies on its right side. The *inner rays* X_i emanate

from x, while *outer rays* Y_i emanate from y. $X_1 \cup X_2$ is the *inner boundary* of V, while $Y_1 \cup Y_2$ is its *outer boundary*. The *left arm*, $L = L(V)$, of V is its portion on the left of yx; i.e., it is the region bounded by rays X_1 and Y_1 and segment xy. The *width* of the left arm, width($L(V)$), is the distance between X_1 and Y_1. The right arm and its width are defined analogously. The *width* of V, width(V), is the larger of the widths of its two arms. V is contained in the union of two strips S_1 and S_2: S_i is delimited by the lines containing X_i and Y_i, respectively; we refer to S_1 and S_2 as the *left* and *right strip* of V, respectively.
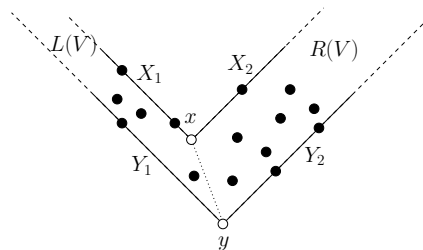
A minimum-width balanced V-shape can be obtained from a minimum-width V-shape by widening the narrower arm until the widths of the arms are equal.

In the remainder of the paper, the n-point set P is fixed. We only consider a particular type of V-shapes that we call canonical, unless otherwise stated. A V-shape is *canonical*, if the bounding rays of each arm pass through exactly three points of P; more precisely if $|X_i \cap P| + |Y_i \cap P| \geq 3$, for $i = 1, 2$ (recall

that, by our general position assumption, $|X_i \cap P|, |Y_i \cap P| \leq 2$); in addition, we require that each arm of a canonical V-shape covering P is locally of minimum width, i.e., neither arm can be narrowed by an infinitesimal motion.

It is not difficult to see that at least one minimum-width V-shape covering P is canonical, so we discard any non-canonical V-shape considered by our algorithm. Moreover we will assume that a canonical minimum-width covering V-shape does not have a zero-width arm (it cannot, as proved in the full version of this paper). The remaining canonical minimum-width V-shapes considered fall into the following three categories:

both-outer. Each outer ray contains two points of P, and each inner ray contains one.

inner-outer. On one arm of the V-shape, the outer ray contains one point of P, and the inner ray contains two. For the other arm, it is the opposite, i.e., the outer ray contains two points of P, and the inner ray contains one.

both-inner. Each outer ray contains one point of P and each inner ray contains two.

3   Computing a Canonical Minimum-Width V-Shape

To find a canonical minimum-width V-shape covering P, we will search independently for the best solution for each of the three types identified above and output the V-shape that minimizes the width. Put $H := \mathrm{CH}(P)$.

V-shapes of both-outer type. Consider a covering V-shape V with outer rays Y_1, Y_2 containing edges e_1, e_2 of H, respectively; refer to the figure on the right. Let ℓ be the bisector of the angle $Y_1 y Y_2$. In the full version of this paper, we argue that we can assume that $L(V)$ lies to the left of ℓ and $R(V)$ lies to the right of ℓ. Hence it is sufficient to find the point farthest from Y_1 (Y_2) and lying to the left (respectively, right) of ℓ. The larger distance determines the width of V. This can be accomplished by building a data structure $D(P)$ on P that supports the following queries: Given a halfplane h and a direction d, return the extreme point of $P \cap h$ in direction d. $O(n^2)$ queries are sufficient to enumerate all choices of e_1, e_2 and identify the best both-outer-type V-shape. $D(P)$ can be constructed in $O(n^2 \log n)$ time and supports logarithmic-time queries, resulting in total running time of $O(n^2 \log n)$.
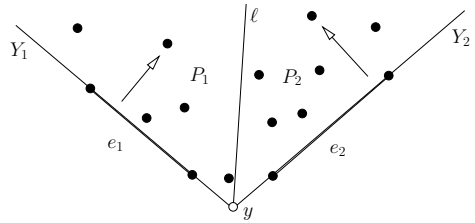
$D(P)$ is constructed as follows: We build the arrangement $\mathcal{A} = \mathcal{A}(P^*)$ of lines dual to points of P. Cells of $\mathcal{A}$ correspond to different ways to partition P by a line. We construct a directed spanning tree T of the cells of $\mathcal{A}$, starting with the bottommost cell and allowing only arcs from a cell f to a cell immediately above f and sharing an edge with it; we use $P_f \subset P$ to denote the set of

points whose dual lines lies below f. Using T as the history tree, we store the convex hull P_f for every face $f \in A$, using a fully persistent version [12] of the semi-dynamic convex hull data structure of [17]. We also preprocess $\mathcal{A}$ for point location. Give a query (say, upper) half-plane h and direction d, we locate the face f of $\mathcal{A}$ containing the point dual to the bounding line of h and consult the data structure associated with f and storing $P_f = P \cap h$ to find the extreme point P_f in direction d, all in logarithmic time. We omit the details in this version.

V-shapes of inner-outer type. In this section, we describe how to find a minimum-width canonical V-shape covering P and having exactly one edge of CH(P), say e, on its outer boundary; due to our general position assumptions, it contains two points of P on the inner bounding ray of its other arm. We handle each choice of e independently, in $O(n \log n)$ time, yielding an overall $O(n^2 \log n)$ time algorithm.

Having fixed an edge e of CH(P), consider a (minimum-width canonical) V-shape V covering P that has e on its boundary. For ease of description, suppose $Y_1 \supset e$, X_2 contains two points $p_1, p_2 \in P$, while both Y_2 and X_1 contain one point of P each, denoted p and q, respectively; see the figure to the right.

Let ℓ be the line containing e and ℓ' be the line containing $e' := p_1 p_2$. Set $Q := S_1 \cap P$ and $Q' := P \setminus Q$. We observe that
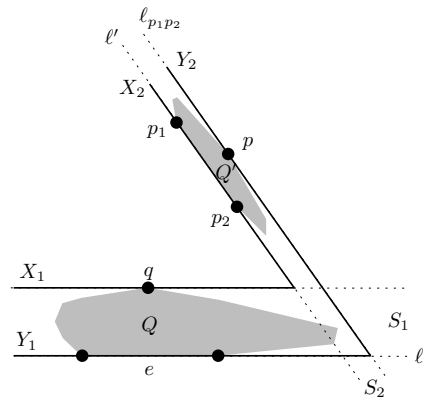
a) Q is the set of points of P at distance at most $\text{dist}(q, \ell)$ from ℓ;
b) $p_1 p_2$ is an edge of CH(Q');
c) Y_2 is contained in a supporting line $\ell_{p_1 p_2}$ of CH(Q') (which must also be a supporting line to CH(P) for V to cover P) parallel to ℓ'; this line lies on the same side of ℓ' as Q';[2] and
d) $\text{width}(V) = \max(\text{width}(S_1), \text{width}(S_2)) = \max(\text{dist}(q, \ell), \text{dist}(p_1 p_2, \ell_{p_1 p_2}))$.

Our algorithm enumerates all choices for the point q, in order of decreasing distance from ℓ. For the current choice of q, it maintains CH(Q'), say as an AVL tree, and, for each edge e' of CH(Q'), the distance to the corresponding supporting line $\ell_{e'}$ to CH(P) (not to CH(Q')). Edges with distances are stored in a min-heap; the minimum such distance gives the minimum width for S_2 for the current choice of S_1; the larger of the two determines the width of the current V-shape. We record the best width of any V-shape encountered in the process.

The algorithm is initialized with Q' set to the set consisting of the point of P furthest from ℓ. A generic step of the algorithm involves moving the current point q from Q to Q'. We update the convex hull of Q' by computing the supporting tangents from q to the old hull, in $O(\log n)$ time. For the two new hull edges e_1, e_2,

---

[2] If $\text{width}(R(V)) = 0$, we have $Q' \subset \ell'$ and $Y_2 \subset \ell'$.

we compute the corresponding supporting lines ℓ_{e_1}, ℓ_{e_2} of CH(P), using a suitable balanced-tree representation of CH(P), also in logarithmic time. We add the new edges with the corresponding widths to the min-heap, after removing from it the entries of all the eliminated edges of CH(Q'). The root of the min-heap yields the best width for S_2 for the current partition $\{Q, Q'\}$. The algorithm requires presorting points by distance from ℓ and then a linear number of balanced-search-tree and heap operations (since the number of edges inserted is less than $2n$ and each cannot be deleted more than once), for a total running time of $O(n \log n)$ for a fixed e, as claimed.

Working through the entire set P (except for the endpoints of e), in order of decreasing distance from ℓ, growing Q' and shrinking Q, we obtain a sequence of a linear number of V-shapes (which include all the canonical minimum-width V-shapes covering P) with e on its outer boundary and two other points of P lying on the opposite arm's inner boundary. It is not difficult to check (the details are omitted in this version) that every combination $(e, q, e', \ell_{e'})$ examined by the algorithm yields a valid V-shape covering P. In the list of all V-shapes considered by the algorithm will appear all the canonical V-shapes of the inner-outer type and therefore a minimum-width canonical V-shape of this type will be included.

To summarize, inner-outer type V-shapes can be handled in total time $O(n^2 \log n)$.

V-shapes of both-inner type. Now a covering V-shape V has points a, b of P on its inner ray X_1 and points c, d on X_2; refer to the figure below; points a, b, c, d are in convex position, in this counterclockwise order. It is known [16] that there are at most $O(n^2)$ such *wedges* $W = W(a, b, c, d)$ determined by some quadruple of points $a, b, c, d \in P$ and empty of points of P; note that W determines V, so it is sufficient to enumerate all empty wedges W.
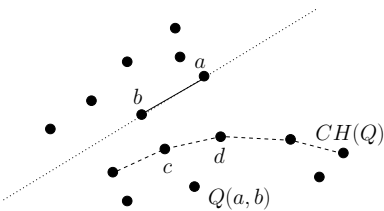
For a pair $a, b \in P$, we compute all pairs c, d, so that $W(a, b, c, d)$ is an empty wedge. Let $Q(a, b)$ be the set of all points of P lying to the left of the directed line ab.

Observation 1. $W(a, b, c, d)$, *in the above notation, is an empty wedge if and only if line cd supports CH(Q) and separates segment ab from $Q = Q(a, b)$ (and a, b, c, d are in this counterclockwise order).*

Now enumerating all k pairs c, d for a fixed choice of a, b can be done in time $O((k + 1) \log n)$, as follows. While handling V-shapes of both-outer type we constructed a data structure $D(P)$ which, for a given line (here ab), produces a balanced search tree storing the convex hull of the points of P lying to one side of the line (here $Q = Q(a, b)$). Using $D(P)$, we find the point z of Q closest to the line ab and traverse the boundary of CH(Q) in both directions from z, to list all edges cd of CH(Q) satisfying the conditions of the above observation. It is sufficient to examine $k + 2$ edges of CH(Q). Repeating the procedure for all choices of a, b and recalling that the number of empty wedges is at most quadratic, we deduce that the enumeration algorithm runs in time $O(n^2 \log n)$.

4   A 13-Approximation Algorithm

Let w be the minimum value such that a set of points P can be covered by a V-shape of width w. We present an algorithm that computes a V-shape covering P of width at most $13w$ in time $O(n \log n)$. For this purpose, we use the $O(n \log n)$ time 6-approximation algorithm for the 2-line-center problem presented by Agarwal, Procopiuc, and Varadarajan [2]. The 2-line-center problem is the following: Given a set P of n points in $\mathbb{R}^2$, cover P by two congruent strips of minimum width.

Observation 2. *If w' is the width of two congruent strips of minimum width covering P, $w' \leq w$.*

Our 13-approximation algorithm is as follows. Use the 6-approximation algorithm of [2] to compute two congruent strips of width w'' that cover P; $w' \leq w'' \leq 6w'$. Find the median lines ℓ_1 and ℓ_2 of the strips. For all points in each strip, project them onto ℓ_1 and ℓ_2 respectively (the points in the intersection of the strips are duplicated and projected onto both ℓ_1 and ℓ_2). Let P' be the resulting set of projected points. Compute an exact minimum width V-shape V' covering P' (see Section 4.1) in $O(n \log n)$ time. The desired approximate V-shape V is obtained by widening V' by $w''/2$ in all directions.

Note that it is possible that the two strips computed above are such that a V-shape defined by them contains P. In this case we return that V-shape. This clearly produces a 6-approximation, due to Observation 2. In the remainder of this section, we will assume that this is not the case, in other words, one of the two strips has points of P on both sides of it.

Theorem 3. *This algorithm computes a 13-approximation of a minimum-width V-shape covering P.*

Proof. Let V_{best} be a minimum-width V-shape of P, V' — a minimum-width V-shape of P', and V_{apx} — the approximate V-shape computed by the algorithm. As the points of P have been moved by a distance of at most $w''/2$ to form P', $\mathrm{width}(V') \leq \mathrm{width}(V_{\mathrm{best}}) + w''$. Since V_{apx} is a widened version of V', it contains the points of P. Moreover, $\mathrm{width}(V_{\mathrm{apx}}) \leq \mathrm{width}(V') + w'' \leq \mathrm{width}(V_{\mathrm{best}}) + 2w'' \leq w + 12w' \leq 13w$ by Observation 2. $\qquad\square$

Remark. Using the $(1 + \varepsilon)$-approximation algorithm of [2] in place of their 6-approximation algorithm in our procedure, we can attain any approximation factor larger than three for the minimum-width V-shape. The running time remains $O(n \log n)$, with the constant of proportionality depending on the quality of the approximation. We do not discuss this extension further, since we present our own $(1 + \varepsilon)$-approximation algorithm for the problem in Section 5.

4.1   Minimum-Width V-Shape for Points on Two Lines

We now describe how to compute the minimum-width V-shape, given a point set P' contained in the union of two lines ℓ_1, ℓ_2 in the plane. Put $z := \ell_1 \cap \ell_2$.

Let $P_1' := P' \cap \ell_1$, and $P_2' := P' \cap \ell_2$. As we have assumed that ℓ_1, ℓ_2 do not already form a V-shape containing P', ℓ_1 separates P_2' in two sets and/or ℓ_2 separates P_1'. The convex hull $\mathrm{CH}(P')$ has three or four vertices. Moreover, by reasoning similar to that of Section 3, the outer boundary of V' contains two, three, or four vertices of $\mathrm{CH}(P')$ (in the case where an outer ray is contained in ℓ_1 or ℓ_2, we consider only the extreme points). Before describing how we handle these cases, we need a technical lemma, whose proof we omit in this version of the paper.

Lemma 1. *Given a line partitioning P' into P_r', P_ℓ' and given their convex hulls $\mathrm{CH}(P_r')$, $\mathrm{CH}(P_\ell')$, the minimum-width canonical V-shape V' of P' containing P_r' in one strip and P_ℓ' in the other can be computed in constant time; some points of P' might lie in both strips of V'.*

Now we consider the different types of canonical V-shapes covering P' and describe how to find a minimum-width V-shape of each type.

Case 1: An outer bounding ray of V' contains an edge e of $\mathrm{CH}(P')$. Let ℓ be the line containing e. For all points p of P', draw a line ℓ_p through p and parallel to ℓ. Apply Lemma 1 to (the partition induced by) ℓ_p. This can be implemented to run in overall time $O(n \log n)$.

In the remaining cases, each of the outer rays of V' contains precisely one vertex of $\mathrm{CH}(P')$ and each inner ray contains two points of P'.

Case 2: An inner ray of V' lies on ℓ_1 or ℓ_2. Suppose an inner ray of V' is contained in ℓ_1. Draw two lines parallel to ℓ_1 and very close to it, one to the left of ℓ_1, one to the right of ℓ_1. Apply Lemma 1 to each of these two lines.

Case 3: Point $z = \ell_1 \cap \ell_2$ lies between the two arms of V'. Draw the two lines passing through z and bisecting the angles between $\ell_1 \cap \ell_2$. Apply Lemma 1 to each of these two lines.

Case 4: Point z is inside an arm of V'. For each pair of consecutive points $p, q \in P'$ on ℓ_1 or on ℓ_2 not separated by z, apply Lemma 1 to the perpendicular bisector of the segment pq.

In the full version of the paper, we argue that the last procedure returns the best minimum-width V-shape V' of P' with two points on its outer boundary and z in one of its arms, correctly handling case 4 and thereby concluding our description.

5   A $(1 + \varepsilon)$-Approximation Algorithm

In this section we describe how to construct, given a point set P and a real number $\varepsilon > 0$, a V-shape V covering P, with $\mathrm{width}(V) \leq (1 + \varepsilon)w_{\mathrm{opt}}$, where w_{opt} is the width of a minimum-width V-shape covering P.

We start by recalling the notion of an anchor pair used in [2]. Given a V-shape V covering P, fix one of the strips of V, say S_1. We say that a pair of points $p, q \in P \cap S_1$ is an *anchor pair*, if $\mathrm{dist}(p, q) \geq \mathrm{diam}(P \cap S_1)/2$. Lemma 3.3 in [2] describes how to identify at most 11 pairs of points in P, such that, for any two-strip cover of P, at least one of the pairs is an anchor pair for one of

the strips; the algorithm requires $O(n \log n)$ time. As covering by a V-shape is a special case of covering by two strips, the definition and the algorithm apply here as well.

We show how to, given a potential anchor pair p, q, construct a $(1 + \varepsilon)$-approximation of the minimum-width V-shape covering P for which p, q is an anchor pair. More precisely, below we prove

Lemma 2. *Given a potential anchor pair $p, q \in P$, we can construct, in time $O((n/\varepsilon) \log n + (n/\varepsilon^{3/2}) \log^2(1/\varepsilon))$, a V-shape covering P, of width at most $1 + \varepsilon$ times the minimum width of any V-shape covering P for which p, q is an anchor pair.*

Applying this procedure at most 11 times, we obtain our desired approximation algorithm:

Theorem 4. *A V-shape covering P and of width at most $(1 + \varepsilon)w_{\mathrm{opt}}$ can be constructed in time $O((n/\varepsilon) \log n + (n/\varepsilon^{3/2}) \log^2(1/\varepsilon))$.*

We first prove that it is sufficient to consider those V-shapes V with anchor pair p, q, for which the strip containing p, q has one of a small set of fixed directions. Setting $\beta := \sin^{-1} \min\{\varepsilon \mathrm{width}(V)/(6d(p, q)), 1\}$ and $\gamma := \beta + \sin^{-1}(\min\{1, \mathrm{width}(V)/d(p, q)\})$, we prove the following

Lemma 3. *Let V-shape V cover P, and let p, q be an anchor pair for $S_1(V)$. Rotating $S_1(v)$ by an angle at most β does not increase the width of the V-shape by more than a factor of $1 + \varepsilon/3$, and the angle between pq and the direction of the rotated strip cannot exceed γ.*

Proof. Put $w := \mathrm{width}(V)$. Let B be the minimum bounding box of $P \cap S_1$. More precisely, it is the shortest rectangle cut out of S_1 by two lines perpendicular to S_1 and containing $P \cap S_1$; refer to Figure 1. Let s and $t \leq w$ be the length (along the axis of S_1) and width of B, respectively. Let S_1' be the minimal parallel strip containing $B \cap V$, whose direction is $\alpha \leq \beta$ away from that of S_1 (there are two choices for S_1', corresponding to rotating clockwise and counterclockwise; only one is shown; the argument applies to both cases). Then
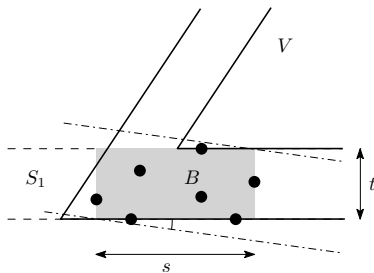
Fig. 1. Rotation by α does not change the width by much

$$\text{width}(S_1') \le s \sin \alpha + t \cos \alpha \le 2d(p,q) \sin \alpha + w$$

$$\le w\left(1 + 2\frac{d(p,q)}{w} \sin \alpha\right) \le t(1 + \varepsilon/3),$$

since $\sin \alpha \le \sin \beta \le \varepsilon w/(6d(p,q))$. Now replace S_1 by S_1' to obtain a new V-shape V' covering P. Its width is $\min\{\text{width}(S_1'), \text{width}(S_2)\} \le (1 + \varepsilon/3)w$, as claimed.

Observe that in the above construction, the angle between pq and the direction of S_1' cannot exceed

$$\alpha + \sin^{-1}(\min\{1, t/d(p,q)\}) \le \beta + \sin^{-1}(\min\{1, \text{width}(V)/d(p,q)\}) = \gamma. \quad \square$$

We conclude that enumerating all V-shapes that contain p, q in their strip S_1 and whose directions are (a) at most γ away from that of $d(p,q)$ and (b) spaced at most β apart, would yield a V-shape whose existence is claimed in Lemma 2. The number of directions to be tested is at most $O(\gamma/\beta) = O(1/\varepsilon)$.

Given a candidate anchor pair p, q, the algorithm proceeds by starting with the direction pq. Since we need not consider V-shapes whose width is larger than the approximate width w_{apx} computed in Section 4 (this is where the 13-approximation algorithm is used to bootstrap our $(1 + \varepsilon)$-approximation), we replace $\text{width}(V)$ by the smaller $w_{\text{apx}}/13$ in the definition of β above and by the larger w_{apx} in the definition of γ, thereby erring on the conservative side in each case. Having computed (conservative estimates of) β and γ, we enumerate the $O(1/\varepsilon)$ directions of the form $\theta_i := \theta_{pq} + i\beta$, where θ_{pq} is the direction of pq and i is an integer ranging from $-\lceil\gamma/\beta\rceil$ to $\lceil\gamma/\beta\rceil$. It remains to explain how to deal with one such direction $\theta := \theta_i$.

Lemma 4. *One can compute a canonical V-shape V covering P with one arm in given direction θ and width at most $1 + \varepsilon/3$ times the minimum width of any such V-shape, in time $O(n \log n + (n/\varepsilon^{1/2}) \log^2(1/\varepsilon))$.*

Proof. We use an approach similar to that of the inner-outer case of our exact algorithm but with a slight twist.

Let ℓ be a line in direction θ supporting $\text{CH}(P)$. We again let q be the furthest point from ℓ in $Q := P \cap S_1$ and let $Q' := P \setminus Q$. When q is fixed, the minimum-width V-shape is determined by the minimum-width strip S_2 covering Q' and not "splitting" P, i.e., such that it does not have points of P on both sides of it. It is easy to ensure that S_2 does not split P by observing that a direction of S_2 lying between the directions of the common outer tangents to $\text{CH}(Q)$ and $\text{CH}(Q')$ is never useful. Depending on the side where the lines supporting these tangents cross, a minimal strip S_2 covering Q' and lying in the range between them either crosses Q (and therefore P) or completely covers Q (and therefore P). In the former case, S_1 and S_2 do not form a legal V-shape covering P and in the latter they form a covering V-shape with one empty strip, which never yields minimum width (proof omitted in this version).

The width of the resulting V-shape is the maximum of $\text{dist}(q, \ell)$ and (the restricted) $\text{width}(S_2)$. The algorithm proceeds by processing points q in order of

decreasing distance to ℓ, keeping track of $\mathrm{dist}(q, \ell)$ and a *coreset* for Q', which is a subset of Q' with the property that its directional width, in every direction, is at least $1 - \varepsilon/3$ that of Q' (and expanding it by $1 + \varepsilon/3$ we get a strip covering Q'). Chan [9], in Theorem 3.7 and remarks in Section 3.4, describes a streaming algorithm that maintains an $O(1/\varepsilon^{1/2})$-size coreset at an amortized cost of $O((1/\varepsilon^{1/2}) \log^2(1/\varepsilon))$ per insertion. For a fixed q, we go through the coreset (after computing its convex hull, if necessary), and determine the narrowest strip covering it and satisfying our angle constraints. The maximum of that and $\mathrm{dist}(q, \ell)$ gives the width of the minimum-width V-shape whose boundary passes through q.[3] The amortized cost per point is dominated by the $O((1/\varepsilon^{1/2}) \log^2(1/\varepsilon))$ cost of insertion. Together with presorting points by distance from ℓ, the total cost is then $O(n \log n + (n/\varepsilon^{1/2}) \log^2(1/\varepsilon))$. □

Combining Lemmas 3 and 4 yields the procedure claimed in Lemma 2 and thereby completes our description of the $(1 + \varepsilon)$-approximation algorithm.

6   Concluding Remarks

As mentioned in the introduction, this work has been inspired by research on curve fitting, in the situations where a curve takes a sharp turn. Besides the exact and approximate versions of the problem studied above, it would be natural to investigate a variant that can handle a small number of outliers. A natural "peeling" approach to the problem would be to eliminate the points defining the optimal V-shape found by our exact algorithm and trying again. We leave this investigation for the full version of this paper.

Are there natural assumptions (perhaps in the style of "realistic input models" [7] or in the form of requiring reasonable sampling density) that would be relevant for the curve-fitting problem, and that would make finding the minimum-width covering V-shape easier?

Returning to the problem studied in the paper, is it possible to find an exact minimum-width covering V-shape in subquadratic time? Is the problem 3SUM-hard?

Is it possible to speed up the approximation algorithm, improving the dependence of its running time on ε? Is time $O(n + f(\frac{1}{\varepsilon}))$ achievable?

Finally, we would like to point out that there are other "reasonable" definitions for a V-shape, if the goal is to approximate a sharp turn of a curve: One can imagine defining a V-shape as the Minkowski sum of a disk with the union of two rays emanating from a common point as in [14]. The width of the V-shape would be the diameter of the disk. Can the exact algorithm from [14] be sped up? Is there a faster approximation algorithm? Is this version of the problem better suited for curve fitting?

---

[3] More precisely, q lies on the boundary of S_1 and may not even appear on the boundary of V. However, as before, all V-shapes we examine are valid and cover P, and the desired approximating V-shape is among them, which is sufficient.

References

1. Agarwal, P.K., Har-Peled, S., Varadarajan, K.R.: Geometric approximation via coresets. In: Goodman, J.E., Pach, J., Welzl, E. (eds.) Current Trends in Combinatorial and Computational Geometry, vol. 52, pp. 1–30. MSRI Publications, Cambridge University Press, New York (2005)
2. Agarwal, P.K., Procopiuc, C.M., Varadarajan, K.R.: A $(1 + \epsilon)$-approximation algorithm for 2-line-center. Comput. Geometry: Theory Appl. 26(2), 119–128 (2003)
3. Agarwal, P.K., Sen, S.: Randomized algorithms for geometric optimization problems. In: Pardalos, J., Rajasekaran, S., Reif, J., Rolim, J. (eds.) Handbook of Randomized Computation, pp. 151–201. Kluwer Academic Press, The Netherlands (2001)
4. Agarwal, P.K., Sharir, M.: Algorithmic techniques for geometric optimization. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 234–253. Springer, Heidelberg (1995)
5. Agarwal, P.K., Sharir, M.: Efficient algorithms for geometric optimization. ACM Computing Surveys 30, 412–458 (1998)
6. Alt, H., Guibas, L.J.: Discrete geometric shapes: matching, interpolation, and approximation. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, ch.3, pp. 121–153 (1999)
7. de Berg, M., Katz, M., van der Stappen, A.F., Vleugels, J.: Realistic input models for geometric algorithms. Algorithmica 34(1), 81–97 (2008)
8. Chan, T.M.: Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus. Inter. J. Comput. Geom. Appl. 12, 67–85 (2002)
9. Chan, T.M.: Faster core-set constructions and data-stream algorithms in fixed dimensions. Comput. Geom. Theory Appl. 35, 20–35 (2006)
10. Cheng, S.-W., Funke, S., Golin, M., Kumar, P., Poon, S.-H., Ramos, E.: Curve reconstruction from noisy samples. Comput. Geom. Theory Appl. 31(1-2), 63–100 (2005)
11. Dey, T.K.: Curve and Surface Reconstruction: Algorithms with Mathematical Analysis. In: Cambridge Monographs on Applied and Computational Mathematics, vol. 23. Cambridge Univ. Press, New York (2007)
12. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Computer System Sci. 38(1), 86–124 (1989)
13. Funke, S., Ramos, E.A.: Reconstructing a Collection of Curves with Corners and Endpoints. In: Proc. 12th ACM-SIAM Symp. Discr. Algorithms, pp. 344–353 (2001)
14. Glozman, A., Kedem, K., Shpitalnik, G.: Computing a double-ray center for a planar point set. Inter. J. Comp. Geom. Appl. 2(9), 103–123 (1999)
15. Mhatre, A., Kumar, P.: Projective clustering and its application to surface reconstruction: extended abstract. In: Proc. 22nd Annu. Symp. Comput. Geom., pp. 477–478 (2006)
16. Pinchasi, R., Radoičić, R., Sharir, M.: On empty convex polygons in a planar point set. Combinat. Theory, Series A 113, 385–419 (2006)
17. Preparata, F.P.: An optimal real time algorithm for planar convex hulls. Comm. ACM 22, 402–405 (1979)

Witness Rectangle Graphs⋆

Boris Aronov[1], Muriel Dulieu[1], and Ferran Hurtado[2]

[1] Department of Computer Science and Engineering, Polytechnic Institute of NYU,
Brooklyn, NY 11201-3840, USA
aronov@poly.edu, mdulieu@gmail.com
[2] Departament de Matemàtica Aplicada II,
Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
ferran.hurtado@upc.edu

Abstract. In a *witness rectangle graph* (WRG) on vertex point set P with respect to witness point set W in the plane, two points x, y in P are adjacent whenever the open rectangle with x and y as opposite corners contains at least one point in W. WRGs are representative of a larger family of witness proximity graphs introduced in two previous papers.

We study graph-theoretic properties of WRGs. We prove that any WRG has at most two non-trivial connected components. We bound the diameter of the non-trivial connected components of a WRG in both the one-component and two-component cases. In the latter case, we prove that a graph is representable as a WRG if and only if each component is a co-interval graph, thereby providing a complete characterization of WRGs of this type. We also completely characterize trees drawable as WRGs.

Finally, we conclude with some related results on the number of points required to stab all the rectangles defined by a set of n points.

1   Introduction

Proximity graphs have been widely used in situations in which there is a need of expressing the fact that some objects in a given set—which are assigned to nodes in the graph—are close, adjacent, or neighbors, according to some geometric, physical, or conceptual criteria, which translates to edges being added to the corresponding graph. In the geometric scenario the objects are often points and the goal is to analyze the shape or the structure of the set of spatial data they describe or represent. This situation arises in Computer Vision, Pattern Recognition, Geographic Information Systems, and Data Mining, among other fields. The paper [19] is a survey from this viewpoint, and several related papers appear in [27]. In most proximity graphs, given a point set P, the adjacency

between two points $p, q \in P$ is decided by checking whether in their *region of influence* there is no other point from P, besides p and q. One may say that the presence of another point is considered an *interference*. There are many variations, depending on the choice of the family of influence regions [19, 9, 21].

Given a combinatorial graph $G = (V, E)$, a *proximity drawing* of G consists of a choice of a point set P in the plane with $|P| = |V|$, for a given criterion of neighborhood for points, such that the corresponding proximity graph on P is isomorphic to G. This question belongs to the subject of *graph drawing problems*, in which the emphasis is on geometrically representing graphs with good readability properties and fulfilling some aesthetic criteria [4]. The main issues are to characterize the graphs that admit a certain kind of representation, and to design efficient algorithms for finding such a drawing, whenever possible.

Proximity drawings have been studied extensively and utilized widely [5, 21]. However, this kind of representation is somehow limited and there have been some attempts to expand the class, for example using weak proximity graphs [6]. Another recently introduced generalization is the concept of *witness proximity graphs* [1, 2], in which the adjacency of points in a given vertex set P is decided by the presence or absence of points from a second point set W—the *witnesses*—in their region of influence. This generalization includes the classic proximity graphs as a particular case, and offers both a stronger tool for neighborhood description and much more flexibility for graph representation purposes.

In the *positive witness* version, there is an adjacency when a witness point is covered by the region of influence. In the *negative witness* version, two points are neighbors when they admit a region of influence free of any witnesses. In both cases the decision is based on the presence or absence of witnesses in the regions of influence, and a combination of both types of witnesses may also be considered. Observe that by taking $W = P$ playing a negative role, we recover the original proximity graphs; so this is a proper generalization. Witness graphs were introduced in [1], where the focus is on the generalization of *Delaunay graphs*. The witness version of *Gabriel graphs* was studied in [2], and a thorough exploration of this set of problems is the main topic of the thesis [14].

In this paper, we consider a positive witness proximity graph related to the rectangle-of-influence graph, the witness rectangle graph. In the *rectangle of influence graph* RIG(P), usually studied as one of the basic proximity graphs [22, 21], $x, y \in P$ are adjacent when the *rectangle $B(x, y)$ they define* covers no third point from P; $B(x, y)$ is the unique open isothetic rectangle with x and y at its opposite corners. The *witness rectangle graph* (WRG) of vertex point set P (or, simply, *vertices*) with respect to witness point set W (*witnesses*), denoted RG$^+(P, W)$, is the graph with the vertex set P, in which two points $x, y \in P$ are adjacent when the rectangle $B(x, y)$ contains at least one point of W. The graph RG$^+(P, \varnothing)$ has no edges. When W is sufficiently large and appropriately distributed, RG$^+(P, W)$ is complete. We also note that a negative-witness version of this graph with $W = P$ would be precisely RIG(P) discussed above; in fact RG$^+(P, P)$ is precisely the complement of RIG(P). An example is shown in Figure 1. We show in this paper that the connected components of WRGs are
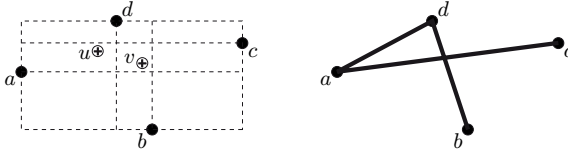
Fig. 1. Left: A set of points $P = \{a, b, c, d\}$ and a witness set $W = \{u, v\}$. Right: the witness rectangle graph $\mathrm{RG}^+(P, W)$. In all our figures for WRGs solid dots denote vertices and dots with a cross denote (positive) witnesses.

geometric examples of graphs with small diameter; these have been attracting substantial attention in the pure graph theory setting, and are far from being well understood, even for diameter two [16, 24, 25].

Besides some computational issues, such as the construction of $\mathrm{RG}^+(P, W)$ for given sets P and W in an output-sensitive manner, in this paper we study several graph-theoretic properties of WRGs: (a) We completely characterize trees drawable as WRGs (Theorem 2). (b) We argue that any WRG has zero, one, or two *non-trivial* connected components (see the definition below and Theorem 3). (c) We prove that the diameter of a single-component WRG is at most six, and that this bound is tight in the worst case (Theorem 3 and subsequent discussion). (d) We prove that the diameter of a (non-trivial) connected component of a two-component WRG is at most three and this can be achieved in the worst case (Theorem 3). (e) In the two-component case, we provide a complete characterization of graphs representable as WRGs. Such graphs, disregarding isolated vertices, are precisely disjoint unions of two co-interval graphs (Theorem 4). This last result allows us to recognize in polynomial time if a combinatorial graph with two non-trivial components can be drawn as a WRG.

Finally, in Section 5, we present some related results on stabbing rectangles defined by a set of points with other points. They can be interpreted as questions on "blocking" rectangular influences.

Terminology and notation. Throughout the paper, we will work with finite point sets in the plane, in which no two points lie on the same vertical or the same horizontal line.

Hereafter, for a graph $G = (V, E)$ we write $xy \in E$ or $x \sim y$ to indicate that $x, y \in V$ are adjacent in G, and generally use standard graph terminology as in [8]. When we speak of a *non-trivial connected component* of a graph, we refer to a connected component with at least one edge (and at least two vertices).

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with disjoint vertex sets, their *join* is the graph $G_1 + G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$ [28].

How to compute a witness rectangle graph. De Berg, Carlsson, and Overmars [7] generalized the notion of dominance in a way that is closely related to witness rectangle graphs by defining dominance pairs p, q of a set of points P, with respect to a set O of so-called obstacle points. More precisely, p is said to *dominate* q *with respect to* O if there is no point $o \in O$ such that p dominates o and o

dominates q. Recall that $p = (p_1, p_2)$ *dominates* $q = (q_1, q_2)$ if and only if $p_i \geq q_i$, for $i = 1, 2$, and $p \neq q$.

Moreover they prove the following theorem:

Theorem 1 (De Berg, Carlsson, and Overmars [7]). *All dominance pairs in a set of points P with respect to a set of points O can be computed in time $O(n \log n + k)$, where $n = |P| + |O|$ and k is the number of answers.*

Collecting all dominance pairs in a set of points P with respect to a set of points W, and repeating the procedure after rotating the plane by $90°$, one obtains the negative version of the witness rectangle graph. A simple modification of their algorithm yields the positive version:

Corollary 1. *Let P and W be two point sets in the plane. The witness rectangle graph $\mathrm{RG}^+(P, W)$ with k edges can be computed in $O(k + n \log n)$ time, where $n := \max\{|P|, |W|\}$.*

2   Structure of Witness Rectangle Graphs

Let $G := \mathrm{RG}^+(P, W)$ be the witness rectangle graph of vertex set P with respect to witness set W. We assume that the set of witnesses is *minimal*, in the sense that removing any one witness from W changes G. Put $n := \max\{|P|, |W|\}$ and let $E := E(G)$ be the edge set of G. We partition E into E^+ and E^- according to the slope sign of the edges when drawn as segments. Slightly abusing the terminology we refer to two edges of E^+ (or two edges of E^-) as having *the same slope* and an edge of E^+ and an edge of E^- as having *opposite slopes*.

Recall that the open isothetic rectangle (or *box*, for short) defined by two points p and q in the plane is denoted $B(p, q)$; for an edge $e = pq$ we also write $B(e)$ instead of $B(p, q)$. Every edge e, say in E^+, defines four regions as in the figure on the right, that we call *(open) corners* and *(closed) bays*.
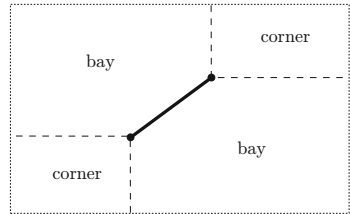
Observation 1. *Every $x \in P$ inside a corner of an edge e is adjacent to at least one endpoint of e.*

Note that for any P, W, and $P' \subset P$, the graph $\mathrm{RG}^+(P', W)$ is an induced subgraph of $\mathrm{RG}^+(P, W)$, so the class of graphs representable as WRGs is closed under the operation of taking induced subgraphs.

Two edges are *independent* when they share no vertices and the subgraph induced by their endpoints contains no third edge. Below we show that G cannot contain three pairwise independent edges, which imposes severe constraints on the graph structure of G.

Lemma 1. *Two independent edges in E^+ (respectively, E^-) cannot cross or share a witness. The line defined by their witnesses is of negative (respectively, positive) slope.*

Proof. Let the two edges be $ab, cd \in E^+$, with $x(a) < x(b)$ and $x(c) < x(d)$. A common witness would have a and c in its third quadrant and b and d in the first, implying $a \sim d$ and $c \sim b$, a contradiction. If ab and cd cross, assume without loss of generality that $x(a) < x(c)$. Neither c nor d can be inside $B(a, b)$ (because of Observation 1) and hence $B(a, d) \cup B(c, b) \supset B(a, b)$, implying $a \sim d$ or $c \sim b$, a contradiction. Finally, the second part of the statement is a direct consequence of Observation 1. $\square$

Lemma 2. *Two independent edges with opposite slopes must share a witness.*

Proof. Let $ab \in E^+$ and $cd \in E^-$ be independent. Let w be a witness for ab and let w' be a witness for cd. The points c and d are not in quadrants I or III of w, as otherwise the two edges would not be independent. If w is shared, we are done. Otherwise it cannot be that c lies in quadrant II of w while d lies in quadrant IV, or vice versa. Therefore c and d are either both in quadrant II or both in quadrant IV of w. Assume, without loss of generality, the former is true. The witness w' is not outside of $B(a, b)$, as we would have $c \sim a$ and/or $c \sim b$ (assuming, without loss of generality, that $x(c) < x(d)$) and the edges would not be independent. Therefore w' is in $B(a, b)$, so w' is a shared witness, as claimed. $\square$

Lemma 3. *There are no three pairwise independent edges in E^+ (or in E^-).*

Proof. Assume that three pairwise independent edges e_1, e_2, e_3 in E^+ are witnessed by w_1, w_2, w_3, respectively, with $x(w_1) < x(w_2) < x(w_3)$. Then, by Lemma 1, $y(w_1) > y(w_2) > y(w_3)$. By the same lemma at least one endpoint of e_1 is in the second quadrant of w_2 and at least one endpoint of e_3 is in its fourth quadrant, contradicting their independence. $\square$

Lemma 4. *G does not contain three pairwise independent edges.*

Proof. By Lemma 3, two edges ab and cd of the three pairwise independent edges ab, cd, and ef have opposite slopes. By Lemma 2, ab and cd share a witness point w. Every quadrant of w contains one of the points a, b, c, or d, therefore both e and f must be adjacent to one of them, a contradiction. $\square$

The preceding results allow a complete characterization of the trees that can be realized as WRGs. An analogous result for rectangle-of-influence graphs was given in [22].

Theorem 2. *A tree is representable as an WRG if and only if it has no three independent edges.*

Sketch of the proof. Examine all combinatorial trees without three independent edges. This requires a case analysis that we omit in this version. Any such tree is a subtree of one of the two maximal trees depicted in Figure 2, both of which happen to be representable as WRGs, as seen in the figure. $\square$
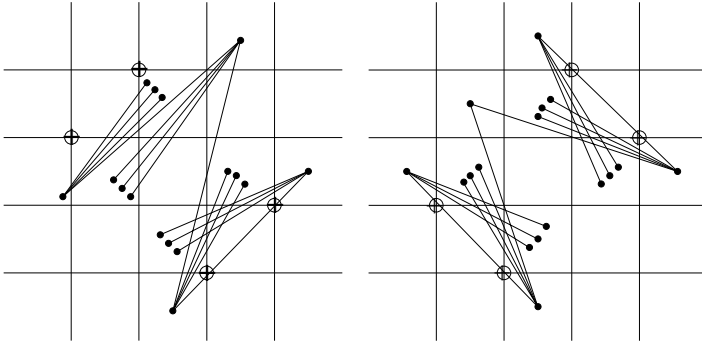
Fig. 2. All WRG trees have this form. Any number of vertices can be added in the regions containing three vertices.

Lemma 4 immediately implies the following structural result that is far from a complete characterization, yet narrows substantially the class of graphs representable as WRGs.

Theorem 3. *A WRG has at most two non-trivial connected components. If there are exactly two, each has diameter at most three. If there is one, its diameter is at most six.*

Note that the bounds on the diameter are tight: the tree in Figure 2 (right) has diameter six and it is easy to draw the disjoint union of two three-link paths as a WRG, by removing one vertex from Figure 2 (right), for example.

3   Two Connected Components

In this section we define a subclass of witness rectangle graphs, called *staircase graphs*. We argue that a WRG with precisely two non-trivial connected components has a very rigid structure. Namely, each of its non-trivial connected components is isomorphic to a staircase graph.

Definition 1. *A* staircase graph of type IV *is a witness rectangle graph, such that the witnesses form an ascending chain (i.e., for every witness, other witnesses lie only in its quadrants I and III) and all the vertices lie above the chain (i.e., quadrant IV of every witness is empty of vertices); refer to Figure 3.*
*   Staircase graphs of types I, II, and III are defined analogously; they are rotated versions of the above. The type of the staircase graph is determined by which quadrant of all witnesses is empty of vertices.*

Note that an induced subgraph of a staircase graph is a staircase graph (of the same type)—a property that immediately follows from the definition and that we will find useful below.
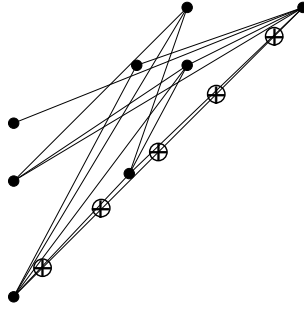
Fig. 3. Staircase graph of type IV

Lemma 5. *A combinatorial graph $G = (V, E)$, isomorphic to a staircase graph, is a join $G_1 + G_2$ if and only if it can be realized as a staircase graph of type IV with some witness containing points corresponding to $V(G_1)$ in quadrant I and points corresponding to $V(G_2)$ in quadrant III.*

Proof. Suppose $G = G_1 + G_2$. The combinatorial graphs G_1 and G_2 are isomorphic to staircase graphs $\mathrm{RG}^+(P_1, W_1)$ and $\mathrm{RG}^+(P_2, W_2)$ of type IV, which can be obtained, for example, by considering any realization of G as a staircase graph of type IV and dropping the points corresponding to $V(G_2)$ and $V(G_1)$, respectively. Create a staircase graph of type IV isomorphic to G by placing a copy of $\mathrm{RG}^+(P_1, W_1)$ in quadrant I of a new witness w and a copy of $\mathrm{RG}^+(P_2, W_2)$ in its quadrant III.

Conversely, given a staircase graph of type IV isomorphic to G such that some of its vertices (call the set P_1) are in the first quadrant of a witness $w \in W$, and the remaining vertices (call them P_2) are in its third quadrant, it is easily checked that $G = G_1 + G_2$, where G_1 and G_2 are the subgraphs of G induced by (the sets of vertices of G corresponding to) P_1 and P_2, respectively. □

Lemma 6. *In a WRG, if witness w has no vertices in one of its quadrants, any witness in the empty quadrant is redundant.*

Proof. Suppose quadrant II of w contains a witness w' but no vertices. Hence quadrant II of w' is empty of vertices as well. Suppose w' witnesses edge ab, and a and b are in its quadrants I and III respectively. (They cannot lie in quadrants II and IV, as quadrant II is empty of vertices.) As quadrant IV of w is included in quadrant IV of w', a and b must be in quadrants I and III of w, respectively, as well. Therefore w witnesses ab. Since this argument applies to all edges witnessed by w', w' is redundant. □

Lemma 7. *In a WRG, if no witness has a vertex in its quadrant IV, then the graph is a staircase graph of type IV (possibly after removing some redundant witnesses).*

Proof. First remove any redundant witnesses, if present. Now apply Lemma 6, to the possibly smaller, new witness set, to conclude that every remaining witness

has its quadrant IV empty of witnesses as well. Therefore the remaining witnesses form an ascending chain and all vertices lie above it, as in the definition of a staircase graph of type IV.   □

Of course, if the empty quadrant in the above lemma is not IV but I, II, or III, we get a staircase graph of the corresponding type.

Theorem 4. *In a witness rectangle graph with two non-trivial connected components, each component is isomorphic to a staircase graph. Conversely, the disjoint union of two graphs representable as staircase graphs is isomorphic to a witness rectangle graph.*

Proof. We distinguish two cases.

All edges have the same slope: Suppose all edges of components C_1, C_2 have the same slope, say positive. Let ab be an edge of C_1 witnessed by w_1 and cd be an edge in C_2 witnessed by w_2, with $x(a) < x(b)$ and $x(c) < x(d)$. By Lemma 1, w_1 and w_2 are distinct. The vertices a and b are in quadrants III and I of w_1, respectively. As c and d are not adjacent to a or b, and as cd doesn't share its witness with ab, c and d are both in quadrant II or both in quadrant IV of w_1. Suppose, without loss of generality, that cd is in quadrant IV of w_1 (see Figure 4). By a symmetric argument, ab is in quadrant II of w_2. This holds for any two edges $ab \in C_1$ and $cd \in C_2$. (Given two edges $ef \in C_1$ and $gh \in C_2$, we say $ef < gh$ if a witness of ef is in quadrant II of a witness of gh, and $gh < ef$ otherwise. We claim that either $ef < gh$ for all choices of edges $ef \in C_1$ and $gh \in C_2$, or $gh < ef$, for all such choices. Otherwise there would have to exist, without loss of generality, a triple of edges $ef, ij \in C_1$ and $gh \in C_2$, with $ef < gh < ij$. This implies that some witnesses w', w'', w'' of ef, gh, ij, respectively, form a descending chain. Considering the relative positioning of the three edges and three witnesses, we conclude that ef and ij must be independent. Hence we have three pairwise independent edges in E^+, contradicting Lemma 3, thereby proving the claim.) Notice that no vertex of C_1 is in quadrant II of any witness of C_1 or it would be connected to C_2. Similarly, no vertex of C_2 is in quadrant IV of any witness of C_2 or it would be connected to C_1. Hence, by Lemma 7, C_1 and C_2 are both staircase graphs.

At least two edges have opposite slopes: There is at least one pair of edges $ab \in C_1$ and $cd \in C_2$ of opposite slopes. Suppose, without loss of generality, that $ab \in E^+$ and $cd \in E^-$. By Lemma 2, ab and cd share a witness w (see Figure 4, center).

Draw isothetic boxes B_1 and B_2, as follows: B_1 is the minimum bounding box of the vertices of C_1, while B_2 is its analog for C_2 (see Figure 4, right).

Consider C_1; we argue that it is isomorphic to a staircase graph; as we can apply the same reasoning to C_2, this will imply the first part of the theorem. By Lemma 2, every edge in $E^+ \cap C_1$ shares some witness with cd; the witness must therefore lie in $B := B_1 \cap B_2$. Let W' be the set of all such witnesses; $w \in W'$. All vertices of C_1 lie in quadrants I and III of every $w' \in W'$, and all vertices of C_2 lie in quadrants II and IV of every w'; otherwise C_1 and C_2 would be connected. Now remove redundant witnesses from W', i.e., pick a minimal subset W'' of
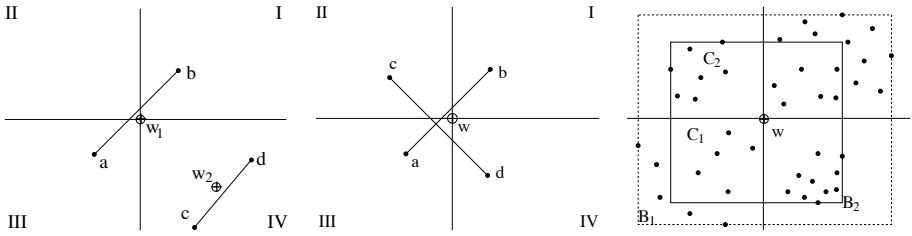
Fig. 4. Cases in the proofs of Lemmas 1, 2, 3, 4 and Theorem 4: Two edges of positive slope (left), two edges of opposite slopes (center). Minimum bounding boxes of C_1: B_1 (dashed), and of C_2: B_2 (plain). C_1 is in quadrants I and III, C_2 is in quadrants II and IV (right).

W such that $H := \mathrm{RG}^+(V(C_1), W')$ coincides with $\mathrm{RG}^+(V(C_1), W'')$. H is a staircase graph and the witnesses of W'' form an ascending chain, by Lemma 7.

Consider the portion of C_1 in quadrant III of w. All witnesses to the left of B_2 and above its lower edge, or below B_2 and to the right of its left edge, have two consecutive quadrants empty of vertices of C_1 (if there were some vertices of C_1 in these quadrants they would be adjacent to vertices of C_2 defining the borders of B_2). Therefore such a witness would not witness any edges and cannot be present in W.

Hence all remaining witnesses (in quadrant III of w) of edges of C_1 must lie below and to the left of B_2 (and, of course, in B_1). All these witnesses have B_2 in their first quadrant, therefore they witness edges of $E^- \cap E(C_1)$. Let W''' be a minimal subset of such witnesses. Each of them has their quadrant III empty of vertices of C_1 (any vertex of C_1 in quadrant III of such a witness w''' would be adjacent to vertices in C_2 as B_2 is in quadrant I of w'''), therefore by Lemma 7, $\mathrm{RG}^+(V(C_1), W''')$ is a staircase graph.

Consider the lowest leftmost witness w_ℓ of W'' (recall that they form an ascending chain). As shown previously, w_ℓ doesn't have any vertex of C_1 in its second and fourth quadrants. Let V_1 be the set of vertices of C_1 in its first quadrant and let V_2 be the vertices of C_1 in its third quadrant. As shown above, $\mathrm{RG}^+(V_1, W'' \setminus \{w'_\ell\})$ and $\mathrm{RG}^+(V_2, W''')$ are staircase graphs, and, therefore, by Lemma 5, $\mathrm{RG}^+(V(C_1), W'' \cup W''')$ is isomorphic to a staircase graph. We apply similar reasoning to quadrant I of B, to conclude that C_1 is a join of at most three staircase graphs and therefore isomorphic to a staircase graph.

C_2 is handled by a symmetric argument (in fact, though it does not affect the reasoning as presented, either C_1 contains negative-slope edges, or C_2 contains positive-slope edges, but not both), concluding the first part of the proof.

Converse: Given two staircase graphs G_1 and G_2, place a scaled and reflected copy of G_1 in quadrant I of the plane, with witnesses on the line $x + y = 1$ and vertices below the line. Place a scaled and reflected copy of G_2 in quadrant III of the plane, with witnesses on the line $x + y = -1$ and vertices above the line. It is easy to check that the result is a witness rectangle graph isomorphic to $G_1 \cup G_2$.   □

4   What Are Staircase Graphs, Really?

The above discussion is unsatisfactory in that it describes one new class of graphs in terms of another such new class. In this section, we discover that the class of graphs representable as staircase graphs is really a well known family of graphs.

Recall that an *interval graph* is one that can be realized as the intersection graph of a set of intervals on a line, i.e., its set of vertices can be put in one-to-one correspondence with a set of intervals, with two vertices being adjacent if and only if the corresponding intervals intersect. A *co-interval graph* is the complement of an interval graph, i.e., a graph representable as a collection of intervals in which adjacent vertices correspond to disjoint intervals.

Lemma 8. *Graphs representable as staircase graphs are exactly the co-interval graphs.*

Proof. Consider a vertex v in a staircase graph $\mathrm{RG}^+(V, W)$. Without loss of generality, assume the witnesses W lie on the line $\ell\colon y = x$ and the vertices lie above it. Create an artificial witness on ℓ lying above all vertices. Associate v with the smallest interval I_v of ℓ containing all witnesses lying to the right and below v as well as the witness immediately above v. It is easily checked that $v \sim v'$ in $\mathrm{RG}^+(V, W)$ if and only if I_v and $I_{v'}$ do not meet. Thus the intersection graph of the intervals $\{I_v \mid v \in V\}$ is isomorphic to the complement of $\mathrm{RG}^+(V, W)$.

Conversely, let H be a co-interval graph on n vertices and $\{I_v\}$ its realization by a set of intervals on the line $\ell\colon x = y$. Extend each interval, if necessary, slightly to ensure that the $2n$ endpoints of the intervals are distinct. Place $2n-1$ witnesses along ℓ, separating consecutive endpoints, and transform each interval $I_v = (a_v, a_v), (b_v, b_v)$ into point $p_v = (a_v, b_v)$. Let W and P be the set of witnesses and points thus generated. Now I_v misses I_w if and only if $a_v < b_v < a_w < b_w$ or $a_w < b_w < a_v < b_v$, which happens if and only if the rectangle $B(p_v, p_w)$ contains a witness. Hence $\mathrm{RG}^+(P, W)$ is isomorphic to H, as claimed.   □

Theorem 4 and Lemma 8 imply the following more satisfactory statement:

Theorem 5. *The class of graphs representable as witness rectangle graphs with two non-trivial connected components is precisely the class of graphs formed as a disjoint union of zero or more isolated vertices and two co-interval graphs.*

Corollary 2. *Whether or not a given combinatorial graph $G = (V, E)$ with two non-trivial connected components can be drawn as a WRG can be tested in time $O(|V| + |E|)$; a drawing, if it exists, can be constructed in the same time.*

Proof. Use the linear-time recognition and reconstruction algorithm for co-interval graphs from [18, 23].   □

5   How to Stab Rectangles, Thriftily

Let P be a set of n points in the plane, and let S be some given family of geometric regions, each with at least two points from P on its boundary. The problem of how many points are required to stab all the elements of S using a second set W of points has been considered several times for different families of regions [1, 2, 20, 10, 26]. For example, among the shapes previously investigated were the family of triangles with vertices in S and the family of disks whose boundary passes through two points of P.

We consider here a variant of this problem that is related to WRGs, in which we focus on the family R of all open isothetic rectangles containing two points of P on their boundary and assume that the points of P have no repeated x- or y-coordinates. We denote by $st_R(n)$ the maximum number of piercing points that are required, when all sets P of n points are considered. Stabbing all the rectangles that have p and q on their boundary is equivalent to just stabbing $B(p, q)$. Therefore we see that

$$st_R(n) = \max_{|P|=n} \min\{|W| : \mathrm{RG}^+(P, W) = K_{|P|}\}.$$

Theorem 6. *The asymptotic value of $st_R(n)$ is $2n - \Theta(\sqrt{n})$.*

Proof. We first construct a set Q of n points, no two of them with equal abscissa or ordinate, that admits a set of $2n - \Theta(\sqrt{n})$ pairwise openly-disjoint rectangles, whose boundary contains two points from Q, which will imply the lower bound.

Start with a grid of size $\sqrt{n} \times \sqrt{n}$, then rotate the whole grid infinitesimally clockwise and finally perturb the points very slightly, so that no point coordinate is repeated and there are no collinearities. The desired set of rectangles contains $B(p, q)$ for every pair of points $p, q \in Q$ that were neighbors in the original grid; refer to Figure 5 (left).

For the proof that $2n - \Theta(\sqrt{n})$ points suffice to stab all the rectangles refer to Theorem 6 in [1] (illustrated in Figure 5 (right)).   □
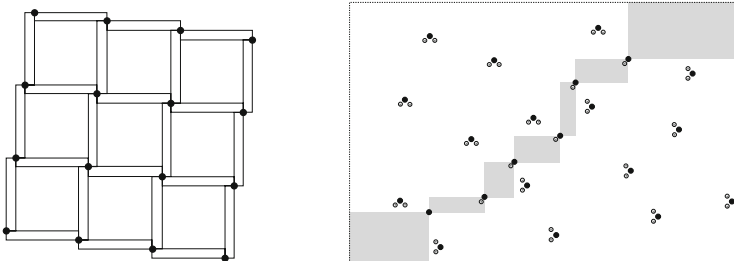
Fig. 5. Construction of a set Q with a large number of rectangles with disjoint interiors, each one with two points from Q on its boundary; every point of Q participates in four rectangles, with the exception of those on the boundary of the configuration (left). Construction, for a point set P, of a set W of positive witnesses such that $\mathrm{RG}^+(P, W) = K_{|P|}$ (right).

References

1. Aronov, B., Dulieu, M., Hurtado, F.: Witness (Delaunay) graphs. Computational Geometry Theory and Applications 44(6-7), 329–344 (2011)
2. Aronov, B., Dulieu, M., Hurtado, F.: Witness Gabriel graphs. Computational Geometry Theory and Applications (to appear)
3. Aurenhammer, F., Klein, R.: Voronoi diagrams. In: Sack, J., Urrutia, G. (eds.) Handbook of Computational Geometry, ch.5, pp. 201–290. Elsevier Science Publishing, Amsterdam (2000)
4. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing: Algorithms for the Visualization of Graphs. Prentice Hall, Englewood Cliffs (1998)
5. Di Battista, G., Lenhart, W., Liotta, G.: Proximity drawability: A survey. In: Tamassia, R., Tollis, I.G. (eds.) GD 1994. LNCS, vol. 894, pp. 328–339. Springer, Heidelberg (1995)
6. Di Battista, G., Liotta, G., Whitesides, S.: The strength of weak proximity. J. Discrete Algorithms 4(3), 384–400 (2006)
7. de Berg, M., Carlsson, S., Overmars, M.: A general approach to dominance in the plane. J. Algorithms 13(2), 274–296 (1992)
8. Chartrand, G., Lesniak, L.: Graphs and Digraphs, 4th edn. Chapman & Hall, Boca Raton (2004)
9. Collette, S.: Regions, Distances and Graphs., PhD thesis. Université Libre de Bruxelles (2006)
10. Czyzowicz, J., Kranakis, E., Urrutia, J.: Dissections, cuts, and triangulations. In: Proc. 11th Canadian Conf. Comput. Geometry, pp. 154–157 (1999)
11. Dillencourt, M.B.: Toughness and Delaunay triangulations. Discrete Comput. Geometry 5(6), 575–601 (1990)
12. Dillencourt, M.B.: Realizability of Delaunay triangulations. Inf. Proc. Letters 33(6), 283–287 (1990)
13. Dillencourt, M.B., Smith, W.D.: Graph-theoretical conditions for inscribability and Delaunay realizability. Discrete Math. 161(1-3), 63–77 (1996)
14. Dulieu, M.: Witness proximity graphs. PhD thesis. Polytechnic Institute of NYU, Brooklyn, New York (2012)
15. Dushnik, B., Miller, E.W.: Partially ordered sets. American J. Math. 63, 600–619 (1941)
16. Erdős, P., Fajtlowicz, S., Hoffman, A.J.: Maximum degree in graphs of diameter 2. Networks 10(1), 87–90 (2006)
17. Fortune, S.: Voronoi diagrams and Delaunay triangulations. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, 2nd edn., ch.23, pp. 513–528. CRC Press, Boca Raton (2004)
18. Habib, M., McConnell, R., Paul, C., Viennot, L.: Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. Theoretical Computer Science 234(1-2), 59–84 (2000)
19. Jaromczyk, J.W., Toussaint, G.T.: Relative neighborhood graphs and their relatives. Proc. IEEE 80(9), 1502–1517 (1992)
20. Katchalski, M., Meir, A.: On empty triangles determined by points in the plane. Acta Math. Hungar. 51, 23–328 (1988)
21. Liotta, G.: Proximity Drawings. In: Tamassia, R. (ed.) Handbook of Graph Drawing and Visualization. Chapman & Hall/CRC Press, (in preparation)
22. Liotta, G., Lubiw, A., Meijer, H., Whitesides, S.: The rectangle of influence drawability problem. Computational Geometry Theory and Applications 10(1), 1–22 (1998)

23. McConnell, R.: Personal communication (2011)
24. McKay, B.D., Miller, M., Širáň, J.: A note on large graphs of diameter two and given maximum degree. Combin. Theory Ser. B 74, 110–118 (1998)
25. Miller, M., Širáň, J.: Moore graphs and beyond: A survey of the degree/diameter problem. Electron. J. Combin. DS14, 61 (2005)
26. Sakai, T., Urrutia, J.: Covering the convex quadrilaterals of point sets. Graphs and Combinatorics 23(1), 343–357 (2007)
27. Toussaint, G.T. (ed.): Computational Morphology. North-Holland, Amsterdam (1988)
28. Weisstein, E.W.: Graph Join. From MathWorld–A Wolfram Web Resource, `http://mathworld.wolfram.com/GraphJoin.html`

Faster Optimal Algorithms for Segment Minimization with Small Maximal Value[⋆]

Therese Biedl[1], Stephane Durocher[2], Céline Engelbeen[3],
Samuel Fiorini[4], and Maxwell Young[1]

[1] David R. Cheriton School of Computer Science, University of Waterloo, ON, Canada
{biedl,m22young}@uwaterloo.ca
[2] Department of Computer Science, University of Manitoba, MB, Canada
durocher@cs.umanitoba.ca
[3] Département de Mathématique, Université Libre de Bruxelles, Brussels, Belgium
{celine.engelbeen,sfiorini}@ulb.ac.be

Abstract. The segment minimization problem consists of finding the smallest set of integer matrices (*segments*) that sum to a given intensity matrix, such that each summand has only one non-zero value (the *segment-value*), and the non-zeroes in each row are consecutive. This has direct applications in intensity-modulated radiation therapy, an effective form of cancer treatment.

We study here the special case when the largest value H in the intensity matrix is small. We show that for an intensity matrix with one row, this problem is fixed-parameter tractable (FPT) in H; our algorithm obtains a significant asymptotic speedup over the previous best FPT algorithm. We also show how to solve the full-matrix problem faster than all previously known algorithms. Finally, we address a closely related problem that deals with minimizing the number of segments subject to a minimum *beam-on-time*, defined as the sum of the segment-values. Here, we obtain an almost-quadratic speedup.

1 Introduction

Intensity-modulated radiation therapy (IMRT) is an effective form of cancer treatment, where radiation produced by a linear accelerator is delivered to the patient through a multileaf collimator (MLC). The MLC is mounted on an arm that can revolve freely around the patient so that he or she can be irradiated from several angles. We focus on the so-called *step-and-shoot* mode, where the radiation is delivered in a series of steps. In each step, two banks of independent metal leaves in the MLC are positioned to obstruct certain portions of the radiation field, while leaving others exposed. Neither the head of the MLC, nor its leaves move during irradiation. A treatment plan specifies the amount of radiation to be delivered along each angle.

For any given angle, the radiation field is discretized and decomposed into $m \times n$ pixels, where m is typically the number of pairs of leaves of the MLC. This determines

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 4 & 4 & 2 & 0 \\ 1 & 1 & 2 & 2 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} = 2 \times \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} + 1 \times \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$
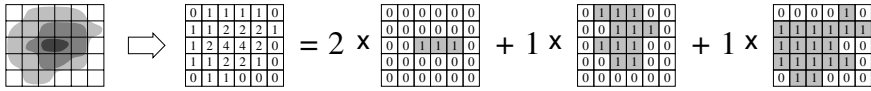
Fig. 1. An example of a segmentation of an intensity matrix where $H = 4$

a decomposition of the radiation beam into $m \times n$ *beamlets*. The amount of radiation is represented as an $m \times n$ *intensity matrix* A of non-negative integer values, whose entries represent the amount of radiation to be delivered through the corresponding pixel.

The leaves of the MLC can be seen as partially covering rows of A; for each row i of A there are two leaves, one of which may slide inwards from the left to cover the elements in columns 1 to $\ell - 1$ of that row, while the other may slide inwards from the right to cover the elements in columns $r + 1$ to n. Thus the entries of A that are not covered form an interval $[\ell, r] := \{\ell, \ell + 1, \dots, r\}$ of consecutive columns. After each step, the amount of radiation applied in that step (this can differ per step) is subtracted from each entry of A that has not been covered. The irradiation is completed when all entries of A have reached 0.

Setting leaf positions in each step requires time. Minimizing the number of steps reduces treatment time, which increases patient comfort, and can result in increased patient throughput, reduced machine wear, and overall reduced cost of the procedure. Minimizing the number of steps for a given treatment plan is the primary objective of this paper.

Formally, a *segment* is a $m \times n$ binary matrix S such that ones in each row of S are consecutive. Each segment S has an associated non-negative integer weight which we call the *segment-value*, denoted by $v_{\mathcal{S}}(S)$ or simply $v(S)$ when $\mathcal{S}$ is understood. We call a segment a t-segment if its value is t. A *segmentation* of A is a set of segments whose weighted sum equals A. So, $\mathcal{S}$ is a segmentation of A if and only if we have $A = \sum_{S \in \mathcal{S}} v(S)S$. Figure 1 illustrates the segmentation of an intensity matrix.

The *(minimum-cardinality) segmentation problem* is, given an intensity matrix A, to find a minimum cardinality segmentation of A. We also consider the special case of a matrix A with one row, which we call the *single-row segmentation problem*, in contrast with the more general *full-matrix segmentation problem* with m rows.

We also briefly examine a different, but closely related *lex-min* problem: find a minimum cardinality segmentation among those with minimum *beam-on-time*, defined as the total value $\sum_{S \in \mathcal{S}} v(S)$ of the segmentation.[1] As the segmentation problem focuses on the time incurred for establishing leaf positions, optimizing the beam-on-time also has implications for making procedures more efficient by reducing the time spent administering the treatment corresponding to the segments themselves.

Related Work: The segmentation problem is known to be NP-complete in the strong sense, even for a single row [9,2,3], as well as APX-complete [4]. Bansal *et al.* [4] provide a $24/13$-approximation algorithm for the single-row problem and give better

---

[1] The lex-min problem is also known as the *min DT-min DC* problem where DT stands for *decomposition time* (i.e., the beam-on-time) and DC stands for decomposition cardinality (i.e., the number of segments); however, we refer to this as the lex-min problem throughout.

approximations for more constrained versions. Work by Collins *et al.* [10] shows that the single-*column* version of the problem is NP-complete and provides some non-trivial lower bounds given certain constraints. Work by Luan *et al.* [16] gives two approximation algorithms for the full $m \times n$ segmentation problem, and Biedl *et al.* [6] extend this work to achieve better approximation factors.

A number of heuristics are known [3,18,11,14] as well as approaches for obtaining optimal (exact) solutions [7,1,17]. Particularly relevant to our work is that of Cambazard *et al.* [8] who show that the segmentation of a single row is fixed-parameter tractable (FPT); specifically, they give an algorithm which achieves an optimal segmentation in $O(p(H)^2 n)$ time, where H is the largest value in A and $p(H)$ is the number of partitions of H.

Kalinowski [15] studies the lex-min problem and gives polynomial time algorithms for the case when H is a constant. In the single-row case, he gives an $O(p(H)^2 n)$ time algorithm. The solution output by this first algorithm is also optimal for the minimum-cardinality segmentation problem (this follows from known results, e.g. [4]). For general $m \times n$ intensity matrices, he provides a $O(2^H \sqrt{H} \, m \, n^{2H+2})$ time algorithm. From this second algorithm, one can derive an algorithm for the full $m \times n$ minimum segmentation problem with time complexity $O(2^H H^{5/2} \, m \, n^{2H+3})$ by guessing the beam-on-time T of a minimum cardinality segmentation and appending a row to the intensity matrix to increase its minimum beam-on-time to T; it can be shown that $T \in O(H^2 n)$.

Our Contributions: We summarize our contributions below:

- For the single-row segmentation problem, we provide a faster exact algorithm. In particular, our algorithm runs in $O(p(H) H n)$ time, which is polynomial in n so long as $H \in O(\log^2 n)$. In comparison to the result of Cambazard *et al.* [8], our algorithms is faster by a factor of $\Omega(p(H)/H)$.

Significant challenges remain in solving the full-matrix problem and here we achieve two important results:

- For general H, we give an algorithm that yields an optimal solution to the full-matrix segmentation problem in $O(m \, n^H / 2^{(1-\epsilon)(H)})$ time for an arbitrarily small constant $\epsilon > 0$. In contrast, applying the variant of Kalinowski's algorithm mentioned above yields a worst-case run-time of $\Omega(m \, n^{2H+3})$. Therefore, our result improves the run-time by more than $\Omega(n^3)$.
- For $H = 2$, the full matrix problem can be solved optimally in $O(m \, n)$ time in contrast to the $O(m \, n^2)$ time implied by the previous result for general H. This result also has implications for the approximation algorithms in [6] where it can be employed as a subroutine to improve results in practice.

Finally, we address the lex-min problem:

- For general H, we give an algorithm that yields an optimal solution to the full-matrix lex-min problem in time $O(m \, n^H / 2^{(\frac{1}{2} - \varepsilon)H})$. In comparison to the previous best result by Kalinowski [15], our algorithm improves the run-time by more than $\Omega(n^2)$.

Therefore, our algorithms represent a significant asymptotic speed-up and the techniques required to achieve these improvements is non-trivial. Due to space restrictions, we omit some proofs and details; these can be found in [5].

2   Single-Row Segmentation

In this section, we give an algorithm for the single-row segmentation problem that is FPT in H, the largest value in the intensity matrix A. Since A has only one row, we represent it as a vector $A[1..n]$. Let $\Delta[j] := A[j] - A[j-1]$ for $j \in [n+1]$ (for the purpose of such definitions, we will assume that $A[0] = A[n+1] = 0$.) We say that there is a *marker* between index $j-1$ and j if $\Delta[j] \neq 0$, i.e., if the value in A changes.

Any segmentation of a row can be *standardized* as follows: (1) Every segment S *begins* (i.e., has its first non-zero entry) and *ends* (i.e., has its last non-zero entry) adjacent to a marker. For if it doesn't, then some other segment(s) must end where S begins (or vice versa), and by moving all these endpoints to the nearest marker, we retain a segmentation without adding new segments. (2) Whenever a segment ends at a marker, then no other segment of the same value begins at that marker. For otherwise the two segments could be combined into one. Note that standardization of a segmentation can only decrease the number of t-segments for all t; hence it can only improve the cardinality of the segmentation and its beam-on time.

For the single-row problem, we can improve segments even further. Call a segmentation of $A[1..n]$ *compact* if any two segments in it begin at different indices end end at different indices. Similarly as above one can show:

Lemma 1. *For any segmentation $\mathcal{S}$ of a single row, there exists a compact segmentation $\mathcal{S}'$ with $|\mathcal{S}'| \leq |\mathcal{S}|$.*

Our algorithm uses a dynamic programming approach that computes an optimal segmentation of any prefix $A[1..j]$ of A. We say that a segmentation of $A[1..j]$ is *almost-compact* if any two segments in it begin at different indices, and any two segments in it either end at different indices or both end at index j. We will only compute almost-compact segmentations; this is sufficient by Lemma 1. We compute the segmentation conditional on the values of the last segments in it.

Let $\mathcal{S}$ be a segmentation of vector $A[1..j]$; each $S \in \mathcal{S}$ is hence a vector $S[1..j]$. Define the *signature* of $\mathcal{S}$ to be the multi-set obtained by taking the value $v(S)$ of each segment ending in j. Note that the signature of a segmentation of $A[1..j]$ is a *partition* of $A[j]$, i.e., a multi-set of positive integers that sum to $A[j] \leq H$. We use operations such as $\cup, \cap$, set-difference, subset, adding/deleting elements generalized to multi-sets in the obvious way.

The key idea of our algorithm is to compute the best almost-compact segmentation of $A[1..j]$ subject to a given signature. Thus define a function f as follows:

Given an integer j and a partition ϕ of $A[j]$, let $f(j,\phi)$ be the minimum number of segments in an almost-compact segmentation $\mathcal{S}$ of $A[1..j]$ that has signature ϕ.

We will show that $f(j,\phi)$ can be computed recursively. To simplify computation we will use $f(0,\cdot)$ as a base case; we assume that $A[0] = A[n+1] = 0$. The only possible partition of 0 is the empty partition, and so $f(0,\emptyset) = 0$ is our base case.

Given a partition ϕ of $A[j]$, let $\Phi_{j-1}(\phi)$ be the set of those partitions of $A[j-1]$ that can be obtained from ϕ by deleting at most one element, and then adding at most one element. The following recursive formula for f can be shown:

Lemma 2. *For $j \geq 1$, $f(j,\phi) = \min\limits_{\psi \in \Phi_{j-1}(\phi)} \{f(j-1,\psi) + ||\phi - \psi||\}$*

Theorem 1. *The single-row segmentation problem can be solved in $O(p(H)H \cdot n)$ time and $O(n + p(H)H)$ space, where $p(H)$ is the number of partitions of H.*

Proof. The idea is to compute $f(j, \phi)$ with Lemma 2 recursively with a dynamic programming approach; the optimal value can then be found in $f(n + 1, \emptyset)$. To achieve the time complexity, we need to store the partitions in a suitable data structure. The key property here is that any partition ϕ of $A[j] \leq H$ has $O(\sqrt{H})$ distinct integers in the set $[H] := \{1, \ldots, H\}$. Thus, we can describe a partition in $O(\sqrt{H})$ space. We store partitions using a trie where each node uses $O(H)$ space but allows access to the correct child in constant time; a partition can then be located in $O(\sqrt{H})$ time.

So to compute $f(n + 1, \emptyset)$, go through $j = 1, \ldots, n$ and through all partitions ϕ of $A[j]$. For each distinct integer $t \in \phi$, compute the partition $\psi \in \Phi_{j-1}(\phi)$ obtained by deleting t and then adding one element so that ψ is a partition of $A[j-1]$. Look up ψ (and the value of $f(j-1, \psi)$ stored with it) in the trie, add $||\phi - \psi||$ to it, and update $f(j, \phi)$ if the result is smaller than what we had before. Analyzing these loops, we see that the running time is $O(n \cdot p(H) \cdot \sqrt{H} \cdot \sqrt{H})$ as desired.   □

Note that the algorithm is fixed-parameter tractable with respect to parameter H. It is known that $p(H) \leq e^{\pi \cdot \sqrt{\frac{2 \cdot H}{3}}}$ [12], so this algorithm is in fact polynomial as long as $H \in O(\log^2 n)$. In the present form, it only returns the size of the smallest segmentation, but standard dynamic programming techniques can be used to retrieve the segmentation in the same run-time with an $O(\log n)$ space overhead.

3   Full-Matrix Segmentation

In this section, we give an algorithm that computes the optimal segmentation for a full matrix, and which is polynomial as long as H is a constant.

3.1   Segmenting a Row under Constraints

The difficulty of full-matrix segmentation lies in that rows cannot be solved independently of each other, since an optimal segmentation of a full matrix does not mean that the induced segmentations of the rows are optimal. Consider for example

$$\begin{bmatrix} 1\,1\,1 \\ 2\,2\,2 \\ 3\,3\,3 \end{bmatrix} = \begin{bmatrix} 1\,1\,1 \\ 0\,0\,0 \\ 1\,1\,1 \end{bmatrix} + \begin{bmatrix} 0\,0\,0 \\ 2\,2\,2 \\ 2\,2\,2 \end{bmatrix}$$

which is optimal, but the induced segmentation for the third row is not optimal.

If $\mathcal{S}$ is a segmentation, then let $m_t(\mathcal{S})$ be the number of t-segments in $\mathcal{S}$; note that this defines a multi-set over $[H]$ which we refer to as the *multi-set $\mathcal{M}(\mathcal{S})$ defined by segmentation $\mathcal{S}$*. We now want to compute whether a row $A[1..n]$ has a segmentation $\mathcal{S}$ such that $\mathcal{M}(\mathcal{S}) \subseteq \nu$ for some given multi-set ν. We do this again with dynamic programming, by further restricting the segmentation to the first j elements and by restricting its signature. Thus define:

> *Given an integer j, a partition ϕ of $A[j]$, and a multiset ν over $[H]$, define $f'(j, \phi, \nu)$ to be 1 if there exists a segmentation $\mathcal{S}$ of $A[1..j]$ with signature ϕ and multi-set $\mathcal{M}(\mathcal{S}) \subseteq \nu$. Define $f(j, \phi, \nu)$ to be 0 otherwise.*

For example, consider $A = [1\ 3\ 2\ 4]$, $\phi = \{1,3\}$ and $\nu = \{1,1,1,2,3\}$. Then $f'(4,\phi,\nu)$ asks whether we can segment A such that at index 4 we use one 1-segment and one 3-segment, and overall we use at most three 1-segments, at most one 2-segment, and at most one 3-segment. The answer in this case is yes ($[1\ 3\ 2\ 4] = [1\ 1\ 0\ 0] + [0\ 2\ 2\ 0] + [0\ 0\ 0\ 1] + [0\ 0\ 0\ 3]$), so $f'(4,\phi,\nu) = 1$. Note that we were allowed one more 1-segment than was actually used; this is acceptable since the multi-set of the segmentation is allowed to be a subset of ν.

We claim that $f'(\cdot,\cdot,\cdot)$ has a simple recursive formula. The base case is again $j = 0$ and $f'(0,\emptyset,\nu) = 1$ for all possible multi-sets ν. For $j \geq 1$, we can compute $f'(j,\phi,\nu)$ from $f'(j-1,\cdot,\cdot)$ as follows (details are in the full paper):

Lemma 3. *For all $j \geq 1$,*

$$f'(j,\phi,\nu) = \max_{\psi \text{ is a partition of } A[j-1]} f'(j-1,\psi,\nu-(\phi-\psi)). \tag{1}$$

We will illustrate it with the above example of $A = [1\ 3\ 2\ 4]$, $\phi = \{1,3\}$ and $\nu = \{1,1,1,2,3\}$. Let $\psi = \{2\}$ and $\nu' = \{1,1,2\}$. Then $f'(3,\psi,\nu') = 1$ since $[1\ 3\ 2] = [1\ 1\ 0] + [0\ 2\ 2]$. Furthermore, we have $\phi-\psi = \{1,3\}$ and $\nu-(\phi-\psi) = \{1,1,2\} = \nu'$. Therefore, the formula says that $f'(4,\phi,\nu)$ should be 1, which indeed it is.

We now turn to the run-time of actually computing f'. In the above definition, we have not imposed any bounds on ν, other than that it is a multi-set over $[H]$. But clearly we can restrict the multi-sets considered. Assume for a moment that we know an optimal segmentation $\mathcal{S}^*$ of the full matrix. We call a multi-set ν *relevant* if $\nu \subseteq \mathcal{M}(\mathcal{S}^*)$. Clearly it suffices to compute f' for all relevant multi-sets.

To find (a superset of) relevant multi-sets without knowing $\mathcal{S}^*$, we exploit that $\mathcal{M}(\mathcal{S}^*)$ cannot contain too many segments of the same value. Recall that a marker is a place where the values within a row change; let ρ_i be the number of markers in row i, and $\rho = \max_i \rho_i$. One can show the following:

Lemma 4. *If all rows of A have at most ρ markers, then there exists a minimum cardinality segmentation that has at most $\rho/2$ segments of value t for all $t \in [H]$.*

Now let $\mathbb{M}$ be all those multi-sets over $[H]$ where all multiplicities are at most $\rho/2$; this contains all relevant multi-sets. We store these in an H-dimensional array with indices in $[0..\rho/2]$; this takes $O((\rho/2)^H)$ space, and allows lookup of a multi-set in $O(H)$ time. We can then compute the values $f'(j,\phi,\nu)$ with Algorithm 1.

The run-time of this algorithm is analyzed as follows. Computing ν' (given ν, ϕ and ψ) can certainly be done in $O(H)$ time. To look up $f'(j-1,\psi,\nu')$, we first look up ν in the array in $O(H)$ time. With each multi-set $\nu \in \mathbb{M}$, we store all partitions of $A[j-1]$ and of $A[j]$ (for the current value of j), and with each of them, the values of $f'(j-1,\psi,\nu)$ and $f'(j,\psi,\nu)$, respectively. Looking up or changing these values (given ν and ψ) can then be done in $O(\sqrt{H})$ time by storing partitions in tries.

So lines 9-11 require $O(H)$ time. They are executed $p(H)$ times from line 8, $p(H)$ times from line 6, $|\mathbb{M}|$ times from line 5, and $n+1$ times from line 4; the run-time is hence $O(n(\rho/2+1)^H p(H)^2 H)$.

As for the space requirements, we need to store all relevant multi-sets, and with each, all partitions of $A[j-1]$ and $A[j]$, which takes $O(H)$ space per partition. So the total space is $O(p(H)H(\rho/2)^H)$.

Algorithm 1

1: Let $\mathbb{M}$ be all multi-sets where all multiplicities are at most $\rho/2$.
2: **for all** multi-sets ν in $\mathbb{M}$ **do**
3:      Initialize $f'(0, \emptyset, \nu) = 1$.
4:      **for** $j = 1, \ldots, n + 1$ **do**
5:          **for all** multi-sets ν in $\mathbb{M}$ **do**
6:              **for all** partitions ϕ of $A[j]$ **do**
7:                  Initialize $f'(j, \phi, \nu) = 0$
8:                  **for all** partitions ψ of $A[j-1]$ **do**
9:                      Compute $\nu' = \nu - (\phi - \psi)$
10:                     **if** $f'(j-1, \psi, \nu') = 1$ **then**
11:                         Set $f'(j, \phi, \nu) = 1$ and **break**
12:                     **end if**
13:                 **end for**
14:             **end for**
15:         **end for**
16:     **end for**
17: **end for**

Lemma 5. *Consider one row $A[1..n]$. In $O(n(\rho/2)^H p(H)^2 H)$ time and $O(p(H)H$ $(\rho/2)^H)$ space we can compute an H-dimensional binary array $\mathcal{F}$ such that for any $m_1, \ldots, m_H \leq \rho/2$ we have $\mathcal{F}(m_1, \ldots, m_H) = 1$ if and only if there exists a segmentation of $A[1..n]$ that uses at most m_t segments of value t for $t \in [H]$.*

3.2   Full-Matrix

To solve the full-matrix problem, compute for all rows i the table $\mathcal{F}_i$ described in Lemma 5. This takes time $O(mn(\rho/2)^H p(H)^2 H)$ total. The space is $O(p(H)H$ $(\rho/2)^H)$ per row, but once done with a row i we only need to keep the $O((\rho/2)^H)$ values for the corresponding table $\mathcal{F}_i$; therefore, in total, it is $O(\max\{m, p(H)H\}(\rho/2)^H)$.

Now, in $O(m(\rho/2)^H)$ time find the numbers $m_1, \ldots, m_H$ for which $\mathcal{F}_i(m_1, \ldots, m_H)$ is 1 for *all* rows i and for which $m_1 + \cdots + m_H$ is minimized. Then by definition we can find a segmentation $\mathcal{S}_i$ for each row i that has at most m_t segments of value t for $t \in [H]$. We can combine these segmentations in the natural way (see also [6]) to obtain a segmentation $\mathcal{S}$ of A with at most m_t segments of value t for $t \in [H]$. This shows that an optimal segmentation has at most $m_1 + \cdots + m_H$ segments, and since we used the minimum possible such sum, no segmentation can be better than this bound. Since the computation for this can be accomplished by scanning all $(\rho/2)^H$ multi-sets across m rows, we have the following result:

Theorem 2. *The full-matrix segmentation problem can be solved in $O(mn(\rho/2)^H$ $p(H)^2 H)$ time and $O(\max\{m, p(H)H\}(\rho/2)^H)$ space if each row has at most ρ markers.*

Note that one could view our result as FPT in parameter is $H + \rho$. However, normally ρ will be large. In particular, if a natural pre-processing step is applied that removes from

each row of A any consecutive identical numbers (this does not affect the size of the optimum solution), then $\rho = n+1$. We therefore prefer to re-phrase our theorem to express the worst-case run-time in terms of m, n and H only. Note that $\rho \leq n + 1$ always, so the run-time becomes $O(mn^{H+1}p(H)^2 H/2^H)$. Recall that $p(H) \leq e^{\pi\sqrt{\frac{2H}{3}}} \leq e^{2.6\sqrt{H}}$ and, therefore, $Hp(H)^2 \leq He^{5.2\sqrt{H}} = 2^{\lg(H)+5.2\sqrt{H}\lg(e)} \leq 2^{8.6\sqrt{H}}$, implying that $p(H)^2 H/2^H \in O(2^{-(1-\epsilon)H})$ for arbitrarily small $\epsilon > 0$ if H is sufficiently large.

Corollary 1. *The full-matrix segmentation problem can be solved in $O(mn^{H+1} /2^{(1-\epsilon)H})$ time, where $\epsilon > 0$ is an arbitrarily small constant, and $O(mn^H)$ space.*

3.3 Further Improvements of the Complexity

We sketch a further improvement that removes a factor of n from the running time. Recall that the function $f'(j, \phi, \nu)$ was defined to be 1 if and only if there exists a segmentation $\mathcal{S}$ of $A[1..j]$ with signature ϕ and multi-set $\mathcal{M}(\mathcal{S}) \subseteq \nu$. In its place, we can instead define a function $f''(j, \phi, \nu)$, which contains the minimum number of 1-segments in a segmentation $\mathcal{S}$ of $A[1..j]$ with signature ϕ and multi-set $\mathcal{M}(\mathcal{S}) \subseteq \nu+\nu_1$. Here, ν_1 is the multi-set that has $m_1(\nu_1) = \infty$ and $m_t(\mathcal{M}_1) = 0$ for all $t \neq 1$. In other words, the segmentation that defines f'' is restricted in the number of t-segments only for $t > 1$, and the restriction on 1-segments is expressed in the return-value of f''. In particular, the value of $f''(j, \phi, \nu)$ is independent of the first multiplicity of ν, and hence must be computed only for those ν with $m_1(\nu) = 0$; there are only $(\rho/2 + 1)^{H-1}$ such multi-sets ν.

It remains to argue that f'' can be computed efficiently, with a similar formula as for f'. This is quite simple. To compute $f''(j, \phi, \nu)$, try all possible partitions ψ of $A[j-1]$, compute $\nu' = \nu - (\phi - \psi)$, and let ν'' be ν' with its first multiplicity changed to 0. Look up the value $f''(j - 1, \psi, \nu'')$ and add to it the number of 1s in $\phi - \psi$. This gives one possible candidate for a segmentation; we find the best one by minimizing over all ψ. We leave the formal proof of correctness to the reader.

We can hence compute $f''(n+1, \emptyset, \nu)$ for all $(\rho/2)^{H-1}$ multi-sets ν in $O(n(\rho/2)^{H-1} p(H)^2 H)$ time. Doing this for all rows, we can compute the maximum of the values $f''(n + 1, \emptyset, \nu)$ over all rows. The optimum segmentation can then be found by choosing the one that minimizes this maximum plus $||\nu||$ over all ν. As before, this only adds an extra $O(m)$ factor to the run-time, which is hence $O(mn(\rho/2)^{H-1}p(H)^2 H)$, and similarly as before this can be simplified to $O(mn^H/2^{(1-\varepsilon)H})$.

Theorem 3. *The full-matrix segmentation problem can be solved in $O(mn^H/2^{(1-\epsilon)H})$ time, for $\epsilon > 0$ an arbitrarily small constant, and $O(mn^{H-1})$ space.*

3.4 Solving the Lex-Min Problem

Recall that the lex-min problem is that of finding a minimum cardinality segmentation among those with minimum *beam-on-time*, defined as the total value $\sum_{S\in\mathcal{S}} v(S)$ of the segmentation. Here, we show how to apply our techniques to achieve a speed up in solving this problem. To this end, we need the notion of the *complexity of row $A[i]$* which is defined as:

$$c(A[i]) := \frac{1}{2} \sum_{j=1}^{n+1} |\Delta[i][j]| = \sum_{j=1}^{n+1} \max\{0, \Delta[i][j]\} = \sum_{j=1}^{n+1} -\min\{0, \Delta[i][j]\},$$

where as before $\Delta[i][j] := A[i][j] - A[i][j-1]$ for $j \in [n+1]$.

Importantly, is was shown in [14] that the minimum beam-on time can be computed efficiently; it is $c(A) := \max_i \{c(A[i])\}$. To solve the lex-min problem, we simply have to change our focus regarding the set $\mathbb{M}$ of interesting multi-sets. Instead of the relevant multi-sets as used earlier, we need all multi-sets ν such that $\sum_{t=1}^{H} t \cdot m_t(\nu)$ equals the minimum beam-on time. Let $\mathbb{M}_{lex}$ be the set of these multi-sets and their subsets. While Lemma 4 no longer applies, we still obtain a useful bound on the size $\mathbb{M}_{lex}$, whose proof is in the full paper.

Lemma 6. *If all rows of A have at most ρ markers, then there exists a minimum cardinality segmentation among all those that have minimum beam-on time that has at most $\rho - 1$ t-segments for all $t \in [H]$. Moreover, for $t > H/2$, there are at most $\rho/2$ t-segments.*

We can hence find and store a (super-set of) $\mathbb{M}_{lex}$ by using all entries in an H-dimensional array $[0, \rho]^{\lfloor H/2 \rfloor} \times [0, \rho/2]^{\lceil H/2 \rceil}$, and there are $O(\rho^H / 2^{H/2})$ such multi-sets. We will compute $f''(n+1, \emptyset, \nu)$ for all such multi-sets ν, and then pick a multi-set ν for which $||\nu|| + \sum_{t=1}^{H} m_t(\nu)$ is minimized, and for which $\sum_{t=1}^{H} tm_t(\nu)$ equals $c(A)$. This is then the multi-set used for a minimum segmentation among those with minimum beam-on time; we can find the actual segmentation by re-tracing the computation of $f''(n+1, \emptyset, \nu)$.

By the same analysis used for the minimum cardinality segmentation problem, and the improvement described in the previous Section 3.3, we have:

Theorem 4. *The lex-min problem can be solved in $O(mn^H / 2^{(\frac{1}{2} - \varepsilon)})$ time and with $O(mn^{H-1})$ space.*

Recall that Kalinowski's algorithm in [15] has a time complexity of $O(2^H \sqrt{H} \cdot m \cdot n^{2H+2})$. So we obtain a significant improvement in the time complexity. Finally, we note that it is intuitively reasonable that our algorithm can be applied to the lex-min problem since the restriction on the space of feasible solutions that the beam-on time be minimized can be captured by modifying appropriately the set of interesting multi-sets $\mathbb{M}_{lex}$.

4   The Special Case of $H = 2$

For $H = 2$ (i.e., a 0/1/2-matrix), the algorithm of Section 3.3 has run-time $O(mn^2)$. As we show in this section, however, yet another factor of n can be shaved off by analyzing the structure of the rows more carefully. In a nutshell, the function f'' of Section 3.3 can be computed from the structure of the row alone, without needing to go through all possible signatures; we explain this now. Throughout Section 4, we assume that all entries in the intensity matrix are 0, 1, or 2.
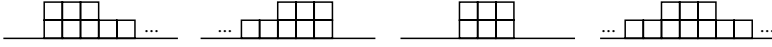
Fig. 2. Two kinds of simple steps, a tower, and a double-step

4.1 Single Row for $H = 2$

As before, let $A[1..n]$ be a single row of the matrix. Consider a maximal interval $[j', j'']$ such that $A[j'..j'']$ has all its entries equal to 2. We call $A[j'..j'']$ a *tower* if $A[j' - 1]$ and $A[j'' + 1]$ both equal 0, a *simple step* if one of $A[j' - 1]$ and $A[j'' + 1]$ equals 1 and the other 0, and a *double step* otherwise. (As usual we assume that $A[0] = A[n + 1] = 0$.) We use t, s and u to denote the number of towers, simple steps and double steps, respectively. Figure 2 illustrates how interpreting $A[i] = t$ as t blocks atop each other gives rise to these descriptive names.

Recall that $c(A[i]) = \sum_{j=1}^{n+1} \max\{\Delta[i][j], 0\}$ is the complexity of a row i of a full matrix A; we use $c(A)$ for the complexity of the single row A under consideration.

Lemma 7. *Define $g(d)$ as follows:*

$$g(d) := \begin{cases} c(A) - 2d & \text{if } d < t, \\ c(A) - t - d & \text{if } t \le d \le s + t, \\ c(A) - 2t - s & \text{if } t + s < d. \end{cases}$$

Then for any $d \ge 0$, $f''(n+1, \emptyset, \{0, d\}) = g(d)$. In other words, any segmentation $\mathcal{S}$ of A with at most d segments of value 2 has at least $g(d)$ segments of value 1. Moreover, there exists a segmentation that has at most d segments of value 2 and at most $g(d)$ segments of value 1.

Proof. Let $\mathcal{S}$ be a segmentation of A that uses at most d segments of value 2. As before, we assume that $\mathcal{S}$ has been standardized, which can be done without increasing the number of 2-segments. Therefore, any tower, step or double-step of A is either entirely covered by a 2-segment, or it does not intersect any 2-segment.

Let s_2, t_2 and u_2 be the number of steps, towers, and double-steps that are entirely covered by a 2-segment. We claim the the number of 1-segments of $\mathcal{S}$ is $c(A) - s_2 - 2t_2$, and can prove this by induction on $s_2 + t_2 + u_2$. If $s_2 + t_2 + u_2 = 0$, then $\mathcal{S}$ has only 1-segments, and since $\mathcal{S}$ is standardized, the number of 1-segments equals $c(A)$. If, say, $t_2 > 0$, then let A' be the vector obtained from A by removing a tower that is covered by a 2-segment (i.e., by replacing the 2s of that tower by 0s), and let $\mathcal{S}'$ be the segmentation of A' obtained from $\mathcal{S}$ by removing the 2-segment that covers that tower. Then A' has $t'_2 = t_2 - 1$ towers covered by 2-segments, and furthermore $c(A') = c(A) - 2$. Since $\mathcal{S}$ and $\mathcal{S}'$ have the same number of 1-segments, the claim easily follows by induction. Similarly one proves the claim by induction if $s_2 > 0$ or $u_2 > 0$.

Therefore the number of 1-segments in $\mathcal{S}$ is $c(A) - s_2 - 2t_2$. We also know that $s_2 + t_2 + u_2 \le d$. So to get a lower bound on the number of 1-segments, we should minimize $c(A) - s_2 - 2t_2$, subject to $s_2 + t_2 + u_2 \le d$ and the obvious $0 \le s_2 \le s$, $0 \le t_2 \le t$ and $0 \le u_2 \le u$. The bound now easily follows by distinguishing whether $d < t$ (the minimum is at $t_2 = d$, $s_2 = u_2 = 0$), or $t \le d < t + s$ (minimum at $t_2 = t$, $s_2 = d - t$, $u_2 = 0$) or $t + s < d$ (minimum at $t_2 = t$, $s_2 = s$, $u_2 = 0$.)

For the second claim, we obtain such a segmentation by using $\min\{d, t\}$ 2-segments for towers, then $\min\{d - t, s\}$ 2-segments for stairs if $d \geq t$, and cover everything else by 1-segments. □

The crucial idea for $H = 2$ is that since $g(\cdot)$ can be described explicitly with only three linear equations that can easily be computed, we can save space and time by not storing $f''(n+1, \emptyset, \{0, d\})$ explicitly as an array of length $\rho/2+1$, and not spending $O(n \cdot \rho/2)$ time to fill it.

4.2   Full Matrix Segmentation for $H = 2$

As in Section 3.3, to solve the full-matrix problem we need to find the value d^* that minimizes $d + \max_i\{f_i''(n + 1, \emptyset, \{0, d\})\} =: D$, where $f_i''(\cdot)$ is function $f''(\cdot) = g(\cdot)$ for row i. We can hence find the optimal segmentation of A as follows. Compute the complexity and the number of towers and stairs in each row; this takes $O(mn)$ time total. Each $f_i''(\cdot)$ is then the maximum of three lines defined by these numbers. Hence $d + \max_i\{f_i''(n + 1, \emptyset, \{0, d\})\}$ is the maximum of $3m$ lines. We hence can compute D (and with it d^*) by taking the intersection of the upper half-spaces defined by the $3m$ lines (this can be done in $O(m)$ expected time easily, and in $O(m)$ worst-case time with a complicated algorithm [13]), and then finding the grid point with the smallest y-coordinate in it.

Once we found d^*, we can easily compute a segmentation of each row that has at most $D - d^*$ segments of value 1 and at most d^* segments of value 2 (see the proof of Lemma 7) and combine them into a segmentation of the full matrix with the greedy-algorithm; this can all be done in $O(mn)$ time. Thus the overall run-time is $O(mn)$.

Theorem 5. *A minimum cardinality segmentation of an intensity matrix with values in* $\{0, 1, 2\}$ *can be found in $O(mn)$ time.*

An immediate application of this result is that it can be combined with the $O(\log h)$ approximation algorithm in [6]. While approximation guarantee remains unchanged, this should result in improved solutions in practice while not substantially increasing the running time.

One naturally asks whether this approach could be extended to higher values of H. This would be feasible if we could find (say for $H = 3$) a simpler expression for the function $f''(n + 1, \emptyset, \{0, d_2, d_3\})$, i.e. the minimum number of 1-segments given that at most d_2 2-segments and d_3-3-segments are used. It seems likely that this function would be piecewise linear (just like $g(d)$ was), but it is not clear how many pieces there are, and whether we can compute them easily from the structure of the row. Thus a faster algorithm for $H = 3$ (or higher) remains to be found.

5   Conclusion

In this work, we developed several algorithms that provide drastic running time improvements for the minimum cardinality problem. At this point, a couple interesting problems remain open. Does the full-matrix problem admit a FPT result if $m > 1$ but m is small (i.e., a small number of rows)? Is the full-matrix problem $W[1]$-hard in H?

References

1. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 1–15. Springer, Heidelberg (2007)

2. Baatar, D., Hamacher, H.W.: New LP model for multileaf collimators in radiation therapy. In: Contribution to the Conference ORP3. Universität Kaiserslautern (2003)

3. Baatar, D., Hamacher, H.W., Ehrgott, M., Woeginger, G.J.: Decomposition of integer matrices and multileaf collimator sequencing. Discrete Applied Mathematics 152(1-3), 6–34 (2005)

4. Bansal, N., Coppersmith, D., Schieber, B.: Minimizing setup and beam-on times in radiation therapy. In: Díaz, J., Jansen, K., Rolim, J.D.P., Zwick, U. (eds.) APPROX 2006 and RANDOM 2006. LNCS, vol. 4110, pp. 27–38. Springer, Heidelberg (2006)

5. Biedl, T., Durocher, S., Engelbeen, C., Fiorini, S., Young, M.: Faster optimal algorithms for segments minimization with small maximal value. Technical Report CS-2011-08. University of Waterloo (2011)

6. Biedl, T., Durocher, S., Hoos, H.H., Luan, S., Saia, J., Young, M.: A note on improving the performance of approximation algorithms for radiation therapy. Information Processing Letters 111(7), 326–333 (2011)

7. Brand, S.: The sum-of-increments constraint in the consecutive-ones matrix decomposition problem. In: Proceedings of the 24th Symposium on Applied Computing (SAC), pp. 1417–1418 (2009)

8. Cambazard, H., O'Mahony, E., O'Sullivan, B.: A shortest path-based approach to the multileaf collimator sequencing problem. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 41–55. Springer, Heidelberg (2009)

9. Chen, D.Z., Hu, X.S., Luan, S., Naqvi, S.A., Wang, C., Yu, C.X.: Generalized geometric approaches for leaf sequencing problems in radiation therapy. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 271–281. Springer, Heidelberg (2004)

10. Collins, M.J., Kempe, D., Saia, J., Young, M.: Non-negative integral subset representations of integer sets. Information Processing Letters 101(3), 129–133 (2007)

11. Cotrutz, C., Xing, L.: Segment-based dose optimization using a genetic algorithm. Physics in Medicine and Biology 48(18), 2987–2998 (2003)

12. de Azevedo Pribitkin, W.: Simple upper bounds for partition functions. The Ramanujan Journal 18(1), 113–119 (2009)

13. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry, 3rd edn. Springer, Heidelberg (2008)

14. Engel, K.: A new algorithm for optimal multileaf collimator field segmentation. Discrete Applied Mathematics 152(1-3), 35–51 (2005)

15. Kalinowski, T.: The complexity of minimizing the number of shape matrices subject to minimal beam-on time in multileaf collimator field decomposition with bounded fluence. Discrete Applied Mathematics 157(9), 2089–2104 (2009)

16. Luan, S., Saia, J., Young, M.: Approximation algorithms for minimizing segments in radiation therapy. Information Processing Letters 101(6), 239–244 (2007)

17. Wake, G.M.G.H., Boland, N., Jennings, L.S.: Mixed integer programming approaches to exact minimization of total treatment time in cancer radiotherapy using multileaf collimators. Computers and Operations Research 36(3), 795–810 (2009)

18. Xia, P., Verhey, L.J.: Multileaf collimator leaf sequencing algorithm for intensity modulated beams with multiple static segments. Medical Physics 25(8), 1424–1434 (1998)

Orthogonal Cartograms with Few Corners Per Face

Therese Biedl and Lesvia Elena Ruiz Velázquez

Cheriton School of Computer Science, University of Waterloo, Waterloo,
ON N2L 3G1, Canada
{biedl,leruizve}@uwaterloo.ca

Abstract. We give an algorithm to create orthogonal drawings of 3-connected 3-regular planar graphs such that each interior face of the graph is drawn with a prescribed area. This algorithm produces a drawing with at most 12 corners per face and 4 bends per edge, which improves the previous known result of 34 corners per face.

1 Introduction

A planar graph is a graph that can be drawn without crossing. Fáry, Stein and Wagner [11,25,28] proved independently that every planar graph has a drawing such that all edges are drawn as straight-line segments.

Sometimes additional constraints are imposed on the drawings. The most famous one is to have integer coordinates while keeping the area small; it was shown in 1990 that this is always possible in $O(n^2)$ area [8,24]. Another restriction might be to ask whether all edge lengths are integral; this exists if the graph is 3-regular [12], but is open in general. We consider drawings with prescribed face areas. Ringel [22] showed that not all planar graphs have straight-line drawings with prescribed face areas. Thomassen [26] showed that they do exist for planar graphs with maximum degree 3. We study here orthogonal drawings.

Graph drawings with prescribed face areas are motivated by cartograms, which are distortions of maps such that faces (i.e., countries in a map) should be proportional to some property of the country, such as population. Note that with a few exceptions, maps have maximum degree 3 in the interior (i.e., no four countries meet in a point.) We use the term *orthogonal cartogram* for an orthogonal drawing with prescribed areas for interior faces.

Known results. Raisz introduced rectangular cartograms [21], which are orthogonal cartograms where every face (including the outer-face) is a rectangle. Kant and He [15] and Ungar [27] characterized (independently) exactly which planar graphs have a rectangular drawing, but not all these graphs have a rectangular drawing that respects given face areas. Eppstein et al. [10] gave an algorithm to test whether a graph has a rectangular cartogram for *all* possible area assignments, but their algorithm is not polynomial.

So to create a cartogram of an arbitrary graph, one must either allow error on the area, or on the adjacencies, or allow more complicated shapes. In 2004, Van

Kreveld and Speckmann gave an algorithm to draw rectangular cartograms, where there can be a small error both on the adjacencies and on the areas [17]. In 2005, de Berg, Mumford, and Speckmann proved that any planar graph with maximum degree 3 and 3-connected dual can be drawn orthogonally with prescribed face areas and at most 60 corners per face [4]. In their journal version they reduced this to 40 corners [6], and Kawaguchi and Nagamochi reduced it further to 34 corners per face [16]. In a different work, Rahman, Miura and Nishizeki, gave an algorithm to draw a special class of graphs called good slicing graphs with at most 8 corners per face [20]. See also [5,7,13,18] for other works on rectangular and orthogonal cartograms.

Our results.   In this paper, we give a new algorithm to create orthogonal cartograms of 3-connected 3-regular planar graphs. The resulting drawings have at most 12 corners per face. The outer-face is a rectangle. Our work hence improves the work by de Berg et al. and Kawaguchi and Nagamochi [6,16]. Contrasting our work to the one by Rahman et al. [20], they use fewer corners, but our result is significantly less restrictive on the graph class. Our drawings can be found in $O(n \log n)$ time.

Our approach is substantially different from the one in [4]. They obtained drawings by modifying the graph until it has a rectangular cartogram, and then "fixing up" adjacencies. Our approach works throughout with the original graph and produces a drawing directly. We find this a more natural approach, and it also eases bounding the number of bends. Our algorithm is based on Kant's canonical ordering [14], a useful tool for many graph drawing algorithms. However, it does not use the ordering directly, but uses it for a divide & conquer approach to split the graph into two smaller graphs (or one graph and a face); this decomposition to our knowledge was not known before and may be of independent interest in the graph drawing community.

Due to space limits we omit some details; a full exposition is in [23].

2   Background and Preliminaries

Let $G = (V, E)$ be a graph with $n = |V|$ vertices. We only consider graphs that are *simple* (no loops or multiple edges) and *planar*, i.e., it can be drawn without crossing. Such a drawing determines *faces*, i.e., the maximal connected pieces of the plane. The unbounded piece is called the *outer-face*; we assume that the outer-face has been fixed. All other faces are called *interior faces*.

An *orthogonal drawing* of a planar graph is a drawing without crossing where vertices are represented by points and edges are represented by sequences of contiguous horizontal or vertical line segments. We often identify the graph-theoretic entity (e.g. vertex) with the geometric entity (e.g. point) that represents it. A place where an edge switches direction is called a *bend*.

For orthogonal drawings, it is important to keep the number of bends small, so that the drawings can be understood easily. For cartogram purposes, it is also important to keep the shape of the face as simple as possible. Hence we will also

bound the number of *corners* in a face, which is the number of angles other than 180°. Every bend is a corner, but corners may also occur at vertices.

Let A be a function that assigns non-negative weights to interior faces of G. We say that a planar drawing of G *respects the given face areas* if every interior face f of G is drawn with area $A(f)$. We use $A(G)$ to denote the sum of $A(f)$ over all interior faces f of a graph G, and $A_{\min}$ to denote the minimum area-requirement of any face of G.

A graph is *3-connected* if it is necessary to remove at least 3 vertices to make it disconnected. A planar 3-connected graph is known to have a *canonical ordering* [14]. This is a partition $V = V_1 \cup \ldots \cup V_K$ such that V_1 contains the endpoints v_1, v_2 of an edge on the outer-face, as well as exactly all vertices on the interior face adjacent to (v_1, v_2). For $1 < k < K$, set V_k can be (a) a singleton $\{z\}$ that has at least two neighbours in $V_1 \cup \ldots \cup V_{k-1}$ and at least one neighbour in $V_{k+1} \cup \ldots \cup V_K$, or (b) a *chain*, i.e., path $\{z_1, \ldots, z_r\}$ of $r \geq 2$ vertices such that z_1 and z_r have exactly one neighbour in $V_1 \cup \ldots \cup V_{k-1}$, no other vertex of the chain has a neighbour in $V_1 \cup \ldots \cup V_{k-1}$, and all vertices of the chain have a neighbour in $V_{k+1} \cup \ldots \cup V_K$. Finally, V_K is a singleton $\{v_n\}$ that belongs to the outer-face and is adjacent to v_1. See Figure 1 for an example, and many graph drawing papers and books (e.g. [14,2,3,19,9]) for more details on the canonical ordering and its applications.

We assume that the input graph is undirected, but during the algorithm we impose directions onto some of the edges, resulting in a *partially oriented graph*. In such a partially oriented graph, the *in-degree* (*out-degree*) of a vertex is the number of its incoming (outgoing) edges.

3   Dart-Shaped Graphs

Assume that $G = (V, E)$ is a 3-connected 3-regular planar graph with a canonical ordering $V = V_1 \cup \ldots \cup V_K$. This naturally implies a partial edge orientation of G which we call a *canonical orientation*: direct the edges from a vertex in V_i to a vertex in V_j if $i < j$. Edges between vertices in the same set remain undirected. This partial orientation is *acyclic*, i.e., it contains no directed cycle.

In a partially oriented graph we use the term *vertical path* for a simple path where all edges are directed from one end to the other. In a canonical orientation a vertical path connects vertices in $V_{i_1}, V_{i_2}, ..., V_{i_k}$ with $i_1 < i_2 < \cdots < i_k$; hence the name. A *horizontal path* is a simple path where all edges are undirected. In a canonical orientation, all vertices in a horizontal path belong to the same set V_i; hence the name.

Definition 1. *Let* $G = (V, E)$ *be a plane graph with an acyclic partial edge orientation.* G *is called* dart-shaped *with respect to the orientation if:*

D1.   *The outer-face consists of a horizontal path P_b (connecting a vertex c_l to a vertex c_r), and two vertical paths, P_l (from c_l to a vertex c_t), and P_r (from c_r to c_t). These paths are interior vertex-disjoint. The vertices c_l, c_r, c_t will be called* left, right *and* top *corner-vertex, and the paths P_b, P_l, P_r will be called* bottom, left *and* right path *of G.*
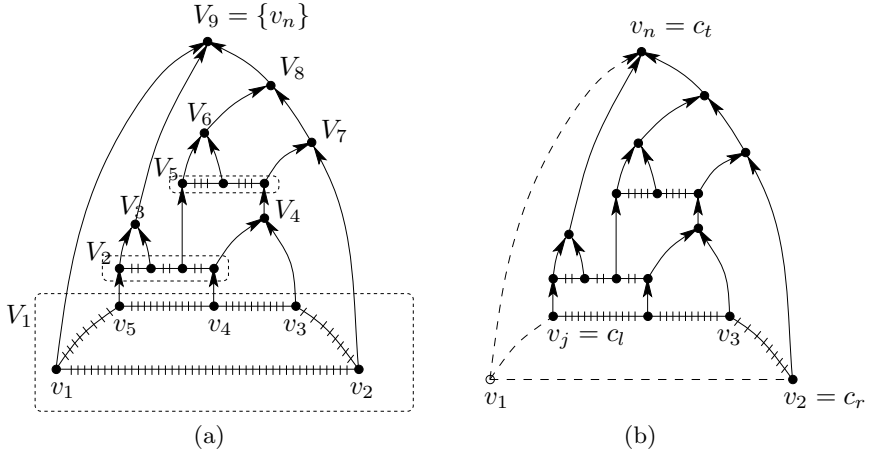
Fig. 1. (a) Canonical orientation of G. (b) $G - v_1$ is dart-shaped. Undirected edges are hatched.

D2.   Walking around any face, we encounter an undirected edge, a non-empty vertical path in the same direction as the walk, a horizontal path (possibly empty), and a non-empty vertical path in the opposite direction to the walk.

D3. $deg(c_l) = deg(c_r) = deg(c_t) = 2$. All other vertices have degree at most 3.

D4.   Every vertex $\neq c_t$ has exactly one outgoing edge.

See also Figure 1b. From now on, whenever we speak of a dart-shaped graph G, we use c_l, c_r, c_t, P_b, P_l, P_r for its corner-vertices and paths without specifically recalling that notation. Notice that D3. and D4. imply that no vertex on P_b has an incoming edge.

Using the properties of a canonical ordering, it is easy to show:

Lemma 1. Let G be a planar 3-connected 3-regular graph. Let $V_1 \cup \ldots \cup V_K$ be a canonical ordering of G, with $V_1 = \{v_1, \cdots, v_j\}$ (in order around the face) and $V_K = v_n$. Then $G' = G - v_1$, with the partial orientation induced by the canonical orientation of G, is dart-shaped and its corner-vertices are $c_l = v_j$, $c_r = v_2$ and $c_t = v_n$.

We now show how to split a dart-shaped graph into smaller dart-shaped graphs, which will allow us later to build a recursive algorithm to create orthogonal cartograms. The partial edge orientation remains the same throughout all splits, and hence will not always be mentioned.

Theorem 1. Let G be a dart-shaped graph. If G has more than one interior face, then:

(a) G can be decomposed into two dart-shaped graphs by splitting along a vertical path from the interior of P_b to the interior of P_l or P_r (see Figure 2a), or
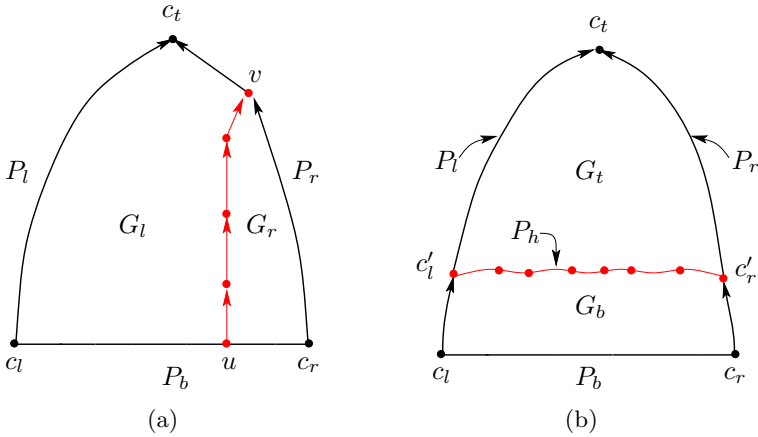
Fig. 2. (a) Vertical decomposition of G into two dart-shaped graphs. (b) Horizontal decomposition of G into a dart shaped graph and a face.

(b) *G can be decomposed into a dart-shaped graph and a face containing c_l and c_r by splitting along a horizontal path from the interior of P_l to the interior of P_r (see Figure 2b).*

Proof. The proof has two cases, leading to two different splits.

Case (1): P_b has at least two edges. Pick any vertex $u \neq c_l, c_r$ in the interior of P_b. Follow the vertical path starting from the outgoing edge of u. This path is not a cycle because G is acyclic, and it is unique since every vertex $\neq c_t$ has out-degree one. Also, the path must reach some vertex $\neq c_l, c_r, c_t$ on P_l or P_r since vertices on P_b have in-degree 0 and c_t has in-degree 2 and its incident edges are one on P_l and the other on P_r. Let v be the first vertex on $P_l \cup P_r$ that is on this path, and let P_v be the path from u to v. P_v divides G into two subgraphs, and it is not hard to verify that they are dart-shaped.

Case (2): P_b is an edge (c_l, c_r). By condition (D1), (c_l, c_r) is undirected. Consider the interior face of G adjacent to (c_l, c_r). By (D2), it consists of the edge (c_l, c_r), a vertical path $P_1 \subseteq P_l$ (since every vertex has only one outgoing edge) starting at c_l and ending at some vertex c_l', a vertical path $P_2 \subseteq P_r$ starting at c_r and ending at some vertex c_r', and (maybe) a horizontal path P_h from c_l' to c_r'.

If P_h is empty, then $c_l' = c_r'$, which (since P_l and P_r are interior vertex-disjoint) implies $c_l' = c_r' = c_t$ and all of G is one interior face, a contradiction. So P_h is non-empty, and it splits G into the face containing (c_l, c_r) and the subgraph G_t containing c_t. It is not hard to see that G_t is dart-shaped as desired. □

4   Algorithm for Orthogonal Cartograms

We first provide an outline of the steps of our algorithm to create orthogonal cartograms. (1) Compute the canonical order of G and the partial edge

orientation. (2) Let $G' = G - v_1$; G' is dart-shaped. (3) Split G' into two dart-shaped graphs G_1 and G_2. (4) Draw G_1 and G_2 separately (recursively) within a prescribed shape. (5) Combine the drawings of G_1 and G_2 into a single drawing. (6) Re-insert v_1 suitably.

The difficulty of this algorithm lies in combining the drawings of G_1 and G_2. These two graphs share vertices, so if we draw G_1 first, this forces some of the vertices to be at fixed locations in the drawing of G_2. Since our algorithm works recursively, we must allow to fix the positions of some vertices on the outer-face of the graph. But then it is not possible to split the drawing region into simple regions, such as rectangles.

Therefore, we use a significantly more complex shape, which we call a *T-staircase*, defined formally in Section 4.1. In order to draw subgraphs in it recursively, we need to break it apart into smaller T-staircases of fixed area; we discuss this in Section 4.2. In Section 4.3 we describe exactly where on a T-staircase the vertices of G' can be fixed, so that G' can be drawn with correct face areas inside any T-staircase. We call this a *correct pinning*. Then we combine everything together and explain the choice of T-staircase in the outermost recursion in Lemma 4.

4.1   T-Staircases

Definition 2. *A* T-staircase *is an x-monotone orthogonal polygon for which the upper chain consists of just one edge (the* top side*) and the lower chain consists of a descending staircase (the* left curve*), one horizontal edge (the* base*) and an ascending staircase (the* right curve*). Furthermore, all corners of the polygon except the two bottommost ones are within distance ε from the top, where $\varepsilon > 0$ will be specified below.*

See Figure 3a for an example. The *top ε-region* is the topmost region inside the T-staircase with height ε; by definition it contains all corners on the left and right curves. Thus, the segments of these curves outside the top ε-region are straight vertical lines. The *left* and *right stairs* are the portion of the left and right curves inside the top ε-region and the *left* and *right sides* are the segments of the left and right curves outside the top ε-region.

The *left/right ε-region* is an $\varepsilon \times h$ rectangle adjacent to the left/right side, and inside the T-staircase. We presume that ε has been chosen so small that the top, left and right ε-regions together have area less than $A_{\min}$ (the minimum area required for any face.) We use the ε-regions to place all necessary bends. Since ε-regions have less area than any face, it is possible to include any portion of the ε-regions in any face and still be able to draw it with correct area.

The *allowed segment* of the top side is the segment starting at the x-coordinate of the right side of the left ε-region and ending at the x-coordinate of the left side of the right ε-region. We will later see that vertices on the top side of a T-staircase will only be assigned to points along the allowed segment. For any T-staircase T, denote its area by $A(T)$.

4.2   Decomposing T-Staircases

The idea now is to break any dart-shaped graph G into two pieces (as in Theorem 1), and place the pieces in a T-staircase T recursively. To do so, we first must argue that T can be divided into two T-staircases suitably, even if the endpoints of the dividing line are restricted in their location. Let a *pinning point* be a point that is either on the interior of a vertical segment of the left or right stairs, or on a reflex corner of the left or right stairs.

We now give two lemmas for how to divide a T-staircase; they mirror (and will be applied to) the two different cases of how to divide a dart-shaped graph.

Lemma 2. *Let T be a T-staircase and let $A_{\min} \leq A \leq A(T) - A_{\min}$. Then we can divide T with an orthogonal path l into two T-staircases of area A and $A(T) - A$. Moreover, we can restrict l as follows:*

- *l begins at a (pre-specified) pinning point v, or at an (un-specified) point on the allowed segment of T.*
- *l ends at an (un-specified) point on the base of T.*
- *l has at most one bend.*

Proof. Assume l is restricted to begin at a pinning point v, which by symmetry we assume to be on the left stairs. Draw a horizontal line from v to some x-coordinate X (to be determined later) that is above the base; this is feasible since v is at a pinning point. Continue vertically to the base from there. See Figure 3b. If l is instead restricted to begin at the allowed segment, then simply make it a vertical line segment at x-coordinate X (to be determined.)

To see that a suitable value X exists, use the mean value theorem. If X is the largest x-coordinate of the left ε-region, then the area to the left of l is less than $A_{\min}$, so this X was too small. If X is the smallest x-coordinate of the right ε-region, then the area to the right of l is less than $A_{\min}$, so this X was too big. By the mean value theorem, hence a suitable X exists.

One can easily verify that the two resulting shapes are T-staircases. $\square$

Lemma 3. *Let T be a T-staircase and let $A_{\min} \leq A \leq A(T) - A_{\min}$. Then we can divide T with an orthogonal path l into a T-staircase T' of area A containing the allowed segment and a polygon B. Moreover, we can restrict l as follows:*

- *l begins at a pinning point on the left stairs, or at the left endpoint of the allowed segment.*
- *l ends at a pinning point on the right stairs, or at the right endpoint of the allowed segment.*
- *l has at most 4 bends.*

Proof. Assume first that the endpoints of l are required to be the pinning points u and v on the left/right stairs. We usually cannot draw a straight line from u to v because their positions are fixed, and the line between them may not be horizontal and/or may not create an area of appropriate size. Thus, we connect u
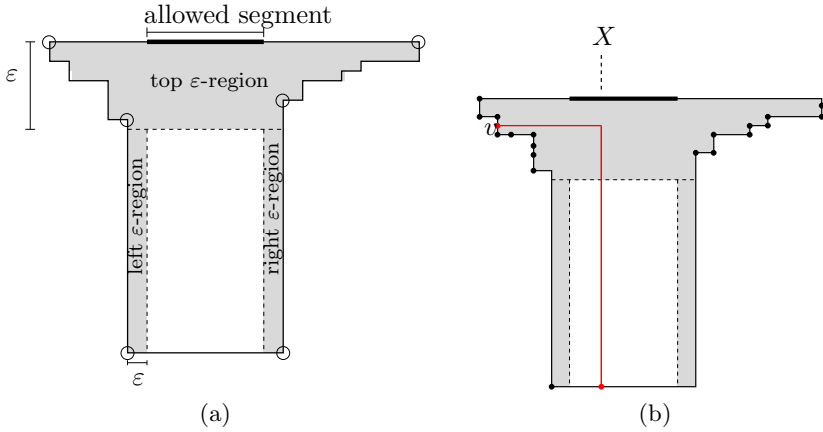
Fig. 3. (a) Example of a T-staircase. (b) Division of T into two T-staircases. Figures are not to scale. (the height of the ε-regions is hugely exaggerated.)
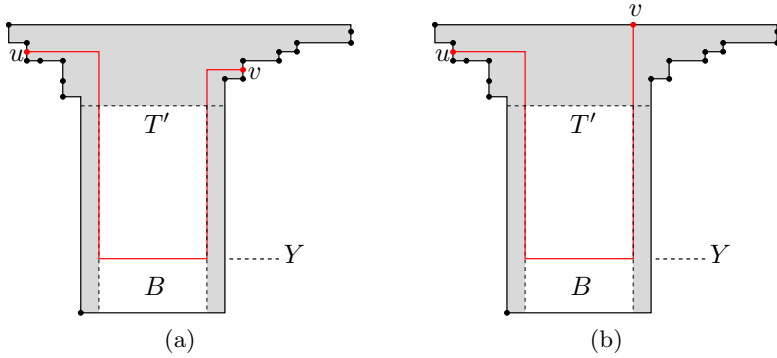
Fig. 4. Example of B and T' when (a) both ends of l are at pinning points, and (b) exactly one end of l is at a pinning point

to v with a path l with four bends, as follows: Go right from u till the boundary of the left ε-region, then go down to some y-coordinate Y (to be determined), then go right till the boundary of the right ε-region, then go up to the y-coordinate of v, then go right to v. The other cases of restrictions on endpoints of l are done similarly, except that we omit the first and/or last segment. See e.g. Figure 4.

In any of the cases above, we can see that the area T' above l is a T-staircase since its corners (except the bottommost ones) are in the top ε-region. It remains to argue that a suitable Y exists. If $Y = 0$ then the area below l would have area less than $A_{\min}$, so $Y = 0$ is too small. If Y is the smallest y-coordinate of the top ε-region, then the area above l would have area less than $A_{\min}$, so this Y is too big. So by the mean value theorem a suitable value for Y exists.   □

4.3   Pinning Dart-Shaped Graphs to T-Staircases

Now we define a condition that guarantees that a dart-shaped graph with prescribed interior face areas can be drawn inside a T-staircase:

Definition 3. *Let G be a dart-shaped graph and T be a T-staircase. A partial assignment of outer-face vertices of G to points on the boundary of T is called a* correct pinning *of G to T if the following constraints are satisfied:*

C1.   *The points assigned to vertices correspond to the order of vertices along the outer-face of G.*

C2.   *Any vertex $v \in P_l \cup P_r$ with $deg(v) = 3$ is either assigned to a pinning point on the left/right stairs, or could be assigned to a point on the allowed segment such that the order of vertices is respected.*

C3.   *Any corner of T has a vertex assigned to it, with at most 6 exceptions: the endpoints of the top side, the endpoints of the base, and the bottommost corner of the left and right stairs. These points are circled in Figure 3a.*

C4.   *If there is a bend at the bottommost corner of the left or right stairs, then there are no vertices assigned to a pinning point on the segment below it.*

Lemma 4. *Let G be a dart-shaped graph that is pinned correctly to a T-staircase T of area $A(G)$. Then G has an orthogonal drawing inside T that respects the pinned vertices and the given face areas.*

Proof. If G has only a single interior face, then the correct pinning ensures that fixed vertices are drawn in order, and unpinned vertices can be added suitably. So assume that G has at least two interior faces. By Theorem 1 G can be split into smaller graphs, either with a vertical path or a horizontal path. We now distinguish two cases, depending on the type of split.

Case (1): Assume G can be divided into two dart-shaped graphs G_l and G_r by a vertical path from a vertex $u \neq c_l, c_r$ in P_b to a vertex $v \neq c_l, c_r, c_t$ in $P_l \cup P_r$. Assume that $v \in P_l$; the other case is similar. Since G is pinned correctly to T and $deg(v) = 3$, vertex v is either assigned to a pinning point on the left stairs or could be placed on the allowed segment. By Lemma 2, we can divide T into two T-staircases T_l and T_r of area $A(G_l)$ and $A(G_r)$ such that the dividing line l begins at the pinning point of v (or at the allowed segment where we then place v), and ends at the base (where we then place u.)

The interior vertices of P_v are pinned to l as follows:

- If v was on the allowed segment, then l is a vertical segment. Pin all interior vertices of P_v in order on l inside the top ε-region. See Figure 5a.
- If v was at a pinning point (hence l has a bend b), then let w be the topmost (closest to v) vertex of degree 3 on P_v for which the neighbour that is not on P_v belongs to G_r. If w does not exist, then all vertices in P_v remain unpinned (they will be placed on the allowed segment for G_l, and have degree 2 in G_r.) If w does exist, then pin it to b, and pin all vertices between w and u on P_l on the vertical segment of l, below w, in order and within the top ε-region. The other points of P_l remain unpinned and will be on the horizontal segment adjacent to w (which is the allowed segment for T_l.) See Figure 5b.

Case (2): Assume G can be divided into a dart-shaped graph G_t and a single face G_b by a horizontal path P_h from a vertex $c'_l \neq c_l, c_t \in P_l$ to a vertex $c'_r \neq c_r, c_t \in P_r$. Since G is pinned correctly to T, c'_l and c'_r are assigned to a pinning point or could be placed on the allowed segment. By Lemma 3 it is possible to draw an orthogonal path l from c'_l to c'_r that divides T into a T-staircase T' of area $A(G_t)$ and a polygon B of area $A(G_b)$. We will not assign any vertices $\neq c'_l, c'_r$ to positions in T'.

The verification that in both cases we get a correct pinning is tedious but straightforward, and left to the reader. So in either case, the subgraphs that have more than one interior face have been pinned correctly to a T-staircase of appropriate area. Hence by induction the subgraphs can be drawn while respecting the pinned vertices, and putting the drawings together gives a drawing of G.   □

Theorem 2. *Any 3-regular 3-connected planar graph $G = (V, E)$ can be drawn orthogonally with given interior face areas, at most 4 bends per edge, and at most 12 corners per face.*

Proof. Compute a canonical order with $V_1 = \{v_1, \ldots, v_j\}$ and $V_K = \{v_n\}$ and orient edges accordingly. Then $G' = G - v_1$ is dart-shaped with corners $c_l = v_j$, $c_r = v_2$ and $c_t = v_n$ (Lemma 1).

Let T be any rectangle with area $A(G')$. Clearly T is a T-staircase. Pin v_j to the bottom left corner, v_2 to the bottom right corner, and v_n to the left top corner of T. Pin all vertices on the vertical path from v_j to v_n to the left side of T, in order, and inside the ε-region. Then G' is pinned correctly to T. By Lemma 4, we can recursively draw G' inside T respecting the face areas.

Any edge is part of a dividing path created during the algorithm, and hence has at most 4 bends. To show that each interior face of G' has at most 12 corners, we prove that it has at most 4 reflex corners. Observe that such corners are necessarily at bends, since vertices $\neq v_1, v_2, v_n$ have degree 3, and v_1, v_2, v_n do not form reflex corners for interior faces.

Recall that any T-staircase has at most 2 bends that are at reflex corners if its graph is pinned correctly to it. Hence any face reached in the base case of the
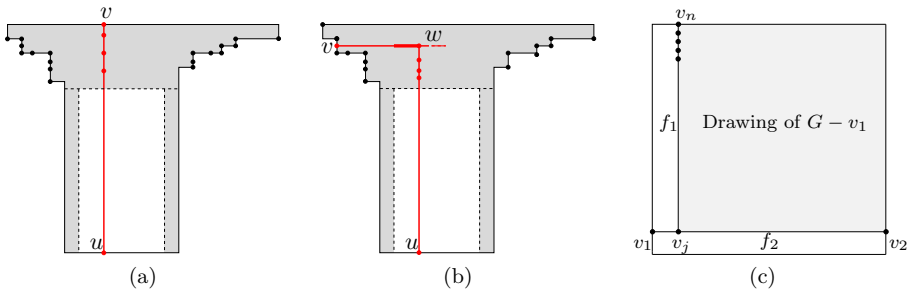
Fig. 5. (a) Pinning vertices when dividing the T-staircase vertically. (b) Pinning vertices when dividing the T-staircase with one bend, and vertex w exists. (c) The base case: Pinning v_2 and the vertices on P_l, and how to add v_1.

recursion has at most 2 reflex corners. The only kind of face not reached in the base case is the polygon B below the dividing path l that we create in Lemma 3. This path adds at most 2 reflex corners to the polygon below, so such a face has at most 4 reflex corners. Either way, each face has at most $k \leq 4$ reflex corners. Since it has $k + 4$ convex corners, the total number of corners is at most 12.

Thus G' is now drawn inside the rectangle with at most 4 bends per edge and 12 corners per face. To draw G, we need to add the vertex v_1 and its three incident edges, and it is easy to do so while respecting areas and with 2 bends in those edges; see Figure 5c.   □

5   Remarks

In this paper, we studied orthogonal cartograms, and showed that 12 corners are sufficient to create an orthogonal cartogram of any 3-connected 3-regular graph. Ours was a theoretical paper, with focus on minimizing the number of corners. We did not consider other aspects of "good" cartograms such as being similar to an input map, or avoiding the "thin connectors" that happen in the top ε-region, and this remains a topic for future study.

Following our construction, it is easy to see that if all areas are rational, then all coordinates are also rational if we choose the rectangle for the outer-most recursion, and the "free" coordinate for pinning vertices to be rational. However, it remains open how small the rational coordinates are, i.e., what grid-size would be needed if we scaled them to be integers. We do not expect a polynomial bound here, but do the coordinates satisfy, say, $O(n!2^n)$ if $A \equiv 1$? Or can we at least bound the minimum edge-segment length or minimum feature size?

Finally, what is the "correct" bound for the corners of faces? A lower bound of 8 is known [29]. A very recent result shows that 10 bends are possible [1]. Is 8 or 10 the correct bound?

References

1. Alam, J., Biedl, T., Felsner, S., Kaufmann, M., Kobourov, S.G.: Proportional contact representations of planar graphs with rectilinear polygons (in preparation)
2. Chrobak, M., Kant, G.: Convex grid drawings of 3-connected planar graphs. Int. J. Comput. Geometry Appl. 7(3), 211–223 (1997)
3. Chrobak, M., Nakano, S.: Minimum-width grid drawings of plane graphs. In: GD 1994. LNCS, vol. 894, pp. 104–110. Springer, Heidelberg (1994)
4. de Berg, M., Mumford, E., Speckmann, B.: On rectilinear duals for vertex-weighted plane graphs. In: Healy, P., Nikolov, N.S. (eds.) GD 2005. LNCS, vol. 3843, pp. 61–72. Springer, Heidelberg (2006)
5. de Berg, M., Mumford, E., Speckmann, B.: Optimal BSPs and rectilinear cartograms. In: ACM Intl. Symp. on Advances in Geographic Information Systems (GIS 2006), pp. 19–26. ACM, New York (2006)
6. de Berg, M., Mumford, E., Speckmann, B.: On rectilinear duals for vertex-weighted plane graphs. Discrete Mathematics 309(7), 1794–1812 (2009)
7. de Berg, M., Mumford, E., Speckmann, B.: Optimal BSPs and rectilinear cartograms. Int. J. Comput. Geometry Appl. 20(2), 203–222 (2010)

8. de Fraysseix, H., Pach, J., Pollack, R.: How to draw a planar graph on a grid. Combinatorica 10, 41–51 (1990)
9. Dujmović, V., Eppstein, D., Suderman, M., Wood, D.: Drawings of planar graphs with few slopes and segments. In: Comput. Geom. Theory Appl., vol. 38, pp. 194–212. Springer, Heidelberg (2005)
10. Eppstein, D., Mumford, E., Speckmann, B., Verbeek, K.: Area-universal rectangular layouts. In: ACM Symposium on Computational Geometry (SoCG 2009), pp. 267–276 (2009)
11. Fáry, I.: On straight line representation of planar graphs. Acta. Sci. Math. Szeged 11, 229–233 (1948)
12. Geelen, J., Guo, A., McKinnon, D.: Straight line embeddings of cubic planar graphs with integer edge lengths. Journal of Graph Theory 58(3), 270–274 (2008)
13. Heilmann, R., Keim, D.A., Panse, C., Sips, M.: Recmap: Rectangular map approximations. In: InfoVis 2004, pp. 33–40. IEEE, Los Alamitos (2004)
14. Kant, G.: Drawing planar graphs using the canonical ordering. Algorithmica 16(1), 4–32 (1996)
15. Kant, G., He, X.: Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. Theor. Comput. Sci. 172(1-2), 175–193 (1997)
16. Kawaguchi, A., Nagamochi, H.: Orthogonal drawings for plane graphs with specified face areas. In: Cai, J.-Y., Cooper, S.B., Zhu, H. (eds.) TAMC 2007. LNCS, vol. 4484, pp. 584–594. Springer, Heidelberg (2007)
17. Kreveld, M. van, Speckmann, B.: On rectangular cartograms. Comput. Geom. Theory Appl. 37(3), 175–187 (2007)
18. Meulemans, W., van Renssen, A., Speckmann, B.: Area-preserving subdivision schematization. In: Fabrikant, S.I., Reichenbacher, T., Kreveld, M. van, Schlieder, C. (eds.) GIScience 2010. LNCS, vol. 6292, pp. 160–174. Springer, Heidelberg (2010)
19. Nishizeki, T., Rahman, M.S.: Planar Graph Drawing. Lecture Notes Series on Computing. World Scientific Publishing Company, Singapore (2004)
20. Rahman, M.S., Miura, K., Nishizeki, T.: Octagonal drawings of plane graphs with prescribed face areas. Comput. Geom. Theory Appl. 42(3), 214–230 (2009)
21. Raisz, E.: The rectangular statistical cartogram. Geographical Review 24(3), 292–296 (1934)
22. Ringel, G.: Equiareal graphs. Contemporary Methods in Graph Theory. BI Wissenschaftsverlag, 503–505 (1990)
23. Ruiz Velázquez, L.E.: Drawing planar graphs with prescribed face areas. Master's thesis. University of Waterloo (2010), `http://uwspace.uwaterloo.ca/bitstream/10012/5481/1/RuizVelazquezLesviaElena.pdf`
24. Schnyder, W.: Embedding planar graphs on the grid. In: SODA 1990: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 138–148. Society for Industrial and Applied Mathematics, Philadelphia (1990)
25. Stein, S.: Convex maps. American Mathematical Society 2, 464–466 (1951)
26. Thomassen, C.: Plane cubic graphs with prescribed face areas. Combinatorics, Probability & Computing 1, 371–381 (1992)
27. Ungar, P.: On Diagrams Representing Maps. Journal of the London Mathematical Society s1-28(3), 336–342 (1953)
28. Wagner, K.: Bemerkungen zum Vierfarbenproblem. Jahresbericht der Deutschen Mathematiker-Vereinigung 46, 26–32 (1936)
29. Yeap, K.H., Sarrafzadeh, M.: Floor-planning by graph dualization: 2-concave rectilinear modules. SIAM Journal on Computing 22, 500–526 (1993)

Smoothed Analysis of Partitioning Algorithms for Euclidean Functionals

Markus Bläser[1], Bodo Manthey[2], and B.V. Raghavendra Rao[1]

[1] Saarland University, Department of Computer Science
{mblaeser,bvrr}@cs.uni-saarland.de
[2] University of Twente, Department of Applied Mathematics
b.manthey@utwente.nl

Abstract. Euclidean optimization problems such as TSP and minimum-length matching admit fast partitioning algorithms that compute near-optimal solutions on typical instances.

We develop a general framework for the application of smoothed analysis to partitioning algorithms for Euclidean optimization problems. Our framework can be used to analyze both the running-time and the approximation ratio of such algorithms. We apply our framework to obtain smoothed analyses of Dyer and Frieze's partitioning algorithm for Euclidean matching, Karp's partitioning scheme for the TSP, a heuristic for Steiner trees, and a heuristic for degree-bounded minimum-length spanning trees.

1 Introduction

Euclidean optimization problems are a natural class of combinatorial optimization problems. In a Euclidean optimization problem, we are given a set X of points in $\mathbb{R}^2$. The topology used is the complete graph of all points, where the Euclidean distance $\|x - y\|$ is the length of the edge connecting the two points $x, y \in X$.

Many such problems, like the Euclidean traveling salesman problem [18] or the Euclidean Steiner tree problem [11], are NP-hard. For others, like minimum-length perfect matching, there exist polynomial-time algorithms. But still these polynomial-time algorithms are sometimes too slow to solve large instances. Thus, fast heuristics to find near-optimal solutions for Euclidean optimization problems are needed.

A generic approach to design heuristics for Euclidean optimization problems are partitioning algorithms: They divide the Euclidean plane into a number of cells such that each cell contains only a small number of points. This allows us to quickly find an optimum solution for our optimization problem for the points within each cell. Finally, the solutions of all cells are joined in order to obtain a solution to the whole set of points. This joining should be done quickly to obtain a fast algorithm.

Although this is a rather simple ad-hoc approach, it works surprisingly well and fast in practice [13, 20]. This is at stark contrast to the worst-case performance of partitioning algorithms: They can both be very slow and output

solutions that are far from the optimal solutions. Thus, as is often the case, worst-case analysis is far too pessimistic to explain the performance of partitioning algorithms. The reason for this is that worst-case analysis is dominated by artificially constructed pathological instances that often do not resemble practical instances.

Both to explain the performance of partitioning algorithms and to gain probabilistic insights into the structure and value of optimal solutions of Euclidean optimization problems, the average-case performance of partitioning algorithms has been studied a lot. In particular, Steele [27] proved complete convergence of Karp's partitioning algorithm [15] for Euclidean TSP. Also strong central limit theorems for a wide range of optimization problems are known. We refer to Steele [28] and Yukich [31] for comprehensive surveys.

However, also average-case analysis has its drawback: Random instances usually have very specific properties with overwhelming probability. This is often exploited in average-case analysis: One shows that the algorithm at hand performs very well if the input has some of these properties. But this does not mean that typical instances share these properties. Thus, although a good average-case performance can be an indicator that an algorithm performs well, it often fails to explain the performance convincingly.

In order to explain the performance of partitioning schemes for Euclidean optimization problems, we provide a smoothed analysis. Smoothed analysis has been introduced by Spielman and Teng [23] in order to explain the performance of the simplex method for linear programming. It is a hybrid of worst-case and average-case analysis: An adversary specifies an instance, and this instance is then slightly randomly perturbed. The perturbation can, for instance, model noise from measurement. Since its invention in 2001, smoothed analysis has been applied in a variety of contexts [4, 22, 9, 3]. We refer to Spielman and Teng [24] for a survey.

We develop a general framework for smoothed analysis of partitioning algorithms for optimization problems in the Euclidean plane (Section 3). We consider the most general model where the adversary specifies n density functions $f_1, \ldots, f_n : [0,1]^2 \to [0, \phi]$. The parameter ϕ controls the adversary's power: The larger ϕ, the more powerful the adversary. (See Section 2.2 for a formal explanation of the model.) We analyze the expected running-time and approximation performance of a generic partitioning algorithm under this model. The smoothed analysis of the running-time for partitioning algorithms depends crucially on the convexity of the worst-case bound of the running-time of the problem under consideration. The main tool for the analysis of the expected approximation ratio is Rhee's isoperimetric inequality [21].

We apply the general framework to obtain smoothed analyses of partitioning algorithms for Euclidean matching, Karp's partitioning scheme for the TSP, Steiner trees, and degree-bounded minimum spanning trees in the Euclidean plane. Table 1 shows an overview. To summarize, for $\phi \leq \log^{O(1)} n$, Dyer and Frieze's partitioning algorithm [7] has an almost linear running-time, namely $O(n \log^{O(1)} n)$. For $\phi \in o(\log^2 n)$, its expected approximation ratio tends to 1

Table 1. Smoothed bounds for some Euclidean optimization problems

problem	running-time	approximation ratio	reference
matching [7]	$O(n\phi^2 \log^4 n)$	$1 + O(\sqrt{\phi}/\log n)$	Corollary 4.2
TSP [15]	poly(n)	$1 + O(\sqrt{\phi/\log n})$	Corollary 5.1
Steiner tree [14]	poly(n)	$1 + O(\sqrt{\phi/\log n})$	Corollary 6.1
degree-bounded MST	poly(n)	$1 + O(\sqrt{\phi \log\log n/\log n})$	Corollary 7.1

as n increases. The approximation ratios of the partitioning algorithms for TSP and Steiner trees tend to 1 for $\phi \in o(\log n)$. For degree-bounded spanning trees, this is the case for $\phi \in o(\log n/\log\log n)$. Our general framework is applicable to many other partitioning algorithms as well.

Due to space limitations, many proofs are omitted and will appear in the full version of the paper.

2 Preliminaries

2.1 Euclidean Functionals

A *Euclidean functional* is a function $\mathsf{F} : ([0,1]^2)^\star \to \mathbb{R}$ that maps a finite point set $X \subseteq [0,1]^2$ to a real number $\mathsf{F}(X)$. The following are examples of Euclidean functionals:

- MM maps a point set to the length of its minimum-length perfect matching (length means Euclidean distance, one point is left out if the cardinality of the point set is odd).
- TSP maps a point set to the length of its shortest Hamiltonian cycle, i.e., to the length of its optimum traveling salesman tour.
- MST maps a point sets to the length of its minimum-length spanning-tree.
- ST maps a point set to the length of its shortest Steiner tree.
- dbMST maps a point set to the length of its minimum-length spanning tree, restricted to trees of maximum degree at most b for some given bound b.

The Euclidean functionals that we consider in this paper are all associated with an underlying combinatorial optimization problem. Thus, the function value $\mathsf{F}(X)$ is associated with an optimum solution (minimum-length perfect matching, optimal TSP tour, ...) to the underlying combinatorial optimization problem. In this sense, we can design approximation algorithms for F: Compute a (near-optimal) solution (where it depends on the functional what a solution actually is; for instance, a perfect matching), and compare the objective value (for instance, the sum of the lengths of its edges) to the function value.

We follow the notation of Frieze and Yukich [31,10]. A Euclidean functional F is called *smooth* [31,21] if there is a constant C such that $\left|\mathsf{F}(X \cup Y) - \mathsf{F}(X)\right| \leq C\sqrt{|Y|}$ for all finite $X, Y \subseteq [0,1]^2$.

Let $C_1, \ldots, C_s$ be a partition of $[0,1]^2$ into rectangles. We call each C_ℓ a *cell*. Note that the cells are not necessarily of the same size. For a finite set $X \subseteq [0,1]^2$ of n points, let $X_\ell = X \cap C_\ell$ be the points of X in cell C_ℓ. Let $n_\ell = |X_\ell|$ be the number of points of X in cell C_ℓ. Let $\mathrm{diameter}(C_\ell)$ be the diameter of cell C_ℓ.

We call F *sub-additive* if

$$\mathsf{F}(X) \leq \sum_{\ell=1}^{s} \big(\mathsf{F}(X_\ell) + \mathrm{diameter}(C_\ell) \big).$$

F is called *super-additive* if

$$\mathsf{F}(X) \geq \sum_{\ell=1}^{s} \mathsf{F}(X_\ell).$$

The Euclidean functions TSP, MM and MST are smooth and sub-additive [31, 27, 28].

A combination of sub-additivity and super-additivity for a Euclidean functional F is a sufficient (but not a necessary) condition for the existence of a partitioning heuristic for approximating F. We will present such a generic partitioning heuristic in Section 3. Following Frieze and Yukich [10], we define a slightly weaker additivity condition that is sufficient for the performance analysis of partitioning algorithms.

Frieze and Yukich [10] call a Euclidean function F *near-additive* if, for all partitions $C_1, \ldots, C_s$ of $[0,1]^2$ into cells and for all finite $X \subseteq [0,1]^2$, we have

$$\big| \mathsf{F}(X) - \textstyle\sum_{\ell=1}^{s} \mathsf{F}(X_\ell) \big| \leq O\big(\textstyle\sum_{\ell=1}^{s} \mathrm{diameter}(C_\ell) \big). \tag{1}$$

It is not hard to see that, if F is sub-additive and super-additive, then F is also near-additive. Euclidean functionals such as TSP, MM, and MST are sub-additive but not super-additive. However, these functionals can be approximated by their corresponding canonical *boundary functionals*, which are super-additive [10, 31]. Yukich [31] has shown that this is a sufficient condition for a Euclidean functional to be near-additive.

Proposition 2.1 (Yukich [31, Lemma 5.7]). *Let F be a sub-additive Euclidean functional. Let F_{B} be a super-additive functional that well-approximates F. (This means that $|\mathsf{F}(X) - \mathsf{F}_{\mathrm{B}}(X)| = O(1)$ for all finite $X \subseteq [0,1]^2$.) Then F is near-additive.*

The functionals MM, TSP, MST, ST, and dbMST are near-additive.

Limit theorems are a useful tool for the analysis of Euclidean functionals. Rhee [21] proved the following limit theorem for smooth Euclidean functionals over $[0,1]^2$. We will mainly use it to bound the probability that F assumes a too small function value.

Theorem 2.2 (Rhee [21]). *Let X be a set of n points drawn independently according to identical distributions from $[0,1]^2$. Let F be a smooth Euclidean functional. Then there exist constants C and C' such that for all $t > 0$, we have*

$$\mathbb{P}\big[\big| \mathsf{F}(X) - \mathbb{E}\big[\mathsf{F}(X)\big] \big| > t \big] \leq C \cdot \exp\big(-C' t^4 / n \big).$$

Remark 2.3. *Rhee proved Theorem 2.2 for the case that $x_1, \ldots, x_n$ are identically distributed. However, as pointed out by Rhee herself [21], the proof carries over to the case when $x_1, \ldots, x_n$ are drawn independently but their distributions are not necessarily identical.*

2.2   Smoothed Analysis

In the classical model of smoothed analysis [23], an adversary specifies a point set $\bar{X}$, and then this point set is perturbed by independent identically distributed random variables in order to obtain the input set X. A different view-point is that the adversary specifies the means of the probability distributions according to which the point set is drawn. This model has been generalized as follows [4]: Instead of only specifying the mean, the adversary can specify a density function for each point, and then we draw the points independently according to their density functions. In order to limit the power of the adversary, we have an upper bound ϕ for the densities: The adversary is allowed to specify any density function $[0, 1]^2 \to [0, \phi]$. If $\phi = 1$, then this boils down to the uniform distribution on the unit square $[0, 1]^2$. If ϕ gets larger, the adversary becomes more powerful and can specify the location of the points more and more precisely. The role of ϕ is the same as the role of $1/\sigma$ in classical smoothed analysis, where σ is the standard deviation of the perturbation. We summarize this model formally in the following assumption.

Assumption 2.4. *Let $\phi \geq 1$. An adversary specifies n probability density functions $f_1, \ldots, f_n : [0, 1]^2 \to [0, \phi]$. We write $f = (f_1, \ldots, f_n)$ for short. Let $x_1, \ldots, x_n \in [0, 1]^2$ be n random variables where x_i is drawn according to f_i, independently from the other points. Let $X = \{x_1, \ldots, x_n\}$.*

If the actual density functions f matter and are not clear from the context, we write $X \sim f$ to denote that X is drawn as described above. If we have a performance measure P for an algorithm (P will be either running-time or approximation ratio in this paper), then the smoothed performance is $\max_f \big(\mathbb{E}_{X \sim f}[P(X)]\big)$. Note that the smoothed performance is a function of the number n of points and the parameter ϕ.

Let F be a Euclidean functional. For the rest of this paper, let $\mu_{\mathsf{F}}(n, \phi)$ be a lower bound for the expected value of F if X is drawn according to the probabilistic model described above. More precisely, μ_{F} is some function that fulfills $\mu_{\mathsf{F}}(n, \phi) \leq \min_f \big(\mathbb{E}_{X \sim f}[\mathsf{F}(X)]\big)$. The function μ_{F} comes into play when we have to bound F (or: the objective value of an optimum solution) from below in order to analyze the approximation ratio.

3   Framework

In this section, we present our framework for the performance analysis of partitioning heuristics for Euclidean functionals. Let $\mathsf{A}_{\mathrm{opt}}$ be an optimal algorithm for some smooth and near-additive Euclidean functional F, and let $\mathsf{A}_{\mathrm{join}}$ be an

algorithm that combines solutions for each cell into a global solution. We assume that $\mathsf{A}_{\mathrm{join}}$ runs in time linear in the number of cells. Then we obtain the following algorithm, which we call A.

Algorithm 3.1 (generic algorithm A). *Input: set $X \subseteq [0,1]^2$ of n points.*

1. *Divide $[0,1]^2$ into s cells $C_1, \ldots, C_s$.*
2. *Compute optimal solutions for each cell using $\mathsf{A}_{\mathrm{opt}}$.*
3. *Join the s partial solutions to a solution for X using $\mathsf{A}_{\mathrm{join}}$.*

We use the following assumptions in our analysis and mention explicitly whenever they are used.

Assumption 3.2.   *1. $\phi \in O(s)$. This basically implies that the adversary cannot concentrate all points in a too small number of cells.*
2. *$\phi \in \omega(s \log n / n)$. This provides a lower bound for the probability mass in a "full" cell, where full is defined in Section 3.1.*
3. *$\phi \in o(\sqrt{n / \log n})$. With this assumption, the tail bound of Theorem 2.2 becomes sub-polynomial.*

These assumptions are not too restrictive: For the partitioning algorithms we analyze here, we have $s = O(n / \log^{O(1)} n)$. Ignoring poly-logarithmic terms, the first and third assumption translate roughly to $\phi = O(n)$ and $\phi = o(\sqrt{n})$, respectively. But $\phi = \Theta(\sqrt{n})$ suffices for the adversary to specify an individual small square for each point, thus we can expect to approach almost worst-case behavior for $\phi = \Omega(\sqrt{n})$. The second assumption roughly says $\phi = \omega(1)$. But for $\phi = O(1)$, we can expect (almost) average-case behavior.

3.1   Smoothed Running-Time

Many of the schemes that we analyze choose the partition in such a way that we have a worst-case upper bound on the number of points in each cell. Other algorithms, like the one for matching, have a fixed partition independent of the input points. In the latter case, the running-time also depends on ϕ.

Let $T(n)$ denote the worst-case running-time of $\mathsf{A}_{\mathrm{opt}}$ on n points. Then the running-time of A is bounded by $\sum_{\ell=1}^{s} T(n_\ell) + O(s)$. The expected running-time of A is thus

$$\sum_{\ell=1}^{s} \mathbb{E}[T(n_\ell)] + O(s). \tag{2}$$

For the following argument, we assume that T (the running-time of $\mathsf{A}_{\mathrm{opt}}$) is a monotonically increasing, convex function, and that the locations of the cells are fixed and all their volumes are equal. (The assumption about the cells is not fulfilled for all partitioning heuristics. For instance, Karp's partitioning scheme [15] chooses the cells not in advance but based on the actual point set. However, in Karp's scheme, the cells are chosen in such a way that there is a good worst-case upper bound for the number of points per cell, so there is no need for a smoothed analysis.) By abusing notation a bit, let $f_i(C_\ell) = \int_{C_\ell} f_i(x) \, \mathrm{d}x$ be the cumulative density of f_i in the cell C_ℓ. Since f_i is bounded from above by ϕ, we

have $f_i(C_\ell) \leq \phi/s$. Let $f(C_\ell) = \sum_{i=1}^{n} f_i(C_\ell)$. Note that $f_i(C_\ell) = \mathbb{P}(x_i \in C_\ell)$ and $f(C_\ell) = \mathbb{E}[n_\ell]$.

We call a cell C_ℓ *full* with respect to f if $f(C_\ell) = n\phi/s$. We call C_ℓ *empty* if $f(C_\ell) = 0$. Our bound (2) on the running-time depends only on the values $f_1(C_\ell), \ldots, f_n(C_\ell)$, but not on where exactly within the cells the probability mass is assumed.

Our goal is now to show that the adversary, in order to make our algorithm as slow as possible, will make as many cells as possible full. Note that there are at most $\lfloor s/\phi \rfloor$ full cells. Assume that we have $\lfloor s/\phi \rfloor$ full cells and at most one cell that is neither empty nor full. Then the number of points in any full cell is a binomially distributed random variable B with parameters n and ϕ/s. By linearity of expectation, the expected running-time is bounded by $\left(\left\lfloor \frac{s}{\phi} \right\rfloor + 1 \right) \cdot \mathbb{E}[T(B)] + O(s)$. Since $\phi = O(s)$ by Assumption 3.2 (1), this is bounded by $O\left(\frac{s}{\phi} \cdot \mathbb{E}[T(B)] + s \right)$. If T is bounded by a polynomial, then this evaluates to $O\left(\frac{s}{\phi} \cdot T(n\phi/s) + s \right)$ by the following Lemma 3.3. This lemma can be viewed as "Jensen's inequality in the other direction" with $p = \phi/s$ for $\phi \in \omega(s \log n/n)$. The latter is satisfied by Assumption 3.2 (2).

Lemma 3.3 (inverse Jensen's inequality). *Let T be any convex, monotonically increasing function that is bounded by a polynomial, and let B be a binomially distributed random variable with parameters $n \in \mathbb{N}$ and $p \in [0,1]$ with $p \in \omega(\log n/n)$. Then $\mathbb{E}[T(B)] = \Theta(T(\mathbb{E}[B]))$.*

What remains to be done is to show that the adversary will indeed make as many cells as possible full. This follows essentially from the convexity of the running-time. Thus, we obtain the following theorem.

Theorem 3.4. *Assume that the running-time of $\mathsf{A}_{\mathrm{opt}}$ can be bounded from above by a convex function T that is bounded by a polynomial. Then, under Assumption 2.4 as well as Assumptions 3.2 (1) and (2), the expected running-time of A on input X is bounded by $O\left(\frac{s}{\phi} \cdot T(n\phi/s) + s \right)$.*

3.2   Smoothed Approximation Ratio

The value computed by A can be bounded from above by $\mathsf{A}(X) \leq \sum_{\ell=1}^{s} \mathsf{F}(X_\ell) + J'$, where J' is an upper bound for the cost incurred by joining the solution for the cells. Since F is near-additive, $\mathsf{A}(X) \leq \mathsf{F}(X) + J$ for $J = J' + O(\sum_{\ell=1}^{s} \mathrm{diameter}(C_\ell))$. Dividing by $\mathsf{F}(X)$ yields

$$\frac{\mathsf{A}(X)}{\mathsf{F}(X)} \leq 1 + O\left(\frac{J}{\mathsf{F}(X)} \right). \tag{3}$$

For estimating the expected approximation ratio $\mathbb{E}[\mathsf{A}(X)/\mathsf{F}(X)]$ for some algorithm A, the main challenge is that $\mathsf{F}(X)$ stands in the denominator. Thus, even if we know $\mathsf{A}(X)$ precisely, we are basically left with the problem of estimating $\mathbb{E}[1/\mathsf{F}(X)]$. Jensen's inequality yields $1/\mathbb{E}[\mathsf{F}(X)] \leq \mathbb{E}[1/\mathsf{F}(X)]$. But this does not help, as we need upper bounds for $\mathbb{E}[1/\mathsf{F}(X)]$. Unfortunately, such

upper bounds cannot be derived easily from $1/\mathbb{E}[\mathsf{F}(X)]$. The problem is that we need strong upper bounds for the probability that $\mathsf{F}(X)$ is close to 0. Theorem 2.2 is too weak for this. This problem of bounding the expected value of the inverse of the optimum objective value arises frequently in bounding expected approximation ratios [8, 9].

There are two ways to attack this problem: The first and easiest way is if A comes with a worst-case guarantee $\alpha(n)$ on its approximation ratio for instances of n points. Then we can apply Theorem 2.2 to bound $\mathsf{F}(X)$ from below. If $\mathsf{F}(X) \geq \mu_{\mathsf{F}}(n, \phi)/2$, then we can use (3) to obtain a ratio of $1 + O\left(\frac{J}{\mu_{\mathsf{F}}(n,\phi)}\right)$. Otherwise, we obtain a ratio of $\alpha(n)$. If $\alpha(n)$ is not too large compared to the tail bound obtained from Theorem 2.2, then this contributes only little to the expected approximation ratio. The following theorem formalizes this.

Theorem 3.5. *Assume that* A *has a worst-case approximation ratio of* $\alpha(n)$ *for instance consisting of n points. Then, under Assumption 2.4, the expected approximation ratio of* A *is*

$$\mathbb{E}\left[\frac{\mathsf{A}(X)}{\mathsf{F}(X)}\right] \leq 1 + O\left(\frac{J}{\mu_{\mathsf{F}}(n, \phi)}\right) + \alpha(n) \cdot \exp\left(-\frac{\mu_{\mathsf{F}}(n, \phi)^4}{Cn}\right).$$

Now we turn to the case that the worst-case approximation ratio of A cannot be bounded by some $\alpha(n)$. In order to be able to bound the expected approximation ratio, we need an upper bound on $\mathbb{E}[1/\mathsf{F}(X)]$. This upper bound is formalized in the following theorem.

Theorem 3.6. *Assume that there exists a* $\beta \leq J$ *and a function* h_n *such that* $\mathbb{P}(F(X) \leq x) \leq h_n(x)$ *for all* $x \in [0, \beta]$. *Then, under Assumption 2.4, the expected approximation ratio of* A *is*

$$\mathbb{E}\left[\frac{\mathsf{A}(X)}{\mathsf{F}(X)}\right] \leq 1 + O\left(J \cdot \left(\frac{1}{\mu_{\mathsf{F}}(n, \phi)} + \frac{\exp\left(-\frac{\mu_{\mathsf{F}}(n,\phi)^4}{Cn}\right)}{\beta} + \int_{1/\beta}^{\infty} h_n\left(\frac{1}{x}\right) \, \mathrm{d}x\right)\right).$$

4 Matching

As a first example, we apply our framework to the matching functional MM defined by the Euclidean minimum-length perfect matching problem. A partitioning algorithm, which we call DF, for approximating MM was proposed by Dyer and Frieze [7]. This algorithm divides $[0, 1]^2$ into k^2 equal-sized sub-squares, computes an optimum matching within these cells and combines the solutions using the so-called strip heuristic.

Let $\mathsf{DF}(X)$ be the cost of the matching computed by the algorithm above on input $X = \{x_1, \ldots, x_n\}$, and let $\mathsf{MM}(X)$ be the cost of a perfect matching of minimum total length. Dyer and Frieze showed that $\mathsf{DF}(X)$ converges to $\mathsf{MM}(X)$ with probability 1 if the points in X are drawn according to uniform distributions on $[0, 1]^2$ (this corresponds to Assumption 2.4 with $\phi = 1$) and n goes to infinity. We extend this to the case when X is drawn as described in Assumption 2.4.

4.1   Smoothed Running-Time

A minimum-length perfect matching can be found in time $O(n^3)$ [1]. By Theorem 3.4, we get the following corollary.

Corollary 4.1. *Under Assumption 2.4 as well as Assumption 3.2 (1) and (2), the expected running-time of* DF *on input X is at most $O\left(\frac{n^3\phi^2}{k^4} + k^2\right)$. If we plug in $k = \sqrt{n}/\log n$, we obtain an expected running-time of at most $O(n\phi^2\log^4 n)$.*

4.2   Smoothed Approximation Ratio

To estimate the approximation performance, we have to specify the function $\mu_{\mathsf{MM}}(n, \phi)$. To obtain a lower bound for $\mu_{\mathsf{MM}}(n, \phi)$, let $\mathsf{NN}(X)$ denote the total edge length of the nearest-neighbor graph for the point set $X \subseteq [0,1]^2$. This means that

$$\mathsf{NN}(X) = \sum_{x \in X} \min_{y \in X: y \neq x} \|x - y\|.$$

We have $\mathsf{MM}(X) \geq \mathsf{NN}(X)/2$. We will use $\mathbb{E}[\mathsf{NN}(X)]$ to bound $\mathbb{E}[\mathsf{MM}(X)]$. Next, one shows the tail bound $\mathbb{P}\big(\mathsf{MM}(X) \leq c\big) \leq (2\phi\pi c)^{n/2}$. This allows us to apply Theorem 3.6

Corollary 4.2. *Under Assumption 2.4 and 3.2 (3), the expected approximation ratio of* DF *is $1 + O\big(\frac{\sqrt{\phi}}{\log n}\big)$.*

Remark 4.3.   *1. There exist other partitioning schemes for Euclidean matching [2], which can be analyzed in a similar way.*
  2. Instead of a standard cubic-time algorithm, we can use Varadarajan's matching algorithm [30], which has a running-time of $O(m^{1.5}\log^5 m)$ for m points, for computing the optimal matchings within each cell. This improves the running-time bound to $O(n\sqrt{\phi}\log(n)\log^5(\phi\log n))$.

5   Karp's Partitioning Scheme for Euclidean TSP

Karp's partitioning scheme [15] is a well-known heuristic for Euclidean TSP. For a point set $X \subseteq [0,1]^2$, let $\mathsf{KP}(X)$ denote the cost of the tour through X computed by Karp's scheme. Steele [27] has proved complete convergence of $\mathsf{KP}(X)$ to $\mathsf{TSP}(X)$ with probability 1, if the points are chosen uniformly and independently. Using our framework developed in Section 3, we extend the analysis of KP to the case of non-uniform and non-identical distributions.

Since Karp's scheme chooses the cells adaptively, based on the point set X, our framework for the analysis of the running-time cannot be applied. However, the total running-time of the algorithm is $T(n) = 2^{O(n/k^2)}\operatorname{poly}(n/k^2) + O(k^2)$, which is, independent of the randomness, polynomial in n for $k^2 = n/\log n$.

The nice thing about the TSP is that every tour has a worst-case approximation guarantee of at most $\frac{n}{2} \cdot \mathsf{TSP}(X)$. Thus, we can use Theorem 3.5 with $\alpha(n) = n/2$.

Corollary 5.1. *Under Assumption 2.4 as well as Assumption 3.2 (3), the expected approximation ratio of* KP *is $\mathbb{E}\big[\frac{\mathsf{KP}(X)}{\mathsf{TSP}(X)}\big] \leq 1 + O\big(\sqrt{\phi/\log n}\big)$.*

6 Euclidean Steiner Trees

Kalpakis and Sherman [14] proposed a partitioning algorithm for the Euclidean minimum Steiner tree problem analogous to Karp's partitioning scheme for Euclidean TSP. The worst case cost of the Steiner tree computed by the algorithm, however, could be larger than optimal by a constant factor. Let $\mathsf{KS}(X)$ denote the cost of the Steiner tree computed.

The running-time of this algorithm is polynomial for the choice of $s = n/\log n$ [6]. For the same reason as for Karp's partitioning scheme, we cannot use our framework to estimate the running-time, because the choice of cells depends on the actual point set.

As for the traveling salesman problem, we have a worst-case approximation ratio of $\alpha(n) = O(n)$. The reason is that, for any two points $x, y \in X$, we have $\|x - y\| \leq \mathsf{ST}(X)$. Since Kalpakis and Sherman's partitioning algorithm outputs a tree with at most a linear number of edges, we have $\mathsf{KS}(X) \leq O(n \cdot \mathsf{ST}(X))$. This gives us a worst-case approximation ratio of $O(n)$ and yields the following corollary of Theorem 3.5.

Corollary 6.1. *Under Assumption 2.4 as well as Assumption 3.2 (3), the expected approximation ratio of* KS *is* $\mathbb{E}\big[\frac{\mathsf{KS}(X)}{\mathsf{ST}(X)}\big] \leq 1 + O\big(\sqrt{\phi/\log n}\big)$.

7 Degree-Bounded Minimum Spanning Tree

A b-degree-bounded minimum spanning tree of a given set of points in $[0,1]^2$ is a spanning tree in which the degree of every point is bounded by b. For $2 \leq b \leq 4$, this problem is NP-hard, and it is solvable in polynomial time for $b \geq 5$ [19]. Let dbMST denote the Euclidean functional that maps a point set to the length of its shortest b-degree-bounded minimum spanning tree. This is a smooth, subadditive, and near-additive Euclidean functional [25].

Naturally, near-additivity implies that Karp's partitioning scheme can be extended to the b-degree-bounded minimum spanning tree problem. Let $\mathsf{P\text{-}bMST}$ be the adaptation of Karp's partitioning algorithm to dbMST with parameter $k^2 = \frac{n \log \log n}{\log n}$. With this choice of k, $\mathsf{P\text{-}bMST}$ runs in polynomial-time as a degree-bounded minimum-length spanning tree on m nodes can be found in time $2^{O(m \log m)}$ using brute-force search.

Again, we cannot use our framework for the running-time. The running-time is guaranteed to be bounded by a polynomial. But we can use Theorem 3.5 to obtain the following result.

Corollary 7.1. *Under Assumption 2.4 as well as Assumption 3.2 (3), the expected approximation ratio is* $\mathbb{E}\big[\frac{\mathsf{P\text{-}bMST}(X)}{\mathsf{dbMST}(X)}\big] \leq 1 + O\big(\sqrt{\phi \log \log n / \log n}\big)$.

8 Concluding Remarks

We have provided a smoothed analysis of partitioning algorithms for Euclidean optimization problems. The results can be extended to distributions over $\mathbb{R}^2$ by

scaling down the instance so that the inputs lie inside $[0, 1]^2$. The analysis can also be extended to higher dimensions. However, the value of ϕ for which our results are applicable will depend on the dimension d.

Even though solutions computed by most of the partitioning algorithms achieve convergence to the corresponding optimal value with probability 1 under uniform samples, in practice they have constant approximation ratios close to 1 [13, 20]. Our results show that the expected function values computed by partitioning algorithms approach optimality not only under uniform, identical distributions, but also under non-uniform, non-identical distributions, provided that the distributions are not sharply concentrated.

One prominent open problem for which our approach does not work is the functional defined by the total edge weight of a minimum-weight triangulation in the Euclidean plane. The two main obstacles for this problem are that, first, the functional corresponding to minimum-weight triangulation is not smooth and, second, the value computed by the partitioning heuristic depends on the number of points in the convex hull of the point set [12]. Damerow and Sohler [5] provide a bound for the smoothed number of points in the convex hull, but this bound is not strong enough for this purpose.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows. Prentice-Hall, Englewood Cliffs (1993)
2. Anthes, B., Rüschendorf, L.: On the weighted Euclidean matching problem in R^d dimensions. Applicationes Mathematicae 28(2), 181–190 (2001)
3. Arthur, D., Manthey, B., Röglin, H.: k-means has polynomial smoothed complexity. In: Proc. 50th Ann. IEEE Symp. on Foundations of Computer Science (FOCS), pp. 405–414. IEEE, Los Alamitos (2009)
4. Beier, R., Vöcking, B.: Random knapsack in expected polynomial time. J. Comput. System Sci. 69(3), 306–329 (2004)
5. Damerow, V., Sohler, C.: Extreme points under random noise. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 264–274. Springer, Heidelberg (2004)
6. Dreyfus, S.E., Wagner, R.A.: The Steiner problem in graphs. Networks 1(3), 195–207 (1971)
7. Dyer, M.E., Frieze, A.M.: A partitioning algorithm for minimum weighted euclidean matching. Inform. Process. Lett. 18(2), 59–62 (1984)
8. Engels, C., Manthey, B.: Average-case approximation ratio of the 2-opt algorithm for the TSP. Oper. Res. Lett. 37(2), 83–84 (2009)
9. Englert, M., Röglin, H., Vöcking, B.: Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP. In: Proc. 18th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 1295–1304. SIAM, Philadelphia (2007)
10. Frieze, A.M., Yukich, J.E.: Probabilistic analysis of the traveling salesman problem. In: Gutin, G., Punnen, A.P. (eds.) The Traveling Salesman Problem and Its Variations, ch.7, pp. 257–308. Kluwer, Dordrecht (2002)
11. Garey, M.R., Graham, R.L., Johnson, D.S.: The complexity of computing Steiner minimal trees. SIAM J. Appl. Math. 32(4), 835–859 (1977)

12. Golin, M.J.: Limit theorems for minimum-weight triangulations, other euclidean functionals, and probabilistic recurrence relations. In: Proc. 7th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 252–260. SIAM, Philadelphia (1996)
13. Johnson, D.S., McGeoch, L.A.: Experimental analysis of heuristics for the STSP. In: Gutin, G., Punnen, A.P. (eds.) The Traveling Salesman Problem and Its Variations, ch.9, pp. 369–443. Kluwer, Dordrecht (2002)
14. Kalpakis, K., Sherman, A.T.: Probabilistic analysis of an enhanced partitioning algorithm for the Steiner tree problem in R^d. Networks 24(3), 147–159 (1994)
15. Karp, R.M.: Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. Math. Oper. Res. 2(3), 209–224 (1977)
16. León, C.A., Perron, F.: Extremal properties of sums of Bernoulli random variables. Statist. Probab. Lett. 62(4), 345–354 (2003)
17. Mitzenmacher, M., Upfal, E.: Probability and Computing. Cambridge University Press, Cambridge (2005)
18. Papadimitriou, C.H.: The Euclidean traveling salesman problem is NP-complete. Theoret. Comput. Sci. 4(3), 237–244 (1977)
19. Papadimitriou, C.H., Vazirani, U.V.: On two geometric problems related to the traveling salesman problem. J. Algorithms 5(2), 231–246 (1984)
20. Ravada, S., Sherman, A.T.: Experimental evaluation of a partitioning algorithm for the steiner tree problem in R^2 and R^3. Networks 24(8), 409–415 (1994)
21. Rhee, W.T.: A matching problem and subadditive euclidean functionals. Ann. Appl. Probab. 3(3), 794–801 (1993)
22. Röglin, H., Teng, S.-H.: Smoothed analysis of multiobjective optimization. In: Proc. 50th Ann. IEEE Symp. on Foundations of Computer Science (FOCS), pp. 681–690. IEEE, Los Alamitos (2009)
23. Spielman, D.A., Teng, S.-H.: Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. J. ACM 51(3), 385–463 (2004)
24. Spielman, D.A., Teng, S.-H.: Smoothed analysis: An attempt to explain the behavior of algorithms in practice. Comm. ACM 52(10), 76–84 (2009)
25. Srivastav, A., Werth, S.: Probabilistic Analysis of the Degree Bounded Minimum Spanning Tree Problem. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 497–507. Springer, Heidelberg (2007)
26. Michael Steele, J.: Complete convergence of short paths in Karp's algorithm for the TSP. Math. Oper. Res. 6, 374–378 (1981)
27. Michael Steele, J.: Subadditive Euclidean functionals and nonlinear growth in geometric probability. Ann. Probab. 9(3), 365–376 (1981)
28. Michael Steele, J.: Probability Theory and Combinatorial Optimization. CBMS-NSF Regional Conf. Series in Appl. Math., vol. 69. SIAM, Philadelphia (1987)
29. Supowit, K.J., Reingold, E.M.: Divide and conquer heuristics for minimum weighted euclidean matching. SIAM J. Comput. 12(1), 118–143 (1983)
30. Varadarajan, K.R.: A divide-and-conquer algorithm for min-cost perfect matching in the plane. In: Proc. 39th Ann. Symp. on Foundations of Computer Science (FOCS), pp. 320–331. IEEE, Los Alamitos (1998)
31. Yukich, J.E.: Probability Theory of Classical Euclidean Optimization Problems. Lecture Notes in Mathematics, vol. 1675. Springer, Heidelberg (1998)

Feedback Vertex Set in Mixed Graphs

Paul Bonsma[1],[*] and Daniel Lokshtanov[2]

[1] Humboldt-Universität zu Berlin, Institut für Informatik, Unter den Linden 6,
10099 Berlin, Germany
bonsma@informatik.hu-berlin.de

[2] University of California, San Diego, Department of Computer Science and Engineering,
9500 Gilman Drive, La Jolla, CA 92093-0404, USA
dlokshtanov@cs.ucsd.edu

Abstract. A mixed graph is a graph with both directed and undirected edges.
We present an algorithm for deciding whether a given mixed graph on n vertices
contains a feedback vertex set (FVS) of size at most k, in time $O(47.5^k \cdot k! \cdot n^4)$.
This is the first fixed parameter tractable algorithm for FVS that applies to both
directed and undirected graphs.

1 Introduction

For many algorithmic graph problems, the variant of the problem for directed graphs
(*digraphs*) is strictly harder than the one for undirected graphs. In particular, replac-
ing each edge of an undirected graph by two arcs going in opposite directions yields
a reduction from undirected to directed graphs for most network design, routing, dom-
ination and independence problems including SHORTEST PATH, LONGEST PATH and
DOMINATING SET.

The *Feedback Vertex Set* problem is an exception to this pattern. A *feedback vertex
set (FVS)* of a (di)graph G is a vertex set $S \subseteq V(G)$ such that $G - S$ contains no
cycles. In the *Feedback Vertex Set* (FVS) problem we are given a (di)graph G and an
integer k and asked whether G has a feedback vertex set of size at most k. Indeed, if
one replaces the edges of an undirected graph G by arcs in both directions, then every
feedback vertex set of the resulting graph is a *vertex cover* of G and vice versa. Hence,
this transformation can not be used to reduce FEEDBACK VERTEX SET in undirected
graphs to the same problem in directed graphs, and other simple transformations do not
seem possible either. Thus FVS problems on undirected and directed graphs are differ-
ent problems; one is not a generalization of the other. This is reflected by the fact that
the algorithms for the two problems differ significantly across algorithmic paradigms,
be it approximation [1,2,11], exact exponential time algorithms [14,15,24] or parame-
terized algorithms [3,6,7,8]. *In this paper we bridge the gap between the parameterized
algorithms for* FEEDBACK VERTEX SET *by giving one algorithm that works for both
directed and undirected graphs*. More generally, we give the first algorithm for FVS in
mixed graphs, which are graphs that may contain both edges and arcs. Cycles in mixed
graphs are defined as expected: these may contain both edges and arcs, but all arcs
should be in the same direction (see Section 2 for precise definitions).

For a mixed graph G on n vertices and an integer k, our algorithm decides in time $2^{O(k)}k!\,O(n^4)$ whether G contains a FVS S with $|S| \leq k$, and if so, returns one. Algorithms of this type are called *Fixed Parameter Tractable (FPT) algorithms*. In general, the input for a *parameterized problem* consists of an instance X and integer parameter k. An algorithm for such a problem is an FPT algorithm if its time complexity is bounded by $f(k) \cdot O(|X|^c)$, where $|X|$ denotes the input size of X, $f(k)$ is an arbitrary computable function of k, and c is a constant independent of k. The function $f(k)$ is also called the *parameter function* of the complexity, or of the algorithm. Since the first systematic studies on FPT algorithms in the '90s (see e.g. [9]), this has become a very active area in algorithms. See [13,21] for recent introductions to the area.

FEEDBACK VERTEX SET is one of the classical graph problems and it was one of the first problems to be identified as NP-hard [19]. The problem has found applications in many areas, see e.g. [12,7] for references, with one of the main applications in *deadlock recovery* in databases and operating systems. Hence the problem has been extensively studied in algorithms [1,2,11,15,14,24,26]. The parameterized complexity of FEEDBACK VERTEX SET on undirected graphs was settled already in 1984 in a monograph by Melhorn [20]. Over the last two decades we have seen a string of improved algorithms [3,9,10,22,18,23,16,8,6] (in order of improving parameter function), and the current fastest FPT algorithm for the problem has running time $O\left(3.83^k k n^2\right)$ [5], where n denotes the number of vertices of the input graph. On the other hand, the parameterized complexity of FEEDBACK VERTEX SET on *directed* graphs was considered one of the most important open problems in Parameterized Complexity for nearly twenty years, until an FPT algorithm with running time $O\left(n^4 4^k k^3 k!\right)$ was given by Chen et al [7] in 2007. Interestingly, in [17], the permanent deadlock resolution problem as it appears in the development of distributed database systems, is reduced to feedback vertex set in mixed graphs. However, to the best of our knowledge, no algorithm for FVS on mixed graphs has previously been described.

We now give an overview of the paper. We start by giving precise definitions in Section 2. In Section 3 we give a sketch of the algorithm, and outline some the obstacles one needs to overcome in order to design an FPT algorithm for FVS in mixed graphs. Our algorithm has three main components: The frame of the algorithm is a standard iterative compression approach described in Section 3. The core of our algorithm consists of two parts: the first is a reduction from a variant of FVS to a multi-cut problem called SKEW SEPARATOR. This reduction, described in Section 4 is a non-trivial modification of the reduction employed for FVS in directed graphs by Chen et al [7]. Our reduction only works on pre-conditioned instances, we describe how to perform the necessary pre-conditioning in Section 5.

2   Preliminaries

We consider edge/arc labeled multi-graphs: formally, mixed graphs consist of a tuple $G = (V, E, A, \psi)$, where V is the vertex set, E is the edge set, and A is the arc set. The incidence function ψ maps edges $e \in E$ to sets $\{u, v\}$ with $u, v \in V$, also denoted as $uv = vu$. Arcs $a \in A$ are mapped by ψ to tuples (u, v) with $u, v \in V$. In the remainder, we will often denote mixed graphs simply by the tuple $G = (V, E, A)$, and denote $e = uv$ for edges $e \in E$ with $\psi(e) = \{u, v\}$, and $a = (u, v)$ for arcs $a \in A$ with

$\psi(a) = (u, v)$. $V(G)$, $E(G)$ and $A(G)$ denote the vertex, edge and arc set respectively of the mixed graph G.

The operation of *contracting* an edge $e = uv$ into a new vertex w consists of the following operations: introduce a new vertex w, for every edge or arc with u or v as end vertex, replace this end vertex by w, and delete u and v. Note that edge identities are preserved: $\psi(e)$ may for instance change from $\{x, u\}$ to $\{x, w\}$, but e is still considered the same edge. Note also that contractions may introduce *parallel edges or arcs* (pairs of edges or arcs e and f with $\psi(e) = \psi(f)$), and *loops* (edges e with $\psi(e) = \{w, w\}$ or arcs a with $\psi(a) = (w, w)$).

For $G = (V, E, A)$ and $S \subseteq V$ or $S \subseteq E \cup A$, by $G[S]$ we denote the subgraph induced by S. In particular, $G[E]$ is obtained by deleting all arcs and resulting isolated vertices. Deletion of S is denoted by $G - S$. The *out-degree* $d^+(v)$ (*in-degree* $d^-(v)$) of a vertex $v \in V$ is the number of arcs $e \in A$ with $\psi(e) = (v, w)$ ($\psi(e) = (w, v)$) for some w. If an arc (v, w) ((w, v)) exists, w is called an *out-neighbor* (*in-neighbor*) of v. Similarly, the *edge degree* $d(v)$ is the number of incident edges, and if $vw \in E$ then w is an *edge neighbor* of v.

A *walk of length l* in a mixed graph $G = (V, E, A)$ is a sequence $v_0, e_1, v_1, e_2, \ldots,$ e_l, v_l such that for all $1 \leq i \leq l$, $e_i \in E \cup A$ and $e_i = v_{i-1} v_i$ or $e_i = (v_{i-1}, v_i)$. This is also called a (v_0, v_l)-*walk*. v_0, v_l are its *end vertices*, $v_1, \ldots, v_{l-1}$ its *internal vertices*. A walk is a *path* if all of its vertices are distinct. A walk $v_0, e_1, v_1, \ldots, v_l$ of length at least 1 is a *cycle* if the vertices $v_0, \ldots, v_{l-1}$ are distinct, $v_0 = v_l$, and all e_i are distinct. (Note that this last condition is only relevant for walks of length 2. Note also that if e is a loop on vertex u, then u, e, u is also considered a cycle.) We will usually denote walks, paths and cycles just by their vertex sequence $v_0, \ldots, v_l$. In addition, we will sometimes encode paths and cycles by their edge/arc set $E_P = \{e_1, \ldots, e_l\}$.

3   Outline of the Algorithm

In this section we give an informal overview of our algorithm, the details are given in the following sections. Similar to many previous FVS algorithms [5,6,7,8,16], we will employ the *iterative compression* technique introduced by Reed, Smith and Vetta [25]. Essentially, this means that we start with a trivial subgraph of G and increase it one vertex at a time until G is obtained, maintaining a FVS of size at most $k + 1$ throughout the computation. Every time we add a vertex to the graph we perform a *compression* step. That is, given a graph G' with a FVS S of size $k + 1$, the algorithm has to decide whether G' has a FVS S' of size k. If the algorithm concludes that G' has no FVS of size k, we can conclude that G does not either, since G' is a subgraph of G. In each compression step the algorithm loops over all 2^{k+1} possibilities for $S \cap S'$. For each choice of $S' \cap S$ we need to solve the following problem.

S-DISJOINT FVS:
INSTANCE: A mixed graph $G = (V, E, A)$ with a FVS S.
TASK: Find a FVS S' of G with $|S'| < |S|$ and $S' \cap S = \emptyset$, or report that this does not exist.

A FVS S' with $|S'| < |S|$ and $S' \cap S = \emptyset$ is called a *small S-disjoint FVS*. The application of iterative compression implies the following lemma.

Lemma 1 (⋆). [1] *Suppose* S-DISJOINT FVS *can be solved in time* $O\left((k+1)!f(k)n^c\right)$, *with* $n = |V|$, $k = |S| - 1$ *and* $f(k)$ *non-decreasing. Then FVS can be solved in time* $O\left(k(k+1)!f(k)n^{c+1}\right)$.

Chen et al [7] gave an algorithm for S-Disjoint FVS restricted to digraphs, which we will call S-*Disjoint Directed FVS*. In Section 4 we show that their algorithm can be extended in a non-trivial way to solve the following generalization of the problem to mixed graphs. Let G be an undirected graph with $S \subseteq V(G)$. A vertex set $S' \subseteq V(G) \backslash S$ is a *multiway cut* for S (in G) if there is no (u,v)-path in $G - S'$ for any two distinct $u,v \in S$.

FEEDBACK VERTEX SET / UNDIRECTED MULTIWAY CUT (FVS/UMC):
INSTANCE: A mixed graph $G = (V, E, A)$ with a FVS S, and integer k.
TASK: Find a FVS S' of G with $|S'| \leq k$ and $S' \cap S = \emptyset$, that is also a multiway cut for S in $G[E]$, or report that this does not exist.

A multiway cut S' for $G[E], S$ is also called an *undirected multiway cut (UMC)* for G, S. The remaining question is: how can the FPT algorithm for FVS/UMC be used to solve S-Disjoint FVS? Let G, S be an S-Disjoint FVS instance. Suppose there exists a small S-Disjoint FVS S' for the graph G. If we know which *undirected* paths between S-vertices do not contain any S'-vertices, then these can be contracted, and S' remains a FVS for the resulting graph G^*. In addition, this gives a new vertex set S^* consisting of the old S-vertices and the vertices introduced by the contractions. This then yields an instance G^*, S^* of FVS/UMC, for which S' is a solution. In Section 5 we prove this more formally. However, since we do not know S', it remains to find which undirected paths between S-vertices do not contain S'-vertices. One approach would be to try all possible combinations, but the problem is that the number of such paths may not be bounded by any function of $k = |S| - 1$, see the example in Figure 1 (a). (More complex examples with many paths exist, where the solution S' is not immediately obvious.) The example in Figure 1 (a) contains many vertices of degree 2, which are simply reduced in nearly all fast undirected FVS algorithms [8,26,16,5]. However in our case we can easily add arcs to the example to prevent the use of (known) reduction rules, see e.g. Figure 1 (b). Because there may be many such paths, and there are no easy ways to reduce these, we will guess which paths do not contain S'-vertices in two stages: this way we only have to consider $2^{O(k)}$ possibilities, which is shown in Section 5.

$\circ : S$
$\odot : S'$

(a)   (b)
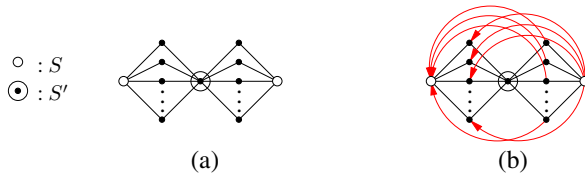
Fig. 1. Graphs with a FVS S and small S-disjoint FVS S', with many undirected S-paths

---

[1] The (full) proofs of claims marked with ⋆ have been omitted due to space restrictions.

4   An Algorithm for FVS/UMC: Reduction to Skew Separator

Let G be a digraph and $\mathcal{S} = s_1, \ldots, s_\ell$ and $\mathcal{T} = t_1, \ldots, t_\ell$ be mutually disjoint vertex sequences such that all $s_i \in V(G)$ have in-degree 0 and all $t_i \in V(G)$ have out-degree 0. A subset $C \subseteq V(G)$ disjoint from $\{s_1, \ldots, s_\ell, t_1, \ldots, t_\ell\}$ is called a *skew separator* if for all $i \geq j$, there is no (s_i, t_j)-path in $G - C$. The vertices in $\mathcal{S}$ will be called *out-terminals* and the vertices in $\mathcal{T}$ *in-terminals*. An FPT algorithm to solve the SKEW SEPARATOR problem defined below is given as a subroutine in the algorithm for DIRECTED FEEDBACK VERTEX SET by Chen et al [7].

SKEW SEPARATOR (SS):
INSTANCE: A digraph G, vertex sequences $\mathcal{S} = s_1, \ldots, s_\ell$ and $\mathcal{T} = t_1, \ldots, t_\ell$ where all $s_i \in V(G)$ have in-degree 0 and all $t_i \in V(G)$ have out-degree 0, and an integer k.
TASK: Find a skew separator C of size at most k, or report that this does not exist.

Theorem 1 (Chen et al [7]). *The Skew Separator problem on instances $G, \mathcal{S}, \mathcal{T}, k$ with $n = |V(G)|$ can be solved in time $4^k k \cdot O(n^3)$.*
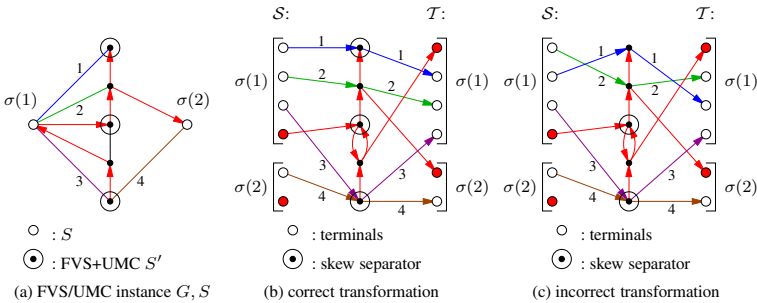
Fig. 2. Correct and incorrect transformations from FVS/UMC to Skew Separator. In- and out-terminals are ordered from top to bottom, so 'allowed paths' go from top left to bottom right.

We will use this to give an algorithm for FVS/UMC, using a non-trivial extension of the way SS is used in [7] to give an algorithm for S-Disjoint Directed FVS. We will transform a FVS/UMC instance G, S to a SS instance $G_{ss}, \mathcal{S}, \mathcal{T}$, in such a way that S' is a FVS and UMC for G, S if and only if it is a skew separator for $G_{ss}, \mathcal{S}, \mathcal{T}$. Since every cycle in G contains at least one vertex from S, this can be done by replacing every S-vertex by a set of in- and out-terminals in G_{ss}. The following proposition shows how the order of these terminals should be chosen. A bijective function $\sigma : \{1, \ldots, \ell\} \to S$ is called a *numbering* of S. It is an *arc-compatible numbering* if there are no arcs $(\sigma(i), \sigma(j))$ in G with $i > j$.

Proposition 1 ($\star$). *Let $C \subseteq V \backslash S$ be a FVS and UMC for the graph $G = (V, E, A)$ and vertex set $S \subseteq V$. Then a numbering σ of S exists such that for all $1 \leq j < i \leq |S|$, there is no path from $\sigma(i)$ to $\sigma(j)$ in $G - C$.*

Since in our case edges are present, we cannot achieve the desired correspondence by introducing just one terminal pair for every S-vertex, as was done by Chen et al [7].

Instead, for every vertex $v \in S$, we introduce a single terminal pair for all arcs incident with v in G, and in addition, for every edge incident with v we introduce a terminal pair specifically for this edge. The transformation is illustrated in Figure 2 (a) and (b). Numbers and colors for edges show how edges in G correspond to arcs in G_{ss}. For every $v \in S$, the red terminal pair is used for all incident arcs. Observe that in this example, a set S' is a FVS and UMC in G, S if and only if it is a skew separator in $G_{ss}, \mathcal{S}, \mathcal{T}$. However, this correspondence does not hold for arbitrary orderings of the edges incident with a vertex $v \in S$, as is shown by the different order used in Figure 2 (c). The indicated skew separator of size 2 does not correspond to a FVS and UMC in G, S.

Construction: We now define the transformation in detail. Let G, S, k be an instance of FVS/UMC, with $|S| = \ell$. We define the relation $\prec$ on $V(G)\backslash S$ as follows: $u \prec v$ if and only if there is a (v, u)-path in $G - S$ but no (u, v)-path (and $u \neq v$). Observe that $\prec$ is transitive and antisymmetric, and therefore a partial order on $V(G)\backslash S$.

For any numbering σ of S, the graph $G_{ss}(G, \sigma)$ is obtained from G as follows: For every $i \in \{1, \ldots, \ell\}$, we do the following: denote $v = \sigma(i)$. let $vw_1, \ldots, vw_d$ be the edges incident with v, ordered such that if $w_x \prec w_y$ then $x < y$. Since $\prec$ is a partial order, such an ordering exists and is given by an arbitrary linear extension of $\prec$. Apply the following operations: **(1)** Add the vertices $s_i^1, \ldots, s_i^{d+1}$ and $t_i^1, \ldots, t_i^{d+1}$. **(2)** For every arc (v, u) with $u \notin S$, add an arc (s_i^{d+1}, u). **(3)** For every arc (u, v) with $u \notin S$, add an arc (u, t_i^1). **(4)** For every edge vw_j, add arcs (s_i^j, w_j) and (w_j, t_i^{j+1}). **(5)** Delete v.

After this is done for every $v \in S$, replace all remaining edges xy with two arcs (x, y) and (y, x). This yields the digraph $G_{ss}(G, \sigma)$ and vertex sequences $\mathcal{S} = s_1^1, \ldots,$ $s_1^{d_1+1}, s_2^1, \ldots, s_2^{d_2+1}, \ldots \ldots, s_\ell^{d_\ell+1}$ and $\mathcal{T} = t_1^1, \ldots, t_1^{d_1+1}, t_2^1, \ldots, t_2^{d_2+1}, \ldots \ldots, t_\ell^{d_\ell+1}$, where $d_i = d(\sigma(i))$ is the edge degree of $\sigma(i)$. The integer k remains unchanged. $G_{ss}(G, \sigma), \mathcal{S}, \mathcal{T}, k$ is an instance for SS.

Lemma 2 ($\star$). *Let S be a FVS for a mixed graph $G = (V, E, A)$, such that $G[S]$ contains no edges and G contains no cycles of length at most 2. Then $C \subseteq V(G)\backslash S$ is a FVS and UMC for G and S if and only if there exists an arc-compatible numbering σ of S such that C is a skew separator for $G_{ss}(G, \sigma), \mathcal{S}, \mathcal{T}$, as constructed above.*

Proof sketch: Let C be a FVS and UMC for G, S. By Proposition 1, we can define a numbering σ of S such that for all $i > j$, there is no path from $\sigma(i)$ to $\sigma(j)$ in $G - C$. Therefore, σ is arc-compatible.

We now show that for this σ, C is a skew separator for $G_{ss}(G, \sigma), \mathcal{S}, \mathcal{T}$. Let $G_{ss} = G_{ss}(G, \sigma)$. Suppose C is not a skew separator, so $G_{ss} - C$ contains a path $P = s_i^x, v_1, \ldots,$ v_ℓ, t_j^y with $i > j$, or with $i = j$ and $x \geq y$. Then $P' = \sigma(i), v_1, \ldots, v_\ell, \sigma(j)$ is (the vertex sequence of) a walk in $G - C$; note that arcs of P may correspond to edges in P' but that the vertex sequence still constitutes a walk. If $i > j$, then all vertices of the walk P' are different and hence it is a $(\sigma(i), \sigma(j))$-path in $G - C$, contradicting the choice of σ. If $i = j$, then P' is a closed walk in $G - C$ of which all internal vertices are distinct. In all cases, it can be shown that P' is a cycle in $G - C$, which gives a contradiction. In the case where P' has length 2 it follows from $x \geq y$ and the construction of G_{ss} that distinct e and f can be chosen to ensure that $P' = \sigma(i), e, v_1, f, \sigma(i)$ is a cycle. Thus, C is a skew separator for G_{ss}.

Let C be a skew separator for $G_{ss} = G_{ss}(G, \sigma)$, for some arc-compatible numbering σ of S. We prove that C is a FVS and UMC for G, S. Suppose $G[E] - C$ contains a (u, v)-path $P = u, v_1, \ldots, v_\ell, v$ with $u, v \in S$, and no internal vertices in S. Let $u = \sigma(i)$ and $v = \sigma(j)$. Since we assumed that $G[S]$ contains no edges, P has length at least 2. Since all edges not incident with S are replaced with arcs in both directions during the construction of G_{ss}, for some x, y this yields both a path $s_i^x, v_1, \ldots, v_\ell, t_j^{y+1}$ in $G_{ss} - C$ and a path $s_j^y, v_\ell, \ldots, v_1, t_i^{x+1}$ in $G_{ss} - C$. One of these paths contradicts that C is a skew separator. This shows that C is a multiway cut for $G[E]$ and S.

Next, suppose $G - C$ contains a cycle K. Since S is a FVS for G, K contains at least one vertex of S. If K contains at least two vertices of S, then K contains a path P from $\sigma(i)$ to $\sigma(j)$ for some $i > j$, with no internal vertices in S. Let $P = \sigma(i), v_1, \ldots, v_\ell, \sigma(j)$. P has length at least two, since σ is arc-compatible, and there are no edges in $G[S]$. Then $P' = s_i^x, v_1, \ldots, v_\ell, t_j^y$ is a path in $G_{ss} - C$ for some x, y, contradicting that C is a skew separator. So now we may suppose that K contains exactly one vertex of S, w.l.o.g. $K = \sigma(i), v_1, \ldots, v_\ell, \sigma(i)$. Every cycle in G has length at least 3, so $v_1 \neq v_\ell$. Using the relation $\prec$ that was used to construct G_{ss}, it can be shown that in every case K yields a path $P = s_i^x, v_1, \ldots, v_\ell, t_i^{y+1}$ in $G_{ss} - C$ for some $x > y$, or that K consists only of edges (proofs omitted). In the latter case, $P' = s_i^y, v_\ell, v_{\ell-1}, \ldots, v_1, t_i^{x+1}$ with $y > x$ is the path in $G_{ss} - C$ that contradicts that C is a skew separator. This concludes the proof that C is a FVS and UMC for G, S. □

Lemma 2 yields a way to reduce FVS/UMC to the SS problem in the case that the input graph G does not contain any short cycles. To solve such an instance of FVS/UMC, we try all possible arc-compatible orderings σ of S (at most $\ell!$) and solve the instances of SS using Theorem 1. The FVS/UMC instance is a yes-instance if and only if at least one of the produced SS instances is. Using simple reduction rules one can reduce general instances of FVS/UMC to instances which do not contain short cycles. This reduction, together with Theorem 1 gives an FPT algorithm for FVS/UMC.

Theorem 2 (⋆). *FVS/UMC on instances G, S, k with $n = |V(G)|$, $k \geq 1$ and $\ell = |S|$ can be solved in time $O(n^3) \cdot \ell! \, 4^k k$.*

5   An Algorithm for S-Disjoint FVS: Contracting Paths

In this section we give an FPT algorithm for S-Disjoint FVS, by reducing it to FVS/UMC. Throughout this section, let $G = (V, E, A)$ be a mixed graph, and S be a FVS for G. The main idea of our algorithm is to try out different guesses for a set of edges $F \subseteq E$ that is not hit by a possible S-disjoint FVS S', and contract F. If a solution S' exists and the appropriate set F that corresponds to S' is considered, then S' remains a FVS, but in addition becomes a UMC. So in the resulting graph, we have an algorithm for finding S'. We now make this precise with the following definition and propositions. Let G^* be the graph obtained from G by contracting a set of edges $F \subseteq E$. Let the set S^* consist of all vertices in G^* resulting from a contraction, and all remaining S-vertices (those that were not incident with an edge from F). Then we say that G^*, S^* *is the result of contracting F in G, S*. The short proof of Proposition 2 is omitted, while Proposition 3 follows easily from the definitions.

Proposition 2 (⋆). *Let S be a FVS in a mixed graph $G = (V, E, A)$. Let G^*, S^* be the result of contracting a set $F \subseteq E$ in G, S, where $G[F]$ is a forest. Then a set $S' \subseteq V(G)$ is an S-disjoint FVS for G that is not incident with edges from F if and only if it is an S^*-disjoint FVS in G^*.*

Proposition 3. *Let S be a FVS in a mixed graph $G = (V, E, A)$, and let S' be an S-disjoint FVS for G. Let $F \subseteq E$ be the set of all edges that lie on a path between two S-vertices in $G[E] - S'$. Let G^*, S^* be the result of contracting F in G, S. Then S' is a UMC for G^*, S^*.*

The previous two propositions show that it is safe to contract sets of edges that contain no cycles (no solutions are introduced), and when considering the appropriate set, a possible solution S' indeed becomes a FVS and UMC in the resulting graph. The remaining task is to find a way to consider only a limited number ($2^{O(k)}$) of possibilities for F, while ensuring that a correct choice is among them. To this end we introduce the following definitions and bounds. The definitions are illustrated in Figure 3.

A *branching vertex* for G, S is a vertex v such that there are at least three internally vertex disjoint paths from v to S in $G[E]$. By $\mathcal{B} = \mathcal{B}(G, S)$ we denote the set of branching vertices for G, S. A *connection path* is a path in $G[E]$ with both end vertices in S and $\mathcal{B}$, and no internal vertices in S and $\mathcal{B}$.
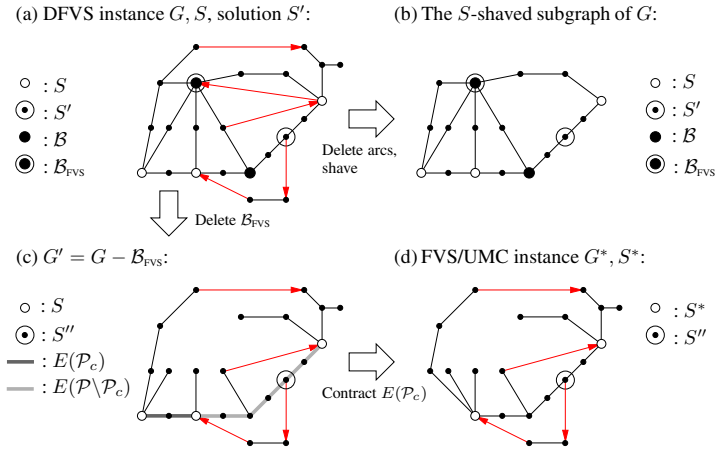
Fig. 3. The graphs and sets defined in Section 5

Before we give a bound on the number of branching vertices and connection paths, we introduce a different viewpoint on these notions. Given a mixed graph $G = (V, E, A)$ and a FVS S, we construct the *S-shaved subgraph of G* by starting with $G[E]$, and iteratively deleting non-S-vertices that have degree 1, as long as possible. Hence the S-shaved subgraph G_S of G is an undirected graph in which every non-S-vertex has degree at least 2. Considering the three internally vertex disjoint paths from a branching vertex $v \in \mathcal{B}$ to S, and using the fact that S-vertices are never deleted, we see that

---

Algorithm 1. An algorithm for S-Disjoint FVS

INPUT: A mixed graph $G = (V, E, A)$ with FVS S, and integer $k = |S| - 1$.
OUTPUT: a small S-disjoint FVS S' for G, S, or 'NO' if this does not exist.
1.   Compute the set $\mathcal{B}$ of branching vertices of G, S.
2.   **if** $|\mathcal{B}| > 3k$ **then return** 'NO'
3.   **for** all $\mathcal{B}_{\text{FVS}} \subseteq \mathcal{B}$ with $|\mathcal{B}_{\text{FVS}}| \leq k$:
4. $k' := k - |\mathcal{B}_{\text{FVS}}|$.
5. $G' := G - \mathcal{B}_{\text{FVS}}$.
6.   Compute the set $\mathcal{P}$ of connection paths of G', S.
7.   **if** $|\mathcal{P}| > 3k + k'$ **then continue**
8.   **for** all $\mathcal{P}_c \subseteq \mathcal{P}$ with $|\mathcal{P}| - |\mathcal{P}_c| \leq k'$:
9.   Let $F = E(\mathcal{P}_c)$.
             Let $F^* \subseteq F$ be the edges of components of $G'[F]$ containing an S-vertex.
10.   **if** $G'[F^*]$ contains a cycle **then continue**
11.   Construct G^*, S^* by contracting F^* in G', S.
12.   **if** G^* contains no loops incident with S^*-vertices and
                 there is a FVS and UMC S'' for G^*, S^* with $|S''| \leq k'$, **then**
13.   **return** $S' := S'' \cup \mathcal{B}_{\text{FVS}}$
14. **return** 'NO'

---

this process does not delete vertices from $\mathcal{B}$, and neither does it delete vertices from connection paths. Furthermore vertices in $\mathcal{B}$ still have degree at least 3 in G_S. In fact, it turns out that this is another way to characterize the branching vertices and connection paths of G, S:

Proposition 4 ($\star$). *Let $\mathcal{B}$ be the set of branching vertices of a mixed graph $G = (V, E, A)$ with FVS S, and let G_S be its S-shaved graph. Then $\mathcal{B}$ is exactly the set of non-S-vertices in G_S that have degree at least 3.*

The graph G_S can be thought of as a forest whose all leaves are in S, but where a vertex in S can be simultaneously used as a leaf for multiple paths, or "branches", of the forest. With this viewpoint in mind one can use counting arguments that relate the number of leaves and vertices of degree at least 3 in forests to prove the following lemma.

Lemma 3 ($\star$). *Let S be a FVS of a mixed graph $G = (V, E, A)$ with $k = |S| - 1$, and let S' be a small S-disjoint FVS for G, S. Then G has at most $3k$ branching vertices with respect to S, and G has at most $3k$ connection paths with no vertices in S'.*

We now present Algorithm 1, the algorithm for S-Disjoint FVS. Recall that the 'continue' statement continues with the next iteration of the smallest enclosing for- (or while-) loop, so it skips the remainder of the current iteration. Note that in Line 6, the set $\mathcal{P}$ of connection paths of $G' = G - \mathcal{B}_{\text{FVS}}$ is considered, not the connection paths of G. For a set of paths $\mathcal{P}_c \subseteq \mathcal{P}$, we denote by $E(\mathcal{P}_c)$ the set of all edges that occur in a path in $\mathcal{P}_c$. The following two lemmas prove the correctness of Algorithm 1.

Lemma 4. *Let S be a FVS of a mixed graph $G = (V, E, A)$ with $k = |S| - 1$. If Algorithm 1 returns a set $S' = S'' \cup \mathcal{B}_{\text{FVS}}$, then S' is a small S-disjoint FVS for G.*

Proof. Suppose a solution $S' = S'' \cup \mathcal{B}_{\text{FVS}}$ is returned in Line 13. Then S'' is a FVS and UMC for G^* and S^*, which are obtained from G', S by contracting the edge set $F^* \subseteq E(G')$. Since $G'[F^*]$ contains no cycles (otherwise the condition in Line 10 is satisfied), S'' is an S-disjoint FVS in G' (Proposition 2). Because $G' = G - \mathcal{B}_{\text{FVS}}$, $S'' \cup \mathcal{B}_{\text{FVS}}$ is then an S-disjoint FVS in G, of size at most $k' + |\mathcal{B}_{\text{FVS}}| = k$. $\qquad\square$

Lemma 5. *Let S be a FVS of a mixed graph $G = (V, E, A)$ with $k = |S| - 1$. If there exists a small S-disjoint FVS S' for G, then a solution is returned by Algorithm 1.*

Proof. Let S' be a small S-disjoint FVS, and let $\mathcal{B} = \mathcal{B}(G, S)$. By Lemma 3, $|\mathcal{B}| \leq 3k$, so the algorithm does not terminate in Line 2. Now consider the iteration of the for-loop in Line 3 that considers $\mathcal{B}_{\text{FVS}} := \mathcal{B} \cap S'$, and thus the graph $G' = G - \mathcal{B}_{\text{FVS}}$ and parameter $k' = k - |\mathcal{B}_{\text{FVS}}|$. Let $S'' = S' \backslash \mathcal{B}_{\text{FVS}}$, which is an S-disjoint FVS for G' of size at most k'. So we may apply the propositions and lemmas from this section to G', S and S''.

Observe that after deleting a subset $\mathcal{B}_{\text{FVS}}$ of branching vertices of G, some other vertices may lose their branching vertex status, but no branching vertices are introduced. In other words, $\mathcal{B}(G', S) \subseteq \mathcal{B}(G, S) \backslash \mathcal{B}_{\text{FVS}}$. Therefore, $S'' \cap \mathcal{B}(G', S) = \emptyset$. From Proposition 4 and the fact that all connection paths of G' are part of the S-shaved subgraph of G', it follows that connection paths of G' share no internal vertices. Combining these two facts shows that at most $|S''| \leq k'$ connection paths of G' are incident with a vertex from S''. Lemma 3 shows that G' contains at most $3k$ connection paths not incident with S'', so there are at most $3k + k'$ connection paths in total. Therefore, in Line 7, the algorithm does not continue to the next iteration.

Now let $\mathcal{P}$ be the set of connection paths of G', S, and let $\mathcal{P}_c \subseteq \mathcal{P}$ be those connection paths that are not incident with an S''-vertex. Since we observed that $|\mathcal{P} \backslash \mathcal{P}_c| \leq k'$, we may consider the iteration of the for-loop in Line 8 that considers $\mathcal{P}_c$. Note that the set F^* constructed in Line 9 contains all edges of G' that lie on some undirected path P between two S-vertices in $G' - S''$, since every such path P consists of a sequence of connection paths. Since S'' is a FVS for G', every component of $G'[F^*]$ is a tree, so in Line 10 the algorithm does not continue to the next iteration. Let G^*, S^* be obtained by contracting F^* in G', S. By Proposition 2, S'' is an S^*-disjoint FVS in G^*. By Proposition 3, S'' is a UMC for G^*, S^*. Hence in Line 13, a solution will be returned. $\qquad\square$

Proposition 5 ($\star$). *For all constants $c > 2$, $\sum_{i=0}^{k} \binom{ck}{i} \in O\left(\left(\frac{c^c}{(c-1)^{c-1}} \right)^k \right)$.*

Theorem 3 ($\star$). *On an instance $G = (V, E, A)$, S with $n = |V|$ and $k = |S| - 1$, Algorithm 1 correctly solves S-Disjoint FVS in time $O\left(k(k+1)! \, 47.5^k \, n^3 \right)$.*

Lemmas 4 and 5 show that Algorithm 1 returns the correct answer, so it only remains to prove the complexity bound. A detailed analysis is deferred to the full version of this article. We argue here that the complexity is bounded by $2^{O(k)} k! \cdot n^{O(1)}$: By Line 2, $|\mathcal{B}| \leq 3k$, so the number of iterations of the first for-loop is at most $\sum_{i=0}^{k} \binom{3k}{i} \in O(6.75^k)$ (Proposition 5). For every such iteration, let $k' = k - |\mathcal{B}_{\text{FVS}}|$. By Line 7, $|\mathcal{P}| \leq 3k + k' \leq 4k$ holds whenever the second for-loop is entered, so this loop iterates at most $\sum_{i=0}^{k'} \binom{4k}{i} \in O(9.49^k)$ times (Proposition 5). At most once for every

iteration, a FVS/UMC problem on the instance G^*, S^*, k' is solved, which can be done with parameter function $|S^*|! \cdot 4^{k'} k'$ (Theorem 2). By construction, every component of $G'[F^*]$ contains an S-vertex, so $|S^*| \leq |S|$, and therefore this contributes at most $(k+1)! 4^k k$ to the parameter function. Hence the total parameter function is bounded by $O(6.75^k \cdot 9.49^k \cdot 4^k \cdot k\ (k+1)!) \subseteq O(256.5^k\ k!)$. The running time dependence on n is dominated by solving the FVS/UMC problem in time $O(n^3)$ (Theorem 2), and the construction of G^*, S^*, which can also be shown to require $O(n^3)$ time. Combining Theorem 3 with Lemma 1 yields the main theorem of this paper.

Theorem 4. *In time $O\left((k+1)! k^2 47.5^k\ n^4\right)$, it can be decided whether a mixed graph $G = (V, E, A)$ with $|V| = n$ contains a FVS S with $|S| \leq k$.*

6   Discussion

Our research showed that for some problems, perhaps surprisingly, combining the undirected case with the directed case may provide a significant challenge. We therefore think that mixed graphs deserve more attention in the area of graph algorithms.

We remark that our algorithms can be used to decide whether a mixed graph G contains a set S of edges and arcs with $|S| \leq k$ such that $G - S$ is acyclic (*Feedback Edge/Arc Set (FE/AS)*). For undirected graphs, this is a trivial problem. For directed graphs this can easily be reduced to directed FVS, by subdividing all arcs with a vertex and replacing all original vertices with $k+1$ copies (to ensure that they are not selected in a FVS of size at most k). For mixed graphs, this last transformation does not work. However, in the full version of the article we extend our algorithms for a certain vertex weighted variant, which can then be used to solve FE/AS.

Our first question is whether the complexity of our algorithm can be improved, in particular whether the $k!$ factor can be removed. Not only does this factor asymptotically dominate the running time, but it also seems to be critical in practice: the 47.5^k factor is based on combining a number of upper bounds and it is unlikely that the worst case complexity bound actually applies to arbitrary instances.

Secondly, one may ask whether FVS in mixed graphs admits a polynomial kernelization (see e.g. [4,26]). Both questions seem to be very challenging, in fact they remain unresolved even when restricted to planar digraphs (see [4]).

References

1. Bafna, V., Berman, P., Fujito, T.: A 2-approximation algorithm for the undirected feedback vertex set problem. SIAM J. Discrete Math. 12, 289–297 (1999)
2. Becker, A., Geiger, D.: Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem. Artif. Intell. 83, 167–188 (1996)
3. Bodlaender, H.L.: On disjoint cycles. Int. J. Found. Comput. Sci. 5, 59–68 (1994)
4. Bodlaender, H.L., Fomin, F.V., Lokshtanov, D., Penninkx, E., Saurabh, S., Thilikos, D.M. (Meta) kernelization. In: FOCS 2009, pp. 629–638. IEEE, Los Alamitos (2009)

5. Cao, Y., Chen, J., Liu, Y.: On feedback vertex set new measure and new structures. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 93–104. Springer, Heidelberg (2010)
6. Chen, J., Fomin, F., Liu, Y., Lu, S., Villanger, Y.: Improved algorithms for feedback vertex set problems. J. Comput. Syst. Sci. 74, 1188–1198 (2008)
7. Chen, J., Liu, Y., Lu, S., O'sullivan, B., Razgon, I.: A fixed-parameter algorithm for the directed feedback vertex set problem. J. ACM 55, 1–19 (2008)
8. Dehne, F., Fellows, M.R., Langston, M., Rosamond, F., Stevens, K.: An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. Theor. Comput. Syst. 41, 479–492 (2007)
9. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness I: Basic results. SIAM J. Comput. 24, 873–921 (1995)
10. Downey, R.G., Fellows, M.R.: Parameterized complexity. Springer, New York (1999)
11. Even, G., Naor, J., Schieber, B., Sudan, M.: Approximating minimum feedback sets and multicuts in directed graphs. Algorithmica 20, 151–174 (1998)
12. Festa, P., Pardalos, P.M., Resende, M.: Feedback set problems. Handbook of Combinatorial Optimization A(Supplement), 209–258 (1999)
13. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Berlin (2006)
14. Fomin, F., Villanger, Y.: Finding induced subgraphs via minimal triangulations. In: STACS 2010. LIPIcs, vol. 5, pp. 383–394. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik (2010)
15. Fomin, F.V., Gaspers, S., Pyatkin, A.V., Razgon, I.: On the minimum feedback vertex set problem: Exact and enumeration algorithms. Algorithmica 52, 293–307 (2008)
16. Guo, J., Gramm, J., Hüffner, F., Niedermeier, R., Wernicke, S.: Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. J. Comput. Syst. Sci. 72, 1386–1396 (2006)
17. Jain, K., Hajiaghayi, M., Talwar, K.: The generalized deadlock resolution problem. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 853–865. Springer, Heidelberg (2005)
18. Kanj, I., Pelsmajer, M., Schaefer, M.: Parameterized algorithms for feedback vertex set. In: Downey, R.G., Fellows, M.R., Dehne, F. (eds.) IWPEC 2004. LNCS, vol. 3162, pp. 235–247. Springer, Heidelberg (2004)
19. Karp, R.: Reducibility among combinatorial problems. Complexity of Computer Computations, 85–103 (1972)
20. Mehlhorn, K.: Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness. Springer, Heidelberg (1984)
21. Niedermeier, R.: Invitation to fixed-parameter algorithms. Oxford University Press, Oxford (2006)
22. Raman, V., Saurabh, S., Subramanian, C.R.: Faster fixed parameter tractable algorithms for undirected feedback vertex set. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 241–248. Springer, Heidelberg (2002)
23. Raman, V., Saurabh, S., Subramanian, C.R.: Faster fixed parameter tractable algorithms for finding feedback vertex sets. ACM Trans. Algorithms 2, 403–415 (2006)
24. Razgon, I.: Computing minimum directed feedback vertex set in $O(1.9977^n)$. In: ICTCS, pp. 70–81 (2007)
25. Reed, B., Smith, K., Vetta, A.: Finding odd cycle transversals. Oper. Res. Lett. 32, 299–301 (2004)
26. Thomassé, S.: A $4k^2$ kernel for feedback vertex set. ACM Trans. Algorithms 6 (2010)

Switching to Directional Antennas with Constant Increase in Radius and Hop Distance

Prosenjit Bose[1], Paz Carmi[2], Mirela Damian[3],
Robin Flatland[4], Matthew J. Katz[5], and Anil Maheshwari[6]

[1] Carleton University, Ottawa, Canada
jit@scs.carleton.ca
[2] Ben-Gurion University, Beer-Sheva, Israel
carmip@cs.bgu.ac.il
[3] Villanova University, Villanova, P.A., USA
mirela.damian@villanova.edu
[4] Siena College, Loudonville, N.Y., USA
flatland@siena.edu
[5] Ben-Gurion University, Beer-Sheva, Israel
matya@cs.bgu.ac.il
[6] Carleton University, Ottawa, Canada
anil@scs.carleton.ca

Abstract. For any angle $\alpha < 2\pi$, we show that any connected communication graph that is induced by a set P of n transceivers using omni-directional antennas of radius 1, can be replaced by a strongly connected communication graph, in which each transceiver in P is equipped with a directional antenna of angle α and radius r_{dir}, for some constant $r_{\text{dir}} = r_{\text{dir}}(\alpha)$. Moreover, the new communication graph is a c-spanner of the original graph, for some constant $c = c(\alpha)$, with respect to number of hops.

Keywords: directional antennas, wireless networks, communication graph, hop spanner.

1 Introduction

Motivation and Problem Definition: Antennas used in wireless networks have traditionally been omni-directional. Such an antenna broadcasts in all directions, and its broadcast region can be represented geometrically by a disk centered at the transceiver. Recently, attention has been given to directional antennas which broadcast over a limited angle. The broadcast region of a directional antenna can be represented geometrically as a (closed) circular sector with an angular aperture α and radius r_{dir}. An antenna orientation is specified by a counter-clockwise angle θ measured from the positive x-axis to the sector's bisector. Directional antennas have the advantage of requiring less power compared to omni-directional antennas of the same radius, or by using the same power they can reach farther. In addition, narrower broadcast regions reduce interference and provide an added measure of security from eavesdroppers.

The *direction assignment problem* is the task of finding orientations for a set of directional antennas such that the induced communication graph has certain desired properties. Let P be a set of n points in the plane representing n transceivers each equipped with an antenna. The induced communication graph has a vertex for each point and an edge directed from a to b if and only if b's point is contained in the broadcast region of a's antenna. Let $DG(r)$ be the induced communication graph when the points in P are equipped with omni-directional antennas of radius r. We will assume r is sufficiently large to ensure that $DG(r)$ is connected. It is easy to see that to achieve connectivity, r must be at least as long as the longest edge in a Euclidean minimum spanning tree of P. Consider now the same point set but with each point equipped instead with a directional antenna of angle α. Our goal is to determine a small radius $r_{dir} = r_{dir}(r, \alpha)$ for the directional antennas and to assign orientations to them, such that the resulting communication graph, G_{dir}, is (i) strongly connected, and (ii) for any two points $p, q \in P$, the number of hops (i.e., edges) in a minimum-hop path from p to q in G_{dir} is bounded by some constant $c = c(\alpha)$ times the number of hops in a minimum-hop path from p to q in $DG(r)$. In other words, condition (ii) requires that G_{dir} is a c-spanner of $DG(r)$ with respect to number of hops. Without loss of generality, we will assume going forward that $r = 1$.

New results: For $\alpha \leq \pi/3$, we show (in Section 2) that by fixing the radius of the antennas to $4\sqrt{2}(3.5k - 6)$, where $k = \lceil 2\pi/\alpha \rceil$, one can assign orientations to the antennas such that G_{dir} is strongly connected and a $(\lceil 8\log k \rceil - 1)$-spanner of $DG(1)$ with respect to the number of hops. In Section 3, we show that for the special case of $\alpha = \pi/3$ a radius of $36\sqrt{2}$ is sufficient to obtain a hop 10-spanner. This result immediately yields a hop 10-spanner using a radius of $4\sqrt{2}(3 + k)$, for any $\alpha > \pi/3$. We also consider a path version of the problem in Section 4 where $DG(1)$ is assumed to be a path, and prove that a radius of $\lceil (3k + 3)/2 \rceil$ is sufficient to obtain a hop $(2k + 4)$-spanner, for $\alpha \leq \pi/3$. For $\alpha > \pi/3$, the result holds (trivially) using the same radius and hops as for $\alpha = \pi/3$. We note that although our directed networks have the advantages of reduced interference and added security, they do not achieve a power savings over the omni-directional network $DG(1)$ since the power savings of the smaller broadcast angle is offset by the larger radius $r_{\mathrm{dir}} > 1$. We leave it as an open problem to find directed networks with r_{dir} small enough to achieve a power savings.

Related Work: Caragiannis et al. [5] consider the problem of orienting directional antennas to form a strongly connected communication graph using minimal r, but they do not attempt to minimize the hops. For any $\alpha \geq 0$, they present an algorithm that constructs a communication graph containing a Hamiltonian tour and achieves a 3-approximation of the optimal radius. However, the number of hops between two nodes at unit distance or less can be linear. In [7], Damian and Flatland minimize both the radius and hops (as we do here) but for antennas with $\alpha \geq \pi/2$. Their approach depends fundamentally on finding orientations for small, proximate groups of antennas such that together the

antennas completely cover an encompassing circular region and at the same time form a strongly connected sub-network amongst themselves. This, however, is not generally achievable for $\alpha < \pi/2$ — consider for example the case when the antennas all lie on a line — and thus their approach does not generalize to smaller angles. To our knowledge, our work here is the first to consider minimizing both the radius and hops for small angles $\alpha < \pi/2$.

In other related work, Nijnatten [10] considers a variant of the problem in which each antenna may have a different radius and the goal is to minimize the overall power consumption of the network. Ben-Moshe et al. [2] consider antennas with $\alpha = \pi/2$ but restrict the orientations to one of the four standard quadrant directions. Carmi et al. [6] show that for any set of points equipped with $\pi/3$ antennas, one can direct the antennas so that the resulting undirected communication graph is connected, assuming the antennas have a radius equal to the diameter of the point set. Ackerman et al. [1] later presented a simpler proof for the same result. Bhattacharya et al. [3] and Dobrev et al. [8] consider transceivers with multiple directional antennas; in the former they focus on minimizing the sum of the antenna angles for a fixed r, and in the latter they show that with k antennas per node, strong connectivity can be achieved using a radius that is at most $2\sin(\frac{\pi}{k+1})$ times the optimal, for any $\alpha \geq 0$. For a survey of recent results on directional antennas, see [9].

2 $\alpha \leq \pi/3$

Recall that $DG(1)$ is the communication graph induced by omni-directional antennas of radius 1 positioned at the points in P. Without loss of generality, we assume the point set is normalized so that the longest edge in a minimum spanning tree of P is at most 1, and thus $DG(1)$ is connected. Replacing each of the omni-directional antennas with a directional antenna of angle $\alpha \leq \pi/3$, we describe here how to determine orientations and a small radius $r_{\text{dir}} = r_{\text{dir}}(\alpha)$ for the antennas such that (i) the resulting communication graph, G_{dir}, is strongly connected, and (ii) for any two points $p, q \in P$, the number of hops in a minimum-hop path from p to q in G_{dir} is bounded by some constant $c = c(\alpha)$ times the number of hops in a minimum-hop path from p to q in $DG(1)$.

Lemma 1. *Let Q be a set of $m \geq 3$ points in the plane. Then, there exist three points $a, b, c \in Q$, such that $\angle abc \leq \pi/m$.*

Proof. Let $m' \leq m$ be the number of vertices in $\text{CH}(Q)$, the convex hull of Q. Then, the sum of the angles at the vertices of $\text{CH}(Q)$ is $(m'-2)\pi$, and there exists a vertex v whose corresponding angle is at most $(m'-2)\pi/m'$. Connect v to each of the points in Q. We obtain $(m-2)$ wedges, and the angle of at least one of them is bounded by $\frac{(m'-2)\pi/m'}{m-2} \leq \frac{\pi}{m}$, since $\frac{m'-2}{m'} \leq \frac{m-2}{m}$. $\square$

For simplicity of exposition, assume for now α is such that $k = \lceil 2\pi/\alpha \rceil$ is a power of 2. We will later eliminate this restriction (in Theorem 1). We first describe our main building block.
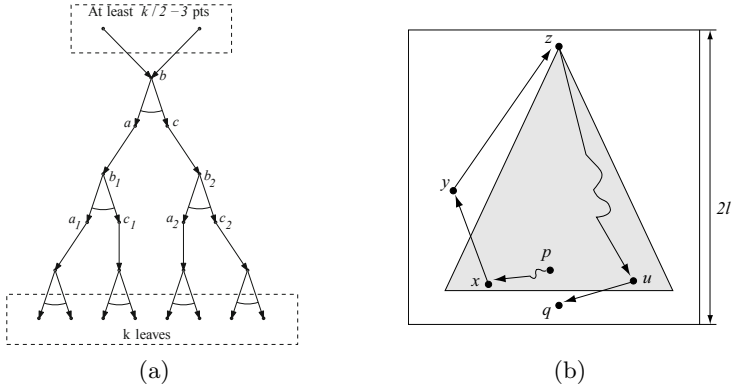
Fig. 1. (a) The communication structure of Theorem 1. (b) A path from p to q.

Tree Construction. We start with the assumption that there are sufficient points available for the construction described here. Later, we will determine a minimum number of points that are necessary to carry out this construction. Let Q be an arbitrary set of points. We construct a rooted binary tree whose nodes are points in Q and whose edges are directed towards the leaves (see Figure 1a). The tree has k leaves, and the angle formed at each internal node is at most α. We construct this tree as follows. Use Lemma 1 to pick three points $a, b, c \in Q$, such that $\angle abc \leq \alpha$. Make b the root of the tree and a and c the left and right children of b, respectively; remove points a, b, c from Q. Now, use Lemma 1 once again to obtain three new points $a_1, b_1, c_1 \in Q$, such that $\angle a_1 b_1 c_1 \leq \alpha$. Make b_1 the single child of a, and make a_1 and c_1 the left and right children of b_1, respectively; remove points a_1, b_1, c_1 from Q. In the next application of Lemma 1, we obtain three new points $a_2, b_2, c_2 \in Q$, such that $\angle a_2 b_2 c_2 \leq \alpha$, and make b_2 the single child of c, etc. We continue applying Lemma 1 until we obtain a balanced tree consisting of $3k - 3$ nodes, of which k are leaves, $k - 1$ have two children each, and $k - 2$ have one child each. See Figure 1a.

We now analyze the minimum number of points in Q necessary for this construction. Note that, in the last application of Lemma 1, we need $\pi/|Q| \leq \alpha$, to be able to select three points in Q forming an angle of at most α. So at least $\pi/\alpha \leq \pi/(2\pi/k) = k/2$ points are necessary in the last iteration. Therefore, $|Q|$ must be at least $(3k-3)+(k/2-3) = 3.5k-6$, since $(3k-3)$ points are selected (as established above), and at least $(k/2 - 3)$ points remain in Q after the last iteration. So the minimum size of Q is $\ell = 3.5k - 6$.

From this point on, whenever we use the term *tree*, we refer to the rooted tree constructed using this method. We refer to those points that are in the tree as *nodes*, to distinguish them from the points not in the tree.

Antenna Orientations. Once we have built our tree, we assign an orientation to each of the antennas at the points in Q according to the following rules:

(A1) At each node p in the tree, orient p's antenna to induce the directed
      edge(s) outgoing from p. This is always possible, because the angle
      spanned by the outgoing edge(s) at each node does not exceed α.
(A2) At each point $p \in Q$ that is not in the tree, orient p's antenna to point
      to the root of the tree (refer to Figure 1a).
(A3) At the k tree leaves, assign antenna orientations so that collectively they
      cover the whole plane (assuming infinite range). By the result by Bose et
      al. [4], this is always possible with k antennas of angle at least $2\pi/k \leq \alpha$.

The following lemma summarizes the properties of the resulting communication
structure. All paths in this structure are *directed*.

Lemma 2. *Let Q be a set of points, each representing a transceiver equipped
with a directional antenna of angle $\alpha \leq \pi/3$ and range $diam(Q)$. Assume that
$k = \lceil 2\pi/\alpha \rceil$ is a power of 2 and that $|Q| \geq 3.5k - 6$. Then, one can assign an
orientation to each of the antennas at the points in Q, such that the resulting
directed graph contains a rooted tree T with the following properties: (i) for any
node q of T other than the root, there exists a path from q to a leaf of T consisting
of at most $(2\log k - 2)$ hops, and (ii) for any point $q \in Q$, there exists a path
from the root of T to q consisting of at most $(2\log k)$ hops.*

Proof. Property (i) is immediate, since the number of hops from the root of T
to a leaf equals the height of T, which is $(2\log k - 1)$; since q is assumed to be a
non-root node, the worst case occurs when q is a child of the root, which yields
$(2\log k - 2)$ hops. Rule (A3) for orienting the antennas at the leaf nodes in T
guarantees the existence of a leaf node p that covers q. This hop, summed up
with the $(2\log k - 1)$ from the root to p, yields the claimed bound of $(2\log k)$
hops from the root to q.

We now describe how to direct the antennas of the transceivers in the input set
P. Recall that we are assuming that $r = 1$. We may assume that $|P| \geq \ell$, because
otherwise, the distance between any two points in P is bounded by $\ell - 2$, and we
can set $r_{\mathrm{dir}} = \ell - 2$ and form a directed cycle with at most $\ell - 2$ hops between
any two points. Lay a grid $\mathcal{G}$ over P such that the length of each side of a cell is
2ℓ. Let $\mathcal{C}$ be a cell of $\mathcal{G}$, and define the *block of $\mathcal{C}$* as the 3×3 portion of $\mathcal{G}$ that
is centered at $\mathcal{C}$. Each of the 8 cells surrounding $\mathcal{C}$ is a *neighbor of $\mathcal{C}$*. A cell of $\mathcal{G}$
is considered *full* if it contains at least ℓ points of P. It is considered *non-full* if
it contains at least one point of P, but less than ℓ points of P.

Proposition 1. *Let $\mathcal{C}$ be a cell of $\mathcal{G}$. Then, any path in $DG(1)$ that begins at
a point in $\mathcal{C}$ and exits the block of $\mathcal{C}$, must pass through a full cell in $\mathcal{C}$'s block
(not including $\mathcal{C}$ itself, which may or may not be full). In particular, if there are
points of P outside $\mathcal{C}$'s block, then at least one of the 8 neighbors of $\mathcal{C}$ is full.*

Proof. This follows immediately from the fact that the maximum distance be-
tween any two adjacent points along this path is 1, and the edge length of a cell
is 2ℓ. See Figure 2(a). □

Let F be the set of full cells of $\mathcal{G}$. We distinguish between two cases: $F = \emptyset$ and $F \neq \emptyset$. If $F = \emptyset$, then it is easy to see (using considerations similar to those used in the proof of Proposition 1) that P can be enclosed in a $2\ell \times 2\ell$ square. In other words, one can enclose P within a single grid cell, by shifting the grid if necessary. In this case, we simply set $r_{\mathrm{dir}} = 2\sqrt{2}\ell$ and apply the construction used by Lemma 2 to the set P, with one simple modification:

(A3a) For each leaf node p of the tree, if p's antenna covers only tree nodes, then rotate it so that it covers the root of the tree.

This alteration guarantees that p can reach any point in P, via the tree root. Note that Lemma 2 still holds after reorienting some antennas, according to the rule (A3a) above. Let G_{dir} be the induced communication graph.

Lemma 3. G_{dir} *is a directed, strongly connected,* $(4 \log k)$*-spanner of* $DG(1)$, *with respect to number of hops.*

Proof. Pick an arbitrary edge $pq \in DG(1)$. We show that G_{dir} contains a path from p to q with at most $(4 \log k)$ hops. Refer to Figure 1b. If p is a tree node other than the root, then by Lemma 2(i), there is a path in G_{dir} from p to a tree leaf x, with at most $(2 \log k - 2)$ hops. Rule (A3a) for reorienting the antennas at the leaves (if necessary), guarantees that x points either to the root z of the tree, or to a non-tree point $y \in P$, which in turn points to z. By Lemma 2(ii), there is a path from z to q in G_{dir} with at most $2 \log k$ hops. Concatenating these paths together yields a path from p to q in G_{dir} with at most $(2 \log k - 2) + 2 + (2 \log k) = (4 \log k)$ nodes. The remaining cases when p is the root of the tree, or a non-tree point in P, are subsumed by the case when p is a tree node discussed above. $\square$

Assume now that $F \neq \emptyset$, i.e., there exists at least one full cell. Before discussing how to handle full cells, we introduce a few definitions. For any full cell $\mathcal{C}^+ \in F$, let $T(\mathcal{C}^+)$ denote the tree associated with $\mathcal{C}^+$. We say that the antenna at a leaf node of $T(\mathcal{C}^+)$ is *useful*, if one of the following three conditions holds: (i) it covers the root of either $T(\mathcal{C}^+)$, or the tree associated with another full cell in $\mathcal{C}^+$'s block; (ii) it covers a point in $\mathcal{C}^+$ that is not a node in $T(\mathcal{C}^+)$; (iii) it covers a point in a non-full cell in $\mathcal{C}^+$'s block, whose antenna covers the root of $T(\mathcal{C}^+)$. These situations are depicted in Figure 2(b)-(d). Define the *hop-distance between a point* $p \in P$ *and a full cell* $\mathcal{C}^+ \in F$ as the number of hops in a minimum-hop path in $DG(1)$ between p and a point in $\mathcal{C}^+$.

We are now ready to define the orientations of the antennas at the points of P:

(A4) For each full cell $\mathcal{C}^+ \in F$, apply the construction of Lemma 2 to its corresponding set of points. Then, for each leaf x of $T(\mathcal{C}^+)$, if the antenna at x is not useful, reorient it so that it covers the root of $T(\mathcal{C}^+)$.

(A5) For each non-full cell $\mathcal{C}^-$ and for each point $p \in \mathcal{C}^-$, direct the antenna at p to the root of $T(\mathcal{C}^+)$, where $\mathcal{C}^+$ is the (hop-distance) closest full cell to p. Ties are broken arbitrarily.
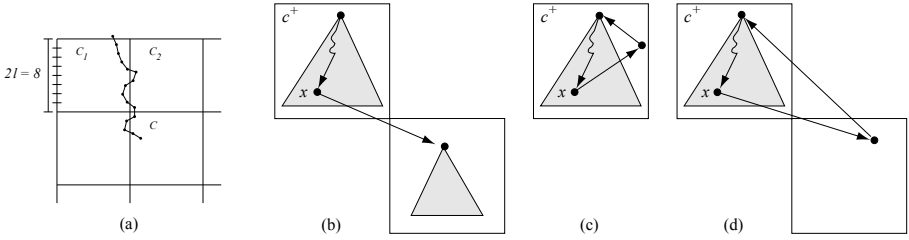
Fig. 2. (a) At least one of $\mathcal{C}$'s neighbors ($\mathcal{C}_1$) is full, (b-d) Situations in which leaf node x is useful

Note that Lemma 2 still holds, when applied to the point set restricted to a single full cell $\mathcal{C}^+$, and the full cell that determines the direction of the antenna at a point p that lies in a non-full cell $\mathcal{C}^-$, is a neighbor of $\mathcal{C}^-$ (this follows from Proposition 1). Lemma 4 below identifies two key properties of the leaf nodes in a full cell $\mathcal{C}^+$.

Lemma 4. *For any full cell $\mathcal{C}^+$ and any leaf node q of $T(\mathcal{C}^+)$, the following properties hold: (i) there is a path from q to the root of $T(\mathcal{C}^+)$ consisting of at most $(2 \log k + 1)$ hops, and (ii) there is a path from q to the root of the tree associated with any full cell in $\mathcal{C}^+$'s block, via the root of $T(\mathcal{C}^+)$, with at most $(4 \log k + 1)$ hops.*

Proof. Our reorientation rules force q to be useful, meaning that q either points to the root of a tree T in $\mathcal{C}^+$'s block (see Figure 2(b)), or can reach the root of $T(\mathcal{C}^+)$ in two hops (see Figures 2(c),(d)). Recall that the original orientation of the antennas at the k leaves of T guaranteed coverage of the entire plane, at infinite range (Rule A3). In particular, one of these antennas covers the root of $T(\mathcal{C}^+)$, proving itself useful (and therefore not subject to reorientation). These two hops (from q to the root of T, and from the leaf of T to the root of $T(\mathcal{C}^+)$), summed up with the $(2 \log k - 1)$ hops from the root to a leaf of T, yield $(2 \log k + 1)$ hops. So Property (i) is settled. To settle Property (ii), pick an arbitrary full cell $\mathcal{C}$ in $\mathcal{C}^+$'s block. We extend the path from the root of $T(\mathcal{C}^+)$ to the root of $T(\mathcal{C})$, in a similar way: one of the leaves of $T(\mathcal{C}^+)$ must cover the root of $T(\mathcal{C})$, and we can reach this leaf from the root of $T(\mathcal{C}^+)$ in $(2 \log k - 1)$ hops. Summing these up, we obtain $(2 \log k + 1) + (2 \log k) = (4 \log k + 1)$ hops. □

Finally, we set the radius r_{dir} of all antennas to $4\sqrt{2}\ell$, so that a point p can reach any other point in a neighboring cell (assuming the antenna at p is directed accordingly). Let G_{dir} be the resulting communication graph.

Lemma 5. *G_{dir} is a directed, strongly connected, $(8 \log k - 1)$-spanner of $DG(1)$, with respect to number of hops.*

Proof. We prove that, for any edge pq of $DG(1)$, there exists a path from p to q in G_{dir} consisting of at most $(8 \log k - 1)$ hops. Let $\mathcal{C}_p$ be the cell containing p, and $\mathcal{C}_q$ the cell containing q. Then, either $\mathcal{C}_p = \mathcal{C}_q$, or $\mathcal{C}_p$ and $\mathcal{C}_q$ are neighboring cells, because $|pq| \leq 1$. We discuss the cases where $\mathcal{C}_p$ is full or non-full.
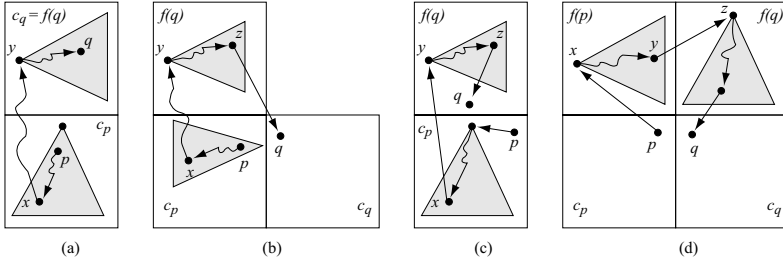
Fig. 3. Path from p to q in G_{dir}. (a) q is a node of $T(f(q))$. (b) q is not a node of $T(f(q))$. (c) p is not a node of $T(\mathcal{C}_p)$. (d) $\mathcal{C}_p$ is not full.

$\mathcal{C}_p$ **is full.** Associate with q a full cell, $f(q)$, as follows. If $\mathcal{C}_q$ is full, then $f(q) = \mathcal{C}_q$. Otherwise, $f(q)$ is the full cell that determines the direction of q's antenna. Since pq is an edge of $DG(1)$, the hop-distance between q and $\mathcal{C}_p$ is at most 1, so the hop-distance between q and $f(q)$ is at most 1. It follows that $f(q)$ is either $\mathcal{C}_p$, or a neighbor of $\mathcal{C}_p$. We now show how to reach q from p via the root of $T(f(q))$. We begin with the worst case scenario, in which p is a node in $T(\mathcal{C}_p)$ other than the root. In this case, from p we can reach an arbitrary leaf node x of $T(\mathcal{C}_p)$, in at most $(2\log k - 2)$ hops (by Lemma 2(i)). Next, we follow the path established in the proof of Lemma 4(ii) to reach the root y of $T(f(q))$, in at most $(4\log k + 1)$ hops. (Notice that $f(q)$ is in $\mathcal{C}_p$'s block, thus enabling us to use Lemma 4.) Finally, from y we follow the path in $T(f(q))$ that leads directly to q, if q is a node in $T(f(q))$ (see Figure 3a); otherwise, we follow the path that leads to the leaf z of $T(f(q))$ that covers q (see Figure 3b). Note that z always exists, since $\mathcal{C}_q$ and $f(q)$ are neighboring cells and q points to y. This latter path is longer, and has $(2\log k - 1) + 1$ hops. Summing up the number of hops along the entire path from p to q, we obtain a total of $(2\log k - 2) + (4\log k + 1) + (2\log k) = (8\log k - 1)$ hops. The cases in which p is the root of $T(\mathcal{C}_p)$ or a point not in $T(\mathcal{C}_p)$ are similar.

$\mathcal{C}_p$ **is non-full.** Let $f(p)$ be the full cell that determines the direction of p's antenna. Since pq is an edge of $DG(1)$, the difference between the hop-distance between p and $f(p)$ and the hop-distance between q and $f(q)$ is at most 1. Moreover, from the definition of $f(p)$ and $f(q)$, it follows that the corresponding paths (in $DG(1)$) from p to $f(p)$ and from q to $f(q)$ do not pass through a full cell (except at their final point). (If $f(q)$ is full, then the latter path is simply the degenerate path q.) This implies that $f(q)$ is either $f(p)$ or a neighbor of $f(p)$. We now show how to reach q from p via the root of $T(f(q))$. From p, we can reach the root of $T(f(p))$ in one hop. Next we follow the path in $T(f(p))$ from x to that leaf node y that covers the root z of $T(f(q))$; this path has $(2\log k - 1)$ hops. One hop takes us from y to z, then $(2\log k - 1)$ more hops take us from z to that leaf node of $T(f(q))$ that covers q. See Figure 3d. The total number of hops along the entire path from p to q is therefore at most $1 + (2\log k) + (2\log k) = (4\log k + 1)$. □

The following theorem summarizes the main result of this section.

Theorem 1. *Let P be a set of n points, each representing a transceiver equipped with a directional antenna of radius $\alpha \leq \pi/3$, and assume that $DG(1)$ is connected. Then, one can assign a direction to each of the n antennas, such that by fixing their transmission range to $4\sqrt{2}(3.5k - 6)$, where $k = \lceil 2\pi/\alpha \rceil$, the resulting communication graph is strongly connected. Moreover, it is a $(\lceil 8 \log k \rceil - 1)$-spanner of $DG(1)$, with respect to number of hops.*

Proof. The assumption that k is a power of two is eliminated here, so the tree with k leaves, used as the core communication structure within each full cell, is not necessarily balanced. More precisely, the bottom level of the tree may be incomplete. In this case, $(\lceil 2 \log k \rceil - 1)$ is an upper bound on the height of the tree. The result of this theorem follows immediately from Lemma 5.   □

In concluding this section, we observe (without proof, due to space constraints) that our method for orienting antennas takes linear time in the number of points.

3 $\alpha = \pi/3$

We now show that we can obtain better constants for the special case $\alpha = \pi/3$. In our solution, we use the following result by Ackerman et al. [1]:

Proposition 2. *[1] Let Q be a set of points in the plane, equipped with antennas of angle $\pi/3$ and range $diam(Q)$. There exist three points $x, y, z \in Q$, whose antennas can be oriented such that: (i) y covers x and z and both x and z cover y, and (ii) every point in Q is covered by at least one of $\{x, y, z\}$.*

Note that Proposition 2 does not claim that the three antennas can cover the entire plane; in fact, this would not be possible with three antennas only. The result is tied to a fixed point set Q: the three points are carefully selected from among the points on the convex hull of Q, so that collectively they cover Q.

For a point set Q of at least nine points, we construct a rooted tree structure T as follows. Select points x, y, z as in Proposition 2 and orient their antennas accordingly. Make y the root of the tree and make x and z its children. Select any six additional points in $Q \backslash \{x, y, z\}$, and for each, make it the child of a point in $\{x, y, z\}$ whose antenna covers it. Orient the antennas of these six points so that collectively they cover the entire plane when their radiuses are infinity. These six leaves serve the same function as the k leaves of the tree structure of Section 2. We note that x or z may also be a leaf of T; for example, x is a leaf if the six additional points are only covered by z's antenna and thus are all made children of z. But to remain consistent with Section 2, in what follows when we refer to the leaves of T, we are only refering to the six additional points, not x or z. To complete the construction, for each point in Q not in the tree, orient its antenna to point to y.

This tree structure is analogous to the tree of Section 2 (with $\alpha = \pi/3$), but it has better hop spanning properties because it has smaller height (at most 2). It is easy to verify properties analogous to those in Lemma 2 from Section 2 for the tree here. Specifically, for any non-root node, there exists a path from it to

a leaf consisting of at most 3 hops (property (i)). For an example requiring 3 hops, consider the path from x to a leaf when all six leaves are children of z. For property (ii), 3 hops are sufficient to go from the root y to any point in Q, since the height of the tree is at most 2 and the leaves cover all points in Q.

Orienting the antennas of the input set P is now done the same as in Section 2, but using the tree structure above in each full cell. Since 9 points are sufficient to build the tree, we set $l = 9$. The grid cells are of size 18×18, and a cell is *full* if it encloses 9 or more points. By concatenating the same paths as in Lemma 4 but using the smaller trees described here, we immediately get the following two analogous properties: for any full cell $\mathcal{C}^+$ and any leaf node q of $T(\mathcal{C}^+)$, (i) there is a path from q to the root of $T(\mathcal{C}^+)$ consisting of at most 4 hops, and (ii) there is a path from q to the root of the tree associated with any full cell in $\mathcal{C}^+$'s block consisting of at most 7 hops. In fact, for the tree structure in this section, it is easy to show that these two properties also hold for the non-leaf nodes $\{x, y, z\}$ of $T(\mathcal{C}^+)$. This is true because both x and z's antennas cover the root y. So from a non-leaf node, it is at most 1 hop to the root y, at most 2 hops down $T(\mathcal{C}^+)$ to the leaf that covers the root of the desired tree in $\mathcal{C}^+$'s block, and then 1 hop to that neighboring tree's root.

By following the same arguments in Lemma 5 but using the smaller trees, we can show that the induced communication graph, G_{dir}, is a strongly connected 10-spanner of $DG(1)$, with respect to hops. As in Lemma 5, the worst case hop count in going from point p to q (where pq is an edge in $DG(1)$) occurs when $\mathcal{C}_p$ is full and p is a node in $T(\mathcal{C}_p)$ that is not the root. In this case, the path goes from p to the root of the neighboring tree $T(f(q))$ (worst case 7 hops, when p is a leaf), down $T(f(q))$ to the leaf that covers q (2 hops), and then to q (1 hop). We now set each antenna range to $36\sqrt{2}$, so that any two points that lie in neighboring cells can reach each other. Let G_{dir} be the resulting communication graph. The following theorem summarizes the main result of this section.

Theorem 2. *Let P be a set of n points, each representing a transceiver equipped with a directional antenna of angle $\alpha = \pi/3$, and assume that $DG(1)$ is connected. Then, one can assign a direction to each of the n antennas, and a transmission range of $36\sqrt{2}$, such that the resulting communication graph is a directed, strongly connected, 10-spanner of $DG(1)$, with respect to number of hops.*

We observe that this result also applies to angles $\alpha > \pi/3$ when using a radius $r_{dir} = 4\sqrt{2}(3 + \lceil 2\pi/\alpha \rceil)$, noting that in this case $\ell = 3 + \lceil 2\pi/\alpha \rceil$. The hop count remains the same.

4   Points along a Path

In this section we consider the special case in which the graph $DG(1)$ is a path. Let k be the smallest integer such that $2\pi/k \le \alpha$, and let $m = \lceil |P|/k \rceil$. Partition $DG(1)$ into a sequence of m subpaths, the first $m-1$ of which have exactly $k+3$ points each. Let P_i denote the sequence of points along the ith subpath. Thus each P_i, for $i < m$, contains $k+3$ points (P_m may have fewer points).
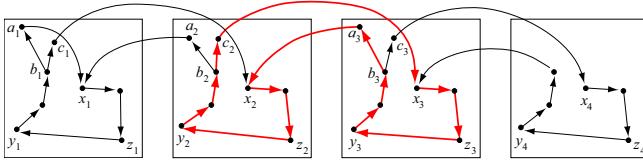
Fig. 4. The highlighted path shows that the hop bound of Theorem 3 is tight

Consider a sequence P_i with $k + 3$ points. Define the *entry* point of P_i to be the midpoint x_i of P_i. Let y_i and z_i be the start and end point of the sequence P_i, respectively. From the point set $Q_i = P_i \setminus \{x_i, y_i, z_i\}$, select three points a_i, b_i and c_i, such that $\angle a_i b_i c_i \leq \alpha$. By Lemma 1, such points always exist, because $|Q_i| = k$. This enables us to orient b_i's antenna so that it covers both a_i and c_i. At each point p in the sequence $R_i = P_i \setminus \{a_i, b_i, c_i\}$, orient p's antenna so that it covers the next point in R_i, with two exceptions: the antenna at the predecessor of x_i is oriented to cover b_i, and the antenna at z_i covers y_i. Finally, orient a_i's antenna to cover the entry point x_{i-1} in the sequence P_{i-1}, if $i > 1$, or the entry point x_i of the same sequence, if $i = 1$. Orient c_i's antenna to cover the entry point x_{i+1} in the sequence P_{i+1}.

If P_m contains $k + 3$ points as well, then we use the same approach for antenna orientations, with the difference that c_m's antenna points to x_m. If P_m contains fewer than $k + 3$ points, then we orient the antenna at each point to cover the next point in the sequence, with two exceptions: the antenna at the predecessor of x_m covers x_{m-1}, and the antenna at z_m covers y_m. A specific example is depicted in Figure 4. Now note that a path from c_i to x_{i+1} in $DG(1)$ can go through at most $k + 1$ points in P_i (this accounts for all points in P_i, with the exception of y_i and a_i), and at most $\lceil (k + 3)/2 \rceil$ points in P_{i+1} (because x_{i+1} is the midpoint of the sequence). So the number of hops from c_i to x_{i+1} in $DG(1)$ is bounded above by $\lceil (3k + 3)/2 \rceil$. The link from c_i to x_{i+1} (and symmetrically, from a_i to x_{i-1}) is one of the longest links that needs to be realized by an antenna, therefore we set $r_{\mathrm{dir}} = \lceil (3k + 3)/2 \rceil$. Lemma 6 below (whose proof is omitted due to space restrictions) summarizes properties of the resulting communication graph, G_{dir}.

Lemma 6. *For each $i < m$, G_{dir} contains directed paths from a_i to x_i, and from c_i to x_i, each with at most $k + 3$ hops. In addition, G_{dir} contains a directed cycle that passes through all points in $P_i \cup P_{i+1} \setminus \{a_i, c_{i+1}\}$, with at most $2k + 4$ hops.*

Theorem 3. *Let P be a set of n points, each representing a transceiver equipped with a directional antenna of radius $\alpha \leq \pi/3$, and assume that $DG(1)$ is a path. Then, one can assign a direction to each of the n antennas, such that by fixing their transmission range to $\lceil (3k + 3)/2 \rceil$, where $k = \lceil 2\pi/\alpha \rceil$, the resulting communication graph G_{dir} is a directed, strongly connected, $(2k + 4)$-spanner of $DG(1)$, with respect to the number of hops.*

Proof. We prove that, for each edge pq of $DG(1)$, there exists a path from p to q in G_{dir} consisting of at most $(2k + 4)$ hops. Note that p and q are either in

the same sequence P_i, or in adjacent sequences P_i and P_{i+1}, for some $i \geq 1$. If P_m contains fewer than $k + 3$ points, assume $i < m - 1$. By Lemma 6, there is a cycle passing through all points in $P_i \cup P_{i+1} \setminus \{a_i, c_{i+1}\}$, with at most $2k + 4$ hops. If both p and q are on this cycle, then at most $2k + 3$ hops separate q from p. Otherwise, p and q must belong to a same sequence, because neither a_i nor c_{i+1} is a start or end point of a sequence (by our construction), and therefore it cannot be adjacent in $DG(1)$ to a point in a different sequence. If $p = a_i$, then one can go from p to x_i in at most $k + 3$ hops (by Lemma 6), then from x_i to q in at most $k + 1$ additional hops, for a total of at most $2k + 4$ hops. The situations $p = c_{i+1}$, $q = a_i$, and $q = c_{i+1}$ are similar.

The case when P_m contains fewer than $k + 3$ points and $i = m - 1$ is similar: G_{dir} contains a directed cycle with at most $2k + 4$ hops, that passes through all points in $P_m \cup P_{m-1} \setminus \{a_{m-1}\}$. From this point on, the analysis is similar to the one above.   □

We note that the transmission radius for the path case discussed in this section is smaller than the transmission radius for the general case (Section 2) by a factor of 26.4. And, although the hop count established by Theorem 3 increases linearly with $1/\alpha$, (as opposed to logarithmically in the general case,) the hop count for the path case is smaller than the hop count for the general case for any $k \leq 16$, which corresponds to $\alpha \geq \pi/8$.

References

1. Ackerman, E., Gelander, T., Pinchasi, R.: On connected wedge-graphs. (manuscript) (2010)
2. Ben-Moshe, B., Carmi, P., Chaitman, L., Katz, M.J., Morgenstern, G., Stein, Y.: Direction Assignment in Wireless Networks. In: CCCG 2010, pp. 39–42 (2010)
3. Bhattacharya, B., Hu, Y., Shi, Q., Kranakis, E., Krizanc, D.: Sensor network connectivity with multiple directional antennae of a given angular sum. In: IPDPS 2009, pp. 1–11 (2009)
4. Bose, P., Guibas, L., Lubiw, A., Overmars, M., Souvaine, D., Urrutia, J.: The floodlight problem. J. Assoc. Comput. Mach. 9, 399–404 (1993)
5. Caragiannis, I., Kaklamanis, C., Kranakis, E., Krizanc, D., Wiese, A.: Communication in wireless networks with directional antennae. In: SPAA, pp. 344–351 (2008)
6. Carmi, P., Katz, M.J., Lotker, Z., Rosén, A.: Connectivity guarantees for wireless networks with directional antennas. (manuscript) (2009)
7. Damian, M., Flatland, R.: Spanning Properties of Graphs Induced by Directional Antennas. In: Electronic Proc. of the 20th Fall Workshop on Computational Geometry, Stony Brook, NY (2010)

8. Dobrev, S., Kranakis, E., Krizanc, E., Opatrny, J., Stacho, L.: Strong Connectivity in Sensor Networks with Given Number of Directional Antennae of Bounded Angle. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part II. LNCS, vol. 6509, pp. 72–86. Springer, Heidelberg (2010)

9. Kranakis, E., Krizanc, D., Morales, O.: Maintaining Connectivity in Sensor Networks Using Directional Antennae. In: Nikoletseas, S., Rolim, J. (eds.) Theoretical Aspects of Distributed Computing in Sensor Networks, Part 2, pp. 59–84. Springer, Heidelberg (2011), ISBN 978-3-642-14848-4

10. van Nijnatten, F.: Range Assignment with Directional Antennas. Master's Thesis. Technische Universiteit Eindhoven (2008)

11. Wu, W., Du, H., Jia, X., Li, Y., Huang, S.C.-H.: Minimum connected dominating sets and maximal independent sets in unit disk graphs. Theoretical Computer Science 352, 1–7 (2006)

Frequency Capping in Online Advertising
(Extended Abstract)

Niv Buchbinder[1], Moran Feldman[2,*],
Arpita Ghosh[3], and Joseph (Seffi) Naor[2,**]

[1] Open University, Israel
niv.buchbinder@gmail.com
[2] Computer Science Dept., Technion, Haifa, Israel
{moranfe,naor}@cs.technion.ac.il
[3] Yahoo! Research, Santa Clara, CA
arpita@yahoo-inc.com

Abstract. We study the following online problem. Each advertiser a_i has a value v_i, demand d_i, and *frequency cap* f_i. Supply units arrive online, each one associated with a user. Each advertiser can be assigned at most d_i units in all, and at most f_i units from the same user. The goal is to design an online allocation algorithm maximizing total value.

We first show a deterministic upper bound of 3/4-competitiveness, even when all frequency caps are 1, and all advertisers share identical values and demands. A competitive ratio approaching $1 - 1/e$ can be achieved via a reduction to a model with arbitrary decreasing valuations [GM07]. Our main contribution is analyzing two 3/4-competitive greedy algorithms for the cases of equal values, and arbitrary valuations with equal demands. Finally, we give a primal-dual algorithm which may serve as a good starting point for improving upon the $1 - 1/e$ ratio.

1 Introduction

Display advertising, consisting of graphic or text-based ads embedded in web-pages, constitutes a large portion of the revenue from Internet advertising, totaling billions of dollars in 2008. Display, or brand, advertising is typically sold by publishers or ad networks on a pay-per-impression basis, with the advertiser specifying the total number of impressions she wants (the demand) and the price she is willing to pay per impression.[1]

Since display ads are sold on a pay-per-impression rather than on a pay-per-click or pay-per-action basis, *effective delivery* of display ads is very important to maximize advertiser value — each impression that an advertiser pays for must

---

[1] In contrast, sponsored search advertisers typically pay per click or per action, and usually have budgets, rather than demands, or quotas, on the number of impressions.

be shown to *as valuable a user* as possible. One aspect of effectively delivering display ads, which has been widely studied, is good targeting — matching ads to users who are likely to be responsive to the content of the ad. Another very important, but less studied, aspect is *limiting user exposure* to an ad - displaying the same ad to a user multiple times diminishes value to the advertiser, since the incremental benefit from repeatedly displaying the same ad to a user is likely to be small (a user is unlikely to react to an ad after he has seen it a few times).

The notion of limiting the number of times a user is exposed to a particular ad is called *frequency capping* [19], and is often cited as a way to avoid banner ad burnout. That is, frequency capping prevents ads from being displayed repeatedly to the point where visitors are being overexposed and response drops.[2] Serving frequency capped ads is a very real requirement to maximize value delivered to display advertisers, particularly in the pay-per-impression structure of the display advertising marketplace. This is recognized by a number of publishers and ad networks (for instance, RightMedia, Google and Yahoo!) who already offer or implicitly implement frequency capping for their display advertisers.

Serving display ads subject to a frequency capping constraint poses an *online assignment* problem since the supply of users, or impressions, is not known to the ad server in advance. How should the ad server allocate impressions to advertisers in this setting? In this paper, we study the simplest abstractions of the assignment problems motivated by frequency capping.

Problem Statement. There are n advertisers. Advertiser i has value per impression v_i, which is the price she is willing to pay for an impression, and a demand d_i, which is the maximum number of impressions she is interested in. In addition, she also has a *frequency cap* f_i, which is the maximum number of times her ad can be displayed to the same user. That is, an advertiser pays v_i only for impressions from users who have not seen her ad more than f_i times. The set of advertisers, and their parameters, is known to the ad server in advance.

Impressions from users arrive online. We say an advertiser is *eligible* for an impression if she still has leftover demand, and has not yet exhausted her frequency cap for the user associated with this impression. When an impression arrives, the ad server must immediately decide which ad, among the set of eligible advertisers, to display for that impression. The total revenue obtained by an algorithm is the sum of the revenues from each impression it allocates. We want to design algorithms that are competitive against the optimal offline allocation, which knows the supply of impressions (with their associated users) in advance. (This problem is captured by the model in [14], see §1.1.)

In the absence of the frequency capping constraint ($f_i = \infty$), the natural greedy algorithm, which assigns each arriving impression to the advertiser with the highest per-impression value v_i, is optimal. However, with the frequency capping constraint, the ad server faces a tradeoff between assigning an arriving impression to an advertiser with high v_i but large frequency cap (since the

---

[2] While it might be argued that multiple displays of an ad to a user reinforces its message, repeated display without an upper limit clearly diminishes value.

supply can stop anytime) and a lower value advertiser with a smaller frequency cap (since small f_i means this advertiser needs to be assigned to many distinct users). In fact, even when all advertisers have identical values (with arbitrary tie breaking), the greedy algorithm is not optimal, as the following example shows: there are two advertisers, the first with $v_1 = 1, f_1 = n$, and the second with $v_2 = 1 - \epsilon$ and $f_2 = 1$; both advertisers have demand n (the $1 - \epsilon$ is used for tie breaking). The sequence of users is $u_1, \ldots, u_n, u_{n+1}, \ldots, u_{n+1}$, where the last user appears n times (n impressions). The greedy allocation gets a value of $n+1$, whereas the optimal offline allocation gets $2n$.

As the next example shows, however, it is not even the different frequency caps that lead to the suboptimality of the greedy algorithm: suppose there are $n + 1$ advertisers each with $f_i = 1$. The first n advertisers have value 1 and demand 1, and the last advertiser has value $1 - \epsilon$ and demand n. With the same arrival sequence of users, a greedy allocation, again, gets a value of $n+1$, whereas the optimal value is $2n$. In fact, as we will show in §3, even when *all* values and demands are equal and *all* frequency caps are 1, no deterministic algorithm can have a competitive ratio better than $3/4$.

Distinction from Online Matching. Finding a matching in a bipartite graph where one side is known and the other side is exposed one vertex at a time is known as *online matching*. While the problem of online allocation with frequency capping constraints appears to be similar to online matching, they are actually quite different. In the frequency capping problem, a-priori each impression can be assigned to any of the advertisers. Now, as the impressions arrive, in the language of online matching, the existence of an edge between an advertiser and an arriving impression *depends* on the previous assignments made by the algorithm because of the frequency capping constraint. Specifically, if the algorithm has already assigned enough impressions from user j to advertiser i, or has exhausted i's demand, there is no edge between advertiser i and a newly arrived impression; otherwise, there is an edge. This means that an adversary can no longer control the set of edges hitting each new impression; instead, the online algorithm determines the set of edges using indirect means. While we expect this property to translate into better competitive ratios for the frequency capping problem, taking advantage of the difference is not easy, a fact which is demonstrated by the involved analysis for the natural greedy algorithm for the problem.

Results. Our online assignment problem can be stated abstractly as follows: There are n agents, each with a total demand d_i, and a value v_i for items. Items of different types arrive one by one in an online fashion and must be allocated to an agent immediately. Agent i wants at most f_i copies of any single type of item. How should an online algorithm assign each arriving item to agents to maximize value? This abstract statement suggests the following simpler questions.

- *Equal values, arbitrary d_i, f_i:* Suppose agents (advertisers) have identical values for items (impressions), that is, $v_i = 1$ for all i. Now, the goal of the online algorithm is simply to assign as many items as possible. Our main technical contribution is the analysis of a novel greedy algorithm, proving

that it is 3/4-competitive; this is optimal for a deterministic algorithm. The first step is to show that we can assume without loss of generality that every advertiser has frequency cap 1, i.e., wants no more than one impression from each user (the reduction is independent of advertisers having the same value, and also applies when advertisers have arbitrary values). This reduction is simple, yet crucial — for each of the cases we study, designing algorithms directly, with arbitrary frequency caps, turns out to be rather hard.

We then analyze our greedy algorithm, which assigns arriving impressions in decreasing order of *total* demand amongst eligible advertisers, for instances with unit frequency cap. (Assigning greedily according to maximum *residual demand* does not work; this algorithm cannot do better than 2/3.) The unit frequency cap means that an advertiser is eligible for an impression if she has leftover demand and has not yet been assigned to this user. We first prove that any *non-lazy* algorithm has competitive ratio 3/4 when all demands are equal (in addition to the equal value); then we build on this analysis to account for the fact that advertisers have unequal demands.

Combinatorial analysis of online algorithms is usually done via a potential function argument which shows that at each step, the change in the potential function plus the algorithm's revenue are comparable to the gain of the optimal solution. Surprisingly, our analysis considers only the final assignment, disregarding the way in which it is reached. This allows us to avoid coming up with a potential function (which in many cases seems to come "out of nowhere"), and skip the tedious consideration of each possible step.

Our result is especially interesting in light of the known upper bounds for unweighted online matching: 0.5 and $1 - 1/e \approx 0.63$ for deterministic and randomized algorithms, respectively [16].

- *Arbitrary values, equal d_i/f_i*: The ideas used in the analysis of the equal values case can be extended to analyze the case where advertisers have different values, but the same ratio of demand to frequency cap. We show here that the natural greedy algorithm, which assigns in decreasing order of value, has a competitive ratio of 3/4; again, this is optimal in the sense that no deterministic algorithm can do better.

- *Arbitrary values, d_i and f_i, with targeting*: Finally, for the general case with arbitrary values, demands and frequency caps, we design a primal-dual algorithm whose competitive ratio approaches $1 - 1/e \approx 0.63$ when $d_i/f_i \gg 1$[3]; we also show an upper bound of $1/\sqrt{2}$ for this case. Our online primal-dual algorithm has an interesting feature: it both increases and decreases primal variables during the execution of the algorithm. The same algorithm and competitive ratio also apply when advertisers have *target sets*, i.e., they have value v_i for impressions from a set S_i of users, and value of 0 for other impressions. For this case, we have a matching upper bound for deterministic online algorithms, using the upper bound on online b-matching [15]. (See §1.1 for a discussion regarding [14] and online primal dual algorithms.)

---

[3] The competitive ratio of $1 - 1/e$ in [14] is under an assumption similar to ours.

1.1   Related Work

Maximizing revenue in online ad auctions has received much attention in recent years [8,7,18,17,6,11,12]. The problem of designing online algorithms to maximize advertising revenue was introduced in the *adwords model* [18]: advertisers have budgets, and bids for different keywords. Keywords arrive online, and the goal is to match advertisers to keywords to maximize revenue, while respecting the advertisers' budget constraints. Goel and Mehta [14] extend the adwords model, allowing advertisers to specify bids for keywords which are decreasing functions of the number of impressions (of the keyword) already assigned to the advertiser. Our frequency capping problem is, in fact, a special case of the model of [14] (but not of the adwords model of [18]), where keywords correspond to users, and the decreasing function takes the form of a step function with cutoff f_i. Hence, the $(1-1/e)$-competitive online algorithm of [14] applies to our problem as well. On the other hand, the upper bounds in [14] do not apply to our problem since the model of [14] also captures online matching. Improving upon the ratio of $1 - 1/e$ in special cases is posed as an open problem in [14].

Our greedy algorithms in §3 and §4 obtain a ratio of $3/4$, improving upon this ratio of $1 - 1/e$. While the competitive ratio of our algorithm in §5 is the same as that in [14], the algorithms are quite different. Moreover, our model does not inherit the upper bound of $1 - 1/e^4$, and in fact, the best upper bound for the case without target sets is $1/\sqrt{2}$. Also, while the most general problem we solve in this paper remains within the model of [14], the most general and realistic version of the frequency capping problem (§6) cannot be stated as a special case of the model of [14]. For this model the question of both a competitive algorithm and an upper bound (tighter than $1 - 1/e$) are open.

The primal dual framework for online problems, first introduced by Buchbinder and Naor [9], has been shown to be useful in many online scenarios including ad auctions, see [4,5,3,2,10,11]. Unlike these primal-dual algorithms (e.g., [9,11]), which simply update the primal variables monotonically in each round, our primal-dual algorithm is novel in that it reassigns primal variables several times during the execution of the algorithm; hence, the primal variables do not necessarily increase monotonically with each round of new supply.

Mirrokni et al. [13] consider frequency capping in a stochastic model, but they leave open the question of improving upon the $1 - 1/e$ ratio in this model. Finally, the work in [1] also addresses user fatigue in the context of sponsored search; however, the model and algorithms substantially differ from ours.

2   Preliminaries

We denote by $A(\sigma)$ the revenue of algorithm A on a sequence σ of arrivals of impressions, and by $OPT(\sigma)$ the revenue of the optimal offline algorithm, which

---

[4] Since the model of [14] captures the adwords model of [18], it inherits an upper bound of $1 - 1/e$ on the competitive factor. The frequency capping problem does not generalize the adwords model, and therefore, does not inherit this upper bound.

knows σ in advance. The goal is to design an online algorithm A that assigns each impression immediately upon arrival, and produce a feasible allocation whose total value $A(\sigma)$ is competitive against $OPT(\sigma)$ for any σ. The *natural greedy algorithm* for the problem, denoted by $GREEDY_V$, allocates each arriving impression to the eligible advertiser with the highest value (breaking ties arbitrarily, but consistently). The examples in the introduction show that the greedy algorithm at most $1/2$-competitive. The next theorem shows that this is tight, due to space limitations, its proof is deferred to a full version of this paper.

Theorem 1. *The competitive ratio of $GREEDY_V$ is $1/2$.*

We now establish a reduction from general frequency caps to unit frequency caps which greatly simplifies our algorithms. The following theorem allows us to assume $f_i = 1$ in the rest of the paper, its proof is also deferred to a full version.

Theorem 2 (Reduction to Unit Frequency Cap). *For every frequency capping instance there is an equivalent instance where all frequency caps are 1. Moreover, any solution to the equivalent instance can be transformed in an online fashion to an equivalent solution in the original model.*

3   Identical Valuations

In this section, we assume all advertisers have identical valuations, i.e., for each advertiser a_i, w.l.o.g., $v_i = 1$. The following theorem gives an upper bound on any deterministic online algorithm; due to space limitations, its proof is deferred to a full version of this paper.

Theorem 3. *No deterministic online algorithm is better than $3/4$-competitive, even if all advertisers have identical values, demands, and frequency caps.*

We now turn to online algorithms. A natural greedy algorithm is one that assigns an arriving impression to an eligible advertiser with the maximum residual demand. However, assigning according to residual demand, breaking ties arbitrarily, cannot have a competitive ratio better than $2/3$, as the following example shows. There are two advertisers, with $d_1 = 1$ and $d_2 = 2$, with ties broken in favor of a_1. The sequence of arrivals is u_2, u_1, u_2. The residual demand algorithm allocates only two impressions: the first impression to a_2 and then the second impression to a_1. The optimal assignment, however, can assign all 3 impressions.

We show that an alternative greedy algorithm, $GREEDY_D$, which assigns according to *total* demand, is $3/4$-competitive. Hereby is algorithm $GREEDY_D$:

1. Sort advertisers $a_1, \ldots, a_n$ in a non-decreasing demand order ($d_1 \geq \ldots \geq d_n$).
2. Assign an arriving impression to the first eligible advertiser in this order.

We need the following notation. Let y_i denote the number of impressions assigned by $GREEDY_D$ to a_i, and let $y^* = \min_i y_i$. Let k denote the number of advertisers whose demand is exhausted by $GREEDY_D$. In §3.1, we analyze the case of equal demands, and in §3.2 we build on this analysis to deal with the case where demands are arbitrary. We include the proof of the equal demands case since it is simpler, yet gives some insight into the proof of the general case.

3.1 Equal Demand Case

Algorithm $GREEDY_D$ is *non-lazy*, i.e., it allocates every impression it receives, unless no advertiser is eligible for it. We show that any non-lazy algorithm, including $GREEDY_D$, is 3/4-competitive if all advertisers have equal demand d.

Theorem 4. *Let ALG be a non-lazy algorithm, and let σ be a sequence of impressions. Then, $ALG(\sigma)/OPT(\sigma) \geq 3/4$.*

Before going into the proof of Theorem 4, consider the example depicted in Figure 1. The rectangle is divided into three areas: R_1 is the total allocation of advertisers who have exhausted their demand, R_2 is the total allocation of advertisers who have not exhausted their demand, and R_3 is "unused" demand. We use two bounds on $|OPT(\sigma)| - |ALG(\sigma)|$: $|R_3| \leq (d - y^*) \cdot (n - k) \leq |R_2| \cdot (d - y^*)/y^*$, and $k \cdot y^* \leq |R_1| \cdot y^*/d$ (note that $y^* > 0$ since an advertiser who has received no impressions can always be assigned at least one impression without violating the frequency cap constraint). The theorem follows from these bounds, and the observation $|ALG(\sigma)| = |R_1| + |R_2|$.

Let A be the set of impressions allocated by OPT, and let $B \subseteq A$ be of size $OPT(\sigma) - ALG(\sigma)$. Associate each impression of B with an advertiser, such that up to $d - y_i$ impressions of B are associated with each advertiser a_i. This is possible since $\sum_{i=1}^{n}(d - y_i) = nd - ALG(\sigma) \geq OPT(\sigma) - ALG(\sigma) = |B|$.

Lemma 1. $|B| = OPT(\sigma) - ALG(\sigma) \leq y^*k$.

Proof. Let a_{i^*} be an advertiser for which $y_{i^*} = y^*$. If $y^* = d$, then $ALG(\sigma) = nd = OPT(\sigma)$, so we can assume $y^* < d$. Thus, each impression ALG fails to allocate belongs to a user already having an impression allocated to a_{i^*} (else ALG could have assigned it to a_{i^*}). Hence, there are at most y^* users having unallocated impressions. Each such user u has at most k more impressions allocated by OPT than by ALG (if u has an unassigned impression, all $n - k$ advertisers with non-exhausted demands must have been assigned an impression of u).

We define two types of payments received by each impression $x \in B$. Suppose impression x is associated with advertiser a_i. The first payment x gets is $p_x = y_i/(d - y_i)$, and the second payment is $p'_x = d/y^*$.
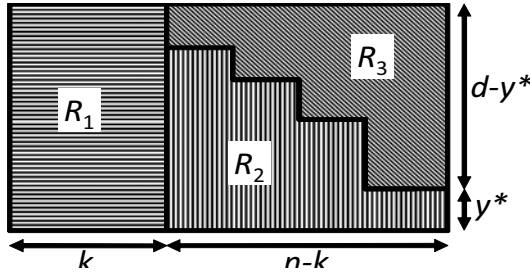
Fig. 1. Each column is an advertiser and each row corresponds to a unit demand

Lemma 2. *The total payment received by all impressions of B is at most $ALG(\sigma)$.*

Proof. Let E denote the set of advertisers whose demand is not exhausted by ALG (i.e., $|E| = n - k$). Let $a_i \in E$. For each impression x associated with a_i, we have $p_x = y_i/(d - y_i)$ and the number of such impressions is at most $d - y_i$. Therefore, the first type of payment received by impressions associated with a_i sums up to at most y_i. Adding up over all advertisers of E, the sum of the first type payments to all impressions in B is at most $\sum_{a_i \in E} y_i$. Since payments of the second type all equal values, they add up to $|B| \cdot \frac{d}{y^*} \leq y^* k \cdot \frac{d}{y^*} = dk$. Note that $dk + \sum_{a_i \in E} y_i = ALG(\sigma)$, since $a_i \notin E \Rightarrow y_i = d$, completing the proof.

Lemma 3. *For each impression $x \in B$, $p_x + p'_x \geq 3$.*

Proof. Suppose x is associated with an advertiser a_i. The total payment received by x is: $\frac{y_i}{d - y_i} + \frac{d}{y^*} \geq \frac{y^*}{d - y^*} + \frac{d}{y^*} = \frac{y^{*2} + d(d - y^*)}{y^*(d - y^*)} = 3 + \frac{(2y^* - d)^2}{y^*(d - y^*)} \geq 3$.

Corollary 1. $ALG(\sigma) \geq 3|B|$.

The proof of Theorem 4 is now immediate:

$$\frac{ALG(\sigma)}{OPT(\sigma)} = \frac{ALG(\sigma)}{ALG(\sigma) + |B|} \geq \frac{3|B|}{3|B| + |B|} = \frac{3}{4} \; . \tag{1}$$

3.2   General Case

In this section we prove the main result of our paper. Unfortunately, the proof from the previous section does not readily generalize; the core of the difficulty is that it is no longer possible to sort the advertisers in non-decreasing demand order such that all exhausted advertisers appear before the non-exhausted advertisers. Instead, exhausted and non-exhausted advertisers might be interleaved in every non-decreasing demand ordering of the advertisers. Thus, it is hard to guarantee the extent to which impressions of exhausted advertisers can be charged. A simple approach to overcome this difficulty is to split the advertisers into blocks, making sure that within each block the exhausted advertisers appear before the non-exhausted ones. However, this fails since OPT and $GREEDY_D$ may place impressions in different blocks. To circumvent this problem we consider subsets of advertisers having demand above a given threshold. The proof then makes a connection between the difference in number of impressions allocated by OPT and $GREEDY_D$ to a subset of the advertisers and the number of exhausted advertisers in the subset, yielding a lower bound on the payment that can be extracted from the impressions of the exhausted advertisers.

The next theorem shows that $GREEDY_D$ is 3/4-competitive also for arbitrary demands; due to space limitations, its proof is deferred to a full version.

Theorem 5. *For any sequence σ of input impressions, $\frac{GREEDY_D(\sigma)}{OPT(\sigma)} \geq 3/4$.*

4   Equal Demands/Arbitrary Valuations

In this section, we assume advertisers have different values, but equal ratio of demand to frequency cap (this can happen, e.g., when each advertiser has frequency cap f_i and wants to advertise to the same number of distinct users u, i.e., $d_i = f_i u$). The reduction to unit frequency caps makes this equivalent to assuming all demands are equal and all frequency caps are 1. The following theorem shows that the natural greedy algorithm $GREEDY_V$, assigning in decreasing order of value, is 3/4-competitive. Note that by Theorem 3, this ratio is optimal.

Theorem 6. *For any sequence σ of input impressions, $\frac{GREEDY_V(\sigma)}{OPT(\sigma)} \geq 3/4$, under the above assumptions.*

The proof builds on the ideas developed in Theorem 5, and due to lack of space, it is deferred to a full version of this paper.

5   Arbitrary Valuations

We now consider arbitrary valuations v_i. We first prove an improved upper bound for this case. Due to space imitations, the proofs of the theorems of this section are deferred to a full version of this paper.

Theorem 7. *No deterministic algorithm is better than $1/\sqrt{2} \approx 0.707$-competitive.*

5.1   A Primal-Dual Algorithm

In order to apply the primal-dual approach to the problem, we first formulate the **offline** allocation problem as a linear program as following. Let A be the set of advertisers. Let B be the set of users. Finally, for each user $j \in B$, let $K(j)$ be the number of impressions of user j. We define variables $y(i, j, k)$ indicating that the k-th impression of user j is assigned to advertiser a_i.

$$\max \sum_{a_i \in A} v_i \sum_{j \in B} \sum_{k=1}^{K(j)} y(i, j, k) \qquad (D)$$

$$\text{s.t.} \sum_{j \in B} \sum_{k=1}^{K(j)} y(i, j, k) \leq d_i \quad \forall a_i \in A$$

$$\sum_{k=1}^{K(j)} y(i, j, k) \leq f_i \qquad \forall a_i \in A, j \in B$$

$$\sum_{a_i \in A} y(i, j, k) \leq 1 \qquad \forall j \in B, k \in \{1, 2 \ldots, K(j)\}$$

$$y(i, j, k) \geq 0$$

The first set of constraints guarantees that at most d_i impressions are assigned to advertiser a_i. The second set of constraints guarantees the frequency cap of each

advertiser. Finally, the last set of constraints guarantees that each impression is assigned only once. For consistency with previous work [9], we refer to the maximization problem as the dual problem. We now define the primal problem. We have variable $x(i)$ for each advertiser a_i, a variable $w(i,j)$ for each pair of advertiser a_i and user j and variable $z(j,k)$ for the k-th impression of user j.

$$\min \sum_{a_i \in A} d_i x(i) + \sum_{a_i \in A, j \in B} f_i w(i,j) + \sum_{j \in B, k} z(j,k) \qquad (P)$$
$$\text{s.t.} \quad x(i) + w(i,j) + z(j,k) \geq v_i \; \forall a_i \in A, j \in B, k$$
$$x, w, z \geq 0$$

The allocation algorithm is as follows. We assume that the reduction to the case where the frequency cap of each advertiser is 1 has already been applied.

Allocation Algorithm: Upon arrival of impression k of user j:

- Let $S(j)$ be those advertisers not yet assigned impressions of user j, and let $\overline{S(j)} = A \setminus S(j)$.
- Let $m_1 \in S(j)$ be the advertiser that maximizes $v_i - x(i)$. Let $m_2 \in S(j) \setminus m_1$ be the advertiser that maximizes $v_i - x(i)$.[a]

1. Assign impression k to advertiser m_1.
2. For each advertiser $i \in \overline{S(j)} \cup m_1$ set: $w(i,j) \leftarrow \max\{0, (v_i - x(i)) - (v_{m_2} - x(m_2))\}$.
3. For each advertiser $i \in S(j) \setminus m_1$ set: $w(i,j) \leftarrow 0$.
4. For each impression $\ell \leq k$ of user j set: $z(j,\ell) \leftarrow v_{m_2} - x(m_2)$.
5. For advertiser m_1: $x(m_1) \leftarrow x(m_1)\left(1 + \frac{1}{d_i}\right) + \frac{v_{m_1}}{c \cdot d_i}$ (c is a constant to be determined later).

---
[a] If $\max_{S(j)}(v_i - x(i)) \leq 0$, or $S(j) = \emptyset$, no assignment is made and no variables are updated. If there is no m_2, we view $v_{m_2} - x(m_2)$ as equal to 0.

Notice that this algorithm differs from the standard online primal-dual approach because it both increases and decreases primal variables.

Theorem 8. *The algorithm is $(1 - (c+1)^{-1})$-competitive, for $c = (1 + \frac{1}{d_{\min}})^{d_{\min}} - 1$, where $d_{\min}$ is the minimum demand of any advertiser.*

Targeting constraints. We assumed thus far that advertisers valued all users equally. In practice, however, when buying display ad space, advertisers can provide *targeting* information, specifying which subset of impressions is acceptable. That is, advertisers have value v_i for acceptable impressions that meet the targeting constraints and value of 0 for others (contracts for display ads typically specify a single price-per-impression that does not vary across the set of acceptable impressions, i.e., v_i does not take on different non-zero values).

Suppose targeting information is user-dependent only, i.e., an advertiser may value only a subset of users with certain characteristics (age, gender, location, etc.), but does not distinguish between different impressions (e.g., when visiting

different webpages) from the same user. In this case, advertiser values have the following form: $v(i,j)$ is either v_i or 0 (i.e., a_i finds users with $v(i,j) = v_i$ *acceptable*, and the rest unacceptable). We observe that the above algorithm also works for this more general setting. The only change is that the sets $S(j)$ and $\overline{S(j)}$ include only advertisers that accept user j. This implies the following.

Theorem 9. *For $c = (1 + \frac{1}{d_{\min}})^{d_{\min}} - 1$, the algorithm remains $(1 - (c+1)^{-1})$-competitive, when $v(i,j) \in \{0, v_i\}$ for all i, j.*

Theorem 10. *With targeting constraints, no deterministic algorithm has a competitive ratio higher than $1 - 1/e$, even when demand are large.*

6   Further Directions

The frequency capping problem is an important practical problem which imposes interesting algorithmic challenges. Here are two main directions for further work.

– *Improving $1 - 1/e$ for arbitrary valuations:* There is a gap between the best upper bound of $1/\sqrt{2}$ and the best algorithm $(1 - 1/e)$ for the case of arbitrary valuations without targeting constraints, discussed in §5. The targeting constraints are to be blamed for the "matching" aspects, leading to the upper bound of $1 - 1/e$ in Theorem 10. By removing these constraints, the difference between our problem and online matching resurfaces, and the upper bound of $1 - 1/e$ does not hold anymore. We believe that our primal-dual algorithm is an excellent starting point for a future online algorithm for frequency capping with arbitrary values that will go beyond $1 - 1/e$.
– *Content-based targeting specifications:* Targeting specifications may be not only user-based, but also depend on the webpage's content. For instance, an advertiser might want to display her ads only to males (user targeting) when they browse a sports related webpage (content targeting); targeting constraints are often of this form. So, advertisers now have valuations of the form $v(i,j,k) \in \{0, v_i\}$, i.e., the value of the k-th impression of the j-th user to advertiser i is either v_i or 0 depending on what page the user was surfing on his k-th impression. Note that the model of [14] does not capture this problem, which entangles a matching aspect with frequency capping. The questions of designing a good online algorithm and finding the smallest upper bound (of course, $1 - 1/e$ is a trivial upper bound since this problem generalizes arbitrary valuations with targeting) are both open.

References

1. Abrams, Z., Vee, E.: Personalized ad delivery when ads fatigue: An approximation algorithm. In: 3rd International Workshop on Internet and Network Economics, pp. 535–540. Springer, Heidelberg (2007)

2. Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., Naor, J.: A general approach to online network optimization problems. ACM Transactions on Algorithms 2(4), 640–660 (2006)
3. Alon, N., Awerbuch, B., Azar, Y., Buchbinder, N., Naor, J.: The online set cover problem. SIAM J. Comput. 39(2), 361–370 (2009)
4. Bansal, N., Buchbinder, N., Naor, J.: A primal-dual randomized algorithm for weighted paging. In: 48th Annual IEEE Symposium on Foundations of Computer Science, pp. 507–517. IEEE Computer Society, Washington, DC (2007)
5. Bansal, N., Buchbinder, N., Naor, J.: Randomized competitive algorithms for generalized caching. In: 40th ACM Symposium on Theory of Computer Science, pp. 235–244. ACM, New York (2008)
6. Bansal, N., Chen, N., Cherniavsky, N., Rudra, A., Schieber, B., Sviridenko, M.: Dynamic pricing for impatient bidders. In: 18th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 726–735. Society for Industrial and Applied Mathematics, Philadelphia (2007)
7. Blum, A., Hartline, J.: Near-optimal online auctions. In: 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1156–1163. Society for Industrial and Applied Mathematics, Philadelphia (2005)
8. Blum, A., Kumar, V., Rudra, A., Wu, F.: Online learning in online auctions. Theor. Comput. Sci. 324(2-3), 137–146 (2004)
9. Buchbinder, N., Naor, J.: Online primal-dual algorithms for covering and packing. Math. Oper. Res. 34(2), 270–286 (2009)
10. Buchbinder, N., Naor, J.: Improved bounds for online routing and packing via a primal-dual approach. In: 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 293–304. IEEE Computer Society, Washington, DC (2006)
11. Buchbinder, N., Jain, K., Naor, J(S.): Online primal-dual algorithms for maximizing ad-auctions revenue. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 253–264. Springer, Heidelberg (2007)
12. Feldman, J., Korula, N., Mirrokni, V., Muthukrishnan, S., Pál, M.: Online ad assignment with free disposal. In: Leonardi, S. (ed.) WINE 2009. LNCS, vol. 5929, pp. 374–385. Springer, Heidelberg (2009)
13. Feldman, J., Mehta, A., Mirrokni, V.S., Muthukrishnan, S.: Online stochastic matching: Beating $1 - 1/e$. In: 50th Annual IEEE Symposium on Foundations of Computer Science, pp. 117–126. IEEE Computer Society, Washington, DC (2009)
14. Goel, G., Mehta, A.: Adwords auctions with decreasing valuation bids. In: Deng, X., Graham, F.C. (eds.) WINE 2007. LNCS, vol. 4858, pp. 335–340. Springer, Heidelberg (2007)
15. Kalyanasundaram, B., Pruhs, K.R.: An optimal deterministic algorithm for online b-matching. Theor. Comput. Sci. 233(1-2), 319–325 (2000)
16. Karp, R.M., Vazirani, U.V., Vazirani, V.V.: An optimal algorithm for on-line bipartite matching. In: 22nd Annual ACM Symposium on Theory of Computing, pp. 352–358. ACM, New York (1990)
17. Mahdian, M., Saberi, A.: Multi-unit auctions with unknown supply. In: 7th ACM Conference on Electronic Commerce, pp. 243–249. ACM, New York (2006)
18. Mehta, A., Saberi, A., Vazirani, U., Vazirani, V.: Adwords and generalized online matching. J. ACM 54(5), 22 (2007)
19. Marketing Terms.com,
   `http://www.marketingterms.com/dictionary/frequency_cap`

Adjacency-Preserving Spatial Treemaps

Kevin Buchin[1], David Eppstein[2], Maarten Löffler[2],
Martin Nöllenburg[3], and Rodrigo I. Silveira[4]

[1] Dept. of Mathematics and Computer Science, TU Eindhoven
[2] Dept. of Computer Science, University of California, Irvine
[3] Institute of Theoretical Informatics, Karlsruhe Institute of Technology
[4] Dept. de Matemàtica Aplicada II, Universitat Politècnica de Catalunya

Abstract. Rectangular layouts, subdivisions of an outer rectangle into smaller rectangles, have many applications in visualizing spatial information, for instance in rectangular cartograms in which the rectangles represent geographic or political regions. A *spatial treemap* is a rectangular layout with a hierarchical structure: the outer rectangle is subdivided into rectangles that are in turn subdivided into smaller rectangles. We describe algorithms for transforming a rectangular layout that does not have this hierarchical structure, together with a clustering of the rectangles of the layout, into a spatial treemap that respects the clustering and also respects to the extent possible the adjacencies of the input layout.

1 Introduction

Spatial treemaps are an effective technique to visualize two-dimensional hierarchical information. They display hierarchical data by using nested rectangles in a space-filling layout. Each rectangle represents a geometric or geographic region, which in turn can be subdivided recursively into smaller regions. On lower levels of the recursion, rectangles can also be subdivided based on non-spatial attributes. Typically, at the lowest level some attribute of interest of the region is summarized by using properties like area or color. Treemaps were originally proposed to represent one-dimensional information in two dimensions [14]. However, they are well suited to represent spatial—two-dimensional—data because the containment metaphor of the nested rectangles has a natural geographic meaning, and two-dimensional data makes an efficient use of space [18].

Spatial treemaps are closely related to rectangular cartograms [13]: distorted maps where each region is represented by a rectangle whose area corresponds to a numerical attribute such as population. Rectangular cartograms can be seen as spatial treemaps with only one level; multi-level spatial treemaps in which every rectangle corresponds to a region are also known as *rectangular hierarchical cartograms* [15, 16]. Spatial treemaps and rectangular cartograms have in common that it is essential to preserve the recognizability of the regions shown [17]. Most previous work on spatial treemaps reflects this by focusing on the preservation of distances between the rectangular regions and their geographic counterparts (that is, they minimize the displacement of the regions). However, often small displacement does not imply recognizability (swapping the position of two small neighboring countries can result in small displacement, but

a big loss of recognizability). In the case of cartograms, most emphasis has been put on preserving adjacencies between the geographic regions. It has also been shown that while preserving the topology it is possible to keep the displacement error small [4, 17].

In this paper we are interested in constructing high-quality spatial treemaps by prioritizing the preservation of topology, following a principle already used for rectangular cartograms. Previous work on treemaps has recognized that preserving neighborhood relationships and relative positions between the regions were important criteria [8, 12, 18], but we are not aware of treemap algorithms that put the emphasis on preserving topology.

The importance of preserving adjacencies in spatial treemaps can be appreciated by viewing a concrete example. Figure 1, from [15], shows a spatial treemap of property transactions in London between 2000 and 2008, with two levels formed by the boroughs and wards of London and colors representing average prices. To see whether housing prices of neighboring wards are correlated, it is important to preserve adjacencies: otherwise it is easy to draw incorrect conclusions, like seeing clusters that do not actually exist, or missing existing ones.
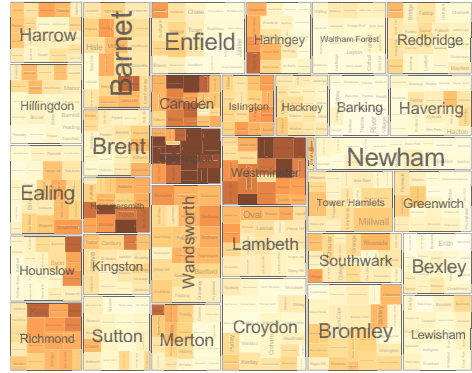
Fig. 1. A 2-level spatial treemap from [15]; used with permission

Preserving topology in spatial treemaps poses different challenges than in (non-hierarchical) rectangular cartograms. Topology-preserving rectangular cartograms exist under very mild conditions and can be constructed efficiently [4, 17]. As we show in this paper, this is not the case when a hierarchy is added to the picture.

In this paper we consider the following setting: the input is a hierarchical rectangular subdivision with two levels. We consider only two levels due to the complexity of the general m-level case. However, the two-level case is interesting on its own, and applications that use only two-level data have recently appeared [15].

Furthermore, we adopt a 2-phase approach for building spatial treemaps. In the first phase, a base rectangular cartogram is produced from the original geographic regions. This can be done with one of the many algorithms for rectangular cartograms [4]. The result will contain all the bottom-level regions as rectangles, but the top-level regions will not be rectangular yet, thus will not represent the hierarchical structure. In the second phase, we convert the base cartogram into a treemap by making the top-level regions rectangles. It is at this stage that we intend to preserve the topology of the base cartogram as much as possible, and where our algorithms come in. See Figure 2 for an example.

The advantage of this 2-phase approach is that it allows for customization and user interaction. Interactive exploration of the data is essential when visualizing large amounts of data. The freedom to use an arbitrary rectangular layout algorithm in the first phase of the construction allows the user to prioritize the adjacencies that he or she considers

(a)   (b)
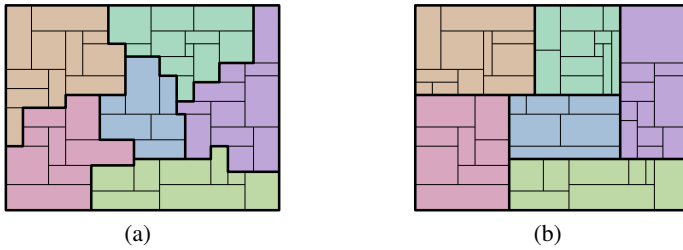
Fig. 2. (a) An example input: a full layout of the bottom level, but the regions at a higher level in the hierarchy are not rectangles. (b) The desired output: another layout, in which as many lower-level adjacencies as possible have been kept while reshaping the regions at a higher level into rectangles.

most essential. In the second phase, our algorithm will produce a treemap that will try to preserve as many as the adjacencies in the base cartogram as possible.

In addition, we go one step further and consider preserving the *orientations* of the adjacencies in the base cartogram (that is, whether two neighboring regions share a vertical or horizontal edge, and which one is on which side). This additional constraint is justified by the fact that the regions represent geographic or political regions, and relative positions between regions are an important factor when visualizing this type of data [4, 17]. The preservation of orientations has been studied for cartograms [5], but to our knowledge, this is the first time they are considered for spatial treemaps.

We can distinguish three types of adjacency-relations: (i) top-level adjacencies, (ii) internal bottom-level adjacencies (adjacencies between two rectangles that belong to the same top-level region), and (iii) external bottom-level adjacencies (adjacencies between two rectangles that belong to different top-level regions). As we argue in the next section, we can always preserve all adjacencies of types (i) and (ii) under a mild assumption, hence the objective of our algorithms is to construct treemaps that preserve as many adjacencies of type (iii) as possible. We consider several variants of the problem, based on whether the orientations of the adjacencies have to be preserved, and whether the top-level layout is given in advance. In order to give efficient algorithms, we restrict ourselves to top-level regions that are orthogonally convex. This is a technical limitation that seems difficult to overcome, but that we expect does not limit the applicability of our results too much: our algorithms should still be useful for many practical instances, for example, by subdividing non-convex regions into few convex pieces.

Results. In the most constrained case in which adjacencies and their orientations need to be preserved and the top-level layout is given, we solve the problem in $O(n)$ time, where n is the total number of rectangles. The case in which the global layout is not fixed is much more challenging: it takes a combination of several techniques based on regular edge labelings to obtain an algorithm that solves the problem optimally in $O(k^4 \log k + n)$ time, for k the number of top-level regions; we expect k to be much smaller than n. Finally, we prove that the case in which the orientations of adjacencies do not need to be preserved is NP-hard; we give worst-case bounds and an approximation algorithm.

2 Preliminaries

Rectangles and Subdivisions. All geometric objects like rectangles and polygons in this paper are defined as rectilinear (axis-aligned) objects in the Euclidean plane $\mathbb{R}^2$. A set of rectangles $\mathcal{R}$ is called a *rectangle complex* if the interiors of none of the rectangles overlap, and each pair of rectangles is either completely disjoint or shares part of an edge; no two rectangles may meet in a single point. Each rectangle of a rectangle complex is a *cell* of that complex. We represent rectangle complexes using a structure that has bidirectional pointers between neighboring cells. Let $\mathcal{R}$ be a rectangle complex. The *boundary* of $\mathcal{R}$ is the boundary of the the union of the rectangles in $\mathcal{R}$. $\mathcal{R}$ is *simple* if its boundary is a simple polygon, i.e., it is connected and has no holes. We say that $\mathcal{R}$ is *convex* if its boundary is orthogonally convex, i.e., the intersection of any horizontal or vertical line with $\mathcal{R}$ is either empty or a single line segment. We say that $\mathcal{R}$ is *rectangular* if its boundary is a rectangle. Let $\mathcal{R}'$ be another rectangle complex. We say that $\mathcal{R}'$ is an *extension* of $\mathcal{R}$ if there is a bijective mapping between the cells in $\mathcal{R}$ and $\mathcal{R}'$ that preserves the adjacencies and their orientations. Note that $\mathcal{R}'$ could have adjacencies not present in $\mathcal{R}$ though. We say that $\mathcal{R}'$ is a *simple extension* of $\mathcal{R}$ if $\mathcal{R}$ is not simple but $\mathcal{R}'$ is; similarly we may call it a *convex extension* or a *rectangular extension*. Every rectangle complex has a rectangular extension (proof in full version [3]).

We define $\mathbb{D} = \{\text{left}, \text{right}, \text{top}, \text{bottom}\}$ to be the set of the four cardinal directions. For a direction $d \in \mathbb{D}$ we use the notation $-d$ to refer to the direction opposite from d. We define an object $O \subset \mathbb{R}^2$ to be *extreme* in direction d with respect to a rectangle complex $\mathcal{R}$ if there is a point in O that is at least as far in direction d as any point in $\mathcal{R}$. Let $R \in \mathcal{R}$ be a cell, and $d \in \mathbb{D}$ a direction. We say R is *d-extensible* if there exists a rectangular extension $\mathcal{R}'$ of $\mathcal{R}$ in which R is extreme in direction d with respect to $\mathcal{R}'$ (or in other words, if its d-side is part of the boundary of $\mathcal{R}'$). A set of simple rectangle complexes $\mathcal{L}$ is called a (rectilinear) *layout* if the boundary of the union of all complexes is a rectangle, the interiors of the complexes are disjoint, and no point in $\mathcal{L}$ belongs to more than three cells. If all complexes are rectangular we say that $\mathcal{L}$ is a *rectangular layout*. We call the rectangle bounding $\mathcal{L}$ the *root box*. Let $\mathcal{L}$ be a rectilinear layout. We define the *global layout* $\mathcal{L}'$ of $\mathcal{L}$ as the subdivision of the root box of $\mathcal{L}$, in which the (*global*) regions are defined by the boundaries of the complexes in $\mathcal{L}$. We say $\mathcal{L}'$ is *rectangular* if all regions in $\mathcal{L}'$ are rectangles.

Dual Graphs of Rectangle Complexes. The *dual graph* of a rectangular complex is an embedded planar graph with one vertex for every rectangle in the complex, and an edge between two vertices if the corresponding rectangles touch (have overlapping edge pieces). The *extended dual graph* of a rectangular complex with a rectangular boundary has four additional vertices for the four sides of the rectangle, and an edge between a normal vertex and an additional vertex if the corresponding rectangle touches the corresponding side of the bounding box. We will be using dual graphs of the whole rectangular layout, of individual complexes, and of the global layout (ignoring the bottom level subdivision); Figure 3 shows some examples. Extended dual graphs of rectangular rectangle complexes are fully triangulated (except for the outer face which is a quadrilateral), and the graphs that can arise in this way are characterized by the following lemma [9, 11, 17]:
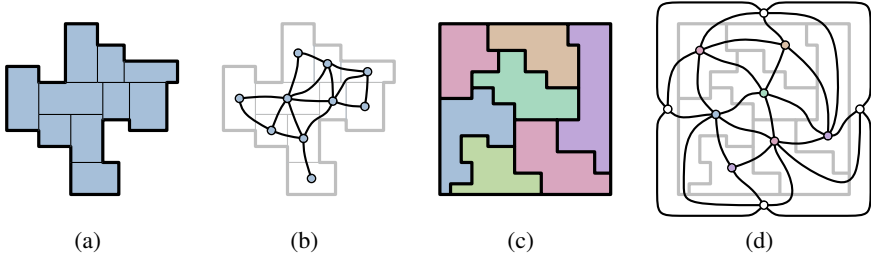
Fig. 3. (a) A bottom level rectangle complex. (b) The dual graph of the complex. (c) A global layout. (d) The extended dual graph of the global layout.

Lemma 1. *A triangulated plane graph G with a quadrilateral outer face is the dual graph of a rectangular rectangle complex if and only if G has no separating triangles.*

Now, consider the three types of adjacencies we wish to preserve: 1) (top-level) adjacencies between global regions, 2) internal (bottom-level) adjacencies between the cells in one rectangle complex, and 3) external (bottom-level) adjacencies between cells of adjacent rectangle complexes.

Observation 1. *It is always possible to keep all internal bottom-level adjacencies.*

Observation 2. *It is possible to keep all top-level adjacencies if and only if the extended dual graph of the global input layout has no separating triangles.*

Observation 1 is proven in the full version [3], and Observation 2 follows from Lemma 1 since the extended dual graph of the global regions is fully triangulated.

From now on we assume that the dual graph of the global regions has no separating triangles, and we will preserve all adjacencies of types 1 and 2. Unfortunately, it is not always possible to keep adjacencies of type 3—see Figure 4—and for every adjacency of type 3 that we fail to preserve, another adjacency that was not present in the original layout will appear. Therefore, our aim is to preserve as many of these adjacencies as possible.
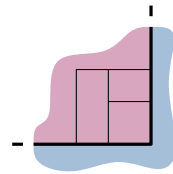
Fig. 4. Not all external adjacencies can be kept

3   Preserving Orientations

We begin studying the version of the problem where all internal adjacencies have to be preserved respecting their original orientations. Additionally, we want to maximize the number of preserved and correctly oriented (bottom-level) external adjacencies. We consider two scenarios: first we assume that the global layout is part of the input, and then we study the case in which we optimize over all global layouts. The former situation is particularly interesting for GIS applications, in which the user specifies a certain global layout that needs to be filled with the bottom-level cells. If, however, the bottom-level adjacencies are more important, then optimizing over global layouts allows to preserve more external adjacencies.
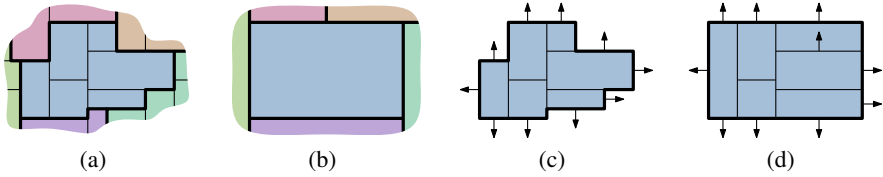
Fig. 5. (a) A region in the input. (b) The same region in the given global layout. (c) Edges of rectangles that want to become part of a boundary have been marked with arrows. Note that one rectangle wants to become part of the top boundary but can't, because it is not extensible in that direction. (d) All arrows that aren't blocked can be made happy.

3.1 Given the Global Layout

In this section we are given, in addition to the initial two-level subdivision $\mathcal{L}$, a global target layout $\mathcal{L}'$. The goal is to find a two-level treemap that preserves all oriented bottom-level internal adjacencies and that maximizes the number of preserved oriented bottom-level external adjacencies in the output.

First observe that in the rectangular output layout any two neighboring global regions have a single orientation for their adjacency. Hence we can only keep those bottom-level external adjacencies that have the same orientation in the input as their corresponding global regions have in the output layout. Secondly, consider a rectangle R in a complex $\mathcal{R}$, and a rectangle B in another complex $\mathcal{B}$. Observe that if R and B are adjacent in the input, for example with R to the left of B, then their adjacency can be preserved only if R is right-extensible in $\mathcal{R}$ and B is left-extensible in $\mathcal{B}$.

The main result in this section is that the previous two conditions are enough to describe all adjacencies that cannot be preserved, whereas all the other ones can be kept. Furthermore, we will show how to decide extensibility for convex complexes, and how to construct a final solution that preserves all possible adjacencies, leading to an algorithm for the optimal solution.

Recall that we assume all regions are orthogonally convex. Consider each rectangle complex of $\mathcal{L}$ separately. Since we know the required global layout and since all cells externally adjacent to our region are consecutive along its boundary, we can immediately determine the cells on each of the four sides of the output region (see Figure 5). The reason is that for a rectangle R that is exterior to its region $\mathcal{R}$, and that is adjacent to another rectangle $B \in \mathcal{B}$, their adjacency is relevant only if $\mathcal{R}$ and $\mathcal{B}$ are adjacent with the same orientation in the global layout. We can easily categorize the extensible rectangles of a convex rectangle complex. For the proof of the following lemma and other proofs in this section we refer to the full version [3].

Lemma 2. *Let $\mathcal{R}$ be a convex rectangle complex, let $R \in \mathcal{R}$ be a rectangle, and $d \in \mathbb{D}$ a direction. R is d-extensible if and only if there is no rectangle $R' \in \mathcal{R}$ directly adjacent to R on the d-side of R.*

Unfortunately, though, we cannot extend all extensible rectangles at the same time. However, we show that we can actually extend all those rectangles that we want to extend for an optimal solution.

We call a rectangle of a certain complex belonging to a global region *engaged* if it wants to be adjacent to a rectangle of another global region, and the direction of their desired adjacency is the same as the direction of the adjacency between these two regions in the global layout. We say it is *d*-engaged if this direction is $d \in \mathbb{D}$.

Therefore, the rectangles that we want to extend are exactly those that are *d*-extensible and *d*-engaged, since they are the only ones that help preserve bottom-level exterior adjacencies. It turns out that extending all these rectangles is possible, because the engaged rectangles of $\mathcal{R}$ have a special property:

Lemma 3. *If we walk around the boundary of a region $\mathcal{R}$, we encounter all d-engaged rectangles consecutively.*

This property of *d*-engaged rectangles is useful due to the following fact.

Lemma 4. *Let $\mathcal{R}$ be a convex rectangle complex composed of r rectangles, and let S be a subset of the extensible and engaged rectangles in $\mathcal{R}$ with the property that if we order them according to a clockwise walk along the boundary of $\mathcal{R}$, all d-extensible rectangles in S are encountered consecutively for each $d \in \mathbb{D}$ and in the correct clockwise order. We can compute, in $O(r)$ time, a rectangular extension $\mathcal{R}'$ of $\mathcal{R}$ in which all d-extensible rectangles in S are extreme in direction d, for all $d \in \mathbb{D}$.*

Therefore, the engaged and extensible rectangles form a subset of rectangles for which Lemma 4 holds, thus by using the lemma we can find a rectangular extension where all extensible and engaged rectangles are extreme in the appropriate direction.

Then we can apply this idea to each region. Now we still have to match up the adjacencies in an optimal way, that is, preserving as many adjacencies from the input as possible. This can be done by matching horizontal and vertical adjacencies independently. It is always possible to get all the external bottom-level adjacencies that need to be preserved. This can be seen as follows. We process first all horizontal adjacencies. Consider a complete stretch of horizontal boundary in the global layout. Then the position and length of the boundary of each region adjacent to that boundary are fixed, from the global layout. The only freedom left is in the *x*-coordinates of the vertical edges of the rectangles that form part of that boundary (except for the leftmost and rightmost borders of each region, which are also fixed). Since the adjacencies that want to be preserved are part of the input, it is always possible to set the *x*-coordinates in order to fulfill them all. The same can be done with all horizontal boundaries. The vertical boundaries are independent, thus can be processed in exactly the same way. This yields the main theorem in this subsection.

Theorem 1. *Let $\mathcal{T}$ be a 2-level treemap, where n is the number of cells in the bottom level, and where all global regions are orthogonally convex. For a given global target layout $\mathcal{L}$, we can find, in $O(n)$ time, a rectangular layout of $\mathcal{T}$ that respects $\mathcal{L}$, preserves all oriented internal bottom-level adjacencies, and preserves as many oriented external bottom-level adjacencies as possible.*

3.2 Unconstrained Global Layout

In this section the global target layout of the rectangle complexes is not given, i.e., we are given a rectilinear input layout and need to find a rectangular output layout

preserving all adjacencies of the rectangle complexes and preserving a maximum number of adjacencies of the cells of different complexes.

We can represent a particular rectangular global layout $\mathcal{L}$ as a *regular edge labeling* [10] of the dual graph $G(\mathcal{L})$ of the global layout. Let $G(\mathcal{L})$ be the extended dual graph of $\mathcal{L}$. Then $\mathcal{L}$ induces an edge labeling as follows: an edge corresponding to a joint vertical (horizontal) boundary of two rectangular complexes is colored blue (red). Furthermore, blue edges are directed from left to right and red edges from bottom to top. Clearly, the edge labeling obtained from $\mathcal{L}$ in this way satisfies that around each inner vertex v of $G(\mathcal{L})$ the incident edges with the same color and the same direction form contiguous blocks around v. The edges incident to one of the external vertices $\{l, t, r, b\}$ all have the same label. Such an edge labeling is called *regular* [10]. Each regular edge labeling of the extended dual graph $G(\mathcal{L})$ defines an equivalence class of global layouts.

In order to represent the family of all possible rectangular global layouts we apply a technique described by Eppstein et al. [6, 5]. Let $\mathcal{L}$ be the rectilinear global input layout and let $G(\mathcal{L})$ be its extended dual graph. The first step is to decompose $G(\mathcal{L})$ by its separating 4-cycles into minors called *separation components* with the property that they do not have non-trivial separating 4-cycles any more, i.e., 4-cycles with more than a single vertex in the inner part of the cycle. If C is a separating 4-cycle the interior separation component consists of C and the subgraph induced by the vertices interior to C. The outer separation component is obtained by replacing all vertices in the interior of C by a single vertex connected to each vertex of C. This decomposition can be obtained in linear time [6]. We can then treat each component in the decomposition independently and finally construct an optimal rectangular global layout from the optimal solutions of its descendants in the decomposition tree. So let's consider a single component of the decomposition, which by construction has no non-trivial separating 4-cycles.

Preprocessing of the bottom level. We start with a preprocessing step to compute the number of realizable external bottom-level adjacencies for pairs of adjacent global regions. This allows us to ignore the bottom-level cells in later steps and to focus on the global layout and orientations of global adjacencies.

Let $\mathcal{L}$ be a global layout, let $\mathcal{R}$ and $\mathcal{S}$ be two adjacent rectangle complexes in $\mathcal{L}$, and let $d \in \mathbb{D}$ be an orientation. Then we define $\omega(\mathcal{R}, \mathcal{S}, d)$ to be the total number of adjacencies between d-engaged and d-extensible rectangles in $\mathcal{R}$ and $-d$-engaged and $-d$-extensible rectangles in $\mathcal{S}$. By Lemma 4 there is a rectangular layout of $\mathcal{R}$ and $\mathcal{S}$ with exactly $\omega(\mathcal{R}, \mathcal{S}, d)$ external bottom-level adjacencies between $\mathcal{R}$ and $\mathcal{S}$.

We show the following (perhaps surprising) lemma:

Lemma 5. *For any pair $\mathcal{L}$ and $\mathcal{L}'$ of global layouts and any pair $\mathcal{R}$ and $\mathcal{S}$ of rectangular rectangle complexes, whose adjacency direction with respect to $\mathcal{R}$ is d in $\mathcal{L}$ and d' in $\mathcal{L}'$ the number of external bottom level adjacencies between $\mathcal{R}$ and $\mathcal{S}$ in any optimal solution for $\mathcal{L}'$ differs by $\omega(\mathcal{R}, \mathcal{S}, d') - \omega(\mathcal{R}, \mathcal{S}, d)$ from $\mathcal{L}$. For adjacent rectangle complexes whose adjacency direction is the same in both global layouts the number of adjacencies in any optimal solution remains the same.*

This basically means we can consider changes of adjacency directions locally and independent from the rest of the layout. Furthermore, since the values $\omega(\mathcal{R}, \mathcal{S}, d)$ are

(a) edge flip *BD* (b) vertex flip *E*
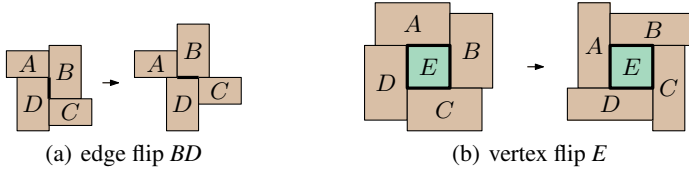
Fig. 6. Flip operations

directly obtained from counting the numbers of d-extensible and d-engaged rectangles in $\mathcal{R}$ (or $-d$-extensible and $-d$-engaged rectangles in $\mathcal{S}$) we get the next lemma.

Lemma 6. *We can compute all values $\omega(\mathcal{R},\mathcal{S},d)$ in $O(n)$ total time.*

Optimizing in a graph without separating 4-cycles. Here we will prove the following:

Theorem 2. *Let G be an embedded triangulated planar graph with k' vertices without separating 3-cycles and without non-trivial separating 4-cycles, except for the outer face which consists of exactly four vertices. Furthermore, let a weight $\omega(e,d)$ be assigned to every edge e in G and every orientation d in $\mathbb{D}$. Then we can find a rectangular subdivision of which G is the extended dual that maximizes the total weight of the directed adjacencies in $O(k'^4 \log k')$ time.*

In order to optimize over all rectangular subdivisions with the same extended dual graph we make use of the representation of these subdivisions as elements in a distributive lattice or, equivalently, as closures in a partial order induced by this lattice [6, 5]. There are two *moves* or *flips* by which we can transform one rectangular layout (or its regular edge labeling) into another one, *edge flips* and *vertex flips* (Figure 6). They form a graph where each equivalence class of rectangular layouts is a vertex and two vertices are connected by an edge if they are transformable into each other by a single move, with the edge directed toward the more counterclockwise layout with respect to this move. This graph is acyclic and its reachability ordering is a distributive lattice [7]. It has a minimal (maximal) element that is obtained by repeatedly performing clockwise (counterclockwise) moves.

   By Birkhoff's representation theorem [2] each element in this lattice is in one-to-one correspondence to a partition of a partial order $\mathcal{P}$ into an upward-closed set U and a downward-closed set L. The elements in $\mathcal{P}$ are pairs (x,i), where x is a flippable item, i.e., either the edge of an edge flip or the vertex of a vertex flip [5, 6]. The integer i is the so-called flipping number $f_x(\mathcal{L})$ of x in a particular layout $\mathcal{L}$, i.e., the well-defined number of times flip x is performed counterclockwise on any path from the minimal element $\mathcal{L}_{\min}$ to $\mathcal{L}$ in the distributive lattice. An element (x,i) is smaller than another element (y,j) in this order if y cannot be flipped for the j-th time before x is flipped for the i-th time. For each upward- and downward-closed partition U and L, the corresponding layout can be reconstructed by performing all flips in the lower set L. $\mathcal{P}$ has $O(k'^2)$ vertices and edges and can be constructed in $O(k'^2)$ time [5, 6]. The construction starts with an arbitrary layout, performs a sequence of clockwise moves
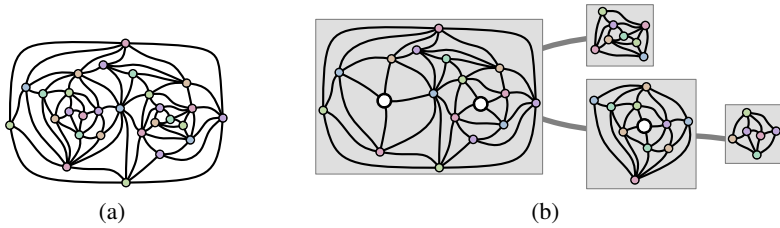
Fig. 7. (a) A graph with non-trivial separating 4-cycles. Note that some 4-cycles intersect each other. (b) A possible decomposition tree of 4-cycle-free graphs (root on the left).

until we reach $\mathcal{L}_{\min}$, and from there performs a sequence of counterclockwise moves until we reach the maximal element. During this last process we count how often each element is flipped, which determines all pairs (x,i) of $\mathcal{P}$. Since each flip (x,i) affects only those flippable items that belong to the same triangle as x, we can initialize a queue of possible flips, and iteratively extract the next flip and add the new flips to the queue in total time $O(k'^2)$. In order to create the edges in $\mathcal{P}$ we again use the fact that a flip (x,i) depends only on flips (x',i'), where x' belongs to the same triangle as x and i' differs by at most 1 from i. The actual dependencies can be obtained from their states in $\mathcal{L}_{\min}$.

Next, we assign weights to the nodes in $\mathcal{P}$. Let $\mathcal{L}_{\min}$ be the layout that is minimal in the distributive lattice, i.e., the layout where no more clockwise flips are possible. For an edge-flip node (e,i) let $\mathcal{R}$ and $\mathcal{S}$ be the two rectangle complexes adjacent across e. Then the weight $\omega(e,i)$ is obtained as follows. Starting with the adjacency direction between $\mathcal{R}$ and $\mathcal{S}$ in $\mathcal{L}_{\min}$ we cycle i times through the set $\mathbb{D}$ in counterclockwise fashion. Let d be the i-th direction and d' the $(i+1)$-th direction. Then $\omega(e,i) = \omega(e,d') = \omega(\mathcal{R},\mathcal{S},d') - \omega(\mathcal{R},\mathcal{S},d)$. For a vertex-flip node (v,i) let $\mathcal{R}$ be the degree-4 rectangle complex surrounded by the four complexes $\mathcal{S}_1,\ldots,\mathcal{S}_4$. We again determine the adjacency directions between $\mathcal{R}$ and $\mathcal{S}_1,\ldots,\mathcal{S}_4$ in $\mathcal{L}_{\min}$ and cycle i times through $\mathbb{D}$ to obtain the i-th directions $d_1,\ldots,d_4$ as well as the $(i+1)$-th directions $d'_1,\ldots,d'_4$. Then $\omega(v,i) = \sum_{j=1}^4 \omega(\mathcal{R},\mathcal{S}_j,d'_j) - \omega(\mathcal{R},\mathcal{S}_j,d_j)$. Equivalently, if the four edges incident to v are $e_1,\ldots,e_4$, we have $\omega(v,i) = \sum_{j=1}^4 \omega(e_j,d'_j)$.

Finally, we compute a maximum-weight closure of $\mathcal{P}$ using a max-flow algorithm [1, Chapter 19.2], which will take $O(k'^4 \log k')$ time for a graph with $O(k'^2)$ nodes.

Optimizing in General Graphs. In this section, we show how to remove the restriction that the graph should have no separating 4-cycles. We do this by decomposing the graph G by its separating 4-cycles and solving the subproblems in a bottom-up fashion.

Lemma 7 (Eppstein et al. [6]). *Given a plane graph G with k vertices, there exists a collection $\mathcal{C}$ of separating 4-cycles in G that decomposes G into separation components that do not contain separating 4-cycles any more. Such a collection $\mathcal{C}$ and the decomposition can be computed in $O(k)$ time.*

These cycles naturally subdivide G into a tree of subgraphs, which we will denote as T_G. Still following [6], we add an extra artificial vertex inside each 4-cycle, which corresponds to filling the void in the subdivision after removing all rectangles inside by a single rectangle. Figure 7 shows an example of a graph G and a corresponding tree T_G.

Now, all nodes of T_G have an associated graph without separating 4-cycles on which we can apply Theorem 2. The only thing left to do is assign the correct weights to the edges of these graphs. For a given node v of T_G, let G_v be the subgraph of G associated to v (with potentially extra vertices inside its 4-cycles).

For every leave v of T_G, we assign weights to the internal edges of G_v by simply setting $\omega(e, d) = \omega(\mathcal{R}, \mathcal{S}, d)$ if e separates $\mathcal{R}$ and $\mathcal{S}$ in the global layout $\mathcal{L}$. For the external edges of G_v (the edges that are incident to one of the "corner" vertices of the outer face), we fix the orientations in the four possible ways, leading to four different problems. We apply Theorem 2 four times, once for each orientation. We store the resulting solution values as well as the corresponding optimal layouts at v in T_G.

Now, in bottom-up order, for each internal node v in T_G, we proceed in a similar way with one important change: for each child μ of v, we first look up the four optimal layouts of μ and incorporate them in the weights of the four edges incident to the single extra vertex that replaced G_μ in G_v. Since these four edges must necessarily have four different orientations, their states are linked, and it does not matter how we distribute the weight over them; we can simply set the weight of three of these edges to 0 and the remaining one to the solution of the appropriately oriented subproblem. The weights of the remaining edges are derived from $\mathcal{L}$ as before, and again we fix the orientations of the external edges of G_v in four different ways and apply Theorem 2 to each of them. We again store the resulting four optimal values and the corresponding layouts at v, in which we insert the correctly oriented subsolutions for all children μ of v.

This whole process takes $O(k^4 \log k)$ time in the worst case. Finally, since weights are expressed as differences with respect to the minimal layout $\mathcal{L}_{\min}$ we compute the value of $\mathcal{L}_{\min}$ and add the offset computed as the optimal solution to get the actual value of the globally optimal solution. This takes $O(n)$ time.

Theorem 3. *Let $\mathcal{T}$ be a 2-level treemap, such that the extended dual graph G of the global layout has no separating 3-cycles. Let n be the number of cells in the bottom level and k the number of regions in the top level. Then we can find a rectangular subdivision that preserves all oriented internal bottom-level adjacencies, and preserves as many oriented external bottom-level adjacencies as possible in $O(k^4 \log k + n)$ time.*

4   Without Preserving Orientations

In this section we do not need to preserve orientations of internal adjacencies. The global regions are convex and we assume that the global layout is given. However, maximizing the number of preserved external adjacencies in this case is NP-hard even if we only have two top-level regions. For two top-level regions we give a $1/3$-approximation algorithm for this problem. Furthermore, this algorithm preserves at least $1/9$ of the external adjacencies. We also show that we sometimes cannot keep more than $1/4$ of the adjacencies. The algorithm extends to more than two regions. In this case it is a $1/6$-approximation and at least $1/18$ of the adjacencies are kept. Due to space restrictions, we defer all details of these results to the full version of this paper [3].

Acknowledgements

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows. Prentice Hall, Englewood Cliffs (1993)
2. Birkhoff, G.: Rings of sets. Duke Mathematical Journal 3(3), 443–454 (1937)
3. Buchin, K., Eppstein, D., Löffler, M., Nöllenburg, M., Silveira, R.I.: Adjacency-preserving spatial treemaps. Arxiv report, arXiv:1105.0398 (cs.CG) (May 2011)
4. Buchin, K., Speckmann, B., Verdonschot, S.: Optimizing regular edge labelings. In: Brandes, U., Cornelsen, S. (eds.) GD 2010. LNCS, vol. 6502, pp. 117–128. Springer, Heidelberg (2011)
5. Eppstein, D., Mumford, E.: Orientation-constrained rectangular layouts. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 266–277. Springer, Heidelberg (2009)
6. Eppstein, D., Mumford, E., Speckmann, B., Verbeek, K.: Area-universal rectangular layouts. In: Proc. SoCG, pp. 267–276 (2009)
7. Fusy, É.: Transversal structures on triangulations: A combinatorial study and straight-line drawings. Discrete Mathematics 309(8), 1870–1894 (2009)
8. Heilmann, R., Keim, D.A., Panse, C., Sips, M.: Recmap: Rectangular map approximations. In: Proceedings of the IEEE Symposium on Information Visualization, pp. 33–40. IEEE Computer Society, Washington, DC, USA (2004)
9. Kant, G., He, X.: Two algorithms for finding rectangular duals of planar graphs. In: van Leeuwen, J. (ed.) WG 1993. LNCS, vol. 790, pp. 396–410. Springer, Heidelberg (1994)
10. Kant, G., He, X.: Regular edge labeling of 4-connected plane graphs and its applications in graph drawing problems. Theoretical Computer Science 172(1-2), 175–193 (1997)
11. Kozminski, K., Kinnen, E.: Rectangular duals of planar graphs. Networks 5(2) (1985)
12. Mansmann, F., Keim, D.A., North, S.C., Rexroad, B., Sheleheda, D.: Visual analysis of network traffic for resource planning, interactive monitoring, and interpretation of security threats. IEEE Transactions on Visualization and Computer Graphics 13, 1105–1112 (2007)
13. Raisz, E.: The rectangular statistical cartogram. Geographical Review 24(2), 292–296 (1934)
14. Shneiderman, B.: Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans. Graph. 11(1), 92–99 (1992)
15. Slingsby, A., Dykes, J., Wood, J.: Configuring hierarchical layouts to address research questions. IEEE Trans. Vis. Comput. Graph. 15(6), 977–984 (2009)
16. Slingsby, A., Dykes, J., Wood, J.: Rectangular hierarchical cartograms for socio-economic data. Journal of Maps v2010, 330–345 (2010), doi:10.4113/jom.2010.1090
17. van Kreveld, M., Speckmann, B.: On rectangular cartograms. CGTA 37(3) (2007)
18. Wood, J., Dykes, J.: Spatially ordered treemaps. IEEE Trans. Vis. Comput. Graph. 14(6), 1348–1355 (2008)

Register Loading via Linear Programming[*]

Gruia Calinescu[1] and Minming Li[2]

[1] Department of Computer Science, Illinois Institute of Technology
calinescu@iit.edu
[2] Department of Computer Science, City University of Hong Kong
minmli@cs.cityu.edu.hk

Abstract. We study the following optimization problem. The input is
a number k and a directed graph with a specified "start" vertex, each
of whose vertices may have one "memory bank requirement", an inte-
ger. There are k "registers", labeled $1 \ldots k$. A valid solution associates to
the vertices with no bank requirement one or more "load instructions"
$L[b, j]$, for bank b and register j, such that every directed trail from the
start vertex to some vertex with bank requirement c contains a vertex
u that has been associated $L[c, i]$ (for some register $i \leq k$) and no ver-
tex following u in the trail has been associated an $L[b, i]$, for any bank
b. The objective is to minimize the total number of associated load in-
structions. We give a $k(k + 1)$-approximation algorithm based on linear
programming rounding, with $(k+1)$ being the best possible unless Vertex
Cover has approximation $2 - \epsilon$ for $\epsilon > 0$. We also present a $O(k \log n)$
approximation, with n being the number of vertices in the input directed
graph. Based on the same linear program, another rounding method out-
puts a valid solution with objective at most $2k$ times the optimum for k
registers, using $2k$ registers.

1 Introduction

We study the following optimization problem, called k-BSIM. The input is a
number k and a directed graph with a specified "start" vertex, each of whose
vertices may have one "memory bank requirement", an integer. There are k
"registers", labeled $1 \ldots k$. A valid solution associates to the vertices with no
bank requirement one or more "load instructions" $L[b, j]$, for bank b and register
j, such that every directed trail from the start vertex to some vertex with bank
requirement c contains a vertex u that has been associated $L[c, i]$ (for some
register $i \leq k$) and no vertex following u in the trail has been associated an
$L[b, i]$, for any bank b. The objective is to minimize the total number of associated
load instructions. This problem has applications in embedded systems. k-BSIM
is used to solve a slightly more complicated problem, described below.

The Original k-Bank Selection Instruction Minimization problem (k-OBSIM)
is defined as follows: The input is a number k and a directed graph, called the

---

Control Flow Graph (CFG), with a specified "start" vertex, and for each vertex we have at most one "memory bank requirement", an integer. The vertices of the CFG correspond to blocks of code in an embedded system, and arcs represent possible jumps in the code. Many embedded systems use partitioned memory architecture, and program variables are stored in "banks" that must be stored in registers before use. A vertex with no associated bank is called *transparent* node and a vertex with one associated bank is called *required* vertex. There are k registers, labeled $1 \ldots k$. Let B be the set of possible banks.

Each vertex of the CFG may "load" a bank (use a bank load instruction), either at the "entrance" or at the "exit" of the vertex (or both). We write $L_{in}^u[b, j]$ for loading bank b in register j at the entrance of vertex u, and $L_{out}^u[b, j]$ for loading at the exit. Although loading is allowed also on the arcs of the CFG, we prefer to subdivide such arcs with transparent vertices to keep the problem description simpler. Also, when $k = 1$, a small proof shows that any arc load can be done instead at the entry of the head of the arc, resulting in another feasible solution not worse in number of load instructions.
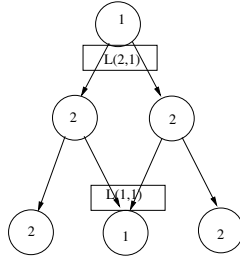
Fig. 1. An example input with a feasible solution when k=1. Circles represent nodes and the number in the circle means the bank required by this node. The start vertex is not represented (or it could be the top vertex, without the bank requirement).

For a directed path P, let $\hat{P}$ denote the set of vertices in P other than the start and end vertices. Let s be the start vertex of the CFG. Bank load instructions must be associated to CFG vertices such that, for any trail (directed path, not necessarily simple) P from s to some node v that has a bank requirement b, P has a vertex w (which may be v) and a register j for some $j \leq k$ such that one of the following holds:

1. $w = v$, and we have $L_{in}^w[b, j]$ but no $L_{in}^w[c, j]$ for any $c \in B$ with $c \neq b$
2. w has $L_{out}^w[b, j]$, and for no $c \in B$ with $c \neq b$ there is $L_{in}^v[c, j]$ or $L_{out}^w[c, j]$ or a vertex u of $\hat{P}$ with either $L_{out}^u[c, j]$ or $L_{in}^u[c, j]$
3. w has $L_{in}^w[b, j]$, and for no $c \in B$ with $c \neq b$ there is $L_{in}^v[c, j]$ or $L_{in}^w[c, j]$ or $L_{out}^w[c, j]$ or a vertex u of $\hat{P}$ with either $L_{out}^u[c, j]$ or $L_{in}^u[c, j]$

See figures 1 and 2 for examples of feasible solutions. In other words, b is always loaded in some register and no other bank loads over b on any path leading to v. The objective is to minimize the total number of bank load instructions (as to keep the embedded code as short as possible).
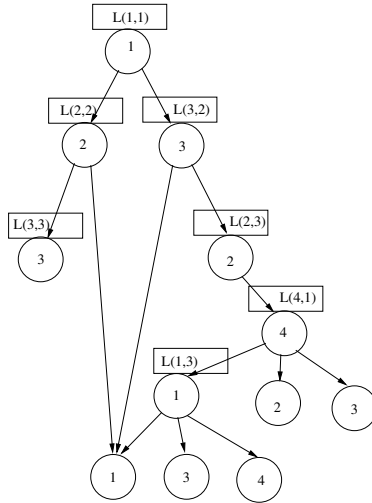
Fig. 2. An example input with a feasible solution when k=3. For the bottom node which accesses bank 1, there are three paths entering it, where two of them have bank 1 loaded in register 1 and one of them has bank 1 loaded in register 3. We abbreviate $L_{out}[b, j]$ or $L_{in}[b, j]$ to $L[b, j]$ since the figure clearly shows the insertion position. The start vertex is not represented and has only one arc, to the top-most vertex.

Partitioned memory architecture is common in 8-bit microcontrollers. For example, Freescale [1] 68HC11 8-bit microcontrollers allow multiple 64KB memory banks to be accessed by their 16-bit address registers with only one bank being active at a time. Zilog [2] Z80 also addresses a maximum of 64 KB memory using 16-bit address registers. Other examples include Intel 8051 processor family and MOS technology 6502 series microcontrollers. For embedded systems using these 8-bit microcontrollers, how to insert bank selection instructions (called by us previously "load") to minimize the code size is an important research topic.

The paper [22] studies 1-OBSIM without transparent nodes, showing NP-Hardness and a 2-approximation algorithm. We generalize this result, by giving a $k(k + 1)$-approximation algorithm for k-OBSIM (transparent nodes also allowed) based on linear programming rounding. In a personal communication, Yuan Zhou [29] claimed that 1-OBSIM without transparent nodes does not have a $2 - \epsilon$ approximation algorithm unless Vertex Cover has a $2 - \epsilon$ approximation algorithm (and it is believed that such an algorithm does not exist). We also present such a reduction, and generalize it to show that k-OBSIM without transparent nodes does not have a $\alpha - \epsilon$ approximation algorithm unless $(k+1)$-uniform Hypergraph Vertex Cover has such an algorithm. It is known that it is NP-hard to approximate Hypergraph Vertex Cover in a r-uniform-hypergraph to within a factor of $(r - 1 - \epsilon)$ [11], and it is believed that an approximation ratio of r is the best a polynomial-time algorithm can do. Thus it is NP-Hard to approximate k-OBSIM ($k \geq 2$) within a factor of $k - \epsilon$.

Based on the same linear program, we also present $O(k \log n)$ approximation, with n being the number of vertices in the input directed graph, and an algorithm with objective at most $2k$ times optimum, however using $2k$ instead of k registers. The linear program contains one "clever" constraint which makes it similar, for $k = 1$, to the linear program used by Garg, Vazirani, and Yannakakis [14] to obtain a 2-approximation for Node Weighted Multiway Cut. In addition to the k-uniform Hypergraph Vertex Cover, k-BSIM inherits some hardness from k-coloring (as in the register allocation papers of Thorup [26] and Kannan and Proebsting [18]) and we see intuitive connections to Directed Steiner Tree [28,8,30] and Multicut in Directed Graphs [9,15,3].

Note that we are not optimizing the run-time of the program (in which case the problem would resemble caching [24]) nor the number of registers needed for a program (as in Thorup [26], Kannan and Proebsting [18], and Jansen and Reiter [17]). An early related work is [16]. More recent work appears in [23], while [13,19,20,25,21] and many other papers deal with practical issues of NP-hard variants of register allocation. Also, as opposed to most theoretical work, we do not assume any structure (such as low treewidth) for the input graph. Most related to our model are the "spill heuristics" discussed in [6,5,7,27,12,10], but as the name suggest we do not know of any previous approximation algorithms. Here "spill" means putting some variables in the RAM instead of registers and the aim of those heuristics is to minimize the number of variables "spilled".

So, while our problem resembles register allocation, it differs in the following ways. We do not force a bank with a "live-range" to stay in the same register but allow the register to change content over time. We also have the restriction that a bank variable cannot be stored in RAM when it is to be visited (it must come back to a register in time, which is different from the Register Allocation problem where a variable can be spilled). In the new setting, our goal is to minimize the total number of content switching instructions for registers inserted into the program.

We omit due to lack of space the approximation-preserving reduction from $(k + 1)$-uniform Hypergraph Vertex Cover to k-OBSIM without transparent nodes. The other reduction appears next. Section 3 presents our integer linear program for 1-BSIM and the rounding procedure giving the 2-approximation. Section 4 presents our results for k-BSIM.

2   Reduction

We continue by showing how k-OBSIM reduces to k-BSIM (with transparent nodes), a problem easier to describe. Given an instance of k-OBSIM, for every node v in CFG with bank requirement b, add a transparent node v_{in}^t which takes in all the incoming arcs of v and has one arc to v, thus v has exactly one incoming arc. Also add a transparent node v_{out}^t which sends out all the outgoing arcs of v, and has one arc from v, thus v has exactly one outgoing arc. If for the k-OBSIM instance, there is a load operation at the entrance of some vertex v with bank requirement b, then in the transformed k-BSIM instance, we do the same load to node v_{in}^t; if there is a load operation at the exit of some vertex v with bank

requirement b, then in the transformed k-BSIM instance, we do the same load to node v_{out}^t. Thus, with the above correspondence, a feasible solution for the k-OBSIM instance can be changed to a feasible solution for the transformed k-BSIM instance. Also, it is easy to see that a feasible solution for the transformed k-BSIM instance can be changed to a feasible solution for the original k-OBSIM instance, without an increase in the objective function.

3   1-BSIM

Assume that every node is reachable from the start vertex s. We again do a similar transformation for the given OBSIM instance. The linear program obtained later after this transformation is more intuitive (but this reduction only works for $k = 1$).

Create a new transparent start vertex, s', with exactly one arc, outgoing to the original s. For every node v in CFG with bank requirement b, split v in two nodes, v_{in} with all the incoming arcs of v, and v_{out} with all the outgoing arcs of v; both have requirement b. Note that we do not have an arc from v_{in} to v_{out}. Moreover, for v_{in}, add a transparent node which takes in all the incoming arcs of v_{in} and has one arc to v_{in}, thus v_{in} has exactly one incoming arc. Also, for v_{out}, add a transparent node which sends out all the outgoing arcs of v_{out}, and has one arc from v_{out}, thus v_{out} has exactly one outgoing arc. We now insist that all load instructions are done at transparent nodes.

Call the resulting directed graph $G = (V, E)$. Let F be the set of transparent nodes, R^I be the required nodes with one incoming arc each (that is, the v_{in} nodes), and R^O be the required nodes with one outgoing arc each (that is, the v_{out} nodes). For $a \in B$, let R_a^I be the subset of R^I with requirement a, and R_a^O be the subset of R^O with requirement a. In G, we insist that for every bank $a \in B$ and every vertex $v \in R_a^I$, every path ending in v and starting at either s' or a vertex of $R^O \setminus R_a^O$ contains a transparent vertex u loading bank a, and no load instructions after u. Call BSIM this new problem. One can check that a 1-OBSIM feasible solution for the original instance corresponds to a BSIM feasible solution to the constructed instance, with the same number of load instructions.

For $v \in V$ and $a \in B$, let $\mathcal{P}_a^v$ be the (possibly infinite) set of (not necessarily simple) paths of G from v to some node of R_a^I. Write the following integer linear program (**IP1**), with variables x_b^v for every node $v \in F$ and bank requirement $b \in B$ (x_b^v in the IP would be 1 if transparent node v loads bank b), and variables d_b^v for every node $v \in (F \cup R^O)$ and bank requirement $b \in B$ (d_b^v in the IP would be 1 if either $\mathcal{P}_b^v = \emptyset$ or, for any $P \in \mathcal{P}_b^v$, $\hat{P}$ contains at least one node that loads bank b).

$$\min \sum_{v \in F, b \in B} x_b^v \text{ subject to}$$

$$\sum_{b \in B} x_b^v \leq 1 \qquad\qquad \forall v \in F \qquad\qquad (1)$$

$$d_a^u \geq 1 \qquad\qquad \forall a \in B \wedge \forall u \in \left(\{s'\} \cup R^O \setminus R_a^O \right) \qquad (2)$$

$$d_a^v \geq x_b^v \qquad\qquad \forall a \neq b \in B \wedge \forall v \in F \qquad\qquad (3)$$

$$d_a^u \leq d_a^v + x_a^v \quad \forall a \in B \wedge \forall u \in \left(F \cup R^O \right) \wedge \forall v \in F \text{ such that } uv \in E \qquad (4)$$

$$d_a^u = 0 \quad \forall a \in B \wedge \forall u \in F \text{ such that } \exists v \in R_a^I \text{ such that } uv \in E \qquad (5)$$

$$d_a^u + d_b^u \geq 1 \qquad\qquad \forall a \neq b \in B \wedge \forall u \in F \qquad\qquad (6)$$

$$x_a^v \geq 0 \qquad\qquad \forall v \in F \wedge a \in B \qquad\qquad (7)$$

$$d_a^v \geq 0 \qquad\qquad \forall v \in \left(F \cup R^O \right) \wedge \forall a \in B \qquad\qquad (8)$$

$$x_a^v, d_a^v \in \mathbb{Z} \qquad\qquad \forall v \in V \wedge \forall a \in B \qquad\qquad (9)$$

We sketch the fact that any IP solution obtained from a BSIM solution satisfies all these constraints, and that we can construct a valid BSIM solution from any IP solution. It is rather obvious the objective function matches. Constraint (1) enforces only one load per transparent vertex. Constraint (2) enforces the condition that for every bank $a \in B$ and every vertex $v \in R_a^I$, every path ending in v and starting at either s' or a vertex of $R^O \setminus R_a^O$ contains a transparent vertex loading bank a; it does not guarantee however no load instructions after that transparent vertex. This is done by Constraint (3), which enforces the following observation: if bank b is loaded in vertex v, then for any path from v to a vertex requiring bank a, there must be at least one load of bank a. Constraint (4) enforces the following: if for bank a and vertices u, v with $uv \in E$, we have that $\mathcal{P}_a^v \neq \emptyset$ and there exists path $P \in \mathcal{P}_a^v$ such that $\hat{P}$ contains no node that loads bank a, and v also does not load a, then $\mathcal{P}_a^u \neq \emptyset$ and there exists path $P' \in \mathcal{P}_a^u$ (namely, edge uv followed by P) such that $\hat{P}'$ contains no node that loads bank a. Constraint (5) means that if $v \in R_a^I$ and $uv \in E$, then $\mathcal{P}_a^u \neq \emptyset$ and there exists path $P' \in \mathcal{P}_a^u$ (namely, edge uv) such that $\hat{P}'$ contains no node that loads bank a.

The trickier to verify constraint is (6), which indeed holds for integer solutions as, if for vertex v and banks $a \neq b$, $\mathcal{P}_a^v$ and $\mathcal{P}_b^v$ are both non-empty, then no matter if or what bank is loaded in v or in any other free vertex, either we must have that every path $P \in \mathcal{P}_a^v$ satisfies that $\hat{P}$ contains at least one node that loads bank a, or we must have that every path $P \in \mathcal{P}_b^v$ satisfies that $\hat{P}$ contains at least one node that loads bank b. Indeed, if there is a path $P \in \mathcal{P}_a^v$ with $\hat{P}$ not loading a, then we must have that either v loads a, or all the paths from $\{s'\} \cup R^0$ to v load a or are coming from R_a^O (and a path from $\{s'\} \cup R^0$ to v must exist since we assumed every vertex of the CFG is reachable from s). Thus if such a P exists, we must have that every path $P' \in \mathcal{P}_b^v$ satisfies that $\hat{P}'$ contains at least one node that loads bank b. It is the crucial (and clever) Constraint (6) that allows good approximation algorithms.

3.1 LP Rounding

Let **LP1** be the linear programming relaxation of **IP1**, which can be solved in polynomial time. Let $\bar{x}_a^v, \bar{d}_a^v$ be an optimum **LP1** solution. Pick uniformly at random real number $\delta \in (0, 1/2)$. Set $x_a^v = 1$ iff $\bar{d}_a^v < \delta \leq \bar{d}_a^v + \bar{x}_a^v$. Set $d_a^v = 1$ iff $\mathcal{P}_a^v = \emptyset$ or any path P in $\mathcal{P}_a^v$ has some $u \in \hat{P}$ with $x_a^u = 1$. It is immediate that $Pr[x_a^v = 1] \leq 2\bar{x}_a^v$, and thus we have a 2-approximation, provided we prove that for any such δ, we get a valid IP solution.

Lemma 1. *For any $\delta \in (0, 1/2)$, and for any $v \in (F \cup R^0)$ and $b \in B$, if $\bar{d}_b^v \geq 1/2$ and $\mathcal{P}_b^v \neq \emptyset$, then any path $P \in \mathcal{P}_b^v$ has a vertex $z \neq v$ with $x_b^z = 1$ (in other words, bank load b at CFG vertex z).*

Proof. Let P be such a path from v to some $x \in R_b^I$. Note that the vertex y before x in P has $\bar{d}_b^y = 0$. Therefore P must have consecutive vertices u and u' such that $\bar{d}_b^{u'} < \delta$ and $\bar{d}_b^u \geq \delta$; here u may be v. Note that $u' \in F$. Constraint (4) also gives $\bar{d}_b^{u'} + \bar{x}_b^{u'} \geq \bar{d}_b^u \geq \delta$, and therefore $x_b^{u'}$ is set to 1 by the algorithm. The lemma holds with $z = u'$. $\qquad\qquad\square$

Now we check the feasibility of all constraints. For Constraint (1), note that for $a \neq b \in B$ and $v \in F$, in order to have both x_a^v and x_b^v be made 1, we must have $\bar{d}_a^v < 1/2$ and $\bar{d}_b^v < 1/2$, leading to $\bar{d}$ violating Constraint (6).

Constraint (2), for $a \in B$ and $u \in R^O \setminus R_a^O$ follows from $\bar{d}_a^u \geq 1$ and the lemma above.

Constraint (3), for $a \neq b \in B$ and $v \in F$ follows as follows: if $x_b^v = 1$, then $\bar{d}_b^v < 1/2$, and therefore by $\bar{d}$ satisfying Constraint (6), $\bar{d}_a^v \geq 1/2$. Therefore, by the lemma above applied to v and a, we set $d_a^v = 1$ whether $\mathcal{P}_a^v = \emptyset$ or not.

Constraint (4), for $a \in B$ and $uv \in E$, follows from the way d was constructed: if both $d_a^v = 0$ and $x_a^v = 0$, then $\mathcal{P}_a^u \neq \emptyset$ since $\mathcal{P}_a^v \neq \emptyset$, and there is a path $P \in \mathcal{P}_a^u$ such that, for all $z \in \hat{P}$, $x_a^z = 0$: use uv and then the path $P' \in \mathcal{P}_a^v$ with, for all $z \in \hat{P}$, $x_a^z = 0$.

Constraint (5), for $a \in B$ and $u \in F$ such that there exists $uv \in E$ with $v \in R_a^I$ is also satisfied since $\mathcal{P}_a^u \neq \emptyset$ and the path with its only arc uv has no interior.

Constraint (6), for $a \neq b \in B$ and $u \in F$ follows as follows: either $\bar{d}_a^u \geq 1/2$ or $\bar{d}_b^u \geq 1/2$, and the lemma above ensures that the one at least $1/2$ becomes 1.

Note that only a polynomial number of values of δ must be tried, so derandomization is immediate. A more complicated analysis, omitted for lack of space, shows that every value of δ works. We do not see half-integrality as in [14], and we do not see a direct primal-dual algorithm.

4 k-BSIM

Assume that every node is reachable from the start vertex s, which is transparent. Let R be the set of required vertices, and, for $a \in B$, let R_a be the subset of R with requirement a. For $v \in V$ and $a \in B$, let $\mathcal{P}_a^v$ be the (possibly infinite) set of (not necessarily simple) paths of G from v to some node of R_a.

Write the following integer linear program (**IP2**), with variables x_b^v for every node $v \in F$ and bank requirement $b \in B$ (x_b^v in the IP would be 1 if transparent node v loads bank b, in any of its registers), and variables d_b^v for every node $v \in ((F \cup R) \setminus R_b)$ and bank requirement $b \in B$ (d_b^v in the IP would be 1 if either $\mathcal{P}_b^v = \emptyset$ or, for any $P \in \mathcal{P}_b^v$, $\hat{P}$ contains at least one node that loads bank b, in any register).

$$\min \sum_{v \in F, b \in B} x_b^v \text{ subject to}$$

$$\sum_{b \in B} x_b^v \leq k \qquad \forall v \in F \qquad (10)$$

$$d_a^u \geq 1 \qquad \forall a \in B \wedge \forall u \in (\{s\} \cup R \setminus R_a) \qquad (11)$$

$$\sum_{a \in B} d_a^u \geq |B| - k \qquad \forall u \in F \qquad (12)$$

$$d_a^u \leq d_a^v + x_a^v \quad \forall a \in B \wedge \forall u \in (F \cup R \setminus R_a) \wedge \forall v \in F \text{ with } uv \in E \quad (13)$$

$$d_a^u = 0 \quad \forall a \in B \wedge \forall u \in F \text{ with } \exists v \in R_a \text{ such that } uv \in E \quad (14)$$

$$1 \geq x_a^v \geq 0 \qquad \forall v \in F \wedge \forall a \in B \qquad (15)$$

$$1 \geq d_a^v \geq 0 \qquad \forall a \in B \wedge \forall v \in (F \cup R \setminus R_a) \qquad (16)$$

$$x_a^v, d_a^v \in \mathbb{Z} \qquad \forall v \in F \wedge \forall a \in B \qquad (17)$$

Constraints (12) are the generalization of the "clever" constraints (6). Constraints (12) indeed hold for integer solutions since, if for vertex v there are $k+1$ banks b with variable $d_b^v = 0$, then for any of these banks b, $\mathcal{P}_b^v \neq \emptyset$ and there is at least one path $P \in \mathcal{P}_b^v$ such that no node in $\hat{P}$ loads bank b in any of its registers. Then no matter what banks arrive or are loaded at v, we do not get a valid k-BSIM solution. Note that constraints

$$\sum_{a \in Q} d_a^u \geq |Q| - k \quad \forall Q \subseteq B \wedge \forall u \in F \qquad (18)$$

are implied by (16) and (12).

IP2 above is not equivalent to k-BSIM, as it does not specify in which register a bank is loaded. Nevertheless, from a k-BSIM solution, we can get an **IP2** solution of the same value (but not vice versa; that will be a coloring problem), by setting x_b^v to be 1 iff transparent node v loads bank b and d_b^v to be 1 iff either $\mathcal{P}_b^v = \emptyset$ or, for any $P \in \mathcal{P}_b^v$, $\hat{P}$ contains at least one node that loads bank b. We relax **IP2** to the linear program **LP2**, and solve it in polynomial time.

Let $\bar{x}_a^v, \bar{d}_a^v$ be an optimum **LP2** solution. Pick uniformly at random real number $\delta \in (0, 1/(k+1))$. Set $x_a^v = 1$ iff $\bar{d}_a^v < \delta \leq \bar{d}_a^v + \bar{x}_a^v$. Set $d_a^v = 1$ iff $\mathcal{P}_a^v = \emptyset$ or any path P in $\mathcal{P}_a^v$ has $u \in \hat{P}$ with $x_a^u = 1$. It is immediate that $Pr[x_a^v = 1] \leq (k+1)\bar{x}_a^v$. We load at v all the q banks a with $d_a^u < 1/(k+1)$ if at least one of them has $x_a^v = 1$, using registers $1, 2, \ldots, q$. We have $q \leq k$, since Constraint (18) implies that, for any vertex u, at most k banks a can have $\bar{d}_a^u < 1/(k+1)$. One needs to

check that indeed this is a valid solution of k-BSIM: pick any $u \in V$, requiring $a \in B$, any path P from s to u has vertex v' with $\bar{d}_a^{v'} < \delta \leq \bar{d}_a^{v'} + \bar{x}_a^{v'}$. Let v be the last vertex on $\hat{P}$ with $\bar{d}_b^v < \delta \leq \bar{d}_b^v + \bar{x}_b^v$, for some $b \in B$; such a vertex v exists since v' is a candidate. Then also $\bar{d}_a^v < \delta \leq 1/(k+1)$ and therefore bank a is loaded in some register at v. As we go on $\hat{P}$ from v to u, no further load instructions are selected by the algorithm, and thus at vertex u bank a is loaded. The expected cost of the solution is at most $k(k+1)$ the LP cost.

For the derandomization, we can try polynomially many values of $\delta \in (0, 1/(k+1))$, or prove any such value will do. The following comprises this discussion (as well as that of Section 3):

Theorem 1. *There is a $k(k+1)$-approximation algorithm for k-BSIM.*

Theorem 2. *There is a polynomial-time algorithm whose output uses at most $2k$ registers and a number of load instructions at most $2k$ times the optimum solution with k registers.*

Proof. If we use $2k - 1$ registers (in a bicriteria fashion), we choose uniformly at random real numbers $\delta_b \in (0, 1/2)$. Then, for every $v \in F$, if there is an a with $\bar{d}_a^v < \delta_a \leq \bar{d}_a^v + \bar{x}_a^v$, we load at node v all the banks b with $\bar{d}_b^v < \delta_b$. There can be at most $2k - 1$ banks b with $\bar{d}_b^v < 1/2$, from Constraint (18). Moreover, for any $u \in V$, requiring $a \in B$, any path P from s to u has vertex v' with $\bar{d}_a^{v'} < \delta_a \leq \bar{d}_a^{v'} + \bar{x}_a^{v'}$. Let v be the last vertex on $\hat{P}$ with $\bar{d}_b^v < \delta_b \leq \bar{d}_b^v + \bar{x}_b^v$, for some $b \in B$. Then also $\bar{d}_a^v < \delta_a$ (as otherwise there is a further, on $\hat{P}$, vertex v'' with $\bar{d}_a^{v''} < \delta_a \leq \bar{d}_a^{v''} + \bar{x}_a^{v''}$). and therefore bank a is loaded in some register at v. As we go on $\hat{P}$ from v to u, no further load instructions are selected by the algorithm, and thus at vertex u bank a is loaded. Let Q_v be the set of banks a with $\bar{d}_a^v < 1/2$. The probability that bank b is loaded at vertex $v \in Q_v$ is

$$Pr[\hat{x}_b^v = 1] \leq \sum_{a \in Q_v} 2\bar{x}_a^v Pr[\bar{d}_b^v < \delta_b], \tag{19}$$

and thus the expected number of loads at node v is at most

$$\sum_{b \in Q_v} Pr[\bar{d}_b^v < \delta_b] 2 \sum_{a \in Q_v} \bar{x}_a^v$$

If $|Q_v| \leq k$, then this quantity is at most $2k \sum_{a \in B} \bar{x}_a^v$. Otherwise, it is at most

$$\sum_{b \in Q_v} (1 - 2\bar{d}_b^v) 2 \sum_{a \in Q_v} \bar{x}_a^v = (|Q_v| - 2 \sum_{b \in Q_v} \bar{d}_b^v) 2 \sum_{a \in Q_v} \bar{x}_a^v$$

$$\leq (|Q_v| - 2(|Q_v| - k)) 2 \sum_{a \in Q_v} \bar{x}_a^v = (2k - |Q_v|) 2 \sum_{a \in Q_v} \bar{x}_a^v \leq 2k \sum_{a \in B} \bar{x}_a^v,$$

where we used Constraint (18) for the first inequality and $|Q_v| \geq k$ for the last. In all cases, the expected number of bank loads is at most $2k$ times the LP solution value. $\qed$

Assuming $\ln n \ll k$, the following result is an improvement:

Theorem 3. *There is a $O(k \ln n)$ approximation algorithm for k-BSIM.*

Proof. Use the rounding method of the previous theorem, with the interval $(0, 1/(8 \ln n))$ for each δ_a. Let Q_v be the set of banks a with $\bar{d}_a^v < 1/(8 \ln n)$; as above $|Q_v| \leq 2k$. Let Q_v' be the (random) set of banks used by the algorithm at v.

Claim. $Pr[|Q_v'| > k] \leq \frac{1}{n^2}$

Proof. We are setting up a Chernoff bound. We define $d_a = \bar{d}_a^v$, $Q = Q_v$, $q = |Q|$, and $\sigma = \sum_{a \in Q} d_a$. We may assume $q > k$ or else the claim is trivially true. For bank $a \in Q$, define the random variables:

$$Z_a = \begin{cases} 1 \text{ if } d_a > \delta_a \\ 0 \text{ otherwise} \end{cases}$$

Define the random variable $Z = \sum_{a \in Q} Z_a$. Let $p_a = 8 d_a \ln n$ and $p = \frac{\sum_{a \in Q} p_a}{q}$. Let X_a ($a \in Q$) be the random variables $Z_a - p_a$. Then X_a are mutually independent with $Pr[X_a = 1 - p_a] = p_a$ and $Pr[X_a = -p_a] = 1 - p_a$. Define the random variable $X = \sum_{a \in Q} X_a$. Then X satisfies Assumptions A.1.3 of [4] and therefore Theorem A.1.13 of [4] states that, for any $\alpha > 0$,

$$Pr[X < -\alpha] < e^{-\alpha^2/2pq}. \tag{20}$$

We have that the event $|Q_v'| > k$ is the event $Z < q - k$, which is the event $X < (q - k) - \sum_{a \in Q} p_a$. Note that

$$-(q - k) + \sum_{a \in Q} p_a = -(q - k) + 8 \ln n \sum_{a \in Q} d_a \geq 7\sigma \ln n,$$

where we used Constraints (18), which state $\sigma \geq (q - k)$. Thus Chernoff's bound from Equation (20) gives

$$Pr[|Q_v'| > k] < e^{-(7\sigma \ln n)^2/(2 \cdot 8\sigma \ln n)} \leq e^{-2\sigma \ln n} \leq e^{-2 \ln n},$$

which is what the claim requires. □

The expected number of banks, computed as in the bicriteria algorithm, does not exceed $8k \ln n$. Thus Markov's inequality gives $Pr[\text{number of banks used} > 16k(\ln n)Z_{LP2}^*] \leq 1/2$, where Z_{LP2}^* is the objective value of LP2. From Claim 4, taken as a union over all v, for only $1/n$ of this event there is a vertex loading more than k banks. So with probability $1/3$ no vertex is overloaded and less than $16k(\ln n)Z_{LP2}^*$ banks are loaded in total. This concludes the proof of Theorem 3. □

5   Conclusion

We leave open the ratio of **LP2** to the optimum k-BSIM solution. Our algorithm can easily be extended to the case when required nodes each have a set of banks $A \subset B$ with $|A| \leq k$, and all banks of A must be loaded.

References

1. Freescale, `http://www.freescale.com`
2. Zilog, `http://www.zilog.com`
3. Agarwal, A., Alon, N., Charikar, M.S.: Improved approximation for directed cut problems. In: Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, STOC 2007, pp. 671–680. ACM, New York (2007)
4. Alon, N., Spencer, J.H.: The Probabilistic Method, 2nd edn. Wiley Interscience, Hoboken (2000)
5. Bergner, P., Dahl, P., Engebretsen, D., O'Keefe, M.: Spill code minimization via interference region spilling. In: Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI 1997, pp. 287–295. ACM, New York (1997)
6. Bernstein, D., Golumbic, M., Mansour, Y., Pinter, R., Goldin, D., Krawczyk, H., Nahshon, I.: Spill code minimization techniques for optimizing compliers. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, PLDI 1989, pp. 258–263. ACM, New York (1989)
7. Briggs, P., Cooper, K.D., Torczon, L.: Coloring register pairs. ACM Lett. Program. Lang. Syst. 1, 3–13 (1992)
8. Charikar, M., Chekuri, C., Cheung, T.-Y., Dai, Z., Goel, A., Guha, S., Li, M.: Approximation Algorithms for Directed Steiner Problems. Journal of Algorithms 33(1), 73–91 (1999)
9. Cheriyan, J., Karloff, H.J., Rabani, Y.: Approximating directed multicuts. Combinatorica 25(3), 251–269 (2005)
10. Cooper, K., Dasgupta, A., Eckhardt, J.: Revisiting Graph Coloring Register Allocation: A Study of the Chaitin-Briggs and Callahan-Koblenz Algorithms. In: Ayguadé, E., Baumgartner, G., Ramanujam, J., Sadayappan, P. (eds.) LCPC 2005. LNCS, vol. 4339, pp. 1–16. Springer, Heidelberg (2006)
11. Dinur, I., Guruswami, V., Khot, S., Regev, O.: A new multilayered PCP and the hardness of Hypergraph Vertex Cover. In: STOC 2003: Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing, pp. 595–601. ACM, New York (2003)
12. Falk, H.: Wcet-aware register allocation based on graph coloring. In: 46th ACM/IEEE, Design Automation Conference, DAC 2009, pp. 726–731 (2009)
13. Fu, C., Wilken, K.: A faster optimal register allocator. In: Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO, vol. 35, pp. 245–256. IEEE Computer Society Press, Los Alamitos (2002)
14. Garg, N., Vazirani, V.V., Yannakakis, M.: Multiway cuts in directed and node weighted graphs. In: Proceedings of the 21st International Colloquium on Automata, Languages and Programming (1994)
15. Gupta, A.: Improved results for directed multicut. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2003, pp. 454–455. Society for Industrial and Applied Mathematics, Philadelphia (2003)
16. Jansen, K.: The allocation problem in hardware design. Discrete Applied Mathematics 43(1), 37–46 (1993)
17. Jansen, K., Reiter, J.: An approximation algorithm for the register allocation problem. Integration 25(2), 89–102 (1998)
18. Kannan, S., Proebsting, T.A.: Proebsting. Register allocation in structured programs. J. Algorithms 29(2), 223–237 (1998)

19. Koes, D., Goldstein, S.C.: A progressive register allocator for irregular architectures. In: Proceedings of the International Symposium on Code Generation and Optimization, CGO 2005, pp. 269–280. IEEE Computer Society, Washington, DC, USA (2005)
20. Koes, D.R., Goldstein, S.C.: A global progressive register allocator. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, pp. 204–215. ACM, New York (2006)
21. Koes, D.R., Goldstein, S.C.: Register allocation deconstructed. In: Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2009, pp. 21–30. ACM, New York (2009)
22. Li, M., Xue, C.J., Liu, T., Zhao, Y.: Analysis and approximation for bank selection instruction minimization on partitioned memory architecture. In: Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (2010)
23. Pereira, F.M.Q., Palsberg, J.: Register allocation by puzzle solving. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2008, pp. 216–226. ACM, New York (2008)
24. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM 28(2), 202–208 (1985)
25. Smith, M.D., Ramsey, N., Holloway, G.: A generalized algorithm for graph-coloring register allocation. In: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI 2004, pp. 277–288. ACM, New York (2004)
26. Thorup, M.: All structured programs have small tree-width and good register allocation. Inf. Comput. 142(2), 159–181 (1998); Preliminary version in WG 1997
27. Zalamea, J., Llosa, J., Ayguad, E., Valero, M.: Modulo scheduling with integrated register spilling. In: Dietz, H. (ed.) LCPC 2001. LNCS, vol. 2624, pp. 115–130. Springer, Heidelberg (2003)
28. Zelikovsky, A.: A series of approximation algorithms for the acyclic directed Steiner tree problem. Algorithmica 18 (1997)
29. Zhou, Y.: Hardness of register loading. Personal Communication (2010)
30. Zosin, L., Khuller, S.: On directed Steiner trees. In: SODA, pp. 59–63 (2002)

Connecting a Set of Circles
with Minimum Sum of Radii

Erin Wolf Chambers[1], Sándor P. Fekete[2], Hella-Franziska Hoffmann[2],
Dimitri Marinakis[3,4], Joseph S.B. Mitchell[5], Venkatesh Srinivasan[4],
Ulrike Stege[4], and Sue Whitesides[4]

[1] Department of Computer Science, Saint Louis University, USA
`echambe5@slu.edu`
[2] Algorithms Group, TU Braunschweig, Braunschweig, Germany
`{s.fekete,h-f.hoffmann}@tu-bs.de`
[3] Kinsol Research Inc., Duncan, BC, Canada
`dmarinak@kinsolresearch.com`
[4] Department of Computer Science, University of Victoria, Victoria, BC, Canada
`{sue,stege,venkat}@uvic.ca`
[5] Department of Applied Mathematics and Statistics, Stony Brook University, USA
`jsbm@ams.stonybrook.edu`

Abstract. We consider the problem of assigning radii to a given set of
points in the plane, such that the resulting set of circles is connected, and
the sum of radii is minimized. We show that the problem is polynomially
solvable if a connectivity tree is given. If the connectivity tree is unknown,
the problem is NP-hard if there are upper bounds on the radii and open
otherwise. We give approximation guarantees for a variety of polynomial-
time algorithms, describe upper and lower bounds (which are matching
in some of the cases), provide polynomial-time approximation schemes,
and conclude with experimental results and open problems.

Keywords: intersection graphs, connectivity problems, NP-hardness
problems, approximation, upper and lower bounds.

1 Introduction

We consider a natural geometric connectivity problem, arising from assigning
ranges to a set of center points. In a general graph setting, we are given a weighted
graph $G = (V, E)$. Each vertex $v \in V$ in the graph is assigned a radius r_v, and
two vertices v and w are connected by an edge f_{vw} in the connectivity graph
$H = (V, F)$, if the shortest-path distance $d(v, w)$ in G does not exceed the sum
$r_v + r_w$ of their assigned radii. In a geometric setting, V is given as a set of points
$P = \{p_1, \ldots, p_n\}$ in the plane, and the respective radii r_i correspond to circular
ranges: two points p_i, p_j have an edge f_{ij} in the connectivity graph, if their circles
intersect. The CONNECTED RANGE ASSIGNMENT PROBLEM (CRA) requires an
assignment of radii to P, such that the objective function $R = \sum_i r_i^\alpha, \alpha = 1$ is
minimized, subject to the constraint that H is connected.

Problems of this type have been considered before and have natural motivations from fields including networks, robotics, and data analysis, where ranges have to be assigned to a set of devices, and the total cost is given by an objective function that considers the sum of the radii of circles to some exponent α. The cases $\alpha = 2$ or 3 correspond to minimizing the overall power; an example for the case $\alpha = 1$ arises from scanning the corresponding ranges with a minimum required angular resolution, so that the scan time for each circle corresponds to its perimeter, and thus radius.

In the context of clustering, Doddi et al. [7], Charikar and Panigraphy [5], and Gibson et al. [9] consider the following problems. Given a set P of n points in a metric space, metric $d(i, j)$ and an integer k, partition P into a set of at most k clusters with minimum sum of a) cluster diameters, b) cluster radii. Thus, the most significant difference to our problem is the lack of a connectivity constraint. Doddi et al. [7] provide approximation results for a). They present a polynomial-time algorithm, which returns $O(k)$ clusters that are $O(\log(\frac{n}{k}))$-approximate. For a fixed k, transforming an instance into a min-cost set-cover problem instance yields a polynomial-time 2-approximation. They also show that the existence of a $(2 - \epsilon)$-approximation would imply $P = NP$. In addition, they prove that the problem in weighted graphs without triangle inequality cannot be efficiently approximated within any factor, unless $P = NP$. Note that every solution to b) is a 2-approximation for a). Thus, the approximation results can be applied to case a) as well. A greedy logarithmic approximation and a primal-dual based constant factor approximation for minimum sum of cluster radii is provided by Charikar and Panigraphy [5]. In a more geometric setting, Bilò et al. [3] provide approximation schemes for clustering problems.

Alt *et al.* [1] consider the closely related problem of selecting circle centers and radii such that a given set of points in the plane are covered by the circles. Like our work, they focus on minimizing an objective function based on $\sum_i r_i^\alpha$ and produce results specific to various values of α. The minimum sum of radii circle coverage problem (with $\alpha = 1$) is also considered by Lev-Tov and Peleg [10] in the context of radio networks. Again, connectivity is not a requirement.

The work of Clementi *et al.* [6] focuses on connectivity. It considers minimal assignments of transmission power to devices in a wireless network such that the network stays connected. In that context, the objective function typically considers an $\alpha > 1$ based on models of radio wave propagation. Furthermore, in the type of problem considered by Clementi *et al.* the connectivity graph is directed; *i.e.* the power assigned to a specific device affects its transmission range, but not its reception range. This is in contrast to our work in which we consider an undirected connectivity graph. See [8] for a collection of hardness results of different (directed) communication graphs.

Carmi et al. [4] prove that an Euclidean minimum spanning tree is a constant-factor approximation for a variety of problems including the *Minimum-Area Connected Disk Graph* problem, which equals our problem with the different objective of minimizing the *area* of the *union* of disks, while we consider minimizing the *sum* of the *radii* (or perimeters) of all circles.

In this paper we present a variety of algorithmic aspects of the problem. In Section 2 we show that for a given connectivity tree, an optimal solution can be computed efficiently. Section 3 sketches a proof of NP-hardness for the problem when there is an upper bound on the radii. Section 4 provides a number of approximation results in case there is no upper bound on the radii. In Section 5 we present a PTAS for the general case, complemented by experiments in Section 6. A concluding discussion with open problems is provided in Section 7.

2 CRA for a Given Connectivity Tree

For a given connectivity tree, our problem is polynomially solvable, based on the following observation.

Lemma 1. *Given a connectivity tree T with at least three nodes. There exists an optimal range assignment for T with $r_i = 0$ for all leaves p_i of T.*

Proof. Assume an optimal range assignment for T has a leaf $p_i \in P$ with radius $r_i > 0$. The circle C_i around p_i with radius r_i intersects circle C_j around p_i's parent p_j with radius r_j. Extending C_j to $r_j := \text{dist}(p_i, p_j)$ while setting $r_i := 0$ does not increase the solution value $R = \sum_{p_i \in P} r_i$. $\qquad\square$

Direct consequences of Lemma 1 are the following.

Corollary 1. *There is an optimal range assignment satisfying Lemma 1 and $r_j > 0$ for all $p_j \in P$ of height 1 in T (i.e., each p_j is a parent of leaves only).*

Corollary 2. *Consider an optimal range assignment for T satisfying Lemma 1. Further let $p_j \in P$ be of height 1 in T. Then $r_j \geq \max_{p_i \text{ is child of } p_j} \{dist(p_i, p_j)\}$.*

These observations allow a solution by dynamic programming. The idea is to compute the values for subtrees, starting from the leaves. Details are omitted.

Theorem 1. *For a given connectivity tree, CRA is solvable in $O(n)$.*

3 Range Assignment for Bounded Radii

Without a connectivity tree, and assuming an upper bound of ρ on the radii, the problem becomes NP-hard. In this extended abstract, we sketch a proof of NP-hardness for the graph version of the problem; for the geometric version, a suitable embedding (based on PLANAR 3SAT) can be used.

Theorem 2. *With radii bounded by some constant ρ, the problem CRA is NP-hard in weighted graphs.*

See Figure 1 for the basic construction. The proof uses a reduction from 3SAT. Variables are represented by closed "loops" at distance ρ that have two feasible connected solutions: auxiliary points ensure that either the odd or the even points in a loop get radius ρ. (In the figure, those are shown as bold black or
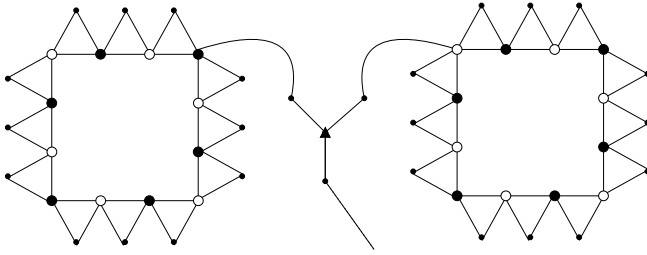
Fig. 1. Two variable gadgets connected to the same clause gadget. "True" and "False" vertices marked in bold white or black; auxiliary vertices are indicated by small dots; the clause vertex is indicated by a triangle. Connectivity edges are not shown.

white dots. The additional small dots form equilateral triangles with a pair of black and white dots, ensuring that one point of each thick pair needs to be chosen, so a minimum-cardinality choice consists of all black or all white within a variable.) Additional "connectivity" edges ensure that all variable gadgets are connected. Each clause is represented by a star-shaped set of four points that is covered by one circle of radius ρ from the center point. This circle is connected to the rest of the circles, if and only if one of the variable loop circles intersects it, which is the case if and only if there is a satisfying variable.

4   Solutions with a Bounded Number of Circles

A natural class of solutions arises when only a limited number of k circles may have positive radius. In this section we show that these *k-circle solutions* already yield good approximations; we start by giving a class of lower bounds.

Theorem 3. *A best k-circle solution may be off by a factor of* $(1 + \frac{1}{2^{k+1}-1})$.
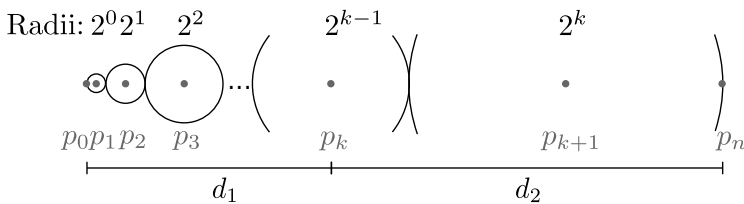
Fig. 2. A class of CRA instances that need $k+1$ circles in an optimal solution

Proof. Consider the example in Fig. 2. The provided solution r is optimal, as $R := \sum r_i = \frac{\text{dist}(p_0, p_n)}{2}$. Further, for any integer $k \geq 2$ we have $d_1 = 2 \cdot \sum_{i=0}^{k-2} 2^i + 2^{k-1} < 2 \cdot 2^k + 2^{k-1} = d_2$. So the radius r_{k+1} cannot be changed in an optimal solution. Inductively, we conclude that exactly $k+1$ circles are needed. Because we only consider integer distances, a best k-circle solution has cost $R_k \geq R + 1$, i.e., $\frac{R_k}{R} \geq 1 + \frac{1}{2^{k+1}-1}$. $\square$

In the following we give some good approximation guarantees for CRA using one or two circles.

Lemma 2. *Let $\mathcal{P}$ a longest (simple) path in an optimal connectivity graph, and let e_m be an edge in $\mathcal{P}$ containing the midpoint of $\mathcal{P}$. Then $\sum r_i \geq \max\{\frac{1}{2}|\mathcal{P}|, |e_m|\}$.*

This follows directly from the definition of the connectivity graph which for any edge $e = p_u p_v$ in $\mathcal{P}$ requires $r_u + r_v \geq |e|$.

Theorem 4. *A best 1-circle solution for CRA is a $\frac{3}{2}$-approximation, even in the graph version of the problem.*

Proof. Consider a longest path $\mathcal{P} = (p_0, \ldots, p_k)$ of length $|\mathcal{P}| = d_{\mathcal{P}}(p_0, \ldots, p_k) := \sum_{i=0}^{k-1} |p_i p_{i+1}|$ in the connectivity graph of an optimal solution. Let $R^* := \sum r_i^*$ be the cost of the optimal solution, and $e_m = p_i p_{i+1}$ as in Lemma 2. Let $\bar{d}_i := d_{\mathcal{P}}(p_i, \ldots, p_k)$ and $\bar{d}_{i+1} := d_{\mathcal{P}}(p_0, \ldots, p_{i+1})$. Then $\min\{\bar{d}_i, \bar{d}_{i+1}\} \leq \frac{\bar{d}_i + \bar{d}_{i+1}}{2} = \frac{d_{\mathcal{P}}(p_0,\ldots,p_i)+2|e_m|+d_{\mathcal{P}}(p_{i+1},\ldots,p_k)}{2} = \frac{|\mathcal{P}|}{2} + \frac{|e_m|}{2} \leq R^* + \frac{R^*}{2} = \frac{3}{2}R^*$. So one circle with radius $\frac{3}{2}R^*$ around the point in P that is nearest to the middle of path $\mathcal{P}$ covers P, as otherwise there would be a longer path. $\qquad\square$
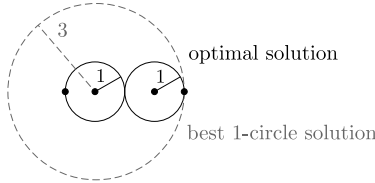
Fig. 3. A lower bound of $\frac{3}{2}$ for 1-circle solutions

Fig. 3 shows that this bound is tight. Using two circles yields an even better approximation factor.

Theorem 5. *A best 2-circle solution for CRA is a $\frac{4}{3}$-approximation, even in the graph version of the problem.*

Proof. Let $\mathcal{P} = (p_0, \ldots, p_k)$ be a longest path of length $|\mathcal{P}| = d_{\mathcal{P}}(p_0, \ldots, p_k) := \sum_{i=0}^{k-1} |p_i p_{i+1}|$ in the connectivity graph of an optimal solution with radii r_i^*. Then $R^* := \sum r_i^* \geq \frac{1}{2}|\mathcal{P}|$. We distinguish two cases; see Fig. 4.

Case 1. There is a point x on $\mathcal{P}$ at a distance of at least $\frac{1}{3}|\mathcal{P}|$ from both endpoints. Then there is a 1-circle solution that is a $\frac{4}{3}$-approximation.

Case 2. There is no such point x. Then two circles are needed. One of them is placed at a point in the first third of $\mathcal{P}$, and the other circle is placed at a point in the last third of $\mathcal{P}$. Let $e_m = p_i p_{i+1}$ be defined as in Lemma 2. Further, let $d_i := d_{\mathcal{P}}(p_0, \ldots, p_i)$, and let $d_{i+1} := d_{\mathcal{P}}(p_{i+1}, \ldots, p_k)$. Then $|e_m| = |\mathcal{P}| - d_i - d_{i+1}$ and $d_i, d_{i+1} < \frac{1}{3}|\mathcal{P}|$.
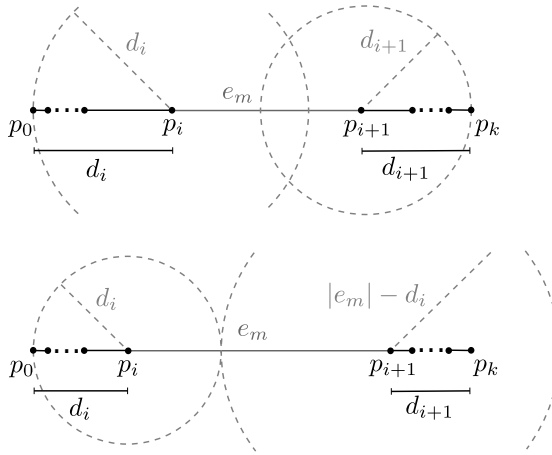
Fig. 4. The two $\frac{4}{3}$-approximate 2-circle solutions constructed in the proof of Theorem 5: (Top) case 2a; (bottom) case 2b

Case 2a. If $|e_m| < \frac{1}{2}|\mathcal{P}|$ then $d_i + d_{i+1} = |\mathcal{P}| - |e_m| > \frac{1}{2}|\mathcal{P}| > |e_m|$. Set $r_i := d_i$ and $r_{i+1} := d_{i+1}$, then the path is covered. Since $d_i, d_{i+1} < \frac{1}{3}|\mathcal{P}|$ we have $r_i + r_{i+1} = d_i + d_{i+1} < \frac{2}{3}|\mathcal{P}| \leq \frac{4}{3}R^*$ and the claim holds.

Case 2b. Otherwise, if $|e_m| \geq \frac{1}{2}|\mathcal{P}|$ then $d_i + d_{i+1} \leq \frac{1}{2}|\mathcal{P}| \leq |e_m|$. Assume $d_i \geq d_{i+1}$. Choose $r_i := d_i$ and $r_{i+1} := |e_m| - d_i$. As $d_{i+1} \leq |e_m| - d_i$ the path $\mathcal{P}$ is covered and $r_i + r_{i+1} = d_i + (|e_m| - d_i) = |e_m|$, which is the lower bound and thus the range assignment is optimal. □

If all points of P lie on a straight line, the approximation ratio for two circles can be improved.

Lemma 3. *Let P be a subset of a straight line. Then there is a non-overlapping optimal solution, i.e., one in which all circles have disjoint interior.*

Proof. An arbitrary optimal solution is modified as follows. For every two overlapping circles C_i and C_{i+1} with centers p_i and p_{i+1}, we decrease r_{i+1}, such that $r_i + r_{i+1} = \text{dist}(p_i, p_{i+1})$, and increase the radius of C_{i+2} by the same amount. This can be iterated, until there is at most one overlap at the outermost circle C_j (with C_{j-1}). Then there must be a point p_{j+1} on the boundary of C_j: otherwise we could shrink C_j contradicting optimality. Decreasing C_j's radius r_j by the overlap l and adding a new circle with radius l around p_{j+1} creates an optimal solution without overlap. □

Theorem 6. *Let P a subset of a straight line g. Then a best 2-circle solution for CRA is a $\frac{5}{4}$-approximation.*

Proof. According to Lemma 3 we are, w.l.o.g., given an optimal solution with non-overlapping circles. Let p_0 and p_n be the outermost intersection points of the optimal solution circles and g. W.l.o.g., we may further assume $p_0, p_n \in P$ and
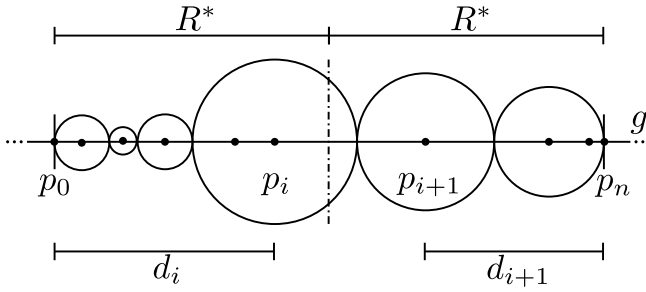
Fig. 5. A non-overlapping optimal solution

$R^* := \sum r_i = \frac{\text{dist}(p_0, p_n)}{2}$ (otherwise, we can add the outermost intersection point of the outermost circle and g to P, which may only improve the approximation ratio). Let p_i denote the rightmost point in P left to the middle of $\overline{p_0 p_n}$ and let p_{i+1} its neighbor on the other half. Further, let $d_i := \text{dist}(p_0, p_i)$, $d_{i+1} := \text{dist}(p_{i+1}, p_n)$ (See Fig. 5). Assume, $d_i \geq d_{i+1}$. We now give $\frac{5}{4}$-approximate solutions using one or two circles that cover $\overline{p_0 p_n}$.

Case 1. If $\frac{3}{4}R^* \leq d_i$ then $\frac{5}{4}R^* \geq 2R^* - d_i = \text{dist}(p_i, p_n)$. Thus, the solution consisting of exactly one circle with radius $2R^* - d_i$ centered at p_i is sufficient.

Case 2. If $\frac{3}{4}R^* > d_i \geq d_{i+1}$ we need two circles to cover $\overline{p_0 p_n}$ with $\frac{5}{4}R^*$.
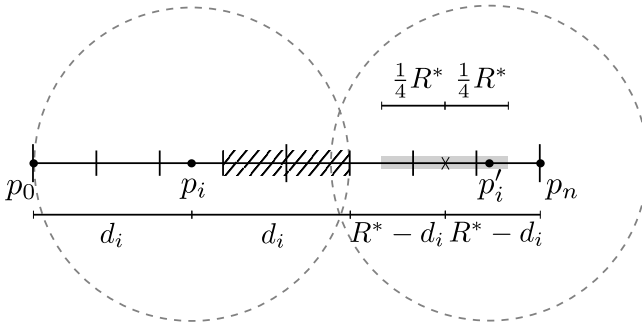
Fig. 6. A $\frac{5}{4}$-approximate 2-circle solution with $d_i < \frac{3}{4}R^*$. The cross marks the position of the optimal counterpart p_i^* to p_i and the grey area sketches A_i.

Case 2a. The point p_i could be a center point of an optimal two-circle solution if there was a point p_i^* with $\text{dist}(C_i, p_i^*) = \text{dist}(p_i^*, p_n) = R^* - d_i$. So in case there is a $p_i' \in P$ that lies in a $\frac{1}{4}R^*$-neighborhood of such an optimal p_i^* we get $\text{dist}(C_i, p_i'), \text{dist}(p_i', p_n) \leq R^* - d_i + \frac{1}{4}R^*$ (see Fig. 6). Thus, $r(p_i) := d_i, r(p_i') := R^* - d_i + \frac{1}{4}R^*$ provides a $\frac{5}{4}$-approximate solution.

Case 2b. Analogously to Case 2a, there is a point $p_{i+1}' \in P$ within a $\frac{1}{4}R^*$-range of an optimal counterpart to p_{i+1}. Then we can take $r(p_{i+1}) := d_{i+1}$, $r(p_{i+1}') := R^* - d_{i+1} + \frac{1}{4}R^*$ as a $\frac{5}{4}$-approximate solution.

Case 2c. Assume that there is neither such a p_i' nor such a p_{i+1}'. Because d_i, d_{i+1} are in $(\frac{1}{4}R^*, \frac{3}{4}R^*)$, we have $\frac{1}{4}R^* < R^* - d_j < \frac{3}{4}R^*$ for $j = i, i+1$, which implies that there are two disjoint areas A_i, A_{i+1}, each with diameter equal to $\frac{1}{2}R^*$ and excluding all points of P. Because p_i, the rightmost point on the left half of $\overline{p_0 p_n}$, has a greater distance to A_i than to p_0, any circle around a point on the left could only cover parts of both A_i and A_{i+1} if it has a greater radius than its distance to p_0. This contradicts the assumption that p_0 is a leftmost point of a circle in an optimal solution. The same applies to the right-hand side. Thus, $A_i \cup A_{i+1}$ must contain at least one point of P, and therefore one of the previous cases leads to a $\frac{5}{4}$-approximation.   □

optimal solution   best 2-circle solution
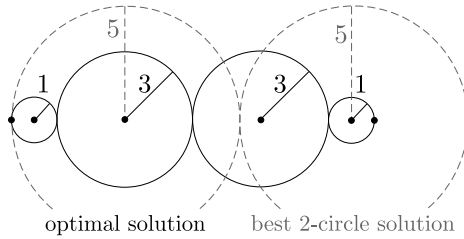
Fig. 7. A lower bound of $\frac{5}{4}$ for 2-circle solutions

Fig. 7 shows that the bound is tight. We believe that this is also the worst case when points are *not* on a line. Indeed, the solutions constructed in the proof of Theorem 6 cover a longest path $\mathcal{P}$ in an optimal solution for a general P. If this longest path consists of at most three edges, $p_i(=: p_{i+1}')$ and $p_{i+1}(=: p_i')$ can be chosen as circle centers, covering all of P. However, if $\mathcal{P}$ consists of at least four edges, a solution for the diameter may produce two internal non-adjacent center points that do not necessarily cover all of P.

5   Polynomial-Time Approximation Schemes

We now consider the problem in which each of the n points of $P = \{p_1, \ldots, p_n\}$ has an associated upper bound, $\bar{r}_i$, on the radius r_i that can be assigned to p_i.

5.1   Unbounded Radii

We begin with the case in which $\bar{r}_i = \infty$, for each i. Consider an optimal solution, with radius r_i^* associated with input point p_i. We first prove a structure theorem that allows us to apply the m-guillotine method to obtain a PTAS. The following simple lemma shows that we can round up the radii of an optimal solution, at a small cost to the objective function:

Lemma 4. *Let $R^* = \sum_i r_i^*$ be the sum of radii in an optimal solution, $\mathcal{D}^*$. Then, for any fixed $\epsilon > 0$, there exists a set, $\mathcal{D}_m$, of n circles of radii r_i centered on points p_i, such that (a). $r_i \in \mathcal{R} = \{D/mn, 2D/mn, \ldots, D\}$, where D is the diameter of the input point set P and $m = \lceil 2/\epsilon \rceil$; and (b). $\sum_i r_i \leq (1 + \epsilon)R^*$.*

Disks centered at the points P of radii in the set $\mathcal{R} = \{D/mn, 2D/mn, \ldots, D\}$ will be referred to as $\mathcal{R}_{\epsilon,P}$-*circles*, or $\mathcal{R}$-*circles*, for short, with the understanding that ϵ and P will be fixed throughout our discussion. Consider the arrangement of all $\mathcal{R}$-circles. We let $\mathcal{I}_x$ (resp., $\mathcal{I}_y$) denote the x-coordinates of the left/right (resp., y-coordinates of the top/bottom) extreme points of these circles. (Specifically, $\mathcal{I}_x = \{x_{p_i} \pm j(D/mn) : 1 \leq i \leq n, 0 \leq j \leq mn\bar{r}_i/D\}$ and $\mathcal{I}_y = \{y_{p_i} \pm j(D/mn) : 1 \leq i \leq n, 0 \leq j \leq mn\bar{r}_i/D\}$.)

We say that a set $\mathcal{D}$ of n $\mathcal{R}$-circles is m-*guillotine* if the bounding box, $BB(\mathcal{D})$, of $\mathcal{D}$ can be recursively partitioned into a rectangular subdivision by axis-parallel "m-perfect cuts" that are defined by coordinates $\mathcal{I}_x$ and $\mathcal{I}_y$, with the finest subdivision consisting of a partition into rectangular faces each of which has no circle of $\mathcal{D}$ strictly interior to it. An axis-parallel cut line ℓ is m-*perfect* with respect to $\mathcal{D}$ and a rectangle ρ if ℓ intersects at most $2m$ circles of $\mathcal{D}$ that have a nonempty intersection with ρ.

Key to our method is a structure theorem, which shows that we can transform an arbitrary set $\mathcal{D}$ of circles centered on points P, having a connected union and a sum of radii R, into an m-guillotine set of $\mathcal{R}$-circles, $\mathcal{D}_m$, having sum of radii at most $(1 + \epsilon)R^*$. More specifically, we show (proof deferred to the full paper):

Theorem 7. *Let $\mathcal{D}$ be a set of circles of radii r_i centered at points $p_i \in P$, such that the union of the circles is connected. Then, for any fixed $\epsilon > 0$, there exists an m-guillotine set $\mathcal{D}_m$ of n $\mathcal{R}$-circles such that the union of the circles $\mathcal{D}_m$ is connected and the sum of the radii of circles of $\mathcal{D}_m$ is at most $(1 + (C/m)) \sum_i r_i$. Here, $m = \lceil 1/\epsilon \rceil$ and C is a constant.*

A detailed proof can be found in the full version of the paper.

We now give an algorithm to compute a minimum-cost (sum of radii) m-guillotine set of $\mathcal{R}$-circles whose union is connected. The algorithm is based on dynamic programming. A subproblem is specified by a rectangle, ρ, with x- and y-coordinates among the sets $\mathcal{I}_x$ and $\mathcal{I}_y$, respectively, of discrete coordinates. The subproblem includes specification of *boundary information*, for each of the four sides of ρ. Specifically, the boundary information includes: (i) $O(m)$ "portal circles", which are $\mathcal{R}$-circles intersecting the boundary, $\partial\rho$, of ρ, with at most $2m$ circles specified per side of ρ; and, (ii) a connection pattern, specifying which subsets of the portal circles are required to be connected within ρ. There are a polynomial number of subproblems, for any fixed m. For a given subproblem, the dynamic program optimizes over all (polynomial number of) possible cuts (horizontal at $\mathcal{I}_y$-coordinates or vertical at $\mathcal{I}_x$-coordinates), and choices of up to $2m$ $\mathcal{R}$-circles intersecting the cut bridge, along with all possible compatible connection patterns for each side of the cut. The result is an optimal m-guillotine set of $\mathcal{R}$-circles such that their union is connected and the sum of the radii is minimum possible for m-guillotine sets of $\mathcal{R}$-circles. Since we know, from the structure theorem, that an optimal set of circles centered at points P can be converted into an m-guillotine set of $\mathcal{R}$-circles centered at points of P, whose union is connected, and we have computed an optimal such structure, we know that

the circles obtained by our dynamic programming algorithm yield an approximation to an optimal set of circles. In summary, we have shown the following result:

Theorem 8. *There is a PTAS for the min-sum radius connected circle problem with unbounded circle radii.*

5.2 Bounded Radii

We now address the case of bounded radii, in which circle i has a maximum allowable radius, $\bar{r}_i < \infty$. The PTAS given above relied on circle radii being arbitrarily large, so that we could increase the radius of a single circle to cover the entire m-span segment. A different argument is needed for the case of bounded radii.

We obtain a PTAS for the bounded radius case, if we make an additional assumption: that for any segment pq there exists a connected set of circles, centered at points of $p_i \in P$ and having radii $r_i \leq \bar{r}_i$, such that p and q each lie within the union of the circles and the sum of the radii of the circles is $O(|pq|)$.

Here, we only give a sketch of the method, indicating how it differs from the unbounded radius case. The PTAS method proceeds as above in the unbounded radius case, except that we now modify the proof of the structure theorem by replacing each m-span bridge $a_m b_m$ by a shortest connected path of $\mathcal{R}$-circles. We know, from our additional assumption, that the sum of the radii along such a shortest path is $O(|a_m b_m|)$, allowing the charging scheme to proceed as before. The dynamic programming algorithm changes some as well, since now the sub-problem specification must include the "bridging circle-path", which is specified only by its first and last circle (those associated with the bridge endpoints a_m and b_m); the path itself, which may have complexity $\Omega(n)$, is implicitly specified, since it is the shortest path (which we can assume to be unique, since we can specify a lexicographic rule to break ties).

Theorem 9. *There is a PTAS for the min-sum radius connected circle problem with bounded circle radii, assuming that for any segment pq, with p and q within feasible circles, there exists a (connected) path of feasible circles whose radii are $O(|pq|)$.*

6 Experimental Results

It is curious that even in the worst case, a one-circle solution is close to being optimal. This is supported by experimental evidence. In order to generate random problem instances, we considered different numbers of points uniformly distributed in a 2D circular region. For each trial considering a single distribution of points, we enumerated all possible spanning trees using the method described in [2], and recorded the optimal value with the algorithm mentioned in Section 2. This we compared with the best one-circle solution; as shown in Fig. 8, the latter seems to be an excellent heuristic choice. These results were obtained in several hours using an i7 PC.
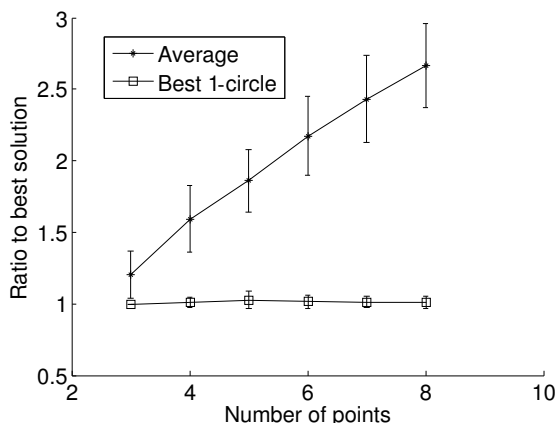
Fig. 8. Ratios of the average over all enumerated trees and of the best 1-circle tree to the optimal $\sum r_i$. Results were averaged over 100 trials for each number.

7   Conclusion

A number of open problems remain. One of the most puzzling is the issue of complexity in the absence of upper bounds on the radii. The strong performance of the one-circle solution (and even better of solutions with higher, but limited numbers of circles), and the difficulty of constructing solutions for which the one-circle solution is not optimal strongly hint at the possibility of the problem being polynomially solvable. Another indication is that our positive results for one or two circles only needed triangle inequality, i.e., they did not explicitly make use of geometry.

One possible way may be to use methods from linear programming: modeling the objective function and the variables by linear methods is straightforward; describing the connectivity of a spanning tree by linear cut constraints is also well known. However, even though separating over the exponentially many cut constraints is polynomially solvable (and hence optimizing over the resulting polytope), the overall polytope is not necessarily integral. On the other hand, we have been unable to prove NP-hardness without upper bounds on the radii, even in the more controlled context of graph-induced distances. Note that some results were obtained by means of linear programming: the tight lower bound for 2-circle solutions (shown in Fig. 7) was found by solving appropriate LPs.

Other open problems are concerned with the worst-case performance of heuristics using a bounded number of circles. We showed that two circles suffice for a $\frac{4}{3}$-approximation in general, and a $\frac{5}{4}$-approximation on a line; we conjecture that the general performance guarantee can be improved to $\frac{5}{4}$, matching the existing lower bound. Obviously, the same can be studied for k circles, for any fixed k; at this point, the best lower bounds we have are $\frac{7}{6}$ for $k = 3$ and $1 + \frac{1}{2^{k+1}}$ for general k. We also conjecture that the worst-case ratio $f(k)$ of a best k-circle solution approximates the optimal value arbitrarily well for large k, i.e., $\lim_{k \to \infty} f(k) = 1$.

Acknowledgments. A short version of this extended abstract appears in the informal, non-competitive European Workshop on Computational Geometry. This work was started during the 2009 Bellairs Workshop on Computational Geometry. We thank all other participants for contributing to the great atmosphere.

References

1. Alt, H., Arkin, E.M., Brönnimann, H., Erickson, J., Fekete, S.P., Knauer, C., Lenchner, J., Mitchell, J.S.B., Whittlesey, K.: Minimum-cost coverage of point sets by disks. In: Proc. 22nd ACM Symp. Comp. Geom. (SoCG), pp. 449–458 (2006)
2. Avis, D., Fukuda, K.: Reverse search for enumeration. Disc. Appl. Math. 65(1-3), 21–46 (1996)
3. Bilò, V., Caragiannis, I., Kaklamanis, C., Kanellopoulos, P.: Geometric clustering to minimize the sum of cluster sizes. In: Brodal, G.S., Leonardi, S. (eds.) ESA 2005. LNCS, vol. 3669, pp. 460–471. Springer, Heidelberg (2005)
4. Carmi, P., Katz, M.J., Mitchell, J.S.B.: The minimum-area spanning tree problem. Comput. Geom. Theory Appl. 35, 218–225 (2006)
5. Charikar, M., Panigrahy, R.: Clustering to minimize the sum of cluster diameters. J. Comput. Syst. Sci. 68, 417–441 (2004)
6. Clementi, A.E., Penna, P., Silvestri, R.: On the power assignment problem in radio networks. Mobile Networks and Applications 9(2), 125–140 (2004)
7. Doddi, S., Marathe, M.V., Ravi, S.S., Taylor, D.S., Widmayer, P.: Approximation algorithms for clustering to minimize the sum of diameters. Nordic J. of Computing 7, 185–203 (2000)
8. Fuchs, B.: On the hardness of range assignment problems. Networks 52(4), 183–195 (2008)
9. Gibson, M., Kanade, G., Krohn, E., Pirwani, I.A., Varadarajan, K.: On clustering to minimize the sum of radii. In: Proc. 19th ACM-SIAM Symp. Disc. Alg. (SODA), pp. 819–825 (2008)
10. Lev-Tov, N., Peleg, D.: Polynomial time approximation schemes for base station coverage with minimum total radii. Computer Networks 47(4), 489–501 (2005)

Streaming and Dynamic Algorithms for Minimum Enclosing Balls in High Dimensions

Timothy M. Chan and Vinayak Pathak

School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada
{tmchan,vpathak}@uwaterloo.ca

Abstract. At SODA'10, Agarwal and Sharathkumar presented a streaming algorithm for approximating the minimum enclosing ball of a set of points in d-dimensional Euclidean space. Their algorithm requires one pass, uses $O(d)$ space, and was shown to have approximation factor at most $(1 + \sqrt{3})/2 + \varepsilon \approx 1.3661$. We prove that the same algorithm has approximation factor less than 1.22, which brings us much closer to a $(1 + \sqrt{2})/2 \approx 1.207$ lower bound given by Agarwal and Sharathkumar.

We also apply this technique to the dynamic version of the minimum enclosing ball problem (in the non-streaming setting). We give an $O(dn)$-space data structure that can maintain a 1.22-approximate minimum enclosing ball in $O(d \log n)$ expected amortized time per insertion/deletion.

1 Introduction

In this paper, we study a fundamental and well-known problem in computational geometry: Given a set P of points in $\mathbb{R}^d$, find the ball with the smallest radius that contains all points in P. This is known as the *minimum enclosing ball* or the *1-center* problem and has various applications, for example, in clustering and facility location. We will not survey the many known exact algorithms for the problem, as the focus of the paper is on *approximation* algorithms in the streaming and the dynamic setting.

In the standard *streaming model*, we consider algorithms that are allowed one pass over the input and can store only a small (usually polylogarithmic) amount of information at any time, as points arrive one at a time. In low constant dimensions, it is not difficult to devise a streaming algorithm that computes a $(1+\varepsilon)$-approximation to the minimum enclosing ball using $O(1/\varepsilon^{(d-1)/2})$ space, by maintaining extreme points along a number of different directions. In fact, streaming techniques for ε-*kernels* [1,8,3,11] allow many other similar geometric optimization problems to be solved with approximation factor $1+\varepsilon$ using $1/\varepsilon^{O(d)}$ space. However, these techniques do not work well in high dimensions because of the exponential dependencies on d.

In high dimensions, there is a trivial streaming algorithm with approximation factor 2: fix the center of the ball B at an arbitrary input point p_0 (say the first point), and whenever a new point p arrives that lies outside B, expand B to include p while keeping the center unchanged (see Section 2.1). Zarrabi-Zadeh and Chan [12] gave a nontrivial analysis showing that another equally simple

streaming algorithm achieves approximation factor 3/2: whenever a new point p arrives that lies outside the current ball B, replace B by the smallest ball enclosing $B \cup \{p\}$. An advantage of these simple algorithms is that they avoid the exponential dependencies on d, using asymptotically the minimal amount of storage, namely, $O(d)$ space. (We assume that a unit of space can hold one coordinate value.)

Most recently, Agarwal and Sharathkumar [2] described a new streaming algorithm that also required just $O(d)$ space but with an even better approximation factor. They proved that the factor is upper-bounded by $(1+\sqrt{3})/2+\varepsilon \approx 1.3661$, where as usual, ε denotes an arbitrarily small positive constant. They also proved a complementary lower-bound result: any algorithm in the one-pass streaming model with space polynomially bounded in d has worst-case approximation factor at least $(1 + \sqrt{2})/2 > 1.207$. The gap between 1.3661 and 1.207 raises an interesting question of what the best constant could be. It also reveals our current lack of general understanding on high-dimensional geometric problems in the streaming model, as the minimum enclosing ball problem is one of the most basic and simplest to consider.

In this paper, we describe an improved upper bound of 1.22 for minimum enclosing ball in the streaming model. The improvement is actually achieved by the same algorithm as Agarwal and Sharathkumar's; our contribution lies in a better analysis of their algorithm. As intermediate solutions, we first present two simple proofs, one yielding upper bounds of $4/3 + \varepsilon \approx 1.333 \ldots$[1] and another yielding $16/13 + \varepsilon \approx 1.2307$. The 1.22 bound is obtained through more involved numerical calculations done with the assistance of a computer program.

In the second part of the paper, we investigate the *dynamic* version of the approximate minimum enclosing ball problem. Here, we are interested in data structures that support insertions and deletions of input points efficiently. Unlike in the streaming model, linear space is acceptable. As before, the problem is not difficult in low dimensions: one can maintain a $(1 + \varepsilon)$-approximation to the minimum enclosing ball in $O(1/\varepsilon^{(d-1)/2} \log n)$ time with a data structure using $O(n/\varepsilon^{(d-1)/2})$ space, by keeping track of extreme points along various directions. The $\log n$ factor in the update time can be reduced to constant in the word RAM model [9].

In the high-dimensional case, it is possible to dynamize the trivial factor-2 method by using a simple randomization trick (see Section 3.1), but we are not aware of any prior work on efficient dynamic data structures that achieve approximation factor smaller than 2 and avoid exponential dependency on d.

We show that Agarwal and Sharathkumar's approach, which was originally intended for streaming, can be applied to the dynamic problem as well, if combined in the right way with ideas from other known techniques: specifically, Bădoiu and Clarkson's static method for high-dimensional minimum enclosing ball [6], and

---

[1] Independent to our work, Agarwal and Sharathkumar (personal communication, Dec. 2010) have also found a proof of the 4/3 upper bound, which will appear in the journal version of their paper. Even compared against the 4/3 bound, our 1.22 bound is a substantial improvement.

Chan's dynamization strategy for low-dimensional ε-kernels [9]. The resulting data structure requires $O(dn)$ space and supports updates in $O(d \log n)$ expected amortized time. Our analysis of Agarwal and Sharathkumar's technique implies that the same 1.22 upper bound carries over to this dynamic data structure.

2 Streaming MEB

2.1 Preliminaries and Agarwal and Sharathkumar's Algorithm

Let P be a set of points in $\mathbb{R}^d$. We use $\text{MEB}(P)$ to denote the minimum enclosing ball of the set P. For a ball B, we use $r(B)$ and $c(B)$ to denote its radius and center respectively. αB stands for the ball with center at $c(B)$ and radius equal to $\alpha r(B)$.

A very simple factor-2 streaming algorithm for approximating the MEB works as follows. Let the first point be p_0. Find the point p_1 in P that is farthest away from p_0. This can be implemented by a one-pass streaming algorithm. Return the ball centered at p_0 of radius $\|p_0 p_1\|$. This ball clearly encloses P. The approximation factor is at most 2, since the MEB of P must enclose p_0 and p_1, and any ball that encloses p and q must have radius at least $\|p_0 p_1\|/2$.

If more than one pass is allowed, we can get better ratios. In particular, Bădoiu and Clarkson [6] (building on prior work by Bădoiu, Har-Peled, and Indyk [7]) proved that we can achieve an approximation factor of $1 + \varepsilon$ in $O(1/\varepsilon)$ passes. The algorithm works as follows. Pick a point $p_0 \in P$. Next, pick p_1 to be the point farthest from p_0 in P. In general, pick p_j to be the point farthest from $c(\text{MEB}(\{p_0, \ldots, p_{j-1}\}))$ in P. It was shown that after $\lceil 2/\varepsilon \rceil$ iterations, the set K of $O(1/\varepsilon)$ chosen points satisfies the following *coreset* property, which implies that $r(\text{MEB}(K))$ (computable by brute force) is a $(1 + \varepsilon)$-approximation:

Definition 1. *Given a set P of points in $\mathbb{R}^d$, an ε-coreset of P is a subset $K \subseteq P$ such that $P \subseteq (1 + \varepsilon)\text{MEB}(K)$.*

Using Bădoiu and Clarkson's algorithm as a subroutine, Agarwal and Sharathkumar [2] gave a streaming algorithm for finding a $((1 + \sqrt{3})/2 + \varepsilon)$-factor MEB of a given set of points. The algorithm works as follows. Let the first point in the input stream be its own coreset and call the coreset K_1. Next, as long as the new arriving points lie inside $(1 + \varepsilon)\text{MEB}(K_1)$, do nothing. Otherwise, if p_i denotes the new point, call Bădoiu and Clarkson's algorithm on the set $K_1 \cup \{p_i\}$. This gives a new coreset K_2. In general, maintain a sequence of coresets $\mathcal{K} = \langle K_1, \ldots, K_u \rangle$ and whenever a new point p_i arrives such that it does not lie in $(1 + \varepsilon)\text{MEB}(K_j)$ for any j, call Bădoiu and Clarkson's algorithm on the set $\bigcup_{j=1}^{u} K_j \cup \{p_i\}$. However, doing this might make the sequence $\mathcal{K}$ too large. To reduce space, whenever a new call to the subroutine is made, the algorithm also removes some of the previous K_i's when $r(\text{MEB}(K_i))$ is smaller than $O(\varepsilon)r(\text{MEB}(K_u))$. Agarwal and Sharathkumar proved that this removal process does not hurt the effectiveness of the data structure, and the following invariants are maintained, where $B_i = \text{MEB}(K_i)$:

(P1) For all i, $r(B_{i+1}) \geq (1 + \Omega(\varepsilon^2))r(B_i)$.
(P2) For all $i < j$, $K_i \subset (1 + \varepsilon)B_j$.
(P3) $P \subset \bigcup_{i=1}^{u}(1 + \varepsilon)B_i$.

The sequence $\mathcal{K}$ of coresets was called an ε-*blurred ball cover* in the paper. Property (P1) ensures that the number of coresets maintained at any time is $u = O(\log(1/\varepsilon))$. Since each coreset has size $O(1/\varepsilon)$, the total space is $O(d)$ for constant ε. Let $B = \text{MEB}(\bigcup_{i=1}^{u} B_i)$ (computable by brute force). Property (P3) ensures that $(1 + \varepsilon)B$ encloses P. Using property (P2), Agarwal and Sharathkumar proved that $r(B) \leq (\frac{1+\sqrt{3}}{2} + \varepsilon)\, r(\text{MEB}(P))$, thus giving a factor-1.366 algorithm for MEB in the streaming model. We show that in fact, the approximation factor is less than 1.22. The proof amounts to establishing the following (purely geometric) theorem:

Theorem 1. *Let $K_1, \ldots, K_u$ be subsets of a point set P in $\mathbb{R}^d$, with $B_i = \text{MEB}(K_i)$, such that $r(B_i)$ is increasing over i and property (P2) is satisfied for a sufficiently small $\varepsilon > 0$. Let $B = \text{MEB}(\bigcup_{i=1}^{u} B_i)$. Then $r(B) < 1.22\, r(\text{MEB}(P))$.*

2.2   An Improved Analysis

We will prove Theorem 1 in the next few subsections. First we need the following well-known fact, often used in the analysis of high-dimesional MEB algorithms:

Lemma 1 (the "hemisphere property"). *Let P be a set of points in $\mathbb{R}^d$. There is no hemisphere of $\text{MEB}(P)$ that does not contain a point from P. In other words, assuming the origin to be at the center of $\text{MEB}(P)$, for any unit vector v, there exists a point $p \in P$ such that p lies on the boundary of $\text{MEB}(P)$ and $v \cdot p \leq 0$.*

We introduce a few notations. Without loss of generality, let $r(B) = 1$ and $c(B)$ be the origin. Let u_i be the unit vector in the direction of the center of B_i and $\sigma_{ij} = u_i \cdot u_j$ be the inner product between the vectors u_i and u_j. Let us also write $r(B_i)$ simply as r_i and set $t_i = 1/(1 - r_i)$. Note that the $t_i \geq 1$ are increasing over i.

Lemma 2. *For all $i < j$ with $t_i \leq t_j < 10$ such that B_i and B_j touch ∂B,*

$$\sigma_{ij} \geq \frac{t_j}{t_i} - t_j + t_i - O(\varepsilon).$$

Proof. Let c, c_i, c_j be the centers of the balls B, B_i, B_j respectively. Figure 1 shows the projection of $B, (1+\varepsilon)B_i, B_j$ onto the plane formed by c, c_i, c_j. Let p be one of the points where $(1 + \varepsilon)B_j$ intersects B_i in this plane. Applying the cosine law to the triangle $c_i c_j c$, we get

$$\|c_i c_j\|^2 = \|cc_j\|^2 + \|cc_i\|^2 - 2\|cc_i\|\|cc_j\|\sigma_{ij}. \tag{1}$$

Next, we apply the hemisphere property to the ball $B_i = \text{MEB}(K_i)$. Choosing v to be the vector $c_j - c_i$, we deduce the existence of a point $q \in K_i$ such that
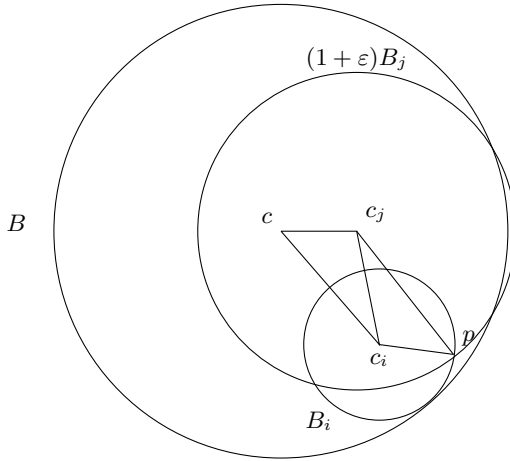
Fig. 1. Proof of Lemma 2

q lies on ∂B_i and $\angle c_j c_i q \geq \pi/2$. By property (P2) of the blurred ball cover, we know that $q \in K_i \subset (1+\varepsilon)B_j$. Since $\|c_i p\| = \|c_i q\|$ and $\|c_j p\| \geq \|c_j q\|$, we have $\angle c_j c_i p \geq \angle c_j c_i q \geq \pi/2$. This means

$$\|c_j p\|^2 \geq \|c_i c_j\|^2 + \|c_i p\|^2. \tag{2}$$

Substituting $\|c_j p\| = (1+\varepsilon)r_j, \|c_i p\| = r_i, \|c c_j\| = 1 - r_j, \|c c_i\| = 1 - r_i$ into (1) and (2) and combining them, we get

$$(1+\varepsilon)^2 r_j^2 \ \geq \ (1-r_j)^2 + (1-r_i)^2 - 2(1-r_i)(1-r_j)\sigma_{ij} + r_i^2.$$

Letting $s_i = 1 - r_i$ and $s_j = 1 - r_j$ and $t_i = 1/s_i$ and $t_j = 1/s_j$, we get

$$
\begin{aligned}
(1+\varepsilon)^2(1 - 2s_j + s_j^2) \ &\geq \ s_i^2 + s_j^2 - 2s_i s_j \sigma_{ij} + (1 - 2s_i + s_i^2) \\
\implies \qquad 2s_i s_j \sigma_{ij} \ &\geq \ 2s_i^2 - 2s_i + 2s_j - O(\varepsilon) \\
\implies \qquad \sigma_{ij} \ &\geq \ t_i - t_j + t_j/t_i - O(\varepsilon t_i t_j).
\end{aligned}
$$

(The assumption $t_i \leq t_j < 10$ allows us to rewrite $O(\varepsilon t_i t_j)$ as $O(\varepsilon)$.) $\qquad \square$

2.3   Proof of Factor 4/3

As a warm-up, in this subsection, we give a short proof of a weaker 4/3 upper bound on the constant in Theorem 1.

Let B_i be the largest ball that touches ∂B. Since B is the minimum enclosing ball of $\bigcup_{\ell=1}^{u} B_\ell$, by applying the hemisphere property to B with $v = u_i$ there must exist another ball B_j such that $\sigma_{ij} \leq 0$. Combining with Lemma 2, we get

$$\frac{t_i}{t_j} - t_i + t_j \ \leq \ O(\varepsilon) \ \implies \ t_i \ \geq \ \frac{t_j - O(\varepsilon)}{1 - 1/t_j}.$$

Since $t_j \geq 1$, the minimum value achievable by t_i that satisfies the above inequality can be easily found to be $4 - O(\varepsilon)$ (attained when $t_j \approx 2$). This translates to a minimum value of $3/4 - O(\varepsilon)$ for $r_i = 1 - 1/t_i$. Since $r(\mathrm{MEB}(P)) \geq r_i$ and $r(B) = 1$, this proves a version of Theorem 1 with the constant $4/3 + O(\varepsilon)$.

Remark: We have implicitly assumed that $t_j \leq t_i < 10$ when applying Lemma 2, but this is without loss of generality since $t_i \geq 10$ would imply $r_i > 0.99$ giving an approximation factor of ≈ 1.01.

2.4   Proof of Factor 16/13

In attempting to find an example where the $4/3$ bound might be tight, one could set $t_i = 4$ and $t_j = 2$, which implies $\sigma_{ij} \approx 0$ by Lemma 2, i.e., u_i and u_j are nearly orthogonal. However, by the hemisphere property, B would not be defined by the 2 balls B_i, B_j alone. This suggests that an improved bound may be possible by considering 3 balls instead of just 2, as we will demonstrate next.

Let B_i be the largest ball that touches ∂B, and B_j be the smallest ball that touches ∂B. Let $\alpha \geq 0$ be a parameter to be set later. By applying the hemisphere property to B with $v = u_i + \alpha u_j$, there must exist a k such that B_k touches ∂B and $u_k \cdot (u_i + \alpha u_j) \leq 0$. This means

$$\sigma_{ik} + \alpha \sigma_{jk} \leq 0. \tag{3}$$

Note that $t_j \leq t_k \leq t_i$. By Lemma 2, we get

$$\frac{t_i}{t_k} - t_i + t_k + \alpha \left(\frac{t_k}{t_j} - t_k + t_j \right) \leq O(\varepsilon)$$

$$\implies t_i \geq \frac{t_k + \alpha(t_k/t_j - t_k + t_j) - O(\varepsilon)}{1 - 1/t_k} \geq \frac{t_k + \alpha(2\sqrt{t_k} - t_k) - O(\varepsilon)}{1 - 1/t_k}.$$

The last step follows since the minimum of $t_k/x + x$ is achieved when $x = \sqrt{t_k}$ (e.g., by the A.M.–G.M. inequality). The final expression from the last step is in one variable, and can be minimized using standard techniques. Obviously, the minimum value depends on α. As it turns out, the best bound is achieved when $\alpha = 4/3$ and the minimum value is $16/3 - O(\varepsilon)$ (attained when $t_k \approx 4$). Thus, $t_i \geq 16/3 - O(\varepsilon)$, implying $r_i = 1 - 1/t_i \geq 13/16 - O(\varepsilon)$ and an upper bound of $16/13 + O(\varepsilon)$ in Theorem 1.

2.5   Proof of Factor 1.22

For our final proof of Theorem 1, the essential idea is to consider 4 balls instead of 3.

As before, let B_i be the largest ball that touches ∂B, and B_j be the smallest ball that touches ∂B. Choose a parameter $\alpha = \alpha(t_j) \geq 0$; unlike in the previous subsection, we find that making α dependent on t_j can help. By the hemisphere property, there must exist a B_k that touches ∂B while satisfying (3):

$\sigma_{ik} + \alpha\sigma_{jk} \leq 0$. By applying the hemisphere property once more with $v = \beta u_i + \gamma u_j + u_k$, for every $\beta, \gamma \geq 0$, there must exists a B_ℓ that touches ∂B satisfying

$$\beta\sigma_{i\ell} + \gamma\sigma_{j\ell} + \sigma_{k\ell} \leq 0. \tag{4}$$

We prove that with Lemma 2, these constraints force $t_i > 5.54546$, implying $r_i = 1 - 1/t_i > 0.8197$ and the claimed 1.22 bound in Theorem 1. We need a noticeably more intricate argument now, to cope with this more complicated system of inequalities. Assume $t_i \leq \tau := 5.54546$.

Note that 2 cases are possible: $t_j \leq t_k \leq t_\ell \leq t_i$ or $t_j \leq t_\ell \leq t_k \leq t_i$. We first eliminate the variable ℓ in (4). By (4), we have $\forall \beta, \gamma \geq 0$:

$$\left[\exists t_\ell \in [t_k, \tau] : \beta\left(\frac{\tau}{t_\ell} - \tau + t_\ell\right) + \gamma\left(\frac{t_\ell}{t_j} - t_\ell + t_j\right) + \frac{t_\ell}{t_k} - t_\ell + t_k \leq O(\varepsilon)\right] \vee$$

$$\left[\exists t_\ell \in [t_j, t_k] : \beta\left(\frac{\tau}{t_\ell} - \tau + t_\ell\right) + \gamma\left(\frac{t_\ell}{t_j} - t_\ell + t_j\right) + \frac{t_k}{t_\ell} - t_k + t_\ell \leq O(\varepsilon)\right].$$

Observe that in each of the 2 cases, multiplying the left hand side by t_ℓ yields a quadratic inequality in t_ℓ of the form $at_\ell^2 + bt_\ell + c \leq 0$. (The $O(\varepsilon)$ terms are negligible.) In the first case, $a = \beta + \gamma/t_j - \gamma + 1/t_k - 1$, $b = -\beta\tau + \gamma t_j + t_k$, and $c = \beta\tau$; in the second case, $a = \beta + \gamma/t_j - \gamma + 1$, $b = -\beta\tau + \gamma t_j - t_k$, and $c = \beta\tau + t_k$. The variable t_ℓ can then be eliminated by the following rule: $(\exists x \in [x_1, x_2] : ax^2 + bx + c \leq 0)$ iff $(ax_1^2 + bx_1 + c \leq 0) \vee (ax_2^2 + bx_2 + c \leq 0) \vee [(a \geq 0) \wedge (b^2 \geq 4ac) \wedge (2ax_1 \leq -b \leq 2ax_2)]$.

For β, we try two fine-tuned choices: (i) $\beta = -\gamma(\tau/t_j - \tau + t_j) - (\tau/t_k - \tau + t_k) + O(\varepsilon)$ (which is designed to make the above inequality tight at $t_\ell = \tau$), and (ii) a root β of the equation $b^2 = 4ac$ where a, b, c are the coefficients of the first quadratic inequality in the preceding paragraph (for fixed t_j, t_k, γ, this is a quadratic equation in β). As it turns out, these two choices are sufficient to derive the contradiction at the end.

Three variables γ, t_j, t_k still remain and the function $\alpha(t_j)$ has yet to be specified. At this point, it is best to switch to a numerical approach. We wrote a short C program to perform the needed calculations. For γ, we try a finite number of choices, from 0 to 1 in increments of 0.05, which are sufficient to derive the desired contradiction. For (t_j, t_k), we divide the two-dimensional search space into grid cells of side length 0.0005. For each grid cell that intersects $\{t_k \leq t_j\}$, we lower-bound the coefficients of the above quadratic inequalities over all (t_j, t_k) inside the cell, and attempt to obtain a contradiction with (4) by the strategy discussed above. If we are not able to get a contradiction for the cell this way, we turn to (3), which implies

$$\frac{\tau}{t_k} - \tau + t_k + \alpha\left(\frac{t_k}{t_j} - t_k + t_j\right) \leq O(\varepsilon);$$

from this inequality, we can generate an interval of α values that guarantees a contradiction in the (t_j, t_k) cell. We set $\alpha(t_j)$ to any value in the intersection of all α-intervals generated in the grid column of t_j. After checking that the intersection is nonempty for each grid column, the proof is complete.

Remarks: Our analysis of the system of inequalities derived from (3) and (4) is close to tight, as an example shows that these inequalities cannot yield a constant better than 1.219 regardless of the choice of the function $\alpha(t_j)$: Consider $t_i = 5.56621$ and $t_j = 2$. If $\alpha < 1.15$, pick $t_k = 2.67$; otherwise, $t_k = 5.08$. In either case, by solving a 2-variable, 100-constraint linear program in β and γ, one can verify that $\forall \beta, \gamma \geq 0$, there exists a t_ℓ from a discrete set of 100 uniformly spaced values in $[t_k, t_i]$ and $[t_j, t_k]$ such that the inequality derived from (4) is satisfied.

By choosing B_k and B_ℓ more carefully, one could add in the constraints $\sigma_{ij} \leq 0$, $\sigma_{ij} \leq \sigma_{ik}$, $\sigma_{ij} \leq \sigma_{i\ell}$, and $\sigma_{ik} + \alpha \sigma_{jk} \leq \sigma_{i\ell} + \alpha \sigma_{j\ell}$, though $t_j \leq t_k, t_\ell$ is no longer guaranteed; however, the system of inequalities becomes even harder to optimize, and we suspect that any improvements would be very small. Likewise, an analysis involving 5 or more balls does not seem to be worth the effort, until new ideas are found to simplify matters.

3   Dynamic MEB

3.1   Preliminaries and a Dynamic Coreset Technique

In this section, we investigate how to maintain the MEB of points in high dimensions if both insertions and deletions are allowed.

The simple factor-2 streaming algorithm from Section 2 can be modified to give a factor-2 dynamic algorithm as follows. In the preprocessing stage, pick any random point p_0 from the point set P uniformly and arrange the rest of the points in a priority queue with the key being the distance of the point from p_0. Let's call p_0 the "anchor point." To insert a new point, simply insert it into the priority queue. This takes time $O(\log n)$, where n is the number of points. The MEB returned at any time is the ball centered at p_0 and having a radius equal to the maximum key. To delete a point, remove it from the priority queue if the point being deleted is not the anchor point itself. Otherwise, rebuild the whole data structure by picking a new random anchor point p and arranging the rest in a priority queue. Since the choice of the anchor point is random, the probability with which it will be deleted is $1/n$. Therefore the expected cost of deletion is $\frac{1}{n}O(n \log n) + O(\log n) = O(\log n)$. The space used is linear. (The update time can be reduced to $O(1)$ in the word RAM model by using an approximate priority queue [9].)

To obtain a ratio better than 2, we modify a dynamization technique by Chan [9]. His method was originally for maintaining a different type of coresets (called ε-*kernels* [1]) which can be applied to many problems in low dimensions, such as computing minimum-volume (non-axis-aligned) bounding boxes. We outline his method here and point out the difficulty in adapting it for high-dimensional MEB.

The starting point is a simple constant-factor approximation algorithm for the minimum bounding box [1,4]. Pick a point $p_0 \in P$. This is the first anchor point. Next, let p_1 be the point farthest from p_0 in P. In general, pick point p_j to be the point farthest from $\text{aff}\{p_0, \ldots, p_{j-1}\}$, where $\text{aff } S$ denotes the affine hull

of a set S. The resulting anchor points $p_0, \ldots, p_d$ form a coreset whose minimum bounding box approximates the minimum bounding box of P to within $O(1)$ factor. The factor can be reduced to $1 + \varepsilon$ by building a grid along a coordinate system determined by the anchor points; the size of the coreset increases to $O(\varepsilon^{-d})$.

Now, to make this algorithm dynamic, the approach is to choose the anchor points in some random way and then whenever an anchor point is deleted, rebuild the whole data structure. Because of the randomness, the deleted point will be an anchor point with only a low probability. Thus instead of choosing p_j to be the point farthest from $\mathrm{aff}\{p_0, \ldots, p_{j-1}\}$, we pick p_j uniformly at random from the set A_j of $\alpha|P|$ farthest points from $\mathrm{aff}\{p_0, \ldots, p_{j-1}\}$ and discard A_j. Thus, after picking all the anchor points, we obtain a set $R = \bigcup_j A_j$ of all discarded points. Since R is not "served" by the anchor points chosen, we recurse on R. Since $|R|$ is a fraction less than $|P|$ if the constant α is sufficiently small, this gives us a collection of $O(\log n)$ coresets. The final coreset returned is the union of all of them. Insertions can be incorporated in a standard way, analogous to the *logarithmic method* [5].

The above technique cannot be directly applied to high-dimensional MEB because of the exponential dependency of the grid size on d. Also, with our weaker form of coreset from Definition 1 for MEB, the union of coresets of a collection of subsets is not necessarily a good coreset for the whole set.

3.2   A New Dynamic Algorithm

To modify the above technique to solve MEB, we propose two ideas. First, instead of the static constant-factor algorithm for minimum bounding box, we use Bǎdoiu and Clarkson's algorithm for MEB (see Section 2.1) as the starting point. A variant of Bǎdoiu and Clarkson's algorithm fits nicely here: instead of picking p_j to be the point in P farthest from $c(\mathrm{MEB}(\{p_1, \ldots, p_{j-1}\}))$, we look at the $\alpha|P|$ farthest points from $c(\mathrm{MEB}(\{p_1, \ldots, p_{j-1}\}))$ and pick p_j to be one of them at random with uniform probability. Secondly, instead of returning the union of the coresets found, we use Agarwal and Sharathkumar's blurred ball cover concept to get a good approximation factor. In order to satisfy property (P2), the key is to add all points from previous coresets found into the current point set. The precise details are given in pseudocode form in Algorithms 1–3.

The set P at the root level is initialized with $\widehat{K}_P = \emptyset$. Let u be the maximum number of levels of recursion. Note that $|\widehat{K}_P| \leq O(u/\varepsilon)$ at all levels. For $|P| \gg u/\varepsilon$, note that $|R|$ is a fraction less than $|P|$ if we make the constants α and δ sufficiently small (relative to ε). Thus, for c sufficiently large, u is bounded logarithmically in n. Let $\mathcal{K} = \langle K_1, \ldots, K_u \rangle$ denote the sequence of coresets K_P over all currently active point sets P, arranged from the root level to the last level. Let $B_i = \mathrm{MEB}(K_i)$. Then property (P3) is satisfied because of Bǎdoiu and Clarkson's algorithm. The trick of inserting coresets at earlier levels to the current set ensures property (P2). We can then use Theorem 1 to infer that $B = \mathrm{MEB}(\bigcup_{i=1}^{u} B_i)$ is a 1.22-approximation to $\mathrm{MEB}(P)$ for a sufficiently small ε.

Algorithm 1. P.preprocess()

---

if $|P| < c \log n$ **then**
    $K_P \leftarrow P$ and return
end if
$Q \leftarrow P$
$p_0 \leftarrow$ random point of P
for $j = 1, \ldots, \lceil 2/\varepsilon \rceil$ **do**
    $A_j \leftarrow$ the $\alpha|P|$ farthest points of Q from $c(\text{MEB}(\{p_0, \ldots, p_{j-1}\}))$
    $Q \leftarrow Q - A_j$
    $p_j \leftarrow$ a random point of A_j
end for
$K_P \leftarrow \{p_0, \ldots, p_{\lceil 2/\varepsilon \rceil}\}$ {an ε-coreset of Q by Bǎdoiu and Clarkson}
$\widehat{K}_R \leftarrow \widehat{K}_P \cup K_P$ {$\widehat{K}_P$ is a union of coresets at earlier levels}
$R \leftarrow (P - Q) \cup \widehat{K}_P$ {remember to add earlier coresets $\widehat{K}_P$ to the next level}
R.preprocess()
P.counter $\leftarrow \delta|P|$

Algorithm 2. P.delete(p), where $p \in P - \widehat{K}_P$

---

if $|P| < c \log n$ **then**
    remove p from P, reset $K_P \leftarrow P$, and return
end if
remove p from P
P.counter $\leftarrow P$.counter $- 1$
if P.counter $= 0$ or $p \in K_P$ **then**
    P.preprocess()   {rebuild all sets after current level}
end if
if $p \in R$ **then**
    R.delete(p)
end if

Algorithm 3. P.insert(p)

---

if $|P| < c \log n$ **then**
    insert p into P, reset $K_P \leftarrow P$, and return
end if
insert p into P
P.counter $\leftarrow P$.counter $- 1$
if P.counter $= 0$ **then**
    P.preprocess()   {rebuild all sets after current level}
end if
R.insert(p)

Instead of applying an exact algorithm to compute B, it is better to first compute a $(1 + \varepsilon)$-approximation B_i' to MEB(K_i) for each i, and then return a $(1 + \varepsilon)$-approximation to MEB($\bigcup_{i=1}^{u} B_i'$). (Note that every K_i has size $O(1/\varepsilon)$, except for the last set, which has size $O(\log n)$.) The latter can be done by a known approximation algorithm of Kumar, Mitchell, and Yildirim [10], which generalizes Bădoiu and Clarkson's algorithm for sets of balls. The time required is $O(du) = O(d \log n)$ (we ignore dependencies on ε from now on). It can be checked that the proof of Theorem 1 still goes through with B_i replaced by B_i', since the hemisphere property is still satisfied "approximately" for B_i'.

The **for** loop in P.preprocess() takes $O(dn)$ time for constant ε. Thus, the total preprocessing time is bounded by a geometric series summing to $O(dn)$. Space is $O(dn)$ as well. In the pseudocode for P.delete(), although the cost of the call to P.preprocess() is $O(d|P|)$, it can be shown [9] that the probability of deleting an anchor point $p \in K_i$ is $O(1/|P|)$ at any fixed level. Excluding the cost of computing B, the analysis of the expected amortized update time is essentially the same as in Chan's paper [9] and yields $O(d \log n)$. (The randomized analysis assumes that the update sequence is oblivious to the random choices made by the algorithm.) We conclude:

Theorem 2. *A factor-1.22 approximation of the* MEB *of points in $\mathbb{R}^d$ can be maintained with an algorithm that takes preprocessing time $O(dn \log n)$, uses space $O(dn)$ and takes expected amortized time $O(d \log n)$ for the updates.*

4   Final Remarks

Agarwal and Sharathkumar [2] have given an example showing that their streaming algorithm has approximation factor strictly greater than their $(1 + \sqrt{2})/2$ lower bound. Thus, if their lower bound is the right answer, a different streaming algorithm would be required. It would be interesting to investigate other high-dimensional geometric problems besides MEB in the streaming model. For example, before the recent developments on MEB, Chan [8] had given a $(5 + \varepsilon)$-factor streaming algorithm for the smallest enclosing cylinder problem with $O(d)$ space. Can the new techniques help in improving the approximation factor further?

On the dynamic MEB problem, the $(1 + \sqrt{2})/2$ lower bound on the approximation factor is not applicable, and the possibility of a $(1 + \varepsilon)$-factor algorithm with $d^{O(1)} \log n$ update time and $d^{O(1)} n$ space has not been ruled out. Also, $O(d \log n)$ may not necessarily be the final bound on the update time. For example, Chan [9] described an $O(1)$-factor dynamic algorithm with $O(d)$ expected amortized update time for the smallest enclosing cylinder problem in the word RAM model.

References

1. Agarwal, P.K., Har-Peled, S., Varadarajan, K.R.: Approximating extent measures of points. Journal of the ACM 51, 606–635 (2004)

2. Agarwal, P.K., Sharathkumar, R.: Streaming algorithms for extent problems in high dimensions. In: Proc. 21st ACM–SIAM Sympos. Discrete Algorithms, pp. 1481–1489 (2010)
3. Agarwal, P.K., Yu, H.: A space-optimal data-stream algorithm for coresets in the plane. In: Proc. 23rd Sympos. Comput. Geom., pp. 1–10 (2007)
4. Barequet, G., Har-Peled, S.: Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. J. Algorithms 38, 91–109 (2001)
5. Bentley, J.L., Saxe, J.B.: Decomposable searching problems I: Static-to-dynamic transformations. J. Algorithms 1, 301–358 (1980)
6. Bădoiu, M., Clarkson, K.L.: Smaller core-sets for balls. In: Proc. 14th ACM-SIAM Sympos. Discrete Algorithms, pp. 801–802 (2003)
7. Bădoiu, M., Har-Peled, S., Indyk, P.: Approximate clustering via core-sets. In: Proc. 34th ACM Sympos. Theory Comput., pp. 250–257 (2002)
8. Chan, T.M.: Faster core-set constructions and data stream algorithms in fixed dimensions. Comput. Geom. Theory Appl. 35, 20–35 (2006)
9. Chan, T.M.: Dynamic coresets. Discrete Comput. Geom. 42, 469–488 (2009)
10. Kumar, P., Mitchell, J.S.B., Yildirim, E.A.: Approximating minimum enclosing balls in high dimensions using core-sets. ACM J. Experimental Algorithmics 8, 1.1 (2003)
11. Zarrabi-Zadeh, H.: An almost space-optimal streaming algorithm for coresets in fixed dimensions. In: Halperin, D., Mehlhorn, K. (eds.) Esa 2008. LNCS, vol. 5193, pp. 817–829. Springer, Heidelberg (2008)
12. Zarrabi-Zadeh, H., Chan, T.M.: A simple streaming algorithm for minimum enclosing balls. In: Proc. 18th Canad. Conf. Comput. Geom., pp. 139–142 (2006)

New Algorithms for 1-D Facility Location and Path Equipartition Problems[*]

Danny Z. Chen and Haitao Wang[**]

Department of Computer Science and Engineering,
University of Notre Dame, Notre Dame, IN 46556, USA
{dchen,hwang6}@nd.edu

Abstract. We study the one-dimensional facility location problems. Given a set of n customers on the real line, each customer having a cost for setting up a facility at its position, and an integer k, we seek to find at most k of the customers to set up facilities for serving all n customers such that the total cost for facility set-up and service transportation is minimized. We consider several problem variations including k-median and k-coverage and a linear model. We also study a related path equipartition problem: Given a vertex-weighted path and an integer k, remove $k-1$ edges so that the weights of the resulting k sub-paths are as equal as possible. Based on new problem modeling and observations, we present improved algorithms for these problems over the previous work.

1   Introduction

A fundamental problem in location theory is to determine the optimal sites for building k facilities among n customers at given positions. It finds many applications, e.g., shape analysis, data compression, information retrieval, data mining, etc. In this paper, we consider the one-dimensional versions, i.e., all customers are on the real line.

Given a set of n sites, $P = \{p_1, \ldots, p_n\}$, sorted on the real line, with each site holding a customer, a typical location problem is to choose a subset F of P to set up facilities to serve all the customers. For convenience, we also use p_i to denote its coordinate on the line. Each p_i is associated with a cost c_i for setting up a facility at the location p_i. For each p_i, define $d(p_i, F)$ as the (Euclidean) distance from p_i to the nearest point in F, i.e., $d(p_i, F) = \min_{p_j \in F}\{|p_i - p_j|\}$; for any two points p_i and p_j, $d(p_i, p_j) = |p_i - p_j|$. For each customer p_i, define $f_i(d(p_i, F))$ as the transportation cost between p_i and the facility set, where $f_i(d)$ is a monotone nondecreasing function for the real distance d and $f_i(0) = 0$. We assume that given any distance d, the value of $f_i(d)$ can be computed in constant time. Let k be an integer with $1 \leq k \leq n$. The objective is to find $F \subseteq P$ such that $|F| \leq k$ and the sum $\sum_{p_i \in F} c_i + \sum_{i=1}^{n} f_i(d(p_i, F))$ is minimized. For any p_i, if $d(p_i, F) = d(p_i, p_j)$ and $p_j \in F$, we say p_i is *served* by the facility at p_j.

[*] This research was supported in part by NSF under Grant CCF-0916606.
[**] Corresponding author.

Table 1. Summary of the results, where $\mathcal{T} = \min\{n\sqrt{k \log n}, n2^{O(\sqrt{\log k \log \log n})}\}$

Problem	Previous	Ours	Ours is better when
k-median	$O(nk)$ [4,11]	$O(\mathcal{T} \log n)$	$k = \Omega(\log^3 n)$
Uniform k-median	$O(nk)$ [4,11]	$O(\mathcal{T})$	$k = \Omega(\log n)$
Infinity k-coverage	$O(nk)$ [11,23]	$O(\mathcal{T})$	$k = \Omega(\log n)$
Linear model	$O(nk)$ [11]	$O(\mathcal{T} \log^2 n)$	$k = \Omega(\log^5 n)$
L_∞ path equipartition	$O(nk \log k)$ [16]	$O(nk)$	always
L_d path equipartition	$O(nk)$ [12]	$O(\mathcal{T})$	$k = \Omega(\log n)$

As shown by Hassin and Tamir [11], the above definition unifies several location problem models and we call it the *general model*. If for each $1 \le i \le n$, $c_i = 0$ and $f_i(d) = a_i \cdot d$ for a constant $a_i \ge 0$, then the general model becomes the *k-median* problem; if all a_i's are equal, we call it *the uniform k*-median. The *k-coverage* problem is the following case: For each $1 \le i \le n$, there are two constants r_i and b_i, and $f_i(d) = 0$ if $d \le r_i$ and $f_i(d) = b_i$ otherwise. We refer to the special case when $b_i = +\infty$ for each $1 \le i \le n$ as the *infinity k-coverage*.

The general model is solvable in $O(n^2)$ time by dynamic programming (DP) [11]. It appears that one cannot do better even when $k = 1$, implying the general model algorithm may not help much for some specific applications. To characterize some special cases without losing much generality, we consider a *linear model*: For each $1 \le i \le n$, $f_i(d) = a_i \cdot d$ for a constant $a_i \ge 0$. Note that the linear model is different from the k-median in that every c_i is zero in the k-median.

All the above problem models have been studied before. We give improved algorithms in this paper. Let $\mathcal{T} = \min\{n\sqrt{k \log n}, n2^{O(\sqrt{\log k \log \log n})}\}$ throughout the paper. The results are summarized in Table 1.

Noted that while we do explore the Monge property of some of these problems, our approaches are not simple applications of the algorithm given by Aggarwal, Schieber, and Tokuyama [3] or the one given by Schieber [22], but hinged on nontrivial observations, new efficient data structures, and extensive modifications of the algorithms in [3,22] (while keeping their main scheme).

A related problem is the L_∞ *path equipartition*. Given a path P with vertices $\{p_1, \ldots, p_n\}$ and edges $\{(p_1, p_2), \ldots, (p_{n-1}, p_n)\}$, each vertex p_i has a weight $c_i \ge 0$. For any sub-path P' of P, let its weight $C(P')$ be the weight sum of all vertices in P'. Given an integer k with $1 \le k \le n$, the goal is to remove $k - 1$ edges from P such that the weights of the k resulting sub-paths $P_1, \ldots, P_k$ are as equal as possible, i.e., $\max_{1 \le i \le k} |C(P_i) - \mu|$ is minimized, where $\mu = C(P)/k$. The problem was solved in $O(nk \log k)$ time [16]. In this paper, we give an $O(nk)$ time solution (actually, the algorithm works for any value μ). If the objective is to minimize the value $\sum_{i=1}^{k} |C(P_i) - \mu|^d$ for a real $d \ge 1$, the problem becomes the L_d *path equipartition*, which was solved in $O(nk)$ time [12]. We present an $O(\mathcal{T})$ time solution, which is an improvement when $k = \Omega(\log n)$. Refer to [12,16] for applications of these problems.

1.1   Related Work

A problem closely related to k-median is the k-center problem, in which the maximum distance of any input point to a facility is minimized. It is well-known that the facility location, the k-median and the k-center problems in the plane or higher-dimensional space are NP-hard, and approximation algorithms have also been studied (see [1] for a survey). Many classical problems are special cases of these problems, e.g., the smallest enclosing circle problem [19], its weighted version [7] and discrete version [15], the Fermat-Weber problem [5], etc. Refer to [6] for many other variants of the facility location problem.

Most one-dimensional problems are solvable efficiently. The k-median was solved in $O(n^2 k)$ time in [17] by DP. By using the concave Monge property, an improved $O(nk)$ time algorithm was shown in [11]. Independently, an $O(nk)$ time algorithm for the k-median was given [4], using the same DP scheme with a somewhat different implementation. We are not aware of any faster algorithm specifically for the uniform k-median. In [11], the general model was solved in $O(n^2)$ time by DP with the concave Monge property; the k-coverage and its infinity case were solved in $O(n^2)$ and $O(nk)$ time, respectively. In [23], based on parametric search, a new k-coverage algorithm of $O(nk \log n)$ time was given and the infinity case was solved in $O(nk)$ time. In addition, an "asymmetric" k-median problem on a path is studied in [24], where Monge property is explored.

In the graph setting, for a tree, the k-center is solvable in $O(n)$ time [8], and both the k-coverage and k-median are solvable in $O(n^2 k)$ time [20].

The L_∞ equipartition can be solved in $O(n^2 k)$ time by DP. By exploring some properties, a shifting algorithm of $O(nk \log k)$ time was given [16]. If $\mu = 0$, the problem is solvable in $O(n)$ time by the algorithm in [8], which however does not work for $\mu \neq 0$. An $O(n)$ time algorithm was also given in [8] for the max-min partition on a tree. Refer to [18,21] for other min-max partition models. For the L_d equipartition (for any real value $d \geq 1$), an $O(nk)$ time algorithm is known [12] based on dynamic programming with the concave Monge property.

The rest of the paper is organized as follows. In Section 2, we discuss our problem modeling on the general model. Section 3 discusses the k-median problem. In Section 4, we present our algorithm for the infinity k-coverage problem. The algorithm for the linear model is given in Section 5. Our results on the L_∞ and L_d path equipartitions are presented in Section 6.

2   The Problem Modeling of the Facility Location

In this section, we reduce the general model to the problem of finding a shortest path with at most k edges in a DAG G of $O(n)$ vertices. We prove that the edge weights of G satisfy the concave Monge property. Note that this result is applicable to any problem in the general model, and particularly, to the k-median, the infinity k-coverage and the linear model.

Given a point set P in sorted order on the real line, an easy but critical observation is that there is an optimal solution for the general model, with $F \subseteq P$ as the facility set, such that for any $p_i \in F$, if $P_i \subseteq P$ is the set
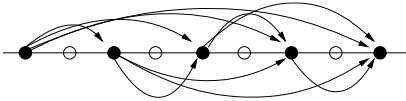
Fig. 1. The DAG G: The solid nodes are the vertices of G and the hollow ones are the customers
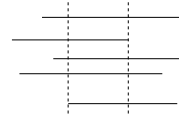
Fig. 2. The common intersection (between the two dashed lines) of a set of intervals

of customers served by p_i, then all customers in P_i are consecutive in P. Our problem modeling is based on this observation.

For the point set P, we build a weighted DAG $G = (V, E)$ as follows. The vertex set V contains $n + 1$ vertices $v_0, v_1, \ldots, v_n$ such that each vertex v_i corresponds to a point between p_i and p_{i+1} on the line (v_0 is to the left of p_1 and v_n is to the right of p_n). For any $0 \le i < j \le n$, we put a directed edge $e(i, j)$ from v_i to v_j in E. Fig. 1 shows an example. Denote by $\mathcal{P}(i + 1, j)$ the special sub-problem with the consecutive points $p_{i+1}, \ldots, p_j$ as the customers, for which we seek a single facility p_t with $i + 1 \le t \le j$ to provide service for all these customers such that the objective value $obj(t, i+1, j) = c_t + \sum_{l=i+1}^{j} f_l(d(p_l, p_t))$ is minimized. Define the weight of the edge $e(i, j)$ as this minimum objective value for $\mathcal{P}(i + 1, j)$, denoted by $w(i, j)$. We call an edge of a path in G a *link* of the path. As in [3,22], we call a path from v_0 to v_n in G a *diameter path*. Clearly, a feasible solution of the general model corresponds to a shortest diameter path in G with at most k links. Below, we will prove the edge weights of G satisfy the concave Monge property, i.e., for all $0 \le i + 1 < j < n$, $w(i, j) + w(i + 1, j + 1) \le w(i, j + 1) + w(i + 1, j)$ holds.

For each $t = i + 1, \ldots, j$, we define $w_l(t, i + 1) = \sum_{m=i+1}^{t} f_m(d(p_m, p_t))$ and $w_r(t, j) = \sum_{m=t+1}^{j} f_m(d(p_m, p_t))$; thus, we have $obj(t, i + 1, j) = c_t + w_l(t, i + 1) + w_r(t, j)$. We will use these notations throughout the paper.

Note that the previous work (e.g., [11]) observed the Monge property on the transportation costs, i.e., $\sum_{l=i}^{j} f_l(d(p_i, p_l))$. In contrast, the Monge property we explore is defined on the $w(i, j)$'s, which contain not only the transportation costs but also the costs for setting up facilities, and is more general.

Denote by $I(i, j)$ the index of the facility in an optimal solution for the special sub-problem $\mathcal{P}(i, j)$; if there are multiple optimal solutions, let $I(i, j)$ be the leftmost one. For convenience, we also use $I(i, j)$ to represent the corresponding point. Denote by $opt(i, j)$ the optimal solution with $I(i, j)$ as the facility. The following observation is self-evident.

Observation 1. *For any $i \le j_1 \le j_2 \le n$, $I(i, j_1) \le I(i, j_2)$ holds; for any $1 \le i_1 \le i_2 \le j$, $I(i_1, j) \le I(i_2, j)$ holds.*

The following lemma proves that the graph G is a concave Monge DAG, i.e., $w(i, j) + w(i + 1, j + 1) \le w(i, j + 1) + w(i + 1, j)$ holds for all $0 \le i + 1 < j < n$.

Lemma 1. *The graph G is a concave Monge DAG.*

Proof. Our goal is to prove the inequality $w(i, j) + w(i + 1, j + 1) \leq w(i, j + 1) + w(i + 1, j)$ holds for all $0 \leq i + 1 < j < n$.

Consider any two integers i and j with $0 \leq i+1 < j < n$. By Observation 1, we have $I(i+1, j) \leq I(i+1, j+1)$ and $I(i+2, j) \leq I(i+2, j+1)$. However, it can be either $I(i+1, j+1) \leq I(i+2, j)$ or $I(i+1, j+1) > I(i+2, j)$. Below, we prove that if $I(i+1, j+1) \leq I(i+2, j)$, then $w(i, j) + w(i+1, j+1) \leq w(i, j+1) + w(i+1, j)$ holds. The proof for the other case is omitted since it is quite similar.

If $I(i+1, j+1) \leq I(i+2, j)$, then since $I(i+2, j) \leq j$, we have $I(i+1, j+1) \leq j$, which means p_{j+1} is not the facility in the optimal solution $opt(i + 1, j + 1)$ for the sub-problem $\mathcal{P}(i + 1, j + 1)$. Thus, if we remove the customer p_{j+1} and its corresponding transportation cost from $opt(i+1, j+1)$, we can obtain a feasible solution for the sub-problem $\mathcal{P}(i + 1, j)$. Recall that $w(i, j + 1)$ is the objective value of $opt(i + 1, j + 1)$. Therefore, the value $w(i, j + 1) - f_{j+1}(d(p_{j+1}, I(i + 1, j + 1)))$ is the objective value of a feasible solution for $\mathcal{P}(i + 1, j)$. Since $w(i, j)$ is the objective value of an optimal solution for $\mathcal{P}(i + 1, j)$, it must be $w(i, j) \leq w(i, j + 1) - f_{j+1}(d(p_{j+1}, I(i + 1, j + 1)))$.

On the other hand, $w(i + 1, j)$ is the objective value of the optimal solution $opt(i+2, j)$ for the sub-problem $\mathcal{P}(i+2, j)$. Similarly, we can connect the customer p_{j+1} to the facility at $I(i + 2, j)$ in $opt(i + 2, j)$ to obtain a feasible solution for the sub-problem $\mathcal{P}(i + 2, j + 1)$. In other words, the value $w(i + 1, j) + f_{j+1}(d(p_{j+1}, I(i + 2, j)))$ is the objective value of a feasible solution for $\mathcal{P}(i + 2, j + 1)$. Since $w(i + 1, j + 1)$ is the objective value of an optimal solution for $\mathcal{P}(i+2, j+1)$, it must be $w(i+1, j+1) \leq w(i+1, j) + f_{j+1}(d(p_{j+1}, I(i+2, j)))$.

In addition, $I(i+1, j+1) \leq I(i+2, j) < j+1$ implies $d(p_{j+1}, I(i+1, j+1)) \geq d(p_{j+1}, I(i+2, j))$. Thus, $f_{j+1}(d(p_{j+1}, I(i+1, j+1))) \geq f_{j+1}(d(p_{j+1}, I(i+2, j)))$ holds. By combining all the above results, we have $w(i, j + 1) + w(i + 1, j) \geq w(i, j) + w(i + 1, j + 1)$.

The lemma thus follows.

By Lemma 1, a shortest diameter path with at most k links can be obtained efficiently. Suppose G' is a concave Monge DAG and any edge weight in G' can be obtained in $O(1)$ time. The algorithms in [3,22] can compute a shortest diameter path with *exactly* k links in G' in $O(\mathcal{T})$ time. Below, we present our $O(\mathcal{T})$ time algorithm, called *Algo-k-link*, for computing a shortest diameter path with *at most* k links in G'.

Denote by $D(k)$ the weight of the shortest diameter path with exactly k links in G'. Corollary 7 of [3] implies that any local minimum of the function $D(k)$ with variable k must also be the global minimum (a similar observation is also given in [22]). The algorithm *Algo-k-link* then works as follows. First, we compute a shortest diameter path in G' (without restricting the number of links), which takes $O(n)$ time [10,13,14]. If this path has at most k links, then we are done. Otherwise, we apply the algorithms in [3,22] to find a shortest diameter path with exactly k-link as our solution. The running time of *Algo-k-link* is dominated by the algorithms in [3,22], which is $O(\mathcal{T})$. Thus, we have the following result.

Lemma 2. *In $O(\mathcal{T} \cdot W)$ time, where $O(W)$ is the time for computing any edge weight in G, we can find a shortest diameter path with at most k links in G,*

*and consequently obtain an optimal solution for the general model with at most
k facilities.*

3   The k-Median Problem

In this section, we present our algorithm for the k-median problem. By Lemma 2,
it is sufficient to design an efficient data structure for answering $w(i,j)$ queries.

Recall that in the k-median problem, for each $1 \le i \le n$, we have $c_i = 0$ and
$f_i(d) = a_i \cdot d$ with $a_i \ge 0$ for any real distance d. To compute $w(i,j)$, we need
to find t with $i+1 \le t \le j$ that minimizes the value $obj(t, i+1, j)$, which is
$w_l(t, i+1) + w_r(t, j)$ (since $c_t = 0$), and the sought t is $I(i+1, j)$. The following
observation (related to the weighted median) is crucial.

Observation 2. *For any $1 \le i \le j \le n$, if there exists an index b with $i < b \le j$
such that $\sum_{t=i}^{b-1} a_t < \sum_{t=b}^{j} a_t$ and $\sum_{t=i}^{b} a_t \ge \sum_{t=b+1}^{j} a_t$, then $I(i+1, j) = b$;
otherwise, $I(i+1, j) = i$.*

As preprocessing, in $O(n)$ time, we compute the prefix sums $\sum_{t=1}^{i} a_t$ for all
$1 \le i \le n$. For any query $q(i,j)$, by Observation 2, we can compute $I(i,j)$ by
binary search in $O(\log n)$ time. For the uniform case, by Observation 2, it is easy
to see that $I(i,j) = \lceil (i+j)/2 \rceil$.

To compute the value of $w(i,j)$, we first compute $I(i+1,j)$ as described above.
Let $b = I(i+1, j)$. We then have $w(i,j) = \sum_{t=i+1}^{b-1}(p_b - p_t) \cdot a_t + \sum_{t=b}^{j}(p_t - p_b) \cdot a_t =
p_b \cdot (\sum_{t=i+1}^{b-1} a_t - \sum_{t=b}^{j} a_t) + \sum_{t=b}^{j}(p_t \cdot a_t) - \sum_{t=i+1}^{b-1}(p_t \cdot a_t)$. Therefore, with $O(n)$
time preprocessing, which computes $\sum_{t=1}^{i}(p_t \cdot a_t)$ for each $1 \le i \le n$, we can
obtain the value of $w(i,j)$ in $O(1)$ time (after $I(i+1,j)$ is known).

Therefore, with $O(n)$ time preprocessing, the value of any $w(i,j)$ can be ob-
tained in $O(\log n)$ time for the k-median, and in $O(1)$ time for its uniform case.
By Lemma 2, we have the following result.

Theorem 1. *The k-median and its uniform case are solvable in $O(\mathcal{T} \log n)$ time
and $O(\mathcal{T})$ time, respectively.*

4   The Infinity k-Coverage Problem

In this section, we present our algorithm for the infinity k-coverage problem.
By Lemma 2, it is sufficient to design an efficient data structure for answering
$w(i,j)$ queries. With $O(n)$ time preprocessing, our data structure computes any
$w(i,j)$ in $O(1)$ time. This data structure actually solves a more general problem
on intervals, and may find other applications as well.

Recall that in the infinity k-coverage problem, for each $1 \le i \le n$, $f_i(d) = 0$ if
$d \le r_i$, and $f_i(d) = +\infty$ otherwise. We define an interval $I_i = [\alpha_i, \beta_i]$ on the real
line for each $1 \le i \le n$, with $\alpha_i = p_i - r_i$ and $\beta_i = p_i + r_i$. To compute $w(i,j)$,
note that it is $obj(t, i+1, j) = c_t + w_l(t, i+1) + w_r(t, j)$ where $t = I(i+1, j)$.
The value of $w_l(t, i+1) + w_r(t, j)$ is either 0 or $+\infty$. We assume $c_i < +\infty$ for
each $1 \le i \le n$ (this assumption is only for the simplicity of the analysis, and
our algorithm still works when the assumption does not hold).

Observation 3. $w(i, j) < +\infty$ if and only if there exists a point $p_t \in P$ such that $i + 1 \le t \le j$ and $p_t \in \cap_{t=i+1}^{j} I_t$.

Observation 3 implies that to compute $w(i, j)$, it suffices to find a point $p_m \in \cap_{t=i+1}^{j} I_t$ with the smallest c_m and $i + 1 \le m \le j$. The index m is $I(i+1, j)$ and $w(i, j) = c_m$. We generalize our problem to the following problem, which may find other applications. Given a set P of n sorted points $p_1 \le p_2 \le \cdots \le p_n$ and a set $\mathcal{I}$ of n intervals $I_t = [\alpha_t, \beta_t]$ with $1 \le t \le n$ on the real line, such that each $p_t \in I_t$ (i.e., $\alpha_t \le p_t \le \beta_t$) and p_t has a weight c_t, for any query $q(i, j)$ with $1 \le i \le j \le n$, report a point p_m with the smallest c_m such that $i \le m \le j$ and $p_m \in \cap_{t=i}^{j} I_t$. We call it the *points in intervals* (PII) query problem.

Below, we build a data structure in $O(n)$ time, which answers any PII query in $O(1)$ time. To provide some intuition, we begin with a preliminary solution which answers each query in $O(1)$ time after an $O(n \log n)$ time preprocessing, and then show how to reduce the preprocessing time to $O(n)$.

For each $1 \le i \le n$, define a new interval $I'_i = [\alpha'_i, \beta'_i]$ which is the maximal interval such that $I'_i \subseteq I_i$ and each endpoint of I'_i is a point of P. Let $\mathcal{I}'$ be the set of all these new intervals. Note that for any $q(i, j)$, the query answer on $\mathcal{I}$ is the same as that on $\mathcal{I}'$. Our algorithm below processes such queries on $\mathcal{I}$.

We first compute $\mathcal{I}'$. For each i, to determine α'_i, we can use binary search to find t such that $p_{t-1} < \alpha_i \le p_t$, in $O(\log n)$ time, and set $\alpha'_i = p_t$. The value of β'_i is determined similarly. Therefore, computing $\mathcal{I}'$ takes $O(n \log n)$ time.

For a query $q(i, j)$, we first determine the interval $I^* = \cap_{t=i}^{j} I'_t$; if $I^* \ne \emptyset$, then we find a point $p_m \in I^*$ with $i \le m \le j$ whose weight c_m is the smallest. Let $I^* = [\alpha^*, \beta^*]$, where $\alpha^* = \max_{i \le t \le j} \alpha'_t$ and $\beta^* = \min_{i \le t \le j} \beta'_t$ (see Fig. 2). Note that $I^* \ne \emptyset$ if and only if $\alpha^* \le \beta^*$. Since all α'_t and β'_t are points in P, α^* and β^* are also points in P. Let the indices of the points α^* and β^* be i^* and j^*, respectively. It should be noted that since α^* is determined by the left endpoints of all intervals from i to j, $i^* < i$ is possible (i.e., if $\alpha'_t \le p_{i-1}$ for every t with $i \le t \le j$); similarly, $j < j^*$ is also possible. Thus, the point p_m whose weight c_m is $\min_{\max\{i^*, i\} \le t \le \min\{j^*, j\}} c_t$ is reported as our answer to the query $q(i, j)$.

To implement the above algorithm, we utilize the range-minima (resp., range-maxima) data structure [9], which, after an $O(n)$ time preprocessing on a given array $A[1 \ldots n]$ (not in any sorted order), can report in $O(1)$ time the smallest (resp., largest) element in the sub-array $A[i \ldots j]$ specified by any query (i, j). Let A be the array of α'_t's, i.e., $A[t] = \alpha'_t$ for each $1 \le t \le n$. Similarly, let B and C be the arrays of β'_t's and c_t's, respectively. We build a range-maxima data structure on A and a range-minima data structure on B, respectively. Then for each PII query $q(i, j)$, α^* and β^* can be obtained in $O(1)$ time. When computing an interval $I'_t = [\alpha'_t, \beta'_t]$, suppose $\alpha'_t = p_s$; then we store the index s together with α'_t. We do the similar thing for β'_t. Thus, once we obtain α^* and β^*, we also know i^* and j^*. We build a range-minima data structure on the array C, which allows us to find the sought point p_m in $O(1)$ time. In summary, after the set $\mathcal{I}'$ is computed in $O(n \log n)$ time, with additional $O(n)$ time preprocessing, we can answer each query $q(i, j)$ in $O(1)$ time.

The dominating part of the above preprocessing is for computing the new interval set $\mathcal{I}'$, which is $O(n \log n)$. To reduce the $O(n \log n)$ preprocessing time, we might need a faster way to compute $\mathcal{I}'$. However, it appears that computing the exact $\mathcal{I}'$ requires $\Omega(n \log n)$ time. To circumvent the difficulty, we choose not to compute $\mathcal{I}'$. Instead, we compute a set of *truncated intervals* in $O(n)$ time, such that any query $q(i, j)$ can be answered based on these truncated intervals.

The truncated intervals are defined as follows. For each $1 \leq i \leq n$, define a *truncated interval* $I_i'' = [\alpha_i'', \beta_i'']$ as the maximal interval such that (1) $I_i'' \subseteq I_i$; (2) each endpoint of I_i'' is a point of P; (3) suppose α_i'' is the point p_s, then p_s is contained in all intervals I_t with $s \leq t \leq i$; (4) similarly, suppose β_i'' is the point p_s, then p_s is contained in all intervals I_t with $i \leq t \leq s$. Let $\mathcal{I}''$ be the set of all truncated intervals. Below, we first show that $\mathcal{I}''$ can be computed in $O(n)$ time; then we prove that the answer for each query $q(i, j)$ can be obtained on $\mathcal{I}''$ by using the same procedure for answering a query on $\mathcal{I}'$.

To compute all α_i'''s for $1 \leq i \leq n$, we scan both P and $\mathcal{I}$ in an incremental fashion. Initially, for p_1 and I_1, set $\alpha_1'' = p_1$. By checking each interval incrementally, we find the first interval I_s such that $p_1 \notin I_s$. For each $1 \leq t \leq s - 1$, set $\alpha_t'' = p_1$. We then continue with p_2 and I_s. It is not difficult to see that if $2 \leq s - 1$, then $p_2 \in I_t$ holds for every $2 \leq t \leq s - 1$. If $p_2 \in I_s$, then we find the first index $s' > s$ such that $p_2 \notin I_{s'}$, and set $\alpha_t'' = p_2$ for each $s \leq t \leq s' - 1$. Otherwise, we consider p_3 and I_s in the same way. We continue this process until the value of α_n'' is set. Clearly, this procedure takes $O(n)$ time. All β_i'''s can also be obtained in $O(n)$ time by scanning P and $\mathcal{I}$ in the reverse order.

We claim that any query $q(i, j)$ can be answered based on the truncated interval set $\mathcal{I}''$. Consider an interval $I_t \in \mathcal{I}$ and the truncated interval $I_t'' \in \mathcal{I}''$ for any $1 \leq t \leq n$. It is possible that there exist some $p_m \in I_t \setminus I_t''$. Clearly, the claim is true if we can show for any such p_m, p_m can never be an answer to any query $q(i, j)$ with $i \leq t \leq j$.

Lemma 3. *For any t with $1 \leq t \leq n$, if a point $p_m \in I_t \setminus I_t''$, then p_m cannot be the answer to any query $q(i, j)$ for $i \leq t \leq j$.*

Proof. Since $p_m \in I_t$ and $p_m \notin I_t''$, it must be either $\alpha_t \leq p_m < \alpha_t''$ or $\beta_t'' < p_m \leq \beta_t$ due to $I_t'' \subseteq I_t$. Assume that it is the former case (the latter case can be analyzed similarly). Since $p_m < \alpha_t''$, it must be $p_m < p_t$ and thus $m < t$.

By the definition of I_t'', there must exist an interval $I_b \in \mathcal{I}$ with $m < b \leq t$ such that $p_m \notin I_b$. For any query $q(i, j)$ with $i \leq t \leq j$, if $i \leq b$, due to $p_m \notin I_b$, we have $p_m \notin \cap_{l=i}^{j} I_l$; otherwise, we have $m < i$. In either case, p_m cannot be an answer to the query $q(i, j)$. The lemma thus follows.

Therefore, the answer to any query $q(i, j)$ on $\mathcal{I}''$ is the same as that on $\mathcal{I}$ or $\mathcal{I}'$. As for the case of $\mathcal{I}'$, with $O(n)$ time preprocessing, any query can be answered in $O(1)$ time. In summary, we have the following result.

Lemma 4. *For the PII query problem, with $O(n)$ time preprocessing, any query $q(i, j)$ can be answered in $O(1)$ time.*

Back to the infinity k-coverage problem, Lemma 4 implies that any edge weight $w(i,j)$ can be obtained in $O(1)$ time after $O(n)$ time preprocessing. By Lemma 2, we have the following conclusion.

Theorem 2. *The infinity k-coverage problem is solvable in $O(\mathcal{T})$ time.*

5   The Linear Model

In this section, we derive an $O(\mathcal{T} \log^2 n)$ time algorithm for the linear model.

We begin with computing a single edge weight $w(i,j)$. Recall that $w(i,j)$ is the objective value of an optimal solution for the sub-problem $\mathcal{P}(i+1,j)$, i.e., $w(i,j) = obj(t, i+1, j) = c_t + w_l(t, i+1) + w_r(t, j)$ where $t = I(i+1, j)$.

Consider computing $w_l(t, i+1)$ for an arbitrary t with $i+1 \leq t \leq j$. We have $w_l(t, i+1) = \sum_{m=i+1}^{t} f_m(d(p_m, p_t)) = \sum_{m=i+1}^{t}(a_m \cdot (p_t - p_m)) = p_t \cdot \sum_{m=i+1}^{t} a_m - \sum_{m=i+1}^{t}(a_m \cdot p_m)$. Thus, if for each $i = 1, 2 \ldots, n$, we have the two prefix sums $\sum_{j=1}^{i} a_j$ and $\sum_{j=1}^{i}(a_j \cdot p_j)$, then each $w_l(t, i+1)$ can be computed in $O(1)$ time. Clearly, it takes $O(n)$ time to compute these prefix sums for $i = 1, 2, \ldots, n$. Similarly, every $w_r(t, j)$ can be computed in $O(1)$ time. Therefore, with $O(n)$ time preprocessing, each objective value $obj(t, i+1, j)$ can be obtained in $O(1)$ time. Consequently, we can compute each edge weight $w(i,j)$ in $O(j-i)$ time. Thus, applying Lemma 2 yields an $O(\mathcal{T} \cdot n)$ time solution for the linear model. Next, we present a more efficient algorithm.

Although computing a single $w(i,j)$ may require $\Omega(n)$ time, for any fixed $i \in \{0, 1, \ldots, n\}$, we are able to compute all $w(i,j)$'s for $j = i+1, \ldots, n$ in only $O(n \log n)$ time (see Lemma 5). Based on this result, we develop new algorithms for column-minima and (unrestricted) shortest paths, and use them to modify the components of our algorithm *Algo-k-link*.

Consider the problem of computing $w(0,j)$'s for all $1 \leq j \leq n$. Our idea is based on the monotonicity of $I(i,j)$ in Observation 1. We first give some intuition. Suppose we know $I(1,j)$ for some $1 \leq j \leq n$ and are to compute $I(1, j+1)$. By Observation 1, $I(1,j) \leq I(1, j+1)$. Clearly, $I(1, j+1) \in \{I(1,j), \ldots, j+1\}$, and we need to compute all $obj(i, 1, j+1)$'s for $I(1,j) \leq i \leq j+1$ to determine $I(1, j+1)$, in $O(j + 2 - I(1,j))$ time. Hence, if the $w(0,j)$'s are computed in an incremental manner, in the worst case it takes $O(n^2)$ time to compute all $w(0,j)$'s for $1 \leq j \leq n$. Lemma 5 gives a faster algorithm.

Lemma 5. *The $w(0,j)$'s, for all $1 \leq j \leq n$, can be computed in $O(n \log n)$ time.*

Proof. Below, we present an algorithm that has $O(\log n)$ stages and each stage runs in $O(n)$ time. In the first stage, we compute $w(0, \frac{n}{2})$ and $I(1, \frac{n}{2})$, which takes $O(\frac{n}{2})$ time. In the following, let $I(i)$ denote $I(1,i)$ for any $1 \leq i \leq n$.

The second stage computes $w(0, \frac{n}{4})$, $I(\frac{n}{4})$ and $w(0, \frac{3}{4}n)$, $I(\frac{3}{4}n)$, as follows. By Observation 1, $I(\frac{n}{4}) \leq I(\frac{n}{2}) \leq I(\frac{3}{4}n)$. Thus, to compute $w(0, \frac{3}{4}n)$ and $I(\frac{3}{4}n)$, we only need to compute the $obj(i, 1, \frac{3}{4}n)$'s for all $I(\frac{n}{2}) \leq i \leq \frac{3}{4}n$. Similarly, to compute $w(0, \frac{n}{4})$ and $I(\frac{n}{4})$, we only need to compute the $obj(i, 1, \frac{n}{4})$'s for all $1 \leq i \leq \min\{I(\frac{n}{2}), \frac{n}{4}\}$. Hence, the number of obj values we need to compute in

this stage is no more than $\frac{3}{4}n$. As discussed before, each objective value can be obtained in $O(1)$ time (after $O(n)$ time preprocessing). Hence, the running time of the second stage is $O(\frac{3}{4}n)$.

In general, in the t-th stage, we can compute the $w(0, \frac{2i-1}{2^t}n)$'s and $I(\frac{2i-1}{2^t}n)$'s, for all $1 \le i \le 2^{t-1}$, in $O(\frac{2^t-1}{2^t}n)$ time. The details are omitted and can be found in our full paper. The lemma thus follows.

By generalizing the algorithm in Lemma 5, we have the following corollary.

Corollary 1. *For any $0 \le i \le j \le n$, the weights $w(i,t)$ for all $i \le t \le j$ and $w(t,j)$ for all $i \le t \le j$ can be obtained in $O((j-i+1)\log(j-i+1))$ time.*

Before giving the shortest path algorithm, we first describe our column-minima algorithm. Given a matrix M, a column-minima algorithm finds the minimum element in every column of M. For a totally monotone $n \times n$ matrix M, the SMAWK algorithm in [2] can find all column minima of M in $O(n)$ time. In our setting, the matrix whose elements are edge weights of G is concave Monge, implying that the matrix is totally monotone. However, since computing a single edge weight takes $O(n)$ time, applying the SMAWK algorithm can only give an $O(n^2)$ time solution. Based on Corollary 1, Lemma 6 shows that the column-minima in our problem can be computed in $O(n \log^2 n)$ time. The lemma proof is omitted due to the space limit and can be found in our full paper.

For any i and j with $0 \le i \le j \le n$, let $M_{ij}[0, j-i; 0, j-i]$ denote the matrix whose elements are the weights of all edges between the vertices of $\{v_i, \ldots, v_j\}$ in G, i.e., each element $M_{ij}[i', j'] = w(i+i', i+j')$ if $0 \le i' \le j' \le j-i$ and $+\infty$ otherwise. By Corollary 1, every column of M_{ij} can be obtained in $O((j-i+1)\log(j-i+1))$ time.

Lemma 6. *The column minima of any matrix M_{ij} can be obtained in $O((j-i+1)\log^2(j-i+1))$ time.*

Our shortest path algorithm computes a shortest diameter path (without restricting its number of links) in G. For a DAG of n vertices whose edge weights satisfy the concave Monge property, the algorithms in [10,13,14] can find a shortest path in linear time provided that any edge weight can be obtained in $O(1)$ time. However, since computing a single edge weight needs $O(n)$ time in our setting, applying these algorithms directly would give an $O(n^2)$ time solution. To derive a faster algorithm, we modify Galil and Park's algorithm [10], as follows. First, the algorithm in [10] uses the SMAWK algorithm to compute the column-minima of some sub-matrices. If applied to our problem, we observe that the elements of each such sub-matrix are the weights of all edges between the vertices in some sequence of consecutive vertices. Thus, our column-minima algorithm in Lemma 6 can be applied. Second, in the algorithm [10], some individual matrix elements are also used besides the SMAWK algorithm. If applied to our problem, each of these individual elements is the weight of an edge connecting two neighboring vertices, i.e., $w(j-1, j)$ with $1 \le j \le n$, which can be obtained in $O(1)$ time. With these modifications, we can find a shortest diameter path in G in $O(n \log^2 n)$ time.

Lemma 7. *A shortest diameter path in G can be computed in $O(n \log^2 n)$ time.*

To find a shortest diameter path with at most k links in G, we modify the algorithm *Algo-k-link*, by modifying the algorithms in [3,22] and using the results in Lemmas 6 and 7. For the algorithm in [3], we first replace the linear time shortest path algorithm by our algorithm in Lemma 7, and then replace the SMAWK algorithm by our algorithm in Lemma 6 to compute the column-minima for sub-matrices. The resulting algorithm takes $O(n\sqrt{k \log n} \log^2 n)$ time. Similar modifications can also be made to the algorithm [22]. We have the result below.

Theorem 3. *In $O(\mathcal{T} \log^2 n)$ time, we can find a shortest diameter path with at most k links in G, and hence obtain an optimal solution for the linear model.*

6 The Path Equipartition Problems

In this section, we discuss the L_∞ and L_d path equipartition problems. We first consider the L_∞ version, which is solved based on dynamic programming.

Given a path $P = (p_1, p_2, \ldots, p_n)$ with vertex weights $c_i \geq 0$, for any $j \leq i$, let P_{ji} denote the sub-path of P from p_j to p_i and $C(P_{ji})$ denote the sum of vertex weights of P_{ji}. For $1 \leq i \leq n$ and $1 \leq l \leq k$, let $W(i, l)$ denote the minimum objective value for partitioning the sub-path P_{1i} into l sub-paths. Our goal is to compute $W(n, k)$. It is easy to see $W(i, l) = \min_{1 \leq j \leq i}\{\max\{W(j - 1, l - 1), h(j, i)\}\}$, where $h(j, i) = |C(P_{ji}) - \mu|$ and $\mu = C(P)/k$. We set $W(0, l) = 0$ for any $1 \leq l \leq k$ and $W(i, 1) = |C(P_{1i}) - \mu|$ for any $1 \leq i \leq n$. Thus, in a straightforward manner, one can find an optimal partition of P in $O(n^2 k)$ time. To reduce the running time, we prove that the matrix involving the values $W(i, l)$ is totally monotone [2], and thus the linear time row-minima algorithm [2] can be applied. Consequently, the above DP can be solved in $O(nk)$ time. The details are omitted due to the space limit. Not that if μ is not $C(P)/k$ but an arbitrary value, the above result is still applicable.

For the L_d path equipartition problem, since Monge property has already been proved in [12], we simply point out here (without giving any further details) that it can be modeled as finding a shortest path with k edges in a concave Monge graph, and thus is solvable in $O(\mathcal{T})$ time [3,22].

References

1. Aggarwal, P.K., Sharir, M.: Efficient algorithms for geometric optimization. ACM Computing Surveys 30(4), 412–458 (1998)
2. Aggarwal, A., Klawe, M., Moran, S., Shor, P., Wilbur, R.: Geometric applications of a matrix-searching algorithm. Algorithmica 2, 195–208 (1987)
3. Aggarwal, A., Schieber, B., Tokuyama, T.: Finding a minimum weight k-link path in graphs with concave Monge property and applications. Discrete & Computational Geometry 12, 263–280 (1994)
4. Auletta, V., Parente, D., Persiano, G.: Placing resources on a growing line. Journal of Algorithms 26(1), 87–100 (1998)

5. Chandrasekaran, R., Tamir, A.: Algebraic optimization: The Fermat-Weber location problem. Mathematical Programming 46(2), 219–224 (1990)
6. Drezner, Z., Hamacher, H.W.: Facility Location: Applications and Theory. Springer, New York (2004)
7. Dyer, M.E.: On a multidimensional search technique and its application to the Euclidean one centre problem. SIAM Journal on Computing 15(3), 725–738 (1986)
8. Frederickson, G.N.: Optimal algorithms for tree partitioning. In: Proc. of the 2nd Annual ACM-SIAM Symposium of Discrete Algorithms, pp. 168–177 (1991)
9. Gabow, H., Bentley, J., Tarjan, R.: Scaling and related techniques for geometry problems. In: Proc. of the 16th Annual ACM Symposium on Theory of Computing (STOC), pp. 135–143 (1984)
10. Galil, Z., Park, K.: A linear-time algorithm for concave one-dimensional dynamic programming. Information Processing Letters 33(6), 309–311 (1990)
11. Hassin, R., Tamir, A.: Improved complexity bounds for location problems on the real line. Operations Research Letters 10, 395–402 (1991)
12. Ho, P.-H., Tamir, A., Wu, B.Y.: Minimum L_k path partitioning – An illustration of the Monge property. Operations Research Letters 36(1), 43–45 (2008)
13. Klawe, M.M.: A simple linear time algorithm for concave one-dimensional dynamic programming. Technical Report 89-16. University of British Columbia, Vancouver, Canada (1989)
14. Larmore, L., Schieber, B.: On-line dynamic programming with applications to the prediction of RNA secondary structure. Journal of Algorithms 12(3), 490–515 (1991)
15. Lee, D.T., Wu, Y.F.: Geometric complexity of some location problems. Algorithmica 1(1-4), 193–211 (1986)
16. Liverani, M., Morgana, A., Simeone, B., Storchi, G.: Path equipartition in the Chebyshev norm. European Journal of Operational Research 123(2), 428–435 (2000)
17. Love, R.F.: One-dimensional facility location-allocation using dynamic programming. Management Science 22(5), 614–617 (1976)
18. Manne, F., Sørevik, T.: Optimal partitioning of sequences. Journal of Algorithms 19(2), 235–249 (1995)
19. Megiddo, N.: Linear-time algorithms for linear programming in R^3 and related problems. SIAM Journal on Computing 12(4), 759–776 (1983)
20. Megiddo, N., Zemel, E., Hakimi, S.L.: The maximum coverage location problem. SIAM J. on Algebraic and Discrete Methods 4(2), 253–261 (1983)
21. Olstad, B., Manne, F.: Efficient partitioning of sequences. IEEE Transactions on Computers 44(1995), 1322–1326 (1995)
22. Schieber, B.: Computing a minimum weight k-link path in graphs with the concave Monge property. Journal of Algorithms 29(2), 204–222 (1998)
23. van Hoesel, S., Wagelmans, A.: On the p-coverage problem on the real line. Statistica Neerlandica 61(1), 16–34 (2007)
24. Woeginger, G.J.: Monge strikes again: optimal placement of web proxies in the Internet. Operations Research Letters 27(3), 93–96 (2000)

Multicut in Trees Viewed through the Eyes of Vertex Cover

Jianer Chen[1,*], Jia-Hao Fan[1], Iyad A. Kanj[2,**],
Yang Liu[3], and Fenghui Zhang[4]

[1] Department of Computer Science and Engineering, Texas A&M University,
College Station, TX 77843
{chen,grantfan}@cs.tamu.edu
[2] School of Computing, DePaul University, 243 S. Wabash Avenue,
Chicago, IL 60604
ikanj@cs.depaul.edu
[3] Department of Computer Science, University of Texas-Pan American,
Edinburg, TX 78539
yliu@cs.panam.edu
[4] Google Kirkland, 747 6th Street South, Kirkland, WA 98033
fhzhang@gmail.com

Abstract. We take a new look at the MULTICUT problem in trees through
the eyes of the VERTEX COVER problem. This connection, together with
other techniques that we develop, allows us to significantly improve the
$O(k^6)$ upper bound on the kernel size for MULTICUT, given by Bous-
quet et al., to $O(k^3)$. We exploit this connection further to present a
parameterized algorithm for MULTICUT that runs in time $O^*(\rho^k)$, where
$\rho = (\sqrt{5}+1)/2 \approx 1.618$. This improves the previous (time) upper bound
of $O^*(2^k)$, given by Guo and Niedermeier, for the problem.

1 Introduction

Let $\mathcal{F}$ be a forest (i.e., a collection of disjoint trees). A *request* is a pair (u, v),
where $u, v \in V(\mathcal{F})$. Let R be a set of requests. A subset of edges $E' \subseteq E(\mathcal{F})$
is said to be an *edge cut*, or simply a *cut*, for R if for every request (u, v) in R,
there is no path between u and v in $\mathcal{F} - E'$. The *size* of a cut E' is $|E'|$. A cut E'
is *minimum* if its cardinality is minimum among all cuts. The (parameterized)
multicut problem in trees is defined as follows:

MULTICUT
Given: A forest $\mathcal{F}$ and a set of requests $R \subseteq V(\mathcal{F}) \times V(\mathcal{F})$.
Parameter: A nonnegative integer k.
Question: Is there a cut for R of size at most k?

The MULTICUT problem has applications in networking [5]. The problem is known to be NP-hard, and its optimization version can be approximated to within ratio 2 [7]. We consider the MULTICUT problem from the parameterized complexity perspective. We mention that the parameterized complexity of several graph separation problems, including variants of the MULTICUT problem, was studied with respect to different parameters by Marx in [9]. Guo and Niedermeier [8] showed that the MULTICUT problem is fixed-parameter tractable by giving an $O^*(2^k)$ time algorithm for the problem. (The asymptotic notation $O^*(f(k))$ denotes time complexity of the form $f(k) \cdot n^{O(1)}$, where n is the input length.) They also showed that MULTICUT has an exponential-size kernel. Recently, Bousquet, Daligault, Thomassé, and Yeo, improved the upper bound on the kernel size for MULTICUT to $O(k^6)$ [3].

In this paper we take a new look at MULTICUT through the eyes of the VERTEX COVER problem. This connection allows us to give an upper bound of $O(k^3)$ on the kernel size for MULTICUT, significantly improving the previous $O(k^6)$ upper bound given by Bousquet et al. [3]. We exploit this connection further to give a parameterized algorithm for MULTICUT that runs in $O^*(\rho^k)$ time, where $\rho = (\sqrt{5} + 1)/2 \approx 1.618$ (golden ratio) is the positive root of the polynomial $x^2 - x - 1$; this improves the $O^*(2^k)$ time algorithm, given by Guo and Niedermeier [8]. To obtain the $O(k^3)$ upper bound on the kernel size, we first group the vertices in the forest into $O(k)$ groups. We then introduce an ordering that orders the leaves in a group with respect to every other group. This ordering allows us to introduce a set of reduction rules that limits the number of leaves in a group that have requests to the vertices in another group. At the core of this set of reduction rules is a rule that utilizes the crown kernelization algorithm for VERTEX COVER [1]. All the above allows us to upper bound the number of leaves in the reduced instance by $O(k^2)$, improving the $O(k^4)$ upper bound on the number of leaves obtained in [3]. Finally, we show that the size of the reduced instance is at most the number of leaves in it multiplied by a linear factor of k, thus yielding an upper bound of $O(k^3)$ on the size of the kernel. To obtain the $O^*(((\sqrt{5} + 1)/2)^k)$ time algorithm, we first establish structural connections between MULTICUT and VERTEX COVER that allow us to simplify the instance of MULTICUT. We then exploit the simplified structure of the resulting instance to present a simple search-tree algorithm for MULTICUT that runs in time $O^*(((\sqrt{5} + 1)/2)^k)$. We note that, even though some connection between MULTICUT and VERTEX COVER was observed in [7,8], this connection was not developed or utilized in kernelization algorithms, nor in parameterized algorithms for MULTICUT.

We mention that, very recently, the multicut problem in general graphs was shown to be fixed-parameter tractable independently by Bousquet, Daligault, and Thomassé [2], and by Marx and Razgon [10], answering an open problem in parameterized complexity theory.

Most of the proofs were omitted from this version due to the lack of space.

2   Preliminaries

We assume familiarity with the basic notations and terminologies about graphs and parameterized complexity (refer, for example, to [6,11]).

For a graph H we denote by $V(H)$ and $E(H)$ the set of vertices and edges of H, respectively. For a set of vertices $S \subseteq V(H)$, we denote by $H[S]$ the subgraph of H induced by the set of vertices in S. For a vertex $v \in H$, $H - v$ denotes $H[V(H) \setminus \{v\}]$, and for a subset of vertices $S \subseteq V(H)$, $H - S$ denotes $H[V(H) \setminus S]$. By *removing* a subgraph H' of H we mean removing $V(H')$ from H to obtain $H - V(H')$. Two vertices u and v in H are said to be *adjacent* or *neighbors* if $uv \in E(H)$. For two vertices $u, v \in V(H)$, we denote by $H - uv$ the graph $(V(H), E(H) \setminus \{uv\})$, and by $H + uv$ the *simple* graph $(V(H), E(H) \cup \{uv\})$. By *removing* an edge uv from H we mean setting $H = H - uv$. For a subset of edges $E' \subseteq E(H)$, we denote by $H - E'$ the graph $(V(H), E(H) \setminus E')$. For a vertex $v \in H$, $N(v)$ denotes the set of neighbors of v in H. The *degree* of a vertex v in H, denoted $deg_H(v)$, is $|N(v)|$. The *degree* of H, denoted $\Delta(H)$, is $\Delta(H) = \max\{deg_H(v) : v \in H\}$. The *length* of a path in a graph H is the number of edges in it. A *matching* in a graph is a set of edges such that no two edges in the set share an endpoint. A *vertex cover* for a graph H is a set of vertices such that each edge in H is incident to at least one vertex in this set. A vertex cover for H is *minimum* if its cardinality is minimum among all vertex covers of H; we denote by $\tau(H)$ the cardinality/size of a minimum vertex cover of H.

A *tree* is a connected acyclic graph. A *leaf* in a tree is a vertex of degree at most 1. A nonleaf vertex in a tree is called an *internal* vertex. The *internal degree* of a vertex v in a tree is the number of nonleaf vertices in $N(v)$. For two vertices u and v, the *distance* between u and v in T, denoted $dist_T(u, v)$, is the length of the unique path between u and v in T. A leaf x in a tree is said to be *attached* to vertex u if u is the unique neighbor of x in the tree. A *caterpillar* is a tree consisting of a path with leaves attached to the vertices on the path. A *forest* is a collection of disjoint trees.

Let T be a tree with root r. For a vertex $u \neq r$ in $V(T)$, we denote by $\pi(u)$ the parent of u in T. A *sibling* of u is a child $v \neq u$ of $\pi(u)$ (if exists), an *uncle* of u is a sibling of $\pi(u)$, and a *cousin* of u is a child of an uncle of u. A vertex v is a *nephew* of a vertex u if u is an uncle of v. For a vertex $u \in V(T)$, T_u denotes the subtree of T rooted at u. The *children* of a vertex u in $V(T)$, denoted $children(u)$, are the vertices in $N(u)$ if $u = r$, and in $N(u) - \pi(u)$ if $u \neq r$. A vertex u is a *grandparent* of a vertex v if $\pi(v)$ is a child of u. A vertex v is a *grandchild* of a vertex u if u is a grandparent of v.

A *parameterized problem* is a set of instances of the form (x, k), where $x \in \Sigma^*$ for a finite alphabet set Σ, and k is a non-negative integer called the *parameter*. A parameterized problem Q is *kernelizable* if there exists a polynomial-time reduction that maps an instance (x, k) of Q to another instance (x', k') of Q such that: (1) $|x'| \leq g(k)$ for some recursive function g, (2) $k' \leq k$, and (3) (x, k) is a yes-instance of Q if and only if (x', k') is a yes-instance of Q. The instance (x', k') is called the *kernel* of (x, k).

Let $(\mathcal{F}, R, k)$ be an instance of MULTICUT, and let uv be an edge in $E(\mathcal{F})$. If we know that edge uv can be included in the solution sought, then we can remove uv from $\mathcal{F}$ and decrement the parameter k by 1; we say in this case that we *cut* edge uv. By *cutting* a leaf we mean cutting the unique edge incident to it. If T is a rooted tree in $\mathcal{F}$ and $u \in T$ is not the root, we say that we *cut* u to mean that we cut the edge $u\pi(u)$. (Note that after cutting an edge uv that is not incident to a leaf we obtain two trees: one containing u and the other containing v. Obviously, any request in R that goes across the two trees can be discarded.) On the other hand, if we know that edge uv can be excluded from the solution sought, we say in this case that edge uv is *kept*, and we can *contract* it by identifying the two vertices u and v, i.e., removing u and v and creating a new vertex with neighbors $(N(u) \cup N(v)) \setminus \{u, v\}$. If edge uv is contracted and w is the new vertex, then any request in R of the form (u, x) or (v, x) is replaced by the request (w, x).

A leaf x in $\mathcal{F}$ is said to be *good* if there exists another leaf y such that x and y are attached to the same vertex in $\mathcal{F}$ and (x, y) is a request in R; otherwise, x is said to be a *bad* leaf. (We differ from the terminology used in [3]. What we call good leaves are called bad leaves in [3], and vice versa.) We define an auxiliary graph for $\mathcal{F}$, denoted G for simplicity, as follows. The vertices of G are the good leaves in $\mathcal{F}$, and two vertices x and y in G are adjacent (in G) if and only if x and y are attached to the same vertex of $\mathcal{F}$ and there is a request between x and y in R. Without loss of generality, we shall call the vertices in G with the same names as their corresponding good leaves in $\mathcal{F}$, and it will be clear from the context whether we are referring to the good leaves in $\mathcal{F}$ or to their corresponding vertices in G. Note that there is no edge in G between two good leaves that are attached to different vertices even though there could be a request between them. Therefore, G consists of isolated subgraphs, each is not necessarily connected and is induced by the set of good leaves that are attached to the same vertex in $\mathcal{F}$. For an internal vertex $u \in \mathcal{F}$ we denote by G_u the subgraph of G induced by the good leaves that are attached to u (if any).

It is not difficult to see that if a set of vertices C in G is a vertex cover for G then $E_C = \{uw \in E(\mathcal{F}) \mid w \in C\}$, which has the same cardinality as C, cuts all the requests between every two good leaves that are attached to the same vertex in $\mathcal{F}$. On the other hand, for any cut, the edges in the cut that are incident to the leaves in G is a vertex cover of G. It follows that the cardinality of the set of edges that are incident to the leaves in G in a minimum cut of $\mathcal{F}$ is at least the size of a minimum vertex cover for G.

3   The Kernel

In this section we prove an upper bound of $O(k^3)$ on the kernel size for MULTICUT. Let $(\mathcal{F}, R, k)$ be an instance of MULTICUT, and let T be tree in $\mathcal{F}$. Two requests (u, v) and (p, q) in R are said to be *disjoint* if the path between u and v in $\mathcal{F}$ is edge-disjoint from the path between p and q in $\mathcal{F}$. A request (p, q) *dominates* a request (u, v) if the path from p to q in $\mathcal{F}$ is a subpath of the path from u to v in $\mathcal{F}$. The following reduction rules for MULTICUT are folklore, easy to verify, and can be implemented to run in polynomial time (see [3,8] for proofs).

Reduction Rule 1 (Useless edge). *If no request in R is disconnected by the removal of edge $uv \in E(\mathcal{F})$, then remove edge uv from $\mathcal{F}$.*

Reduction Rule 2 (Useless pair). *If $(u, v) \in R$ where u, v are in two different trees of $\mathcal{F}$, then remove (u, v) from R.*

Reduction Rule 3 (Unit request). *If $(u, v) \in R$ and $uv \in E(\mathcal{F})$, then remove uv from $\mathcal{F}$ and decrement k by 1.*

Reduction Rule 4 (Disjoint requests). *If there are $k + 1$ pairwise disjoint requests in R, then reject the instance $(\mathcal{F}, R, k)$.*

Reduction Rule 5 (Unique direction). *Let x be a leaf or an internal degree-2 vertex in $\mathcal{F}$. If all the requests from x have the same direction then: if x is a leaf then contract the edge incident to x, and if x is an internal degree-2 vertex then contract the edge incident to x that is not on any of the paths corresponding to the requests from x.*

Reduction Rule 6 (Domination/Inclusion). *If a request (p, q) dominates another request (u, v) then remove (u, v) from R.*

It was shown in [3] that the number of good leaves (called bad leaves in [3]) is $O(k^2)$. We introduce a reduction rule next that allows us to derive the same upper bound on the number of good leaves in $\mathcal{F}$, and which uses Buss' kernelization algorithm for the VERTEX COVER problem [4] (this reduction rule was implicitly observed in [8]). (The VERTEX COVER problem is: Given a graph H and a parameter k, decide if there is a vertex cover for H of size at most k.) Recall that the graph G is the graph whose vertices are the good leaves in $\mathcal{F}$ and whose edges correspond to the requests between good leaves that are attached to the same vertex in $\mathcal{F}$.

Reduction Rule 7 (Bound on good leaves). *Apply Buss' algorithm for VERTEX COVER [4] to (G, k): for every vertex x in G whose degree (in G) is at least $k + 1$, cut leaf x in $\mathcal{F}$. If the number of remaining good leaves in $\mathcal{F}$ is more than $2k^2$, then reject the input instance $(\mathcal{F}, R, k)$.*

We shall assume henceforth that none of Reduction Rules 1 – 7 applies to $(\mathcal{F}, R, k)$. We shall also assume that isolated vertices are removed from $\mathcal{F}$ at all times. The statements in the following lemma were shown in [3]:

Lemma 1. *The following are true:*

  a. *(Claim 5 in [3]) The number of internal vertices in $\mathcal{F}$ of internal degree 1 is at most k.*
  b. *(Claim 6 in [3]) The number of internal vertices in $\mathcal{F}$ of internal degree at least 3 is at most k.*

We now define a grouping of the vertices in $\mathcal{F}$ into three types of groups.

Definition 1. A *type-I group* consists of an internal vertex u of $\mathcal{F}$ that has at least one good leaf attached to it, together with all the leaves (bad and good) that are attached to u; we say that vertex u *forms* the type-I group. A *type-II group* consists of an internal vertex u in $\mathcal{F}$ of internal degree at least 3 that does not have any good leaves attached to it, together with all the (bad) leaves attached to u (if any); we say that vertex u *forms* the type-II group. A *type-III group* consists of the vertices (internal and leaves) of a caterpillar in $\mathcal{F}$ such that: (1) every internal vertex of the caterpillar has internal degree 2 in $\mathcal{F}$, and (2) there is no request between any two vertices (internal-internal, leaf-internal, nor leaf-leaf) of the caterpillar. Note that condition (2) implies that all the leaves attached to the internal vertices in a type-III group are bad leaves.

Lemma 2. $V(F)$ *can be partitioned in polynomial time into* $O(k)$ *type-I, type-II, and type-III groups.*

Definition 2. Let G_i be a type-I, type-II, or a type-III group. The *intergroup edges* of G_i are the edges in $\mathcal{F}$ with exactly one endpoint in G_i; the *intergroup degree* of G_i, denoted d_i, is the number of intergroup edges of G_i. Note that if G_i is a type-I or a type-II group, where u is the internal vertex in $\mathcal{F}$ that forms G_i, then d_i is the internal degree of u in $\mathcal{F}$. On the other hand, if G_i is a type-III group then $d_i = 2$. The *internal vertices of* G_i are the internal vertices of $\mathcal{F}$ that are in G_i. The *internal edges of* G_i are the edges between the internal vertices of G_i. Note that only type-III groups can have internal edges. The *leaves (resp. good/bad leaves) of* G_i are the leaves (resp. good/bad leaves) attached to the internal vertices of G_i.

Lemma 3

$$\sum_{G_i \ is \ a \ group} d_i = O(k).$$

We introduce next a reduction rule that is used to bound the number of bad leaves that have requests to good leaves. We apply the crown reduction kernelization algorithm, described in [1], to the instance (G, k) of VERTEX COVER. This algorithm partitions $V(G)$ into three sets I, H, and O, such that: (1) I is an independent set of G, and no edge exists between the vertices in I and those in O, (2) there exists a minimum vertex cover of G containing H, (3) there exists a matching M that matches every vertex in H to a vertex in I, and (4) $|O| \leq 3k$ if a solution to (G, k) exists [1].

Reduction Rule 8 (Crown reduction). *Apply the crown reduction algorithm to* (G, k) *to partition* $V(G)$ *into the three sets* H, I, O. *If* $|O| > 3k$ *or* $|H| > k$, *then reject the instance* $(\mathcal{F}, R, k)$.

Consider G_u, where u is a vertex in $\mathcal{F}$ that forms a type-I group. Denote by H_u, I_u, O_u the intersection of H, I, O with $V(G_u)$, respectively. Clearly, the matching M matching H into I in G induces a matching M_u in G_u that matches H_u into I_u. Let OUT_u be the set of vertices in I_u that are not matched by M_u (i.e., $I_u \setminus V(M_u)$). We have the following lemma:

Lemma 4. *Let u be a vertex in $\mathcal{F}$ that forms a type-I group. Any vertex cover of G_u that contains ℓ vertices from OUT_u has size at least $\tau(G_u) + \ell$.*

Corollary 1. *Let u be a vertex in $\mathcal{F}$ that forms a type-I group G_i. If $(\mathcal{F}, R, k)$ has a solution, then it has a solution that cuts at most $d_i - 1 = d_u - 1$ leaves from OUT_u, where d_u is the internal degree of u.*

Lemma 5. *If there exists a solution to the instance $(\mathcal{F}, R, k)$, then there exists a solution S to $(\mathcal{F}, R, k)$ such that, for any group G_i: if G_i is a type-I or a type-II group then S cuts at most $d_i - 1$ bad leaves of G_i, and if G_i is a type-III group then the number of bad leaves and internal edges of G_i that are cut by S is at most $d_i - 1 = 1$.*

Next, we introduce reduction rules to bound the number of bad leaves in $(\mathcal{F}, R, k)$. The main idea behind these reduction rules is to use several orderings (defined later) on the set bad leaves of a group G_i w.r.t. to another group G_j, to limit the number of bad leaves of G_i that have requests to bad leaves or vertices of G_j to at most $d_i \times d_j$. For a leaf x of a group G_i, we shall refer to the internal vertex in G_i that x is attached to by $\nu(x)$.

Reduction Rule 9. *Let x be a vertex, and let G_i be a group. If there are at least d_i bad leaves in G_i that have requests to x, then let L_x be the list containing the bad leaves in G_i that have requests to x sorted in a nondecreasing order of their distance to x, where ties are broken arbitrarily. For every bad leaf z in G_i whose rank in L_x is at least d_i, replace the request (z, x) in R with the request $(\nu(z), x)$.*

Proof. Suppose that the above reduction rule applies to a group G_i in $(\mathcal{F}, R, k)$ and some vertex x, and let $(\mathcal{F}, R', k)$ be the resulting instance. Clearly, any solution to $(\mathcal{F}, R', k)$ is also a solution to $(\mathcal{F}, R, k)$. Therefore, it suffices to prove that if there exists a solution for $(\mathcal{F}, R, k)$ then there also exists a solution for $(\mathcal{F}, R', k)$. Suppose that there is a solution to $(\mathcal{F}, R, k)$. By Lemma 5, we can assume that there is a solution S that cuts at most $d_i - 1$ bad leaves from G_i. It follows from the preceding statement that if y is the the bad leaf whose rank in L_x is d_i, then S must cut an edge on the path between $\nu(y)$ and x (otherwise S would cut the first d_i bad leaves in L_x). Consequently, any request (z, x) from a bad leaf z whose rank in L_x is at least d_i that was replaced with the request $(\nu(z), x)$ is cut by S, and S is a solution to $(\mathcal{F}, R', k)$. $\square$

Definition 3. Let G_i and G_j be two distinct groups. If x is a bad leaf in G_i that has a request to at least one internal vertex in G_j, then among all internal vertices in G_j that x has requests to, we define the *vertex-offset* of x with respect to G_j, denoted $v\text{-}offset_j(x)$, to be *the* vertex of minimum distance to x. If x is a bad leaf in G_i that has a request to a bad leaf in G_j, then among all bad leaves in G_j that x has requests to, we define the *leaf-offset* of x with respect to G_j, denoted $l\text{-}offset_j(x)$, to be *a* leaf of minimum distance to x. Let u be the vertex in G_j with the minimum distance to the vertices in G_i. We define an order $\preceq_j^v$ on the set of vertex-offsets of the bad leaves in G_i that have requests to internal

vertices in G_j as follows. For two vertex-offsets y and y' of two bad leaves in G_i, $y \preceq_j^v y'$ if and only if the distance from u to y is smaller or equal to the distance from u to y'. The $\preceq_j^v$ order on the vertex-offsets in G_j of the bad leaves in G_i having requests to internal vertices in G_j induces an order $\preceq_j^v$ on the bad leaves in G_i in a natural way: for two bad leaves x and x' in G_i that have requests to internal vertices in G_j: $x \preceq_j^v x'$ if and only if v-offset$_j(x) \preceq_j^v$ v-offset$_j(x')$. Similarly, we can define an order $\preceq_j^l$ on the set of leaf-offsets of the bad leaves in G_i, which induces an order $\preceq_j^l$ on the bad leaves in G_i (that have requests to bad leaves in G_j) as follows: for two bad leaves x and x' in G_i that have requests to bad leaves in G_j: $x \preceq_j^l x'$ if and only if l-offset$_j(x) \preceq_j^v$ l-offset$_j(x')$.

Reduction Rule 10. *Let G_i and G_j be two distinct groups. If there are at least d_i bad leaves in G_i that have requests to internal vertices of G_j, then consider all bad leaves in G_i that have requests to internal vertices of G_j, and sort them in a non-decreasing order with respect to the order $\preceq_j^v$; let L_i be the sorted list. For every bad leaf x in L_i whose rank in L_i is at least d_i, replace every request (x, p) in R from x to an internal vertex p of G_j with the request $(\nu(x), p)$.*

Reduction Rule 11. *Suppose that Reduction Rule 9 does not apply to $(\mathcal{F}, R, k)$. Let G_i and G_j be two distinct groups. If there are at least $(d_i - 1) \times d_j + 1$ bad leaves in G_i that have requests to bad leaves in G_j, then consider all the bad leaves in G_i that have requests to bad leaves of G_j, and sort them in a non-decreasing order with respect to the order $\preceq_j^l$; let L_i be the sorted list. For every bad leaf x in G_i whose rank in L_i is at least $(d_i - 1) \times d_j + 1$, replace every request (x, y) in R from x to a bad leaf y of G_j with the request $(\nu(x), y)$.*

Reduction Rule 12. *Suppose that Reduction Rule 9 does not apply to $(\mathcal{F}, R, k)$. Let u be a vertex such that u forms a type-I group G_j, and let $G_i \neq G_j$ be a group. If there are at least $d_j \times (d_i - 1) + 1$ many bad leaves in G_i that have requests to leaves in OUT_u, let L_i be the list of bad leaves in G_i that have requests to vertices in OUT_u sorted in a non-decreasing order of their distance from u. For each bad leaf x in L_i whose rank is at least $d_j \times (d_i - 1) + 1$, replace every request (x, y) in R from x to a leaf y in OUT_u with the request $(\nu(x), y)$.*

Definition 4. The instance $(\mathcal{F}, R, k)$ is said to be *reduced* if none of Reduction Rules 1 – 12 is applicable to it.

This lemma follows from Reduction Rules 8–12 after summing over all groups:

Lemma 6. *Let $(\mathcal{F}, R, k)$ be a reduced instance. The number of leaves in $\mathcal{F}$ is $O(k^2)$.*

This lemma follows from Lemma 2 and Lemma 6:

Lemma 7. *Let $(\mathcal{F}, R, k)$ be a reduced instance. The number of vertices in $\mathcal{F}$ that are not internal degree-2 vertices is $O(k^2)$.*

Lemma 8. *Let $(\mathcal{F}, R, k)$ be a reduced instance. The number of internal degree-2 vertices in $\mathcal{F}$ is $O(k^3)$.*

Theorem 1. *The* MULTICUT *problem has a kernel of at most* $O(k^3)$ *vertices.*

Proof. The statement of the theorem follows directly from Lemmas 7 and 8, and the fact that Reduction Rules 1 – 12 can be implemented in polynomial time.   □

4   The Algorithm

Let $(\mathcal{F}, R, k)$ be a reduced instance of MULTICUT. Since $(\mathcal{F}, R, k)$ is reduced, we can assume that every tree in $\mathcal{F}$ is nontrivial (i.e., contains at least two vertices), and that there is at least one request between the vertices of every tree in $\mathcal{F}$. We shall assume that every tree in $\mathcal{F}$ is rooted at some internal vertex in the tree (chosen arbitrarily). Let T be a tree in $\mathcal{F}$ rooted at a vertex r. A vertex $u \in V(T)$ is *important* if all the children of u are leaves. For a set of vertices $V' \subseteq V(T)$ and a vertex $u \in V'$, u is *farthest* from r w.r.t. V' if $dist_T(u, r) = \max\{dist_T(w, r) \mid w \in V'\}$. The following lemma, which again emphasizes the importance of VERTEX COVER for both kernelization and parameterized algorithms for MULTICUT, will be pivotal:

Lemma 9. *Let* $(\mathcal{F}, R, k)$ *be a reduced instance of* MULTICUT. *Let* T *be a tree in* $\mathcal{F}$ *rooted at* r. *There exists a minimum cut* E_{min} *for the requests of* R *in* T *such that, for every important vertex* $u \in V(T)$, *the subset of edges in* E_{min} *that are incident to the children of* u *(if any) corresponds to a minimum vertex cover of* G_u. *(Recall that* G_u *is the subgraph of* G *induced by the vertices in* G *that correspond to the good leaves attached to* u.)

Reduction Rule 13. *Let* $(\mathcal{F}, R, k)$ *be a reduced instance of* MULTICUT, *let* T *be a tree in* $\mathcal{F}$ *rooted at* r, *and let* $u \neq r$ *be a vertex in* T. *If there exists no request between a vertex in* $V(T_u)$ *and a vertex in* $V(T_{\pi(u)}) \setminus V(T_u)$ *then contract the edge* $u\pi(u)$.

Reduction Rule 14. *Let* $(\mathcal{F}, R, k)$ *be a reduced instance of* MULTICUT, *let* T *be a tree in* $\mathcal{F}$ *rooted at* r, *and let* u *be an important vertex in* T *such that* $\Delta(G_u) \leq 2$. *If there exists a (leaf) child* l *of* u *that is not in any minimum vertex cover of* G_u, *then contract the edge* ul.

Reduction Rule 15. *Let* $(\mathcal{F}, R, k)$ *be a reduced instance of* MULTICUT, *let* T *be a tree of* $\mathcal{F}$ *rooted at* r, *and let* w *be an important vertex in* T *such that* $\Delta(G_w) \leq 2$. *For every path in* G_w *of even length, cut the leaves in* $children(w)$ *that correspond to the unique minimum vertex cover of* P.

Definition 5. *Let* $(\mathcal{F}, R, k)$ *be a reduced instance of* MULTICUT, *let* T *be a tree of* $\mathcal{F}$ *rooted at* r, *and let* $w \neq r$ *be an important vertex in* T. *A request between a vertex in* $V(T_w)$ *and a vertex in* $V(T_{\pi(w)}) \setminus V(T_w)$ *is called a* cross request.

Reduction Rule 16. *Let* $(\mathcal{F}, R, k)$ *be a reduced instance of* MULTICUT, *let* T *be a tree rooted at* r *in* $\mathcal{F}$, *and let* $w \neq r$ *be an important vertex in* T *such that* $\Delta(G_w) \leq 2$. *If there is a minimum vertex cover of* G_w *such that cutting the leaves in this minimum vertex cover cuts all the cross requests from the vertices in* $V(T_w)$ *then contract* $w\pi(w)$.

Definition 6. The instance $(\mathcal{F}, R, k)$ of MULTICUT is said to be *strongly reduced* if $(\mathcal{F}, R, k)$ is reduced and none of Reduction Rules 13 – 16 is applicable to it.

Proposition 1. *Let $(\mathcal{F}, R, k)$ be a strongly reduced instance, and let T be a tree in $\mathcal{F}$ rooted at a vertex r.*

- (i) *For any vertex $u \in V(T)$, there exists no request between u and $\pi(u)$.*
- (ii) *For any vertex $u \neq r$ in $V(T)$, there exists a request between some vertex in $V(T_u)$ and some vertex in $V(T_{\pi(u)}) \setminus V(T_u)$.*
- (iii) *For any internal vertex $u \in V(T)$, there exists at least one request between the vertices in $V(T_u) - u$.*
- (iv) *For any important vertex $w \in V(T)$ such that $\Delta(G_w) \leq 2$ and any child u of w, there exists a request between u and a sibling of u, and hence all the children of an important vertex are good leaves.*
- (v) *For any important vertex $w \in V(T)$ such that $\Delta(G_w) \leq 2$, G_w contains no path of even length.*
- (vi) *For any important vertex $w \neq r$ in $V(T)$ such that $\Delta(G_w) \leq 2$, there is no minimum vertex cover of G_w such that cutting the leaves in this minimum vertex cover cuts all the cross requests from the vertices in $V(T_w)$.*

Let $(\mathcal{F}, R, k)$ be a strongly reduced instance of MULTICUT. The algorithm is a branch-and-search algorithm, and its execution can be depicted by a search tree. The running time of the algorithm is proportional to the number of leaves in the search tree, multiplied by the time spent along each root-leaf path, which will be polynomial in k. Therefore, the main step in the analysis of the algorithm is to derive an upper bound on the number of leaves $L(k)$ in the search tree. We shall assume that the instance $(\mathcal{F}, R, k)$ is strongly reduced before every branch of the algorithm, and that the branches are considered in the listed order. We first make the following observations.

Observation 2. Let T be a tree in $\mathcal{F}$ rooted at r, let $w \neq r$ be an important vertex in T, and let $S \subseteq children(w)$ be such that S is contained in some minimum vertex cover of G_w. If edge $w\pi(w)$ is in some minimum cut of T, then the edges incident to the leaves of any minimum vertex cover of G_w are contained in some minimum cut: simply replace all the edges that are incident to the children of w in a minimum cut that contains $w\pi(w)$ with the edges incident to the leaves corresponding to the desired minimum vertex cover of G_w. Since S is contained in some minimum vertex cover of G_w, there is a minimum cut that contains the edges that are incident to the (leaf) children of w that are in S. Therefore, if we choose edge $w\pi(w)$ to be in the solution, then we can choose the edges in $\{wu \mid u \in S\}$ to be in the solution as well. If we choose to cut the children of w that are in S when we cut edge $w\pi(w)$ in a branch, then we say that we *favor* the vertices in S in this branch; this observation will be very useful when branching. If S consists of a single vertex u, we simply say that we *favor* vertex u. Note that if we favor the vertices in S, then using the contrapositive of the statement "if w is cut then the vertices in S are cut", if we decide not to cut a *certain* vertex in S in a branch, then we can assume that w will not be cut as well in the same branch. If we decide not to cut an edge in a certain branch, we say that the edge is *kept* and we can contract it.

Observation 3. Let T be a tree in $\mathcal{F}$ and let $w \in V(T)$ be an important vertex. Let $v \in G_w$, and recall that $deg_G(v)$ denotes the degree of v in G_w. By Lemma 9, we can assume that the set of edges in T_w that are contained in the solution that we are looking for corresponds to a minimum vertex cover of G_w. Since any minimum vertex cover of G_w either contains v, or excludes v and contains its neighbors, we can branch by cutting v in the first side of the branch, and by cutting the neighbors of v in G_w in the second side of the branch. Note that by part (iv) of Proposition 1, and the fact that there is no request between a child and its parent (unit request rule), there must be at least one request between v and another child of w, and hence, $deg_G(v) \geq 1$. Therefore, we have:

BranchRule 4. *Let T be a tree in $\mathcal{F}$, and let $w \in V(T)$ be an important vertex. If there exists a vertex $v \in G_w$ such that $deg_G(v) \geq 3$, then branch by cutting v in the first side of the branch, and by cutting the neighbors of v in G_w in the second side of the branch. Cutting v reduces the parameter k by 1, and cutting the neighbors of v in G_w reduces k by at least 3. Therefore, the number of leaves in the search tree of the algorithm, $L(k)$, satisfies $L(k) \leq L(k-1) + L(k-3)$.*

We can now assume that for any important vertex w, we have $\Delta(G_w) \leq 2$, and hence, G_w consists of a collection of disjoint paths and cycles. Let T be a tree in $\mathcal{F}$ rooted at r. Among all important vertices in T, let w be a vertex that is farthest from r. Since every subtree of T contains an important vertex, w must be a farthest vertex among all internal vertices of T. By part (ii) of Proposition 1, there exists a cross request between a vertex in $V(T_w)$ and a vertex in $V(T_{\pi(w)}) \setminus V(T_w)$. Since w is farthest from r, the cross request between a vertex in $V(T_w)$ and a vertex in $V(T_{\pi(w)}) \setminus V(T_w)$ can be either a request: (1) between w and a sibling of w, (2) between w and a nephew of w, (3) between a child of w and its grandparent $\pi(w)$, (4) between a child of w and an uncle, or (5) between a child of w and a cousin. By symmetry (and by the choice of w), the case when there is a request between w and a nephew is identical to the case when there is a request between a child of w and an uncle. Therefore, we shall only treat the latter case.

Case 1. **Vertex w has a cross request to a sibling w'.** In this case at least one of w, w' must be cut. We branch by cutting w in the first side of the branch, and cutting w' in the second side of the branch. Note that by part (iii) of Proposition 1, the size of a minimum vertex cover in G_w is at least 1. Moreover, a minimum vertex cover for G_w can be computed in polynomial time since $\Delta(G_w) \leq 2$. Therefore, in the first side of the branch we end up cutting the edges corresponding to a minimum vertex cover of G_w, which reduces the parameter further by at least 1. Therefore, we have $L(k) \leq L(k-2) + L(k-1)$.

Case 2. **There is a child u of w that has a cross request to its grandparent $\pi(w)$.** In this case we can cut u. This can be justified as follows. Any minimum cut of T either cuts $w\pi(w)$ or does not cut it. If the minimum cut cuts $w\pi(w)$, then we can assume that it cuts edge wu as well because by Reduction Rule 14, u is in some minimum vertex cover of G_w. On the other hand, if the

minimum cut does not cut $w\pi(w)$, then it must cut edge wu since $(u, \pi(w)) \in R$. It follows that in both cases there is a minimum cut that cuts wu. We have $L(k) \le L(k-1)$ in this case.

Case 3. **There is a child u of w such that u has a cross request to an uncle w'.** We favor u and branch as follows. In the first side of the branch we cut u. In the second side of the branch we keep edge uw, and cut the neighbor(s) of u in G_w. Since u is not cut in the second side of the branch and u is favored, w is not cut as well, and hence w' must be cut. Noting that u has at least one neighbor in G_w, $L(k)$ satisfies $L(k) \le L(k-1) + L(k-2)$.

Case 4. **There is a child u of w such that u has a cross request to a cousin u'.** Let $w' = \pi(u')$ and note that $\pi(w) = \pi(w')$. We favor u and u'. We branch as follows. In the first side of the branch we cut u. In the second side of the branch uw is kept and we cut the neighbor(s) of u in G_w. Since in the second side of the branch uw is kept and u is favored, $w\pi(w)$ is kept as well, and u' must be cut (otherwise, w' is not cut as well because u' is favored) since $(u, u') \in R$. Therefore, $L(k)$ satisfies $L(k) \le L(k-1) + L(k-2)$.

Theorem 5. *The* MULTICUT *problem can be solved in time $O^*(\rho^k)$, where $\rho = (\sqrt{5} + 1)/2 \approx 1.618$ is the positive root of the polynomial $x^2 - x - 1$.*

References

1. Abu-Khzam, F.A., Collins, R., Fellows, M., Langston, M., Suters, W., Symons, C.: Kernelization algorithms for the vertex cover problem: theory and experiments. In: Proceedings of ALENEX, pp. 62–69 (2004)
2. Bousquet, N., Daligault, J., Thomassé, S.: Multicut is fpt. In: CoRR, abs/1010.5197, 2010 (to appear in STOC 2011)
3. Bousquet, N., Daligault, J., Thomassé, S., Yeo, A.: A polynomial kernel for multicut in trees. In: Proceedings of STACS, pp. 183–194 (2009)
4. Buss, J., Goldsmith, J.: Nondeterminism within P. SIAM Journal on Computing 22, 560–572 (1993)
5. Costa, M., Letocart, L., Roupin, F.: Minimal multicut and maximal integer multiflow: A survey. European Journal of Operational Research 162(1), 55–69 (2005)
6. Downey, R., Fellows, M.: Parameterized Complexity. Springer, New York (1999)
7. Garg, N., Vazirani, V.V., Yannakakis, M.: Primal-dual approximation algorithms for integral flow and multicut in trees. Algorithmica 18(1), 3–20 (1997)
8. Guo, J., Niedermeier, R.: Fixed-parameter tractability and data reduction for multicut in trees. Networks 46(3), 124–135 (2005)
9. Marx, D.: Parameterized graph separation problems. Theoretical Computer Science 351(3), 394–406 (2006)
10. Marx, D., Razgon, I.: Fixed-parameter tractability of multicut parameterized by the size of the cutset. In: CoRR, abs/1010.3633, 2010 (to appear in STOC 2011)
11. West, D.B.: Introduction to graph theory. Prentice Hall Inc., Upper Saddle River (1996)

Beyond Triangulation: Covering Polygons with Triangles

Tobias Christ

Institute of Theoretical Computer Science, ETH Zürich, Switzerland

Abstract. We consider the *triangle cover problem*. Given a polygon P, cover it with a minimum number of triangles contained in P. This is a generalization of the well-known polygon triangulation problem. Another way to look at it is as a restriction of the convex cover problem, in which a polygon has to be covered with a minimum number of convex pieces. Answering a question stated in the Handbook of Discrete and Computational Geometry, we show that the convex cover problem without Steiner points is NP-hard. We present a reduction that also implies NP-hardness of the triangle cover problem and which in a second step allows to get rid of Steiner points. For the problem where only the boundary of the polygon has to be covered, we also show that it is contained in NP and thus NP-complete and give an efficient factor 2 approximation algorithm.

1   Introduction

A triangulation of a polygon P is a set of triangles such that their union is P and the intersection of two triangles is either empty, a common vertex or an edge. It is well-known that triangulations always exist and how to construct one for a given polygon. Furthermore, it is possible to triangulate every polygon without introducing any new vertices that have not been polygon vertices before, which for a simple polygon with n vertices always results in a triangulation consisting of $n-2$ triangles. So usually one forbids new vertices (Steiner points) right away and insists that the edges of the triangles are either diagonals or polygon edges.

Depending on the application, we might not care about the way the triangles can intersect, which leads to the natural problem of describing a polygon as a union of arbitrary triangles. A triangulation is always a solution to this problem, but having dropped the second condition, we might be able to find such a description of a simple polygon using less than $n-2$ triangles. We can see this as a variant of the *polygon cover problem*: one wants to find a set of *primitive* polygons inside a polygon P such that they cover a certain subset of P. In the usual variant, this subset is the whole interior of P itself, in the *boundary cover problem* just the boundary ∂P and in the *vertex cover problem* the vertex set $V(P)$. Several ways of specifying different primitives have been proposed. The classical art gallery problem is nothing else than the problem of covering a polygon with star-shaped polygons. Beside that, pseudotriangles, spiral, convex, and monotone polygons have been studied as possible primitives [1]. Using triangles as primitives we arrive at the same problem as described above. Usually, one

wants to minimize the number of primitives in a solution. Here, we formulate it as a decision problem. *The Triangle Cover Problem:* Given a polygonal region P and a positive integer k, are there triangles $t_1, \ldots, t_k$ such that $P = \bigcup_{i=1}^{k} t_i$?

Refining the proof of Culberson and Reckow [2] for the convex cover problem, we show NP-hardness of the triangle cover problem. The reduction can be shown to be gap-preserving, implying APX-hardness. Then we modify the reduction to settle the complexity of the convex cover problem without Steiner points, a question not addressed by Culberson and Reckow and appearing as an open problem in the Handbook of Discrete and Computational Geometry [3].

As it is the case for many geometrical problems (e.g. the minimum-weight triangulation problem [4]), it is not clear whether these covering problems are in NP (see the remarks in O'Rourke [5], p. 232 or in Culberson and Reckow [2] in the conclusion). Forbidding Steiner points, the covering problems obviously are in NP, as we can describe a solution in terms of the polygon vertices and check it efficiently. The typical decomposition problems for rectilinear polygons are trivially in NP, too. But for the general variants of the cover problems allowing Steiner points, it is not obvious how a solution can be described in terms of the input. The situation changes if the primitives are required to cover the boundary of the input region only. *The Boundary Cover Problem:* Given a polygonal region P and an integer k, are there triangles $t_1, \ldots, t_k \subseteq P$ such that $\partial P \subset \bigcup_{i=1}^{k} t_i$? As for the convex boundary cover problem [2], the boundary cover problem with triangles remains NP-hard. But interestingly, it can be shown to be in NP. The question remains whether we can find efficient approximation algorithms. The usual approaches for covering problems boil down to finding an abstract set cover greedily and therefore lead to factor $O(\log n)$ approximation algorithms [6,7]. But in the context of triangles, this is not satisfactory. If the input polygon does not have any collinear edges, a triangle can cover at most three polygon edges. So simply triangulating is a factor 3 approximation algorithm for the problem of minimizing the number of triangles in a cover, both for the boundary and the general cover problem. Having shown APX-hardness on the other side, there remains a small range for interesting approximation ratios only. For the boundary cover problem, we give an efficient factor 2 approximation algorithm.

Related Work. It has been known for a long time that covering a polygon P with a minimal number of convex polygons is NP-hard, as shown in 1983 by O'Rourke and Supowit [8] along with other polygon cover problems. In 1988, Culberson and Reckow [2] gave another NP-hardness proof, which also works for polygons without holes. In 2001, Eidenbenz and Widmayer [7] presented a $O(\log n)$-approximation algorithm for the convex cover problem and furthermore, they observed that the reduction of Culberson and Reckow also implies APX-hardness of the problem. If we look at the problem of *partitioning* a polygon instead of covering it, the situation changes. We say a set of polygons is a *partition* of P if they are pairwise internally disjoint and their union equals P. It is possible to partition a simple polygon into convex pieces in time $O(n^3)$ using dynamic programming [9,10]. If we allow the polygons to contain holes, the problem turns NP-hard as shown by Lingas in 1983 [11]. The proof by Lingas uses

Table 1. Time complexity of decomposing a polygon on n vertices. The mark $*$ indicates the results of this paper. GP stands for general position.

	Steiner pts.	convex	triangles
cover interior	No	NP-complete [*]	NP-complete (GP: $O(n \log n)$) [*]
	Yes	NP-hard [2]	NP-hard [*]
cover boundary	Yes	NP-hard [2]	NP-complete [*] (GP: open)
partition	No	$O(n^3)$ [13]	$O(n^3)$ [12] (GP: $O(n)$ [14])
simple polygons	Yes	$O(n^3)$ [9]	open [12]
partition polygon	No	NP-cpl. [8]	NP-cpl. [11] (GP: $O(n \log n)$ [12])
with holes	Yes	NP-hard [11]	NP-hard [11]

Steiner points. But as he points out, the NP-hardness of the convex partition problem disallowing Steiner points follows using the reduction of O'Rourke and Supowit [8] reducing from planar 3SAT. If we ask for a partition into triangles instead of convex polygons, it is known that it is possible to find a minimal partition efficiently if we do not allow Steiner points, as shown by Asano, Asano, and Pinter [12] and later refined by Chen and Chang [10]. Note that if we restrict to polygons in general position and still do not allow Steiner points, the problem of partitioning a polygon into triangles is equivalent to the well-known problem of triangulating a polygon. If we consider polygons with holes, again the problem turns NP-hard as shown by Lingas [11]. The different results are summarized in Table 1. In this work we close many gaps left by the research in the 1980's, but note that there are open problems still. In particular, it remains open if more of the "NP-hard" entries in the table can be replaced by "NP-complete".

Notation. We think of polygons as closed subsets of the plane including the interior, i.e., a simple polygon P is a simply connected closed subset of the plane whose boundary ∂P is a simple cycle of line segments. Simple means that nonadjacent segments do not intersect and adjacent segments at their common endpoint only. A polygon P is in *general position* if no more than two of its vertices lie on a common line. We denote the edges of P by $E(P)$, the vertices by $V(P)$ and $n(P) = |V(P)|$. By a *polygonal region* we mean a closed subset P of the plane such that its boundary ∂P is a finite disjoint union of simple cycles of line segments. A polygonal region may consist of several connected components, each of which being a polygon possibly containing holes.

2   NP-Hardness of the Triangle Cover Problem

To show NP-hardness we reduce the Boolean satisfiability problem (SAT) to the triangle cover problem. Let F be a CNF formula with clauses $C_1, \ldots, C_m$ on the variables $x_1, \ldots, x_n$. We call the number of clauses a variable x_i appears in the *degree* of x_i and denote it by $\deg(x_i) := |\{C_j : x_i \in C_j \text{ or } \overline{x_i} \in C_j\}|$.

We define different gadgets, all of which are simple polygons. We will glue them to the boundary of a big triangle to form one big simple polygon. The first
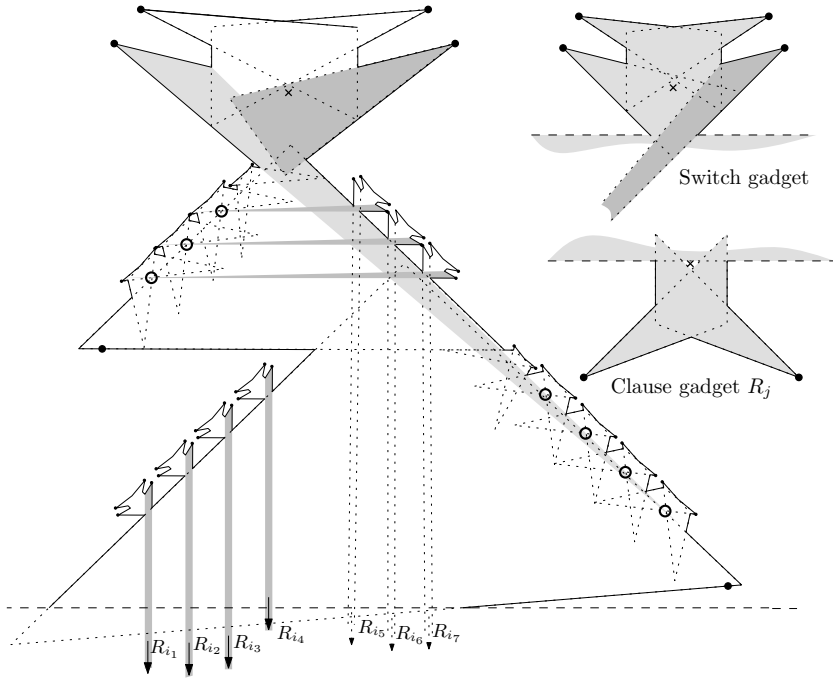
Fig. 1. A variable gadget Q_i for x_i with $\deg x_i = 7$, $x_i \in C_{i_1}, C_{i_2}, C_{i_3}, C_{i_4}$ and $\overline{x_i} \in C_{i_5}, C_{i_6}, C_{i_7}$. Special points are marked by dots, uncovered spots by disks.

basic ingredient are the *switch gadgets*, see Fig. 1. Such a switch gadget can be covered optimally in essentially two ways. Thus, it can be used to encode the truth value of a variable. Depending on how it is covered, one of the triangles in its cover can be extended through the opening (i.e., where it is going to get glued to the rest) to cover additional points far away. The other basic ingredient is the *clause gadget*. We define a clause gadget R_j for every clause C_j. If we use two triangles to cover it, there remains a small spot uncovered (depicted by a cross). For every variable x_i, we define a variable gadget Q_i, which consists of a big *main switch* gadget and $\deg(x_i)$ many *small switches*. To the opposite of every small switch, we put a small clause-gadget-like structure: like the clause gadgets, they can be covered with two triangles leaving an uncovered spot. But in contrast to the clause gadgets, where the uncovered spots are shielded away from each other, now they lie more to the interior. The small switches corresponding to clauses C_j with $x_i \in C_j$ are put to the bottom left of the main switch, the small switches corresponding to clauses C_j with $\overline{x_i} \in C_j$ are put to the right. Depending on how the main switch is covered, namely which of the possible triangles gets stretched out, either the uncovered spots of the left clause-gadget-like structures or the uncovered spots of the right clause-gadget-like structures get covered. If the spots at the bottom right get covered, the small switches at the bottom left that correspond to positive occurrences of x_i can be used to cover

the small spot left by the corresponding clause gadgets, but the small switches at the top right (corresponding to negative occurrences of x_i) cannot cover their corresponding clause gadget, because we are forced to extend their other free triangle so they can cover the small spots left by their own clause-gadget-like structures at the top left. If the spots at the top left get covered by the main switch, then the exact opposite happens: the small switches at the top right can be used to cover their corresponding clause gadgets, whereas the small switches at the bottom left are bound to stay inside the variable gadget. (In addition to the clause-gadget-like structures leaving an uncovered spot, there are two more smaller such structures that do not leave an uncovered spot and are merely used to connect the others with the main switch leaving room for the triangle of the main switch if it should get extended.) Finally, glue the variable gadgets to the upper edge and the clause gadgets to the lower edge of a huge triangle and adjust the small switches in the variable gadgets such that their free triangle can cover what it is supposed to cover, namely exactly the uncovered spot of the specific clause gadget the small switch stands for. The whole procedure results in a simple polygon $\phi(F)$. Note that $\phi(F)$ is not in general position as there are many collinear edges. It is possible to get rid of them, but for lack of space, we do not give the details here.

Lemma 1. *If F is satisfiable, then $\phi(F)$ can be covered with k triangles.*

Proof. Depending on the truth value of x_i we cover the variable gadget Q_i either extending the left or the right triangle of the main switch with $6 \deg(x_i) + 10$ triangles as shown in Fig. 1, and the clause gadgets with two triangles each. As F is satisfied by the assignment, for every clause there is at least one variable gadget that is covered in such a way that there is a long thin triangle that extends to the uncovered spot of the corresponding clause gadget. The huge space between variable and clause gadgets can be covered by one additional triangle. We get a cover with $k = \sum_{i=1}^{n}(6 \deg(x_i) + 10) + 2m + 1$ triangles. $\qquad \square$

Lemma 2. *If $\phi(F)$ can be covered with k triangles, then F is satisfiable.*

Proof. Fix *special points* as shown in Fig. 1 by disks. Additionally, put one special point somewhere into the huge triangle connecting everything invisible to any other special point. Notice that all special points are pairwise invisible. No two of them can be covered by the same triangle. There are exactly k special points, so each triangle covers exactly one special point. Now we derive a satisfying assignment. Look at Q_i. In the middle of its main switch there is a *central point* denoted by a cross Fig. 1. At least one triangle covers it. Now the only special points that are visible from there are the leftmost and the rightmost convex vertex of the main switch. So the only triangles that are able to cover the cross are those corresponding to these two special points. At most one of the two does *not* cover the central point, so we define the truth value of x_i accordingly: If the triangle corresponding to the left special point does not cover the central point, we define x_i to be positive, else we set x_i to negative. Now we claim that this yields a satisfying assignment of F. Look at a clause gadget

R_j. We show that C_j is satisfied. There is a triangle t in the cover that covers the *central point* of R_j, again depicted by a cross. Now the only special points visible from the central point of R_j are those belonging to small switches of corresponding variable gadgets, i.e., variable gadgets Q_i such that $x_i \in C_j$ or $\overline{x_i} \in C_j$. So t corresponds to a small switch gadget of a variable gadget, say Q_i, and assume without loss of generality $x_i \in C_j$. The central point of the small clause-gadget-like structure that lies opposite of this small switch to the right must be covered by a triangle, call it t'. Now because t covers the central point of the clause gadget, it cannot cover the central point of the small switch gadget, which therefore has to be covered by the other free triangle of the small switch, which cannot be t'. So t' must belong to the main switch of Q_i and it cannot cover the central point, which implies that the variable x_i is positive.   □

3   Inapproximability

Lemma 3. *If F is a 3-CNF where each variable has degree at most 13, then ϕ is gap-preserving: If F is satisfiable, $\phi(F)$ can be covered with $k = \sum_{i=1}^{n}(6\deg(x_i) + 10) + 2m + 1$ triangles. If at most a $1 - \delta$ fraction of the clauses of F can be satisfied, then we need at least $(1 + \delta/144)k$ triangles to cover $\phi(F)$.*

Proof. Let F be a 3-CNF where every variable has degree 13 and at most $(1-\delta)m$ of the clauses can be satisfied. Assume for contradiction that $\phi(F)$ can be covered by less than $(1+\delta/144)k$ triangles. Consider a cover as in the proof of Lemma 2. But now there might be up to $\delta k/144$ more triangles than special points. We define a triangle to be *uncontrollable* if it does not cover any special point. If there are several triangles covering the same special point, we arbitrarily pick one to be responsible for the special point and call the other triangles uncontrollable as well. Consider a variable gadget Q_i. If one of the uncontrollable triangles intersects Q_i, it might happen that Q_i is both positive and negative, in the sense that both triangles of the small switches to the left and of the small switches to the right extend down to the corresponding clause gadgets. There are strictly less than $\delta k/144$ uncontrollable triangles, so at most that many variable gadgets are *ambiguous* in that they cover both clause gadgets they appear positively in and clause gadgets they appear negatively in. So from the guarding we only get a partial truth value assignment α. Go through the remaining unset variables and set them to true or false depending on which satisfies more of the remaining clauses. For each x_i, $\deg(x_i) \leq 13$. So there are at most 13 unsatisfied clauses x_i appears in. At most 6 of those remain unsatisfied after choosing a truth value for x_i. This leads to a complete assignment α', which satisfies at least a $1-\delta$ fraction of the clauses: Look at a clause gadget R_j. There must be a triangle covering its central point. Either there is one of the uncontrollable triangles directly covering it (which it can do for at most one clause gadget) or there is a triangle of a small switch of a variable gadget that covers it. If the latter is the case, then either this variable gadget is unambiguous and so the clause already got satisfied by the partial assignment α or the variable gadget is ambiguous and then there is one of the uncontrollable triangles responsible for it. An uncontrollable triangle

can either cover at most one central point of a clause gadget or it can cause at most one variable gadget to be ambiguous. If a variable is ambiguous, after choosing the better truth value for it, it leaves at most 6 clauses unsatisfied. Therefore, an uncontrollable triangle is responsible for at most 6 clauses that remain unsatisfied by α. At most $6\delta k/144 = \delta k/24$ clauses remain unsatisfied. An easy calculation shows $k \leq 24m$. $\square$

Theorem 1. *There is a constant $\delta > 0$ such that it is NP-hard to approximate the triangle cover problem within a factor smaller than $1 + \delta$.*

Proof. For every $\varepsilon > 0$, there is a polynomial reduction from 3SAT to MAX3SAT that maps satisfiable to satisfiable formulas and unsatisfiable to formulas where at most a $7/8 + \varepsilon$ fraction of the clauses can be satisfied. Furthermore, there is a reduction from MAX3SAT to MAX3SAT(13) such that satisfiable formulas remain satisfiable and if at most a $1 - \gamma$ fraction is satisfiable, at most a $1 - \gamma/19$ fraction is satisfiable in the reduced formula, for any $\gamma > 0$ [15]. So putting these two reduction and ϕ together, we get a reduction from SAT to triangle cover such that satisfiable formulas get mapped to polygons coverable with k triangles. And unsatisfiable formulas get mapped to polygons where every cover needs at least $(1+\delta/144)k = (1+\gamma/(19 \cdot 144))k = ((1+(1/8-\varepsilon)/2736)k = (1+1/19152-\varepsilon')k$ triangles, for a constant ε' we can make arbitrarily small. $\square$

4   Covering without Steiner Points

The Non-Steiner Triangle Cover Problem: Given a polygonal region P and an integer k, are there triangles $t_1, \ldots, t_k$ with $V(t_i) \subset V(P)$ such that $P = \bigcup_{i=1}^{k} t_i$?

Theorem 2. *If P is a polygon in general position with h holes, then a Non-Steiner triangle cover of P contains at least $n(P) + 2h - 2$ triangles. If it has exactly $n(P) + 2h - 2$ triangles, then it is a triangulation of P.*

Proof. Assume we have a Non-Steiner triangle cover of P consisting of k triangles. Let $v \in V(P)$ and t be a triangle of the cover with $v \in t$. Because of the general position assumption, this implies $v \in V(T)$. Define $\alpha(v)$ as the interior angle of the polygon at a vertex v. The (interior) angles at v of all triangles covering v summed up must be at least $\alpha(v)$. Summing up this inequality over all polygon vertices we get $(n + 2h - 2)\pi = \sum_{v \in V(P)} \alpha(v) \leq k\pi$. If we have equality, then no two triangles overlap at any vertex v. We want to show that then it is a triangulation. Assume there are two triangles t and t' that overlap. A vertex of t' cannot be in the interior of t, neither can it be in the relative interior of an edge of t. So we find a vertex $v \in V(t)$ such that there is an edge $e \in E(t')$ intersecting both edges of t incident to v. Pick for a fixed t the edge e of another triangle t' such that there is no other triangle edge crossing t between e and v. Now look at the edge e. It must be a diagonal of P. On one of its sides there is t', on the other side there is another triangle t'' it is an edge of. t'' has a third vertex w not incident to e. If $w = v$, we have a contradiction, because then t and t'' overlap at v, if $w \neq v$, we get a contradiction to the choice of e. $\square$

This theorem shows that in the case of general position, an optimal Non-Steiner cover is equivalent to a triangulation. A triangulation can be found in linear time in the case of simple polygons [14] or in time $O(n \log n)$ for a polygon with holes [12]. However, if we drop the general position assumption, there are examples of polygons that can be covered by fewer than $n-2$ triangles, think of the Star of David for example. It turns out that finding an optimal Non-Steiner cover is NP-hard if we allow collinear edges. To prove this, we basically use the same reduction as before, we just have to get rid of all Steiner points. We adjust $\phi(F)$ such that every triangle that appears in the solution as given in Lemma 1 has all vertices on $\partial\phi(F)$. Instead of only allowing polygon vertices to be triangle vertices, we fix a finite set W with $V(P) \subset W \subset \partial P$, which are the allowed triangle vertices in a solution. We call W the set of *generalized vertices*.

Theorem 3. *If P is a polygon and $V(P) \subset W \subset \partial P$ a finite set, then we can compute a polygon P' in polynomial time, which is coverable with $k+|W \setminus V(P)|$ Non-Steiner triangles iff P can be covered by k triangles using only points in W.*

Proof. Let $m := |W \setminus V(P)|$. Scale P such that its diameter is 1. Draw a line through every pair of points in W. Let $\varepsilon > 0$ be smaller than the area of every 2-dimensional cell of this arrangement. Construct P' as follows. For every $w \in W \setminus V(P)$ we attach a small spike such that the interior angle of the spike equals $\arctan(\varepsilon/(k+m))$, and the width of the spike is at most $\varepsilon/(100(k+m))$ and small enough to make the spike fit without intersecting the polygon. If necessary, tilt it a little bit such that the sector we get by prolonging the two edges of the spike does not contain any point of W. First we observe that a k-cover of P using only vertices in W carries over to a $(k+m)$-cover of P': Keep all the triangles as they are and add one for every spike. Now assume that P has no k-cover on W. So every possible set of k triangles in P using vertices from W leaves at least one cell of the arrangement uncovered. Assume for contradiction that P' has a Non-Steiner $(k+m)$-cover. At least one triangle has to cover the tip of a spike. No triangle can cover two tips. All these m triangles have area strictly less than $\varepsilon/(k+m)$. So after removing them we adjust the remaining triangles that have vertices on a spike-vertex by replacing these vertices by the point the spike came from. This can decrease the area of a triangle by at most $\varepsilon/(k+m)$ (every vertex of the triangle gets moved by at most $\varepsilon/(100(k+m))$). We get a cover of P that for sure leaves less than an $(k+m)\varepsilon/(k+m) = \varepsilon$ area uncovered. □

Theorem 4. *The Non-Steiner triangle cover problem is NP-complete.*

Proof. We adapt the reduction in Sect. 2 such that all triangle vertices lie on $\partial\phi(F)$. See Fig. 2. This reduces SAT to the cover problem with a set of generalized vertices. We apply Theorem 3, yielding a polygon $\phi'(F) := (\phi(F))'$ where all the generalized vertices have been replaced by spikes. $\phi'(F)$ can be covered with a certain number k of triangles if and only if the formula F is satisfiable. □

Observe that this reduction works for convex covers as well, thus resolving a question stated in the Handbook of Discrete and Computational Geometry [3].
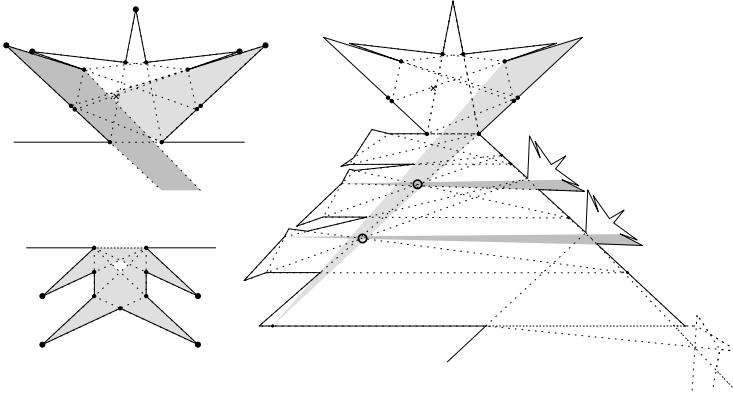
Fig. 2. How the gadgets can be adapted to get rid of Steiner points

The reduction yields a polygon $\phi'(F)$ with collinear edges. But whereas in the case of triangles the problem becomes easy if we ask for general position, in the convex setting we can change the points slightly to achieve general position and the problem remains NP-hard. Due to space limitations we omit the proof.

5   Covering the Boundary

We consider the setting where only the edges of P have to be covered by triangles, which, however, still have to be inside P. We say a line segment s is *covered* by a triangle $t \subseteq P$ if t intersects s in more than just one point. If $s \subset t$, we say t covers s *completely*. Otherwise, if only a subsegment of s is contained in t, we say t covers s *partly*. In a cover of P, every edge has to be covered completely by some triangle or partly by at least two triangles. In this setting, it is convenient to allow triangles to be degenerate: line segments are also considered to be triangles.

Theorem 5. *The boundary cover problem is NP-complete.*

Proof. We show that the problem is in NP by restricting the set of possible triangles in a solution. We define a triangle contained in a polygonal region P to be *nice* if each of the three lines defined by its edges contains at least two vertices of P. So nice triangles have a compact description in terms of the input. We call a triangle *problematic* if it covers two polygon edges $e, f \in E(P)$ only partly and one of its edges a has its startpoint in the relative interior of e and its endpoint in the relative interior of f and does not cover any polygon edges. Note that problematic triangles are not nice in general. Two of their defining lines are given by polygon edges (namely by e and f) whereas the third triangle edge a can be assumed to contain one polygon vertex $v \in V(P)$ (if not, just move a outwards until it does) but not more. If we rotate a around v keeping the other two defining lines fixed, the triangle covers more of, say, e while it covers less of f, see Fig. 3. We call $a = a(t)$ the *free edge* of the problematic triangle t.

Fig. 3. A problematic triangle and how its free edge (fat) gets rotated to make it fair

Let t be a triangle in a boundary cover of P. Replace t by a triangle t' such that $t \cap (\partial P \setminus V(P)) \subseteq t' \subseteq P$ and t' is nice or problematic. If each edge of t covers a polygon edge (partly or completely), then t is nice and we set $t' = t$. If an edge of t does not cover any ed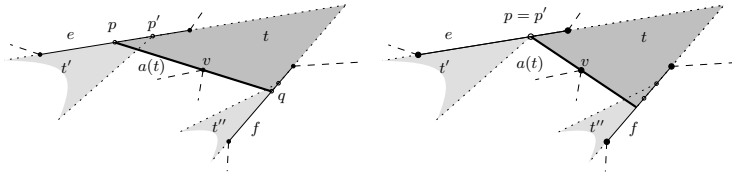ge of P, then the only points of ∂P it contains are vertices of P. If only one of the edges of t covers a polygon edge, then we can replace t by the degenerate triangle t' which only consists of this one edge, and therefore is nice. If t has exactly one edge a that does not cover any edge of P, then either t is problematic and we set $t' = t$ or we can shrink t by moving a and keeping the other two defining lines fixed until at least one of the endpoints of a is on a polygon vertex $v \in V(P)$. Then we enlarge t again by rotating a around v and keeping the rest fixed, until it contains a second polygon vertex. This results in a triangle t' which is nice and $t \cap (\partial P \setminus V(P)) \subseteq t'$. From now on we assume that there are nice or problematic triangles only.

Let t be a problematic triangle whose free edge $a(t) = \overline{pq}$ does not cover any edge and the other two edges cover the polygon edges $e, f \in E(P)$ partly, $p \in e$ and $q \in f$. In the cover there has to be another triangle t' also covering e and $p \in t'$. And there has to be a third triangle t'' covering f and $q \in t''$. We may assume that a contains exactly one polygon vertex $v \in V(P)$ in its relative interior. (If not, blow t up.) If one of the triangles t' or t'' is nice, shrink the part of e covered by t (of f, respectively) rotating a around v until $t \cap t' \cap e$ ($t \cap t'' \cap f$, respectively) is a single point, thus increasing the part it covers on the other side. Then, t is not necessarily nice, but it has a compact description: two of the lines defined by the edges are directly given by polygon vertices and the third line through a is now defined by a polygon vertex and a vertex of another triangle in the cover. See Fig. 3. We call such a triangle that can be described by five polygon vertices and one vertex of another triangle *fair*. It might happen that both t' and t'' are problematic as well. In this case, we say that t *depends* on both t' and t''. Formally, we call two problematic triangles t and t' *dependent*, if they cover a common edge $e \in E(P)$ and $t \cap t' \cap e = \overline{pp'}$, where $p = e \cap a(t)$ and $p' = e \cap a(t')$ are the endpoints of the free edges of t and t' (possibly $p = p'$). This defines an abstract graph on the problematic triangles, in which each triangle has degree at most 2. If a problematic triangle has degree at most one in this graph, we can make it fair as described above, and then remove it from the graph. Let $t_1, \ldots, t_k$ be a cycle in the graph. We replace the triangles $t_1, \ldots, t_k$ by degenerate triangles $b_1, \ldots, b_k$, where b_i is defined as the line segment which is the union of the edges of t_i and t_{i+1} that together cover a polygon edge.

So we may assume that every triangle is either nice or fair and that there are no cyclic dependencies among the fair ones. Constructing the triangles involves intersecting lines. Computing the coordinates of a fair triangle using another fair triangle, only one of the points comes from this other triangle and the other points involved are in $V(P)$. So the bit size of the coefficients of the defining lines only grows by an additive term in each step (and this term is bounded by the input size). So we have replaced the arbitrary boundary cover by a cover consisting of triangles which can be described in polynomial space. Given such a cover, it can be checked in polynomial time if it covers P. For lack of space, we omit the NP-hardness proof, which is similar to the one in Sec. 2.   □

Theorem 6. *Given a polygonal region P, we can find a boundary cover in time polynomial in $n(P)$ using at most twice as many triangles as an optimal solution.*

Proof. We define two edges $e, f \in E(P)$ to be equivalent if there is a line segment $s \subset P$ such that both $e \subset s$ and $f \subset s$. (One might also require e and f to have P on the same side, but for the proof it does not matter.) This defines an equivalence relation on $E(P)$. We call the classes *merged edges* and denote them by E', $n' := |E'|$. We define an abstract graph G on E': $e, f \in E'$ are adjacent if there is a triangle $t \subset P$ such that both $e \subset t$ and $f \subset t$. If P is a simple polygon, G can be constructed in time $O(n^2)$ by constructing the visibility graph of $V(P)$ first. If there are holes, in time $O(n^2 \log n)$: Fix a merged edge e and find the neighbors of e in G in time $O(n \log n)$. Pick a point p in the interior of e such that $p \in \partial P$. Construct the visibility polygon W with respect to p, which can be done using an angular sweep around p in time $O(n \log n)$ [16]. Then preprocess W to allow ray-shooting queries in $O(\log n)$ time, which can be done in time $O(n \log n)$ as well, using a geodesic triangulation [17]. Finally, go through the edges of W. If we find an edge f of W which is also a complete edge of P, we have to find out if there is a triangle in W containing e and the merged edge f is part of, which can be done by two ray-shooting queries.

The second step is to find a maximum matching M in G, which can be done in time $O(n^{2.376})$ using the algorithm by Mucha and Sankowski [18]. Now M corresponds to a set of triangles T such that in total, the triangles in T cover at least $2|M|$ merged edges completely. For all the remaining $n' - 2|M|$ merged edges, we choose its own triangle to cover it and add it to T. Now the triangles in T cover all merged edges completely and therefore the whole boundary of P. So we have found a boundary cover of P consisting of $n' - |M|$ triangles. The running time of the algorithm is dominated by the computation of the matching.

Let a be the number of triangles in a fixed optimal solution that cover at least 2 merged edges completely and exclusively (i.e., no other triangle covers it). Initially, every merged edge gets a charge of 1. Now we distribute this charge evenly among all triangles that cover this merged edge. A triangle can cover at most 3 merged edges. Therefore a triangle contributing to a gets charged at most 3 and all other triangles get charged at most 2 (namely at most 1 from a merged edge possibly covered completely, and at most $1/2$ from two other edges). So we get $3a + 2(OPT - a) \geq n'$, where OPT is the number of triangles in an optimal solution. This implies $a \geq n' - 2OPT$. $|M| \geq a$, therefore $n' - |M| \leq 2OPT$.   □

References

1. Keil, J.M.: Polygon decomposition. In: Handbook of Computational Geometry, pp. 491–518. North-Holland, Amsterdam (2000)
2. Culberson, J.C., Reckhow, R.A.: Covering polygons is hard. J. Algorithms 17(1), 2–44 (1994)
3. O'Rourke, J., Suri, S.: Polygons. In: Goodman, J.E., O'Rourke, J. (eds.) Handbook of Discrete and Computational Geometry, pp. 583–606. CRC Press, LLC, Boca Raton, FL (2004)
4. Mulzer, W., Rote, G.: Minimum weight triangulation is NP-hard. In: Proc. 22nd Annu. ACM Sympos. Comput. Geom., pp. 1–10 (2006)
5. O'Rourke, J.: Art gallery theorems and algorithms. International Series of Monographs on Computer Science. The Clarendon Press Oxford University Press, New York (1987)
6. Ghosh, S.K.: Approximation algorithms for art gallery problems in polygons. Discrete Appl. Math. 158(6), 718–722 (2010)
7. Eidenbenz, S.J., Widmayer, P.: An approximation algorithm for minimum convex cover with logarithmic performance guarantee. SIAM J. Comput. 32(3), 654–670 (2003) (electronic)
8. O'Rourke, J., Supowit, K.J.: Some NP-hard polygon decomposition problems. IEEE Trans. Inform. Theory IT-30, 181–190 (1983)
9. Chazelle, B., Dobkin, D.P.: Optimal convex decompositions. In: Computational geometry. Mach. Intelligence Pattern Recogn., vol. 2, pp. 63–133. North-Holland, Amsterdam (1985)
10. Chen, C., Chang, R.: On the minimality of polygon triangulation. BIT 30(4), 570–582 (1990)
11. Lingas, A.: The power of non-rectilinear holes. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 369–383. Springer, Heidelberg (1982)
12. Asano, T., Asano, T., Pinter, R.Y.: Polygon triangulation: Efficiency and minimality. J. Algorithms 7(2), 221–231 (1986)
13. Keil, M., Snoeyink, J.: On the time bound for convex decomposition of simple polygons. Internat. J. Comput. Geom. Appl. 12(3), 181–192 (2002)
14. Chazelle, B.: Triangulating a simple polygon in linear time. Discrete Comput. Geom. 6(5), 485–524 (1991)
15. Arora, S.: Exploring complexity through reductions. In: Computational complexity theory. IAS/Park City Math. Ser., vol. 10, pp. 101–126. Amer. Math. Soc., Providence (2004)
16. Suri, S., O'Rourke, J.: Worst-case optimal algorithms for constructing visibility polygons with holes. In: Proc. 2nd Annu. ACM Sympos. Comput. Geom., pp. 14–23 (1986)
17. Chazelle, B., Edelsbrunner, H., Grigni, M., et al.: Ray shooting in polygons using geodesic triangulations. Algorithmica 12(1), 54–68 (1994)
18. Mucha, M., Sankowski, P.: Maximum matchings via gaussian elimination. In: FOCS, pp. 248–255. IEEE Computer Society, Los Alamitos (2004)

Lossless Fault-Tolerant Data Structures with Additive Overhead

Paul Christiano, Erik D. Demaine⋆, and Shaunak Kishore

MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St.,
Cambridge, MA 02139, USA
{paulfc,edemaine,skishore}@mit.edu

Abstract. We develop the first dynamic data structures that tolerate δ memory faults, lose no data, and incur only an $\tilde{O}(\delta)$ additive overhead in overall space and time per operation. We obtain such data structures for arrays, linked lists, binary search trees, interval trees, predecessor search, and suffix trees. Like previous data structures, δ must be known in advance, but we show how to restore pristine state in linear time, in parallel with queries, making δ just a bound on the rate of memory faults. Our data structures require $\Theta(\delta)$ words of safe memory during an operation, which may not be theoretically necessary but seems a practical assumption.

1 Introduction

As computer memory systems increase in size, complexity, and density, so does the chance that some bit flips during the lifetime of a computation or database. A survey of practical studies [14] concludes that 1,000–5,000 *soft errors* (not caused by permanent hardware failure) per billion hours per megabit is typical on modern memory devices. On a modern PC with 24 gigabytes of memory, this rate would imply one failure roughly every one to five hours. A recent study on production Ask.com servers [12] suggests that the observed rate of soft errors on ECC SDRAM is only 0.56 per billion hours per megabit, or roughly one failure per year.

While these failure rates are reasonably small, in a complex data structure, a single bit error can be catastrophic. For example, corrupting a single pointer can make most of a data structure unreachable, losing data and likely causing the system to crash. A natural problem, considered over the past 15 years, is to design data structures that do not crash after a small number of memory faults. Here we aim for the stronger property of losing none of the data in the structure, preventing any higher-level algorithm from crashing. More precisely, we develop data structures that tolerate a reasonable rate of memory failures while remaining correct and efficient.

Model. Our model of failure-prone computation is the *faulty RAM* of [11]. At any time, an adversary can corrupt the value of any word in the random-access memory to an arbitrary value, provided that the total number of corrupted words remains at most a known parameter δ. A *fault-tolerant* algorithm nonetheless reports the same result as if no words were changed; if all operations are fault tolerant, we call the data structure *fault tolerant* or *lossless*.

In addition to the main memory, we assume a *working store* of memory that cannot fail (e.g., CPU registers and level-1 cache) during the execution of a single operation. The working store is temporary, and cannot be used to store actual data between operations. This ephemerality means that the model supports multiple data structures with a single working store (CPU hardware). Naturally, our results also hold if there is no working store, but the adversary can modify words only between operations and not during (e.g., external storage media that might be damaged between uses).

While we state results in terms of an upper bound δ on the total number of faults, our data structures in fact support *live recovery*. In $O(\delta)$ time, we can examine a block of $O(\delta)$ consecutive memory locations, correct all errors within the block, restoring it to the state as if the errors never occurred. In this way, the recovery process can proceed in parallel with other operations with minimal locking, can be parallelized over multiple threads, and performs well when memory transfers occur in blocks. Thus δ really becomes an upper bound on the number of faults that can occur between passes of a background linear-time recovery process.

Our results. We develop the first fault-tolerant data structures that are simultaneously lossless and dynamic, as well as the first fault-tolerant data structures that are simultaneously lossless and efficient, requiring only an additive $\tilde{O}(\delta)$ overhead in overall space and per-operation time. Furthermore, we show that our techniques apply to a wide family of dynamic data structures: arrays, linked lists, binary search trees, interval trees, predecessor structures, and suffix trees. By contrast, previous work considered only dictionaries supporting membership, linked lists, and static problems; and either lost data from memory errors, or was static and required a lengthy recovery process.

Table 1 summarizes our results and how they compare to previous work. All of the problems we consider have a trivial $\Omega(\delta)$ lower bound on query time and overall space: with $o(\delta)$ time, everything examined can be corrupted, and with $o(\delta)$ space, some information must be lost. Hence, other than our interval trees which have an additional $\Theta(\log \delta)$ factor, our time and space bounds are optimal. The only suboptimal part is the requirement of $\Theta(\delta)$ words of working store. Improving this bound would require an advance in decoding error-correcting codes of size $\Theta(\delta)$ using failure-prone intermediate results.

Our data structures require $\delta^{O(1)}$ preprocessing time at the very beginning to construct the necessary error-correcting codes. The dynamic interval tree data structure (Section 5.1) uses randomization to solve stabbing queries, making them prone to reporting corrupted results. Assuming an oblivious adversary, each stabbing query is guaranteed to succeed with probability $1 - \varepsilon$ by spending a factor of

Table 1. Summary of new and previous on fault-tolerant data structures. All time bounds are amortized. Here δ denotes the upper bound on the number of faults, and ε denotes query failure probability (guaranteed to be 0 except where ε appears in bounds). Quantities (including δ) are measured in words (elements).

Problem	Loss/ error	Time/op. overhead	Space overhead	Working store	Model	Ref.
Dynamic linked list	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	2.7
Dynamic hash table	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	2.8
Dynamic predecessor	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	3.1
Dynamic binary search tree	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	3.1
Dynamic interval tree	0	$+O(\delta \lg \delta), \times \lg \frac{1}{\varepsilon}$	$+O(\delta \lg \delta)$	$O(\delta)$	RAM	5.1
Suffix tree construction	0	$+O(\delta)$	$+O(\delta)$	$O(\delta)$	RAM	4.1
Dynamic linked list	$O(\delta)$	$O(1)$	$O(1)$	$O(\delta)$	pointer	[2]
Dynamic binary search tree	$O(\delta)$	$O(1)$	$O(1)$	$O(\delta)$	pointer	[2]
Static sorting	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[9]
Static search	1	$+\delta$	$+O(\delta)$	$O(1)$	RAM	[9]
Dynamic search tree	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[8]
Dynamic hash table	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[8]
Dynamic priority queues	1	$+O(\delta)$	$+O(\delta)$	$O(1)$	RAM	[3]
Static dictionary, $\delta = O(n)$	0	$O(1)$	$2^{2^{O(\sqrt{\lg n})}}$	$O(1)$	cell probe	[6]

$O(1/\varepsilon)$ in time. We analyze according to an adaptive adversary (following [11]), which enables the adversary to corrupt any δ additional queries.

Related work. Finocchi and Italiano [11] introduced the faulty-memory RAM model of failure-prone computation that we use, which led to the bulk of the research in fault-tolerant data structures [9,8,3]. All of the results in this line, however, tolerate the loss of one data element per fault, replacing it with an arbitrary corrupted value. The idea behind this tolerance was to obtain better bounds—additive instead of multiplicative $\tilde{O}(\delta)$ overheads—by avoiding replication. (Hence the conference version of [11] had a title ending "...(without redundancy)".). Our data structures show, however, that it is possible to never lose data yet keep additive $\tilde{O}(\delta)$ overheads.

Before the work in the RAM model, Aumann and Bender [2] considered analogous problems in the pointer machine. Here it is necessary to assume that $O(\delta)$ nodes of the structure cannot be corrupted, in order not to lose the entire structure. Furthermore, because the model essentially prevents low-level manipulation of data, without replication pointer-machine data structures necessarily lose data from each error.

The only other work that considers lossless data structures is by de Wolf [6]. His results are all static, and rely on locally decodable codes, for which an optimal trade-off between local decodability and total space is an open problem. As far as we know, the best known upper bound on space for a static dictionary is superpolynomial in n, as indicated in Table 1.

2   Fundamental Techniques

Previous results in fault-tolerant data structures use one of two techniques to store data in faulty memory: data replication or error-correcting codes. On the one hand, a single word can be reliably stored by maintaining $\Theta(\delta)$ copies—the *resilient variables* of [10]. A resilient variable can be accessed quickly, but requires $\Theta(\delta)$ multiplicative space overhead. On the other hand, δ words can be reliably stored in an error-correcting code of size $\Theta(\delta)$. Although error-correcting codes reduce the space overhead, reading a single word from an error-correcting code requires $\Theta(\delta)$ time. Our main contribution is the introduction of data structures that combine these two techniques to achieve simultaneous space and time efficiency.

2.1   Blocked Fault-Tolerant Data Structures

Our data structures consist of "fault-tolerant blocks", which interpolate gracefully between the performance of resilient variables and error-correcting codes. To construct fault-tolerant blocks, we use the linear-time encodable and decodable error-correcting codes due to Spielman [13]. A concise statement of some of their results is the following:

Theorem 2.1. [13] *In $k^{O(1)}$ time, we can compute an $O(k)$-size description C_k of an error-correcting code which can be used as input to encoding and decoding algorithms E and D. Given a string x of k words, $E(x, C_k)$ encodes x into a string y of $O(k)$ words such that any string z differing from y on at most k words can be decoded: $D(z, C_k) = x$.*

We call k the *multiplicity* of the error-correcting code. For our purposes, this quantity is both the number of words stored by the code and the maximum number of word errors the code corrects. We will always set the multiplicity k to a power of 2 between 1 and δ. When creating any fault-tolerant data structure, we precompute C_k for each of these values of k. Because these descriptions have total size $O(\delta)$, we can store them all in safe memory. This precomputation takes $\delta^{O(1)}$ time, constituting the preprocessing needed by our data structures.

A *fault-tolerant block of multiplicity k* stores a k-word string x in $\Theta(\delta)$ space by storing $\lceil \frac{2\delta+1}{k} \rceil$ copies of the error-correcting code $E(x, C_k)$ contiguously in memory. Our data structures will call for fault-tolerant blocks of varying multiplicities. When a small amount of data must be accessed frequently, we will store it in fault-tolerant blocks of constant multiplicity; in this limit, fault-tolerant blocks reduce to resilient variables. When a large amount of data must be accessed infrequently, we will store it in fault-tolerant blocks of multiplicity $\Theta(\delta)$; in this limit, at least in spirit, a fault-tolerant block reduces to a single error-correcting code which corrects δ errors. In between these two extremes, we obtain useful trade-offs between space and time costs.

The data in a fault-tolerant block cannot be corrupted by fewer than $\delta + 1$ word errors. The value stored in an error-correcting code in a fault-tolerant block

of multiplicity k can only be changed if more than k of its words are corrupted. Thus the total multiplicity of all corrupted codes is at most δ, while the total multiplicity of all codes is at least $2\delta + 1$. It follows that the value stored in a fault-tolerant block can be correctly recovered in time $O(\delta)$ by decoding all of its error-correcting codes and taking the majority value. We can remove all errors from a fault-tolerant block by safely reading its value in this way and then writing a new fault-tolerant block which stores that value without errors.

A *blocked fault-tolerant data structure* is an array of fault-tolerant blocks; see Fig. 1. All of our fault-tolerant data structures are blocked. One benefit is that error detection and correction can be performed locally as described above. The problem of removing all errors from a blocked fault-tolerant data structure is embarrassingly parallel. (However, each parallel processor needs its own $O(\delta)$ working store of safe memory.)

Theorem 2.2 (Live recovery). *All errors can be removed from a blocked fault-tolerant data structure of size n by $p \leq \frac{n}{\delta}$ processors in $O(n/p)$ time.*

2.2   Fault-Resistant Operations

We can read a fault-tolerant block of multiplicity k in time $O(k)$ rather than $O(\delta)$ by reading only one of its $\lceil \frac{2\delta+1}{k} \rceil$ error-correcting codes. This speed-up comes at the expense of accuracy: by corrupting $k < \delta$ memory locations, an adversary may change the value returned by this operation. Our general approach is to compose fast but inaccurate operations with slow but reliable end-to-end verification. Whenever an operation fails, we implicitly discover the location of a memory fault, which we may avoid in the future, enabling us to amortize away the cost. In this section, we formalize this approach.

We adopt the notion of *prejudice numbers* from [10]. In their setting, each variable is copied $2\delta + 1$ times, and the prejudice number p represents the minimal "trusted index." Formally, whenever an operation needs to access a variable, it uses the pth copy. If an operation fails, then a fault must have occured in the pth copy of some variable, so the prejudice number is incremented and the operation repeated.

Fig. 1. Blocked data structures consist of a sequence of blocks each divided into $\Theta(\delta/k)$ codes of multiplicity k, for varying (power-of-two) values of k. If the prejudice number is 3, then an operation will read the shaded codes. The index 3 is faulty if any of those codes are faulty.

In our setting, the prejudice numbers play a slightly different role; refer to Fig. 1. Each prejudice number $p \in \{1, 2, \ldots, 2\delta + 1\}$ is associated with one error-correcting code from each fault-tolerant block, as follows: in a fault-tolerant block of multiplicity k, the prejudice number p is associated with the $\lceil \frac{p}{k} \rceil$th error-correcting code. Thus k prejudice numbers are assigned to any given error-correcting code of multiplicity k.
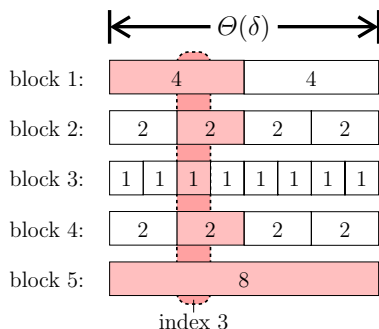
An error-correcting code is *faulty* if it no longer decodes to the correct value. A prejudice number is *faulty* if it is associated with some faulty code. Because multiple prejudice numbers may be assigned to the same error-correcting code, a single faulty code may cause multiple indices to become faulty. By simple counting, we can bound the number of faulty indices:

Lemma 2.3. *The number of faulty indices is at most the number of memory faults.*

An operation is fault-*resistant* if it succeeds when given a nonfaulty prejudice number. We can fault-resistantly read a resilient variable in constant time by reading the copy indexed by the prejudice number. Using a similar idea, we can quickly read a fault-tolerant block:

Lemma 2.4. *We can fault-resistantly read a fault-tolerant block of multiplicity k in $O(k)$ time.*

2.3   Fault-Tolerant Operations

An operation is fault-*tolerant* if it always succeeds. In particular, fault-tolerant operations do not depend on the choice of a prejudice number. In this section, we describe our basic approach to implementing fault-tolerant operations, which is to compose a fault-resistant query with fault-tolerant verification.

Lemma 2.5. *If a query Q has a fault-resistant implementation that runs in t_1 time and does not modify the data structure, and it is possible to fault-tolerantly verify the result of Q in t_2 time, then Q has a fault-tolerant implementation with amortized running time $O(t_1 + t_2)$, assuming at least δ invocations.*

Proof (sketch). We start with prejudice number $p = 1$. Whenever we perform an operation, we verify the result. If the operation failed, then we increment the prejudice number p and try again. Because the original operation is fault-resistant, we only need to repeat the operation if the prejudice number is faulty. By Lemma 2.3, this happens at most δ times. Some technical details arise when this transformation is applied in general and when Q is probabilistic, which we address in the full version of the paper.   □

2.4   Fault-Tolerant Memory

A *fault-tolerant memory* is a large array implemented by a sequence of fault-tolerant blocks of multiplicity δ. Fault-tolerant memory allows us to establish a relationship with data structures in the external-memory model.

Lemma 2.6. *Suppose that an operation can be completed in the external-memory model using T memory transfers and t computation steps, with block size $B = \delta$ and local memory $M = O(\delta)$. Then the same operation can be completed fault-tolerantly in the faulty RAM model in $O(\delta T + t)$ time.*

This correspondence does not generally produce efficient data structures because it does not make use of fault-tolerant blocks with lower multiplicities, but it is helpful in some cases.

Theorem 2.7. *There is a fault-tolerant linked list that stores n elements using $O(n + \delta)$ space. It supports advancing a pointer k steps, inserting k elements, and deleting k elements in $O(k + \delta)$ time.*

Theorem 2.8. *There is a fault-tolerant dictionary that stores n elements using $O(n + \delta)$ space and supports insertions, deletions, and lookups in $O(\delta)$ time per operation.*

3   Fault-Tolerant Predecessor

A *predecessor data structure* stores a set of keys x_i from some fixed ordered universe and supports insertions, deletions, and the predecessor query: given a key x, find the largest x_i that is at most x. Important examples of predecessor structures include balanced binary search trees, van Emde Boas priority queues, y-fast trees, and fusion trees.

We present a general technique for making any predecessor data structure fault-tolerant. Asymptotically, the transformation introduces only $O(\delta)$ additive space and time overhead. We require two technical conditions: (1) that predecessor queries do not modify the data structure, and (2) that the space use grows at least linearly with the number of keys. Splay trees do not satisfy the first condition, although AVL trees, red-black trees, and scapegoat trees do. Van Emde Boas queues whose size depends on the universe do not satisfy the second condition, although van Emde Boas queues with hashing do.

Our construction is similar in spirit to the resilient dictionaries of [10]. The main difference is that we use error-correcting codes within the leaves of our structure rather than accepting a small number of errors. Our results also hold for general predecessor structures rather than only for binary search trees. Our amortized startup cost is higher, however: our amortization holds only after $O(\delta)$ operations, rather than δ^{ε} operations.

Theorem 3.1. *Suppose that P is a predecessor data structure that stores n keys in space $s(n)$, supports queries in worst-case time $t_q(n)$ without modifying the data structure, and supports updates in amortized time $t_u(n)$. Then there is a fault-tolerant data structure P' that stores n keys in space $O\left(\delta s\left(\frac{n}{\delta}\right)\right)$, supports queries in $O(t_q(n)+\delta)$ amortized time, and supports updates in $O(t_q(n)+t_u(n)+\delta)$ amortized time. The data structure also incurs $\delta^{O(1)}$ worst-case preprocessing cost, plus an amortized startup cost of $O(\delta)$ operations. If the time bounds $t_q(n)$ or $t_u(n)$ are valid only in expectation, then the amortized time bounds for P' are also valid only in expectation.*

Proof (sketch). We divide the keys into $\frac{n}{\delta}$ contiguous groups of size $\Theta(\delta)$. We choose one representative from each group, and put these $\frac{n}{\delta}$ representatives into

an instantiation of P which we call the *summary structure*. This copy of P is stored in resilient variables (i.e., copied $2\delta + 1$ times), while the keys themselves are stored in fault-tolerant memory.

In order to find the predecessor of x, we perform a fault-*resistant* query in the summary structure to find the group representative which precedes x. We then fault-*tolerantly* access the corresponding group, and search for the predecessor of x there. By Lemma 2.5, we can combine these operations to perform a fault-tolerant predecessor query.

In order to insert or delete a key, we first perform a predecessor query to find the group that should contain that key and then modify that group. Every $\Omega(\delta)$ operations, a group may grow too large or too small. When this happens, we either split the group in two or merge it with an adjacent group, updating the summary structure appropriately. $\qquad\qquad\square$

4   Suffix Trees

A *suffix tree* stores a string S and supports the substring query: given a pattern P, report the positions of all occurrences of P in S. Each character of S may be an arbitrary machine word. Suffix trees occupy linear space, and can be constructed in linear time plus the time required to sort the stored string [7]. In this section, we describe a fault-tolerant suffix tree with the following guarantees:

Theorem 4.1. *There is a fault-tolerant suffix tree that stores a string S of length n in $O(n + \delta)$ space. The tree can be constructed in $O(n \log n + \delta)$ time, and supports substring queries in $O(m + k + \delta)$ amortized time, where m is the length of the pattern and k is the number of its occurrences. The data structure also incurs $\delta^{O(1)}$ worst-case preprocessing cost, plus an amortized startup cost of $O(\delta)$ operations.*

Our construction is based on a new decomposition of a tree into short paths and small subtrees, which we describe first. We then show how to to apply this decomposition to build a suffix tree that supports fault-resistant substring queries. Finally, we describe how to store some extra data which makes it possible to efficiently verify the results of substring queries.

4.1   Fault-Resistant Tries

A trie is a rooted tree in which each edge is assigned a constant-size label. Each vertex may also have a constant-size label. Tries support the retrieval operation: given a pattern P of length m, return the vertex obtained by starting at the root and walking down the tree, following the edges labeled by the corresponding elements of P. Let T be a trie of n nodes. We show how to build an implementation of T that supports fault-resistant retrieval operations.

First we partition T into micro
trees and micro paths as described by
the following lemma; refer to Fig. 2.

Lemma 4.2. *Any rooted tree on n
nodes can be partitioned into a col-
lection of rooted subtrees of at most
δ nodes, called* micro trees, *and $O(\frac{n}{\delta})$
paths of length at most δ, called* micro
paths.

Fig. 2. Our fault-tolerant trie first decom-
poses into micro trees, rooted subtrees of
size at most δ, and splits the remainder into
nonbranching paths of length at most δ.
Here $\delta = 3$.

Proof (sketch). We start with the
micro-tree/macro-tree decomposition
of [1]. Then we split non-branching
paths in the macro tree into subpaths
of length at most δ.   □

The central idea of our construction is to store a micro tree or micro path of
size k in a fault-tolerant block of multiplicity k. Every micro path traversed in a
retrieval operation is read in its entirety, except perhaps the last one. By storing
each path in its own fault-tolerant block, we are able to perform these accesses
in total time that is linear in $|P|$. Only one micro tree gets touched during a
traversal, which we can access in $O(\delta)$ time.

The use of fault-tolerant blocks of intermediate multiplicity is essential, be-
cause a micro path of size k must be stored in $O(\delta)$ space and must be traversable
in $O(k)$ time. This is precisely the guarantee afforded by our fault-tolerant blocks.

In order to perform the tree decomposition safely in the presence of memory
faults, we use the correspondence observed in Lemma 2.6 and standard tech-
niques for manipulating trees in external memory.

Lemma 4.3. *Given an Euler tour of a trie of size n in fault-tolerant memory,
we can fault-tolerantly construct a fault-resistant trie in $O(n + \delta)$ time.*

We can efficiently perform a fault-resistant retrieval query analogously to a re-
trieval query on a traditional trie. Essentially, each time the retrieval accesses
a micro path or micro tree, we read the entire path or tree and load the result
into safe memory.

Theorem 4.4. *There is a fault-resistant algorithm to retrieve a pattern P of
length m (provided as a stream) in a fault-resistant trie T in $O(m + \delta)$ time.*

4.2   Fault-Tolerant Suffix Trees

Suppose that S is a string of length n stored in external memory in blocks of
size B. There is an external-memory algorithm that computes the Euler tour
of the compressed suffix tree for S which runs in $O(n \log n)$ time and $O(\frac{n \log n}{B})$
memory transfers [4]. Given access to a string S as a stream, we can construct a
fault-tolerant suffix tree for S as follows. By Lemma 2.6, there is a fault-tolerant

algorithm that computes the Euler tour of the suffix tree for S in $O(n \log n + \delta)$ time and that stores the result in fault-tolerant memory. By Lemma 4.4, we can build a fault-resistant version of this trie. Our fault-tolerant suffix tree is composed of the string S and an Euler tour of the suffix tree, stored in fault-tolerant memory, and of the fault-resistant trie. This additional information allows us to perform fault-tolerant queries on the suffix tree.

In order to perform a fault-tolerant suffix tree operation, we first perform a fault-resistant query in the compressed suffix trie. With some care, we can verify the result of this query by fault-tolerantly examining the corresponding location of S. We then apply Lemma 2.5.

5   Interval Trees

An *interval* is a pair $[a_i, b_i]$ of elements from some fixed ordered universe with $a_i \leq b_i$. Interval (a_i, b_i) contains x if $a_i \leq x \leq b_i$. An *interval tree* maintains a set of intervals and supports insertions, deletions, and the stabbing query: given element x, report all intervals containing x. In this section, we describe a fault-tolerant interval tree with the following guarantees:

Theorem 5.1. *There is a fault-tolerant interval tree that stores n intervals in $O(n + \delta \log \delta)$ space, supports fault-tolerant updates in $O(\log n + \delta \log \delta)$ amortized time, and supports fault-tolerant stabbing queries in $O((\log n + k) \log \frac{1}{\varepsilon} + \delta \log \delta)$ amortized time, where k is the number of reported intervals and ε is an upper bound on the probability of failure. The data structure also incurs $\delta^{O(1)}$ worst-case preprocessing cost, plus an amortized startup cost of $O(\delta)$ operations. An adversary may adaptively corrupt the results of δ stabbing queries, and each other stabbing query fails with probability at most ε.*

We follow the construction of interval trees from [5]. The interval tree stores every interval's endpoints in the leaves of a binary tree. Each interval is stored twice at the least common ancestor of the leaves that store its endpoints: once in a list sorted in increasing order of left endpoint, and once in a list sorted in decreasing order of right endpoint. To perform a stabbing query at x, we search the binary search tree for x. Every interval containing x was stored in one of the nodes passed during this search. Whenever we move to the left child of a node, we report all intervals with left endpoint less than x; whenever we move to the right child of a node, we report all intervals with right endpoint at least x. If x is contained in k of the intervals stored at an internal node, we can find all k in $O(k)$ time by scanning the appropriate list until we find an interval not containing x.

The principal difficulty in making this structure fault-tolerant is that we do not know in advance how many intervals we will need to read from an internal node. In order to achieve a runtime $O(\log n + k) + \tilde{O}(\delta)$, we need to report each interval in amortized constant time. But in order to report m intervals from an internal node in $O(m)$ time using the techniques we have seen so far, we need to store them in a fault-tolerant block of multiplicity $O(m)$. This strategy would

incur significant space overhead, because m may be much smaller than δ. To overcome this difficulty, we store the intervals in a *prefix list*, composed of a sequence of fault-tolerant blocks of exponentially increasing size; see Fig. 3.

Theorem 5.2. *A fault-tolerant prefix list stores n elements from an ordered universe in $O(n + \delta \log \delta)$ space, supports fault-resistant access to the first k elements in $O(k)$ time for any k, and supports insertions and deletions in $O(\log n + \delta \log \delta)$ time.*

Proof (sketch). We store the first 2^i elements in a fault-tolerant block of multiplicity 2^i for $i = 0, 1, \ldots, \log \delta$. Because very little data is stored redundantly, the total space overhead is only $O(\delta \log \delta)$, but to read k elements, we can read a single fault-tolerant block of multiplicity at most $2k$, in $O(k)$ time. $\square$
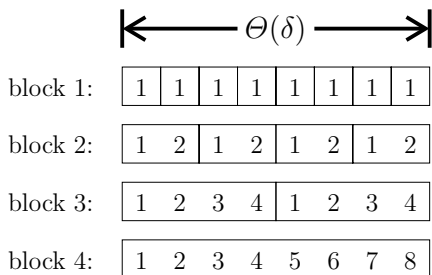
Fig. 3. The prefix list stores δ copies of the first element, $\delta/2$ copies of the first and second elements in a fault-tolerant block of multiplicity 2, $\delta/4$ copies of the first four elements in a fault-tolerant block of multiplicity 4, etc., and 1 copy of the first δ elements.

By a similar argument, a fault-tolerant prefix list supports finding all k elements less than a fixed value x in $O(k)$ time.

We can now implement a fault-resistant interval tree by carefully transforming the classical construction, using a prefix list to store the intervals at each interior node. In contrast with our previous data structures, we have no way to verify the results of an interval stabbing query. Instead, we guarantee that our results are correct with high probability by reading several error-correcting codes at random from each fault-tolerant block we access.

References

1. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proc. 39th Annual Symposium on Foundations of Computer Science, pp. 534–543 (1998)
2. Aumann, Y., Bender, M.A.: Fault tolerant data structures. In: Proc. 37th Annual Symposium on Foundations of Computer Science, pp. 580–589 (1996)
3. Brodal, G.S., Fagerberg, R., Finocchi, I., Grandoni, F., Italiano, G.F., Jørgensen, A.G., Moruz, G., Mølhave, T.: Optimal resilient dynamic dictionaries. In: Arge, L., Hoffmann, M., Welzl, E. (eds.) ESA 2007. LNCS, vol. 4698, pp. 347–358. Springer, Heidelberg (2007)
4. Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 139–149 (1995)
5. Chiang, Y.-J., Tamassia, R.: Dynamic algorithms in computational geometry. Proc. IEEE 80(9), 1412–1434 (1992)

 6. de Wolf, R.: Error-correcting data structure. In: Proc. 26th International Sympo-
    sium on Theoretical Aspects of Computer Science, pp. 313–324 (February 2009)
 7. Farach-Colton, M., Ferragina, P., Muthukrishnan, S.: On the sorting-complexity of
    suffix tree construction. J. ACM 47, 987–1011 (2000)
 8. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient search trees. In: Proc. 18th
    Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 547–553 (2007)
 9. Finocchi, I., Grandoni, F., Italiano, G.F.: Optimal resilient sorting and searching in
    the presence of memory faults. Theoretical Computer Science 410(44), 4457–4470
    (2009)
10. Finocchi, I., Grandoni, F., Italiano, G.F.: Resilient dictionaries. ACM Transactions
    on Algorithms 6(1) (2009)
11. Finocchi, I., Italiano, G.F.: Sorting and searching in faulty memories. Algorith-
    mica 52(3), 309–332 (2008)
12. Li, X., Shen, K., Huang, M.C., Chu, L.: A memory soft error measurement on
    production systems. In: Proc. 2007 USENIX Annual Technical Conference, pp.
    21:1–21:6 (2007)
13. Spielman, D.A.: Linear-time encodable and decodable error-correcting codes. IEEE
    Transactions on Information Theory 42(6), 1723–1732 (1996)
14. Tezzaron Semiconductor. Soft errors in electronic memory. White paper (January
    2004), http://www.tezzaron.com/about/papers/soft_errors_1_1_secure.pdf

Binary Identification Problems for Weighted Trees

Ferdinando Cicalese[1], Tobias Jacobs[2], Eduardo Laber[3], and Caio Valentim[3]

[1] University of Salerno, Italy
[2] National Institute of Informatics, Japan
[3] PUC – Rio de Janeiro, Brazil

Abstract. The Binary Identification Problem for weighted trees asks for the minimum cost strategy (decision tree) for identifying a node in an edge weighted tree via testing edges. Each edge has assigned a different cost, to be paid for testing it. Testing an edge e reveals in which component of $T - e$ lies the vertex to be identified. We give a complete characterization of the computational complexity of this problem with respect to both tree diameter and degree. In particular, we show that it is strongly NP-hard to compute a minimum cost decision tree for weighted trees of diameter at least 6, and for trees having degree three or more. For trees of diameter five or less, we give a polynomial time algorithm. Moreover, for the degree 2 case, we significantly improve the straightforward $O(n^3)$ dynamic programming approach, and provide an $O(n^2)$ time algorithm. Finally, this work contains the first approximate decision tree construction algorithm that breaks the barrier of factor $\log n$.

1 Introduction

We study the Binary Identification Problem (BIP) [8] when the underlying space of objects and tests can be represented by a weighted tree. By a weighted tree we understand a pair $(T, \mathbf{c})$ where T is a tree and $\mathbf{c}$ is a cost assignment to the edges $E(T)$ of T, i.e., $\mathbf{c} : e \in E(T) \mapsto c(e) \in \mathbb{R}_0^+$.

The Binary Identification Problem for weighted trees. A decision tree for a weighted tree $(T, \mathbf{c})$ is a binary tree recursively defined as follows: if the tree T has only one vertex, then the decision tree is a single leaf labeled with the only vertex in T. If T has at least one edge, a decision tree for T has its root r labeled with one edge $e = \{u, v\}$ in T, and the subtrees rooted at the children of r are decision trees for the connected components T_u and T_v of $T - e$.

For the sake of distinguishing between the input tree and the decision tree, we shall reserve the term *node* to the decision tree and the term *vertex* to the input tree.

A decision D for $(T, \mathbf{c})$ naturally defines a strategy for identifying an initially unknown vertex x from T via edge queries. If node w of D is labeled with the edge $e = \{u, v\}$ of T, we map w to the question "Is x in T_u or in T_v?", where T_u (resp. T_v) denotes the component of $T - e$ which contains u (resp. v). The search strategy now consists in starting with the query at the root of D and then recursively continuing with the subtree being a decision tree for the component indicated in the answer. Accordingly, each leaf ℓ of D is then labeled with the vertex of T uniquely identified by the sequence of questions and answers corresponding to the path from the root of D to ℓ.

The cost of a decision tree D for T is 0 if D consists of just one leaf (i.e., T has only one vertex), and otherwise it is the cost of the edge in the root of D plus the maximum of the costs of the decision trees rooted at the children of the root of D, in formulae

$$cost(D) = c(root(D)) + \max\{cost(D_L), cost(D_R)\},$$

where D_L and D_R are the decision trees rooted at the left and right child of the root of D, respectively.

We also define the cost of searching a single vertex $u \in T$ according to D as the sum of the costs of the edges labeling the nodes in the path from the root of D to the leaf labeled with u. With this definition, we have that the cost of D is equal to the maximum among the search costs of the vertices from T according to D.

Given a weighted tree $(T, \mathbf{c})$ the Binary Identification Problem asks for the decision tree for T of minimum cost.

Our results. We provide a complete characterization of the complexity of the Binary Identification Problem for weighted trees. We show strong NP-hardness of both the class of instances with diameter 6 and the class of degree 3 instances. Both thresholds are tight. In fact, we show a polynomial time algorithm for instances of bounded diameter at most 5. We reserve special attention to the case of instances of maximum degree 2 (simple paths). It is easy to see that for such instances, a natural dynamic programming approach results in an $O(n^3)$ algorithm for building an optimal decision tree, and, to the best of our knowledge, no algorithm with better asymptotic was known prior to this paper. We present a non-trivial DP based algorithm which provides the optimal decision tree in $O(n^2)$ time. Such a speed up has been obtained in analogous problems by employing the Knuth-Yao technique [11,19]. However, this technique cannot be directly applied to the problem considered here as we discuss in Section 4.

Finally, for general trees, we provide an $o(\log n)$-approximation algorithm. Although this result is not a significant improvement, in numerical terms, over the existing $O(\log n)$ approximation [6], it is interesting as it shows a sharp separation in the complexity picture of the binary identification problem with costs. This is because the general BIP (not restricted to tree instances), even with uniform weights, does not admit an $o(\log n)$-approximation unless $P = NP$ [13].

Related work. The binary identification problem (BIP) for unweighted trees has been extensively studied in the contexts of searching and edge ranking [9,5,14,1,17,18]. The edge ranking problem and its connection to the problem studied here is precisely explained later when we discuss some applications. Linear time algorithms that construct an optimal decision tree for unweighted trees are presented in [14,17].

The BIP for weighted trees was first studied by [6] in the context of edge ranking. In this initial paper, the problem was defined and proved to be NP-complete already for the class of instances of diameter at most 10. In addition, an $O(\log n)$ approximation algorithm was also provided. In fact, $O(\log n)$ approximation can be attained for a more general version of the problem (not restricted to tree instances), via a simple greedy procedure [3].

When the weighted tree is a path, the BIP is equivalent to the problem of searching in an ordered array with costs depending on the position probed. A natural DP approach

solves this problem in $O(n^3)$ time. A linear time algorithm with constant approximation factor is presented in [12]. In [2], Charikar *et al.* consider this problem from a competitive analysis perspective.

Applications. The BIP is a basic problem in computer science and has applications in many different scenarios.

The BIP for (weighted) trees arises when one has to identify the faulty component of a system. As an example, a system is represented by a network (in our case a tree) and its faulty component (vertex) has to be found. Different points of the network might require more or less expensive operations for the inspection. Inspecting one spot (edge) in the network reveals only directional information about the location of the failure w.r.t. the inspected point. One such problem is described, e.g., in [15] as searching for holes in an oil pipeline. In [18], the problem of finding a bug in a software application is mentioned.

As already mentioned, the BIP for trees is equivalent to the edge ranking problem for trees. An edge ranking of T is an assignment to each edge e of T of an integer $r(e)$ (the rank of e) s.t. for any two edges $e_1, e_2 \in T$ if $r(e_1) = r(e_2)$, then the path connecting e_1 and e_2 contains an edge e with $r(e) > r(e_1)$. The cost of an edge ranking of a weighted tree $(T, \mathbf{c})$, denoted by rankcost$(T, \mathbf{c})$ is defined as follows: If T has only one node, then rankcost$(T, \mathbf{c}) = 0$. Otherwise, rankcost$(T, \mathbf{c}) = c(e^*) + \max\{\text{rankcost}(T_u, \mathbf{c}), \text{rankcost}(T_v, \mathbf{c})\}$, where $e^* = uv$ is the edge with maximum rank in T and T_u (resp. T_v) is the connected component of $T - e$ that contains u (resp. v). Given a weighted tree $(T, \mathbf{c})$ the edge ranking problem asks for the minimum cost ranking. The equivalence to the decision tree problem is easily seen (see also [7]).

The edge ranking problem arises in the context of parallel query processing in large database systems [16]. Also, it can be used to optimize multi-part product assembly [10,6]. Assume that each edge represents the operation of assembling two parts of a product and the weight of an edge represents the time necessary to complete the corresponding assembly operation. In addition, edges that share an endpoint correspond to dependent operations so that they cannot be processed simultaneously. An edge ranking provides a scheduling of the assembly operations with the guarantee that only independent operations are scheduled simultaneously. Moreover, the cost of the edge ranking is the total time necessary for completely assembling the product.

2   Proofs of Strong NP-Hardness

Our proofs of NP-hardness proceed in two steps. In the first step, we reduce from a certain scheduling problem, which we call *Flexible Machine Scheduling (FMS)*. This problem can be reduced both to the BIP on weighted trees of diameter 6 and to the problem on degree 3 trees. Both reductions have the property that the edge costs of the resulting weighted tree instance are polynomial in the processing times and deadlines of the scheduling instance we reduce from. In the second step, we reduce Problem 3SAT to FMS and thereby show strong NP-hardness of that scheduling problem.

The Flexible Machine Scheduling problem is defined as follows: We are given k pairwise disjoint sets of jobs $S_1, \ldots, S_k$. Each job J from one of those sets is characterized by its length $l(J) \in \mathbb{R}_0^+$ and deadline $d(J) \in \mathbb{R}_0^+$. Furthermore, each job set S_i has a so-called

separation time $s_i \in \mathbb{R}_0^+$. Initially, the jobs have to be scheduled on a single machine M. However, at any point of time t we can open an extra machine M_i, $1 \le i \le k$, where the remaining jobs from set S_i will be processed afterwards. Opening machine M_i takes time s_i, which means that, during the time interval $(t, t + s_i)$, neither M nor M_i can process a job. We are interested in the problem to decide whether for a given FMS instance there exists a feasible solution, i.e. one where each job is completely processed before its deadline is reached.

In the following we show how to reduce FMS to the bounded diameter case of BIP on weighted trees. After that, we will explain how to modify the reduction in order to obtain bounded degree instances.

It will be comfortable to talk of a decision tree in terms of the search strategy it defines. Recall that, in this perspective, we interpret the edge labels of the nodes of the decision tree as queries. Also, we talk about the search cost of a vertex v in the input tree, as the sum of the costs of the edges queried in the decision tree on the path from the root to the leaf labeled with v. We will call such path the search path to/of v. We also say that this path isolates v.

Let I be an instance of FMS, determined by the job sets $S_1, \ldots, S_k$ and the corresponding separation times $s_1, \ldots, s_k$. Let $S = S_1 \cup \ldots \cup S_k$. For each job $J \in S$, the tree T contains two edges $e(J), e'(J)$. The cost of $e(J)$ is the length $l(J)$ of J. The cost of $e'(J)$ is $A - d(J)$, where $A \in \mathbb{R}$ is a large constant, depending on instance I, whose exact value will be determined later. The edges $e(J)$ and $e'(J)$ have a common endpoint denoted $v(J)$. The other endpoint of $e'(J)$ is a leaf.

Before continuing with the description of $(T, \mathbf{c})$, we give some first intuition about the idea of the reduction. I will be reduced to the problem of deciding whether there exists a search strategy for $(T, \mathbf{c})$ where the search cost of any vertex is no more than A. Observe that, in order to isolate the vertex $v(J)$, the edge $e'(J)$ has to be queried. This means that the total cost of all other queries on the search path of vertex $v(J)$ must not exceed $d(J)$.

For each job set S_i, $i = 1, \ldots, k$, the tree T contains a vertex u_i, which serves as the common endpoint of all edges $e(J)$, so that $e(J)$ connects u_i with $v(J)$ for each $J \in S_i$. In addition u_i has one further incident edge f_i, whose cost is set to the separation time s_i. The construction of T is completed by letting $f_1, \ldots, f_k$ share the common endpoint u. Clearly, T has diameter 6, see Figure 1(a) for an example.

Intuitively, a query to edge f_i will correspond to opening machine M_i in the FMS instance. This causes additional search cost s_i to all vertices $v(J)$ that have not been separated from the central vertex u by a query to $e(J)$ or to f_j with $V \in S_j$, $j \ne i$.

Lemma 1. *There is a feasible solution to I if and only if there is a decision tree for $(T, \mathbf{c})$ of cost not larger than A.*

Proof. "$\Rightarrow$" Assume that there is a feasible schedule for I. If some machines are not opened during the execution of the schedule, we can equivalently assume that these machines are opened after all jobs have been processed. We can also assume that any idle time between two consecutive jobs on M is completely used for opening new machines, and there is no idle time between jobs on any other machine. A feasible schedule with that property can easily be constructed from an arbitrary feasible schedule.

A search strategy for $(T, \mathbf{c})$ is constructed from the schedule by interpreting the assignments to machine M as the search path to vertex u. Assigning job J to M corresponds to a query to edge $e(J)$, and opening machine M_i corresponds to a query to edge f_i. Hence, the cost of the search path to u equals the point of time when the last job has been finished on M and the last machine has been opened. For $i = 1, \ldots, k$, node u_i shares its search path with u until edge f_i is queried. After that, its search path is represented by the order of the jobs on machine M_i, where job $J \in S_i$ corresponds to edge $e(J)$. The cost of the search path to u_i therefore corresponds to the point of time when the last job on M_i has been completed and the machine has been opened. For each $J \in S$, the edge $e'(J)$ is queried right after $e(J)$. Therefore, the search cost of $v(J)$ is by the cost of $e'(J)$ larger than the time when job J has been processed. As the schedule is feasible, the search cost is at most $d(J) + c(e'(J)) = A$. The search path to the leaf incident to $e'(J)$ has the same cost as $v(J)$. For A large enough, the search cost of u as well as the search costs of $u_1, \ldots, u_k$ are not larger than the search costs to the $v(J)$s, and thus we have a search strategy where the search cost of each vertex is no more than A.

"$\Leftarrow$" Assume that there is a search strategy for $(T, \mathbf{c})$ where each vertex has search cost A or less. For any $J \in S$, the search cost of the leaf adjacent to $e'(J)$ is not larger than the search cost of $v(J)$. This is because the search path to both vertices is the same until $e'(J)$ is queried, and after that query the leaf is already isolated.

We can assume that $e(J)$ is queried before $e'(J)$ for any $J \in S$, because otherwise we can interchange the queries to $e(J)$ and $e'(J)$, which makes no search path except the one to the leaf adjacent to $e'(J)$ more expensive, and the latter search path has cost which is not larger than the search cost of $v(J)$.

Under this assumption, a schedule for I can be directly constructed from the search strategy. The schedule with respect to machine M processes jobs J in exactly the order of the search path to u, where a query to $e(J)$ corresponds to processing of job J, and a query to an edge f_i is translated into opening machine M_i. After M_i has been opened, it processes jobs in the order the edges $e(J)$ with $J \in S_i$ that are queried after f_i. This way, we achieve that the completion time of job J is at most $c(e'(J)) = A - d(J)$ less than the search cost of $e(J)$. As that search cost is no more than A by assumption, J is completed by time $d(J)$. $\square$

To ensure that the search costs of the $v(J)$s are not smaller than the search costs of the other internal vertices, we set A to $\sum_{J \in S} c(e(J)) + \sum_{i=1}^{k} c(f_i) + \max\{d(J) \mid J \in S\} + 1$. We achieve that $e'(J)$ has a cost larger than the cost of the search paths to u and all u_i.

Reduction of FMS to bounded degree instances. The tree T has diameter 6, but its degree is unbounded. For constructing a bounded degree tree instead, we need to replace the star structure of T with a binary tree structure. Construct a binary tree rooted at u, having k leaves $u_1, \ldots, u_k$. The edges $f_1, \ldots f_k$ are the edges of that binary tree that end in $u_1, \ldots, u_k$, respectively. Now enhance the tree constructed so far by making u_i the root of a binary tree having $|S_i|$ leaves $v(J), J \in S_i$, for $i = 1, \ldots, k$. For $J \in S_i$, the edge incident to $v(J)$ is $e(J)$. Finally, for $J \in S$ add a further outgoing edge to $v(J)$, namely, the edge $e'(J)$. The other end points of the edges $e'(J), J \in S$, are the final leaves of the constructed tree T'. The edges $e(J), e'(J)$ and f_i have the same costs as before, except that we need A to have a different value.

Let E' be the set of all edges that do not occur in the diameter 6 realization of T. We need to ensure that no edge from E' appears on the search path to some $v(J)$. This is achieved by making them expensive: we assign cost $c' = \max\{d(J), J \in S\} + 1$ to them, which implies that no search strategy querying an edge from E' during the search for a $v(J)$ can reach the cost bound A. As we still need the search costs of the $e(J)$s to be dominating the other nodes, we set $A = |T'|c'$.

Strong Hardness of FMS. We show it by a reduction from 3SAT.

Definition 1 (3SAT). *Given a set of m clauses $C_1, \ldots, C_m$ over a set of n boolean variables $x_1, \ldots, x_n$, where each clause depends on exactly three variables, decide whether there is an assignment to the variables such that each clause is satisfied.*

Let $C_1, \ldots, C_m$ be an instance of 3SAT with variables $x_1, \ldots, x_n$. We show how to construct an equivalent instance I of FMS.

Instance I consists of $2n$ sets of jobs $S_1, \ldots, S_n, \bar{S}_1, \ldots, \bar{S}_n$ which correspond to the literals $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$, respectively. The separation time of all machines associated to the job sets is 2. For $i = 1, \ldots, n$, S_i contains a job J_i with processing time 1 and deadline $5(i-1) + 3$. Furthermore, S_i contains job K_i with processing time 2 and deadline $5i$. The set $\bar{S}_i$ contains a pair of jobs $\bar{J}_i, \bar{K}_i$ with the same characteristics as J_i, K_i. Those $4n$ jobs will enforce that at time $5n$ any feasible schedule has opened either the machine associated with S_i or with $\bar{S}_i$ for $i = 1, \ldots, n$. Note that the construction so far is independent of the clauses.

Now, for $j = 1, \ldots, m$, add jobs $L_{j1}, \ldots, L_{jn}, \bar{L}_{j1}, \ldots, \bar{L}_{jn}$ to $S_1, \ldots, S_n, \bar{S}_1, \ldots, \bar{S}_n$, respectively, all having processing time 1. The jobs added to the three sets corresponding to the literals in C_j have deadline $5n + (j-1)n + 2$, while the deadline of all $2n - 3$ other jobs is $5n + jn$. The idea of this construction is that we need at least one machine corresponding to a literal in C_j to be open, because we cannot process all three jobs with deadline $5n + (j-1)n + 2$ on machine M. The proof of the following result is omitted.

Lemma 2. *The 3SAT instance has a solution iff a feasible schedule exists for I.*

Theorem 1. *The BIP on weighted trees is strongly NP-hard on instances of diameter 6. The same complexity holds for degree 3 instances.*

Proof. By Lemma 1, a pseudo-polynomial algorithm for diameter 6 or degree 3 instances of BIP, implies a pseudo-polynomial algorithm for FMS. Therefore, the statement immediately follows by the NP-hardness of FMS (Lemma 2). □

3   A Polynomial Time Algorithm for Diameter 5 Instances

In this section we show that with respect to the diameter the threshold of 6 in the hardness result of the previous section is tight. We provide a polynomial time solution for instances of diameter not larger than 5. The following lemmas allow us to focus on some particular optimal strategies. In addition, Lemma 4 below allows us to assume that in the tree under consideration each node has at most one leaf neighbor. Due to the space constraints, the proofs are deferred to the extended version of the paper.

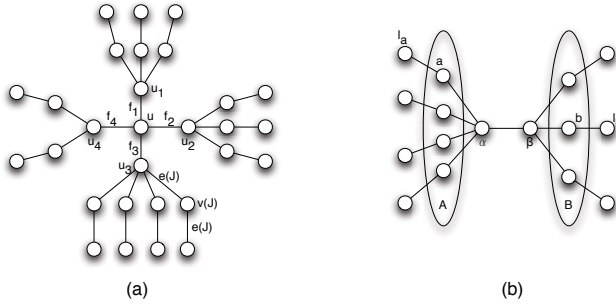For any node u in T, let $L(u)$ be the set of leaves adjacent to u.

Fig. 1. (a) A tree obtained from the reduction of an instance of FMS. Here J is a job in S_3. (b) An instance of a diameter 5 tree.

Lemma 3. *For any tree T there exists an optimal search strategy where no leaf is separated from its neighbor u before u has been separated from all its non-leaf neighbors.*

Lemma 4. *For some internal node u of T, let $(\tilde{T}, \tilde{c})$ be obtained from $(T, \mathbf{c})$ by replacing nodes $L(u)$ and edges $\{(u,v) \mid v \in L(u)\}$ with a single leaf l_u and edge $\{u, l_u\}$ with cost $\tilde{c}(\{u, l_u\}) = \sum_{v \in L(u)} c(\{u, v\})$, while for all other edges, $\tilde{\mathbf{c}}$ and $\mathbf{c}$ coincide. Then T and $\tilde{T}$ are equivalent problem instances, i.e. an optimal solution for $(T, \mathbf{c})$ can be transformed into one for $(\tilde{T}, \tilde{c})$ and vice versa.*

Under the assumption of this last lemma, any diameter 5 tree can be described as follows. There is a "central" edge $\{\alpha, \beta\}$, and α and β are connected to the set of nodes $\beta \cup A$ and $\alpha \cup B$, respectively. Each $a \in A$ is connected to a leaf l_a, and each $b \in B$ is connected to a leaf l_b. Although there exist diameter 5 trees where some a does not have a neighbor besides α, we can assume in that case that the edge $\{a, l_a\}$ has cost zero. See Figure 1(b) for an example tree with $|A| = 4$ and $|B| = 3$.

The edges $\{a, l_a\}, a \in A$ and $\{b, l_b\}, b \in B$ are called *outer edges*, and the edges incident to α and β, except edge $\{\alpha, \beta\}$, are called *inner edges*. From Lemma 3 we know that any outer edge $\{a, l_a\}$ or $\{b, l_b\}$ can be assumed to be queried after the query to the respective inner edge $\{\alpha, a\}$ or $\{\beta, b\}$. We therefore only need to reason about the optimal strategy for querying the edges incident to α and β. Furthermore, we can ignore the search costs to the leaves, because of Lemma 3. The following lemma shows that we can restrict ourselves to certain ordered strategies.

Lemma 5. *There is an optimal solution that is* ordered, *i.e.*
(a) the inner edges $(\alpha, a), a \in A$, are queried in the order of the nonincreasing cost of their respective outer edges $\{a, l_a\}$. The same holds for the edges $\{\beta, b\}, b \in B$.
(b) the inner edges queried before $\{\alpha, \beta\}$ are queried in order of the nonincreasing cost of their respective outer edges.

From Lemma 5 one can straightforwardly derive an optimal algorithm. First, sort the inner edges by the cost of their adjacent outer edges. For $j = 0, \ldots, |A| + |B|$, evaluate the search strategy which queries the first j inner edges, then performs a query to $\{\alpha, \beta\}$, then queries the remaining elements of A and B in two different branches of the search

tree, but according to the same order. Finally, choose the best among the $|A| + |B| + 1$ evaluated solutions.

We have constructed an algorithm for trees having diameter exactly 5. However, trees with diameter less than 5 can be reduced to diameter 5 trees by adding nodes that are connected to the original nodes via cost 0 edges. This leads to the following result.

Theorem 2. *The problem to compute an optimal search strategy for trees of diameter at most 5 admits a polynomial time algorithm.*

4   A Quadratic Time Algorithm for Path Instances

In this section we consider the particular case when the tree T is a simple path $P = e_1, \ldots, e_n$, with n edges. A natural dynamic programming procedure finds the optimal decision tree for the path in $O(n^3)$ time. It is based on the observation that the cost of the optimal decision tree $OPT[i, j]$ for the subpath $P[i, j] = e_i, \ldots, e_j$ can be determined as

$$OPT[i, j] = \min_{k=i \ldots j} \left(c(e_k) + \max\{OPT[i, k-1], OPT[k+1, j]\} \right) \tag{1}$$

for $j > i$, and otherwise $OPT[i, i] = c(e_i)$ and $OPT[i, i-1] = 0$. There are $O(n^2)$ subproblems, and for each subproblem one has to compare $O(n)$ different possibilities for index k, which is why this algorithm has cubic runtime.

We shall now present a dynamic programming algorithm which cuts a factor of n from the natural DP. The Knuth-Yao Quadrangle Inequality [11,19] is a well known technique to speed up dynamic programs of the same flavor—unfortunately, it does not hold here. The Quadrangle Inequality would imply $OPT[i, j] + OPT[i', j'] \leq OPT[i, j'] + OPT[i', j]$, for each $i \leq i', \leq j \leq j'$. However, for the problem instance $e_1, \ldots, e_7$ with $c(e_1) = 1999, c(e_2) = 2, c(e_3) = 3, c(e_4) = c(e_5) = c(e_6) = 1000$, and $c(e_7) = 3$, one can verify that $OPT[1, 6] + OPT[6, 7] > OPT[1, 7] + OPT[6, 6]$.

Assume that we want to compute $OPT[i, j]$ using Equation 1. We need the previously computed values of $OPT[i, i], OPT[i, i+1], \ldots, OPT[i, j-1]$, and we need the values $OPT[i+1, j], OPT[i+2, j] \ldots, OPT[j, j]$. Straightforward argumentation (or the use of Lemma 7, in the next section) shows that the first sequence is nondecreasing and the second one is nonincreasing.

Therefore, for $j > i$, let b_{ij} be the smallest integer s in $[i+1, j]$ such that $OPT[i, s-1] \geq OPT[s+1, j]$. Note that b_{ij} is well defined because $OPT[i, j-1] \geq OPT[j+1, j] = 0$. In addition, the monotonicity property, mentioned in the above paragraph, implies that $OPT[i, k-1] < OPT[k+1, j]$ for each $k \in [i+1, b_{ij}-1]$ and $OPT[i, k-1] \geq OPT[k+1, j]$ for each $k \in [b_{ij}, j]$. We call b_{ij} the *transition index* of the set $\{i, \ldots, j\}$.

Let $LC(i, j) := c(e_j) + OPT[i, j-1]$ the *left cost* of j with respect to i. Analogously, we let $RC(i, j) := c(e_i) + OPT[i+1, j]$ and we call it the *right cost* of i w.r.t. j. Exploiting the transition index, Equation 1 can be rewritten as

$$OPT[i, j] = \min \left\{ \min_{k=i, \ldots, b_{ij}-1} RC(k, j), \min_{k=b_{ij}, \ldots, j} LC(i, k) \right\} .$$

Motivated by this formula, we let

$$\ell_{ij} = \operatorname{argmin}_{k \in [b_{ij}, j]} \{LC(i, k)\} \quad \text{and} \quad r_{ij} = \operatorname{argmin}_{k \in [i, b_{ij}-1]} \{RC(k, j)\},$$

so that $OPT[i,j] = \min\{LC(i,\ell_{ij}), RC(r_{ij},j)\}$. Thus, we can find the cost of a minimum cost decision tree for $P[1,n]$ through the following simple procedure.

```
Algorithm PATHOPT(P, c, n)
For i = 1,...n do
    OPT(i, i − 1) = 0 ; OPT(i, i) ← c(e_i)
For len = 2,...n do
    For i = 1...(n − len + 1) do
        j ← (i + len − 1)
        Find b_{ij}
        Find ℓ_{ij} and r_{ij}
        OPT(i, j) = min{LC(i, ℓ_{ij}), RC(r_{ij}, j)}.
    End do
End do
```

We will now show that we can find b_{ij}, ℓ_{ij} and r_{ij} in $O(1)$ amortized time. We start with b_{ij}. The following monotonicity property turns out to be useful

Lemma 6. *Let i,j be such that $1 \le i < j \le n$ and $j \ge i+2$. Then, $b_{i,j-1} \le b_{ij} \le b_{i+1,j}$.*

Proof. We only show $b_{i,j-1} \le b_{ij}$, because the other inequality can be shown symmetrically. By the definition of $b_{i,j-1}$, we have $OPT[i,k-1] < OPT[k+1,j-1]$ for each $k \in [i+1, b_{i,j-1}-1]$. By Lemma 7 it holds that $OPT[k+1,j-1] \le OPT[k+1,j]$ for any k. It follows that $OPT[i,k-1] < OPT[k+1,j]$ for $k \in [i+1, b_{i,j-1}-1]$, which is why b_{ij} cannot be smaller than $b_{i,j-1}$. $\qquad\square$

The above lemma implies that for computing b_{ij} it suffices to consider the positions between $b_{i,j-1}$ and $b_{i+1,j}$. Among these positions, b_{ij} is computed as the smallest index k such that $LC(i,k) \ge RC(k,j)$, where all values of LC and RC required for this computation can be determined reusing previously computed values of $OPT[\cdot,\cdot]$. Fix a value of len (algorithm's outer loop); the total number of positions that are considered for determining the indices b_{ij}, with $j - i = len - 1$, is

$$\sum_{i=1}^{n-len+1}(b_{i+1,i+len-1} - b_{i,i+len-2} + 1) = (n - len + 1) + b_{n-len+2,n} - b_{1,len-1} < 2n.$$

Thus, the algorithm spends $O(n^2)$ to find all b_{ij}'s.

It remains to show how to find the indices ℓ_{ij} and r_{ij} in $O(1)$ amortized time. We just detail how to compute ℓ_{ij} because r_{ij} can be computed in a symmetric way.

Assume some i to be fixed throughout the following paragraphs. In order to find the ℓ_{ij}'s in an efficient way, we organize candidates for $\ell_{i,j}$ in an additional data structure L_i. Recall that the candidates for $\ell_{i,j}$ are the indices $b_{ij}, \ldots, j$. If $k > k'$ and $LC(i,k') \ge LC(i,k)$ we say that k' is *left dominated* by k with respect to i. Note that if k' is left dominated by some k then we can conclude that $\ell_{ij} \neq k'$ for $j = k, \ldots, n$ since k is a choice better than k' (whenever k' is a candidate). This is a key property for finding the ℓ_{ij}'s efficiently because it allows to discard some candidates from L_i during the algorithm's execution.

The algorithm implements L_i as a queue and maintains the following invariants:

(a) right after processing the path $P[i, j-1]$ in the above pseudo code, L_i stores the indices from $[b_{i,j-1}, j-1]$ that are not left dominated, with respect to i, by any other index in that set.

(b) The indices in L_i are sorted by increasing order of their left costs w.r.t. i.

These conditions together imply that, right after processing $P[i, j-1]$, L_i is also sorted by increasing order of edge indices.

When $j = i+1$, we set $\ell_{i,i+1} = i+1$ and then we put this index into the initially empty queue L_i. Since $b_{i,i+1} = i+1$ we have that both (a) and (b) are satisfied right after processing $P[i, i+1]$. To find ℓ_{ij} in later iterations and maintain the invariants we proceed as follows: (Step 1) Remove from L_i every index smaller than b_{ij}. This is done because, as b_{ij} is nondecreasing in j, these indices can never be candidate indices any more; (Step 2) Remove from L_i all indices k such that $LC(i, j) \leq LC(i, k)$. This is because these indices are left dominated by j; (Step 3) Insert j at the tail of the queue; (Step 4) Set ℓ_{ij} as the index at the head of L_i.

Using the size of L_i as a potential function, one easily show that the time for computing some ℓ_{ij} is amortized constant. This is because the only nonlinear time operations are the deletions in Step 1 and 2, and here the runtime coincides with the decrease of the potential function. Thus we have the following theorem.

Theorem 3. *There is an $O(n^2)$ time algorithm that finds the optimal search tree for the problem of searching in a weighted path.*

5   An $o(\log n)$ Approximation Algorithm

We shall first present two natural lower bounds on the cost of the optimal decision tree for a given weighted tree. We shall use $OPT(T, \mathbf{c})$ to denote the cost of an optimal decision tree for the weighted tree instance $(T, \mathbf{c})$. When the cost assignment is clear from the context we use $OPT(T)$ instead of $OPT(T, \mathbf{c})$. In this section, by writing $T' \subseteq T$ we mean that T' is a subtree obtained from T by the deletion of vertices and edges. We prepare the following easy lower bounds on the cost of an optimal solution, whose proof is omitted due to the space constraints.

Lemma 7. $OPT(T, \mathbf{c}) \geq \max_{T' \subseteq T} OPT(T', \mathbf{c})$

Lemma 8. $OPT(T, \mathbf{c}) \geq \max_{T' \subseteq T} c_{min}(T') \log |T'|$, where $c_{min}(T')$ is the minimum cost of an edge of T' according to the cost assignment $\mathbf{c}$.

For a forest F we denote by $\mathscr{K}(F)$ the set of connected components (trees) in F. We use $K(F)$ to denote the size of the largest component of F, i.e., $K(F) = \max_{C \in \mathscr{K}(F)} |C|$.

Let t be a parameter whose exact value will be determined in the course of the analysis. The algorithm makes use of a maximal set of vertices S such that all edges of the subtree $T[S]$, induced by S, are reasonably good separators for T, that is, $K(T - e) \leq (1-t)|T|$ for every $e \in T[S]$. More specifically, the set S is constructed starting with the centroid of T (i.e., the unique vertex v with $K(T - v) \leq |T|/2$ and then keeping on

adding vertices v such that v is adjacent to some vertex $u \in S$ for which $K(T - \{u, v\}) \leq (1 - t)|T|$. This procedure stops when there is no vertex in $T - S$ which satisfies the required conditions.

Proposition 1. *Each component of $\mathcal{K}(T - S)$ has size at most $t \cdot |T|$.*

Proof. Let C be a component of $\mathcal{K}(T - S)$ and let e be the edge that connects C to $T[S]$. In addition, let u be the endpoint of e that belongs to C. Since u is not added to S, the larger component of $T - e$ has size larger than $(1 - t)|T|$, hence the smaller component of $T - e$ has size smaller than or equal to $t|T|$. This smaller component must be C, for otherwise C would contain the centroid vertex, contradicting the construction of S. $\quad\square$

The algorithm constructs the set S as explained above and then, depending on the size of S, proceeds with one of the following two cases:

Case 1: $|S| > \log n$. Let $e^* = \{u, v\}$ be the edge of minimum cost in S. Let T_u (resp. T_v) be the connected component of $T - e^*$ containing u (resp v). The algorithm probes the edge e^* and then recurses in the subtrees T_u and T_v. The choice of e^* guarantees that the algorithm probes an edge which is both cheap and a reasonably good separator. This yields the following equation

$$APP(|T|) \leq \frac{c(e^*) + \max\{cost(T_v), cost(T_u)\}}{OPT(T)} \leq 1/\log\log n + APP((1 - t) \cdot |T|),$$

where $APP(s)$ denotes the approximation ratio achieved by the algorithm on an input tree with s vertices. In the above equation, $c(e^*)/OPT(T) \leq 1/\log|S| \leq 1/\log\log n$ follows from Lemma 8.

Case 2: $|S| \leq \log n$. In this case the algorithm takes advantage of the fact that S is a good separator for T (Proposition 1) and that an optimal decision tree for S can be constructed in $O(n \log n)$ through dynamic programming (cf. [4]).

Let S^T be the vertices of S that are adjacent to some vertex in $T - S$. The algorithm proceeds as follows: (i) Build an optimal decision tree for $T[S]$; (ii) For each $v \in S^T$ build the optimal decision tree for S_v, where S_v is the star induced by v and the vertices of $T - S$ adjacent to v. (iii) Recurse in the components of $\mathcal{K}(T - S)$; (iv) Assemble a decision tree D for T as follows: (a) For each $v \in S^T$, replace the leaf of the optimal tree for $T[S]$ corresponding to v with the optimal decision tree for S_v; (b) For each $v \in S^T$ and for each $w \in S_v - \{v\}$, replace the leaf of the optimal decision tree for S_v corresponding to w with the decision tree recursively constructed for the component of $T - S$ that contains w.

Note that any decision tree for a star is optimal. The construction yields the following estimate of the approximation achieved by the algorithm:

$$APP(|T|) \leq \frac{OPT(T[S]) + \max_{v \in S^T}\{OPT(S_v)\} + \max_{C \in \mathcal{K}(T-S)}\{cost(C)\}}{OPT(T)} \leq 2 + APP(t \cdot |T|) .$$

Putting together the two cases, we obtain

$$App(|T|) \leq \max\{1/\log\log n + APP((1 - t) \cdot |T|), 2 + APP(t \cdot |T|)\}$$

By setting $t = 1/\log\log\log n$, standard methods for solving recurrences give $APP(n) = O(\log n / \log\log\log\log n) = o(\log n)$. Note that t is fixed throughout the algorithm, that is, it does not change together with the size of the different trees which are considered in the recursion process. Summarizing, we have shown the following result.

Theorem 4. *There is an $o(\log n)$-approximation algorithm for the problem of searching in weighted trees.*

References

1. Ben-Asher, Y., Farchi, E., Newman, I.: Optimal search in trees. SIAM Journal on Computing 28(6), 2090–2102 (1999)
2. Charikar, M., Fagin, R., Guruswami, V., Kleinberg, J.M., Raghavan, P., Sahai, A.: Query strategies for priced information. J. of Comp. and System Sc. 64(4), 785–819 (2002)
3. Cicalese, F., Jacobs, T., Laber, E., Molinaro, M.: On greedy algorithms for decision trees. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 206–217. Springer, Heidelberg (2010)
4. Cicalese, F., Jacobs, T., Laber, E., Molinaro, M.: On the complexity of searching in trees: Average-case minimization. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 527–539. Springer, Heidelberg (2010)
5. de la Torre, P., Greenlaw, R., Schäffer, A.: Optimal edge ranking of trees in polynomial time. Algorithmica 13(6), 592–618 (1995)
6. Dereniowski, D.: Edge ranking of weighted trees. DAM 154, 1198–1209 (2006)
7. Dereniowski, D.: Edge ranking and searching in partial orders. DAM 156, 2493–2500 (2008)
8. Garey, M.: Optimal binary identification procedures. SIAM J. Appl. Math. 23(2), 173–186 (1972)
9. Iyer, A., Ratliff, H., Vijayan, G.: On an edge ranking problem of trees and graphs. Discrete Applied Mathematics 30(1), 43–52 (1991)
10. Iyer, A.V., Ratliff, H.D., Vijayan, G.: On an edge ranking problem of trees and graphs. Discrete Appl. Math. 30, 43–52 (1991)
11. Knuth, D.: Optimum binary search trees. Acta. Informat. 1, 14–25 (1971)
12. Laber, E., Milidiú, R., Pessoa, A.: On binary searching with non-uniform costs. In: Proc. of SODA 2001, pp. 855–864 (2001)
13. Laber, E., Nogueira, L.: On the hardness of the minimum height decision tree problem. Discrete Applied Mathematics 144(1-2), 209–212 (2004)
14. Lam, T.W., Yue, F.L.: Optimal edge ranking of trees in linear time. In: Proc. of SODA 1998, pp. 436–445 (1998)
15. Lipman, M., Abrahams, J.: Minimum average cost testing for partially ordered components. IEEE Transactions on Information Theory 41(1), 287–291 (1995)
16. Makino, K., Uno, Y., Ibaraki, T.: On minimum edge ranking spanning trees. J. Algorithms 38, 411–437 (2001)
17. Mozes, S., Onak, K., Weimann, O.: Finding an optimal tree searching strategy in linear time. In: Proc. of SODA 2008, pp. 1096–1105 (2008)
18. Onak, K., Parys, P.: Generalization of binary search: Searching in trees and forest-like partial orders. In: Proc. of FOCS 2006, pp. 379–388 (2006)
19. Yao, F.F.: Efficient dynamic programming using quadrangle inequalities. In: Proc. of STOC 1980, pp. 429–435 (1980)

Computing the Fréchet Distance between Folded Polygons[*]

Atlas F. Cook IV[1], Anne Driemel[1], Sariel Har-Peled[2],
Jessica Sherette[3], and Carola Wenk[3]

[1] Department of Information and Computing Sciences,
University of Utrecht, Netherlands
{atlas,driemel}@cs.uu.nl
[2] Department of Computer Science, University of Illinois, USA
sariel@uiuc.edu
[3] Department of Computer Science, University of Texas at San Antonio, USA
{jsherett,carola}@cs.utsa.edu

Abstract. Computing the Fréchet distance for surfaces is a surprisingly hard problem and the only known algorithm is limited to computing it between flat surfaces. We adapt this algorithm to create one for computing the Fréchet distance for a class of non-flat surfaces which we call folded polygons. Unfortunately, the original algorithm cannot be extended directly. We present three different methods to adapt it. The first of which is a fixed-parameter tractable algorithm. The second is a polynomial-time approximation algorithm. Finally, we present a restricted class of folded polygons for which we can compute the Fréchet distance in polynomial time.

Keywords: Computational Geometry, Shape Matching, Fréchet Distance.

1 Introduction

The *Fréchet distance* is a similarity metric for continuous shapes such as curves and surfaces. In the case of computing it between two (directed open) curves there is an intuitive explanation of the Fréchet distance. Suppose a man walks along one curve, a dog walks along the other, and they are connected by a leash. They can vary their relative speeds but cannot move backwards. Such a walk pairs every point on one curve to one and only one point on the other curve (i.e., creates homeomorphism) in a continuous way. The Fréchet distance of the curves is the minimum leash length required for the man and dog to walk along these curves. Although less intuitive, the idea is similar for surfaces.

While the Fréchet distance between polygonal curves can be computed in polynomial time [1], computing it between surfaces is much harder. In [2] it was shown that even computing the Fréchet distance between a triangle and

---

a self-intersecting surface is NP-hard. This result was extended in [3] to show that computing the Fréchet distance between 2d terrains as well as between polygons with holes is also NP-hard. Furthermore, while in [4] it was shown to be upper semi-computable, it remains an open question whether the Fréchet distance between general surfaces is even computable.

On the other hand, in [5] a polynomial time algorithm is given for computing the Fréchet distance between two (flat) simple polygons. This was the first paper to give any algorithm for computing the Fréchet distance for a nontrivial class of surfaces and remains the only known approach. Our contribution is to generalize their algorithm to a class of non-flat surfaces we refer to as folded polygons. Given that theirs is the only known approach it is of particular importance to explore extending it to new classes of surfaces. The major problem we encountered was that the mappings between the folded polygons which need to be considered are less restricted than those between simple polygons. We address three different methods to resolve this problem. In Sect.3, we outline a fixed-parameter tractable algorithm. In Sect.4, we describe a polynomial-time approximation algorithm to compute the Fréchet distance between folded polygons within a constant factor. In Sect.5, we describe a nontrivial class of folded polygons for which the original algorithm presented in [5] will compute an exact result.

2   Preliminaries

The Fréchet distance is defined for two k-dimensional hypersurfaces $P, Q : [0, 1]^x \to \mathbb{R}^d$, where $x \leq d$, as

$$\delta_F(P, Q) = \inf_{\sigma:A \to B} \sup_{p \in A} \|P(p) - Q(\sigma(p))\|$$

where σ ranges over orientation-preserving homeomorphisms that map each point $p \in P$ to an image point $q = \sigma(p) \in Q$. Lastly, $\| \cdot \|$ is the Euclidean norm but other metrics could be used instead.

Let $P, Q : [0, 1]^2 \to \mathbb{R}^d$ be connected polyhedral surfaces for each of which we have a convex subdivision. We assume that the dual graphs of the convex subdivisions are acyclic, which means that the subdivisions do not have any interior vertices. We will refer to surfaces of this type as *folded polygons*. We refer to the interior convex subdivision edges of P and Q as *diagonals* and *edges* respectively. Let m and n be the complexities of P and Q respectively. Let k and l be the number of diagonals and edges respectively. Assume without loss of generality the number of diagonals is smaller than the number of edges. Let $T_{matrixmult}(N)$ denote the time to multiply two $N \times N$ matrices.

2.1   Simple Polygons Algorithm Summary

Buchin et al. [5] show that, while the Fréchet distance between a convex polygon and a simple polygon is the Fréchet distance of their boundaries, this is not the case for two simple polygons. They present an algorithm to compute the

Fréchet distance between two simple polygons P, Q. The idea is to find a convex subdivision of P and map each of the convex regions of it continuously to distinct parts of Q such that taken together the images account for all of Q.

The decision problem $\delta_F(P, Q) \leq \varepsilon$ can be solved by (1) mapping the boundary of P, which we denote by ∂P, onto the boundary of Q, which we denote by ∂Q, such that $\delta_F(\partial P, \partial Q) \leq \varepsilon$ and (2) mapping each diagonal d in the convex subdivision of P to a shortest path $f \subseteq Q$ such that both endpoints of f lie on ∂Q and such that $\delta_F(d, f) \leq \varepsilon$.

In order to solve subproblem (1) they use the notion of a free space diagram. For open curves $f, g : [0, 1] \to \mathbb{R}^d$ it is defined as $FS_\varepsilon(f, g) = \{(x, y) \mid x \in f, \ y \in g, \ \|x - y\| \leq \varepsilon\}$ where $\varepsilon \geq 0$. A monotone path starting at the bottom left corner of the free space diagram going to the top right exists if and only if the curves are within Fréchet distance ε. As shown in [1], this can be extended to closed curves by concatenating two copies of the free space diagram to create a *double free space diagram* and searching for a monotone path which covers every point in P exactly once, see Fig.1. This algorithm can be used to show whether $\delta_F(\partial P, \partial Q) \leq \varepsilon$ and find the particular mapping(s) between ∂P and ∂Q. In turn this defines a *placement* of the diagonals, i.e., a mapping of the endpoints of the diagonals to endpoints of the corresponding image curves in Q.
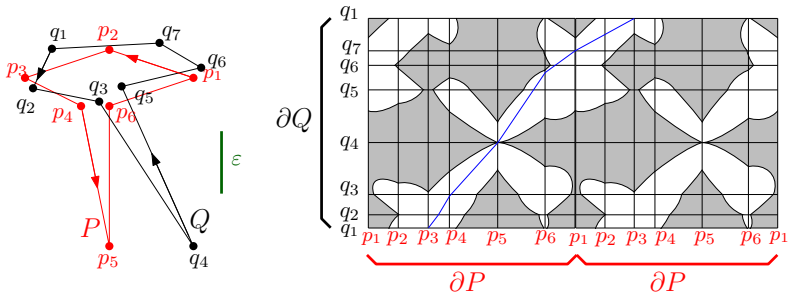
Fig. 1. The white areas are those in free space diagram. The surfaces are within Fréchet distance ε since there is a monotone path starting at the bottom of the free space diagram and ending at the top which maps every point on the boundary of P exactly once. This figure was generated using an ipelet created by Günter Rote.

Subproblem (2) is solved by only considering paths through the free space diagram that map a diagonal d onto an image curve f such that $\delta_F(d, f) \leq \varepsilon$. Naturally the particular placement of the diagonals determined in subproblem (1) could affect whether this is true. Therefore, they must check this for many paths in the free space diagram. Fortunately they can show that it is sufficient to only consider mapping a diagonal to an image curve which is the shortest path between the end points determined by the placement.

Solving these subproblems generates a mapping between P and Q for ε. This mapping might not be a homeomorphism but the authors show by making very small perturbations of image curves the mapping can be made into one. These perturbations can be arbitrarily small. Thus, because the Fréchet distance is the

infimum of all homeomorphisms on the surfaces, the Fréchet distance is ε. For simplicity we will refer to these generated mappings as homeomorphisms. By performing a binary search on a set of critical values they can use the above algorithm for the decision problem to compute the Fréchet distance of P and Q.

2.2   Shortest Path Edge Sequences

Our algorithm extends the simple polygons algorithm to one for folded polygons. The idea of the algorithm is to subdivide one surface, P, into convex regions and pair those with corresponding regions in the other surface, Q. Those regions of Q are now folded polygons rather than just simple polygons. The authors of [5] show that the Fréchet distance of a convex polygon and a simple polygon is just the Fréchet distance of their boundaries. Using essentially the same argument we prove the following lemma.

Lemma 1. *The Fréchet distance between a folded polygon P and a convex polygon Q is the same as that between their boundary curves.*

For the simple polygon algorithm it suffices to map diagonals onto shortest paths between two points on ∂Q. By contrast, there are folded polygons where a homeomorphism between the surfaces does not exist when diagonals are mapped to shortest paths but does exist when the paths are not restricted, see Fig.2. We must therefore consider mapping the diagonals to more general paths. Fortunately, we can show that these more general paths still have some nice properties for folded polygons.
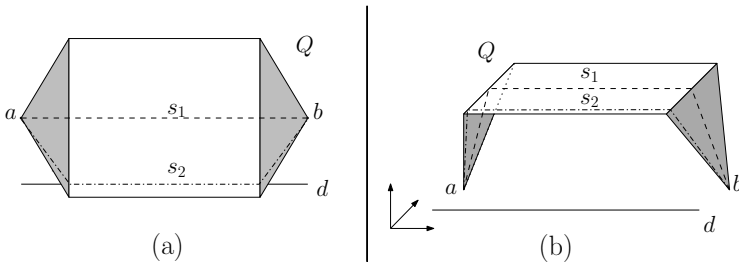
Fig. 2. The curve s_1 is the shortest path in Q between the points a and b but the curve s_2 has smaller Fréchet distance to d than s_1 has. (a) overhead view, (b) sideview

Lemma 2. *Let u and v be points such that $u, v \in \partial Q$, $E = \{e_0, e_1, \ldots, e_s\}$ be a sequence of edges in the convex subdivision of Q, and d be a line segment. Given $\varepsilon > 0$, a curve f in Q that follows the edge sequence E from u to v such that $\delta_F(d, f) \leq \varepsilon$ can be computed, if such a curve exists, in $O(s)$ time.*

Proof. We construct a series $F = FS_\varepsilon(e_0, d), FS_\varepsilon(e_1, d), \ldots, FS_\varepsilon(e_s, d)$, of 2-dimensional free space diagrams. Any two edges e_i and e_{i+1} are on the boundary of the same convex polygon in the convex subdivision of Q so we can assume

without loss of generality that f consists of straight line segments between the edge intersections. This is similar to the shortcutting argument used to prove Lemma 3 in [5]. Thus, we only need to check the points where f crosses an edge of Q. For $\delta_F(d, f) \leq \varepsilon$ to be true, the preimages of those crossing points must be monotone along d. Let $FS'_\varepsilon(e_i, d)$ be the projection of $FS_\varepsilon(e_i, d)$ onto d. Let $F' = FS'_\varepsilon(e_1, d), FS'_\varepsilon(e_2, d), \dots, FS'_\varepsilon(e_s, d)$.

To verify that the preimage points on d can be chosen such that they are monotone, we check the intervals of F'. Specifically, for $i < j$, the point on d mapped to e_i must come before the one mapped to e_j. This can be checked by greedily scanning left to right and always choosing the smallest point on d which can be mapped to some edge. A search of this form takes $O(s)$ time. □

The dual graph of the faces of Q is acyclic. This implies that there is a unique sequence of faces through which a shortest path from u to v, where $u, v \in \partial Q$, must pass. Necessarily, there must also be a unique edge sequence that the shortest path follows. We refer to this as the *shortest path edge sequence.*

Lemma 3. *Let Q be a folded polygon and d be a diagonal. If there is a curve $f \subseteq Q$ with $\delta_F(d, f) \leq \varepsilon$ then there is a curve $g \subseteq Q$ which follows the* shortest path edge sequence *such that $\delta_F(d, g) \leq \varepsilon$.*

Proof. Let E_f and E_g be the edge sequences of f and g respectively. By definition the dual graph of the faces of Q is acyclic, so E_g must be a subsequence of E_f. E_g induces a sequence of free space intervals. If there is a monotone path in the free space interval sequence induced by E_f, we can cut out some intervals and have a monotone path in the free space for E_g. □

From Lemma 3, we just need to consider paths that follow the shortest path edge sequence. We refer to paths that follow this edge sequence and consist of straight line segments between edges as *Fréchet shortest paths*. In addition, s in Lemma 2 is bounded by the number of edges along the shortest path edge sequence between u and v. This implies the following theorem.

Theorem 1. *Let Q be a folded polygon, u and v be points such that $u, v \in \partial Q$, and d be a line segment. Given $\varepsilon > 0$, we can in $O(l)$ time find a curve f in Q from u to v such that $\delta_F(d, f) \leq \varepsilon$ if such a curve exists.*

Suppose we have a homeomorphism between ∂P and ∂Q. The endpoints of the image curves must appear on ∂Q in the same order as their respective diagonal endpoints on ∂P. The homeomorphism also induces a direction on the diagonals in P and on the edges in Q. Specifically, we consider diagonals and edges to start at their first endpoint along ∂P or ∂Q respectively in a counterclockwise traversal of the boundaries. We denote by D_e the set of diagonals whose associated shortest path edge sequences contain an edge $e \subseteq Q$. Observe that pairwise non-crossing image curves must intersect an edge e in the same order as their endpoints occur on ∂Q. We refer to this as the *proper intersection order* for an edge e.

2.3   Diagonal Monotonicity Test and Untangleability

We now define a test between two folded polygons P and Q which we call the *diagonal monotonicity test*. For a given ε this test returns true if the following two things are true. First, $\delta_F(\partial P, \partial Q) \le \varepsilon$. Second, for every diagonal d_i in the convex subdivision of P, the corresponding Fréchet shortest path f_i in Q has $\delta_F(d_i, f_i) \le \varepsilon$. We refer to the class of mappings of the folded polygons generated by this test as *monotone diagonal mappings*. This is similar to the test used by [5] except ours uses Fréchet shortest paths instead of the shortest paths.

Unfortunately, because the image curves of the diagonals are no longer shortest paths, they may cross each other and we will no longer be able to generate a mapping between the folded polygons which is a homeomorphism. Thus the diagonal monotonicity test might return true when in fact a homeomorphism does not exist. We must explicitly ensure that the image curves of all diagonals are non-crossing. In particular, we refer to a set of image curves $F = \{f_1 \ldots f_k\}$ as *untangleable* for ε if and only if there exists a set of image curves $F' = \{f'_1 \ldots f'_k\}$ where f_i and f'_i have the same end points on ∂Q, $\delta_F(d_i, f'_i) \le \varepsilon$, and the curves of F' are pairwise non-crossing. A homeomorphism exists between the folded polygons for ε if and only if there exists a monotone diagonal mapping whose image curves are untangleable for ε. The proof of this is straight forward and, due to lack of space, is omitted.

As shown in Theorem 1, computing Fréchet shortest paths instead of shortest paths does not increase the asymptotic run time. To optimize this ε we can perform a binary search on a set of critical values. As in [5], the number of critical values is $O(m^2 n + m n^2)$. The three types of critical values between a diagonal and its corresponding path through Q are very similar to those outlined in the simple polygons algorithm. So, by following the paradigm set forth by [5], we arrive at the following theorem:

Theorem 2. *The minimum ε for which two folded polygons P and Q, pass the diagonal monotonicity test can be computed in time $O(k T_{matrixmult}(mn) \log(mn))$.*

3   Fixed-Parameter Tractable Algorithm

In this section we outline an algorithm to decide for a fixed mapping between the boundaries of a pair of folded polygons whether the image curves induced from the mapping are untangleable. From this we create a fixed-parameter tractable algorithm for computing the Fréchet distance between a pair of folded polygons.

3.1   Untangleability Space

Let e be an edge in Q which is crossed by the image curves of h diagonals, $d_1, \ldots, d_h$. We assume without loss of generality that the image curves of the diagonals cross e in proper intersection order if, for all $1 \le i, j \le h$ where $i < j$, the image curve of d_i crosses e before the image curve of d_j crosses e. Let the *untangleability space* U_e contain all h-tuples of points on the diagonals which can

be mapped to crossing points on the edge e within distance ε and such that the crossing points are in the proper intersection order along e. U_e can be shown to be convex yielding the following theorem.

Theorem 3. $U_e(d_1, \ldots, d_h)$ *is convex.*

This theorem can be proven by linearly interpolating between points in U_e. Due to lack of space the proof is deferred to the full version of this paper.

3.2   Fixed-Parameter Tractable Algorithm

We assume the complexity of k and l, the convex subdivisions of P and Q are constant. Checking for the existence of a set of image curves which are untangleable can be done by using the untangleability spaces of the edges. Assume we are given some homeomorphism between ∂P and ∂Q from which we get a placement of the diagonals. We first choose an edge in Q to act as the root of the *edge tree* that corresponds to the dual graph of Q. We propagate constraints imposed by each untangleability space up the tree to the root node to determine if the set of image curves induced by the placement of the diagonals is untangleable.

The untangleability space of an edge e, U_e, contains exactly those sets of points on the diagonals in D_e which can be mapped to the edge in the proper intersection order. The point chosen in U_e imposes a constraint on what points may be chosen in other untangleability spaces. In particular the corresponding points on all of the diagonals must be monotone with respect to their edge sequence. We define $C(U_{e_i})$ as the Minkowski sum of U_{e_i} with a ray in the opposite direction of the constraint on each of the diagonals in D_{e_i}, see Fig.3. The direction of this constraint depends on which side of the edge e_i the next edge is. $C(U_{e_i})$ contains exactly those sets of points on the diagonals not excluded from having a monotone mapping with U_{e_i}.
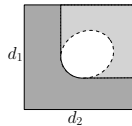
Fig. 3. U_e is shown in white and $C(U_e)$ is the union of the white and light gray portions

We define for every edge e a k-dimensional *propagation space* P_e. If e is a leaf in the tree, then $P_e = U_e$. Otherwise, define

$$P_e = U_e \cap C(P_{e_1}) \cap \ldots \cap C(P_{e_j})$$

where e is the parent of the edges $e_1, e_2, \ldots, e_j$. $C(P_{e_j})$ contains only those points that are not excluded by the constraints of the tree rooted at e_j from being used to untangle on the parent of e_j. The propagation space for the root will be empty if and only if this set of image curves are not untangleable. From our

assumptions, the propagation space of the root can be computed in constant time as the intersection of semi-algebraic sets [6]. Let $F(k, l)$ be the time complexity of computing this intersection.

Consider two different mappings between ∂P and ∂Q. These determine different placements of the diagonals. If all of the image curves of all of the diagonals have the same shortest path edge sequence in both of the mappings the test will return the same result. Thus, we only need to test paths through the free space diagram which cross the diagonals and edges in a different order. The free space diagram for ∂P and ∂Q contains $2k$ vertical line segments that will each contribute $O(kl)$ different mappings of the diagonals and edges. Hence, there are $O((kl)^{2k})$ paths through the free space diagram which we need to test. For each of these we can check whether a global untangling exists as described above in constant time. Similar to the algorithm for polygonal curves [1] we can perform Cole's [7] technique for parametric search [8] to optimize the value of ε. For a constant number of edges l and diagonals k, this yields a fixed-parameter tractable algorithm with runtime polynomial in m and n.

Theorem 4. *We can compute the Fréchet distance of two folded polygons in time $O((F(k, l)(kl)^{2k} + kT_{matrixmult}(mn)) \log(mn))$.*

4   Constant Factor Approximation Algorithm

In this section we present an approximation algorithm which avoids the problem tangles altogether. First we prove the following theorem:

Theorem 5. *If two folded polygons, P and Q, pass the diagonal monotonicity test for some ε, then $\delta_F(P, Q) \leq 9\varepsilon$.*

Proof. Consider the image curves of the diagonals of P found by performing our diagonal monotonicity test. To pass the diagonal monotonicity test there must be a homeomorphism between ∂P and ∂Q. The image curves of the diagonals will be mapped in the proper order along the boundary of Q. Therefore, if a pair of image curves cross in Q they must do so an even number of times.

Take two consecutive points u, v on the boundary of P that are connected by a diagonal $d = \overline{uv}$, and consider the convex ear of P that d "cuts off". Let d' be the image curve of d in Q. In order to create a homeomorphism between P and Q, d' should cut off an ear of Q which can be mapped to the ear of P. Unfortunately, some image curves may cross this d' and cause tangles. Consider the arrangement of image curves in Q and let d'' be the highest level of this arrangement closest to the top of the ear (∂Q), such that d'' connects the image points u and v on the boundary of Q.

Observe that if an image curve d'_1 crosses d' from below then that intersection point a' has a pre-image on both d and d_1. These points a on d and a_1 on d_1 can be no more than 2ε apart since they both map to the intersection within distance ε. In addition, d'_1 must cross back below d' eventually since all image curves which cross do so an even number of times (take the first such occurrence

after the initial crossing). The preimage points b and b_1 of this second intersection b' are also no more than 2ε apart. Since both d and d' are line segments, every point on the line segment $\overline{ab}$ on d is $\leq 2\varepsilon$ distance from some point on the line segment $\overline{a_1b_1}$ on d_1. For an approximation of 3ε we can map a point on d to its corresponding point on d_1 and then to where that point maps on d_1', see Fig.4(a). Thus, d can be mapped to an image curve above d_1' within Fréchet distance 3ε.
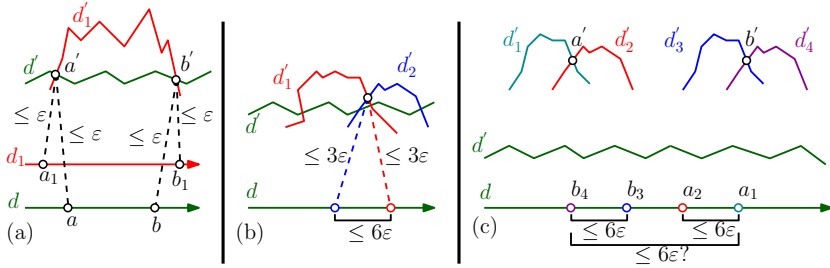

Fig. 4. (a) the intersections between d and d_1 imply that $\overline{ab}$ can be mapped to the region of d_1' between a' and b' within Fréchet distance 3ε. (b) this is the preimage of the intersection between d_1' and d_2' on d. (c) this is an example of the preimages of two intersections occurring out of order of d.

If this image curve d_1' then crosses another image curve d_2' this argument above cannot be just repeated because the approximation factor would depend linearly on the number of image curves which cross each other. The preimage points on d of such an intersection not involving d' are separated by at most 6ε. This is because both diagonals involved in the intersection have a 3ε correspondence between the region of them mapped above d' and d. If the preimages are in order there is no problem. If they occur out of order they cause a monotonicity constraint. Fortunately, we can collapse this region on d to the leftmost preimage with 6ε and then map it to the corresponding point on d_2' in 3ε for a total of 9ε, see Fig.4(b).

Thus, we can approximate away single intersections with 9ε. We must also verify that if the preimages of single intersections occur out of order it does not effect our approximation, see Fig.4(c). Due to lack of space the proof of these technical cases is deferred to the full version of this paper.

From these we get that $\delta_F(d, d'') \leq 9\varepsilon$. Now collapse the ear we initially selected in P to d. Likewise in Q collapse the corresponding ear to d''. This is okay because d'' is above all of the other image curves in Q. This pairs the ear we cut off of P with the part of Q above d'' which is a folded polygon. By Lemma 1 these regions must be within Fréchet distance 9ε.

Choose another ear in P. We can repeat the above arguments to remove this new ear and its corresponding ear in Q. The dual graph of P is a tree. Each time we repeat this argument we are removing a leaf from the tree. Eventually, the tree will contain only a single node which corresponds to some convex portion of P which we map to the remainder of Q. □

As a direct consequence of Theorems 2 and 5 we get the following theorem:

Theorem 6. *We can compute a* 9*-approximation of the Fréchet distance of two folded polygons in time* $O(kT_{matrixmult}(mn)\log(mn))$.

5   Axis-Parallel Folds and L_∞ Distance

In this section we outline a special class of surfaces for which using the L_∞ metric allows us to avoid the problem of untangling. Specifically, if all of the line segments in the convex subdivision of the surfaces are parallel to the x-axis, y-axis, or z-axis, we show that it is sufficient to use shortest paths instead of Fréchet shortest paths. Since shortest paths never cross we can use the simple polygons algorithm [5] to compute the Fréchet distance of the surfaces. We first prove the following lemma.

Lemma 4. *Let R be a half-space such that the plane bounding it, ∂R, is parallel to the xy-plane, yz-plane, or xz-plane. Given a folded polygon Q with edges parallel to the x-axis, y-axis, or z-axis and points $a, b \in Q \cap R$, let f be a path in Q, which follows the shortest path edge sequence between a and b. If f is completely inside of R so is the shortest path f' between a and b.*

For the lemma to be false there must exist a Q, R, and f which serve as a counter example. There must be at least one edge e_j in Q such that $f \cap e_j \in R$ and $f' \cap e_j \notin R$. In particular, let e_j be the first edge where this occurs along the shortest path edge sequence. First consider a Q where all of the edges of it are perpendicular to ∂R. A line segment in the shortest path f' connects the endpoints of two edges in Q. Let e_i and e_k be the edges that define the line segment in f' that passes through e_j. We now consider several cases in how those edges are positioned.

Case (I) occurs when e_k is completely outside of R, see Fig.5(a). While this does force f' to cross e_j outside of R, there is no f which can pass through e_k while remaining inside of R. Because Q is a folded polygon any path between a and b must path through the edges in the shortest path edge sequence including e_k. Thus no f can exist entirely within R.

Case (II) occurs when part of e_k is in R and f' crosses it in the part in R, see Fig.5(b). In this case f' does not cross e_j outside of R.

Due to space limitations the discussion of the remaining cases have been omitted. Each of these remaining cases can be reduced to these first two. Using this lemma we can prove the following theorem:

Theorem 7. *The Fréchet distance between two surfaces, P and Q, both with only diagonals/edges parallel to the x-axis, y-axis, or z-axis, can be computed in time* $O(kT_{matrixmult}(mn)\log(mn))$.

Proof. Let d be a diagonal and f' be the shortest path between points a and c on ∂Q. Using Lemma 4 we prove that if there exists a Fréchet shortest path f between points a and c such that $\delta_F(d, f) \leq \varepsilon$, then $\delta_F(d, f') \leq \varepsilon$.
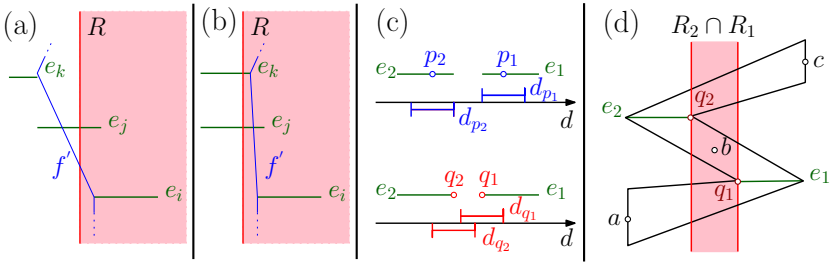
Fig. 5. (a), (b) are examples of case (I) and case (II). (c) is an example of intervals for the two different paths. (d) together the edges e_1 and e_2 cause a monotonicity constraint.

Minkowski Sum Constraints. Since we are using the L_∞ distance, the unit ball is a cube. The Minkowski sum of a diagonal d in P and a cube of side length ε yields a box. Points in the diagonal d can only map to points in this region. It can be defined by the intersection of 6 half-spaces; all of these have boundaries parallel to either the xy-axis, the xz-axis, or the yz-axis. Thus, from Lemma 4, we know that if any path through Q is completely within this box, then the shortest path f' will be, too. This means that for each edge e_i on the shortest path edge sequence $f' \cap e_i$ is within distance ε of some non-empty interval of d.

Monotonicity Constraints. For the shortest path f' between the boundary points to have $\delta_F(d, f') > \varepsilon$, at least two of these intervals must be disjoint and occur out of order along d, see Fig.5(c). Such a case introduces a monotonicity constraint on ε. If no such intervals existed then we could choose a monotone sequence of points along d such that each point is within distance ε of an edge and the sequence of edges they map to would have the same order as the shortest path edge sequence showing that $\delta_F(d, f') \leq \varepsilon$.

Let e_1 and e_2 be two edges along the shortest path edge sequence for which such bad intervals occur. Let p_1 and p_2 be points on the shortest path where it intersect edges e_1 and e_2 respectively. Let q_1 and q_2 be the same for f. Finally, let d_r denote all of the points on d which are within distance ε of the point r. Since $\delta_F(d, f) \leq \varepsilon$, d_{q_1} and d_{q_2} must overlap or occur in order along d.

Let R_1 be the half-space whose bounding plane contains q_1 and is perpendicular to d. likewise let R_2 be the half-space whose bounding plane contains q_2 and is perpendicular to d, see Fig.5(d). Let R_1 extend to the left along d and R_2 extend to the right along d. ∂R_2 must occur before ∂R_1 along d or the edges are in order and no monotonicity constraint is imposed. Assume R_1 encloses all of f' between a and e_2. If it does not we can choose a new edge between a and e_2 to use as e_1 for which this is true. Doing so only increases the monotonicity constraint. Likewise we can assume R_2 encloses all of f' between e_2 and c.

Assume a, b, and c lie on f'. Specifically, let a and c be the end points of f' on ∂Q. Naturally, a shortest path must exist between a and c and it must contain at least one point in $R_1 \cap R_2$ which we call b. f follows the shortest path edge

sequence between a and c, so it must also cross all of the edges in the shortest path edge sequence between a and b. Therefore, to show that p_2 is inside of R_1 we can directly apply Lemma 4 to the points a and b. A similar method can be used for e_2 with points b and c to show p_1 is inside R_2. Since d_{q_1} and d_{q_2} overlap or are in order, d_{p_1} and d_{p_2} must as well. Therefore, $\delta_F(d, f^{'}) \leq \varepsilon$ and shortest paths can be used for this variant of folded polygons instead of Fréchet shortest paths. Because we are using shortest paths we can just use the simple polygons algorithm. This yields Theorem 7. □

6   Future Work

The constant factor approximation outlined in Sect.4 can likely still be improved. Specifically, we consider only the worst case for each of the out-of-order mappings which may not be geometrically possible to realize. In addition, we currently approximate the Fréchet distance by mapping image curves one-by-one to the top of the arrangement of other image curves. It would of course be more efficient to untangle image curves by mapping them to some middle curve rather than forcing one to map completely above the others.

Finally, while the problem of untangling seems hard, it is also possible that a polynomial-time exact algorithm could exist. The acyclic nature of our surfaces seems to limit the complexity of our mappings. The methods used to prove that computing the Fréchet distance between certain classes of surfaces is NP-hard in [3] are not easy to apply to folded polygons.

References

1. Alt, H., Godau, M.: Computing the Fréchet distance between two polygonal curves. International Journal of Computational Geometry and Applications 5, 75–91 (1995)
2. Godau, M.: On the complexity of measuring the similarity between geometric objects in higher dimensions. PhD thesis. Freie Universität Berlin, Germany (1998)
3. Buchin, K., Buchin, M., Schulz, A.: Fréchet distance of surfaces: Some simple hard cases. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 63–74. Springer, Heidelberg (2010)
4. Alt, H., Buchin, M.: Can we compute the similarity between surfaces? Discrete and Computational Geometry 43, 78–99 (2010)
5. Buchin, K., Buchin, M., Wenk, C.: Computing the Fréchet distance between simple polygons in polynomial time. In: 22nd Symposium on Computational Geometry (SoCG), pp. 80–87 (2006)
6. Basu, S., Pollack, R., Roy, M.F.: Algorithms in real algebraic geometry. Algorithms and Computation in Mathematics (2006)
7. Cole, R.: Slowing down sorting networks to obtain faster sorting algorithms. J. ACM 34, 200–208 (1987)
8. Megiddo, N.: Applying parallel computation algorithms in the design of serial algorithms. J. ACM 30, 852–865 (1983)

Parameterized Reductions and Algorithms for Another Vertex Cover Generalization

Peter Damaschke and Leonid Molokov

Department of Computer Science and Engineering,
Chalmers University, 41296 Göteborg, Sweden
{ptr,molokov}@chalmers.se

Abstract. We study a novel generalization of the VERTEX COVER problem which is motivated by, e.g., error correction in the inference of chemical mixtures by their observable reaction products. We focus on the important case of deciding on one of two candidate substances. This problem has nice graph-theoretic formulations situated between VERTEX COVER and 3-HITTING SET. In order to characterize their parameterized complexity we devise parameter-preserving reductions, and we show that some minimum solution can be computed faster than by solving 3-HITTING SET in general. More explicitly, we introduce the UNION EDITING problem: In a hypergraph with red and blue vertices, edit the colors so that the red set becomes the union of some hyperedges. The case of degree 2 is equivalent to STAR EDITING: In a graph with red and blue edges, edit the colors so that the red set becomes the union of some stars, i.e., vertices with *all* their incident edges.

1 Introduction

Definitions: A computational problem with input size n and another input parameter k is fixed-parameter tractable (FPT) if it can be solved in $O(p(n) \cdot f(k))$ time where p is a polynomial and f any function. Since we focus on the $f(k)$ factor, we adopt the $O^*(f(k))$ notation that suppresses polynomial factors. (The polynomial factor cannot be neglected in practice, but usually it is moderate, so that $f(k)$ "dominates" the complexity.) For introductions to parameterized algorithms we refer to [6,13], in particular we assume that the reader is familiar with the notions of bounded search trees, branching vector, and branching number. For brevity, a branching rule is said to be a *bv rule* if its branching number is less than or equal to that of branching vector bv. We deal with graph problems and denote by n the number of vertices.

The following is a classical FPT problem, and a tremendous amount of work has been devoted to its parameterized complexity.

VERTEX COVER: In a graph, find a set C of at most k vertices being incident to all edges.

A hypergraph is a vertex set equipped with a family of subsets of vertices called hyperedges. The degree of a vertex is the number of hyperedges it belongs

to, and the degree of a hypergraph is the maximum vertex degree. We introduce the following problem.

UNION EDITING: Given a hypergraph whose vertices are colored red and blue, paint at most k blue vertices red, and paint at most m red vertices blue, so that the set of red vertices becomes exactly the union of some of the hyperedges. (Note that the union is not necessarily disjoint.)

UNION EDITING (DEGREE r) is, as the name suggests, the UNION EDITING problem on hypergraphs of degree at most r.

Motivation: The problem arises from chemical analytics. In fact, it was proposed in [11], however without algorithmic analysis. Every hyperedge represents a possible substance in an unknown mixture, and the vertices therein represent the reaction products that can be directly identified by experiments. For instance, unknown protein mixtures can be analyzed by splitting the proteins enzymatically into peptides which are then identified by, e.g., mass spectrometry. A database of proteins and their peptides is finally used to identify the proteins from the set of peptides. Many peptides, especially those with large masses, are unique for a protein, others appear in r different proteins, very often just $r = 2$. The problem to resolve these ambiguities [5,9,12] is just SET COVER or HITTING SET, as also observed by biologists. Case $r = 2$ is VERTEX COVER again.

However, here we do not address this inference step but another problem that appears prior to the actual mixture reconstruction: Ideally, the set of observed vertices should be exactly the union of some of the hyperedges (those representing the substances in the mixture). But in practice, observations may be corrupted in two ways: by at most k vertices that should appear but are not observed, and by at most m vertices that are observed but should not appear. (Our red vertices are the observed ones.) The question arises how we can efficiently correct a limited number of such errors, assuming that reliable bounds k and m are known from experience. Different problem versions may be considered: decide the existence of a solution with k and m errors, enumerate them all, or enumerate all vertices being recolored in some solution (the error candidates).

Contributions: UNION EDITING (DEGREE 1) is a trivial problem, in a sense: As the hyperedges are pairwise disjoint, we only have to decide for every hyperedge whether to color it completely red or blue. For each pair (k, m) we can solve the problem in polynomial time by dynamic programming, as it is essentially a knapsack problem with unary representations of numbers. We can even enumerate all possible solutions in an implicit way, as the system of all paths in a certain directed graph of partial solutions. All this is a straightforward exercise in dynamic programming, therefore we skip the details. In this paper we maily deal with UNION EDITING (DEGREE 2) which is the "smallest nontrivial case" but is already important in the intended application.

In Section 2 we introduce several equivalent formulations as (hyper)graph editing problems. We relate them to each other and to established problems. These problems are NP-complete but also in FPT, with the solution size as

the parameter. UNION EDITING (DEGREE 2) turns out to be a special case of the well-known 3-HITTING SET problem. In Section 3 we give a parameterized $O^*(1.9^{k+m})$ time algorithm for minimizing $k + m$ (corresponding to the total number of error corrections). This is clearly faster than the state-of-the-art result for 3-HITTING SET in general. The gain is not very high (e.g., about 10% larger instances can be solved in a given time budget, compared to the 3-HITTING SET algorithm in [15]), nevertheless the result indicates that our problem is also an interesting subcase from an algorithmic point of view. We can also compute, within the same time bound, a kernel that contains all optimal solutions.

Related work: The currently fastest algorithm [2] runs in $O^*(1.2738^k)$ time. Several VERTEX COVER variants and generalizations have been addressed in, e.g., [1,7,8,10,14], including vertex covers with additional constraints and partial vertex covers. One of our problem formulations is a case of partial vertex covers where, however, the number of uncovered edges is also limited by a parameter.

2 Some Equivalent Parameterized Graph Editing Problems

Consider the problem UNION EDITING (DEGREE 2). We wish to translate it into an equivalent graph problem. Since we are then on more familiar grounds, this will make it easier to characterize the complexity of the problem. Our graphs may contain loops and parallel edges. In a graph, we call the set of *all* edges incident to some vertex v the *star with center v*. (This should not be confused with other notions of "star" in graph theory.) Specifically we define:

STAR EDITING: In a graph with red and blue edges, recolor at most k blue edges and m red edges, so that the union of some stars becomes exactly the set of red edges, or equivalently, the edge set of some induced subgraph becomes exactly the set of blue edges.

Theorem 1. UNION EDITING (DEGREE 2) *and* STAR EDITING *are equivalent, through some polynomial-time reduction that preserves parameters k and m.*

Proof. Given any hypergraph H of degree 2, we construct a graph G as follows. For every hyperedge of H create a vertex of G. Every vertex of H that belongs to the intersection $u \cap v$ of hyperedges u and v becomes an edge uv in G. (Thus, parallel edges may appear.) Every vertex of H that belongs to only one hyperedge v becomes a loop at v in G. This defines a one-to-one correspondence between hypergraphs of degree 2 and graphs with loops and parallel edges, as we can also reconstruct H from G: Every edge of G becomes a vertex of H, and every star in G becomes a hyperedge in H. Obviously, the solutions of the two problems in H and G correspond to each other.   □

Alternatively we may view STAR EDITING as a vertex selection, rather than edge selection, problem. This is what we show next.

DEFICIENT VERTEX COVER WITH COST EDGES: In a graph with red and blue edges, find a subset C of vertices incident to at most k blue edges and non-incident to at most m red edges.

This problem extends the ordinary VERTEX COVER problem (in graphs with only red edges) in two directions: m red edges may remain uncovered by C, and the "cost" k of C is the number of incident blue edges. In the special case when all blue edges are loops, they merely encode an integer-valued cost function on the vertices. The new twist is that any two vertices u and v joined by a blue edge uv share one unit of cost, in the sense that only one of them, say u, pays for this blue edge when $u \in C$, while the cost of adding v to C is reduced by one. Because of this role of the blue edges we also call them cost edges. These pairwise dependencies of costs have no counterpart in the ordinary VERTEX COVER problem. Also note that, to any vertex cover C, we may add, at zero cost, those vertices incident only to blue edges being already incident with C.

Let VERTEX COVER WITH COST EDGES denote the special case when $m = 0$. We also remark that the other natural special case, DEFICIENT VERTEX COVER where all blue edges are loops, was also studied in [1], though in a weighted version and under a different name. Both FPT and W[]-hardness results are shown in [14] for another interesting VERTEX COVER generalization called VECTOR DOMINATING SET where the number of uncovered incident edges is individually prescribed for each vertex.

Theorem 2. *Problems* STAR EDITING *and* DEFICIENT VERTEX COVER WITH COST EDGES *are equivalent, via a polynomial-time reduction that preserves the parameters k and m.*

Proof. Consider a set C that solves DEFICIENT VERTEX COVER WITH COST EDGES for parameters k and m. By recoloring the blue edges incident to C and the red edges non-incident to C we get a solution to STAR EDITING for parameters k and m, where C is the set of centers of stars whose union consists of exactly the red edges. For the other direction, consider a solution to STAR EDITING where at most k blue and m red edges are recolored. Then, the set C of all centers of entirely red stars (after recoloring) solves DEFICIENT VERTEX COVER WITH COST EDGES.   □

VERTEX COVER WITH COST EDGES still appears as a proper generalization of VERTEX COVER, so the following reduction to VERTEX COVER might be a little surprising.

We define a *conflict triple* as a set of three edges: two blue edges uv and xy joined by a red edge vx, where $v \neq x$. (The other vertices are not necessarily distinct, i.e., we can have $u = y$, or the blue edges may be loops, or parallel to vx.) The red and blue degree of a vertex is the number of incident red and blue edges, respectively, where a loop counts only once.

Theorem 3. VERTEX COVER *and* VERTEX COVER WITH COST EDGES *are equivalent, through some polynomial-time reductions that preserve parameter k.*

Proof. The reduction from VERTEX COVER and VERTEX COVER WITH COST EDGES is trivial, as already mentioned: Attach to every vertex a blue loop that encodes the vertex cost.

Conversely, consider any instance of VERTEX COVER WITH COST EDGES, i.e., an edge-colored graph G and a parameter k. If a red loop is attached to some vertex v, clearly we must put v in the solution C, moreover we remove v and all incident edges and reduce k by the blue degree of v. If parallel red and blue edges join two vertices u and v then, since some of u and v must eventually be in C, we can immediately remove the blue edges uv and subtract their number from k. After these data reductions, G has neither red loops nor blue edges parallel to red edges.

Now we construct a graph G' as follows. Every blue edge of G becomes a vertex of G'. For any conflict triple uv, vx, xy, we create an edge between vertices uv and xy in G'. Thus, the same red edge vx may give rise to several edges of G'; we call them copies of vx.

Consider any vertex cover C of cost k in G. Let C' be the set of blue edges incident to C. Observe that C' has size k and is a vertex cover in G': For any red edge vx in G, we have $v \in C$ or $x \in C$, by symmetry assume $v \in C$. Hence all blue edges incident with v are in C'. It follows that all copies, in G', of the edge vx are covered by vertices from C'.

To see the opposite direction of the equivalence, consider any vertex cover C' of size k in G'. For every red edge vx from G, all blue edges at v or all blue edges at x must belong to C', in order to cover all copies of edge vx in G'. Define C as the set of all vertices v of G where all blue edges incident to v are in C'. Due to the sentence before, C is a vertex cover in G. The cost of C is the number of incident blue edges, which is at most k, as these edges were in C'.   □

A hitting set in a hypergraph is a subset of vertices that intersects every hyperedge. The rank of a hypergraph is the maximum size of hyperedges. We denote by r-HITTING SET the problem of finding a hitting set with at most k vertices in a hypergraph of rank r. Hence 2-HITTING SET is VERTEX COVER. We refine r-HITTING SET to a two-parameter problem:

s, r-HITTING SET: Given is a hypergraph whose vertices are colored red and blue, where every hyperedge consists of at most s blue and r red vertices. (We also say that the blue rank and red rank is s and r, respectively.) Find a hitting set C of at most k blue and m red vertices.

Theorem 3 essentially relies on $m = 0$. For $m > 0$ we do not see any parameter-preserving reduction from DEFICIENT VERTEX COVER WITH COST EDGES to VERTEX COVER. It seems that VERTEX COVER with both missed edges and cost edges is intrinsically more difficult. However we can reduce it to the next higher problem in the "hitting set hierarchy":

Theorem 4. DEFICIENT VERTEX COVER WITH COST EDGES, *is reducible to* 2,1-HITTING SET, *through some polynomial-time reduction that preserves the parameters k and m.*

Proof. Every edge of the given graph G becomes a vertex of a hypergraph H. The hyperedges of H are the following sets of size 2 or 3: every red loop with every incident blue edge; every red edge with every parallel blue edge; and every conflict triple. We call the first two cases conflict pairs.

Let C be any solution to DEFICIENT VERTEX COVER WITH COST EDGES in G. We claim that the set F consisting of all blue edges incident to C and all red edges non-incident to C is a hitting set in H. To prove the claim, consider any red edge vx. Assume that some end vertex is in C, say $v \in C$. If vx forms a conflict pair with some blue edge, then either $v = x$ (red loop), or vx and the blue edge are parallel. In both cases the blue edge is incident to C, thus F intersects the conflict pair. If vx forms a conflict triple with blue edges uv and xy, then the blue edge uv is incident to C, too. If $v, x \notin C$ then $vx \in F$. Note that F intersects every conflict pair/triple containing vx.

Conversely, let F be any hitting set of k blue edges and m red edges in H. We construct a vertex set C in G and show that C is incident to at most k blue edges and non-incident to at most m red edges. Let vx be any red edge with $vx \notin F$. If $v = x$, we put this vertex in C. Note that the incident blue edges are all in F, since F intersects all conflict pairs with the red loop vx. In the following let be $v \neq x$. All blue edges at v or all blue edges at x are in F, since otherwise some conflict triple with vx in the middle, or some conflict pair with a parallel blue edge vx, would be disjoint to F. We put v in C if all blue edges incident to v are in F, and similarly for x. Due to the construction of C, a blue edge is incident to C, and a red edge is non-incident to C, only if it belongs to F. $\square$

3   Solving Star Editing Faster than 3-Hitting Set

We defined STAR EDITING as a problem with two separate parameters k and m. Nevertheless, let us consider the simplest version of the problem where we only want to find some solution minimizing the total number $k + m$ of edits. In the application mentioned in Section 1 this means to find a minimum number of error corrections that may explain the data.

By Theorem 4 we can reduce this problem to 2,1-HITTING SET, and even further to 3-HITTING SET as we need not distinguish red and blue vertices as long as we only aim for a minimum $k + m$. Thus, we can find some solution using any parameterized algorithm for 3-HITTING SET, such as the $O^*(2.076^{k+m})$ time algorithm from [15]. (Slightly faster algorithms have been announced but are apparently unpublished.) However, the hypergraphs of rank 3 from our reduction have a special structure, thus we might be able to solve STAR EDITING significantly faster than 3-HITTING SET in general. In fact, we will now devise an $O^*(1.9^{k+m})$ algorithm.

We use the DEFICIENT VERTEX COVER WITH COST EDGES formulation of our problem, which is more convenient for stating the branching rules. Accordingly, we use C as a generic variable for a solution. We describe our algorithm as a list of rules, each of which is applied as long as possible before going to the next one.

(1) We will always remove every edge incident to some vertex just added to C, and if it was a blue edge, we subtract 1 from parameter k. If we decided not to put some vertex v in C, we turn every incident edges into a loop at its other end vertex. If both vertices of a red edge are not put in C, we remove this red edge and subtract 1 from parameter m. These trivial actions are usually not mentioned any more in the branching rules below.

(2) If two parallel red edges connect some vertices v and x, then removing one of them can only lower the parameter deductions in some branching rules, in those branches where $v, x \notin C$. Hence, in a worst-case analysis we can safely assume that no parallel red edges exist.

(3) In our input graph $G = (V, E)$ we can assume that all red and blue degrees are positive: If some vertex v has red degree 0, we can immediately decide $v \notin C$. A vertex v with blue degree 0 can be put in C at no cost.

(4) Denote by V_1, V_2, V_3 the set of vertices with blue degree 1, 2, and at least 3, respectively. Since by (3) all blue degrees are positive, we have $V = V_1 \cup V_2 \cup V_3$. If some red edge vx connects two vertices $v \in V_2$ and $x \in V_3$, we decide whether $v, x \notin C$ or $v \in C$ or $x \in C$. (The last two branches do not rule out insertion of the other vertex in C later on.) Obviously this gives us a $(1, 2, 3)$ rule, and its branching number evaluates to 1.84. A red edge within V_3 (either normal edge or loop) gives even better branching numbers, as is easy to check. We apply these rules as long as possible to obtain a graph with red edges only within $V_1 \cup V_2$ and between V_1 and V_3.

(5) Next we make a key observation relying on the fact that we are only to find some (arbitrary) solution with minimum total cost $k+m$ in this problem version. Consider any red edge vx with $v \in V_1$. If we decide $v, x \notin C$, this red edge costs 1. If we take $v \in C$ instead, we pay 1 for the blue edge incident to v, and this decision is no worse than the case $v, x \notin C$. This exchange argument shows that we can ignore the branch $v, x \notin C$. Due to this domination rule we can set the red edge vx permanent, that is, we commit to put v or x in C, but defer the actual choice of v or x. The same reasoning holds for red loops at any $v \in V_1$, but there we can decide $v \in C$ instantly. Now all red edges except those in V_2 are permanent and non-loops.

(6) As long as some red edge is incident with V_3, being permanent due to (5), we obviously have a $(1, 3)$ rule. By covering these red edges by new members of C and applying also (1), we eventually make $V_3 = \emptyset$. Similarly, as long as some (permanent!) red edges connect V_2 and V_1, we have a $(1, 2)$ branching rule, with branching number 1.62. After covering these red edges, too, our graph has blue degree at most 2, and red edges only within V_2 and within V_1, and the latter ones are permanent.

(7) If any two vertices $u, v \in V_2$ are joined by two parallel blue edges, we can assume that both u and v or none of them are in C, because if we put one

vertex in C, we can add the other vertex for free. Hence we can identify u and v, thereby turning the blue edges in two blue loops at $u = v$. Parallel red edges are already excluded by (2). (Alternatively we might argue as follows: If any two vertices $u, v \in V_2$ are joined by two parallel red edges, we have a $(2, 2, 2)$ rule with branching number 1.74.) This excludes parallel edges of equal color in V_2.

(8) Consider any $w \in V_2$ of red degree at least 2, and let wu and wv be red edges. Due to the structure already established, we have $u, v \in V_2$ and $u \neq v$. Now we can decide $w \in C$ with cost 2, or $w \notin C$. In the latter case we also make the decisions for u and v. For each of them the cost is either 2 for the incident blue edges (if we put the vertex in C), or 1 for the red edge (if we don't). In the worst case however, one blue edge uv exists, which reduces the parameter deduction by 1 in the branch where $u, v \in C$. Altogether, this still yields a $(2, 2, 3, 3, 3)$ rule with branching number 1.9. By applying also this rule as long as possible, all vertices in V_2 get red degree 1.

(9) As said before, we can suppose that no parallel red edges exist in V_1 (in particular). Next assume that some edge uv with $u, v \in V_1$ is blue. As earlier, if we put either of u and v in C, the other vertex can be added to C for free, hence we identify u and v and shrink uv to a blue loop.

(10) Consider any $w \in V_1$ of red degree at least 2, and let wu and wv be red edges. From the preceding rules we have that $u, v \in V_1$ and $u \neq v$, and these red edges are permanent. Hence we must take $w \in C$ or $u, v \in C$. This is a $(1, 2)$ rule, since no blue edge uv exists, due to (9).

To summarize the current situation: All vertices are in $V_1 \cup V_2$, all red degrees are 1 (the red edges build a matching), red edges exist only within V_2 and within V_1, all red edges within V_1 are permanent, and blue edges exist only within V_2, between V_2 and V_1, and as blue loops in V_1.

(11) Blue loops in V_1 are removed as follows. When some $v \in V_1$ has a blue loop attached, and vx is a red edge (note $x \in V_1$), we can safely decide $x \in C$, as the option $v \in C$ is only worse.

(12) Let vx be a red edge in V_1. Since vx is permanent, we have to put some vertex in C, say $v \in C$, the other case is symmetric. Vertex v is also involved in a blue edge uv, where $u \in V_2$. Vertex u in turn is incident to some red edge yu, where $y \in V_2$. Since we decided $v \in C$, the blue degree of u drops to 1, thus u moves to V_1, so that we can make uy permanent, using (5). Since we have to choose $u \in C$ or $y \in C$, this gives us a $(1, 2)$ rule. We argue similarly in the symmetric case starting with $x \in C$. This way we have appended a $(1, 2)$ rule to both branches of a $(1, 1)$ rule. Altogether this makes a $(2, 2, 3, 3)$ rule. Repeated application of this rule empties V_1.

After application of all the preceding rules, it remains a graph where every vertex has exactly red degree 1 and blue degree 2. At this point, an optimal solution consists in not adding any further vertices to C. The cost of this claimed

optimal solution is the number of the remaining red edges. We consider any different solution ($C \neq \emptyset$) and show that, in fact, it cannot be cheaper: Every vertex added to C reduces the cost of the red edges by at most 1 but has to pay for two blue edges. And trivially, at most two vertices in C can share the cost of a blue edge. Consequently the total cost has not improved. Since rule (8) is the worst, this finally shows:

Theorem 5. *A solution to* DEFICIENT VERTEX COVER WITH COST EDGES *(or* STAR EDITING*) with minimal $k + m$ can be found in $O^*(1.9^{k+m})$ time.* $\square$

The optimization problem only asks for *some* solution with minimum $k + m$. But it might not be unique, and in the error correction application one cannot simply assume that an arbitrary optimal solution explains the data correctly. It is more appropriate to return all possible minimum solutions, but if these are exponentially many, this raises new issues. A nice compromise is to return just *the set of all edges that are recolored* in solutions with minimum $k + m$, i.e., the potential errors.

Theorem 6. *All edges recolored in all solutions to an instance of* STAR EDITING *with minimum $k + m$ can be found in $O^*(1.9^{k+m})$ time.*

Proof. The idea is natural, only its correct implementation needs a little bit of care: Let $c = k + m$ be the minimum number of recolorings. For every edge $e = uv$ we test (from scratch) whether there is a solution where e is recolored, and hence $c - 1$ other edges are recolored. In the following we use the equivalent DEFICIENT VERTEX COVER WITH COST EDGES formulation.

If e is red, we color it blue, that is, we decide $u, v \notin C$, and apply the trivial postprocessing steps mentioned above in step (1). If e is blue, we color it red and mark it permanent. In both cases we solve the residual problem with parameter value $c - 1$. For the latter case, however, we have to extend the DEFICIENT VERTEX COVER WITH COST EDGES problem in yet another direction: We allow permanent red edges already in the input graph. Now we argue that this generalization can still be solved in $O^*(1.9^{k+m})$ time, using the algorithm from Theorem 5 with slight modifications. In all branchings we can abandon the branches with $u, v \notin C$, if there are some. Trivially, deleting some branches can only improve the branching numbers. Once we reach a graph where all vertices have red degree 1 and blue degree 2, and some red edge is permanent, we clearly have a $(2, 2)$ rule and can continue. All other situations are resolved as in Theorem 5. $\square$

The number of such edges recolored in all minimal solutions is at most quadratic in $k + m$, as can be shown by simple degree arguments. Since the problem generalizes VERTEX COVER, the worst-case lower bound is also quadratic, due to our result for VERTEX COVER in [4].

4   Open Questions

It would be desirable to improve the base 1.9 for STAR EDITING further. One may try and improve rule (8), or look for a completely different FPT algorithm

design technique. Also, our STAR EDITING result does *not* imply that we can find in $O^*(1.9^{k+m})$ time a solution where each of k and m separately respects some prescribed values: Rule (5) does not apply here, as it might be beneficial to pay for an uncovered red edge if k is "too small".

Our focus on degree 2 is, of course, a limitation, therefore it is worth looking at the general case. UNION EDITING (DEGREE s), for any fixed degree s, can be transformed into hypergraph editing problems analogously to the case $s = 2$, where *edge* is replaced with *hyperedge* of size at most s, and *vertex cover* is replaced with *hitting set*.

Then, our results raise some further questions for general s: Can we solve the corresponding optimization problem significantly faster than s, 1-HITTING SET and $(s + 1)$-HITTING SET, and even faster than $O^*(s^{k+m})$? Can we enumerate all solutions (compute the transversal hypergraph) faster than in these HITTING SET instances?

Furthermore: Is there a linear kernel for the optimization version of STAR EDITING (cf. [1])? What is the parameterized complexity of our problems when k and m are limited separately (as in the Pareto framework we proposed in [3])?

Acknowledgment

References

1. Bar-Yehuda, R., Hermelin, D., Rawitz, D.: An Extension of the Nemhauser-Trotter Theorem to Generalized Vertex Cover with Applications. SIAM J. Discr. Math. 24, 287–300 (2010)
2. Chen, J., Kanj, I.A., Xia, G.: Improved Upper Bounds for Vertex Cover. Theor. Comput. Sci. 411, 3736–3756 (2010)
3. Damaschke, P.: Pareto complexity of two-parameter FPT problems: A case study for partial vertex cover. In: Chen, J., Fomin, F. (eds.) IWPEC 2009. LNCS, vol. 5917, pp. 110–121. Springer, Heidelberg (2009)
4. Damaschke, P., Molokov, L.: The Union of Minimal Hitting Sets: Parameterized Combinatorial Bounds and Counting. J. Discr. Algor. 7, 391–401 (2009)
5. Dost, B., Bandeira, N., Li, X., Shen, Z., Briggs, S., Bafna, V.: Shared peptides in mass spectrometry based protein quantification. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 356–371. Springer, Heidelberg (2009)
6. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
7. Fernau, H., Manlove, D.: Vertex and Edge Covers with Clustering Properties: Complexity and Algorithms. J. Discr. Algor. 7, 149–167 (2009)
8. Guo, J., Niedermeier, R., Wernicke, S.: Parameterized Complexity of Vertex Cover Variants. Theory Comput. Syst. 41, 501–520 (2007)

9. He, Z., Yang, C., Yang, C., Qi, R.Z., Tam, J.P.M., Yu, W.: Optimization-based peptide mass fingerprinting for protein mixture identification. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 16–30. Springer, Heidelberg (2009)
10. Kneis, J., Langer, A., Rossmanith, P.: Improved upper bounds for partial vertex cover. In: Broersma, H., Erlebach, T., Friedetzky, T., Paulusma, D. (eds.) WG 2008. LNCS, vol. 5344, pp. 240–251. Springer, Heidelberg (2008)
11. Molokov, L.: Application of Combinatorial Methods to Protein Identification in Peptide Mass Fingerprinting. In: Int. Conf. on Knowledge Discovery and Info. Retrieval KDIR 2010, pp. 307–313. SciTePress (2010)
12. Nesvizhskii, A.I., Aebersold, R.: Interpretation of Shotgun Proteomic Data: The Protein Inference Problem. Mol. Cellular Proteomics 4, 1419–1440 (2005)
13. Niedermeier, R.: Invitation to Fixed-Parameter Algorithms. Oxford Lecture Series in Math. and its Appl. Oxford Univ. Press (2006)
14. Raman, V., Saurabh, S., Srihari, S.: Parameterized algorithms for generalized domination. In: Yang, B., Du, D.Z., Wang, C.A. (eds.) COCOA 2008. LNCS, vol. 5165, pp. 116–126. Springer, Heidelberg (2008)
15. Wahlström, M.: Algorithms, Measures, and Upper Bounds for Satisfiability and Related Problems. PhD Thesis 1079, Linköping Studies in Science and Technol. (2007)

Path Minima Queries in
Dynamic Weighted Trees

Gerth Stølting Brodal[1], Pooya Davoodi[1], and S. Srinivasa Rao[2]

[1] MADALGO[★], Department of Computer Science, Aarhus University
{gerth,pdavoodi}@cs.au.dk
[2] School of Computer Science and Engineering, Seoul National University, S. Korea
ssrao@cse.snu.ac.kr

Abstract. In the path minima problem on trees each tree edge is assigned a weight and a query asks for the edge with minimum weight on a path between two nodes. For the dynamic version of the problem on a tree, where the edge-weights can be updated, we give comparison-based and RAM data structures that achieve optimal query time. These structures support inserting a node on an edge, inserting a leaf, and contracting edges. When only insertion and deletion of leaves in a tree are needed, we give two data structures that achieve optimal and significantly lower query times than when updating the edge-weights is allowed. One is a semigroup structure for which the edge-weights are from an arbitrary semigroup and queries ask for the semigroup-sum of the edge-weights on a given path. For the other structure the edge-weights are given in the word RAM. We complement these upper bounds with lower bounds for different variants of the problem.

1 Introduction

We study variants of the path minima problem on weighted unrooted trees, where each edge is associated with a weight. The problem is to maintain a data structure for a collection of trees (a forest) supporting the query operation:

- pathmin(u,v): return the edge with minimum weight on the path between two given nodes u and v.

In the dynamic setting, a subset of the following update operations is provided:

- make-tree(v): make a single-node tree containing the node v.
- update(e,w): change the weight of the edge e to w.
- insert(e,v,w): split the edge $e = (u_1, u_2)$ by inserting the node v along it. The new edge (u_1, v) has weight w, and (u_2, v) has the old weight of e.
- insert-leaf(u,v,w): add the node v and the edge (u, v) with weight w.
- contract(e): delete the edge $e = (u, v)$, and contract u and v to one node.
- delete-leaf(v): delete both the leaf v and the edge incident to it.

[★] Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

- link(u,v,w): add the edge (u, v) with weight w to the forest, where u and v are in two different trees.
- cut(e): delete the edge e from the forest, splitting a tree into two trees.

We present path minima data structures to maintain trees under either updating leaf edges or updating arbitrary edge-weights. Additionally, we study the complexity of path minima queries on forests that can be updated by link and cut.

We define three different models depending on how data the algorithms can operate on the edge-weights: (1) the *comparison-based* model, where the only allowed operation on the edge-weights are comparisons; (2) the *RAM* model, where any standard RAM operation is allowed on the edge-weights; and (3) the *semigroup* model, in which the edge-weights are from an arbitrary semigroup and queries ask to compute the semigroup-sum of the edge-weights along a path (notice that a data structure over the semigroup $(\mathbb{R}, \max)$ can be used to make a comparison-based structure for the path minima problem). Except for the computation on the edge-weights, our algorithms are in the unit-cost word RAM with word size $\Theta(\log n)$ bits, where n is the total number of nodes in the input trees.

1.1 Previous Work

Static weighted trees. The minimum spanning tree verification problem on a spanning tree of a graph can be solved by asking a sequence of offline path minima queries in the tree, which can be supported using amortized $O(1)$ comparisons for each query [17]. In the online setting, Pettie proved that $\Omega(\alpha(n))$ amortized comparisons are required to answer each query [21], which is a tight lower bound due to [8].

For online queries, Alon and Sheiber [1] considered trade-offs between the preprocessing time and the query time in the semigroup model. They presented a data structure that supports queries in at most $4k - 1$ semigroup operations with $O(nk\alpha_k(n))$ preprocessing time and space for a parameter $k \geq 1$, where $\alpha_k(n)$ is a function in the inverse-Ackermann hierarchy (defined in Section 1.3). They also proved that $\Omega(n\alpha_{2k}(n))$ preprocessing time is required to obtain such a query time in the worst case. Notice that $\alpha_{2k}(n) = o(\alpha_k(n))$ for $k > 1$ (Section 1.3). Similar trade-offs are known in the comparison-based model [21].

Dynamic weighted trees. In the comparison-based model, Demaine, Landau, and Weimann [10] showed how to maintain an input tree under inserting and deleting leaves in $O(\log n)$ amortized time and supporting queries in $O(1)$ time using linear space. In the RAM model, $O(1)$ amortized update time with the same query time and space can be achieved [2,15].

Tarjan was interested in maintaining a collection of rooted trees under the operation link (incremental trees) to support a class of path minima queries, where a root is one of the end points of the query paths [23, Section 6]. In the semigroup model, for a sequence of m offline queries and updates, he obtained $O((m + n) \cdot \alpha(m + n, n))$ time using $O(m + n)$ space. In the RAM model, this running time can be improved to $O(1)$ for each online query and amortized $O(1)$ for each offline update [14]. Alstrup and Holm showed that arbitrary queries can

be supported in $O(\alpha(n))$ time while online updates are performed in constant amortized time in the RAM model [2]. Finally, Kaplan and Shafrir generalized this result to arbitrary links in unrooted trees [15].

Dynamic trees (link-cut trees) of Sleator and Tarjan support many operations including pathmin, link, cut, root, and evert in $O(\log n)$ amortized time, in the semigroup model [22]. The operation root finds the root of the tree containing a given node, and evert changes the root of the tree containing a given node such that the given node becomes the root, by turning the tree "inside out". Essentially, this data structure can solve all the variants of the dynamic path minima problem.

Relation to range minimum queries. A special case of the path minima problem, is the one-dimensional *range minimum query* (1D-RMQ) problem, where the input tree is a path. In this problem, we are given an array containing n elements, and we have to find the position of the minimum element within a query range.

The following lower bounds are derived from known lower bounds by reduction from the 1D-RMQ problem: (1) In the semigroup model, with linear space, $\Omega(\alpha(n))$ semigroup operations are required to answer a path minima query [25]; (2) In the RAM model, using $O(n/c)$ bits of additional space, the number of cell probes required to answer a path minima query is at least $\Omega(c)$, for a parameter $1 \leq c \leq n$ [7] (here, we assume that the edge-weights are given in a read-only array); (3) In the RAM model, if we want to update the edge-weights in polylogarithmic time, $\Omega(\log n / \log \log n)$ cell probes are required to support path minima queries [3, Section 2.2]; (4) In the comparison-based model, if we want to answer a path minima query in polylogarithmic time, $\Omega(\log n)$ comparisons are required to update the edge-weights [6]; (5) In the semigroup model, logarithmic time to support path minima queries implies $\Omega(\log n)$ time to update the edge-weights, and vice versa [20].

Cartesian trees [24] are a standard structure that along with lowest common ancestor structures can support range minimum queries in constant time with linear space and preprocessing time [13]. Cartesian trees can be also defined for weighted trees as follows: The Cartesian tree T_C of a weighted tree T is a binary tree, where the root of T_C corresponds to the edge e of T with minimum weight, and the two children of the root in T_C correspond to the Cartesian trees of the two components made by deleting e from T. The internal nodes of T_C are the edges of T, and the leaves of T_C are the nodes of T.

Similar to range minimum queries, Cartesian trees can be used to support path minima queries in $O(1)$ time using linear space [18, Section 3.3.2], [5, Section 2], and [10, Theorem 2]. But constructing a Cartesian tree requires $\Omega(n \log n)$ comparisons derived by a reduction from *sorting* [10]. This lower bound implies a logarithmic lower bound to maintain the Cartesian tree under inserting new leaves in the original tree, which is tight due to the following lemma.

Lemma 1. ([10]) *The Cartesian tree of a tree with n nodes can be maintained in a linear space data structure that can be constructed in $O(n \log n)$ time, and supports path minima queries in $O(1)$ time and inserting leaves in $O(\log n)$ time.*

1.2   Our Results

In Section 2, we present a comparison-based data structure that supports path minima queries in $\Theta(\log n / \log \log n)$ time, and supports updates to the edge-weights in $\Theta(\log n)$ amortized time. In the RAM model, the update time is improved to $O(\log n / \log \log n)$ amortized. Both data structures support the operations insert, insert-leaf, and contract with the same update times.

In Section 3, we dynamize the data structure of Alon and Shieber [1] in the semigroup model, to support path minima queries in at most $7k - 4$ semigroup operations using $O(n\alpha_k(n))$ space, while supporting insertions and deletions of leaves in $O(\alpha_k(n))$ amortized time. Using Lemma 1, we can obtain a RAM structure that supports path minima queries in constant time and inserting and deleting leaves in constant amortized time, giving an alternative approach to achieve a known result [2,15].

In Section 4, we provide cell probe lower bounds for query-update trade-offs when data structures are served by the operations pathmin, link and cut, in the RAM model. We prove that if we want polylogarithmic update time, we cannot hope for answering path minima queries in faster than $\Omega(\log n / \log \log n)$ time. We also show that with logarithmic update time, $\Theta(\log n)$ query time achieved by the dynamic trees of Sleator and Tarjan is the best possible. Furthermore, we prove that with sub-logarithmic query time, obtaining logarithmic update time is impossible.

1.3   Preliminaries

In Sections 2 and 3, we design our data structures for rooted trees, though every unrooted tree can be transformed to a rooted tree by choosing an arbitrary node as the root. Notice that all the update operations except link play the same role on rooted trees, whereas link in rooted trees is restricted to add new edges between a root and another node. Moreover, we transform rooted trees to binary trees using a standard transformation [11]: Each node u with d children is represented by a path with $\max\{1, d\}$ nodes connected by $+\infty$ weighted edges. Each child of u becomes the left child of one of the nodes. Then, the operations in rooted trees translate to a constant number of operations in binary trees.

Since we make our data structures for rooted trees, we can divided each path minima query into two subqueries as follows. Every pathmin(u, v) is reduced to two subqueries pathmin(c, u) and pathmin(c, v), where c is the lowest common ancestor (LCA) of u and v. It is possible to maintain a tree under inserting leaves and internal nodes, deleting leaves and internal nodes with one child, and determining the LCA of any two nodes all in worst-case $O(1)$ time [9]. Therefore, we only consider queries pathmin(u, v), where u is an ancestor of v.

In our data structures, we utilize a standard decomposition of binary trees denoted by *micro-macro decompositions* [4]. Given a binary tree T with n nodes and a parameter x, where $1 \leq x \leq n$, the set of nodes in T is decomposed into $O(n/x)$ disjoint subsets, each of size at most x, where each subset is a connected subtree of T called a micro tree. Furthermore, the division is constructed such

that at most two nodes in a micro tree are adjacent to nodes in other micro trees. These nodes are denoted by *boundary nodes*. The root of every micro tree is a boundary node except for the micro tree that contains the root of T. The macro tree is a tree of size $O(n/x)$ consisting of all the boundary nodes, such that it contains an edge between two nodes if either they are in the same micro tree or there is an edge between them in T.

We use a variant of the inverse-Ackermann function α defined in [10,19]. First, we define the inverse-Ackermann hierarchy for integers $n \geq 1$: $\alpha_0(n) = \lceil n/2 \rceil$, $\alpha_k(1) = 0$, and $\alpha_k(n) = 1 + \alpha_k(\alpha_{k-1}(n))$, for $k \geq 1$. Note that $\alpha_1(n) = \log n$, $\alpha_2(n) = \log^* n$, and $\alpha_3(n) = \log^{**} n$. Indeed for $k \geq 2$, $\alpha_k(n) = \log^{**\cdots*} n$, where the $*$ is repeated $k-1$ times in the superscript. In other words, $\alpha_k(n) = \min\{j \mid \alpha_{k-1}^{(j)}(n) \leq 1\}$, where $\alpha_k^{(1)}(n) = \alpha_k(n)$, and $\alpha_k^{(j)}(n) = \alpha_k(\alpha_k^{(j-1)}(n))$ for $j \geq 2$. The inverse-Ackermann function is defined as: $\alpha(n) = \min\{k \mid \alpha_k(n) \leq 3\}$. The two-parameter version of the inverse-Ackermann function for integers $m, n \geq 1$ is defined as follows: $\alpha(m, n) = \min\{k : \alpha_k(n) \leq 3 + m/n\}$. This definition of the function satisfies $\alpha(m, n) \leq \alpha(n)$ for every m and n.

2   Data Structures for Dynamic Weights

In this section, we present two path minima data structures that support all the update operations except link and cut in an input tree. The first data structure is in the comparison-based model and achieves $\Theta(\log n / \log \log n)$ query time, $\Theta(\log n)$ time for update, and $O(\log n)$ amortized time for insert, insert-leaf, and contract. The second data structure improves the update time to $O(\log n / \log \log n)$ in the RAM model. Both the structures are similar to the ones in [15]. In the following, we first describe the comparison-based structure, and then we explain how to convert it to the RAM structure.

2.1   Comparison-Based Data Structure

We decompose the input binary tree T into micro trees of size $O(\log^\varepsilon n)$ with the maximum limit $3 \log^\varepsilon n$, for some constant ε, where $0 < \varepsilon < 1$, using the micro-macro decomposition (Section 1.2). In our data structure, we do not use macro trees. Each micro tree contracts to a super-node, and a new tree T' is built containing these super-nodes. If there is an edge in T between two micro trees, then there is an edge in T' between their corresponding super-nodes. The weight of this edge is the minimum weight along the path between the root of the child micro tree and the root of the parent micro tree. We binarize T', and then we recursively decompose it into micro trees of the same size $O(\log^\varepsilon n)$.

Consider a path minima query between the nodes u and v, where u is an ancestor of v. If u and v do not lie within the same micro tree, the query is divided into at most four subqueries of three different types: (1) an edge that is between two micro trees; (2) a query that is within a micro tree; and (3) a query that is between the root of the micro tree containing v and a boundary node of the micro tree containing u. Queries of type 1 are trivial, since we store the

edge-weights in all the levels of the decomposition. To support queries of type 2 efficiently, we precompute the answer of all possible queries within all possible micro trees. Queries of type 3 are divided into subqueries recursively. There are at most one subquery of type 3 at each level, and thus the overall query time is determined by the number of levels.

Updating edge-weights and insertions are performed in the appropriate micro tree in the first level, and if it is required we propagate them to the next levels recursively. To support updates within each micro tree, we precompute the result of all possible updates within all possible micro trees. We maintain the edge-weights of each micro tree in sorted order in a balanced binary search tree that supports insertions and deletions of new edge-weights in $O(\log(\log^\varepsilon n))$ time. Additionally, we assign a local ID to each node within a micro tree, which is the insertion time of the node.

We set the size of each micro tree in all the levels to $O(\log^\varepsilon n)$, thus the number of levels is $O(\log n / \log \log n)$.

Data structure. Let T_0 denote an input tree, and for $i \geq 1$, let T_i be the tree that is built of super-nodes (to which micro trees contract) in the level i of the decomposition. The data structure consists of the following parts:

- We explicitly store T_i in all the levels of the decomposition, including T_0.
- For each node in T_i, we store a pointer to the micro tree in T_i that contains the node. We also store the local ID of the node.
- We represent each micro tree μ with the tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ of size $o(\log n)$ bits, where s_μ, p_μ, and r_μ are arrays defined as follows. The array s_μ is the binary encoding of the topology of μ. The array p_μ maintains the local IDs of the nodes within μ, and enables us to find a given node inside μ. The array r_μ maintains the rank of the edge-weights according to the preorder traversal of μ.
- For each micro tree μ, we store a balanced binary search tree containing all the edge-weights of μ. This allows us to find the rank of a new edge-weight within μ in $O(\log(\log^\varepsilon n))$ time.
- For each micro tree μ in T_i, we store an array of pointers that point to the original nodes in T_i given the local IDs.

Precomputed tables. We precompute and store in a table all possible results of performing each of the following operations within all possible micro trees: pathmin, update, insert, insert-leaf, contract, LCA, root and child-ancestor. For a micro tree μ, root returns the local ID of the root of μ, LCA returns the local ID of the LCA of two given nodes in μ, and child-ancestor(u, v) returns the local ID of the child of u that is also an ancestor of v (if such a child does not exist, returns null). Each precomputed table is indexed by the tuples $(s_\mu, p_\mu, r_\mu, |\mu|)$ and the arguments of the corresponding operation. To perform update, insert, and insert-leaf within μ, we find the rank of the new edge-weight among the existing edge-weights of μ using its balanced binary search tree in $O(\log |\mu|) = O(\log \log n)$ time. This rank becomes an index for the corresponding tables. Using appropriate tables, we can achieve the following lemma.

Lemma 2. *Within a micro tree of size $O(\log^\varepsilon n)$, we can support* pathmin, LCA, root, child-ancestor, *and moving a subtree inside the tree in $O(1)$ time. The operations* update, insert, insert-leaf, *and* contract *can be supported in $O(\log \log n)$ time using the balanced binary search tree of the micro tree and precomputed tables of total size $o(n)$ bits that can be constructed in $o(n)$ time.*

Proof. Let μ be the micro tree. In the table used to perform pathmin, each entry is a pointer to an edge of μ which can be stored using $O(\log \log n)$ bits. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, and (ii) two indexes in the range $[1 \cdots |\mu|]$ which represent two query nodes. The number of different arrays s_μ is $2^{|\mu|}$. The number of different arrays p_μ and r_μ is $O(|\mu|!)$. Therefore, the table is stored in $O(2^{|\mu|} \cdot |\mu|! \cdot |\mu|^3 \cdot \log |\mu|) = o(n)$ bits.

In the table used for update, each entry is an array r_μ which maintains the rank of the edge-weights of μ after updating an edge-weight. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, (ii) an index in the range $[1 \cdots |\mu|]$ which represents an edge to be updated, and (iii) the rank of the new edge-weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot |\mu|! \cdot |\mu|^4 \cdot \log |\mu|) = o(n)$ bits.

In the table used for insert-leaf, each entry is the tuple $(s_\mu, p_\mu, r_\mu, |\mu|)$ which represents μ after adding the new leaf. The index to the table consists of (i) $(s_\mu, p_\mu, r_\mu, |\mu|)$, (ii) an index in the range $[1 \cdots |\mu|]$ which represents the node that a new leaf is adjacent to, and (iii) the rank of the new edge-weight. Therefore, the table can be stored in $O(2^{|\mu|} \cdot |\mu|! \cdot |\mu|^4 \cdot \log |\mu|) = o(n)$ bits.

The sizes of the other tables used for LCA, root, child-ancestor, moving a subtree, insert, and contract are analyzed similarly. Since the total number of entries in all the tables is less than $o(2^{|\mu|^2})$ and each entry can be computed in time $O(|\mu|)$, all the tables can be constructed in $o(n)$ time.

To compute the rank of the new edge-weight, which is part of the index to the tables for updates, we search and then update the balanced binary search tree of μ in $O(\log \log n)$ time.   □

Query time. Each path minima query is divided into subqueries, and at most one subquery is divided recursively into subqueries. The division continues for $O(\log n / \log \log n)$ levels, and at each level all subqueries (of types 1 and 2) are supported in constant time. Therefore, the query time is $O(\log n / \log \log n)$.

Update. We perform update(e, w) by updating the data structure in all the ℓ levels. W.l.o.g. assume that $e = (u, v)$, where u is the parent of v. Let μ be the micro tree in T_0 that contains v. We start to update from the first level, where the tree is T: (1) Update the weight of e in T. (2) If v is not the root of μ, then we update μ using Lemma 2. If v is the root of μ, i.e., e connects μ to its parent micro tree, we do not need to update any micro tree. (3) Perform check-update(μ) which recursively updates the edge-weights in T_1 between μ and its child micro trees as follows. We check if pathmin along the path between the root of μ and the root of each child micro tree of μ needs to be updated. We can check this using pathmin within μ. If this is the case, for each one, we go to the next level and perform the three-step procedure on T_1 recursively. Since each micro tree has at most one boundary node that is not the root, then at most

one of the child micro trees of μ can propagate the update to the next level, and therefore the number of updates does not grow exponentially. Step 2 takes $O(\log \log n)$ time, and thus update takes totally $O(\log n)$ time in the worst case.

Insertion. We perform insert(e, v, w) using a three-step procedure similar to update. Let μ be the micro tree in T that contains u_2, where $e = (u_1, u_2)$ and u_1 is the parent of u_2. We start from the first level, where the tree is T: (1) To handle insert in the transformed binary tree, we first insert v along e in μ. Note that if u_2 is the root of μ, then v is inserted as the new root of μ. This can be done in $O(\log \log n)$ time using Lemma 2. (2) If $|\mu|$ exceeds the maximum limit $3 \log^\varepsilon n$, then we split μ, in linear time, into $k \le 4$ new micro trees, each of size at most $2 \log^\varepsilon n + 1$ such that each new micro tree has at most two boundary nodes including the old boundary nodes of μ. These k micro trees are contracted to nodes that should be in T_1. One of the new micro trees that contains the root of μ corresponds to the node that is already in T_1 for μ. The other $k-1$ new micro trees are contracted and inserted into T_1 with appropriate edge-weights, using insert recursively. Let μ' be the new micro tree that contains the boundary node of μ which is not the root of μ. We perform check-update(μ') to recursively update the edge-weights in T_1 between μ' and its child micro trees. (3) Otherwise, i.e., if $|\mu|$ does not exceed the maximum limit, we do check-update(μ) to recursively update the edge-weights in T_1 between μ and its child micro trees, which takes $O(\log n)$ time.

To perform insert-leaf(u, v, w), we use the algorithm of insert with the following changes. In step (1), we insert v as a child of u. This can be done in $O(\log \log n)$ time. The step (3) is not required.

A sequence of n insertions into T_0, can at most create $O(n/ \log^\varepsilon n)$ micro trees (since any created micro tree needs at least $\log^\varepsilon n$ node insertions before it splits again). Since the number of nodes in $T_0, T_1, \ldots, T_\ell$ is geometrically decreasing, the total number of micro tree splits is $O(n/ \log^\varepsilon n)$. Because each micro tree split takes $O(\log^\varepsilon n)$ time, the amortized time per insertion is $O(1)$ for handling micro tree splits. Thus, both insert and insert-leaf can be performed in $O(\log n)$ amortized time.

Edge contraction. We perform contract(e) by marking v as contracted and updating the weight of e to ∞ by performing update. When the number of marked edges exceeds half of all the edges, we build the whole structure from scratch using insert-leaf for the nodes that are not marked and the edges that do not have weight of ∞. Thus, the amortized deletion time is the same as insertion time.

Theorem 1. *There exists a dynamic path minima data structure for an input tree of n nodes in the comparison-based model, supporting* pathmin *in* $O(\log n/ \log \log n)$ *time,* update *in* $O(\log n)$ *time,* insert*,* insert-leaf*, and* contract *in* $O(\log n/ \log \log n)$ *amortized time using* $O(n)$ *space.*

2.2   RAM Structure

In this section, we improve the update time of the the structure of Theorem 1 to $O(\log n/ \log \log n)$ in the RAM model. The bottleneck in our comparison-based

data structure is that we maintain a balanced binary search tree for the edge-weights of each micro tree to find the rank of new edge-weights in $O(\log \log n)$ time. We improve this by using a Q-heap structure [12] to maintain the edge-weights of each micro tree to find the rank of new edge-weights under insertions and deletions in $O(1)$ time with linear space and preprocessing time. The following theorem states our result.

Theorem 2. *There exists a dynamic path minima data structure for an input tree of n nodes in the RAM model, which supports* pathmin *and* update *in $O(\log n / \log \log n)$ time, and* insert*,* insert-leaf *and* contract *in $O(\log n / \log \log n)$ amortized time using $O(n)$ space.*

3   Data Structures for Dynamic Leaves

In this section, we first present a semigroup structure that supports path minima queries, and leaf insertions/deletions but no updates to edge-weights. We then describe a RAM structure supporting the same operations.

3.1   Optimal Semigroup Structure

Alon and Schieber [1] presented two static data structures to support path minima queries in the semigroup model. We observe that their structures can be made dynamic. The following theorems summarize our result. To prove this we need an additional restricted operation insert(e,v,w), where w is larger than the weight of e.

Theorem 3. *There exists a semigroup data structure of size $O(n\alpha_k(n))$ to maintain a tree containing n nodes, that supports path minima queries with at most $7k - 4$ semigroup operations and leaf insertions/deletions in $O(\alpha_k(n))$ amortized time, for a parameter k, where $1 \leq k \leq \alpha(n)$.*

By substituting k with $\alpha(n)$, we obtain the following.

Corollary 1. *There exists a path minima data structure in the semigroup model using $O(n)$ space, that supports* pathmin *in $O(\alpha(n))$ time,* insert-leaf *and* delete-leaf *in amortized $O(1)$ time.*

3.2   RAM Structure

We also present a RAM structure that supports path minima queries and leaf insertions/deletions. This structure does not give a new result (due to [2,15]) but is a another approach to solve the problem.

We decompose a tree into micro trees of size $O(\log n)$ and each micro tree into micro-micro trees of size $O(\log \log n)$ using the micro-macro decomposition (see Section 1.3). Decomposition of the tree into micro trees generates a macro-macro tree of size $O(n/\log n)$, and decomposition of each micro tree into micro-micro trees generates $O(n/\log n)$ macro trees, each of size $O(\log n/\log \log n)$.

The operations within each micro-micro tree is supported using precomputed tables and Q-heaps [12]. We do not store any representation for the micro trees. We represent the macro-macro tree and each macro tree with a Cartesian tree.

The query can be solved in $O(1)$ time by dividing it according to the three levels of the decomposition. A new leaf is inserted into the appropriate micro-micro tree. When the size of a micro-micro tree exceeds its maximum limit, we split it, and insert the new boundary nodes into the appropriate macro tree, and split this macro if exceeds its maximum limit. Our main observation is the following.

Lemma 3. *When a micro tree splits, we can insert the new boundary nodes by performing* insert-leaf *using the Cartesian tree of the corresponding macro tree.*

We represent each Cartesian tree using Lemma 1. Thus, Lemma 3 allows us to achieve the following.

Theorem 4. *There exists a dynamic path minima data structure for an input tree of n nodes using $O(n)$ space that supports* pathmin *in $O(1)$ time, and supports* insert-leaf *and* delete-leaf *in amortized $O(1)$ time.*

4   Lower Bounds

We consider the path minima problem with the update operations link and cut. Let t_q denote the query time, and t_u denote the maximum of the running time of link, and cut. In the cell probe model, we prove that if we want to support link and cut in a time within a constant factor of the query time, then $t_q = \Omega(\log n)$. Moreover, if we want a fast query time $t_q = o(\log n)$, then one of link or cut cannot be supported in $O(\log n)$ time, e.g., if $t_q = O(\log n/\log\log n)$, then $t_u = \Omega(\log^{1+\varepsilon} n)$ for some $\varepsilon > 0$. We also show that $O(\log n/\log\log n)$ query time is the best achievable for polylogarithmic update time, e.g., a faster query time $O(\log n/(\log\log n)^2)$ causes t_u to blow-up to $(\log n)^{\Omega(\log\log n)}$.

We reduce the *fully dynamic connectivity* and *boolean union-find* problems to the path minima problem with link and cut.

The fully dynamic connectivity problem on forests is to maintain a forest of undirected trees under the three operations connect, link, and cut, where connect(x,y) returns true if there exists a path between the nodes x and y, and returns false otherwise. Let t_{con} be the running time of connect, and t_{update} be the maximum of the running times of link and cut. Pătraşcu and Demaine [20] proved the lower bound $t_{con}\log(2 + t_{update}/t_{con}) = \Omega(\log n)$ in the cell probe model. This problem is reduced to the path minima by putting a dummy root r on top of the forest, and connect r to an arbitrary node of each tree with an edge of weight $-\infty$. Thus the forest becomes a tree. For this tree, we construct a path minima data structure. The answer to connect(x,y) is false iff the answer to pathmin(x,y) is an edge of weight $-\infty$. To perform link(x,y), we first run pathmin(x,r) to find the edge e of weight $-\infty$ on the path from r to x. Then we remove e and insert the edge (x,y). To perform cut(x,y), we first run

pathmin(x,r) to find the edge e of weight $-\infty$. Then we change the weight of e to zero, and the weight of (x, y) to $-\infty$. Now, by performing pathmin(x,r), we figure out that x is connected to r through y, or y is connected to r through x. W.l.o.g. assume that x is connected to r through y. Therefore, we delete the edge (x, y), insert (x, r) with weight $-\infty$, and change the weight of e back to $-\infty$. Thus, we obtain the trade-off $t_q \log \frac{t_q+t_u}{t_q} = \Omega(\log n)$. From this, we e.g., conclude that if $t_q = O(\log n / \log \log n)$, then $t_u = \Omega(\log^{1+\varepsilon} n)$, for some $\varepsilon > 0$. We can also show that if $t_u = O(t_q)$, then $t_q = \Omega(\log n)$.

The boolean union-find problem is to maintain a collection of disjoint sets under the operations: find(x,A): returns true if $x \in A$, and returns false otherwise; union(A,B): returns a new set containing the union of the disjoint sets A and B. Kaplan et al. [16] proved the trade-off $t_{\text{find}} = \Omega(\frac{\log n}{\log t_{\text{union}}})$ for this problem in the cell probe model, where t_{find} and t_{union} are the running time of find and union. The incremental connectivity problem is the fully dynamic connectivity problem without the operation cut. The boolean union-find problem is trivially reduced to the incremental connectivity problem. The incremental connectivity problem is reduced to the path minima problem with the same reduction used above. Therefore, we obtain $t_q = \Omega(\frac{\log n}{\log(t_q+t_u)})$. We can conclude that when $t_q = O(\log n/(\log \log n)^2)$, slightly less than $O(\log n / \log \log n)$, then the running time of t_u blows-up to $(\log n)^{\Omega(\log \log n)}$.

References

1. Alon, N., Schieber, B.: Optimal preprocessing for answering on-line product queries. Technical report, Department of Computer Science, School of Mathematical Sciences, Tel Aviv University (1987)
2. Alstrup, S., Holm, J.: Improved algorithms for finding level ancestors in dynamic trees. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 73–84. Springer, Heidelberg (2000)
3. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proc. 39th Annual Symposium on Foundations of Computer Science, p. 534. IEEE Computer Society, Washington, DC, USA (1998)
4. Alstrup, S., Secher, J., Spork, M.: Optimal on-line decremental connectivity in trees. Information Processing Letters 64(4), 161–164 (1997)
5. Bose, P., Maheshwari, A., Narasimhan, G., Smid, M., Zeh, N.: Approximating geometric bottleneck shortest paths. Computational Geometry 29(3), 233–249 (2004)
6. Brodal, G.S., Chaudhuri, S., Radhakrishnan, J.: The randomized complexity of maintaining the minimum. Nordic Journal of Computing 3(4), 337–351 (1996)
7. Brodal, G.S., Davoodi, P., Rao, S.S.: On space efficient two dimensional range minimum data structures. Algorithmica, Special issue on ESA 2010 (2011) (in press)
8. Chazelle, B.: Computing on a free tree via complexity-preserving mappings. Algorithmica 2, 337–361 (1987)
9. Cole, R., Hariharan, R.: Dynamic lca queries on trees. SIAM Journal on Computing 34(4), 894–923 (2005)

10. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 341–353. Springer, Heidelberg (2009)
11. Frederickson, G.N.: Data structures for on-line updating of minimum spanning trees, with applications. SIAM Journal on Computing 14(4), 781–798 (1985)
12. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. Journal of Computer and System Sciences 48(3), 533–551 (1994)
13. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th Annual ACM Symposium on Theory of Computing, pp. 135–143. ACM Press, New York (1984)
14. Harel, D.: A linear time algorithm for finding dominators in flow graphs and related problems. In: Proc. 17th Annual ACM Symposium on Theory of Computing, pp. 185–194. ACM Press, New York (1985)
15. Kaplan, H., Shafrir, N.: Path minima in incremental unrooted trees. In: Halperin, D., Mehlhorn, K. (eds.) Esa 2008. LNCS, vol. 5193, pp. 565–576. Springer, Heidelberg (2008)
16. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and boolean union-find. In: Proc. 34th Annual ACM Symposium on Theory of Computing, pp. 573–582. ACM Press, New York (2002)
17. King, V.: A simpler minimum spanning tree verification algorithm. Algorithmica 18(2), 263–270 (1997)
18. Neto, D.M.: Efficient cluster compensation for lin-kernighan heuristics. PhD thesis. University of Toronto, Toronto, Ontario, Canada (1999)
19. Nivasch, G.: Inverse ackermann without pain (2009),
    http://www.yucs.org/~gnivasch/alpha/
20. Pătraşcu, M., Demaine, E.D.: Logarithmic lower bounds in the cell-probe model. SIAM Journal on Computing 35(4), 932–963 (2006)
21. Pettie, S.: An inverse-ackermann type lower bound for online minimum spanning tree verification. Combinatorica 26(2), 207–230 (2006)
22. Sleator, D., Endre Tarjan, R.: A data structure for dynamic trees. Journal of Computer and System Sciences 26(3), 362–391 (1983)
23. Tarjan, R.E.: Applications of path compression on balanced trees. J. ACM 26(4), 690–715 (1979)
24. Vuillemin, J.: A unifying look at data structures. Communications of the ACM 23(4), 229–239 (1980)
25. Yao, A.C.-C.: Space-time tradeoff for answering range queries (extended abstract). In: Proc. 14th Annual ACM Symposium on Theory of Computing, pp. 128–136. ACM Press, New York (1982)

On Rectilinear Partitions with Minimum Stabbing Number*

Mark de Berg[1], Amirali Khosravi[1], Sander Verdonschot[2],
and Vincent van der Weele[3]

[1] Department of Mathematics and Computing Science, TU Eindhoven,
Eindhoven, The Netherlands
{mdberg,akhosrav}@win.tue.nl
[2] School of Computer Science, Carleton University, Ottawa, Canada
sverdons@connect.carleton.ca
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
vdweele@mpi-inf.mpg.de

Abstract. Let S be a set of n points in $\mathbb{R}^d$, and let r be a parameter with $1 \leqslant r \leqslant n$. A rectilinear r-partition for S is a collection $\Psi(S) := \{(S_1, b_1), \ldots, (S_t, b_t)\}$, such that the sets S_i form a partition of S, each b_i is the bounding box of S_i, and $n/2r \leqslant |S_i| \leqslant 2n/r$ for all $1 \leqslant i \leqslant t$. The (rectilinear) stabbing number of $\Psi(S)$ is the maximum number of bounding boxes in $\Psi(S)$ that are intersected by an axis-parallel hyperplane h. We study the problem of finding an optimal rectilinear r-partition—a rectilinear partition with minimum stabbing number—for a given set S. We obtain the following results.

– Computing an optimal partition is NP-hard already in $\mathbb{R}^2$.
– There are point sets such that any partition with disjoint bounding boxes has stabbing number $\Omega(r^{1-1/d})$, while the optimal partition has stabbing number 2.
– An exact algorithm to compute optimal partitions, running in polynomial time if r is a constant, and a faster 2-approximation algorithm.
– An experimental investigation of various heuristics for computing rectilinear r-partitions in $\mathbb{R}^2$.

1 Introduction

Motivation. Range searching is one of the most fundamental problems in computational geometry. In its basic form it can be stated as follows: preprocess a set S of objects in $\mathbb{R}^d$ into a data structure such that the objects intersecting a query range can be reported (or counted) efficiently. The range-searching problem has many variants, depending for example on the type of objects (points, or some other type of objects), on the dimension of the underlying space (two- or higher dimensional), and on the type of query range (boxes, simplices, etc.)—see the survey of Agarwal and Erickson [1] for an overview.

A range-searching data structure that is popular in practice is the *bounding-volume hierarchy*, or BVH for short. This is a tree in which each object from S is stored in a leaf, and each internal node stores a bounding volume of the objects in its subtree. Often the bounding volume is a *bounding box*: the smallest axis-aligned box containing the objects in the subtree. When a BVH is stored in external memory, one usually uses a B-tree [5, Chapter 18] as underlying tree structure; the resulting structure (with bounding boxes as bounding volumes) is then called an *R-tree*. R-trees are one of the most widely used external-memory data structures for spatial data, and they have been studied extensively—see for example the book by Manolopoulos *et al.* [11]. In this paper we study a fundamental problem related to the construction of R-trees, as explained next.

One common strategy to construct R-trees is the top-down construction. Top-down construction algorithms partition S into a number of subsets S_i, and then recursively construct a subtree $\mathcal{T}_i$ for each S_i. Thus the number of subsets corresponds to the degree of the R-tree. When a range query with a range Q is performed, one has to recursively search in the subtrees $\mathcal{T}_i$ for which the bounding box of S_i (denoted b_i) intersects Q. If $b_i \subset Q$, then all objects stored in $\mathcal{T}_i$ lie inside Q; if, however, b_i intersects ∂Q (the boundary of Q) then we do not know if the objects stored in $\mathcal{T}_i$ intersect Q. Thus the overhead of the search algorithm is determined by the bounding boxes intersecting ∂Q. If Q is a box, as is often the case, then the number of bounding boxes b_i intersecting ∂Q is bounded, up to a factor $2d$, by the maximum number of bounding boxes intersecting any axis-parallel plane. Thus we want to partition S into subsets so as to minimize the number of bounding boxes intersecting any axis-parallel plane.

Further background and problem statement. Let S be a set of n points in $\mathbb{R}^d$, and let r be a parameter with $1 \leqslant r \leqslant n$. A *rectilinear r-partition* for S is a collection $\Psi(S) := \{(S_1, b_1), \dots, (S_t, b_t)\}$ such that the sets S_i form a partition of S, each b_i is the bounding box of S_i, and $n/2r \leqslant |S_i| \leqslant 2n/r$ for all $1 \leqslant i \leqslant t$. Note that even though the subsets S_i form a (disjoint) partition of S, the bounding boxes b_i need not be disjoint. The *stabbing number* of an axis-parallel hyperplane h with respect to $\Psi(S)$ is the number of boxes b_i whose relative interior intersects h, and the *(rectilinear) stabbing number* of $\Psi(S)$ is the maximum stabbing number of any axis-parallel plane h. Observe that our rectilinear r-partitions are the axis-parallel counterpart of the (fine) simplicial partitions introduced by Matoušek [12].

It has been shown that there are point sets S for which any rectilinear r-partition has stabbing number $\Omega(r^{1-1/d})$ [12]; an example is the case when the points in S form a grid of size $n^{1/d} \times \cdots \times n^{1/d}$. Moreover, any set S admits a rectilinear r-partition with stabbing number $O(r^{1-1/d})$; such a rectilinear r-partition can be obtained by a construction similar to a kd-tree [6]. Thus from a worst-case and asymptotic point of view the problem of computing rectilinear r-partitions with low stabbing number is solved. However, any specific point set may admit a rectilinear r-partition with a much lower stabbing number than $\Theta(r^{1-1/d})$. For instance, if the points from S are all collinear on a diagonal line, then there is a rectilinear r-partition with stabbing number 1. The question now

arises: given a point set S and a parameter r, can we compute a rectilinear r-partition that is *optimal for the given input set S*, rather than worst-case optimal? In other words, we want to compute a rectilinear r-partition that has the minimum stabbing number over all rectilinear r-partitions for S.

Our results. We start by a theoretical investigation of the complexity of the problem of finding optimal rectilinear r-partitions. In Section 2 we show that already in $\mathbb{R}^2$, finding an optimal rectilinear r-partition is NP-hard if r is considered as a parameter. In Section 3 we then give an exact algorithm for computing optimal rectilinear r-partitions which runs in polynomial time if r is a constant, and a 2-approximation with a better running time. We conclude our theoretical investigations by showing that algorithms only considering partitions with disjoint bounding boxes cannot have a good approximation ratio: there are point sets such that any partition with disjoint bounding boxes has stabbing number $\Omega(r^{1-1/d})$, while the optimal partition has stabbing number 2. We also perform an experimental investigation of various heuristics for computing rectilinear r-partitions with small stabbing number in $\mathbb{R}^2$. A simple variant of a kd-tree approach, which we call the *windmill kd-tree* turns out to give the best results.

2   Finding Optimal Rectilinear r-Partitions Is NP-Hard

The exact problem we consider in this section is as follows.

OPTIMAL RECTILINEAR r-PARTITION
Input: A set S of n points in $\mathbb{R}^2$ and two parameters r and k.
Output: YES if S admits a rectilinear r-partition w.r.t. r with stabbing number at most k, NO otherwise.

We will show that this problem is already NP-complete for fixed values of k.

Theorem 1. OPTIMAL RECTILINEAR r-PARTITION *is NP-complete for $k = 5$.*

To prove the theorem we use a reduction from 3-SAT, which is similar to the proof by Fekete *et al.* [8] of the NP-hardness of minimizing the stabbing number of a matching on a planar point set. Let $\mathcal{U} := \{x_1, \ldots, x_m\}$ be a set of m boolean variables, and let $\mathcal{C} := C_1 \wedge \cdots \wedge C_s$ be a CNF formula defined over these variables, where each clause C_i is the disjunction of three variables. The 3-SAT problem is to decide whether such a boolean formula is satisfiable; 3-SAT is NP-hard [9]. Our reduction will be such that there is a rectilinear r-partition with stabbing number $k = 5$ for the OPTIMAL RECTILINEAR r-PARTITION instance if and only if the 3-SAT instance is satisfiable. To simplify the reduction we assume that $n = 72r$ (to make $\sqrt{2n/r}$ an integer greater than or equal to 12); however, the reduction works for any $n = \alpha \cdot r$ for $\alpha \geqslant 72$. We first describe the various gadgets we need and then explain how to put them together.

The barrier gadget. A barrier gadget is a set G of $25 \cdot h^2$ points, where $h \geqslant 12$ and $h^2 = 2n/r$, arranged in a regular $5h \times 5h$ grid. To simplify the construction we fix $h = 12$. Thus a barrier gadget is simply a 60×60 grid placed in a small square.

The idea is essentially that if we partition a barrier gadget and require stabbing number 5, then both the vertical and the horizontal stabbing numbers will be 5. This will prevent any other bounding boxes from crossing the vertical strip (and, similarly, the horizontal strip) whose bounding lines contain the vertical (resp. horizontal) edges of the square containing the barrier gadget. Thus the barrier gadget can be used to make sure there is no interaction between different parts of the global construction. Lemma 1 below makes this precise by giving a bound on the minimum stabbing number of any r-partition of a barrier gadget. In fact, we are interested in the case where G is a subset of a larger set S. In our construction we will place any barrier gadget G in such a way that the points in $S \setminus G$ lie outside the bounding box of G, so when analyzing the stabbing number of a barrier gadget we will always assume that this is the case.

Let G be a barrier gadget and $S \supset G$ be a set of n points. We define $\Psi(S \downarrow G)$, the *restriction to G* of a rectilinear r-partition $\Psi(S) = \{(S_1, b_1), \ldots, (S_t, b_t)\}$, as

$$\Psi(S \downarrow G) := \{(S_i \cap G, b_i) : 1 \leqslant i \leqslant t \text{ and } S_i \cap G \neq \emptyset\}.$$

In other words, the boxes in $\Psi(S \downarrow G)$ are the boxes from $\Psi(S)$ whose associated point set contains at least one point from the barrier. The following lemma, which is proved in the full version of the paper, gives a bound on the vertical and horizontal stabbing numbers of a rectilinear partition of a barrier gadget, where the vertical (horizontal) stabbing number is defined as the maximum number of boxes intersected by any vertical (horizontal) line.

Lemma 1. *A barrier gadget G can be covered by a set of 25 boxes with stabbing number 5. Moreover, for any rectilinear r-partition $\Psi(S)$ of stabbing number 5, the restriction $\Psi(S \downarrow G)$ has vertical as well as horizontal stabbing number 5.*

The variable gadget. Fig. 1 shows the variable gadget. The three subsets in the left part of the construction, and the three subsets in the right part, each contain $n/2r = 36$ points. Because of the barrier gadgets, the points from one subset cannot be combined with other points and must be put together into one rectangle in the partition. The six subsets in the middle part of the construction each contain $4n/r = 288$ points. To make sure the stabbing number does not exceed 5, these subsets can basically be grouped in two different ways. One grouping corresponds to setting the variable to true, the other grouping to false— see Fig. 1. Note that the gadget defines two vertical *slabs*. If the variable is set to true then the left slab has stabbing number 2 and the right slab has stabbing number 4, otherwise the opposite is the case.

The clause gadget. A clause gadget consists of three subsets of $4n/r = 288$ points, arranged as shown in Fig. 2(a), and placed in the left or right slab of the corresponding variables: a positive literal is placed in the left slab, a negative lateral in the right slab. If the stabbing number of the slab is already 4, which is the case when the literal evaluates to false, then the subset of $4n/r$ points in the clause gadget must be grouped into two "vertical" rectangles. Hence, not all literals in a clause can evaluate to false if the stabbing number is to be at most 5.
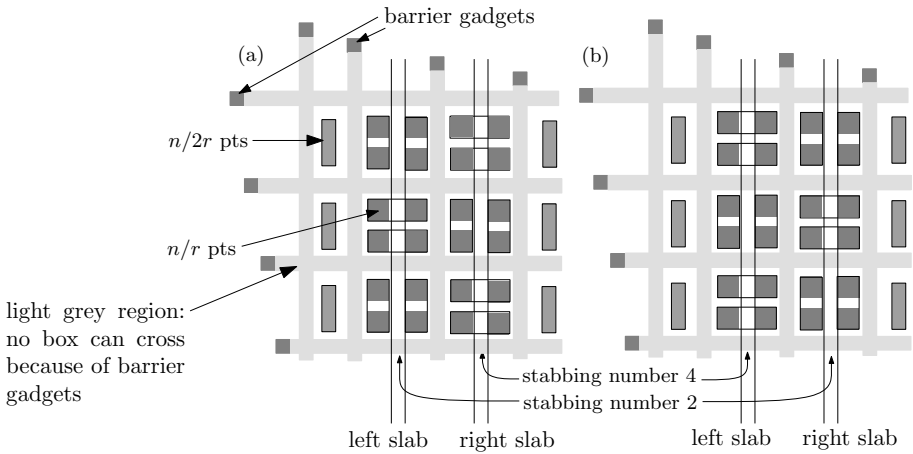
Fig. 1. The variable gadget. (a) True setting. (b) False setting.

The global structure. The global construction is shown in Fig. 2(b). There are variable gadgets, clause gadgets, and barrier gadgets. The variable gadgets are placed diagonally and the clause gadgets are placed below the variables. We also place barriers separating the clause gadgets from each other. Finally, the gadgets for occurrences of the same variable in different clauses should be placed such that they are not stabbed by a common vertical line. This concludes our sketch of the construction which proves Theorem 1. A formal proof of the theorem can be found in the full version of the paper.

3   Polynomial Time Algorithms for Constant r

In the previous section we showed that OPTIMAL RECTILINEAR r-PARTITION is NP-hard when r is considered part of the input. Now we give a simple algorithm
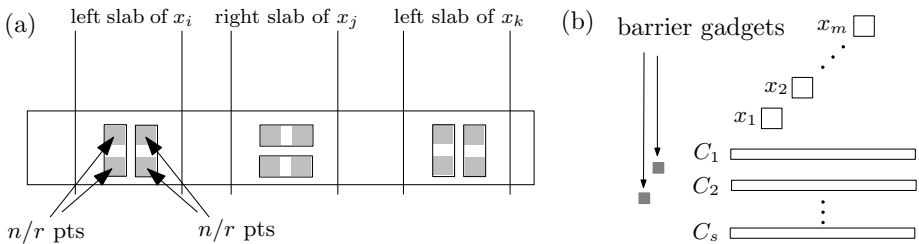

Fig. 2. (a) A clause gadget for $(x_i \lor \overline{x_j} \lor x_k)$, and one possible grouping of the points. (b) The global structure.

to show that the problem in $\mathbb{R}^d$ can be solved in polynomial time for fixed r in dimension d. Our algorithm works as follows.

1. Let C be the set of all boxes defined by at most $2d$ points in S. Note that $|C| = O(n^{2d})$.
2. For each t with $r/2 \leqslant t \leqslant 2r$, proceed as follows. Consider all $O(n^{2dt})$ possible subsets $B \subset C$ with $|B| = t$. Check whether B induces a valid solution, that is, whether we can assign the points in S to the boxes in B such that (i) each point is assigned to a box containing it, and (ii) each box is assigned between $n/2r$ and $2n/r$ points. How this is done will be explained later.
3. Over all sets B that induce a valid solution, take the set with the smallest stabbing number. Replace each box in it with the bounding box of the points assigned to it, and report the partition.

To implement Step 2 we construct a flow network with node set $\{v_{\text{source}}, v_{\text{sink}}\} \cup S \cup B$. The source node v_{source} has an arc of capacity 1 to each point $p \in S$, each $p \in S$ has an arc of capacity 1 to every $b_j \in B$ that contains p, and each $b_j \in B$ has an arc of capacity $2n/r$ to the sink node v_{sink}. The arcs from the boxes to the sink also have (besides the upper bound of $2n/r$ on the flow) a lower bound of $n/2r$ on the flow. The set B induces a valid rectilinear r-partition if and only if there is an integer flow of n units from v_{source} to v_{sink}. Such a flow problem can be solved in $O(\min(V^{3/2}, E^{1/2})E \log(V^2/E + 2) \log c)$ time [2], where V is the number of vertices in the network, E is the number of arcs, and c is the maximum capacity of any arc. We have $V = O(n)$, $E = O(nr)$, and $c = 2n/r$. Since we have to check $O(n^{4dr})$ subsets B, the running time is $O(n^{4dr} \cdot (nr)^{3/2} \log^2(n/r + 2))$ and is polynomial (assuming r is a constant). We obtain the following result.

Theorem 2. *Let S be a set of n points in $\mathbb{R}^d$, and r a constant. Then we can compute a rectilinear r-partition with optimal stabbing number in time $O(n^{4dr+3/2} \log^2 n)$*

We can significantly improve the running time if we are satisfied with a 2-approximation. The trick is to place a collection H_i of $3r$ hyperplanes orthogonal to the x_i-axis (the ith-axis) such that there are at most $n/3r$ points from S in between any two consecutive hyperplanes in H_i. Instead of finding $O(n^{2d})$ boxes in the first step of the algorithm, we now find $O(r^{2d})$ boxes defined by the hyperplanes in $H := H_1 \cup \cdots \cup H_d$. Then we have $|C| = O(r^{2d})$. We apply the Step 2 of the algorithm and find for $r/2 \leqslant t \leqslant 2r$ all the $O(r^{2dt})$ subsets $B \subset C$. We check for each subset whether it is a valid solution, and take the best valid solution. In the full version, we show that this gives a 2-approximation.

Theorem 3. *Let S be a set of n points, and r a constant. Then we can compute a rectilinear r-partition with stabbing number at most $2 \cdot \text{OPT}$, where OPT is the minimum stabbing number of any rectilinear partition for S, in time $O(n^{3/2} \log^2 n)$.*

4   Arbitrary versus Disjoint Rectilinear r-Partitions

Since computing optimal rectilinear r-partitions is NP-hard, one should look at approximation algorithms. It may be easier to develop an approximation algorithm considering only rectilinear r-partitions with disjoint bounding boxes. The next theorem shows that in $\mathbb{R}^2$ such an approach will not give a good approximation ratio. The same argument holds for $\mathbb{R}^d$. (see the full version)

Theorem 4. *Assume that $32 \leqslant r \leqslant 4 \cdot \sqrt{n}$. Then there is a set S of n points in $\mathbb{R}^2$ whose optimal rectilinear r-partition has stabbing number 2, while any rectilinear r-partition with disjoint bounding boxes has stabbing number $\Omega(\sqrt{r})$.*

Proof. Let $\mathcal{G}$ be a $\sqrt{r/8} \times \sqrt{r/8}$ grid in $\mathbb{R}^2$. (For simplicity assume that $\sqrt{r/8}$ is an integer. Since $32 \leqslant r$ we have $\sqrt{r/8} \geqslant 2$.) We put each grid point in S and call them *black points*. We call the lines forming the grid $\mathcal{G}$ *black lines*. Note that there are $r/8$ black points. Fig. 3 shows an example with $r = 128$. Next we refine the grid using $2(\sqrt{r/8} - 1)$ additional axis-parallel *grey lines*. At each of the new grid points that is not fully defined by gray lines—the grey dots in the figure—we put a tiny cluster of $2n/r$ points, which we also put in S. If the cluster lies on one or more black lines, then all points from the cluster lie in the intersection of those lines, as shown in Fig. 3.

So far we used $(2r/8 - 2 \cdot \sqrt{r/8}) \cdot 2n/r + r/8$ points. Since $r \leqslant 4 \cdot \sqrt{n}$, the number of points which we used so far is less than n. The remaining points can be placed far enough from the construction (not influencing the coming argument.) Next, we rotate the whole construction slightly so that no two points have the same coordinate in any dimension. This rotated set is our final point set S.

To obtain a rectilinear r-partition with stabbing number 2, we make each of the clusters into a separate subset S_i, and put the black points into one separate subset; the latter is allowed since $r/8 \leqslant 2n/r$. (If $r/8 < n/2r$ we can use some of the remaining points or the points of gray dots to fill up the subset.)

If the clusters are small enough, then the rotation we have applied to the point set guarantees that no axis-parallel line can intersect two clusters at the same time. Any line intersects at most one of the clusters and the rectangle containing the black points, and the stabbing number of this rectilinear r-partition is 2.

We claim that any disjoint rectilinear r-partition for S has stabbing number $\Omega(\sqrt{r/8})$. To see this, observe that no subset S_i in a disjoint rectilinear r-partition can contain two black points. Indeed, the bounding box of any two black points contains at least one full cluster and, hence, together with the black points would be too many points. We conclude that each black point is assigned to a different bounding box. Let B be the collection of these bounding boxes. Now duplicate each of the black lines, and move the two duplicates of each black line slightly apart. This makes a set H of $O(\sqrt{r/8})$ axis-parallel lines such that each bounding box in B intersects at least one line from H.

Then the total number of intersections between the boxes in B and the lines in H is $\Omega(r)$, implying that there is a line in H with stabbing number $\Omega(\sqrt{r/8})$. $\square$
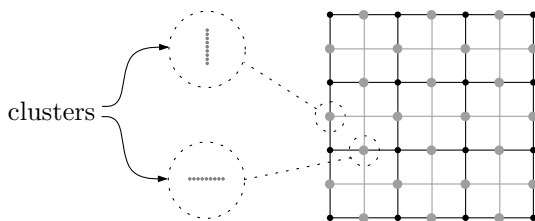
Fig. 3. Every rectilinear r-partition with disjoint bounding boxes has stabbing number $\Omega(\sqrt{r})$ while there exists a partition with stabbing number 2
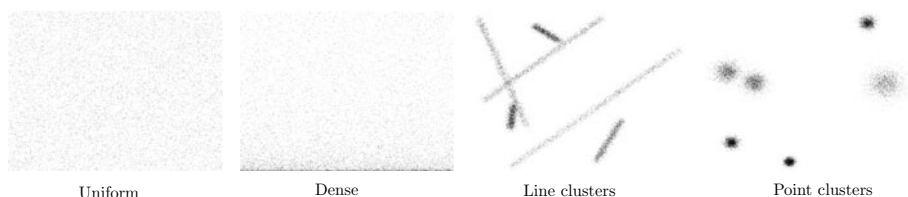

| Uniform | Dense | Line clusters | Point clusters |

Fig. 4. The different types of input sets

5   Experimental Results

In the previous sections we studied the complexity of finding an optimal rectilinear r-partition of a given point set. For arbitrary r the problem is NP-hard, and for constant r the exact algorithm was polynomial but still very slow. Hence, we now turn our attention to heuristics. In the initial experiments on which we report below, the focus is on comparing the various heuristics and investigating the stabbing numbers they achieve as a function of r, for fixed n.

Data sets. We tested our heuristics on four types of point sets–see Fig. 4. The *Uniform* data set picks the points uniformly at random from the unit square. For the *Dense* data set we take a *Uniform* data set and square all y-coordinates, so the density increases near the bottom. For the *Line Clusters* data set we first generated a few line segments, whose endpoints are chosen uniformly at random in the unit square. To generate a point in P, we pick one of the line segments randomly, select a position along the line segment at random and add some Gaussian noise. The *Point Clusters* data set is similar, except that it clusters around points instead of line segments. All sample sets contain $n = 50,000$ points and the reported stabbing numbers are averages over 20 samples.

Next we describe our heuristics. Let P denote the set of points in $\mathbb{R}^2$ for which we want to find a rectilinear r-partition with low stabbing number.

The windmill kd-tree. A natural heuristic is to use a kd-tree [6]: partition the point set P recursively into equal-sized subsets, alternately using vertical and horizontal splitting lines, until the number of points in each region drops below
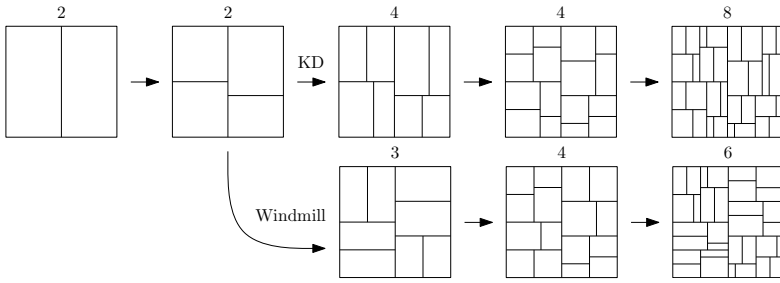
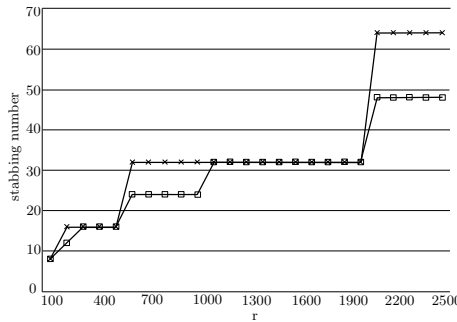**Fig. 5.** A kd-tree, a windmill kd-tree, and their stabbing numbers

Fig. 6. Comparison of the kd-tree and the windmill kd-tree

$2n/r$. For each region R of the kd-tree subdivision, put the pair $(R \cap P, b_R)$ into the rectilinear r-partition, where b_R is the bounding box of $R \cap P$. Note that this method runs in $O(n \log n)$ time, and gives a stabbing number $O(\sqrt{r})$, which is worst-case optimal. The *windmill kd-tree* is a version of kd-tree in which for two of the four nodes of depth 2 the splitting line has the same orientation as the splitting line at their parents. This is done in such a way that the subdivision induced by the nodes at level 2 has stabbing number 3 rather than 4–see Fig. 5. It turns out that the windmill kd-tree is always at least as good as the regular kd-tree, and often performs significantly better. The results for the uniform Data set are shown in Fig. 6 for r ranging from 100 to 2,500 with step size 100. The figure shows that, depending on the value of r, the stabbing number of the kd-tree and the windmill kd-tree are either the same, or the windmill has 25% lower stabbing number. The switch between these two cases occurs exactly at the values of r where the depth switches from even to odd (or vice versa), which is as expected when looking at Fig. 5. In the remainder we only compare the windmill kd-tree to the other methods, and ignore the regular kd-tree.

The greedy method. We first compute a set $\mathcal{B}$ of candidate boxes such that $n/2r \leqslant |b_i \cap P| \leqslant 2n/r$ for each box $b_i \in \mathcal{B}$. Each box $b_i \in \mathcal{B}$ has a certain *cost* associated to it. We then take the cheapest box $b_i \in \mathcal{B}$, put the pair $(b_i, P \cap b_i)$ into the rectilinear r-partition, and remove the points in b_i from P. Finally, boxes
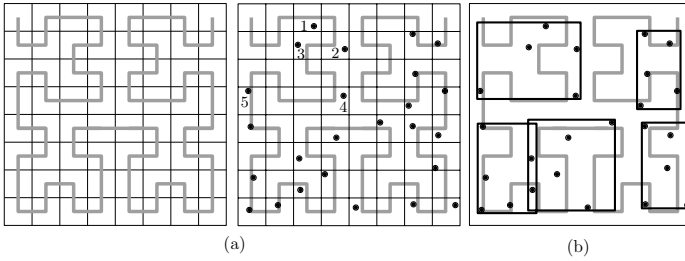
Fig. 7. A Hilbert curve and its use to generate a rectilinear r-partition

that now contain too few points are discarded, the costs of the remaining boxes are updated, and the process is repeated. The method ends when the number of points drops below $2n/r$; these points are then put into a single box (which we allow to contain less than $n/2r$ points if needed).

This method can be implemented in various ways, depending on how the set $\mathcal{B}$ and the cost of a box are defined. In our implementation we took m vertical lines with (roughly) $n/(m-1)$ points in between any two consecutive lines, and m horizontal lines in a similar manner. $\mathcal{B}$ then consists of all $O(m^4)$ boxes that can be constructed by taking two vertical and two horizontal lines from these lines. In our experiments we used $m = 50$, because this was the largest value that gave reasonable computation times. The cost of a box b_i is defined as follows. We say that a point $p \in P$ is *in conflict with* b_i if $p \notin b_i$ and the horizontal or the vertical line through p intersects b_i. Let C_i be the set of points in conflict with b_i. Then the cost of b_i is $|C_i|/|b_i \cap P|$. The idea is that we prefer boxes containing many points and in conflict with few points.

The Hilbert curve. A commonly used approach to construct R-trees is to use a space-filling curve such as a Hilbert curve [10]—see Fig. 7(a). We can also use a Hilbert curve to compute a rectilinear r-partition: first, sort the given points according to their position on the Hilbert curve, and then generate the subsets in the rectilinear r-partition by taking consecutive subsets along the Hilbert curve. Since the lowest stabbing number is usually achieved by using as few rectangles as possible, we do this in a greedy manner: put the first $2n/r$ points in the first subset, the next $2n/r$ rectangles in the second subset, and so on—see Fig. 7(b).

K-Means. The final method we tested was to compute r clusters using K-means—in particular, we used K-Means++ [3]—and then take the clusters as the subsets in the rectilinear r-partition. Some of the resulting clusters may contain too many or too few points. We solved this by shifting points into neighboring clusters.

5.1 Results of the Comparisons

Figs. 8.a-d shows the results of our experiments. The clear conclusion is that the windmill kd-tree outperforms all other methods on all data sets. The Hilbert-curve approach always comes in second, except for the *Dense* data set. Note
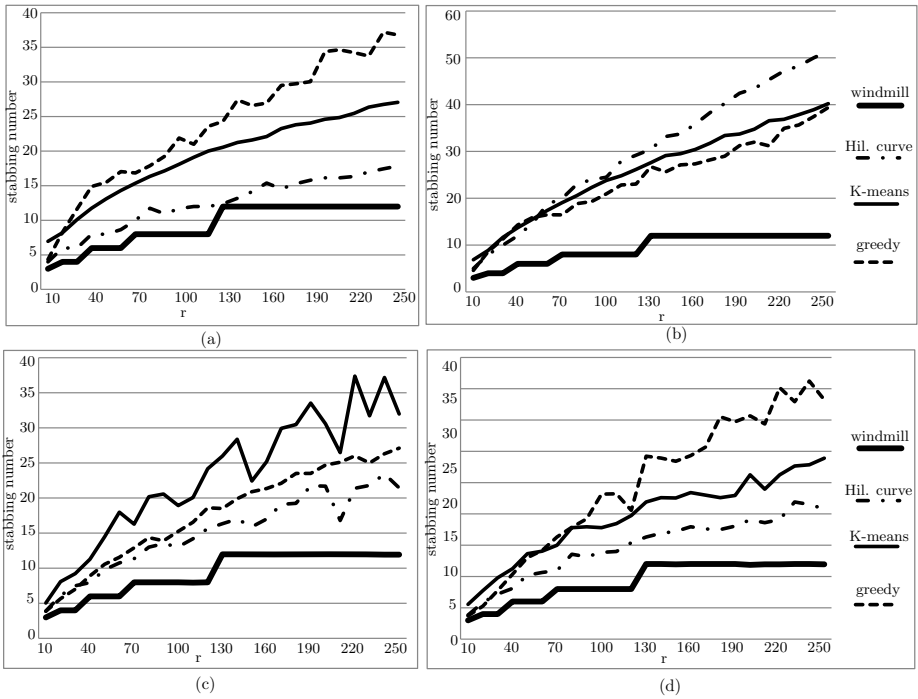
Fig. 8. The results of the comparison of methods on (a)*uniform* (b)*dense* (c)*line clusters* (d)*point cluster* point sets

that the windmill and the greedy method give the same results for the *Uniform* data set and the *Dense* data set—which is easily explained, since the rectilinear r-partition computed by these methods only depends on the ranks (their position in the sorted order) of the coordinates, and not on their actual values—while the other two methods perform worse on the *Dense* data set: apparently they do not adapt well to changing density. The windmill and the Hilbert-curve method not only gave the best results, they were also the fastest. Indeed, both methods could easily deal with large data sets. (On inputs with $n = 10,000,000$ and $r = 500$ they only took a few minutes.)

6 Conclusion

We studied the problem of finding optimal rectilinear r-partitions of point sets. On the theoretical side, we proved that the problem is NP-hard when r is part of the input, although it can be solved in polynomial time for constant r. The algorithm for constant r is still unpractically slow, however, so it would be interesting to come up with faster exact algorithms (or perhaps a practically efficient PTAS).

We also tested a few heuristics and concluded that our so-called windmill kd-tree is the best method. This immediately leads to the question whether the windmill approach could also lead to R-trees that are practically efficient. This is, in fact, unclear. What we have tried to optimize is the maximum stabbing number of any axis-parallel line. When querying with a rectangular region, however, we are interested in the number of regions intersected by the boundary of the region. First of all, the boundary does not consist of full lines, but of line segments that in practice are possibly small compared to the data set. Secondly, the boundary of the rectangle consists of horizontal and vertical segments. Now, what the windmill does (as compared to a regular kd-tree) is to balance the horizontal and vertical stabbing number, so that the maximum is minimized. The sum of the horizontal and vertical stabbing number in the subdivision does not change, however. So it might be that the windmill approach is good to minimize the worst-case query time for long and skinny queries. This would require further investigation. It would also be interesting to find rectilinear r-partitions whose (maximum or average) stabbing number is optimal with respect to a given set of query boxes, or try to minimize the full partition tree, instead of just one level.

References

1. Agarwal, P.K., Erickson, J.: Geometric Range Searching and its Relatives. In: Chazelle, B., Goodman, J., Pollack, R. (eds.) Advances in Discrete and Computational Geometry, pp. 1–56 (1998)
2. Ahuja, P.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
3. Arthur, D., Vassilvitskii, S.: K-means++: the Advantages of Careful Seeding. In: Proc. of the 18th Annual ACM-SIAM Sym. of Desc. Alg., pp. 1027–1035 (2007)
4. Chazelle, B., Welzl, E.: Quasi-optimal Range Searching in Spaces of Finite VC-dimension. Arch. Rat. Mech. Anal. 4, 467–490 (1989)
5. Ahuja, P.K., Magnanti, T.L., Orlin, J.B.: Introduction to Algorithms, 2nd edn. MIT Press and McGraw-Hill (2001)
6. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational Geometry: Algorithms and Applications, 3rd edn. Springer, Heidelberg (2008)
7. Dijkstra, E.W., Feijen, W.H.J., Sterringa, J.: A Method of Programming. Addison-Wesley, Reading (1988)
8. Fekete, S.P., Lübbecke, M.E., Meijer, H.: Minimizing the Stabbing Number of Matchings, Trees, and Triangulations. Discr. Comput. Geom. 40, 595–621 (2008)
9. Garey, M.R., Johnson, D.S.: Computers and Interactibility: A Guide to the Theory of NP-Completness. W.H. Freeman and Co., New York (1979)
10. Haverkort, H., van Walderveen, F.: Four-dimensional Hilbert Curves for R-trees. In: Proc. Workshop on Algorithms Engineering and Experiments, ALANEX (2009)
11. Manolopoulos, Y., Nanopoulos, A., Theodoridis, Y., Papadopoulos, A.: R-trees: Theory and Applications. Series in Adv. Inf. and Knowledge Processing. Springer, Heidelberg (2005)
12. Matoušek, J., Tarantello, G.: Efficient Partition Trees. Discr. Comput. Geom. 8, 315–334 (1992)

Flattening Fixed-Angle Chains
Is Strongly NP-Hard*

Erik D. Demaine and Sarah Eisenstat

MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar St., Cambridge, MA 02139, USA
{edemaine,seisenst}@mit.edu

Abstract. Planar configurations of fixed-angle chains and trees are well studied in polymer science and molecular biology. We prove that it is strongly NP-hard to decide whether a polygonal chain with fixed edge lengths and angles has a planar configuration without crossings. In particular, flattening is NP-hard when all the edge lengths are equal, whereas a previous (weak) NP-hardness proof used lengths that differ in size by an exponential factor. Our NP-hardness result also holds for (nonequilateral) chains with angles in the range $[60° - \varepsilon, 180°]$, whereas flattening is known to be always possible (and hence polynomially solvable) for equilateral chains with angles in the range $(60°, 150°)$ and for general chains with angles in the range $[90°, 180°]$. We also show that the flattening problem is strongly NP-hard for equilateral fixed-angle trees, even when every angle is either $90°$ or $180°$. Finally, we show that strong NP-hardness carries over to the previously studied problems of computing the minimum or maximum span (distance between endpoints) among non-crossing planar configurations.

Keywords: geometric folding, linkages, hardness, polymers.

1   Introduction

Molecular geometry (also called *stereochemistry*) studies the 3D geometry of the atoms (and the bonds between them) that constitute a molecule [5]. If we represent an atom by a vertex and a bond by an edge, we obtain a graph structure; this structure comes equipped with fixed edge (bond) lengths, making a *linkage*, and fixed (bond) angles between incident edges, making a *fixed-angle linkage*. In general, a *fixed-angle linkage* is a geometrically embedded graph that can reconfigure (change embedding) so long as it preserves the fixed edge lengths and angles [4]. Typical edge (bond) lengths in polymers are 100–270 picometers, and typical (bond) angles are around $72°$, $90°$, $109°$, $120°$, and $180°$.

Most large (macro)molecules are *polymers*, and many are *nonbranching polymers*, meaning that the graph structure decomposes into a chain of substructures of small size. Examples of nonbranching polymers include proteins, DNA strands, and RNA strands. Motivated by this reality, the computational study

---

of fixed-angle linkages [4] usually focuses on *fixed-angle chains*, where the graph is a path (representing the *backbone* of the polymer), and on *fixed-angle trees*, where the graph is a tree, especially caterpillars, representing a small amount of additional structure attached to the backbone.

Motivated by these applications to polymer science, Soss and Toussaint [9,8] introduced several computational problems, three of which we study here:

Flattening: Given a fixed-angle linkage, decide whether it has a continuous non-crossing motion that results in a flat configuration (lying in the plane).

Min flat span: Compute the flat configuration of a fixed-angle linkage with minimum possible *span*—distance between the two endpoints.

Max flat span: Compute the flat configuration of a fixed-angle linkage with maximum possible span.

They proved that all three of these problems are weakly NP-hard, by reducing from the integer partition problem. Because the integer partition problem is weakly NP-hard, it is only hard when the numbers in the problem are exponentially large. Therefore, the reductions given by Soss and Toussaint show only that the flattening and span problems are hard when the edge lengths differ by exponential factors, or when the angles have polynomially many bits of precision.

Our results. In this paper, we show that all three problems are strongly NP-hard, and thus hard when the edge lengths are all very close (or even identical) and a constant number of different angles are used. More specifically, we prove the following special cases to be strongly NP-hard:[1]

Problem	Linkage	Edge lengths	Angle range	Theorem
Flattening	fixed-angle chain	equilateral	$[16.26°, 180°]$	4
Flattening	fixed-angle chain	$\Theta(1)$	$[60 - \varepsilon°, 180°]$	5
Flattening	fixed-angle caterpillar tree	equilateral	$\{90°, 180°\}$	3
Min flat span	fixed-angle chain	equilateral	$[16.26°, 180°]$	6
Max flat span	fixed-angle chain	equilateral	$[16.26°, 180°]$	7

The proofs given in this paper show the NP-hardness of the related problem of deciding the existence of a non-crossing flat state. It is easy to modify our proofs to establish hardness of flattening (by a continuous motion). The $16.26° \approx$ arcsin $\frac{7}{25}$ angle bound can easily be improved to around $22.6°$, and perhaps further to $30°$ or $45°$.

Overview. Our proofs start in Section 3 with an artificial problem, flattening *semi-rigid* fixed-angle chains, as a building block for the more interesting results above. In a semi-rigid chain, some sections of the chain can be marked *rigid*, meaning that the vertices in the section cannot move relative to each other. Naturally, this additional set of constraints makes flattening more difficult, and we show that the problem is NP-hard even for equilateral chains with all angles in $\{90°, 180°\}$. Then we show in Section 4 how to remove the semi-rigidity

---

[1] A linkage is *equilateral* if all edge lengths are equal. "$\Theta(1)$" denotes that all edge lengths are within constant factors of each other.

constraint using either sharper angles or fixed-angle trees. Finally, in Section 5, we show how to transform the flat state into one with an especially small or large span, and guarantee that the chain has a flat state in all cases (of possibly suboptimal span). Omitted proof details will be available in the full version.

2   Definitions

2.1   Linkages

Definition 1. *A* linkage *consists of a graph $G = (V, E)$ and edge lengths $\ell :$ $E \to \mathbb{R}_{\geq 0}$. G is called the* structure graph *of the linkage. A* configuration *of a linkage in d dimensions is a mapping $C : V \to \mathbb{R}^d$ satisfying the constraint $\ell(u, v) = \|C(u) - C(v)\|$ for each edge $(u, v) \in E$. A configuration is* non-crossing *if any two edges $e_1, e_2 \in E$ intersect only if the two edges are incident in the structure graph, and intersect only at their shared vertex.*

Definition 2. *A* fixed-angle linkage *is a linkage with an additional set of constraints specifying an angle function $\theta_i : \mathcal{N}(v_i) \times \mathcal{N}(v_i) \to [0°, 180°]$ for each vertex v_i, where $\mathcal{N}(v_i)$ is the set of neighbors of v_i. In addition to satisfying the length constraints of the linkage, any configuration of the linkage has the property that for each vertex $v_i \in V$ and each pair of its neighbors $v_j, v_k \in \mathcal{N}(v_i)$, the angle $\angle v_j v_i v_k$ has measure $\theta_i(v_j, v_k)$.*

Definition 3. *A* chain *of length n is a linkage whose structure graph is $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{(v_1, v_2); (v_2, v_3); \ldots; (v_{n-1}, v_n)\}$.*

Definition 4. *A linkage is* equilateral *if $\ell(e) = 1$ for all $e \in E$.*

Definition 5. *A fixed-angle linkage is* orthogonal *if all angles $\theta_i(v_j, v_k)$ are either $90°$ or $180°$.*

Definition 6. *A* flat state *of a fixed-angle linkage is a non-crossing 2D configuration of the linkage. A 3D configuration of a fixed-angle chain can be* flattened *if there exists a continuous sequence of non-crossing configurations starting at the current configuration and ending in a flat state.*

Definition 7. *The* span *of a flat state of a fixed-angle chain is the distance between v_1 and v_n in that configuration.*

Finally, we define a new kind of fixed-angle chain, which places an additional constraint on the locations of the vertices in a configurations.

Definition 8. *A* semi-rigid chain *of length n is a fixed-angle chain of length n with constraints to ensure that parts of the chain are rigid. These constraints are specified in two parts: a sequence $s_0 < s_1 < \ldots < s_\ell$ such that $s_0 = 1$ and $s_\ell = n$; and the distance functions $d_1, \ldots, d_\ell$, where each d_i gives all pairwise distances between the vertices $\{v_{s_{i-1}}, v_{s_{i-1}+1}, \ldots, v_{s_i}\}$. The* articulation points *of a semi-rigid chain are the vertices $v_{s_0}, v_{s_1}, \ldots, v_{s_\ell}$.*

The additional restrictions imposed by the semi-rigid chain make it easier to prove that flattening the chain is NP-hard. The relationship between semi-rigid chains and fixed-angle chains makes it possible to give a reduction from one to the other.

2.2   Rectilinear Planar Monotone 3-SAT

One variant of the standard 3-SAT problem is *planar 3-SAT*, where the graph of the variables and clauses, with edges between variables and the clauses that contain them, has a planar embedding. Planar 3-SAT is known to be NP-complete [7]. One variant of planar 3-SAT, *rectilinear planar 3-SAT*, places three additional restrictions on the planarity of the graph:

1. All variables and clauses are rectangles.
2. All of the variables lie along a single horizontal line.
3. All edges lie along vertical lines.

Rectilinear planar 3-SAT is also known to be NP-complete [6]. In 2010, de Berg and Khosravi introduced an even more restricted version of rectilinear planar 3-SAT [3]: an instance of the *rectilinear monotone planar 3-SAT problem* is a rectilinear planar 3-SAT instance such that every clause is either all positive or all negative, all positive clauses lie above the line of variables, and all negative clauses lie below the line of variables. They proved the following theorem:

Theorem 1. *It is NP-complete to decide whether an instance of rectilinear monotone planar 3-SAT is satisfiable.*

3   Flattening Semi-rigid Chains

In this section, we begin by constructing gadgets for a semi-rigid chain which have a limited number of flat states. Then in Theorem 2, we use those gadgets to show that it is NP-hard to find a flat state for an equilateral orthogonal semi-rigid chain.

Lemma 1. *Up to reflection, the semi-rigid chain depicted in Figure 1(a) has three possible flat states, depicted in Figures 1(a), 1(b), and 1(c).*

Lemma 2. *Given the location of the section of chain between a_0 and a_1, each flat state of the semi-rigid chain depicted in Figure 2 has the following properties:*

1. *The point a_{17} has coordinates $(3, 0)$.*
2. *The y-coordinate of at least one of b_1, b_2, or b_3 must be negative.*

Lemmas 1 and 2 can both be proved by case analysis. We now use the results of Lemma 2 to show the following theorem.

Theorem 2. *There exists a polynomial-time algorithm reducing from an instance ϕ of rectilinear planar monotone 3-SAT to an orthogonal equilateral semi-rigid chain which can be flattened if and only if ϕ is satisfiable.*

Proof. The *pins* of the clause gadget depicted in Fig. 2 are the rigid chains between between articulation points a_8 and a_9, between a_{11} and a_{12}, and between a_{14} and a_{15}. Lemma 2 shows that all possible flat states for that clause gadget

Fig. 1. The three possible flat states for the semi-rigid chain given in Fig. 1(a). Each labeled point is an articulation point; all other sections of the chain are rigid.

have the property that at least one of b_1, b_2, or b_3 must lie below a certain line. A clause has the property that at least one of its literals must be true. So to set up the reduction from one to the other, our literals should be pieces of a semi-rigid chain such that, if the literal is false, the chain will intersect with the corresponding pin when it protrudes below the line. That way, if there is a flat state, then at least one of the literals for that clause must be true. We will accomplish this using gadgets like those depicted in Fig. 3. If the pin extends below the line, then the literal gadget must also dip below the line. If the pin does not extend below the line, then the literal gadget can go either way.

Because we are reducing from monotone rectilinear planar 3-SAT, we know that each clause will contain either all negative or all positive literals, and that all positive clauses will lie above the variables while all negative clauses will lie below the variables. Our choice of gadget for the literal allows us to construct a clause involving the literal's negation by mirroring a clause gadget over the horizontal line and making the pins point upwards instead of downwards.

Unfortunately, there are two problems with the idea we have sketched. The first is direction. In the rigid chain which is partially depicted in Fig. 3, it would be equally valid to have a flat state where the clause gadget is mirrored across the line so that its pins point up. If there is only one clause gadget, then we may say without loss of generality that the clause gadget will fall above the line. However, as soon as there is more than one clause gadget, we may have to consider the possibility of flat states where one clause gadget is in the right
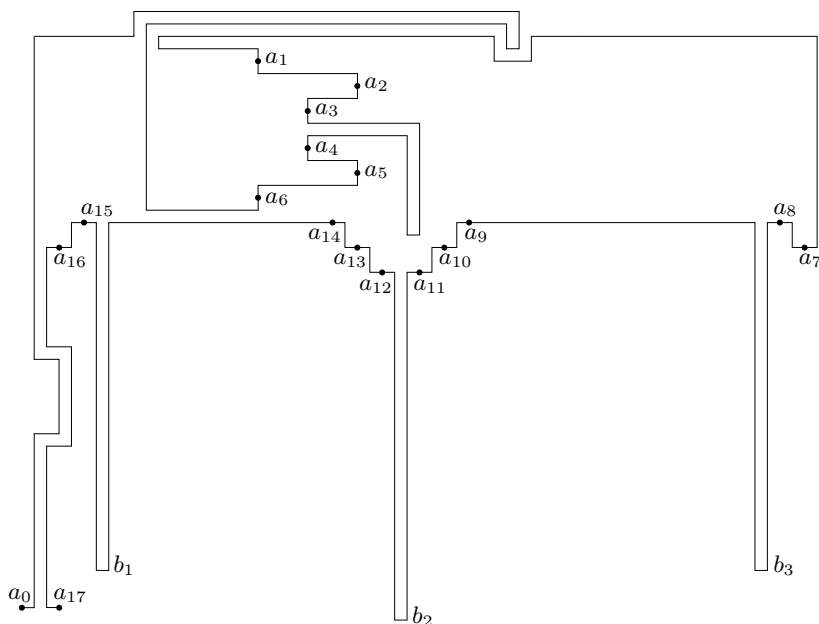
Fig. 2. The semi-rigid chain used for a clause gadget. The articulation points are $a_0, \ldots, a_{17}$. The coordinates of all labelled points are given in Table 1. The parameter h adjusts the height of the gadget; the parameters w_1 and w_2 adjust the distances between the points $b_1, b_2,$ and b_3.

position while the other is in the wrong position. Hence, we need a way to make sure that each clause gadget extends in the right direction. The second problem we must consider is consistency. In order to correctly convert from the rectilinear structure to our fixed-angle chain, we must be able to have a clause gadget appear in between two literal gadgets for the same variable. But if the two literal gadgets are independent, there is no way to ensure that the two gadgets will take on the same value.

The modification we make will solve both of these problems. We will have three separate sections of the chain running parallel to each other. The chain in the middle will consist of a number of long variable gadgets, one for each variable in the original formula. The chain above the middle chain will contain the clause gadgets for all of the positive clauses, as well as smaller literal gadgets. This will ensure that each clause gadget must extend above the chain; if it extended below, it would intersect with the chain in the middle. The chain below the middle chain will contain the clause gadgets for all of the negative clause gadgets, as well as a number of smaller literal gadgets. A sample of this is depicted in Fig. 4. We will connect the three chains as depicted in Fig. 5.

We say that a variable is true if the long gadget for that variable in the middle chain dips below the center line of the middle chain; the variable is false otherwise. Hence, if a variable is false, then all of the smaller gadgets for that

Table 1. The coordinates for the labeled points in Fig. 2

label	x-coord	y-coord	label	x-coord	y-coord	label	x-coord	y-coord
a_0	0	0	a_7	$w_1 + w_2 + 10$	h	a_{14}	$w_1 + 1$	$h + 2$
a_1	$w_1 - 5$	$h + 15$	a_8	$w_1 + w_2 + 8$	$h + 2$	a_{15}	5	$h + 2$
a_2	$w_1 + 3$	$h + 13$	a_9	$w_1 + 12$	$h + 2$	a_{16}	3	h
a_3	$w_1 - 1$	$h + 11$	a_{10}	$w_1 + 10$	h	a_{17}	3	0
a_4	$w_1 - 1$	$h + 8$	a_{11}	$w_1 + 8$	$h - 2$	b_1	7	3
a_5	$w_1 + 3$	$h + 6$	a_{12}	$w_1 + 5$	$h - 2$	b_2	$w_1 + 7$	-1
a_6	$w_1 - 5$	$h + 4$	a_{13}	$w_1 + 3$	h	b_3	$w_1 + w_2 + 7$	3
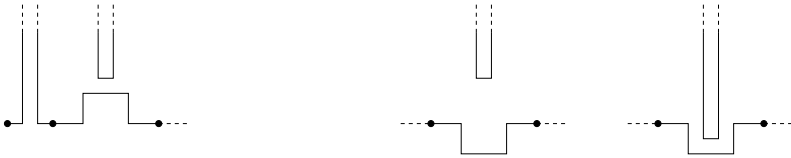
Fig. 3. The interaction between the pins in the clause gadget and the gadgets used for the literals

variable in the top chain must rise above the center line. So a clause containing positive literals cannot lower one of its pins for a variable which is false. If on the other hand a variable is true, then any smaller gadgets for that variable in the bottom chain must also dip below the center line. Hence, any clause with all negative literals cannot raise the pin for a variable which is true. In other words, for any clause, the pin which is lowered (or raised, depending on the clause type) cannot correspond to a false literal. So the only way to get a non-intersecting flat state is to have at least one true literal in each clause.   □

4   Flattening Fixed-Angle Chains and Trees

In this section, we give reductions from semi-rigid chains to several kinds of fixed-angle linkages. In Theorem 3, we provide a reduction to orthogonal equilateral fixed-angle trees. In Theorem 4, we provide a reduction to equilateral fixed-angle chains with minimum angle $\arcsin \frac{7}{25} \approx 16.26°$. In Theorem 5, we provide a reduction to general fixed-angle chains with edge lengths $\Theta(1)$ and angles $> 60° - \varepsilon$.

Theorem 3. *There exists a polynomial-time algorithm which takes as input an orthogonal equilateral semi-rigid chain, and outputs an orthogonal equilateral fixed-angle tree that can be flattened if and only if the semi-rigid chain can be flattened.*

Proof. The first step in the conversion process is to merge adjacent edges within the same rigid piece which have an angle of 180° between them. This means that our semi-rigid chain is no longer equilateral, and instead has integer lengths
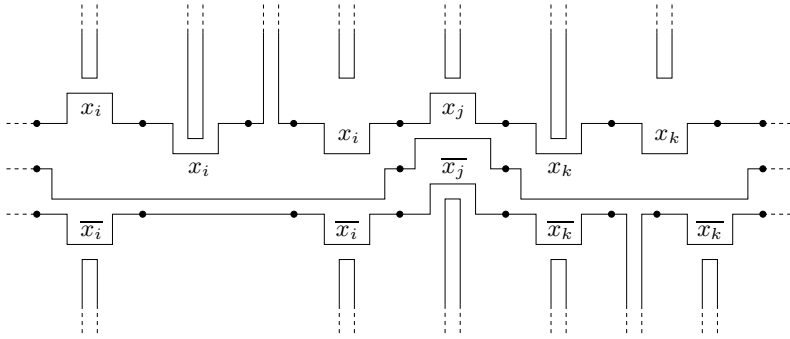
Fig. 4. A sample of the three pieces of the semi-rigid chain which will be used for the reduction from rectilinear planar monotone 3-SAT

Fig. 5. The way in which we connect the three semi-rigid chains from Fig. 4. Once the locations of the top and bottom chains are fixed, we know that clause gadgets will protrude upwards from the top chain and downwards from the bottom chain. So the only possible location for the middle chain is between the two other chains.

which are between 1 and n, where n is the length of the original chain. We then scale up our semi-rigid chain by a factor of 6. Our goal is to replace the rigidity constraints of the original chain with some new structure.

Say that points $v_{i-3}, v_{i-2}, v_{i-1}$, and v_i all lie within the same rigid piece. If we have the locations of v_{i-1} and v_{i-2}, then there are two possible locations for v_i. To determine which location is correct, it is sufficient to know whether v_{i-3} lies above or below the line between v_{i-1} and v_{i-2}. Hence, to impose the rigidity constraints, it is sufficient to create two types of local gadgets: one gadget which can only be flattened if v_{i-3} and v_i lie on the same side of the line between v_{i-1} and v_{i-2}; and one gadget which can only be flattened if v_{i-3} and v_i lie on different sides of the line between v_{i-1} and v_{i-2}. Those gadgets are depicted in Fig. 6. We attach each gadget halfway down the edge between v_{i-2} and v_{i-1}.

Any flat state of the fixed-angle tree can be converted to a flat state of the original semi-rigid chain by removing the new gadget edges. Each gadget can be thought of as thickening the edge between v_{i-1} and v_{i-2}, because the gadgets can lie above or below the edge. The original edges were infinitely thin; the new edges have thickness 4. Because each edge was scaled up by a factor of 6, any non-crossing flat state of the original semi-rigid chain will not become self-intersecting when the gadgets are added. Therefore, any flat state of the original semi-rigid chain is a flat state of the fixed-angle tree we have created.   □
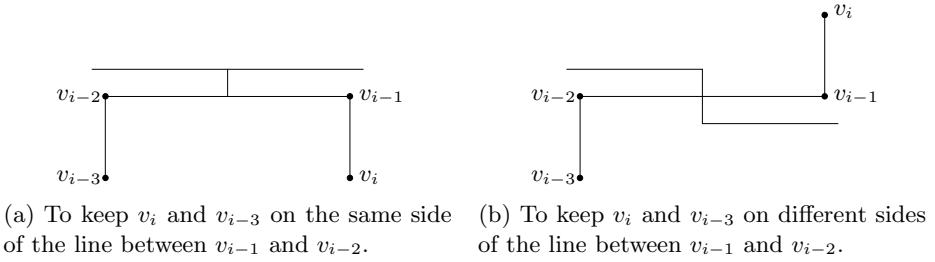
(a) To keep v_i and v_{i-3} on the same side of the line between v_{i-1} and v_{i-2}.

(b) To keep v_i and v_{i-3} on different sides of the line between v_{i-1} and v_{i-2}.

Fig. 6. The gadgets used in Theorem 3

(a) Zig-zag gadget

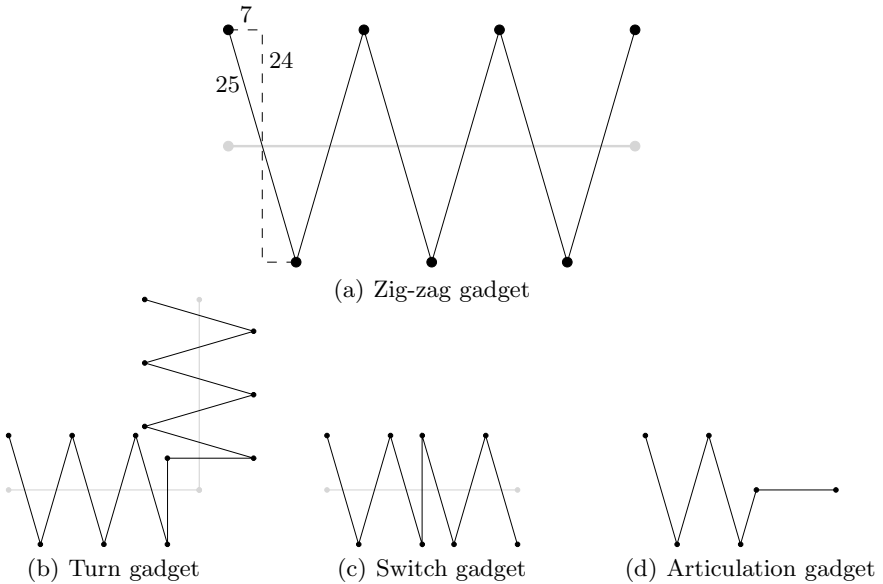(b) Turn gadget   (c) Switch gadget   (d) Articulation gadget

Fig. 7. The gadgets used for the proof of Theorem 4

Theorem 4. *There exists a polynomial-time algorithm which takes as input an orthogonal equilateral semi-rigid chain, and outputs an equilateral fixed-angle chain such that each flat state of one chain corresponds to a flat state of the other chain, and the spans differ by a fixed constant factor c.*

Proof. We begin by replacing each edge in the original semi-rigid chain with three edges connected with a fixed angle of 180°. Next, we introduce several types of gadgets, each of which can be used to replace a section of the semi-rigid chain. The first gadget is used to replace any interior edge in a rigid piece which has fixed-angle 180° with the edges on either side. The gadget we use will zig-zag across the original location of the edge, as depicted in Fig. 7(a). Each edge in the depicted gadget has length 50, so we can consider each such edge to be a sequence of 50 smaller equilateral edges.

The second gadget used is known as the turn gadget, which is depicted in Fig. 7(b). It is used to cause the zig-zag to turn by a total of 90°. The depicted flat state for the turn gadget is the only possible flat state, barring reflection of the whole gadget. If the turn gadget is connected to a zig-zag with a fixed-angle of $2\arcsin(7/25)$, then the direction that the zig-zag goes in (that is, whether the final point in the zig-zag is up or down) determines the direction of rotation for the turn gadget.

If we were to use only zig-zags and turn gadgets, the result would be a spiral, because each turn gadget would cause a rotation in the same direction. So in order to allow us to switch directions, we use the gadget depicted in Fig. 7(c), which is known as a switch gadget. When a switch gadget is used, it changes the direction of the zig-zag. This means that when the next turn gadget is used, the turn will go in the opposite direction to previous turns. Because we scaled up the original chain, there will always be room to place a switcher gadget between adjacent turns.

Together, these three gadgets ensure that if the first three points in our chain are fixed, then there is only one way to arrange the rest of the chain. This lets us enforce the rigidity constraints for each rigid piece of the original semi-rigid chain. To join these rigid pieces together, we use the articulation gadget depicted in Fig. 7(d). The articulation gadget has only one possible flat state, barring reflection. It is used to replace the edges adjacent to an articulation point. The right half of the articulation gadget lies in the same location as the end of the replaced edge. Therefore, when we apply the fixed-angle constraint from the original semi-rigid chain, it places the correct restriction on the angle between the two rigid pieces. In addition, the use of this gadget for each articulation point (including the ends of the original chain) means that when we transform a flat state of the original semi-rigid chain to a flat state of the new fixed-angle chain, the distances between articulation points will be scaled up by a constant factor.

Each of these gadgets replaces a single edge of length 1 with a gadget whose flat state has length 84 and width 48. Just as in Theorem 3, the fact that we scaled up the original chain by a factor of 3 means that the substitution of these gadgets for the original edges of the tree will not create intersections.   □

In 2002, Aloupis et al. showed that every fixed-angle chain with angles between 90° and 180° has a canonical flat state [1]. In 2006, Aloupis and Meijer showed that every equilateral fixed-angle chain with angles strictly between 60° and 150° has a canonical flat state [2]. We have shown that it is NP-hard to compute a flat state for some equilateral fixed-angle chains with angles between 16.26° and 180°. This naturally leads to the question of how large the minimum angle can be while still ensuring that flattening is NP-hard. In our next result, we show that it is NP-hard to compute a flat state for some fixed-angle chains with angles between $\theta < 60°$ and 180°. This result does not use equilateral chains, but all edges used in this reduction have length $\Theta(1)$.

Theorem 5. *Given any constant $\theta < 60°$, there exists a polynomial-time algorithm which takes as input an orthogonal equilateral semi-rigid chain, and*

(a) For the min span problem.
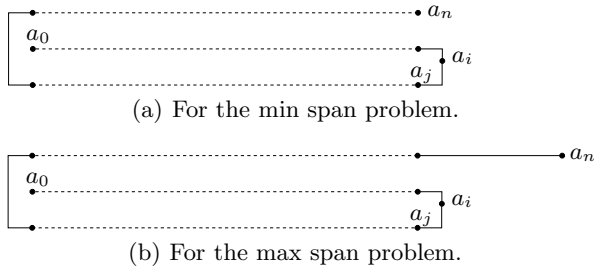

(b) For the max span problem.

Fig. 8. How to connect the three semi-rigid chains from Theorem 2 when reducing to the minimum and maximum span problems. Note the new articulation point a_i.

outputs a fixed-angle chain with minimum angle $\geq \theta$ that can be flattened if and only if the semi-rigid chain can be flattened.

Proof. Just as in Theorem 4, we will use four different gadgets: a zig-zag, a turn gadget, a switcher, and an articulation gadget. Although most of the gadgets are far more complex than their equilateral equivalents, their size depends only on θ. The gadget we use to replace a single edge in the original semi-rigid chain is a zig-zag with edge length 1 and fixed angle θ between each piece. To construct a turn gadget capable of turning $90°$, we construct a turn gadget capable of turning α degrees, and chain multiple gadgets together to produce a turn of $90°$. The switcher gadget involves a zigzag with edge lengths $x < 1$ and fixed angle θ. The articulation gadget is constructed in a way similar to the turn gadget, but scaled down. These gadgets are combined as in Theorem 4 to produce the desired result. □

5   Flat Span

In this section, we adapt the proof of Theorem 2 to show the NP-hardness of the related problems of minimum and maximum flat span.

Theorem 6. *There exists a polynomial-time algorithm which takes as input a rectilinear planar monotone 3-SAT instance ϕ, and outputs an equilateral fixed-angle chain and a distance d such that the minimum span of the chain in any flat state is less than d if and only if ϕ is satisfiable.*

Proof. In Theorem 2, we saw a reduction that involved constructing three separate chains and connecting them as in Fig. 5. For this reduction, we connect the same three chains as depicted in Fig. 8(a). In the depicted flat state, which is non-crossing if and only if ϕ is satisfiable, a_0 is at $(0, 0)$, a_i is at $(w + 2, -1)$, and a_n is at $(w, 3)$, so the span is $\sqrt{w^2 + 9}$. There are two other flat states, both of which can be made non-crossing regardless of whether ϕ is satisfiable. In the first flat state, we reflect the middle chain over the articulation point a_i, which moves a_0 to coordinates $(2w + 4, 0)$. In the second flat state, we flip the

middle chain over the articulation point a_i and then over the articulation point a_j, which moves a_0 to coordinates $(2w + 4, -6)$. The span of either flat state will be $> \sqrt{w^2 + 9}$. By applying Theorem 4, we get an equilateral chain whose minimum span depends on the satisfiability of ϕ.   □

Theorem 7. *There exists a polynomial-time algorithm which takes as input a rectilinear planar monotone 3-SAT instance ϕ, and outputs an equilateral fixed-angle chain and a distance d such that the maximum span of the chain in any flat state is greater than d if and only if ϕ is satisfiable.*

Proof. The argument is similar to Theorem 6, but with the three chains from Theorem 2 arranged as depicted in Fig. 8(b), so that a_n is at $(w + 12, 3)$.   □

References

1. Aloupis, G., Demaine, E.D., Vida, E.D., Erickson, J., Langerman, S., Meijer, H., O'Rourke, J., Overmars, M.H., Soss, M.A., Streinu, I., Toussaint, G.T.: Flat-state connectivity of linkages under dihedral motions. In: Bose, P., Morin, P. (eds.) ISAAC 2002. LNCS, vol. 2518, pp. 369–380. Springer, Heidelberg (2002)
2. Aloupis, G., Meijer, H.: Reconfiguring planar dihedral chains. In: European Conference on Computational Geometry (March 2006)
3. de Berg, M., Khosravi, A.: Optimal binary space partitions in the plane. In: Thai, M.T., Sahni, S. (eds.) COCOON 2010. LNCS, vol. 6196, pp. 216–225. Springer, Heidelberg (2010)
4. Demaine, E.D., O'Rourke, J.: Fixed-Angle Linkages. Cambridge University Press, Cambridge (2007)
5. Gillespie, R.J., Popelier, P.L.A.: Molecular geometry and the vsepr model. In: Chemical Bonding and Molecular Geometry: From Lewis to Electron Densities, ch.4. Oxford University Press (2001)
6. Knuth, D.E., Raghunathan, A.: The problem of compatible representatives. SIAM J. Discrete Math. 5(3), 422–427 (1992)
7. Lichtenstein, D.: Planar formulae and their uses. SIAM J. Comput. 11(2), 329–343 (1982)
8. Soss, M.: Geometric and Computational Aspects of Molecular Reconfiguration. PhD thesis. School of Computer Science, McGill University (2001)
9. Soss, M., Toussaint, G.T.: Geometric and computational aspects of polymer reconfiguration. Journal of Mathematical Chemistry 27(4), 303–318 (2000)

An $O(n \log n)$ Algorithm for a
Load Balancing Problem on Paths

Nikhil R. Devanur[1] and Uriel Feige[2,*]

[1] Microsoft Research, Redmond, WA
nikdev@microsoft.com
[2] Weizmann Institute of Science, Rehovot, Israel
uriel.feige@weizmann.ac.il

Abstract. We study the following load balancing problem on paths (PB). There is a path containing n vertices. Every vertex i has an initial load h_i, and every edge $(j, j+1)$ has an initial load w_j that it needs to distribute among the two vertices that are its endpoints. The goal is to distribute the load of the edges over the vertices in a way that will make the loads of the vertices as balanced as possible (formally, minimizing the sum of squares of loads of the vertices). This problem can be solved in polynomial time, e.g, by dynamic programming. We present an algorithm that solves this problem in time $O(n \log n)$.

As a mental aide in the design of our algorithm, we first design a hydraulic apparatus composed of bins (representing vertices), tubes (representing edges) that are connected between bins, cylinders within the tubes that constrain the flow of water, and valves that can close the connections between bins and tubes. Water may be poured into the various bins, to levels that correspond to the initial loads in the input to the PB problem. When all valves are opened, the water flows between bins (to the extent that is feasible due to the cylinders) and stabilizes at levels that are the correct output to the respective PB problem. Our algorithm is based on a fast simulation of the behavior of this hydraulic apparatus, when valves are opened one by one.

1 Introduction

We describe a problem that we shall call *Path Balancing* (PB).

An instance of PB is a path on n vertices. Every vertex v_i has an initial height $0 \leq h_i \leq 1$. Every edge $e_j = (v_j, v_j + 1)$ has weight $0 \leq w_j \leq 1$. A feasible solution splits the weight of every edge in an arbitrary way between its endpoints, thus contributing to the heights of its endpoints. The goal is to make the vector of heights as balanced as possible. (Here and elsewhere, heights, in contrast to initial heights, will refer to the heights of vertices in a solution and not in the input.) In a perfectly balanced solution all heights are identical. When

there is no perfectly balanced solution, the notion of balance that we use is that of minimizing the sum of squares of the heights.

The problem above can be formulated as a convex program as follows. For $1 \leq i \leq n-1$, let x_i denote the amount of weight that edge e_i gives to vertex v_i. The rest of the weight of e_i which is $w_i - x_i$ is given to vertex v_{i+1}. Then there are $n-1$ constraints of the form $0 \leq x_i \leq w_i$, and the objective function is to minimize $(h_1 + x_1)^2 + \sum_{i=2}^{n-1}(h_i + (w_{i-1} - x_{i-1}) + x_i)^2 + (h_n + (w_{n-1} - x_{n-1}))^2$. For simplicity of notation, we shall introduce fictitious $x_n = 0$, $w_0 = 0$ and $x_0 = 0$, and let $h(v_i)$ denote the value of $h_i + (w_{i-1} - x_{i-1}) + x_i$. Hence the objective function can be written as $\sum_{i=1}^{n}(h(v_i))^2$. (Actually, the optimal solution with the objective being any convex function of the $h(v_i)$'s will turn out to be the same, we just choose $h(v_i)^2$ for convenience.)

We are interested in efficient algorithms for PB. Since it can be formulated as a convex program, it follows that it can be solved in polynomial time. In fact, a natural dynamic programming approach gives a running time of $O(n^3)$ and with some effort, one can obtain an algorithm that runs in time $O(n^2)$. (The full version of the paper will contain a description and analysis of these algorithms.) In this paper we show that this problem can be solved in $O(n \log n)$ time. In measuring the running time of algorithms we shall count the number of basic operations that they perform, without worrying too much about the cost of each operation (e.g., the cost of basic arithmetic operations as a function of the precision needed), or about the data structures that are needed in order to implement the algorithms efficiently. Our assumption is that these issues can be addressed while imposing only acceptable overhead on the algorithms.

Besides being a natural problem, the hope is that such an algorithm for this problem might be useful in finding fast algorithms for computing maximum cardinality matchings in bipartite graphs. The fastest algorithms known for this problem are by Hopcroft and Karp [HK71, HK73] (time $O(m\sqrt{n})$), by Ibarra and Moran [IM81] (time $O(n^\omega)$)[1] and by Feder and Motwani [FM95] (time $O(m\sqrt{n} \log_n(n^2/m))$). Recently, Goel et. al. [GKK09] gave an $O(n \log n)$ algorithm to find a perfect matching in regular bipartite graphs. (The case of regular bipartite graphs is easier; $O(m)$ time algorithm was known earlier [COS01]). One approach to solve the matching problem is an interior point approach, which searches through fractional matchings, updating them in each step. Our problem can be thought of as an analog of updating along an augmenting path for fractional matchings. A vertex i in our problem corresponds to a vertex i on one side (say L) of the bipartite graph. The edge i in our problem corresponds to a vertex i' on the other side (say R). Each vertex $i' \in R$ is adjacent to the vertices i and $i + 1 \in L$. h_i corresponds to the total amount of edges matched to i from vertices other than i' and $(i-1)' \in R$. w_i corresponds to 1 minus the total amount of edges matched to i' from vertices other than i and $i + 1 \in L$.

The PB problem is also a special case of a *power-minimizing scheduling problem* that has been well studied [YDS95, LY05, LYY06]: suppose that there are n jobs to be scheduled on a single machine. Each job has an arrival time, a

---

[1] ω is the exponent in the matrix multiplication algorithm.

deadline, and needs some amount of CPU cycles. The machine can be run at different speeds, if it is run at speed s then it can supply s CPU cycles per unit time, and consumes a power of s^2 per unit time. (Again, it could be any convex function of s.) The goal is to schedule the jobs on the machine and determine the speeds so as to minimize the total power consumed. Li, Yao and Yao [LYY06] gave the fastest known algorithm for this problem that runs in time $O(n^2 \log n)$. Designing an $O(n \log n)$ time algorithm is an open problem. The PB problem is the following special case: there are n jobs with [arrival time, deadline] $= [i, i+1]$ which require h_i CPU cycles, for $i = 1$ to n. There are $n - 1$ jobs with [arrival time, deadline] $= [i, i+2]$ which require w_i CPU cycles, for $i = 1$ to $n - 1$.

The design of our algorithm is aided by physical intuition. We first design a hydraulic apparatus that may serve as an analog (rather than digital) computing device that solves the PB problem. Thereafter, we design an efficient algorithm that quickly simulates the operation of the hydraulic apparatus.

2   Preliminaries

Proposition 1. *The optimal solution is unique.*

The proof of the proposition, and all others in this section are in the full version of the paper.

Proposition 2. *There is a linear time algorithm for checking if there is a perfectly balanced solution.*

When there is no perfectly balanced solution, we provide a structural characterization of the unique optimal solution.

Definition 1. *A solution is said to have a* block structure *(BS) if it can be partitioned into blocks in which each block is a consecutive set of vertices that have the same height, and every edge between two adjacent blocks allocates all its weight to the vertex of lower height (and hence is said to be* oriented *towards that node).*

Lemma 1. *For any PB problem, there is a unique solution with a block structure.*

Lemma 2. *A solution is optimal if and only if it has a block structure.*

It follows that to solve the PB problem it suffices to find a BS solution.

2.1   Hydraulic Apparatus

Our goal is to design more efficient algorithms for the PB problem. But before that, we describe a hydraulic apparatus that solves the PB problem (See Figure 1). The apparatus is constructed from a row of n identical bins arranged from left to right, where each bin has base area 1 square unit and height 4 units.

Every two adjacent bins are connected by a horizontal cylindrical tube of base area 1 square unit and length one unit. Inside the tube there is a solid cylinder that exactly fits the width of the tube (no water can flow around it) and has width $(1 - w_j)$ for tube e_j. (It would be desirable to have solid cylinders whose width can be varied so as to encode different instances of the PB problem, but the physical design of such cylinders is beyond the scope of this manuscript). The openings between the tube and each of the adjacent bins have smaller diameter than the tube, and hence the cylinder cannot extend out of the tube. There is a valve between every tube and the bin to the left of it.
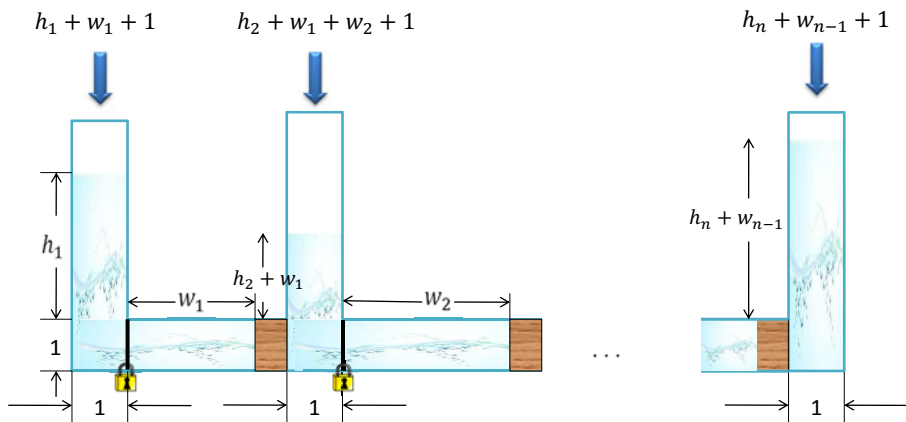

Fig. 1. Illustration of the Hydraulic Apparatus

To input the initial conditions of the PB problem, first one shuts all valves. Then, iteratively for i from 1 up to $n-1$, one opens valve i, pours $(h_i+w_{i-1}+w_i+1)$ cube units of water into bin i (that now fill the tube to the right (ensuring this is the reason for the $+1$ term in the volume of water) pushing the cylinder all the way to the right), and closes valve i (closing valve i is not strictly necessary, but helps understand the algorithms that will follow). For bin n there is no valve to open, and one simply pours into it $(h_n + w_{n-1} + 1)$ cubic units of water. Observe that the initial condition corresponds to the case that the vertex to the right of an edge gets all the weight of the edge (the bin to the left of a tube also gets a volume of water corresponding to the weight of the corresponding edge, but this volume is spent on filling the tube). Now one opens all the valves. As a result, some of the cylinders may drift towards the left in their tubes (to an extent that depends on the relative water levels of adjacent bins). This corresponds to the situation where the corresponding edge allocates part (or all) of its weight to the left. The water levels when the system stabilizes (minus 1) are the solution to the PB problem.

Our algorithm is obtained by simulating (quickly) the action of the hydraulic apparatus. Our algorithm will be *monotone* in the sense that in the mathematical program, the variables x_i are initially all set to 0, and in every step of the

algorithm can only be raised. (This corresponds to cylinders only drifting to the left and never to the right.)

Every edge will be in one of three states:

- *closed.* This corresponds to the situation when the valve of the corresponding tube is closed. All the edge weight has to be allocated to the right. Equivalently, the corresponding variable x_i is set to 0. At this point, the PB problem is broken into two independent subproblems, one to the left and one to the right of the edge.
- *open.* This corresponds to the situation when the valve is open. The weight of an edge may be distributed in an arbitrary way (including still allocating all the weight to the right). Once an edge is open, it is never closed again. Also, an edge is open unless it is blocked, which is the next state.
- *blocked.* This corresponds to the situation that all the weight of the edge is allocated to the left. Equivalently, the corresponding variable x_i is set to w_i. For the hydraulic apparatus, this means that the cylinder drifted all the way to the left of the tube. Since our algorithms will be monotone, once an edge is blocked it will never become unblocked again. Hence again, the PB problem is broken into two independent subproblems, one to the left and one to the right of the edge. However, since the edge will never reopen, the subproblems remain independent until the algorithm ends.

Having introduced the notion of closed edges, we extend the notion of block structure to that of *constrained block structure* (CBS). Here, some edges may be designated as being closed, and the PB problem is broken into independent subproblems separated by the closed edges, and one seeks BS solution for every subproblem. In particular, the initial state of the hydraulic apparatus corresponds to a CBS with all edges closed, and the final solution is a CBS with no edge closed. Given the set of closed edges, there is a unique corresponding CBS.

3   An $O(n \log n)$ Algorithm

Our algorithm for PB will go through a sequence of CBS's. Initially, all edges are closed. At every step one more edge is opened, and the corresponding CBS is found. Eventually, all edges become open and the final BS is found. We now focus on the opening of one edge.

Opening one edge at a time. When a valve is opened, the water in the hydraulic apparatus re-adjusts itself to get to a stable point. We refer to this process as one *round*. Suppose valve i is opened to begin a round. If at this point $h(v_i) \geq h(v_{i+1})$ then the system is already in a stable situation, and the old CBS is also the new CBS. Otherwise, cylinder i moves to the left until it comes to rest because either the heights of the vertices v_i and v_{i+1} become the same, or edge e_i becomes blocked. During this process we track the instantaneous block structure (IBS) of the system: this is the block structure defined by the instantaneous heights of the bins where consecutive bins with the same height belong to the same

block. An IBS satisfies all the conditions of a CBS, except at the newly opened edge. As the cylinder i moves farther to the left, the IBS goes through a sequence of changes, and the IBS when the cylinder i comes to rest is the new CBS. We now identify the (only) two types of events that change the IBS.

Type 1 Events: Consider an edge e_j which has been opened prior to the round, but remained oriented to the right. That is, all its weight is allocated to the vertex v_{j+1} and x_j is set to 0. This is because prior to the round, $h(v_j) > h(v_{j+1})$. If at some point during the round the heights become the same, then cylinder j starts to move to the left, and the edge e_j is no longer being oriented. At this point the IBS changes: the blocks on either side of e_j merge to become a single block. For such an event we also say that an edge *starts to move*.

Type 2 Events: The other type of event is when an edge becomes blocked. Again the IBS changes: the block containing the edge is split into two blocks on either side of it.

Opening the rightmost edge. We now consider a special case, suppose that we have the CBS where all edges except e_{n-1} are open. We then open edge e_{n-1} and find the new CBS (which will be the BS solution). We use an algorithm for this special case as a subroutine to design an algorithm for the PB problem. For now we prove the following theorem which guarantees a fast algorithm for this case.

Theorem 1. *Given the CBS solution with all edges but e_{n-1} open, the BS can be found in $O(n)$ time.*

First, we present two lemmas that describe how the IBS changes when we open the edge e_{n-1}. For the discussion that follows, we introduce a notion of time t. t is set to 0 when the round begins. We assume that the cylinder $n-1$ moves to the left at unit speed and calculate all other values as a function of t. We denote the speed at which cylinder i moves by $\frac{dx_i}{dt}$. We will also be interested in the height of the block containing vertex v_{n-1} and denote it by h. We denote the speed with which h increases by $\frac{dh}{dt}$.

Lemma 3. *Let the block containing v_{n-1} be $[j, n-1]$.*

1. $\frac{dh}{dt} = \frac{1}{n-j}$. *The time at which the edge e_{j-1} starts to move, if no other event happens earlier, is $(n-j)(h(v_{j-1}) - h(v_j))$.*
2. $\frac{dx_i}{dt} = \frac{i-j+1}{n-j}$. *The time at which edge e_i becomes blocked, if no other event happens earlier, is $(w_i - x_i)(n-j)/(i-j+1)$.*

Proof. 1. Water that flows into v_{n-1} at unit rate is distributed equally among all the $n-j$ vertices in the block $[j, n-1]$. Edge e_{j-1} starts to move when $\Delta h = h(v_{j-1}) - h(v_j)$, that is, when $\Delta t = (n-j)(h(v_{j-1}) - h(v_j))$.
2. There are $i - j + 1$ vertices in the block to the left of e_i, each of which accumulates water at rate $\frac{1}{n-j}$. The time at which edge i becomes blocked is precisely when $\Delta x_i = w_i - x_i$.

Lemma 4. *The events happen in the following order: Type 1 events happen from right to left (decreasing order), and after all such events, Type 2 events happen from left to right (increasing order).*

Proof. Let the current block containing v_{n-1} be $[j, n-1]$. Clearly, all the edges that started to move in this round lie in the current block, and the only edge that can start to move next is e_{j-1}. Also, if the last event was an edge blocking event, then it must have been the edge e_{j-1}. In this case any subsequent event does not effect the vertices in $[1, j-1]$. Therefore the only subsequent events that can happen are edges becoming blocked in $[j, n-1]$. (Or n joins the block $[j, n-1]$ and the system stabilizes.) Thus, if the events upto some time follow the given order, then the next event also follows the same order. The proof follows by induction on the sequence of events.

The algorithm computes the sequence of events that happen and other relevant information such as the heights of the blocks when these events happen, and then the eventual BS. The block structure is represented using an array. The ith element of the array contains information about the vertex i, whether it is the left end of a block, the right end (or both), or in the middle of the block. If it is the left end, then the position of the right end of the block is stored, and vice versa if it is the right end. The height of the block is stored at both the ends. Finally, for a vertex that is at an end of the block, we also store whether the adjacent edge is closed.

Given a CBS, we compute the solution (x_i values) that respects the CBS. It is easy to see that this can be done in $O(n)$ time.

Our algorithm proving Theorem 1 is composed of three procedures, where each procedure makes gradual progress towards the solution. Procedure 1 assumes a simplified version of the problem in which Type 2 events (blocking events) are assumed not to happen. Hence only Type 1 events (edges starting to move) happen, and the order of them happening is from right to left. The output of Procedure 1 is a tentative sequence $O_1, O_2, \ldots O_{n_1}$ of Type 1 events in the order in which they happen. For each event O_k, we store the edge j_k that started to move, the time t_k at which it happened, and the height $\hat{h}_k$ of the block at that time. We also store the total number of such events, n_1. Procedure 2 removes a suffix of the tentative sequence $O_1, O_2, \ldots O_{n_1}$, leaving a prefix that contains only those Type 1 events that actually do happen. To do this, one considers potential Type 2 events from left to right, and checks whether they would have prevented a Type 1 event to the left of them. If so, the respective Type 1 event is removed from the tentative sequence. Even though Procedure 2 considers potential Type 2 events, its only goal is to gather sufficient information about Type 2 events so as to be able to determine the correct sequence of Type 1 events. In particular, potential Type 2 events that are deemed irrelevant to this goal are not considered by Procedure 2. The task of determining the correct sequence of Type 2 events is left to Procedure 3, which is called only after the correct sequence of Type 1 events was determined.

Procedure 1. Find Type 1 events

- $k = 1$, $t_0 = 0$.
- $j = $ `the left end of the block whose right end is at` $n-1$.
- `While` $j > 1$ `and edge` $j-1$ `is not blocked, do`
  - $j_k = j - 1$ /* The next edge that opens is immediately to the left of j */
  - $t_k = t_{k-1} + (n-j)(h(v_{j-1}) - h(v_j))$, $\hat{h}_k = h(v_{j-1})$.
  - /* Move to the next block to the left */
  - $j = $ `the left end of the block whose right end is at` $j-1$.
  - $k = k + 1$.
- $n_1 = k - 1$.

Lemma 5. *Procedure 1 runs in $O(n)$ time.*

We now describe Procedure 2. Let $t = t_{n_1}$ be the time at which the last Type 1 event happens (according to the output of Procedure 1). We start with $i = j_{n_1} + 1$ and see if edge e_i becomes blocked before time t. If not, then we move to the edge to the right (by setting $i = i+1$) and continue. If e_i does become blocked before t, we update t to be the time at which the previous Type 1 event happened (set $n_1 = n_1 - 1$, and $t = t_{n_1}$). If i is still to the right of the new j_{n_1} (since $j_k < j_{k-1}$ for all k), we continue with the same i, otherwise we set $i = j_{n_1} + 1$. We end when $i = n$.

One difficulty here is that we need to determine if e_i becomes blocked by time t in $O(1)$ steps. Let $\Delta x_i(t)$ be the distance traveled by cylinder i at time t (in the current round). Then e_i is blocked by time t iff $\Delta x_i(t) \geq w_i - x_i$. Note that $\Delta x_i(t) = \Delta x_{i-1}(t) + $ the increase in the height of v_i at time t. This increase in height is $(\hat{h}_{n_1} - h(v_i))$. So we can iteratively compute $\Delta x_i(t)$ in $O(1)$ steps.

This gives rise to another difficulty, if t changes then we might have to go back and recompute Δx_i starting from $i = j_{n_1} + 1$. This might make the procedure run in quadratic time. We get around this by using the observation that $\Delta x_i(t_k) - \Delta x_i(t_{k-1}) = $ the distance traveled by cylinder i in the time interval $[t_{k-1}, t_k]$, which by Lemma 3 is equal to

$$\frac{(i - j_k + 1)(t_k - t_{k-1})}{n - j_k}.$$

Thus when we update t, we can also update $\Delta x_i(t)$ in $O(1)$ steps and continue with the same i.

Procedure 2. Eliminate Type 1 events

- $i = j_{n_1} + 1$, $\Delta x_i = \hat{h}_{n_1} - h(v_i)$.
- `While` $i < n$ `and` $n_1 > 0$ `do`
  - `If` $\Delta x_i > w_i - x_i$ `then` /* Edge i would prevent edge j_{n_1} from opening */
    - * $n_1 = n_1 - 1$.
    - * `If` $i > j_{n_1}$ `then` /* i is still to the right of the new j_{n_1} */

$$\cdot \ \Delta x_i = \Delta x_i - \frac{(i - j_{n_1} + 1)(t_{n_1+1} - t_{n_1})}{n - j_{n_1} + 1}.$$

* Else

$$\cdot \ i = j_{n_1} + 1, \ \Delta x_i = \hat{h}_{n_1} - h(v_i).$$

• Else

* $i = i + 1$.

* $\Delta x_i = \Delta x_{i-1} + \hat{h}_{n_1} - h(v_i)$.

Lemma 6. *Procedure 2 runs in $O(n)$ time.*

We now describe Procedure 3 that computes the sequence of Type 2 events, and the times at which these happen. Note that the time at which an edge could potentially become blocked depends on all the events that happen prior to that, since each event changes the speed at which the cylinder moves. In particular, it depends on when any of the edges to the left become blocked. In addition, whether an edge ever becomes blocked depends on the Type 2 (blocking) events that happen to the right of the edge. Thus the dependencies go both ways and resolving these dependencies is a challenge. A naive algorithm that attempts to iteratively find the next event in the sequence takes $O(n^2)$ time, whereas our goal is to compute the entire sequence in $O(n)$ time. To do so we build the sequence of Type 2 events from left to right. We will borrow an approach that we used for constructing the sequence of Type 1 events, which was to first build the sequence under a simplifying assumption that certain blocking events do not happen, and then correct for the fact that they do happen. For Type 1 events, this construction took two stages, Procedure 1 and Procedure 2. For Type 2 events, we have only one stage, Procedure 3, but this procedure takes many rounds. Procedure 3 scans edges from left to right, and at every round it considers one more edge. It assumes that no blocking event happens to the right of the edge currently scanned. This implies that this edge (say edge e_i) necessarily eventually becomes blocked and is tentatively added to the sequence of blocking events. At this time we do another scan from right to left of the tentative sequence of the Type 2 events we have constructed so far to determine which ones can be removed because e_i is blocked earlier to them. In fact this is necessary to determine the exact time at which e_i becomes blocked. This nested loop hints at a quadratic running time, but we show that the time is indeed $O(n)$ based on the observation that once an event is removed from the sequence it is never returned.

First of all, before we proceed further, we update the x_i values upto time $\tau_0 = t_{n_1}$, the time of the last Type 1 event. Note that at this point all the heights that have changed are in $[j_{n_1} + 1, n - 1]$, and they are all equal to $\hat{h}_{n_1}$. It is easy to see that this update can be done in $O(n)$ time.

Our algorithm builds the following iteratively, starting with the left most edge and moving right: the sequence of Type 2 events, ending at e_i becoming blocked, assuming no events happen to the right of e_i. The sequence of events, say $E_1, E_2, \ldots, E_{n_2}$ in the order in which they happen, is maintained as an array. For each event E_k in the sequence, we store the corresponding edge that was blocked, say i_k, the time at which the event happened, say τ_k, and the increase

in the height of the block when that event happened, say Δh_k. The total number of such events n_2 is also maintained. Also for the sake of convenience, set i_0 to be the edge at the left end of the block containing j_{n_1}, that is $[i_0 + 1, j_{n_1}]$ is a block. The time τ_0 as mentioned earlier is set to t_{n_1}. Δh_0 is set to 0.

At the beginning of the ith iteration, we have the sequence of events upto $i-1$, that is the last event is e_{i-1} becoming blocked. In the ith iteration, we check if e_i becomes blocked before e_{i-1}. If not then we insert e_i after e_{i-1} and proceed to the next iteration. Otherwise, we iteratively consider the previous event in the sequence and do the same, until we either find an event that happens before e_i is blocked, or we eliminate all events in the sequence in which case e_i will be the only event in the new sequence.

We now show how to determine whether e_i becomes blocked before E_k or not. As before, let $\Delta x_i(\tau_k)$ be the distance moved by cylinder i at time τ_k. Let $j = i_k$ be the edge that was blocked during event E_k. Note that

$$\Delta x_i(\tau_k) = \Delta x_j(\tau_k) + (i - j)\Delta h_k.$$

This is because, the distance moved by cylinder i is equal to the distance moved by cylinder j plus the water that accumulated at the vertices in the range $[j+1, i]$. Further we know that $\Delta x_j(\tau_k) = w_j - x_j$ since e_j became blocked at τ_k. The exception is when $k = 0$ in which case $\Delta x_i(\tau_k) = 0$. Thus we can determine if $\Delta x_i(\tau_k) \geq w_i - x_i$, which gives us the required answer.

Finally, once we have determined the right position k, we update E_{k+1} to be the event that e_i becomes blocked. We set $i_{k+1} = i$ and let $j = i_k$. The time τ_{k+1} is given by

$$w_i - x_i = \Delta x_i(\tau_{k+1}) = \Delta x_i(\tau_k) + \frac{i - j}{n - j}(\tau_{k+1} - \tau_k),$$

from which one gets

$$\tau_{k+1} = (w_i - x_i - \Delta x_i(\tau_k))\frac{n - j}{i - j} + \tau_k.$$

$\Delta h_{k+1} = \Delta h_k + \frac{1}{n-j}(\tau_{k+1} - \tau_k)$. Also n_2 is set to $k + 1$.

Procedure 3. Find Type 2 events

- $n_2 = 0$.
- For $i = i_0 + 1$ to $n - 1$ do /* When is edge e_i blocked? */
  - $k = n_2$, $j = i_k$.
  - If $k \neq 0$, then $\Delta x_i = w_j - x_j + (i - j)\Delta h_k$, Else $\Delta x_i = 0$.
  - While $k > 0$ and $\Delta x_i > w_i - x_i$, /* e_i is blocked before E_k */
    * $k = k - 1$, $j = i_k$.
    * If $k \neq 0$, then $\Delta x_i = w_j - x_j + (i - j)\Delta h_k$, Else $\Delta x_i = 0$.
  - /* Insert e_i being blocked as the event E_{k+1} */
  - $i_{k+1} = i$.
  - $\tau_{k+1} = (w_i - x_i - \Delta x_i)\frac{n-j}{i-j} + \tau_k$.

- $\Delta h_{k+1} = \Delta h_k + \frac{1}{n-j}(\tau_{k+1} - \tau_k)$.
- $n_2 = k + 1$.

Lemma 7. *Procedure 3 runs in time $O(n)$ time.*

Proof. Naively, each time the inner (While) loop for Procedure 3 might go from n_2 to 0 and this would give an n^2 bound. However, note that each iteration of the inner While loop eliminates an edge blocking event, and every such event can be eliminated only once. Thus there can be only $O(n)$ iterations of the inner loop overall and hence the running time of this procedure is $O(n)$.

Procedure 3 finds all the Type 2 events upto edge $n - 1$, assuming that nothing happens to the right of $n - 1$. That is, Procedure 3 ignores the possibility that the heights of v_{n-1} and v_n might become the same and the round ends due to that. Therefore we next compute at what time the heights become equal and determine if some events need to be eliminated because of that. The height of v_n at time t is simply $h(v_n) - t$. The height of v_{n-1} however depends on the sequence of events that happen. Recall that for the Type 1 events, we actually stored the height of v_{n-1} at each t_k, which was $\hat{h}_k$. So for every k from 1 to n_1, we can compare the heights of v_{n-1} and v_n and see if they cross over, that is at time t_k, the height of v_{n-1} is smaller than that of v_n but at time t_{k+1} it is larger. In that case, we set $n_1 = k$, and $n_2 = 0$. If the heights never cross over during Type 1 events, we then move on to Type 2 events. Once again, we stored the *height increment* of v_{n-1} at each τ_k, which was Δh_k. Therefore as before we can compare the heights of the two vertices at time τ_k for every k from 1 to n_2 and see if they cross over. If they do cross over at k, then we set $n_2 = k$.

Finally, once we have determined the entire sequence of events in a round, we can update the block structure and the new x_i values. Suppose first that the round ended with e_{n-1} becoming blocked. In this case the new blocks are $[i_0 + 1, i_1], [i_1 + 1, i_2], \ldots, [i_{n_2-1} + 1, n - 1]$. Everything to the left of i_0, that is everything in the range $[1, i_0]$ remains unchanged. We also know the heights of each of these blocks, so finding the new x_i values is easy. In case the round ended with the heights of v_{n-1} and v_n becoming equal, then everything is as before, except that the last block is $[i_{n_2} + 1, n]$. It is easy to see that everything after Procedure 3 can be done in $O(n)$ time. This completes the proof of Theorem 1.

Divide and Conquer. We now show how the technology developed for the special case of opening the valve for the rightmost edge can be used to solve the PB problem. First, we show that essentially the same algorithm can be used to solve the case when it is the middle edge whose valve is closed to begin with. Then we show how to use this case to apply a divide and conquer technique to solve the entire problem.

Lemma 8. *Given the CBS solution with all edges but $e_{n/2}$ open, the BS solution can be found in $O(n)$ time.*

The divide and conquer strategy we follow is the most natural one: recursively, each half can be solved separately by keeping the middle valve closed. We then combine them by opening the middle valve.

Theorem 2. *The BS can be found in $O(n \log n)$ time.*

4 Conclusion and Open Problems

We gave an $O(n \log n)$ algorithm for a natural load balancing problem on paths. The same problem can be generalized to trees, and trees in hypergraphs. Extending our techniques to handle these cases is an interesting open problem. Our problem is also a special case of a power-minimizing scheduling problem for which the best known algorithm runs in time $O(n^2 \log n)$. A challenging open problem is if our algorithm can be extended to solve this problem. Also, the original motivation for our problem was that it could be useful in obtaining a faster algorithm for bipartite matching. Improving the running time for this problem is a long-standing open problem.

Acknowledgements

References

[COS01]   Cole, R., Ost, K., Schirra, S.: Edge-coloring bipartite multigraphs in $O(E \log D)$ time. Combinatorica 21(1), 5–12 (2001)

[FM95]   Feder, T., Motwani, R.: Clique partitions, graph compression and speeding-up algorithms. J. Comput. Syst. Sci. 51(2), 261–272 (1995)

[GKK09]   Goel, A., Kapralov, M., Khanna, S.: Perfect matchings in O(n logn) time in regular bipartite graphs. In: CoRR, abs/0909.3346 (2009) (also to appear in STOC 2010)

[HK71]   Hopcroft, J.E., Karp, R.M.: A $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. In: FOCS, pp. 122–125 (1971)

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

[IM81]   Ibarra, O.H., Moran, S.: Deterministic and probabilistic algorithms for maximum bipartite matching via fast matrix multiplication. Inf. Process. Lett. 13(1), 12–15 (1981)

[LY05]   Li, M., Yao, F.F.: An efficient algorithm for computing optimal discrete voltage schedules. In: Jedrzejowicz, J., Szepietowski, A. (eds.) MFCS 2005. LNCS, vol. 3618, pp. 652–663. Springer, Heidelberg (2005)

[LYY06]   Li, M., Yao, A.C., Yao, F.F.: Discrete and continuous min-energy schedules for variable voltage processors. Proceedings of the National Academy of Sciences of the USA 103, 3983–3987 (2006)

[YDS95]   Yao, F.F., Demers, A.J., Shenker, S.: A scheduling model for reduced cpu energy. In: FOCS, pp. 374–382 (1995)

Fully-Dynamic Hierarchical Graph Clustering Using Cut Trees

Christof Doll, Tanja Hartmann, and Dorothea Wagner

Department of Informatics, Karlsruhe Institute of Technology (KIT)[⋆]
christof@doll.de.com, {t.hartmann,dorothea.wagner}@kit.edu

Abstract. Algorithms or target functions for graph clustering rarely admit quality guarantees or optimal results in general. However, a hierarchical clustering algorithm by Flake et al., which is based on minimum s-t-cuts whose sink sides are of minimum size, yields such a provable guarantee. We introduce a new degree of freedom to this method by allowing arbitrary minimum s-t-cuts and show that this unrestricted algorithm is complete, i.e., any clustering hierarchy based on minimum s-t-cuts can be found by choosing the right cuts. This allows for a more comprehensive analysis of a graph's structure. Additionally, we present a dynamic version of the unrestricted approach which employs this new degree of freedom to maintain a hierarchy of clusterings fulfilling this quality guarantee and effectively avoid changing the clusterings.

1 Introduction

Graph clustering has become a central tool for the analysis of networks in general, with applications ranging from the field of social sciences to biology and to the growing field of complex systems. The general aim of *graph clustering* is to identify dense subgraphs (*clusters*) that are sparsely connected in networks, i.e., a good clustering conforms to the paradigm of *intra-cluster density* and *inter-cluster sparsity*. Countless formalizations thereof exist, however, the overwhelming majority of algorithms for graph clustering relies on heuristics and do not allow for any structural guarantee on their outputs [1,2]. Inspired by the work of Kannan et al. [6], Flake et al. [3] recently presented a hierarchical clustering algorithm that does guarantee a very reasonable bottleneck-property based on an input parameter and returns clusterings at different levels of granularity. Their elegant approach exploits properties of *cut trees*, pioneered by Gomory and Hu [4]. It partially constructs those trees using minimum s-t-cuts whose sink sides are of minimum size. Due to this restriction the returned clusterings are unique. However, the algorithm possibly misses convenient clusterings in graphs where minimum s-t-cuts and cut trees are not unique.

    We show that a restriction to specific cuts is not necessary, i.e., permitting arbitrary minimum s-t-cuts is a feasible degree of freedom. This makes the method more powerful since construction may actually use the most appropriate cut, depending on the application (cp. Fig. 1). We further prove that the unrestricted approach is even complete, i.e., any clustering hierarchy based on minimum s-t-cuts can be returned by choosing the right cuts. Additionally, we develop the first update algorithm that efficiently and

dynamically maintains a whole hierarchy of clusterings, as found by our unrestricted method, for a dynamically changing graph. This algorithm allows arbitrary atomic changes, and employs the new degree of freedom to save costs and keep consecutive clusterings on the same level similar (what we call *temporal smoothness*).

(a) Old clustering (gray).   (b) New unrestricted clus.   (c) New restricted clus.

Fig. 1. Edge $\{a,b\}$ changes, unrestricted approach allows to retain old clustering and avoids singletons.

We briefly give our notational conventions and two fundamental insights in Sec. 2. Then, in Sec. 3, we revisit the static hierarchical algorithm by Flake et al. [3] and prove correctness and completeness of this approach when using arbitrary minimum cuts. In Sec. 4 we present our new update algorithm and its analysis, concluding in Sec. 5.

2 Preliminaries and Notation

Throughout this work we consider an undirected, weighted graph $G = (V, E, c)$ with vertex set V, edge set E and a non-negative edge weight function c. We write $c(u, v)$ as a shorthand for $c(\{u, v\})$ with $u \sim v$, i.e., $\{u, v\} \in E$. We reserve the term *node* (or *super-node*) for compound vertices of abstracted graphs, which may contain several basic vertices; however, we identify singleton nodes with the contained vertex without further notice. Dynamic modifications of G will solely concern edges as vertex insertions and deletions are trivial for a disconnected vertex. Thus, a modification of G always involves an edge $\{b, d\}$, yielding $G^{\oplus}$ if $\{b, d\}$ is newly inserted into G, and $G^{\ominus}$ if it is deleted from G. We write $G^{\oplus\ominus}$ as a shorthand for $G^{\oplus}$ *or* $G^{\ominus}$. Decreasing edge weights can be handled by the same method as deletions, the techniques for edge insertions also apply for increasing weights. We further assume G to be connected; otherwise one can work on each connected component independently and the results still apply.

An edge $e_T = \{u, v\}$ of a tree $T(G) = (V, E_T, c_T)$ on V induces a cut in G by decomposing $T(G)$ into two connected components. A weighted tree $T(G)$ is called a *cut tree* [4,5] if edge weights correspond to cut weights and if for any vertex pair $\{u, v\} \in \binom{V}{2}$ the cheapest edge on the unique path between u and v induces a minimum u-v-cut in G. Neither must this edge be unique, nor $T(G)$. Note that we sometimes identify e_T with the cut it induces in G.

A *contraction* of G by $N \subseteq V$ means replacing the set N in G by a single node, denoted by $[N]$, and leaving this node adjacent to all former adjacencies u of vertices of N, with edge weight equal to the sum of all former edges between N and u.

Our understanding of a *clustering* $\mathcal{C}(G)$ of G is a partition of V into subsets C, which define vertex-induced subgraphs, called *clusters*. In the context of dynamic graphs and edge modifications of $\{b, d\}$ we particularly designate C^b, C^d and $C^{b,d}$ containing b and d, respectively. A *hierarchy of clusterings* is a sequence $\mathcal{C}_1(G) \leq \cdots \leq \mathcal{C}_r(G)$ of clusterings such that $\mathcal{C}_i(G) \leq \mathcal{C}_j(G)$ implies that each cluster in $\mathcal{C}_i(G)$ is a subset of a cluster in $\mathcal{C}_j(G)$. We say $\mathcal{C}_i(G) \leq \mathcal{C}_j(G)$ are *hierarchically nested*.

We start by giving two fundamental insights about cuts in static and dynamic graphs. Lemma 1 results from the basic properties of cut trees and is proven in App. B of [10]. We will use Observation 2 without further notice.
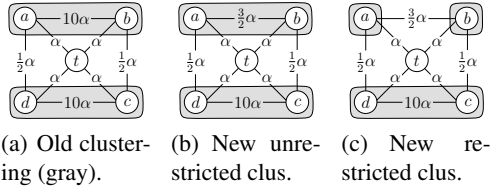
Lemma 1. *Let $(U, V \setminus U)$ denote a minimum u-v-cut in G, $u \in U$ and $x \in U$. Then there exists a minimum x-v-cut $(X, V \setminus X)$ in G, $x \in X$, such that $X \subseteq U$.*

Observation 2. Suppose edge $\{b, d\}$ changes in G yielding $G^{\oplus\ominus}$. Let θ denote a minimum u-v-cut in $G^{\oplus\ominus}$ and $\hat{\theta}$ a min-u-v-cut in G, both not separating b and d. Then $c^{\oplus\ominus}(\theta) = c(\theta) = c(\hat{\theta}) = c^{\oplus\ominus}(\hat{\theta})$, i.e., $\hat{\theta}$ is a minimum u-v-cut in $G^{\oplus\ominus}$.

3   The Static Hierarchical Clustering Algorithm

Flake et al. [3] propose and evaluate a hierarchical algorithm, which clusters instances in a way that yields a certain guarantee on the quality of the clusters. This quality guarantee is inherited from a basic clustering procedure, which computes one clustering. Applying this procedure iteratively to instances obtained by contracting foregoing clusters yields a clustering hierarchy.

The Basic Clustering Procedure.   Inspired by a bicriterial approach for good clusterings by Kannan et al. [6], Flake et al. [3] design a basic clustering procedure that, given parameter α, asserts: [1]

$$\underbrace{\frac{c(C, V \setminus C)}{|V \setminus C|}}_{\text{inter-cluster cut}} \leq \alpha \leq \underbrace{\frac{c(P, Q)}{\min\{|P|, |Q|\}}}_{\text{expansion of intra-cluster cut } (P,Q)} \qquad \forall C \in \mathcal{C}(G) \quad \forall P, Q \neq \emptyset \quad P \dot\cup Q = C$$

This quality guarantee is due to special properties of cut trees, which are used by the procedure: Given a graph G and parameter $\alpha > 0$, augment G by inserting an artificial vertex t and connecting t to each vertex in G by an edge of weight α. Then compute a cut tree $T(G_\alpha)$ of the resulting graph G_α. Finally, remove t from $T(G_\alpha)$, which decomposes $T(G_\alpha)$ into connected components, which are returned as clusters in $\mathcal{C}(G)$. In the following we call a clustering that can be computed by this procedure a *cut-clustering*, and we denote by $G_\alpha^\ominus$ and $G_\alpha^\oplus$ the augmented *and* modified graphs.

Flake et al. further point out that, instead of constructing a whole cut tree, only knowing the edges of $T(G_\alpha)$ incident to t would suffice. According to Lemma 3, which directly follows from a lemma introduced by Gusfield [5], Alg. 1 (LCC), with $S = \emptyset$, returns a cut-clustering by constructing such a partial cut tree, which is in fact a *star* with center t. The parameter S will be used later for the dynamic approach. The number of cuts calculated in LCC depends on the sequence of chosen sinks and the shape of the returned cuts. Already known cuts might be covered by later cuts in line 7, i.e., possibly computed without need.

Lemma 3 (Gusfield [5], Lemma 1). *Let $(C^i, V_\alpha \setminus C^i)$ be a min-t-$r(C^i)$-cut in G, with $r(C^i) \in C^i$. Let $(H, V_\alpha \setminus H)$ be a min-t-u-cut, with $t, u \in V_\alpha \setminus C^i$ and $r(C^i) \in H$. Then the cut $(C^i \cup H, (V_\alpha \setminus C^i) \cap (V_\alpha \setminus H))$ is also a min-t-u-cut.*

Line 3 in LCC represents the new degree of freedom. Whenever used in a hierarchical context, Flake et al. restricted this to minimum t-u-cuts whose sink sides are of

---

[1] The disjoint union $A \cup B$ with $A \cap B = \emptyset$ is denoted by $A \dot\cup B$.

Algorithm 1. LEVEL CUT-CLUSTERING (LCC as a shorthand)

Input: Graph $G_\alpha = (V_\alpha, E_\alpha, c_\alpha)$, set S

1 $\mathcal{C}(G) \leftarrow S$; $V \leftarrow V_\alpha \setminus (\{t\} \cup \bigcup_{C \in S} C)$
2 **while** $\exists\, u \in V$ **do**
3 $\quad$ $(U, V_\alpha \setminus U) \leftarrow$ min-t-u-cut in G_α, $t \notin U$ $\qquad\qquad$ // new degree of freedom
4 $\quad$ $C^u \leftarrow U$; $r(C^u) \leftarrow u$
5 $\quad$ **forall** $C^i \in \mathcal{C}(G)$ **do**
6 $\quad\quad$ **if** $r(C^i) \in C^u$ **then** $\qquad\qquad\qquad\qquad$ // $C^u =: H$ covers C^i
7 $\quad\quad\quad$ $C^u \leftarrow C^u \cup C^i$; $\mathcal{C}(G) \leftarrow \mathcal{C}(G) \setminus \{C^i\}$ $\qquad$ // reshaping by Lem.3
8 $\quad\quad$ **else** $C^u \leftarrow C^u \setminus C^i$ $\qquad\qquad$ // reshaping by Lem.3, $C^u =: V_\alpha \setminus H$
9 $\quad$ $\mathcal{C}(G) \leftarrow \mathcal{C}(G) \cup \{C^u\}$; $V \leftarrow V \setminus C^u$

---

minimum size and called the minimum sink side the *community* of u and u a *representative* of its community. Analogously, we call U a *cut side* with *representative* $r(U)$ if $(U, V_\alpha \setminus U)$ is a minimum t-u-cut in G_α, with $u \in U$. We assume, that the final clustering $\mathcal{C}(G)$ found by LCC stores at least one representative per cluster. In the following we identify t-u-cuts $(U, V_\alpha \setminus U)$ with vertex sets U, $u \in U$ and $t \notin U$.

The Hierarchical Algorithm. Flake et al. developed a hierarchical clustering approach (HCC), which uses LCC iteratively (see Alg. 2). On each level the returned hierarchy provides a cut-clustering $\mathcal{C}_i(G)$ of G with respect to a particular α_i, i.e., $\mathcal{C}_i(G)$ holds the quality guarantee. We call such a hierarchy a *cut-clustering hierarchy*. Iterating a cut-clustering hierarchy bottom-up the α_i-values decrease, i.e.,

---

Algorithm 2. HIERARCHICAL CC

Input: $G = (V, E, c)$, $\alpha_1 > \cdots > \alpha_r$

1 $\mathcal{C}_0(G) \leftarrow \{\{v\} \mid v \in V\}$; $r(\{v\}) \leftarrow v$
2 **for** $i = 1, \ldots, r$ **do**
3 $\quad$ **forall** $C \in \mathcal{C}_{i-1}(G)$ **do**
4 $\quad\quad$ contract C in G_{α_i}
5 $\quad\quad$ associate $[C]$ with $r(C)$
6 $\quad$ $\mathcal{C}_i(G) \leftarrow$ LCC$(G_{\alpha_i}, \emptyset)$

$\alpha_i > \alpha_j$ for $i < j$. For the proof of correctness of Alg. 2 Flake et al. employed special nesting properties of communities. These properties guarantee that communities do not change in line 7 and 8 of Alg. 1 and that communities in the contracted graph (Alg. 2, line 4) correspond to communities in the original graph. Thus, the restricted LCC applied to the contracted graph also returns a valid cut-clustering for G, and the resulting hierarchy is a cut-clustering hierarchy.

Correctness and Completeness of Unrestricted HCC. In the following we show that HCC remains correct if we apply LCC with arbitrary minimum t-u-cuts, and that this unrestricted approach is complete. We further characterize the set of cut-clustering hierarchies.

Theorem 4. *Unrestricted HCC is correct and complete.*

In order to prove the correctness of HCC independently from special nesting properties of communities, we state the following lemma and show that arbitrary minimum t-u-cuts in the contracted graph are also cut sides in the original graph. Otherwise, LCC applied to the contracted graph would possibly not return a valid cut-clustering for G.

Lemma 5. *Let $(U, V_{\alpha_j} \setminus U)$ denote a min-t-u-cut in G_{α_j} with $u \in U$, and for $\alpha_i > \alpha_j$ let $(X, V_{\alpha_i} \setminus X)$ denote a minimum t-x-cut in G_{α_i} with $x \in X$. Then it holds (a) $X \subseteq U$ if $x \in U$ and (b) $X \cap U = \emptyset$ if $x \notin U$ and $u \notin X$.*

Figure 2 sketches X and U and the conclusions (dashed cuts) proven by contradiction in App. C of [10]. Note that for our purpose the case $x \notin U$ but $u \in X$ is irrelevant. Lemma 5 tells us the following: Consider a minimum t-$r(C)$-cut θ in the original graph G_{α_j} with $r(C)$ a representative of a designated node $[C]$ in the contracted graph, and let $[C']$ denote an arbitrary node

(a) It is $X \subseteq U$ if $x \in U$.

(b) $X \cap U = \emptyset$ if $x \notin U$ and $u \notin X$.

Fig. 2. Sketch to proof of Lem. 5

in the contracted graph. If $r(C')$ is in θ then θ also contains $[C']$; in particular, θ contains $[C]$. If $r(C')$ is not in θ then $\theta \cap C' = \emptyset$. Thus, θ is a proper cut in the contracted graph and contains $[C]$. Conversely, each minimum t-$[C]$-cut in the contracted graph is a proper cut in G_{α_j} and contains $r(C)$. Consequently, there exists a 1-1-correspondence between minimum t-$r(C)$-cuts in the original graph and minimum t-$[C]$-cuts in the contracted graph, and LCC applied to the contracted graph returns a valid cut-clustering for G.

According to the proof of correctness, by choosing the right cuts HCC is capable to return any cut-clustering hierarchy where the representatives of clusters on one level are a subset of the representatives on the level below. The following lemma shows that this property holds for any cut-clustering hierarchy. Thus, Lemma 5 and Lemma 6 together witness the completeness of HCC. The proof of Lemma 6 is in App. C of [10].

Lemma 6. *Let $\mathcal{C}_i(G)$ and $\mathcal{C}_j(G)$ denote two cut-clusterings with respect to $\alpha_i > \alpha_j$ and let $C' \in \mathcal{C}_i(G)$ and $C \in \mathcal{C}_j(G)$ denote two clusters with $r(C') \neq r(C)$ but $r(C) \in C'$. Then it holds $C' \subseteq C$ and $r(C')$ is a representative of C in $\mathcal{C}_j(G)$.*

We further give the following simple characterization of all cut-clustering hierarchies and present Corollary 8, which we will apply later to prove temporal smoothness and the feasibility of certain vertex contractions. For a proof of Theorem 7 see App. C of [10].

Theorem 7. *Given a sequence $\alpha_1 > \cdots > \alpha_r$ of parameter values each set of cut-clusterings $\mathcal{C}_1(G), \ldots, \mathcal{C}_r(G)$ forms a hierarchy.*

Corollary 8. *A cluster $C \in \mathcal{C}_j(G)$ separates G into C and $V \setminus C$ such that both parts are clustered independently with respect to $\alpha_i > \alpha_j$, i.e., minimum cuts in G_{α_i} with representatives in C do not cover any vertex in $V \setminus C$ and vice versa.*

Otherwise there would exist a cut-clustering $\mathcal{C}_i(G)$ that is not hierarchically nested in $\mathcal{C}_j(G)$ contradicting Theorem 7.

4   Update Algorithm for Dynamic Clustering Hierarchies

The second part of this work addresses a dynamic version of HCC. We give a method that employs the new degree of freedom for consecutively updating cut-clustering hierarchies with respect to a given sequence of α's. Based on Theorem 7 this can be already
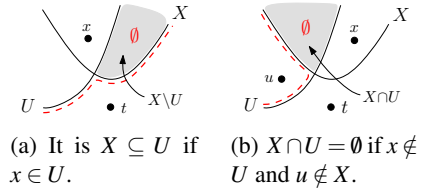
done by simply updating each level independently using a dynamic version of the basic clustering procedure LCC given by Hartmann et al. [8] (which corrects an approach proposed by Saha and Mitra [7]). Since the static LCC, as introduced by Flake et al. [3], is not restricted to communities, the dynamic version by Hartmann et al. also allows for the use of arbitrary cuts, and thus, already achieves good temporal smoothness and some cost savings. However, in the following we present a more efficient algorithm, which also exploits the hierarchical structure to save costs and to provide high temporal smoothness.

The Basic Clustering Procedure in a Dynamic Scenario. Hartmann et al. [8] developed an algorithm for dynamically updating single cut-clusterings. We will refer to this algorithm by LU (for level update). Given a cut-clustering $\mathcal{C}(G)$, we distinguish four cases of edge modification: inter-cluster deletion (*inter-del*), where the deleted edge is incident to vertices in different clusters, intra-cluster deletion (*intra-del*), i.e., an edge within a cluster is removed, and analogously, inter- and intra-cluster insertion (*inter-ins, intra-ins*). LU reshapes cuts in order to prevent previous clusters from splitting. In this way some clusters are guaranteed to remain clusters or at least subsets of clusters after a change. Regarding different modification cases the following facts hold [8]:

a)   all clusters in $\mathcal{C}(G) \setminus \{C^b, C^d\}$ (for *inter-ins*) and in $\{C^b, C^d\}$ (for *inter-del*) are still cut sides in $G_\alpha^{\oplus\ominus}$ with respect to their previous representatives.

b)   if $C^{b,d}$ (for *intra-del*) or C^b and C^d (for *inter-ins*) are still cut sides with respect to any representative after the change, $\mathcal{C}(G)$ is still a cut-clustering for $G^{\oplus\ominus}$. We call this the *copy-property* of $\mathcal{C}(G)$. However, the previous representatives of $C^{b,d}$ or C^b, C^d possibly become invalid.

c)   for *intra-ins*, $\mathcal{C}(G)$ fulfills the copy-property retaining all representatives.

d)   for *inter-del*, LU computes at most $|\mathcal{C}(G)| - 2$ minimum cuts, and updating $\mathcal{C}(G)$ by LU yields $\mathcal{C}(G) = \mathcal{C}(G^\ominus)$ with valid representatives if $\mathcal{C}(G)$ fulfills the copy-property.

e)   for any *deletion*, consider $C \in \mathcal{C}(G)$ with $b, d \notin C$. There exists a minimum t-$r(C)$-cut X in $G_\alpha^\ominus$ with $C \subseteq X$.

An Intelligent Hierarchical Approach from Scratch. The naive way to compute a new hierarchy after a change in G is to apply HCC from scratch. In Sec. 3 we showed that HCC allows for the use of arbitrary cuts, i.e., the construction may use the most appropriate cut, depending on the application. Given an appropriate initial hierarchy we present a hierarchical approach that still calculates a new hierarchy from scratch but adopts appropriate cuts applied before. To this end we modify HCC by improving LCC: When computing a new min-t-u-cut θ (u may be a node) let C denote the cluster that contains $r(u)$ in the old clustering on the same level. If $c^{\oplus\ominus}(\theta) = c^{\oplus\ominus}(C)$ in $G_\alpha^{\oplus\ominus}$, LCC takes C as new minimum t-u-cut.

Lemma 9. *In the situation described above it is $u \subseteq C$ and C is a minimum t-u-cut in the contracted graph (Alg. 2, line 4) resulting from $G_\alpha^{\oplus\ominus}$. Thus, the intelligent hierarchical approach is correct.*

For a proof see App. D of [10]. In the following we will refer to the improved LCC by *intelligent* LCC (or ILCC). We will further express the costs of our new update

algorithm in terms of costs of the intelligent hierarchical approach: Given a hierarchy for G and a new hierarchy from level 1 to level $i-1$ we denote the costs for extending the hierarchy to level j by $\mathcal{T}([i,j],G^{\oplus\ominus})$.

For one level, $\mathcal{T}([i,i],G^{\oplus\ominus})$ consists of the costs for contracting the clusters on level $i-1$ and the costs of ILCC applied to the contracted graph. The latter depend on the size of the contracted graph, which influences the runtime of the cut computations, and the number of calculated cuts.

Reusable Parts of the Hierarchy in a Dynamic Scenario. Given an edge modification a cut-clustering hierarchy decomposes into two parts. Levels where the modification induces an inter-cluster event form the lower part, intra-cluster event levels build the upper part. The first idea in this paragraph considers levels of intra-cluster events. According to Fact c) each intra-ins level can be copied to a new hierarchy. An intra-del level can be copied if $C^{b,d}$ remains a cut side, cf. Fact b). The following lemma gives a further indicator for an intra-del level fulfilling the copy property. We sketch the proof in App. D of [10].

Lemma 10. *Let $\mathcal{C}(G)$ denote an intra-del cut-clustering with $b,d \in C^{b,d}$. If no cut-clustering $\mathcal{C}(G^{\ominus})$ exists with b,d in different clusters, $\mathcal{C}(G)$ fulfills the copy-property. If there exists a cut-clustering $\mathcal{C}_i(G^{\ominus})$ with $b,d \in C_i^{b,d}$, each cut-clustering $\mathcal{C}_j(G)$ with $\alpha_i > \alpha_j$ fulfills the copy-property.*

According to Lemma 10 and Fact c) we get the following:

Theorem 11. *Given a cut-clustering hierarchy, let k denote the lowest intra-del level that fulfills the copy-property (deletion) or just the lowest intra-ins level (insertion). Then all levels $i \geq k$ can be reused as part of a new hierarchy (however, in case of deletion some representatives possibly become invalid, cf. Fact b)).*

A second idea is to consider subtrees of clusters. A *subtree* of a cluster C on level i consists of C and all clusters on lower levels in the hierarchy that are nested in C. Lemma 12 (proof in App. D of [10]) and Theorem 13 attest that in some cases we can preserve the whole subtree of a cluster after a change in G.

Lemma 12. *Let $C \not\ni b,d$ denote a cluster in $\mathcal{C}_j(G)$ that remains a cut side for $r(C)$ (which is equivalent to any representative) in $G_{\alpha_j}^{\oplus\ominus}$. Let further denote $C' \subseteq C$ a cluster in $\mathcal{C}_i(G)$, $i < j$. Then C' remains a cut side for $r(C')$ in $G_{\alpha_i}^{\oplus\ominus}$.*

If C in Lemma 12 even remains a cluster in a new cut-clustering $\mathcal{C}_j(G^{\oplus\ominus})$, according to Corollary 8 the following holds:

Theorem 13. *In a cut-clustering hierarchy let $C \not\ni b,d$ denote a cluster in $\mathcal{C}_j(G)$ that is also a cluster in a cut-clustering $\mathcal{C}_j(G^{\oplus\ominus})$. Then the whole subtree of C can be used as part of a new hierarchy (representatives remain valid).*

We define the root of a (inclusion-) maximal reusable subtree as a *highest root*.

Our New Update Approach. Our new update approach treats the two parts of inter- and intra-cluster event levels of the hierarchy differently. We start by applying Theorem 11 and Theorem 13 to intra-cluster event levels and estimate the costs in terms of costs of the intelligent HCC.

Algorithm 3. UPDATE INTRA-DEL LEVEL

Input: Graph $D_\alpha^\ominus$, cut-clustering $\mathcal{C}(G) \ni C^{b,d}$

1 **if** $\exists C \in \mathcal{C}(G)$ *that is not a proper union of nodes in V* **then**   // $V := V(D_\alpha^\ominus)$
2    $\quad \mathcal{C}(G^\ominus) \leftarrow \text{ILCC}(D_\alpha^\ominus, \emptyset)$ // ILCC takes nodes containing...
3    $\quad$ **return** $(\mathcal{C}(G^\ominus), \text{false})$ // ...representatives in $\mathcal{C}(G)$ first

4 **while** $\exists u \in V$ *with* $u \subseteq C^{b,d}$ **do**   // start with $u \ni r(C^{b,d})$
5    $\quad U \leftarrow$ community of u in $D_\alpha^\ominus$
6    $\quad$ **if** $\exists x \in U$ *with* $x \not\subseteq C^{b,d}$ **then** apply line 4 to 9 of Algo 1; goto line 9
7    $\quad$ **if** $c^\ominus(U) = c(C^{b,d})$ **then** $\mathcal{C}(G^\ominus) \leftarrow \mathcal{C}(G)$; $r(C^{b,d}) \leftarrow u$; **return** $(\mathcal{C}(G^\ominus), \text{true})$
8    $\quad$ apply line 4 to 9 of Algo 1 (ILCC)

9 **while** $\exists u \in V$ **do**   // ILCC takes nodes containing rep. in $\mathcal{C}(G)$ first
10    $\quad$ apply line 3 to 9 of Algo 1 (ILCC)

11 **return** $(\mathcal{C}(G^\ominus), \text{false})$

In case of insertion, Theorem 11 tells us that we can just copy each intra-ins level to a new hierarchy without further costs (cf. upper shaded area in Fig. 3).

In case of deletion, we search for the lowest intra-del level k that fulfills the copy property. To this end, beginning at the lowest intra-del level ℓ we iteratively apply Alg. 3 until the first copy-property level k is found. Alg. 3 takes an intra-del clustering $\mathcal{C}_i(G)$ and a graph $D_{\alpha_i}^\ominus$ obtained from $G_{\alpha_i}^\ominus$ by contracting clusters on level $i-1$. Line 2 catches a case where $\mathcal{C}_i(G)$ obviously does not fulfill the copy-property and applies ILCC in this case. If $\mathcal{C}_i(G)$ fulfills the copy-property, according to Fact b) it suffices to find a valid representative for $C_i^{b,d}$. Thus, lines 4 ff. search for such a representative and return $\mathcal{C}_i(G)$ together with the representative if one is found and continue ILCC otherwise. Lemma 17 in App. D of [10] shows that Alg. 3 finds a valid representative of $C_i^{b,d}$ if there is one. The costs for updating level ℓ to $k-1$ are about $\mathcal{T}([\ell, k-1], G^\ominus)$ since Alg. 3 is just a modified LCC (see Fig. 3, area (1)).

After we found level k we can actually copy all levels $i > k$ according to Theorem 11, apart from the representatives of $C_i^{b,d}$, $i = k+1, \ldots, r$. Hence, we apply the while-loop in line 4 of Alg. 3 instead of copying the levels, since this additionally returns valid representatives. This costs about $\sum_{i=k}^r \mathcal{T}([i,i], C_i^{b,d})$ also including the costs for level k (see area (2), Fig.3).

However, in order to apply Alg. 3 the first time on level ℓ we need to compute a clustering $\mathcal{C}_{\ell-1}(G^\ominus)$ on the highest inter-del level acting as a base for contracting the initial instance. To this end, we contract $C_{\ell-1}^b$ and $C_{\ell-1}^d$ in $G_{\alpha_{\ell-1}}^\ominus$ and associate the nodes with $r(C^b)$ and $r(C^d)$. Then we apply LU to the obtained graph, which is feasible and costs about $\mathcal{T}([l-1, l-1], G^\ominus)$; see App. E of [10].

In both cases, insertion and deletion, we can further reuse the subtrees of all clusters $C \in \mathcal{C}_k(G) \setminus \{C_k^{b,d}\}$ by Theorem 13 (see lower shaded area in Fig. 3). This already updates parts of inter-cluster event levels. In case of deletion, the clusters of subtrees overlapping levels $\ell-1$ to $k-1$ already exist in $\mathcal{C}_{\ell-1}(G^\ominus), \ldots, \mathcal{C}_{k-1}(G^\ominus)$ since Alg. 3 and LU construct reusable subtrees, apart from highest roots, by default according to Corollary 8 (see also Lemma 18 and Lemma 19 in App. D of [10]).
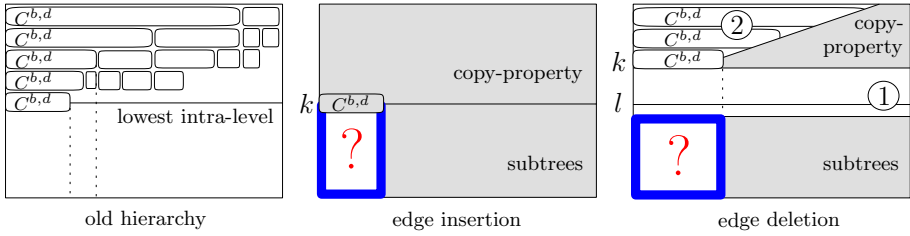
Fig. 3. Sketch of costs for updating a hierarchy using our first update approach. Shaded areas represent saved costs compared to a hierarchical construction from scratch.

Observation 14. Each level $i \geq \ell$ (intra-del) or $i \geq k$ (intra-ins) that fulfills the copy-property and each reusable subtree that is rooted on level $i \geq k$ is part of the new hierarchy (with valid representatives) resulting from our update approach.

By updating the intra-cluster event levels with this approach, we reduce the problem of updating a cut-clustering hierarchy of r levels to an update of $k - 1$ levels (insertion) or $\ell - 2$ levels (deletion), regarding an instance just as big as $C_k^{b,d}$ (cf. boxed question mark in Fig. 3).

Strategies for Completing the Hierarchy on Inter-Cluster Event Levels. After our first update step we still need to fill in the question marks in Fig. 3, i.e., construct a hierarchy based on the vertices in $C_k^{b,d}$. According to Corollary 8, $C_k^{b,d}$ and $V \setminus C_k^{b,d}$ in G are clustered independently on the missing levels. Thus, when updating level i in the following, we consider $G_{\alpha_i}^{\oplus\ominus}$ with $V \setminus C_k^{b,d}$ contracted into a node representing the subtrees already used.

In case of insertion, we iterate the missing levels bottom-up contracting $G_{\alpha_i}^{\oplus}$ as the hierarchical approach does. On each level we apply Alg. 4, which is a modified LCC. It takes an inter-ins clustering $\mathcal{C}_i(G)$ and a graph $G_{\alpha_i}^{\oplus}$ contracted as described above. Line 1 further contracts $G_{\alpha_i}^{\oplus}$, which, together with line 2, enables the algorithm to save the costs for explicitly constructing reusable trees, as we will see later. The contraction is as follows: Contract each $C \in \mathcal{C}_i(G) \setminus \{C^b, C^d\}$ that is a proper union of nodes in the current instance $G_{\alpha_i}^{\oplus}$. Associate a new node $[C]$ with $r(X)$, where $X \in \mathcal{C}_{i-1}(G^{\oplus})$ contains $r(C)$. In Lemma 21, found in App. E of [10], we prove that applying ILCC to the obtained graph $D_{\alpha_i}^{\oplus}$ is correct, i.e., returns a cut-clustering for $G^{\oplus}$. Line 3 catches a case where $\mathcal{C}_i(G)$ obviously does not fulfill the copy-property and applies ILCC in this case. If $\mathcal{C}_i(G)$ fulfills the copy-property according to Fact b) it suffices to find a valid representative for C_i^b and C_i^d. Thus, lines 6 ff. search for those representatives and return the part of $\mathcal{C}_i(G)$ that is nested in $C_k^{b,d}$ together with the representatives if some are found or continue ILCC otherwise. The proof that Alg. 4 finds valid representatives of C_i^b and C_i^d if some exist is analog to Alg. 3. Although Alg. 4 detects each level that fulfills the copy-property, when updating inter-ins levels we cannot directly benefit from their copy-property. Thus, applying Alg. 4 to $k - 1$ inter-ins levels costs about $\mathcal{T}([1, k-1], C_k^{b,d})$; see App. E of [10].

Furthermore, the bottom-up iteration makes the reuse of subtrees impossible. However, Alg. 4 counterbalances the missing subtree conservation. Using the same

---

Algorithm 4. UPDATE INTER-INS LEVEL

Input: Graph $G_\alpha^\oplus$, partial clustering $P := \{C \in \mathcal{C}(G) \mid C \subseteq C_k^{b,d}\} \supseteq \{C^b, C^d\}$

1   $D_\alpha^\oplus \leftarrow$ contract some $C \in P \setminus \{C^b, C^d\}$ in $G_\alpha^\oplus$ according to text description

2   $S \leftarrow \{C \in P \mid [C] \text{ in } D_\alpha^\oplus \text{ formed in line 1}\}$ `//identify` $V := V(G_\alpha^\oplus)$ `with` $C_k^{b,d}$

3   **if** $\exists\, C \in P$ that is not a proper union of nodes in V **then**

4      $\big\lfloor\;\; \mathcal{C}(G^\oplus) \leftarrow \text{ILCC}(D_\alpha^\oplus, S);$ **return** $(\mathcal{C}(G^\oplus), \text{false})$

5   $\mathcal{C}(G^\oplus) \leftarrow S;\; V \leftarrow V \setminus \bigcup_{C \in S} C;\; b \leftarrow \text{false};\; d \leftarrow \text{false}$ `//` $V = C^b \cup C^d$

6   **while** $\exists\, u \in V$ **do**   `// start with` $u_b \ni r(C^b)$ `and` $u_d \ni r(C^d)$

7      $U \leftarrow$ community of u in $D_\alpha^\oplus$

8   **if** $\exists\, x \in U$ with $x \not\subseteq C^b$ or $x \not\subseteq C^d$ **then** skip line 3 in later iterations

9   **if** $c^\oplus(U) = c(C^b)$ or $c^\oplus(U) = c(C^d)$ **then**

10         $b \leftarrow \text{true};\; r(C^b) \leftarrow u;\; U \leftarrow C^b$ (if currently $u \subseteq C^b =: Z$)

11         $d \leftarrow \text{true};\; r(C^d) \leftarrow u;\; U \leftarrow C^d$ (if currently $u \subseteq C^d =: Z$)

12   in later iterations only consider $u \not\subseteq Z$, in line 6

13   **if** b and d **then** $\mathcal{C}(G^\oplus) \leftarrow P;$ **return** $(\mathcal{C}(G^\oplus), \text{true})$

14   apply line 4 to 9 of Algo 1 (ILCC)

15   **return** $(\mathcal{C}(G^\oplus), \text{false})$

---

techniques as Alg. 3, Alg. 4 returns all reusable subtrees by default, apart from highest roots. It even saves the costs for explicitly constructing such trees, due to lines 1 and 2 as follows: By Corollary 8 each cluster of a reusable subtree is contracted in line 1 and added to S in line 2. Due to Fact a) the nodes in S are considered as cut sides that are already known, and thus, omitted when choosing sinks for cut computations in ILCC. Particularly, Alg. 4 avoids cut computations for clusters in reusable subtrees. Hence, we deduct the costs T for explicitly constructing reusable subtrees (see Fig. 4(a)).

In case of deletion, a bottom-up approach would not allow the reuse of subtrees. Thus, we iterate the old hierarchy top-down updating each level in the same way as level $\ell - 1$ in the first update step, but using a smaller instance due to already known subtrees. As we have seen before, this method detects each reusable subtree, possibly apart from highest roots. Thus, we copy those subtrees to the new hierarchy and merge the found roots with the node in $G_{\alpha_i}^\ominus$ that represents previously found subtrees in order to save costs. Hence, completing the hierarchy in case of deletion costs about $\mathcal{T}([1, \ell - 2], C_k^{b,d}) - T$ (see Fig. 4(a)). Since for inter-cluster deletions LU bases on ILCC, it further respects the copy-property (cf. Fact d)).

Observation 15. Each level $i \leq \ell - 1$ (inter-del) or $i \leq k - 1$ (inter-ins) that fulfills the copy-property and each reusable subtree that is rooted on level $i \leq k - 1$ is part of the new hierarchy (with valid representatives), possibly apart from highest roots.

Performance. In the following we just sum up the costs and the observations regarding temporal smoothness already given with the description of our new update approach. The latter—which we left unformalized— in parts synergizes with cost saving, an observation foremost reflected in the first update step.

Theorem 16. *Each level fulfilling the copy-property and each reusable subtree (possibly apart from highest roots) is part of the new hierarchy (with valid representatives)*

Table 1. Sketch of costs, cpc = costs per cut. $\mathcal{C}_i^*(G) := \{C \in \mathcal{C}_i(G) \mid C \subseteq C_{i+1}^*\}$ with $C_{i+1}^* := C_{i+1}^{b,d}$ or $C_{i+1}^* := C_{i+1}^b \cup C_{i+1}^d$

	arbitrary hierarchy general costs	hierarchy remains valid lowest possible costs
insertion	$\mathcal{T}([1,k-1],C_k^{b,d}) - T$	$2(k-1)$ cpc
deletion	$\sum_{i=k}^r \mathcal{T}([i,i],C_i^{b,d}) + \mathcal{T}([l-1,k-1],G^{\ominus})$ $+ \mathcal{T}([1,l-2],C_k^{b,d}) - T$	$(r-k+1) + \sum_{i=1}^{k-1} |\mathcal{C}_i^*(G)|$ cpc

(a) Costs regarding an arbitrary hierarchy.   (b) Lower bounds if hierarchy still valid.
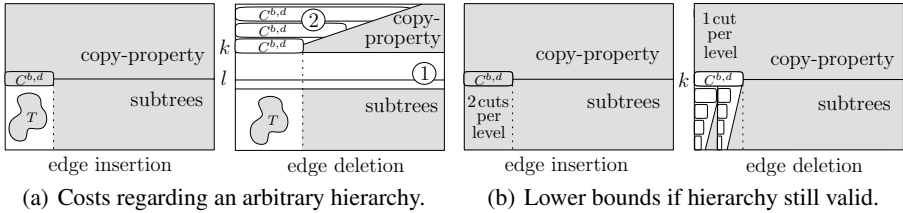
Fig. 4. Sketch of costs for updating a hierarchy applying our new update approach. Shaded areas represent saved costs compared to a hierarchical approach from scratch.

build by our update algorithm. In particular, our algorithm returns the previous hierarchy if this is still a cut-clustering hierarchy after the change.

We sketch the general costs for updating an arbitrary hierarchy in Table 1 and visualize them in Fig. 4(a). Furthermore, we consider the—possibly rather common—case that the old hierarchy is still valid after some graph modification. For this case we list the lowest possible costs in Table 1, which occur if on each inter-ins level Alg. 4 in line 6 chooses valid representatives for C^b and C^d as first nodes, or if on each intra-del level Alg. 3 in line 4 hits a representative for $C^{b,d}$ at the beginning (see Fig. 4(b)).

5   Conclusion

The hierarchical clustering algorithm by Flake et al. [3] returns a set of clusterings at different levels of granularity. The striking feature of graph clusterings computed by this method is that they are guaranteed to yield a certain *expansion*—a bottleneck measure—within and between clusters, tunable by an input parameter α. However, their method, which is based on minimum s-t-cuts, was restricted to the use of communities, and hence, was not complete. We have proven that the hierarchical approach by Flake et al. [3] remains correct if we introduce a new degree of freedom by permitting the use of arbitrary minimum s-t-cuts instead of communities. This makes the method more powerful since construction may actually use the most appropriate cut, depending on the application. We have further given a simple characterization of the set of all clustering hierarchies based on minimum s-t-cuts and have shown that the unrestricted approach is complete, i.e., any clustering hierarchy in this set can be found by choosing the right cuts. This allows for a more detailed analysis of a graph's structure.

Furthermore, we have presented an algorithm that efficiently and fully-dynamically maintains an entire hierarchy of clusterings, as computed by the unrestricted method. Clusterings in the updated hierarchy fulfill the same quality guarantee regarding expansion and, as a secondary criterion, we encourage temporal smoothness, i.e., changes to the clustering hierarchies are kept at a minimum, whenever possible. Thereby, our update algorithm employs the new degree of freedom which allows to reuse clusters independently of their special shape, and thus, saves computational costs and increases temporal smoothness (cp. Fig. 1). We also conjecture our new update algorithm, by implementing some small modifications, to be a correct dynamic version of the restricted hierarchical clustering algorithm by Flake et al., i.e., when restricted to maintaining clusters that are communities (see [9] for a first study).

Future work includes the proof of the conjecture, a systematic comparison of our algorithm to other dynamic clustering techniques and the analysis of batch updates.

References

1. Brandes, U., Erlebach, T. (eds.): Network Analysis: Methodological Foundations LNCS, vol. 3418. Springer, Heidelberg (2005)
2. Brandes, U., Delling, D., Gaertler, M., Görke, R., Höfer, M., Nikoloski, Z., Wagner, D.: On Modularity Clustering. IEEE TKDE 20(2), 172–188 (2008)
3. Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K.: Graph Clustering and Minimum Cut Trees. Internet Mathematics 1(4), 385–408 (2004)
4. Gomory, R.E., Hu, T.: Multi-terminal network flows. Journal of the Society for Industrial and Applied Mathematics 9(4), 551–570 (1961)
5. Gusfield, D.: Very simple methods for all pairs network flow analysis. SIAM Journal on Computing 19(1), 143–155 (1990)
6. Kannan, R., Vempala, S., Vetta, A.: On Clusterings: Good, Bad and Spectral. JACM 51(3), 497–515 (2004)
7. Saha, B., Mitra, P.: Dynamic Algorithm for Graph Clustering Using Minimum Cut Tree. In: Proc. of the 2007 SIAM Int. Conf. on Data Mining, pp. 581–586 (2007)
8. Görke, R., Hartmann, T., Wagner, D.: Dynamic Graph Clustering Using Minimum-Cut Trees. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 339–350. Springer, Heidelberg (2009)
9. Doll, C.: Hierarchical Cut Clustering in Dynamic Scenarios. Student Research Project, KIT Karlsruhe Institute of Technology, Department of Informatics (February 2011), http://i11www.iti.uni-karlsruhe.de/_media/teaching/theses/studienarbeitchristofdoll.pdf
10. Doll, C., Hartmann, T., Wagner, D.: Fully-Dynamic Hierarchical Graph Clustering Using Cut Trees. Karlsruhe Reports in Informatics 2011-10, KIT Karlsruhe Institute of Technology (2011)

Flow Computations on Imprecise Terrains

Anne Driemel[1,★], Herman Haverkort[2], Maarten Löffler[3,★★],
and Rodrigo I. Silveira[4,★]

[1] Dept. of Computer Science, Utrecht University
[2] Dept. of Computer Science, Eindhoven University of Technology
[3] Computer Science Dept., University of California, Irvine
[4] Dept. de Matemàtica Aplicada II, Universitat Politècnica de Catalunya

Abstract. We study water flow computation on imprecise terrains. We
consider two approaches to modeling flow on a terrain: one where water
flows across the surface of a polyhedral terrain in the direction of steepest
descent, and one where water only flows along the edges of a predefined
graph, for example a grid or a triangulation. In both cases each vertex has
an imprecise elevation, given by an interval of possible values, while its
(x, y)-coordinates are fixed. For the first model, we show that the problem
of deciding whether one vertex may be contained in the watershed of
another is NP-hard. In contrast, for the second model we give a simple
$O(n \log n)$ time algorithm to compute the minimal and the maximal
watershed of a vertex, where n is the number of edges of the graph.
On a grid model, we can compute the same in $O(n)$ time.

1 Introduction

Simulating the flow of water on a terrain is a problem that has been studied for a
long time in geographic information science (GIS), and has received considerable
attention from the computational geometry community due to the underlying
geometric problems. It can be an important tool in analyzing flash floods for
risk management [1], for stream flow forecasting [12], and in the general study of
geomorphological processes [2], and it could contribute to obtaining more reliable
climate change predictions [17].

When modeling the flow of water across a terrain, it is generally assumed
that water flows downward in the direction of steepest descent. It is common
practice to compute drainage networks and catchment areas directly from a
digital elevation model of the terrain. Most hydrological research in GIS models
the terrain surface with a grid in which each cell can drain to one or more of its
eight neighbors (e.g. [16]). This can also be modeled as a computation on a graph,
in which each node represents a grid cell and each edge represents the adjacency
of two neighbors in the grid. Alternatively, one could use an irregular network
in which each node drains to one or more of its neighbors. We will refer to this
as the *network model*, and we assume that, from every node, water flows down
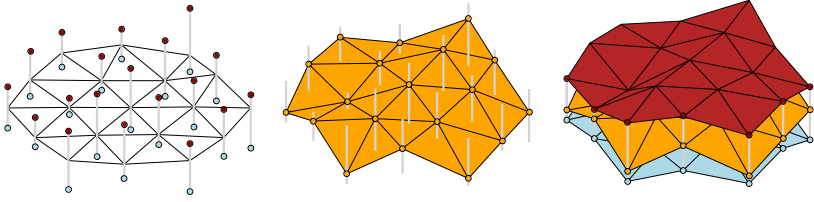
Fig. 1. Left: An imprecise terrain. Each vertex of the triangulation has a elevation interval (gray). Center: a realization of the imprecise terrain. Right: the same realization together with the *highest* and *lowest* possible realizations of the imprecise terrain.

along the steepest incident edge. Assuming the elevation data is exact, drainage networks can be computed efficiently in this model (e.g. [3]). In computational geometry and topology, researchers have studied flow path and drainage network computations on triangulated polyhedral surfaces (e.g. [4,6,14]). In this model, which we call the *surface model*, the flow of water can be traced across the surface of a triangle. This avoids creating certain artifacts that arise when working with grid models. However, the computations on polyhedral surfaces may be significantly more difficult than on network models [5].

Naturally, all computations based on terrain data are subject to various sources of uncertainty, like measurement, interpolation, and numerical errors. The GIS community has long recognized the importance of dealing with uncertainty explicitly, in particular for hydrological modeling. A possible way to deal with this imprecision is to model the elevation at a point of the terrain using stochastic methods [19]. However, the models available in the hydrology literature are unsatisfactory [15] and computationally expensive [18]. A particular challenge is posed by the fact that hydrological computations can be extremely sensitive to small elevation errors [10,13].

A non-probabilistic model of imprecision that has received some attention in computational geometry consists in representing an imprecise attribute (such as location) by a region that is guaranteed to contain the true value. This approach has also been applied to polyhedral terrains (e.g. [8]), replacing the exact elevation of each surface point by an imprecision interval (see Figure 1). In this way, each terrain vertex does not have one fixed elevation, but a whole range of possible elevations which includes the true elevation. Assuming error only in the z-coordinate (and not in the x, y-coordinates) is motivated by the fact that error in the x, y-coordinates normally produces elevation error, and that commercial terrain data suppliers often only report elevation error [7]. Choosing a concrete elevation for each vertex results in a *realization* of the imprecise terrain. The realization is a (precise) polyhedral terrain. Since the set of all possible realizations is guaranteed to include the true (unknown) terrain, one can now obtain bounds on parameters of the true terrain by computing the best- and worst-case values of these parameters over the set of all possible realizations.

In this paper we apply this model of imprecise terrains to problems related to the simulation of water flow, both on terrains represented by surface models and

on terrains represented by network models. The *watershed* of a point in a terrain is the part of the terrain that drains to this point. One of the most fundamental questions one may ask about water flow on terrains is whether water flows from a point p to another point q. In the context of imprecise terrains, reasonable answers may be "definitely not", "possibly", and "definitely". Phrasing the same question in terms of watersheds leads us to introduce the concepts of *potential* (maximal) and *core* and *persistent* (minimal) watersheds.

Results. In Section 2 we show that the problem of deciding whether water can flow between two given points in the surface model is NP-hard. Fortunately, the situation is much better for the network model, and therefore as a special case also for the D-8 model which is widely adopted in GIS applications. In Section 3 we present an algorithm to compute the potential watershed of a point in this model. On a terrain with n edges, our algorithm runs in $O(n \log n)$ time; for grid models the running time can even be improved to $O(n)$. We extend these techniques and achieve the same running times for computing the core and persistent watersheds of a point and its potential downstream area.

Definitions and Notation. We define an ***imprecise terrain*** T as a possibly non-planar geometric graph in $\mathbb{R}^2$ in which each node $v \in \mathbb{R}^2$ has an imprecise third coordinate, which represents its **elevation**. We denote the bounds of the elevation of v with $low(v)$ and $high(v)$. A ***realization*** R of an imprecise terrain T consists of the given graph together with an assignment of elevations to nodes, such that for each node v its elevation $elev_R(v)$ is at least $low(v)$ and at most $high(v)$. We denote with R^- the realization, such that $elev_{R^-}(v) = low(v)$ for every vertex v and similarly the realization R^+, such that $elev_{R^+}(v) = high(v)$. We denote the set of all realizations of an imprecise terrain T with $\mathcal{R}_T$.

2 NP-Hardness in the Surface Model

In the ***surface model*** water flows across the surface of a polyhedral terrain. This surface is formed by the realization of an imprecise terrain as defined above, where the graph that represents the terrain forms a planar triangulation in the (x, y)-domain. The water that arrives at any particular point p on this surface will always follow the true direction of steepest descent at p across the surface, possibly across the interior of a triangle. In this section we prove that it is NP-hard to decide whether water potentially flows from a point s to another point t in this model. The reduction is from 3-SAT; the input is an instance with n variables and m clauses.

We present a global description of the proof, the technical details are left to the full version of the paper. The construction, depicted in Figure 2, consists of a grid with $O(m) \times O(n)$ squares, where each clause corresponds to a column and each variable to a row of the grid; the construction also contains some columns and rows that do not directly correspond to clauses and variables. The grid is placed across the slope of a "mountain" with a shape similar to that of
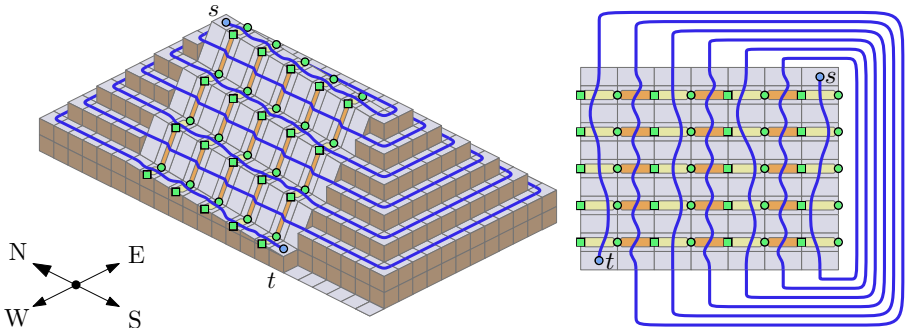
Fig. 2. Left: Global view of the NP-hardness construction, showing the grid on the mountain slope (all vertical faces can be made non-vertical). The fixed parts are shown in gray, the variable parts are shown yellow (for divider gadgets) and orange (for connector gadgets). Right: Top down view of the grid showing the locations of the gadgets, and the $n \times 2m$ green vertices, the only ones with imprecise elevations.

a pyramidal frustum; columns are oriented north-south and rows are oriented east-west. The idea is to create a spiraling water flow path from s at the top of the mountain, through all these clause constructions, to t. The water can reach t if and only if the 3-SAT formula can be satisfied.

The key element in the construction is the *divider* gadget (Figure 3, left), which is placed at every intersection of a clause column and a variable row. It contains two imprecise vertices with a long edge between them and ensures that only if the two imprecise vertices are at opposite extreme elevations, any water can pass the gadget, otherwise it will flow to a local minimum.

In order to link the values of the elevations of the imprecise vertices in the divider gadgets that belong to the same variable together, we need to make sure that neighboring vertices have opposite extremal elevations, just like in divider gadgets. For this, we use a *connector gadget*, which has basically the same construction as the divider gadget, see Figure 3 (right). A complicating factor is that the elevations of these vertices differ vastly. As soon as a water stream enters the imprecise triangle, it will plummet toward the west. But since the direction of steepest descent on this part of the terrain is still toward the south, if only slightly, the water will still cross the variable triangles if we make them sufficiently narrow. As in the divider gadget, we only let the water escape if the elevations of the imprecise vertices are at opposite extremes.

Thus the connector gadgets link the divider gadgets in each variable row together such that a spiraling flow path from s to t can only cross each gadget successfully if either (i) all divider gadgets have their west-side vertex on a low elevation and their east-side vertex on a high elevation, or (ii) all divider gadgets have their west-side vertex on a high elevation and their east-side vertex on a low elevation. This defines two valid states of each row: we let low west-side vertices correspond to the variable being true, and we let low east-side vertices correspond to the variable being false.
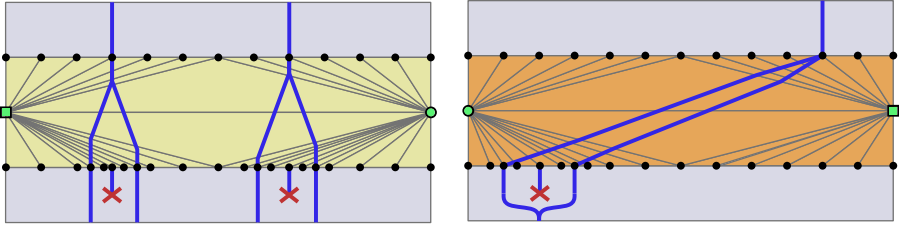
Fig. 3. Left: A divider gadget contains two imprecise vertices with an edge between them. Right: A connector gadget. The triangles must be much narrower, and the water streams need to be much closer to the center of the construction than in the picture.

Across each divider gadget, water may flow in several courses, which may each veer off to the west or the east, depending on the elevations of the imprecise vertices: to the west if the variable is true, or to the east if the variable is false. To encode each clause, we let the water flow to a local minimum if and only if the clause is not satisfied. The figure to the right shows an example. We define possible water courses for each of the eight possible combinations of truth values for the variables in the clause. The course that water would take if the clause is not satisfied leads to a local minimum; the other seven courses merge again into one after passing all divider gadgets. These possible water courses will also cross divider gadgets of variables that are not part of the clause: in that case, each course splits into two courses, which are merged again immediately after emerging from the divider gadget.
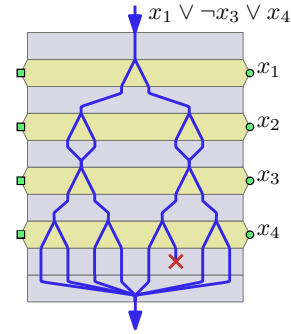
Thus water can only flow through each clause plateau if the variables are such that each clause is true. Therefore, we conclude that deciding whether there exists a realization of T such that water can flow from s to t is NP-hard. The exact coordinates of the vertices in our construction, and the proof that with these coordinates flow is directed as required, can be found in the full version of the paper.

3   Watersheds in the Network Model

An imprecise terrain is defined as a semi-embedded graph as described in Section 1. A realization is an embedded graph. In the **network model** we assume that water only flows along the edges of this graph. The water that arrives at a node of the graph continues along one of the downwards pointing edges incident to this node. The steepness of descent (slope) along an edge (p, q) in a realization R is defined as $\sigma_R(p, q) = (elev_R(p) - elev_R(q))/|pq|$, where $|pq|$ is

the distance between p and q in the plane. The water that arrives at a particular node p, flows to a neighbor q, such that $\sigma_R(p, q)$ is positive and maximal over all edges incident to p. If water from p reaches q in a realization R we write $p \underset{R}{\rightarrow} q$ ("p flows to q in R"); note that we have $p \underset{R}{\rightarrow} p$ for all p, R. Now, on horizontal edges water may flow in either direction, and if the steepest descent neighbor of a node is not unique, then water may leave this node on multiple edges. However, for simplicity of exposition, we assume in this abstract that the steepest descent neighbor is always unique and that edges are never horizontal in the realizations considered. The issues with possible ties and horizontal flow are discussed and resolved in the full version of the paper.

For any set of nodes Q, we define its neighborhood as the set $\{s : (s, t) \in E, t \in Q, s \notin Q\}$, that is, the set of nodes outside Q that are adjacent to nodes of Q. Given a realization R, we call a node q a **local minimum** if all nodes in the neighborhood of q have elevation higher than $elev_R(q)$. We also call a connected set of nodes at the same elevation a local minimum, if all nodes in its neighborhood have higher elevation. Water that arrives in any local minimum Q does not continue to flow to any node outside Q.

The **watershed** of a vertex q in a realization R is defined as the set $\mathcal{W}(R, q) = \{p : p \underset{R}{\rightarrow} q\}$. The **potential watershed** of a node q in a terrain T is

$$\mathcal{W}_\cup(q) = \bigcup_{R \in \mathcal{R}_T} \mathcal{W}(R, q),$$

that is, it is the set of points p for which there exists a realization R, such that water flows from p to q. Similarly, we can define $\mathcal{W}_\cap(q) = \bigcap_{R \in \mathcal{R}_T} \mathcal{W}(R, q)$, which is the set of points from which water flows to q in every realization. We call this the **core watershed** of a node q. We will discuss possible issues with this definition in Section 3.3.

3.1   Potential Watersheds

We prove that for any given node q in an imprecise terrain, there exists a realization R such that $\mathcal{W}(R, q) = \mathcal{W}_\cup(q)$. For this we introduce the notion of the overlay of a set of watersheds. Informally, the overlay sets every node that is contained in one of these watersheds to the lowest elevation it has in any of these watersheds.

Definition 1. *Given a set of watersheds* $\mathcal{W}(R_1, q_1), ..., \mathcal{W}(R_k, q_k)$, *we define their* **watershed-overlay** *as the realization* R^* *such that for every node* v, *we have that* $elev_{R^*}(v) = high(v)$ *if* $v \notin \bigcup \mathcal{W}(R_i, q_i)$ *and otherwise*

$$elev_{R^*}(v) = \min_{i : v \in \mathcal{W}(R_i, q_i)} elev_{R_i}(v).$$

Lemma 1. *Let* R^* *be the watershed-overlay of* $\mathcal{W}(R_1, q), \ldots, \mathcal{W}(R_k, q)$, *then* $\mathcal{W}(R^*, q)$ *contains* $\mathcal{W}(R_i, q)$, *for any* $i \in \{1, \ldots, k\}$.

Proof. Let u be a node of the terrain, which is contained in one of the given watersheds. Let R_i be a realization from $R_1, ..., R_k$ such that $elev_{R^*}(u) = elev_{R_i}(u)$. To prove the lemma, we show that u is contained in $\mathcal{W}(R^*, q)$ by induction on increasing elevation of u in R^*. The base case is that u is equal to q, and in this case the claim holds trivially.

Now, consider the node v which is reached from u by taking the steepest descent edge in R_i. Since $elev_{R^*}(v) \leq elev_{R_i}(v) < elev_{R_i}(u) = elev_{R^*}(u)$, it holds that v lies lower than u in R^*. Therefore, by induction, $v \in \mathcal{W}(R^*, q)$. If v is still the steepest descent neighbor of u in R^*, then this implies $u \in \mathcal{W}(R^*, q)$. Otherwise, there is a node $\hat{v}$ such that $\sigma_{R^*}(u, \hat{v}) > \sigma_{R^*}(u, v)$. There must be an R_j such that $\hat{v} \in \mathcal{W}(R_j, q)$, since otherwise, by construction of the watershed-overlay, we have $elev_{R^*}(\hat{v}) = high(\hat{v}) \geq elev_{R_i}(\hat{v})$ and thus, $\sigma_{R_i}(u, \hat{v}) \geq \sigma_{R^*}(u, \hat{v}) > \sigma_{R^*}(u, v) \geq \sigma_{R_i}(u, v)$ and v would not be the steepest descent neighbor of u in R_i. Therefore, by induction, also $\hat{v} \in \mathcal{W}(R^*, q)$ and, again, $u \in \mathcal{W}(R^*, q)$.   □

The above lemma implies that for any node q, the watershed-overlay $R_\cup(q)$ of $\mathcal{W}(R, q)$ over all possible realizations $R \in \mathcal{R}_T$, realizes the potential watershed of q, that is, $\mathcal{W}_\cup(q) = \mathcal{W}(R_\cup(q), q)$. Therefore, we call $R_\cup(q)$ the **canonical realization** of the potential watershed $\mathcal{W}_\cup(q)$.

Remark 1. There is a natural extension of the definitions given above to the watershed of a set of nodes Q. This would be the set of nodes such that water flows to at least one node in Q. The lemma given above and the algorithms that follow can also be applied in this case. For simplicity of exposition we only discuss the algorithms for single nodes.

Algorithm. Next, we describe how to compute $\mathcal{W}_\cup(q)$ and its canonical realization for a given node q. The idea of the algorithm is to compute the nodes of $\mathcal{W}_\cup(q)$ and their elevations in the canonical realization in increasing order of elevation, similar to the way in which Dijkstra's shortest path algorithm computes distances from the source. The algorithm is laid out in Algorithm 1. A key ingredient of the algorithm is a subroutine, EXPAND(q', z'), defined as follows.

Definition 2. *Let* EXPAND(q', z') *denote a function that returns for a node q' and an elevation $z' \in [low(q'), high(q')]$ the set of neighbors P of q', such that for each $p \in P$, there exists a realization R with $elev_R(q') \in [z', high(q')]$, such that $p \xrightarrow{R} q'$. In particular, it returns tuples of the form (p, z), where z is the minimum elevation of p over all such realizations R.*

Preprocessing. Before presenting the algorithm for the expansion of a node, we discuss a data structure that allows us to make the algorithm more efficient.

We define the **slope diagram** of a node p as the set of points $\hat{q}_i = (\delta_i, high(q_i))$, such that q_i is a neighbor of p and δ_i is its distance to p in the (x, y)-projection. Let $q_1, q_2, ...,$ be a subset of the neighbors of p indexed such that $\hat{q}_1, \hat{q}_2, ...$ appear in counter-clockwise order along the boundary of the convex hull of the slope diagram, starting from the leftmost point and continuing to the lowest point. We ignore neighbors that do not lie on this lower left chain.

Algorithm 1. COMPUTEPWS(q)

1: Enqueue (q, z) with key $z = low(q)$
2: **while** the Queue is not empty **do**
3:     $(q', z') =$ DequeueMin()
4:     **if** q' is not already in the output set **then**
5:         Output q' and set $elev_{R^*}(q') = z'$
6:         Enqueue each $(p, z) \in$ EXPAND(q', z')

Let H_i be the halfplane in the slope diagram that lies above the line through $\widehat{q}_i$ and $\widehat{q}_{i+1}$. Let $U(p)$ be the intersection of these halfplanes $H_1, H_2, ...$, the halfplane right of the vertical line through the leftmost point, and the halfplane above the horizontal line through the bottommost point of the convex chain, see the shaded area in the figure. We compute $U(p)$ for all nodes p of the terrain in a preprocessing phase.
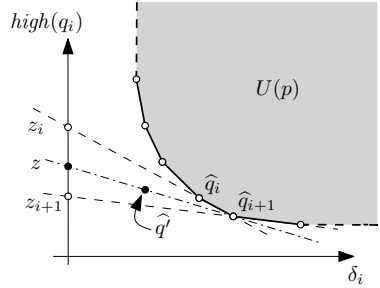
Expansion of a node. For a neighbor p of q', we can now compute the elevation of p as it should be returned by EXPAND(q', z') by computing the lower tangent to $U(p)$ which passes through the point $\widehat{q'} = (\delta', z')$, where δ' is the distance from q' to p in the (x, y)-projection. This can be done via a binary search on the boundary of $U(p)$. Intuitively, this tangent intersects the corner of $U(p)$ which corresponds to the neighbor of p that the node q' has to compete with for being the steepest-descent neighbor of p. The elevation z at which the tangent intersects the vertical axis, is the lowest elevation of p such that q' wins, see the figure. See the full version of this paper for a full description and special cases. The computations can be done in time logarithmic in the degree of p. This leads to the following lemma and theorem.

Lemma 2. *After precomputations in $O(n \log d_{\max})$ time and $O(n)$ space, the algorithm EXPAND(q', z') can be implemented to run in $O(d \log d_{\max})$ time, where d is the node degree of q', $d_{\max}$ is the maximum node degree in the terrain, and n is the number of edges of the terrain.*

Theorem 1. *Algorithm COMPUTEPWS(q) computes the potential watershed $\mathcal{W}_\cup(q)$ of a node q and its canonical realization $R_\cup(q)$ in time $O(n \log n)$, where n is the number of edges in the terrain. The canonical realization of the potential watershed $\mathcal{W}_\cup(Q)$ of a set of nodes Q can be computed in the same time.*

To prove Theorem 1 we use an induction on the nodes extracted from the priority queue in the order of their extraction. The full proof can be found in the full version of the paper. In the analysis of the running time, Lemma 2 implies that the total time spent on expanding nodes is $O(n \log d_{\max})$. For grid terrains,

$d_{\max} = O(1)$, and thus, preprocessing and expansions take only $O(n)$ time. We can use the techniques from Henzinger et al. [11] for shortest paths to overcome the priority queue bottleneck, and obtain the following result (details in the full version):

Theorem 2. *The potential watershed of a set of cells Q and its canonical realization in an imprecise grid terrain of n cells can be computed in $O(n)$ time.*

3.2 Core Watersheds

In this section we show how to compute the core watershed $\mathcal{W}_\cap(q)$ of a given node q, which is the set of nodes for which water *always* flows to q. Observe that this set is the complement of the set of nodes, for which it is possible that water does *not* flow to q in some realization. This nice observation enables us to give an efficient algorithm that is based on computing the potential watersheds of nodes where the water would otherwise flow to, if it does not flow to q.

More specifically, we can characterize this set as follows. Clearly, a node cannot be in the core watershed $\mathcal{W}_\cap(q)$ if it is not in the potential watershed $\mathcal{W}_\cup(q)$. Furthermore, it may not flow to q if it is a local minimum. We call a node r for which there exists a realization in which r is a local minimum a **potential local minimum**. Now, a node p of the terrain is contained in $\mathcal{W}_\cap(q)$ *unless* one of the following holds:

(i) p is a potential local minimum (unless $p = q$);
(ii) $p \notin \mathcal{W}_\cup(q)$;
(iii) There exists a realization R in which $p \xrightarrow{R} r$, where r is of type (i) or (ii) and q is not on the flow path from p to r.

Nodes of type (i) are easy to identify in $O(n)$ time, and nodes of type (ii) can be identified in $O(n \log n)$ time by computing $\mathcal{W}_\cup(q)$ and taking the complement. In order to identify the nodes of type (iii), we define *r-avoiding potential watersheds*. Note that it is different from the potential watershed of q in the terrain T' that is obtained by removing r and its incident edges from T.

Definition 3. *The **r-avoiding potential watershed** of a set of nodes Q is the set of nodes p, such that there is a realization R and a node $q \in Q$ such that $p \xrightarrow{R} q$ and r is not on the flow path from p to q. We denote this set with $\mathcal{W}_\cup^{\backslash r}(Q)$.*

Lemma 3. *There is an algorithm which outputs the r-avoiding potential watershed of Q and takes time $O(n \log n)$, where n is the number of edges of the terrain. (proof in the full version)*

Note that we now have:

$$\mathcal{W}_\cap(q) = V \setminus \mathcal{W}_\cup^{\backslash q}\left(L \cup (V \setminus \mathcal{W}_\cup(q))\right),$$

where V denotes the set of all nodes of the terrain and L denotes the set of potential local minima. By applying Lemma 3 to compute the complement of the nodes of type (iii), we obtain:

Corollary 1. *We can compute the core watershed $\mathcal{W}_\cap(q)$ of q in time $O(n \log n)$, where n is the number of edges of the terrain.*

3.3   Persistent Watersheds – An Alternative Definition

Although the definition of core watersheds as presented above is very natural in its context, it is questionable whether this definition is meaningful enough for practical purposes. Observe that water can "get stuck" in a potential local minimum as soon as the imprecision intervals of neighboring nodes overlap in the vertical dimension. This would not only happen in relatively flat terrains, but could also lead to problems in non-flat terrains. Consider the case of a measuring device with a constant elevation error. It is possible that, by increasing the density of measurement points, the extent of a core watershed can be reduced arbitrarily. Based on these considerations we propose a definition of a *persistent* watershed, which can be computed using the same techniques.

Definition 4. *Let V be the nodes of the graph that define an imprecise terrain. We define the* **persistent watershed** *of a given node q as the set*

$$\mathcal{W}_\cap(q) = V \setminus \mathcal{W}_\cup^{\setminus q} (V \setminus \mathcal{W}_\cup(q)) .$$

This is the set of nodes that do not have a potential flow path to a node outside the potential watershed of q, unless this path goes through q.

To be able to design data structures that store imprecise watersheds and answer queries about flow of water between nodes efficiently, it would be convenient if the watersheds satisfy the following ***nesting condition***: if p is contained in the watershed of q, then the watershed of p is contained in the watershed of q. *Core* watersheds satisfy this condition. However, *persistent* watersheds are not nested in this way, a counter-example can be found in the full version of the paper. To overcome this limitation, we propose the following regularity condition:

Definition 5. *Given an imprecise terrain T, let S be any set of nodes that forms a local minimum in R^-. We call T a* **regular** *terrain if any such set S contains a local minimum in any realization and no proper subset $S' \subset S$ has this property.*

On regular terrains, the persistent watersheds satisfy the nesting condition: when $p \in \mathcal{W}_\cap(q)$, we have $\mathcal{W}_\cap(p) \subseteq \mathcal{W}_\cap(q)$, and even $\mathcal{W}_\cup(p) \subseteq \mathcal{W}_\cup(q)$. However, even on regular terrains the potential watersheds are generally not nested. Interestingly, it may also happen that a persistent watershed is not simply connected. These results can be found in the full version of the paper.

The regularity condition could be guaranteed by a preprocessing step which raises the lower bounds on the elevations such that local minima that violate the regularity condition are removed from the lowest possible realization of the terrain. We could do so with the algorithm from Gray et al. [9] while still respecting the given upper bounds on the elevations. Indeed, in hydrological applications it is common practice to preprocess terrains by removing local minima before doing flow computations [16].

3.4   Potential Downstream Areas

Similar to the potential watershed of q, we can define the set of points that potentially *receive* water from q. Naturally, there exists no canonical realization for this set, however, it can be computed in a similar way as described in Section 3.1 using a priority queue that processes nodes in decreasing order of their maximal elevation such that they would still receive water from q. The following result can be found in the full version of the paper.

Theorem 3. *Given a set of nodes Q of an imprecise terrain, we can compute the set $\bigcup_{R \in \mathcal{R}_T} \{p : \exists\, q \in Q \text{ s.t. } q \xrightarrow{R} p\}$ in time $O(n \log n)$, where n is the number of edges in the terrain.*

4   Conclusions

In this paper we studied flow computations on imprecise terrains under two general models of water flow. For the surface model we showed NP-hardness for deciding whether water can flow between two points. For the network model we gave efficient algorithms to compute potential, core, and persistent watersheds. Our algorithms also work for sets of nodes (lakes, river beds, etc.), and can be modified for related concepts, such as potential downstream areas.

An important contribution of this paper is at a conceptual level. Surprisingly, the most natural definitions of a minimal watershed lack properties that seem natural, most notably the lack of robustness of the core watersheds in the presence of overlapping elevation intervals, and the fact that persistent watersheds are not nested. Interestingly, there are some parallels to observations made in the GIS literature. Firstly, Hebeler et al. [10] observe that the watershed is more sensitive to elevation error in "flatlands". Secondly, simulations have shown that also potential local minima or "small sub-basins" can severely affect the outcome of hydrological computations [13]. We propose a regularity assumption for terrains for future research on efficient data structures for watershed hierarchies. However, we also leave it as an open question whether an alternative, more robust and informative definition of *persistent water flow* can be proposed.

Furthermore, the contrast between the results in Section 2 and Section 3 leaves room for further research questions, e.g., can we develop a model to measure the quality of approximations of water flow, and how does it relate to the network model? Other flow models have been proposed in the GIS literature, e.g., D-∞, in which the incoming water at a vertex is distributed among the outgoing descent edges according to steepness. These models can be seen as modified network models which approximate the steepest descent direction more truthfully. In order to apply the techniques we developed for watersheds, we first need to formalize *to which extent* a node is part of a watershed in these models.

References

1. Borga, M., Gaume, E., Creutin, J., Marchi, L.: Surveying flash floods: gauging the ungauged extremes. Hydrological Processes 22, 3883–3885 (2008)
2. Craddock, W., Kirby, E., Harkins, N., Zhang, H., Shi, X., Liu, J.: Rapid fluvial incision along the Yellow River during headward basin integration. Nature Geoscience 3, 209–213 (2010)
3. Danner, A., Mølhave, T., Yi, K., Agarwal, P.K., Arge, L., Mitásová, H.: TerraStream: from elevation data to watershed hierarchies. In: Proc. 15th ACM Int. Symp. on Geographic Information Systems (ACM-GIS 2007), pp. 212–219 (2007)
4. de Berg, M., Cheong, O., Haverkort, H., Lim, J.-G., Toma, L.: The complexity of flow on fat terrains and its I/O-efficient computation. Comput. Geom. Theory Appl. 43(4), 331–356 (2010)
5. de Berg, M., Haverkort, H., Tsirogiannis, C.: Flow on noisy terrains: An experimental evaluation. In: 27th Eur. Worksh. Comput. Geom., pp. 111–114 (2011)
6. de Berg, M., Haverkort, H., Tsirogiannis, C.: Implicit flow routing on terrains with applications to surface networks and drainage structures. In: Proc. 22nd ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 285–296 (2011)
7. Fisher, P.F., Tate, N.J.: Causes and consequences of error in digital elevation models. Progress in Physical Geography 30(4), 467–489 (2006)
8. Gray, C., Evans, W.: Optimistic shortest paths on uncertain terrains. In: Proc. 16th Canad. Conf. on Comput. Geom., pp. 68–71 (2004)
9. Gray, C., Kammer, F., Löffler, M., Silveira, R.I.: Removing local extrema from imprecise terrains. In: CoRR, abs/1002.2580 (2010)
10. Hebeler, F., Purves, R.: The influence of elevation uncertainty on derivation of topographic indices. Geomorphology 111(1-2), 4–16 (2009)
11. Henzinger, M., Klein, P., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. J. Computer and System Sciences 55(1), 3–23 (1997)
12. Koster, R.D., Mahanama, S.P.P., Livneh, B., Lettenmaier, D.P., Reichle, R.H.: Skill in streamflow forecasts derived from large-scale estimates of soil moisture and snow. Nature Geoscience 3(9), 613–616 (2010)
13. Lindsay, J., Evans, M.: The influence of elevation error on the morphometrics of channel networks extracted from DEMs and the implications for hydrological modelling. Hydrological Processes 22(11), 1588–1603 (2008)
14. Liu, Y., Snoeyink, J.: Flooding triangulated terrain. In: Proc. 11th Int. Symp. Spatial Data Handling, Berlin, pp. 137–148 (2005)
15. Montanari, A.: What do we mean by 'uncertainty'? The need for a consistent wording about uncertainty assessment in hydrology. Hydr. Proc. 21, 841–845 (2006)
16. Tarboton, D.: A new method for the determination of flow directions and upslope areas in grid dig. elev. models. Water Resources Research 33(2), 309–319 (1997)
17. Tetzlaff, D., McDonnell, J., Uhlenbrook, S., McGuire, K., Bogaart, P., Naef, F., Baird, A., Dunn, S., Soulsby, C.: Conceptualizing catchment processes: simply too complex? Hydrological Processes 22, 1727–1730 (2008)
18. Vrugt, J.A., Diks, C.G.H., Gupta, H.V., Bouten, W., Verstraten, J.M.: Improved treatment of uncertainty in hydrologic modeling: Combining the strengths of global optimization and data assimilation. Water Resources Research 41 (2005)
19. Wechsler, S.P.: Uncertainties associated with digital elevation models for hydrologic applications: a review. Hydrology and Earth System Sc. 11(4), 1481–1500 (2007)

Tracking Moving Objects with Few Handovers

David Eppstein, Michael T. Goodrich, and Maarten Löffler

Dept. of Computer Science, Univ. of California, Irvine

Abstract. We study the online problem of assigning a moving point to a base-station region that contains it. Our goal is to minimize the number of *handovers* that occur when the point moves outside its assigned region and must be assigned to a new one. We study this problem in terms of a competitive analysis measured as a function of Δ, the *ply* of the system of regions, that is, the maximum number of regions that cover any single point.

1 Introduction

A common problem in wireless sensor networks involves the online tracking of moving objects [6, 9, 14, 20, 21]. Whenever a moving object leaves a region corresponding to its tracking sensor, a nearby sensor must take over the job of tracking the object. Similar *handovers* are also used in cellular phone services to track moving customers [16]. In both the sensor tracking and cellular phone applications, handovers involve considerable overhead [6, 9, 14, 16, 21], so we would like to minimize their number.

Geometrically, we can abstract the problem in terms of a set of n closed regions in $\mathbb{R}^d$, for a constant d, which represent the sensors or cell towers. We assume that any pair of regions intersects at most a constant number of times, as would be the case, say, if they were unit disks (a common geometric approximation used for wireless sensors [6, 9, 14, 21]). We also have one or more moving entities, which are represented as points traveling along 1-dimensional curves (which we do not assume to be smooth, algebraic, or otherwise well-behaved, and which may not be known or predictable by our algorithms) with a time stamp associated to each point on the curve (Figure 1).

We need to track the entities via regions that respectively contain them; hence, for each moment in time, we must assign one of the regions to each entity, p, with the requirement that p is inside its assigned region at each moment in time. Finally, we
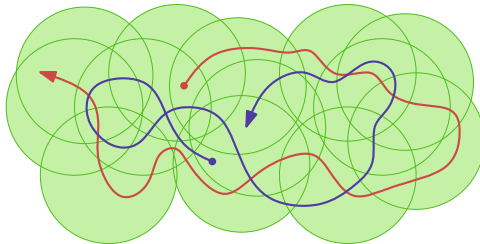

Fig. 1. Example input

want to minimize the number of times that we must change the assignment of the region tracking an entity, so as to minimize the number of handovers.

We also consider a generalized version of this problem, where each point p is required to be assigned to c regions at each moment in time. This generalization is motivated by the need for *trilateration* in cellular networks and wireless sensor networks (e.g., see [18]), where directional information from three or more sensors is used to identify the coordinates of a moving point.

Related Work. There has been considerable previous work in the wireless sensor literature on mobile object tracking. So, rather than providing a complete review of this area, let us simply highlight some of the most relevant work from the wireless sensor literature.

Zhou *et al.* [21] introduce the idea of using handovers to reduce energy in mobile object tracking problems among wireless sensor networks. Pattem *et al.* [14] study energy-quality trade-offs for various strategies of mobile object tracking, including one with explicit handovers. He and Hou [9] likewise study mobile object tracking with respect to handover minimization, deriving probabilistic upper and lower bounds based on distribution assumptions about the moving objects and wireless sensors. Ghica *et al.* [6] study the problem of tracking an object among sensors modeled as unit disks so as to minimize handovers, using probabilistic assumptions about the object's future location while simplifying the tracking requirements to discrete epochs of time.

The analysis tool with which we characterize the performance of our algorithms comes from research in *online algorithms*, where problems are defined in terms of a sequence of decisions that must be made one at a time, before knowing the sequence of future requests. In *competitive analysis* [15], one analyzes an online algorithm by comparing its performance against that of an idealized adversary, who can operate in an offline fashion, making his choices after seeing the entire sequence of items.

We are not aware of any previous work that applies competitive analysis to the problem of handover minimization. Nevertheless, this problem can be viewed from a computational geometry perspective as an instantiation of the *observer-builder* framework of Cho *et al.* [2], which itself is related to the *incremental motion* model of Mount *et al.* [12], the *observer-tracker* model of Yi and Zhang [20], and the well-studied *kinetic data structures* framework [8, 7]. In terms of the observer-builder model, our problem has an *observer* who watches the motion of the point(s) we wish to track and a *builder* who maintains the assignment of tracking region(s) to the point(s). This assignment would define a set of Boolean *certificates*, which become *violated* when a point leaves its currently-assigned tracking region. The observer would notify the builder of any violation, and the builder would use information about the current and past states of the point(s) to make a new assignment (and define an associated certificate). The goal, as in the previous work by Cho *et al.* [2], would be to minimize the number of interactions between the observer and builder, as measured using competitive analysis. Whereas Cho *et al.* apply their model to the maintenance of net trees for moving points, in our case the interactions to be minimized correspond to handovers, and our results supply the algorithms that would be needed to implement a builder for handover minimization. Yi and Zhang [20] study a general online tracking problem, but with a different objective function than ours: when applied to mobile object tracking, rather than optimizing

the number of handovers, their scheme would aim to minimize the distance between objects and the base-station region to which they are each assigned.

Several previous papers study overlap and connectivity problems for geometric regions, often in terms of their *ply*, the maximum number of regions that cover any point. Guibas *et al.* [7] study the maintenance of connectivity information among moving unit disks in the kinetic data structure framework. Miller *et al.* [11] introduce the concept of ply and show how sets of disks with low ply possess small geometric separators. Eppstein *et al.* [3,5] study road network properties and algorithms using a model based on sets of disks with low ply after outliers are removed. Van Leeuwen [17] studies the minimum vertex cover problem for disk graphs, providing an asymptotic FPTAS for this problem on disk graphs of bounded ply. Alon and Smorodinsky [1] likewise study coloring problems for sets of disks with low ply.

Our problem can also be modeled as a *metrical task system* in which the sensor regions are represented as states of the system, the cost of changing from state to state is uniform, and the cost of serving a request is zero for a region that contains the request point and two for other regions. Known randomized online algorithms for metrical task systems [10] would give a competitive ratio of $O(\log n)$ for our problem, not as good as our $O(\log \Delta)$ result, and known lower bounds for metrical task systems would not necessarily apply to our problem.

New Results. In this paper, we study the problem of assigning moving points in the plane to containing base station regions in an online setting and use the competitive analysis to characterize the performance of our algorithms. Our optimization goal in these algorithms is to minimize the number of *handovers* that occur when an object moves outside the range of its currently-assigned base station and must be assigned to a new base station. We measure the competitive ratio of our algorithms as a function of Δ, the *ply* of the system of base station regions, that is, the maximum number of such regions that cover any single point. When object motions are known in advance, as in the offline version of the object traking problem, a simple greedy strategy suffices to determine an optimal assignment of objects to base stations, with as few handovers as possible. For the online problem, on the other hand, for moving points in one dimension, we present a deterministic online algorithm that achieves a competitive ratio of $O(\log \Delta)$, with respect to the offline optimal algorithm, and we show that no better ratio is possible. For two or more dimensions, we present a randomized algorithm that achieves a competitive ratio of $O(\log \Delta)$, and a deterministic algorithm that achieves a competitive ratio of $O(\Delta)$; again, we show that no better ratio is possible.

2   Problem Statement and Notation

Let $\mathscr{D}$ be a set of n regions in $\mathbb{R}^d$. These regions represent the areas that can be covered by a single sensor. We assume that each region is a closed, connected subset of $\mathbb{R}^d$ and that the boundaries of any two regions intersect $O(1)$ times – for instance, this is true when each region is bounded by a piecewise algebraic curve in $\mathbb{R}^2$ with bounded degree and a bounded number of pieces. With these assumptions, the arrangement of the pieces has polynomial complexity $O(n^d)$. The *ply* of $\mathscr{D}$ is defined to be the maximum over $\mathbb{R}^d$

of the number of regions covering any point. We always assume that $\mathcal{D}$ is fixed and known in advance.

Let T be the trajectory of a moving point in $\mathbb{R}^d$. We assume that T is represented as a continuous and piecewise algebraic function from $[0, \infty)$ to $\mathbb{R}^d$, with a finite but possibly large number of pieces. We also assume that each piece of T crosses each region boundary $O(1)$ times and that it is possible to compute these crossing points efficiently. We also assume that $T([0, \infty)) \subset \cup \mathcal{D}$; that is, that the moving point is always within range of at least one sensor; this assumption is not realistic, and we make it only for convenience of exposition. Allowing the point to leave and re-enter $\subset \cup \mathcal{D}$ would not change our results since the handovers caused by these events would be the same for any online algorithm and therefore cannot affect the competitive ratio.

As output, we wish to report a *tracking sequence S*: a sequence of pairs (τ_i, D_i) of a time τ_i on the trajectory (with $\tau_0 = 0$) and a region $D_i \in \mathcal{D}$ that covers the portion of the trajectory from time τ_i to τ_{i+1}. We require that for all i, $\tau_i < \tau_{i+1}$. In addition, for all i, it must be the case that $T([\tau_i, \tau_{i+1}]) \subseteq D_i$, and there should be no $\tau' > \tau_{i+1}$ for which $T([\tau_i, \tau']) \subset D_i$; in other words, once a sensor begins tracking the moving point, it continues tracking that point until it moves out of range and another sensor must take over. Our goal is to minimize $|S|$, the number of pairs in the tracking sequence. We call this number of pairs the *cost* of S; we are interested in finding tracking sequences of small cost.

Our algorithm may not know the trajectory T completely in advance. In the *offline tracking problem*, T is given as input, and we must find the tracking sequence S that minimizes $|S|$; as we show, a simple greedy algorithm accomplishes this task. In the *online tracking problem*, T is given as a sequence of *updates*, each of which specifies a single piece in a piecewise algebraic decomposition of the trajectory T. The algorithm must maintain a tracking sequence S that covers the portion of T that is known so far, and after each update it must extend S by adding additional pairs to it, without changing the pairs that have already been included. As has become standard for situations such as this one in which an online algorithm must make decisions without knowledge of the future, we measure the quality of an algorithm by its *competitive ratio*. Specifically, if a deterministic online algorithm A produces tracking sequence $S_A(T)$ from trajectory T, and the optimal tracking sequence is $S^*(T)$, then the competitive ratio of A (for a given fixed set $\mathcal{D}$ of regions) is

$$\sup_T \frac{|S_A(T)|}{|S^*(T)|}.$$

In the case of a randomized online algorithm, we measure the competitive ratio similarly, using the expected cost of the tracking sequence it generates. In this case, the competitive ratio is

$$\sup_T \frac{E[|S_A(T)|]}{|S^*(T)|}.$$

As a variation of this problem, stemming from trilateration problems in cellular phone network and sensor network coverage, we also consider the problem of finding tracking sequences with *coverage c*. In this setting, we need to report a set of c tracking sequences $S_1, S_2, \ldots, S_c$ for T that are *mutually disjoint* at any point in time: if a region D appears for a time interval $[\tau_i, \tau_{i+1}]$ in one sequence S_k and a time interval $[\sigma_j, \sigma_{j+1}]$
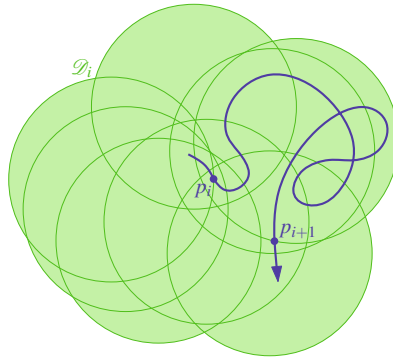
Fig. 2. The set $\mathscr{D}_i$ of disks containing p_i, and the point p_{i+1} where the trajectory leaves the last disk of $\mathscr{D}_i$

in some other sequence S_l, we require that the intervals $[\tau_i, \tau_{i+1}]$ and $[\sigma_j, \sigma_{j+1}]$ are disjoint. We wish to minimize the total cost $\sum_{i=1}^{c} |S_i|$ of a set of tracking sequences with coverage c, and in both the offline and online versions of the problem.

3   Offline Tracking

Due to space restrictions, we defer to the full version [4] our discussion of the offline problem. We describe a simple greedy approach to solve the problem, either in the simple coverage or c-coverage cases, by choosing at each handover the region that will cover the trajectory the longest. Intuitively, the time the moving point spends within each region may be viewed as forming a set of intervals of the real timeline, and we are applying a standard greedy algorithm to find the smallest subset of the intervals that covers the timeline. We prove the following:

Theorem 1. *The greedy algorithm solves the offline tracking problem optimally, in polynomial time.*

4   Online Tracking

We now move on to the dynamic setting. We assume that we are given the start locations of the trajectory, and receive a sequence of updates extending the trajectory. From these updates we can easily generate a sequence of *events* caused when the trajectory crosses into or out of a region. We will describe three algorithms for different settings, which are all based on the following observations.

Let T be the (unknown) trajectory of our moving entity, and recall that $T(\tau)$ denotes the point in space that the entity occupies at time τ. Let τ_0 be the starting time. We will define a sequence of times τ_i as follows. For any i, let $p_i = T(\tau_i)$ be the location of the entity at time τ_i, and let $\mathscr{D}_i \subset \mathscr{D}$ be the set of regions that contain p_i. For each $D_{ij} \in \mathscr{D}_i$, let τ'_{ij} be first the time after τ_i that the entity leaves D_{ij}. Now, let $\tau_{i+1} = \max_j \tau'_{ij}$ be

the moment that the entity leaves the last of the regions in $\mathscr{D}_i$ (note that it may have re-entered some of the regions). Figure 2 shows an example. Let τ_k be the last assigned time (that is, the entity does not leave all disks $\mathscr{D}_k$ before the the end of its trajectory).

Observation 2 *Any tracking sequence S for trajectory T must have length at least k.*

Proof. For any i, a solution must have a region of $\mathscr{D}_i$ at time τ_i. However, since by construction there is no region that spans the entire time interval $[\tau_i, \tau_{i+1} + \varepsilon]$ (for any $\varepsilon > 0$), there must be at least one handover during this time, resulting in at least $k-1$ handovers, and at least k regions.   □

Randomized Tracking with Logarithmic Competitive Ratio. With this terminology in place, we are now ready to describe our randomized algorithm. We begin by computing τ_0, p_0 and $\mathscr{D}_0$ at the start of T. We will keep track of a set of candidate regions $\mathscr{C}$, which we initialize to $\mathscr{C} = \mathscr{D}_0$, and select a random element from the candidate set as the first region to track the entity. Whenever the trajectory leaves its currently assigned region, we compute the subset $\mathscr{C} \subset \mathscr{D}_i$ of all regions that contain the whole trajectory from p_i to the event point, and if $\mathscr{C}$ is not empty we select a new region randomly from $\mathscr{C}$. When $\mathscr{C}$ becomes empty, we have found the next point p_{i+1}, giving us a new nonempty candidate set $\mathscr{C}$. Intuitively, for each point p_i, if the set of candidate regions containing p_i is ordered by their exit times, the selected regions form a random increasing subsequence of this ordering, which has expected length $O(\log \Delta)$, whereas the optimal algorithm incurs a cost of one for each point p_i. Refer to the full version for a more formal description of the algorithm, as well as the proof of the following lemma.

Lemma 1. *The randomized algorithm produces a valid solution of expected length $O(k \log \Delta)$.*

Combining Observation 2 and Lemma 1, we see that the algorithm has a competitive ratio of $O(\log \Delta)$.

Deterministic Tracking with Linear Competitive Ratio. We now describe a deterministic variant of the algorithm. The only thing we change is that, instead of selecting a random member of the set $\mathscr{C}$ of candidate regions, we select an arbitrary element of this set. Here we assume that $\mathscr{C}$ is represented in some deterministic way that we make no further assumptions about. For example, if the elements in $\mathscr{D}$ are unit disks we might store them as a sorted list by the x-coordinate of their center points.

This strategy may seem rather naïve, and indeed produces a competitive ratio that is exponentially larger than that of the randomized strategy of the previous section. But we will see in Section 5 that this is unavoidable, even for the specific case of unit disks.

Again, the full version contains a more formal description of the algorithm, as well as the proof of the following lemma.

Lemma 2. *The deterministic algorithm produces a valid solution of length $O(k \Delta)$.*

As before, combining Observation 2 and Lemma 2, we see that this algorithm has a competitive ratio of $O(\Delta)$.

Deterministic Tracking in One Dimension. In the 1-dimensional case, a better deterministic algorithm is possible. In this case, the regions of $\mathscr{D}$ can only be connected intervals, due to our assumptions that they are closed connected subsets of $\mathbb{R}$.

Now, when we want to pick a new sensor, we have to choose between $c = |\mathscr{C}|$ intervals that all contain the current position of the entity. For each interval C_i, let ℓ_i be the number of intervals in $\mathscr{C} \setminus \{C_i\}$ that contain the left endpoint of C_i, and let r_i be the number of intervals in $\mathscr{C} \setminus \{C_i\}$ that contain the left endpoint of C_i. We say that an interval C_i is *good* if $\max(\ell_i, r_i) \leq c/2$. Our deterministic algorithm simply chooses a good sensor at each step. Figure 3 illustrates this.
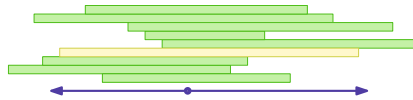

Fig. 3. A set of 8 intervals covering the current location of the entity (blue dot). A good interval is highlighted; this interval has $\ell_i = 3 \leq 8/2$ and $r_i = 2 \leq 8/2$.

The new algorithm is described in the full version, where we also prove the following lemma.

Lemma 3. *The deterministic one-dimensional algorithm produces a valid solution of length $O(k \log \Delta)$.*

Combining Observation 2 and Lemma 3, we conclude that this algorithm also has a competitive ratio of $O(\log \Delta)$.

Summary of Algorithms. Our input assumptions ensure that any trajectory can be transformed in polynomial time into a sequence of events: trivially, for each piece in the piecewise description of the trajectory, we can determine the events involving that piece in time $O(n)$ (where $n = |\mathscr{D}|$) and sort them in time $O(n \log n)$.

Once this sequence is known, it is straightforward to maintain both the set of regions containing the current endpoint of the trajectory, and the set $\mathscr{C}$ of candidate regions, in constant time per event. Additionally, each event may cause our algorithms to select a new region, which may in each case be performed given the set $\mathscr{C}$ in time $O(|\mathscr{C}|) = O(\Delta)$. Therefore, if there are m events in the sequence, the running time of our algorithms (once the event sequence is known) is at most $O(m\Delta)$.

Additionally, geometric data structures (such as those for point location among fat objects [13]) may be of use in more quickly finding the sequence of events, or for more quickly selecting a region from $\mathscr{C}$; we have not carefully analyzed these possibilities, as our focus is primarily on the competitive ratio of our algorithms rather than on their running times.

We summarize these results in the following theorem:

Theorem 3. *Given a set $\mathscr{D}$ of n connected regions in $\mathbb{R}^d$, and a trajectory T,*

- *there is a randomized strategy for the online tracking problem that achieves a competitive ratio of $O(\log \Delta)$; and*
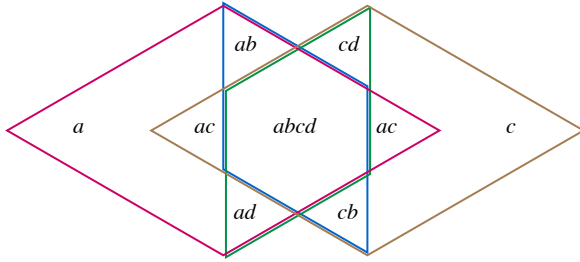
Fig. 4. Four similar rhombi form a set of regions for which no stateless algorithm can be competitive

- *there are deterministic strategies for the online tracking problem that achieve a competitive ratio of $O(\log \Delta)$ when $d = 1$ or $O(\Delta)$ when $d > 1$.*

Each of these strategies may be implemented in polynomial time.

5   Lower Bounds

We now provide several lower bounds on the best competitive ratio that any deterministic or randomized algorithm can hope to achieve. Our lower bounds use only very simple regions in $\mathscr{D}$: similar rhombi, in one case, unit disks in $\mathbb{R}^d$ in a second case, and unit intervals in $\mathbb{R}$ in the third case. These bounds show that our algorithms are optimal, even with strong additional assumptions about the shapes of the regions.

Lower Bounds on Stateless Algorithms. An algorithm is *stateless* if the next sensor that covers the moving point, when it moves out of range of its current sensor, is a function only of its location and not of its previous state or its history of motion. Because they do not need to store and retrieve as much information, stateless algorithms provide a very enticing possibility for the solution of the online tracking problem, but as we show in this section, they cannot provide a competitive solution.

Theorem 4. *There exists a set $\mathscr{D}$ of four similar rhombi in $\mathbb{R}^2$, such that any stateless algorithm for the online tracking problem has unbounded competitive ratio.*

Proof. The set $\mathscr{D}$ is shown in Figure 4. It consists of four rhombi a, b, c, and d; these rhombi partition the plane into regions (labeled in the figure by the rhombi containing them) such that the common intersection $abcd$ of the rhombi is directly adjacent to regions labeled ab, ac, ad, bc, and cd.

Let G be a graph that has the four rhombi as its vertices, and the five pairs ab, ac, ad, bc, and cd as its edges. Let A be a stateless algorithm for $\mathscr{D}$, and orient the edge xy of G from x to y if it is possible for algorithm A to choose region y when it performs a handover for a trajectory that moves from region $abcd$ to region xy. If different trajectories would cause A to choose either x or y, orient edge xy arbitrarily.

Because G has four vertices and five edges, by the pigeonhole principle there must be some vertex x with two outward-oriented edges xy and xz. There exists a trajectory
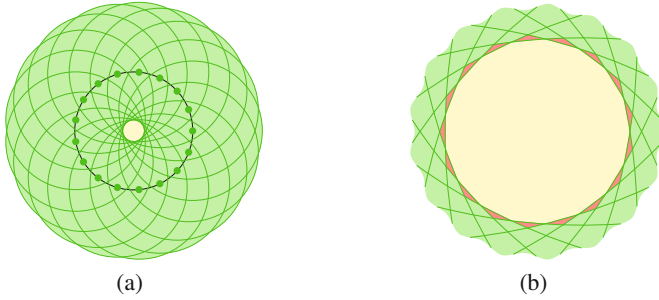
Fig. 5. (a) A set of Δ disks whose centers are equally spaced on a circle. (b) The heart of the construction, zoomed in. The yellow cell is inside all disks; the red cells are inside all but one disk.

T that repeatedly passes from region *abcd* to *xy*, back to *abcd*, to *xz*, and back to *abcd*, such that on each repetition algorithm A performs two handovers, from z to y and back to z. However, the optimal strategy for trajectory T is to cover the entire trajectory with region x, performing no handovers. Therefore, algorithm A has unbounded competitive ratio.   □

Lower Bounds on Deterministic Algorithms. Next, we show that any deterministic algorithm in two or more dimensions must have a competitive ratio of Δ or larger, matching our deterministic upper bound and exponentially worse than our randomized upper bound. The lower bound construction consists of a set of Δ unit disks with their centers on a circle, all containing a common point (Figure 5). The idea is that if the trajectory starts at this common point, it can exit from any single disk, in particular, the one that a deterministic algorithm previously chose.

Theorem 5. *There exists a set $\mathscr{D}$ of unit disks in $\mathbb{R}^2$, such that any deterministic algorithm for the online tracking problem has competitive ratio at least $\Delta-1$.*

Proof. Let $\mathscr{D}$ be a set of Δ unit disks whose centers are equally spaced on a given circle C of slightly less than unit radius, as in Figure 5(a). Let the moving point to be tracked start at the point p_0 at the center of C, in the common interior of all disks. For each disk $D_i \in \mathscr{D}$, there exists a cell X_i in the arrangement that is interior to all disks in $\mathscr{D} \setminus \{D_i\}$, but outside D_i itself. Furthermore, this cell is directly adjacent to the center cell. See Figure 5(b) for an illustration.

Now, let A be any deterministic algorithm for the online tracking problem, and construct a sequence of updates to trajectory T as follows. Initially, T consists only of the single point p_0. At each step, let algorithm A update its tracking sequence to cover the current trajectory, let D_i be the final region in the tracking sequence constructed by algorithm A, and then update the trajectory to include a path to X_i and back to the center.

Since X_i is not covered by D_i, algorithm A must increase the cost of its tracking sequence by at least one after every update. That is, $|S_A(T)| \geq |T|$. However, in the optimal tracking sequence, every $\Delta-1$ consecutive updates can be covered by a single region D_i, so $S^*(T) \leq |T|/(\Delta-1)$. Therefore, the competitive ratio of A is at least $\Delta-1$.   □
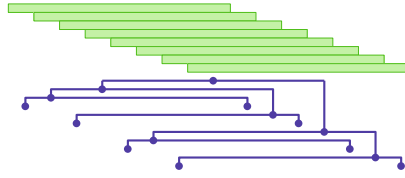
Fig. 6. A set of $\Delta = 8$ intervals, and a tree of 8 different trajectories in $\mathbb{R}^1$ (horizontal dimension)

This construction generalizes to any $d > 2$.

Lower Bounds on Randomized Algorithms. The above lower bound construction uses the fact that the algorithm to solve the problem is deterministic: an adversary constructs a tracking sequence by reacting to each decision made by the algorithm. For a randomized algorithm, this is not allowed. Instead, the adversary must select an entire input sequence, knowing the algorithm but not knowing the random choices to be made by the algorithm. Once this selection is made, we compare the quality of the solution produced by the randomized algorithm to the optimal solution. By Yao's principle [19], finding a randomized lower bound in this model is equivalent to finding a random distribution R on the set of possible update sequences such that, for every possible deterministic algorithm A, the expected value of the competitive ratio of A on a sequence from R is high.

Our lower bound construction consists of Δ unit intervals that contain a common point, and a tree of Δ different possible paths for the moving object to take, each of which leaves the intervals in a different ordering, in a binary tree-like fashion. Half of the trajectories start by going to the left until they are outside the right half of the intervals, the others start towards the right until they are outside the left half of the intervals, and this recurses, as shown in Figure 6.

More formally, let us assume for simplicity that Δ is a power of 2. Let $\mathcal{D}$ be a set of Δ distinct unit intervals in $\mathbb{R}$, containing a common point p_0. For any $k \in [1, \Delta]$ we define point p_k to be a point outside the leftmost k intervals but in the interior of the rest, and p_{-k} to be a point outside the rightmost k intervals but in the interior of the rest.

Now, for each $j \in [1, \Delta]$, we construct a trajectory T_j with $h = \log \Delta$ steps, as follows. We define an index $\xi(j, i)$ for all $j \in [1, \Delta]$ and all $i \in [1, h]$ such that trajectory T_j is at point $p_{\xi(j,i)}$ at step i. At step 0, all trajectories start at $\xi(j, 0) = 0$. Then, at step i:

- all T_j with $j \mod 2^{h-i} \leq 2^{h-i-1}$ move to the left to $\xi(j, i) = \min_{l<i} \xi(j, l) - 2^{h-i}$,
- all T_j with $j \mod 2^{h-i} > 2^{h-i-1}$ move to the right to $\xi(j, i) = \max_{l<i} \xi(j, l) + 2^{h-i}$.

Figure 6 shows $\mathcal{T}$ be the resulting set of these Δ trajectories in a tree representation.

Theorem 6. *There exists a set $\mathcal{D}$ of unit intervals in $\mathbb{R}$, for which any randomized algorithm to solve the online tracking problem has competitive ratio $\Omega(\log \Delta)$.*

Proof. Let $\mathcal{D}$ and the set of trajectories $\mathcal{T}$ be as described above. Let R be a probability distribution over the set of all possible trajectories that has a probability of $1/\Delta$ to be any element of $\mathcal{T}$, and a probability of 0 elsewhere.

Now, let A be any deterministic algorithm for the online tracking problem. At each level of the tree, each region D_i that algorithm A might have selected as the final region in its tracking sequence fails to cover one of the two points that the moving point could move to next, and each of these points is selected with probability $1/2$, so algorithm A must extend its tracking sequence with probability $1/2$, and its expected cost on that level is $1/2$. The number of levels is $\log_2 \Delta$, so the total expected cost of algorithm A is $1 + \frac{1}{2} \log_2 \Delta$, whereas the optimal cost on the same trajectory is 1. Therefore the competitive ratio of algorithm A on a random trajectory with distribution R is at least $1 + \frac{1}{2} \log_2 \Delta$.

It follows by Yao's principle that the same value $1 + \frac{1}{2} \log_2 \Delta$ is also a lower bound on the competitive ratio of any randomized online tracking algorithm.   □

Although the trajectories formed in this proof are short relative to the size of $\mathscr{D}$, this is not an essential feature of the proof: by concatenating multiple trajectories drawn from the same distribution, we can find a random distribution on arbitrarily long trajectories leading to the same $1 + \frac{1}{2} \log_2 \Delta$ lower bound. This construction generalizes to unit balls in any dimension $d > 1$ as well.

6   Trilateration

We can extend our results to the case where the entity needs to covered with c sensors at any time. We refer to the full version for details; in particular, we prove the following theorems:

Theorem 7. *There exists a randomized algorithm that solves the trilateration problem with a competitive ratio of $O(\log(\Delta - c))$.*

Theorem 8. *There exists a set $\mathscr{D}$ of intervals in $\mathbb{R}$ of two different lengths, for which any randomized algorithm to solve the online tracking problem has competitive ratio $\Omega(\log(\Delta - c))$.*

7   Conclusions

We studied the online problem of tracking a moving entity among sensors with a minimal number of handovers, combining the kinetic data and online algorithms paradigms. We provided several algorithms with optimal competitive ratios. Interestingly, randomized strategies are able to provably perform significantly better than deterministic strategies, and arbitrarily better than stateless strategies (which form a very natural and attractive class of algortihms in our application).

We are able to track multiple entities using the same algorithms, by simply treating them independently. As a future direction of research, it would be interesting to study the situation where each sensor has a maximum capacity C, and cannot track more than C different entities at the same time. Another possible direction of research is to analyze and optimize the running times of our strategies for particular classes of region shapes or trajectories, something we have made no attempt at.

References

1. Alon, N., Smorodinsky, S.: Conflict-free colorings of shallow discs. In: Proc. 22nd Symp. on Computational Geometry (SoCG), pp. 41–43. ACM, New York (2006)
2. Cho, M., Mount, D., Park, E.: Maintaining Nets and Net Trees under Incremental Motion. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1134–1143. Springer, Heidelberg (2009)
3. Eppstein, D., Goodrich, M.T.: Studying (non-planar) road networks through an algorithmic lens. In: GIS 2008: Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 1–10 (2008)
4. Eppstein, D., Goodrich, M.T., Löffler, M.: Tracking moving objects with few handovers, arXiv.org/abs/1105.0392 (2011)
5. Eppstein, D., Goodrich, M.T., Trott, L.: Going off-road: transversal complexity in road networks. In: Proc. 17th ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems (ACM GIS), pp. 23–32. ACM, New York (2009)
6. Ghica, O., Trajcevski, G., Zhou, F., Tamassia, R., Scheuermann, P.: Selecting Tracking Principals with Epoch Awareness. In: Proc. 18th ACM SIGSPATIAL Internat. Conf. on Advances in Geographic Information Systems, ACM GIS (2010)
7. Guibas, L., Hershberger, J., Suri, S., Zhang, L.: Kinetic Connectivity for Unit Disks. Discrete Comput. Geom. 25(4), 591–610 (2001)
8. Guibas, L.J.: Kinetic data structures — a state of the art report. In: Agarwal, P.K., Kavraki, L.E., Mason, M. (eds.) Proc. Workshop Alg. Found. Robot., pp. 191–209 (1998)
9. He, G., Hou, J.: Tracking targets with quality in wireless sensor networks. In: 13th IEEE Conf. on Network Protocols (ICNP), pp. 1–12 (2005)
10. Irani, S., Seiden, S.: Randomized algorithms for metrical task systems. Theor. Comput. Sci. 194(1-2), 163–182 (1998)
11. Miller, G.L., Teng, S.-H., Thurston, W., Vavasis, S.A.: Separators for sphere-packings and nearest neighbor graphs. J. ACM 44, 1–29 (1997)
12. Mount, D.M., Netanyahu, N.S., Piatko, C.D., Silverman, R., Wu, A.Y.: A computational framework for incremental motion. In: Proc. 20th Symp. on Computational Geometry (SoCG), pp. 200–209. ACM, New York (2004)
13. Overmars, M., van der Stappen, F.: Range Searching and Point Location among Fat Objects. J. Algorithms 21(3), 629–656 (1996)
14. Pattem, S., Poduri, S., Krishnamachari, B.: Energy-Quality Tradeoffs for Target Tracking in Wireless Sensor Networks. In: Zhao, F., Guibas, L. (eds.) IPSN 2003. LNCS, vol. 2634, pp. 32–46. Springer, Heidelberg (2003)
15. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Commun. ACM 28, 202–208 (1985)
16. Tekinay, S., Jabbari, B.: Handover and channel assignment in mobile cellular networks. IEEE Communications Magazine 29(11), 42–46 (1991)
17. van Leeuwen, E.J.: Better Approximation Schemes for Disk Graphs. In: Arge, L., Freivalds, R. (eds.) SWAT 2006. LNCS, vol. 4059, pp. 316–327. Springer, Heidelberg (2006)
18. Yang, Z., Liu, Y.: Quality of Trilateration: Confidence-Based Iterative Localization. IEEE Trans. on Parallel and Distributed Systems 21(5), 631–640 (2010)
19. Yao, A.: Probabilistic computations: Toward a unified measure of complexity. In: 18th IEEE Symp. on Foundations of Computer Science (FOCS), pp. 222–227 (1977)
20. Yi, K., Zhang, Q.: Multi-dimensional online tracking. In: Proc. of the 20th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 1098–1107. SIAM, Philadelphia (2009)
21. Zhao, F., Shin, J., Reich, J.: Information-driven dynamic sensor collaboration. IEEE Signal Processing Magazine 19(2), 61–72 (2002)

Inducing the LCP-Array

Johannes Fischer*

KIT, Institut für Theoretische Informatik, 76131 Karlsruhe, Germany
johannes.fischer@kit.edu

Abstract. We show how to modify the linear-time construction algorithm for suffix arrays based on *induced sorting* (Nong et al., DCC'09) such that it computes the array of *longest common prefixes* (LCP-array) as well. Practical tests show that this outperforms recent LCP-array construction algorithms (Gog and Ohlebusch, ALENEX'11).

1   Introduction

The *suffix array* [15] is an important data structure in text indexing. It is used to solve many tasks in string processing, from exact and inexact string matching to more involved tasks such as data compression, repeat recognition, and text mining. It is also the basic building block for the more complex text index called the *suffix tree*, either indirectly for index construction, or directly when dealing with *compressed* suffix trees [21]. In all of the above applications (possibly apart from exact string matching), the suffix array is accompanied by its sister-array, the array of longest common prefixes (LCP-array for short).

Since their introduction in the early 1990's, much research has been devoted to the fast construction of suffix arrays. Although it is in principle possible to derive the suffix array from the suffix tree, for which linear-time algorithms had already been discovered earlier [23], for reasons of time and space the aim was to construct the suffix array *directly*, without help of the tree. This long line of research (see [20] for a good reference) culminated in three linear-time algorithms [11,13,14]. However, these algorithms were notorious for being "linear but not fast" [2], as they were slower than other non-linear algorithms that had been discovered before and continued to be discovered afterwards.

This un-satisfactory situation (at least for theoretical practitioners or practical theoreticians, who want linear-time algorithms to perform faster than super-linear ones) changed substantially when in 2009 a new linear-time algorithm based on *induced sorting* was presented [18]. A careful implementation of this approach due to Yuta Mori led to one of the fastest known suffix array construction algorithms, called *sais-lite*, often outperforming all other linear or super-linear implementations known by that time (see http://sites.google.com/site/yuta256/ for some results). Later, the same author came up with an even faster implementation, called *libdivsufsort*, also relying heavily on the idea of induced sorting.

---

* Supported by the German Research Foundation (DFG).

Less emphasis has been put on the efficient construction of the LCP-array. Manber and Myers [15] mentioned that it can be constructed along with their method for constructing the suffix array, but their algorithm ran in $O(n \lg n)$ time and performed rather poor in practice. Kasai et al. [12] gave an elegant algorithm for constructing the LCP-array in linear time, given the suffix array. A few refinements of this algorithm led to improvements in either space [16] or in running time [10]. However, these algorithms could not compete with the carefully tuned algorithms for suffix arrays. This led to the odd situation that the rather difficult task of *sorting* suffixes could be solved faster than the seemingly simpler task of computing longest common prefixes.

This situation changed only recently when a theoretically slow $O(n^2)$ but practically fast LCP-array construction algorithm was presented [7]. Their algorithm exploits properties of the Burrows-Wheeler-Transformation (BWT) of the text, which must be computed before. The authors of [7] also sketch how their approach yields a linear-time algorithm (for constant alphabets, otherwise it takes $O(n \lg \sigma)$ time).

Driven by the success of the fast linear-time algorithm based on induced sorting [18], we show in Sect. 3 of this paper how it can be adapted such that it also induces the LCP-values. This results in a new linear-time algorithm for constructing LCP-arrays (for integer alphabets). In Sect. 4 we show that an ad-hoc implementation of the theoretical ideas leads to a fast practical algorithm that outperforms all other previous algorithms. An additional advantage of our algorithm is that it does not rely on the BWT, and is hence preferable in situations where the BWT is not already present (such as compressed suffix arrays *not* based on the BWT [17], for example).

Before detailing our theoretical and practical contributions, in Sect. 2 we first introduce some notations, and then review the induced sorting algorithm for suffix arrays.

2   Previous Work and Concepts

2.1   Suffix- and LCP-Arrays

Let $T = t_1 \ldots t_n$ be a text consisting of n characters drawn from an ordered alphabet Σ of size $\sigma = |\Sigma|$. The substring of T ranging from i to j is denoted by $T_{i..j}$, for $1 \leq i \leq j \leq n$. The substring $T_{i..n}$ is called the i'th *suffix* of T and is denoted by S_i. As usual, for convenience we assume that T ends in a unique character \$ which is not present elsewhere in the text, and that \$ $< a$ for all $a \in \Sigma$.

The *suffix array* $\mathsf{SA}[1, n]$ of T is a permutation of the integers in $[1, n]$ such that $S_{\mathsf{SA}[i-1]} <_{\mathrm{lex}} S_{\mathsf{SA}[i]}$ for all $1 < i \leq n$. In other words, SA describes the lexicographic order of the suffixes. As already mentioned in the introduction, the suffix array can be built in linear time for integer alphabets.

The array LCP of *longest common prefixes* is based on the suffix array. It holds the lengths of the longest common prefixes of lexicographically adjacent suffixes, in symbols: $\mathsf{LCP}[i] = \max\{\ell \geq 0 \mid T_{\mathsf{SA}[i]..\mathsf{SA}[i]+\ell-1} = T_{\mathsf{SA}[i-1]..\mathsf{SA}[i-1]+\ell-1}\}$ for

$1 < i \leq n$, and $\mathsf{LCP}[1] = 0$. A direct "naive" algorithm emerging from this definition runs in $O(n^2)$ time (compare characters until finding a mismatch). Again, we already mentioned that there are also algorithms for constructing LCP in linear time.

2.2   Constructing Suffix Arrays by Induced Sorting

As the basis of our new LCP-array construction algorithm is the *induced sorting* algorithm for constructing suffix arrays [18], we explain this latter algorithm in the following. Induced sorting has a venerable history in suffix sorting, see [9, 14, 22]. Its basic idea is to sort a certain subset of suffixes, either directly or recursively, and then use this result to *induce* the order of the remaining suffixes. In the rest of this section, we follow the presentation of Okanohara and Sadakane [19].

Definition 1. *For $1 \leq i < n$, suffix S_i is said to be* S-type *if $S_i <_{\mathrm{lex}} S_{i+1}$, and* L-type *otherwise. The last suffix is defined to be S-type. For brevity, we also use the terms* S- *and* L-suffixes *for suffixes of the corresponding type.*

The type of each suffix can be determined in linear time by a right-to-left scan of T: first, S_n is declared as S-type. Then, for every i from $n-1$ to 1, S_i is classified by the following rule:

$$S_i \text{ is S-type iff either } t_i < t_{i+1}, \text{ or } t_i = t_{i+1} \text{ and } S_{i+1} \text{ is S-type.}$$

We further say that an S-suffix S_i is of *type S^** iff S_{i-1} is of type L. (Note that the S-suffixes still include the S*-suffixes in what follows.)

In SA, all suffixes starting with the same character $c \in \Sigma$ form a consecutive interval, called the *c-bucket* henceforth. Observe that in any c-bucket, the L-suffixes precede the S-suffixes. Consequently, we can sub-divide buckets into S-type buckets and L-type buckets.

Now the induced sorting algorithm can be explained as follows:

1. Sort the S*-suffixes. This step will be explained in more detail below.
2. Put the sorted S*-suffixes into their corresponding S-buckets, without changing their order.
3. Induce the order of the L-suffixes by scanning SA from left to right: for every position i in SA, if $S_{\mathsf{SA}[i]-1}$ is L-type, write $\mathsf{SA}[i]-1$ to the current head of the L-type c-bucket ($c = t_{\mathsf{SA}[i]-1}$), and increase the current head of that bucket by one. Note that this step can only induce "to the right" (the current head of the c-bucket is larger than i).
4. Induce the order of the S-suffixes by scanning SA from *right to left*: for every position i in SA, if $S_{\mathsf{SA}[i]-1}$ is S-type, write $\mathsf{SA}[i] - 1$ to the current *end* of the S-type c-bucket ($c = t_{\mathsf{SA}[i]-1}$), and *decrease* the current end of that bucket by one. Note that this step can only induce "to the left," and might intermingle S-suffixes with S*-suffixes.

It remains to explain how the S*-suffixes are sorted (step 1 above). To this end, we define:

Definition 2. *An* S*-substring *is a substring $T_{i..j}$ with $i \neq j$ of T such that both S_i and S_j are S*-type, but no suffix in between i and j is also of type S*.*

Let $R_1, R_2, \ldots, R_{n'}$ denote these S*-substrings, and σ' be the number of different S*-substrings. We assign a *name* $v_i \in [1, \sigma']$ to any such R_i, such that $v_i < v_j$ if $R_i <_{\text{lex}} R_j$ and $v_i = v_j$ if $R_i = R_j$. We then construct a new text $T' = v_1 \ldots v_{n'}$ over the alphabet $[1, \sigma']$, and build the suffix array SA' of T' by applying the inducing sorting algorithm *recursively* to T' if $\sigma' < n'$ (otherwise there is nothing to sort, as then the order of the S*-suffixes is given by the order of the S*-substrings). The crucial property [18] to observe here is that the order of the suffixes in T' is the same as the order of the respective S*-suffixes in T; hence, SA' determines the sorting of the S*-suffixes in T. Further, as at most every second suffix in T can be of type S*, the complete algorithm has worst-case running time $T(n) = T(n/2) + O(n) = O(n)$, provided that the *naming* of the S*-substrings also takes linear time, which is what we explain next.

The naming of the S*-substrings is similar to the inducing of the S-suffixes in the induced sorting algorithm (steps 2–4 above), with the difference that in step 2 we put the *unsorted* S*-suffixes into their corresponding buckets (hence they are only sorted according to their first character). Steps 3 and 4 work exactly as described above. At the end of step 4, we can assign names to the S*-substrings by comparing adjacent S*-suffixes naively until we find a mismatch or reach their end; this takes overall linear time.

3   Inducing the LCP-Array

We now explain how the induced sorting algorithm (Sect. 2.2) can be modified to also compute the LCP-array. The basic idea is that whenever we place two S- or L-suffixes S_{i-1} and S_{j-1} at adjacent places $k-1$ and k in the final suffix array (steps 3 and 4 in the algorithm), the length of their longest common prefix can be induced from the longest common prefix of the suffixes S_i and S_j. As the latter suffixes are exactly those that caused the inducing of S_{i-1} and S_{j-1}, we already know their LCP-value ℓ (by the order in which we fill SA), and can hence set $\mathsf{LCP}[k]$ to $\ell + 1$.

3.1   Basic Algorithm

We now describe the algorithm in more detail. We augment the steps of the induced sorting algorithm as follows:

1′. Compute the LCP-values of the S*-suffixes (see Sect. 3.3).
2′. Whenever we place an S*-suffix into its S-bucket, we also store its LCP-value at the corresponding position in LCP.
3′. Suppose that the inducing step just put suffix $S_{\mathsf{SA}[i]-1}$ into its L-type c-bucket at position k. If $S_{\mathsf{SA}[i]-1}$ is the first suffix in its L-bucket, we set $\mathsf{LCP}[k]$ to 0. Otherwise, suppose further that in a previous iteration $i' < i$ the inducing step placed suffix $S_{\mathsf{SA}[i']-1}$ at $k-1$ in the same c-bucket. Then if i' and

i are in different buckets, the suffixes $S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[i']}$ start with different characters, and we set $\mathsf{LCP}[k]$ to 1, as the suffixes $S_{\mathsf{SA}[i]-1}$ and $S_{\mathsf{SA}[i']-1}$ share only a common character c at their beginnings. Otherwise (i' and i are in the same c'-bucket), the length ℓ of the longest common prefix of the suffixes $S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[i']}$ is given by the *minimum* value in $\mathsf{LCP}[i'+1, i]$, all of which are in the same c'-bucket and have therefore already been computed in previous iterations. We can hence set $\mathsf{LCP}[k]$ to $\ell + 1$.

4'. As in the previous step, suppose that the inducing step just put suffix $S_{\mathsf{SA}[i]-1}$ into its S-type c-bucket at position k. Suppose further that in a previous iteration $i' > i$ the inducing step placed suffix $S_{\mathsf{SA}[i']-1}$ at $k+1$ in the same c-bucket (if k is the last position in its S-bucket, we skip the following steps). Then if i' and i are in different buckets, their suffixes start with different characters, and we set $\mathsf{LCP}[k+1]$ to 1, as the suffixes $S_{\mathsf{SA}[i]-1}$ and $S_{\mathsf{SA}[i']-1}$ share only a common character c at their beginnings. Otherwise (i' and i are in the same c'-bucket), the length ℓ of the longest common prefix of the suffixes $S_{\mathsf{SA}[i]}$ and $S_{\mathsf{SA}[i']}$ is given by the *minimum* value in $\mathsf{LCP}[i+1, i']$, all of which are in the same c'-bucket and have therefore already been computed. We can hence set $\mathsf{LCP}[k+1]$ to $\ell + 1$.

(We will resolve the problem of computing the LCP-value between the last L-suffix and the first S-suffix in a bucket at the end of this section.)

3.2   Finding Minima

To find the minimum value in $\mathsf{LCP}[i'+1, i]$ or $\mathsf{LCP}[i+1, i']$ (steps 3' and 4' above), we have several alternatives. The simplest idea is to scan the whole interval from $i'+1$ to i; this results in overall $O(n^2)$ running time. A better alternative would be to keep an array M of size σ, such that the minimum is always given by $M[c]$ if we induce an LCP-value in bucket c. To keep M up-to-date, after each step i we first set $M[c]$ to $\mathsf{LCP}[i]$, and further update all other entries in M that are larger than $\mathsf{LCP}[i]$ by $\mathsf{LCP}[i]$; this approach has $O(n\sigma)$ running time. A further refinement of this technique stores the values in M in sorted order and uses binary search on M to find the minima, similar to the stack used by [7]. This results in overall $O(n \lg \sigma)$ running time.

Yet, we can also update the minima in $O(1)$ amortized running time, as explained next. Let us first focus on the left-to-right scan (step 3'); we will comment on the differences to the right-to-left scan (step 4') at the end of this section. Recall that the queries lie within a single bucket (called c'), and every bucket is subdivided into an L- and an S-bucket. The idea is to also subdivide the query into an L- and an S-query, and return the minimum of the two. The S-queries are simple to handle: in step 3', only S*-suffixes will be scanned, and these are static. Hence, we can preprocess every S-type bucket with a static data structure for constant-time range minima, using overall linear space [4, Thm. 1]. The L-queries are more difficult, as elements keep being written to them during the scan. However, these updates occur in a very regular fashion, namely in a left-to-right manner. This makes the problem simpler: we maintain a *Two-Dimensional*

Min-Heap [4, Def. 2] $\mathcal{M}_{c'}$ for each bucket c', which is initially empty (no L-suffixes written so far). When a new L-suffix along with LCP-value $\ell + 1$ is written into its c'-bucket, we climb up the rightmost path of $\mathcal{M}_{c'}$ until we find an element x whose corresponding array-entry is strictly smaller than $\ell + 1$ ($\mathcal{M}_{c'}$ has an artificial root holding LCP-value $-\infty$ which guarantees that such an element always exists). The new element is then added as x's new rightmost leaf; an easy amortized argument shows that this results in overall linear time. Further, $\mathcal{M}_{c'}$ is stored along with a data structure for constant-time *lowest common ancestor queries* (LCAs) which supports dynamic leaf additions in $O(1)$ worst-case time [3]. Then the minimum in any range in the processed portion of the L-bucket can be found in $O(1)$ time [4, Lemma 2].[1]

In the right-to-left scan (step $4'$), the roles of the L- and S-buckets are reversed: the L-buckets are static and the S-buckets dynamic. For the former, we already have the range minimum data structures from the left-to-right scan (the 2d-Min-Heaps together with LCA). For the S-buckets, we now build an additional 2d-Min-Heap along with dynamic LCAs; this works because the S-buckets are filled in a strict right-to-left manner.

What we have described in the preceding two paragraphs was actually more general than what we really needed: a solution to the *semi-dynamic range minimum query problem* with constant $O(1)$ query- and amortized $O(1)$ insertion-time, with the restriction that new elements can only be appended at the end (or beginning, respectively) of the array. Our solution might also have interesting applications in other problems. In our setting, though, the problem is slightly more specific: the sizes of the arrays to be prepared for RMQs are known in advance (namely the sizes of the L- or S-buckets); hence, we can use any of the (more practical) preprocessing-schemes for (static) RMQs in $O(1)$ worst-case time [1,5], and update the respective structures, which are essentially precomputed RMQs over suitably-sized blocks, whenever enough elements have arrived.

3.3   Computing LCP-Values of S*-Suffixes

This section describes how to compute the LCP-values of the suffixes in the sample set (step $1'$ above). The recursive call to compute the suffix array SA' for the text T' (the text formed by the names of the S*-substrings) also yields the LCP-array LCP' for T'. The problem is that these LCP-values refer to characters v_i in the reduced alphabet $[1, \sigma']$, which correspond to S*-substrings R_i in T. Hence, we need to "scale" every LCP-value in LCP' by the lengths of the actual S*-substrings that constitute this longest common prefix: a value $\mathsf{LCP}'[k]$ refers

---

[1] Note that it is important to use the Two-Dimensional Min-Heap rather than the usual Cartesian Tree for achieving overall linear time, for the following reason: Although the Cartesian Tree also has $O(1)$ amortized update-time for the operation "append at end;" it also needs to relink entire subtrees, rather than only inserting new leaves to the rightmost path [6]. For the relink-operation, no constant-time solutions exist for maintaining $O(1)$-LCAs in the tree (not even in an amortized sense); the best solution we are aware of takes $\alpha(\cdot, n)$ update time [8], $\alpha(\cdot, \cdot)$ being the inverse Ackermann function.

to the substring $v_{\mathsf{SA}'[k]} \cdots v_{\mathsf{SA}'[k]+\mathsf{LCP}'[k]-1}$ of T', and actually implies an LCP-value of $\sum_{i=0}^{\mathsf{LCP}'[k]-1} |R_{\mathsf{SA}'[k]+i}|$ between the corresponding S*-suffixes in T.

A naive implementation of this calculation could again result in $O(n^2)$ running time, consider the text $T = \mathtt{abab\ldots ab}$. However, we can make use of the fact that the suffixes of T' appear lexicographically ordered in T': when "scaling" $\mathsf{LCP}'[k]$, we know that the first $m = \min(\mathsf{LCP}'[k-1], \mathsf{LCP}'[k])$ S*-substrings match, and can hence compute the actual LCP-value as

$$\sum_{i=0}^{\mathsf{LCP}'[k]-1} |R_{\mathsf{SA}'[k]+i}| = \underbrace{\sum_{i=0}^{m-1} |R_{\mathsf{SA}'[k]+i}|}_{\text{already computed}} + \sum_{i=m}^{\mathsf{LCP}'[k]-1} |R_{\mathsf{SA}'[k]+i}| \ .$$

This way, by an amortized argument it is easy to see that each character in T contributes to at most 2 additions, resulting in an overall $O(n)$ running time.

It is possible to stop the recursive LCP-calculation at a specified depth and use any other LCP-array construction algorithm on the remaining (sparse) set of sorted suffixes.

3.4   Computing LCP-Values at the L/S-Seam

There is one subtlety in the above inducing algorithm we have withheld so far, namely that of computing the LCP-values between the last L-suffix and the first S-suffix in a given c-bucket (we call this position the *L/S-seam*). More precisely, when reaching an L/S-seam in step $3'$, we have to re-compute the LCP-value between the first S*-suffix in the c-bucket (if it exists) and the last L-suffix in the same c-bucket (the one that we just induced), in order to induce correct LCP-values when stepping through the S*-suffixes in subsequent iterations. Likewise, when placing the very first S-suffix in its c-bucket in step $4'$, we need to compute the LCP-value between this induced S-suffix and the largest L-suffix in the same c-bucket. (Note that step 4 might place an S-suffix before all S*-suffixes, so we cannot necessarily re-use the LCP-value computed at the L/S-seam in step $3'$.)

The following lemma shows that the LCP-computation at L/S-seams is particularly easy:

Lemma 1. *Let S_i be an L-suffix, S_j an S-suffix, and $t_i = c = t_j$ (the suffixes are in the same c-bucket in SA). Further, let $\ell \geq 1$ denote the length of the longest common prefix of S_i and S_j. Then*

$$T_{i\ldots i+\ell-1} = c^\ell = T_{j\ldots j+\ell-1} \ .$$

Proof. Assume that $t_{i+k} = c' = t_{i+k}$ for some $2 \leq k < \ell$ and $c' \neq c$. Then if $c' < c$, both S_i and S_j are of type L, and otherwise ($c' > c$), they are both of type S. In any case, this is a contradiction to the assumption that S_i is of type L, and S_j of type S.

In words, the above lemma states that the longest common prefix at the L/S-seam can only consist of equal characters. Therefore, a *naive* computation of the

LCP-values at the L/S-seam is sufficient to achieve overall linear running time: every character t_i contributes at most to the computation at the L/S-seam in the t_i-bucket, and not in any other c-bucket for $c \neq t_i$.

4   Experimental Results

We implemented a prototype of the algorithm from the previous section in C and ran several tests on an AMD Dual Core Opteron 270, running at 2.0 GHz with a 1MB L2-cache per core and 4GB of main memory.[2] The basis of our implementation was Yuta Mori's linear-time C-implementation of the induced-sorting algorithm [18], called sais-lite version 2.4.1 (`http://sites.google.com/site/yuta256/sais`). We made the following implementation decisions: instead of calculating the LCP-values of the S*-suffixes recursively, we used a sparse variant of the Φ-algorithm [10] immediately on the first level, which calculates the LCP-values of the S*-suffixes in overall linear time. For the inducing step, we used the $O(n\sigma)$-variant described by Gog and Ohlebusch [7, Sect. 4] (we constrained the stack size to about 10kB as in their implementations). The resulting algorithm is called inducing henceforth. Its space consumption (apart from negligibly small arrays) is no more than that of sais-lite plus the space for the final LCP-array, since the arrays needed by the sparse Φ-algorithm can be first stored within the LCP-array and subsequently be overwritten with actual LCP-values.

We compared our implementation to the following LCP-array construction algorithms:

KLAAP: the original linear-time method for constructing LCP [12], implemented in a space-saving variant [16].
Φ: the Φ-algorithm of Kärkkäinen et al. [10], which is a clever variant of KLAAP that avoids cache-misses by reorganizing the computations.
GO: the hybrid algorithm as described by [7]. It needs the Burrows-Wheeler-Transformation (BWT) for LCP-array construction, and computes small LCP-values naively, from which the larger LCP-values are deduced.
GO2: a semi-external variant of GO [7].
naive: for a sanity check, we also included the *naive* $O(n^2)$-computation of the LCP-array (step through the suffix array and compare the corresponding suffixes character by character).

We used the implementations from the *succinct data structures library* (sdsl 0.9.0) [7] wherever possible. All programs were compiled using the same compiler options (-ffast-math -O9 -funroll-loops -DNDEBUG).

We chose the test suite from `http://pizzachili.dcc.uchile.cl/` for evaluation, which is by now a de-facto standard. It includes texts from natural languages (English), biology (dna and proteins), and structured documents (dblp.xml and sources). Because the authors of [7] point out that the human

---

[2] The source code is available at `http://algo2.iti.kit.edu/1829.php`

Table 1. Running times (user+system in seconds) for LCP- and suffix-array construction. The first block of columns shows the running times for pure LCP-array construction (for KLAAP and Φ, these times include construction of the inverse suffix- and the Φ-array, respectively). The second block shows the construction times of those arrays that need to be constructed before LCP: SA (always) and BWT (for GO and GO2). The third block shows the overall running times for computing both SA and LCP for the best possible combinations of algorithms.

	pure LCP-array construction						SA		BWT	SA+LCP		
	KLAAP [12]	Φ [10]	GO [7]	GO2 [7]	naive	inducing(*) [this paper]	divsufsort	sais-lite		GO+BWT+divsufsort	naive+divsufsort	inducing+sais-lite
20MB dna	7.4	6.4	4.2	6.3	3.3	3.7	4.9	6.8	2.7	11.8	**8.2**	10.5
english	6.4	5.7	10.1	12.4	135.0	4.0	4.9	6.5	2.7	17.7	139.9	**10.5**
dblp.xml	5.6	5.2	4.3	6.1	3.9	3.7	4.0	5.5	2.6	10.9	**7.9**	9.2
sources	5.3	5.0	4.6	6.8	4.1	3.5	3.5	5.4	2.2	10.3	**7.6**	8.9
proteins	6.3	5.7	7.8	9.9	11.9	3.8	5.1	7.3	2.5	15.4	17.0	**11.1**
hs (33MB)	12.6	10.9	6.9	10.2	4.8	6.2	8.2	10.9	4.3	19.4	**13.0**	17.1
50MB dna	21.2	18.2	10.9	15.8	9.0	10.2	14.2	18.0	7.0	32.1	**23.2**	28.2
english	15.1	16.3	21.5	27.2	198.4	11.0	13.5	18.0	6.9	41.9	211.9	**29.0**
dblp.xml	15.1	14.2	10.9	15.5	10.3	9.8	11.1	14.3	6.4	28.4	**21.4**	24.1
sources	14.7	13.7	14.8	20.2	18.5	9.3	9.7	14.4	5.8	30.3	28.2	**23.7**
proteins	19.7	17.3	15.7	21.1	19.5	11.0	15.8	22.3	6.8	38.3	35.3	**33.3**
100MB dna	47.8	41.3	22.3	32.6	19.1	22.7	32.3	38.9	15.1	69.7	**51.4**	61.6
english	41.6	36.5	39.0	49.1	556.3	25.0	29.8	39.7	14.7	83.5	586.1	**64.7**
dblp.xml	31.8	30.0	22.0	31.5	21.9	20.9	24.2	29.6	13.3	59.5	**46.1**	50.5
sources	30.2	28.7	28.3	38.5	111.2	19.4	20.9	30.2	12.3	61.5	132.1	**49.6**
proteins	43.4	36.7	36.0	47.0	51.7	24.3	35.4	48.5	14.6	86.0	87.1	**72.8**
200MB dna	104.6	92.7	46.2	66.9	55.2	51.3	76.3	87.0	33.2	155.7	**131.5**	138.3
english	91.7	81.5	83.0	103.7	3272.4	56.5	68.8	88.8	31.8	183.6	3341.2	**145.3**
dblp.xml	69.7	64.9	44.6	64.0	46.8	50.4	53.3	63.6	27.9	125.8	**100.1**	114.0
sources	66.8	62.4	58.7	79.7	142.0	32.4	46.4	65.2	26.3	131.4	188.4	**97.6**
proteins	92.8	83.4	82.5	104.0	124.4	36.8	76.4	103.6	31.5	190.4	200.8	**140.4**

(*) As inducing is inherently coupled with SA-construction (sais-lite in our implementation), the running times for pure LCP-array construction were calculated by taking the difference of "inducing+sais-lite" and "sais-lite."

chromosome 22 from Manzini's corpus (hs) is a particular hard case for some algorithms, it was also included.

The results are shown in Tbl. 1. The first block of columns shows the running times for pure LCP-array construction. For KLAAP and Φ, these times include construction of the inverse suffix- and the Φ-array, respectively, as they are needed for LCP-array computation. For GO and GO2, the times for computing the BWT are *not* included; the reason is that in some cases the BWT is also needed for other purposes, so it might already be in memory. As inducing is inherently coupled with SA-construction [18], we could not measure its running times for pure LCP-array construction directly; the figures in column "inducing" of Tbl. 1 are hence obtained by first running the *pure* SA-construction (sais-lite), then the combined LCP- and SA-construction, and finally taking the difference of both running times. Measured this way, inducing takes almost always less time than all other methods.

A fairer comparison of the algorithms is shown in the last three columns of Tbl. 1, where the combined running times for SA- and LCP-array construction are given (for a selection of the best-performing LCP-algorithms). This is because all other methods for LCP-array construction are independent of the method for constructing SA, and can hence be combined with faster SA-construction algorithms. It is by now widely agreed that Yuta Mori's divsufsort in version 2.0.1 is the fastest known such algorithm (`http://code.google.com/p/libdivsufsort/`). Hence, for methods GO and naive we give the overall running times combined with divsufsort, whereas for inducing we give the overall running time of sais-lite, adapted to induce LCP-values as well. Further, for GO we also add the times to compute the BWT, as it is needed for LCP-array construction.

Inspecting the results from Tbl. 1, we see that inducing+sais-lite is usually the best possible combination, sometimes outperformed by naive+divsufsort. In fact, the naive algorithm is rather competitive (especially for small inputs up to 50MB), apart from the English text, which consists of long repetitions of the same texts (and hence has large average LCP).

5 Conclusions and Outlook

We showed how the LCP-array can be induced along with the suffix array. A rather ad-hoc implementation outperformed all state-of-the-art algorithms. We point out the following potentials for practical improvements: (1) As suffix- and LCP-values are always written to the same place, an interleaved storage of SA and LCP could result in fewer cache misses. (2) As the faster divsufsort is also based on induced sorting, incorporating our ideas into that algorithm could result in better overall performance. (3) Computing the LCP-values of the S*-suffixes recursively up to a certain (well-chosen) depth could be faster than just using the Φ-algorithm on level 0, as in our implementation.

Acknowledgments

We thank Moritz Kobitzsch for help on programming, and Peter Sanders for interesting discussions.

References

1. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: A survey and a new algorithm for a distributed environment. Theory Comput. Syst. 37, 441–456 (2004)
2. Antonitio, Ryan, P.J., Smyth, W.F., Turpin, A., Yu, X.: New suffix array algorithms — linear but not fast? In *Proc. Fifteenth Australasian Workshop Combinatorial Algorithms (AWOCA)*, pages 148–156, 2004.
3. Cole, R., Hariharan, R.: Dynamic LCA queries on trees. SIAM J. Comput. 34(4), 894–923 (2005)
4. Fischer, J.: Optimal succinctness for range minimum queries. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 158–169. Springer, Heidelberg (2010)
5. Fischer, J., Heun, V.: A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Chen, B., Paterson, M., Zhang, G. (eds.) ESCAPE 2007. LNCS, vol. 4614, pp. 459–470. Springer, Heidelberg (2007)
6. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. STOC, pp. 135–143. ACM Press, New York (1984)
7. Gog, S., Ohlebusch, E.: Fast and lightweight LCP-array construction algorithms. In: Proc. ALENEX, pp. 25–34. SIAM Press, Philadelphia (2011)
8. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984); See also FOCS 1980
9. Itoh, H., Tanaka, H.: An efficient method for in memory construction of suffix arrays. In: Proc. SPIRE/CRIWG, pp. 81–88. IEEE Press, Los Alamitos (1999)
10. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009 Lille. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 1–19 (2006)
12. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
13. Kim, D.K., Sim, J.S., Park, H., Park, K.: Constructing suffix arrays in linear time. J. Discrete Algorithms 3(2-4), 126–142 (2005)
14. Ko, P., Aluru, S.: Space efficient linear time construction of suffix arrays. J. Discrete Algorithms 3(2-4), 143–156 (2005)
15. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
16. Manzini, G.: Two space saving tricks for linear time LCP array computation. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 372–383. Springer, Heidelberg (2004)
17. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), Article No. 2 (2007)

18. Nong, G., Zhang, S., Chan, W.H.: Linear suffix array construction by almost pure induced-sorting. In: Proc. DCC, pp. 193–202. IEEE Press, Los Alamitos (2009)
19. Okanohara, D., Sadakane, K.: A linear-time burrows-wheeler transform using induced sorting. In: Karlgren, J., Tarhio, J., Hyyrö, H. (eds.) SPIRE 2009. LNCS, vol. 5721, pp. 90–101. Springer, Heidelberg (2009)
20. Puglisi, S.J., Smyth, W.F., Turpin, A.: A taxonomy of suffix array construction algorithms. ACM Computing Surveys 39(2) (2007)
21. Sadakane, K.: Compressed suffix trees with full functionality. Theory of Computing Systems 41(4), 589–607 (2007)
22. Seward, J.: On the performance of BWT sorting algorithms. In: Proc. DCC, pp. 173–182. IEEE Press, Los Alamitos (2000)
23. Weiner, P.: Linear pattern matching algorithms. In: Proc. Annual Symp. on Switching and Automata Theory, pp. 1–11. IEEE Computer Society, Los Alamitos (1973)

Horoball Hulls and Extents in Positive Definite Space[⋆]

P. Thomas Fletcher, John Moeller, Jeff M. Phillips,
and Suresh Venkatasubramanian

University of Utah

Abstract. The space of positive definite matrices P(n) is a Riemannian mani-
fold with variable nonpositive curvature. It includes Euclidean space and hyper-
bolic space as submanifolds, and poses significant challenges for the design of
algorithms for data analysis. In this paper, we develop foundational geometric
structures and algorithms for analyzing collections of such matrices. A key tech-
nical contribution of this work is the use of *horoballs*, a natural generalization of
halfspaces for non-positively curved Riemannian manifolds. We propose gener-
alizations of the notion of a convex hull and a centerpoint and approximations of
these structures using horoballs and based on novel decompositions of P(n). This
leads to an algorithm for approximate hulls using a generalization of extents.

1 Introduction

Data analysis and Euclidean geometry have traditionally been strongly linked, by rep-
resenting data as points in a Euclidean space and comparing data using the Euclidean
distance. However, as models for data analysis grow more sophisticated, it is becoming
clearer that accurate modeling of data requires the use of non-Euclidean geometry and
the induced geodesic distances. This geometry might be as simple as a surface embed-
ded in a Euclidean space but in general may be represented as a Riemannian manifold
with variable curvature [4].

One such manifold is P(n), the manifold of real symmetric positive definite matrices.
There are many application areas where the basic objects of interest, rather than points
in Euclidean space, are elements of P(n). In diffusion tensor imaging [3], matrices in
P(3) model the flow of water at each voxel of a brain scan, and a goal is to *cluster* these
matrices into groups that capture common flow patterns along fiber tracts. In mechanical
engineering [11], stress tensors are modeled as elements of P(6), and identifying groups
of similar tensors helps locate homogeneous regions in a material from samples. Kernel
matrices in machine learning are elements of P(n) [26], and motivated by the problems
of learning and approximating kernels for machine learning tasks, there has been recent
interest in studying the geometry of P(n) and related spaces [20,7,28].

In all these areas, a problem of great interest is the analysis [15,16] of collections
of such matrices (finding central points, clustering, doing regression). It is important to
note that in all these examples, the *Riemannian* structure of P(n) is a crucial element
of the modelling process: merely treating the matrices as points in $\mathbb{R}^{n^2}$ and endowing

the space with the Euclidean distance[1] does not capture the correct notion of distance or closeness that is meaningful in these applications [22,21,27,25,23]. For example, if we want to interpolate between two matrices of similar volume (determinant), a line between the two in $\mathbb{R}^{n^2}$ contains matrices of volume well outside the range of volumes of the two matrices, but all matrices on a Riemannian geodesic have volume in the range.

Performing data analysis in such spaces requires the standard geometric toolkit that has proven successful for Euclidean data: methods for summarizing data sets (extents and core sets), finding representatives (centerpoints), and even performing accurate sampling (VC dimension estimates and ε-samples). In order to compute these objects, we need appropriate generalizations of the equivalent concepts in Euclidean spaces.

Our Contributions. In this paper we initiate a study of geometric algorithms on $P(n)$. We develop appropriate generalizations of halfspaces and convex hulls and also prove bounds on the VC-dimension of associated range spaces. We apply these results to the problem of estimating approximate extents for collections of matrices in $P(n)$, as well as studying approximate center points for such collections. Our results indicate that the *horoball* (a generalization of a halfspace) retains many, but not all, of the same combinatorial and structural properties of halfspaces, and is therefore a crucial building block for designing algorithms in this space.

1.1 Hulls and Convexity: From $\mathbb{R}^d$ to $P(n)$

$P(n)$ is a good model space for algorithmic analysis. Like other Cartan-Hadamard (C-H) manifolds [8], its metric balls under the geodesic distance are (geodesically) convex, which is an important property[2]. $P(n)$ has been extensively studied analytically, and there are simple closed form expressions for geodesics, the geodesic distance and other important constructs (see Section 2 for details). This is in contrast to other negatively-curved spaces like the δ-hyperbolic spaces [17], where in general one must assume some oracle to compute distances between points.

However, the variable curvature of $P(n)$ poses significant challenges for the construction of standard geometric primitives. A natural notion of a halfspace that is both "flat" and "convex" seems elusive, and it is not even clear whether a finite description of the geodesic convex hull of 3 points exists! To understand why this is the case, and why generalizing convex hulls to $P(n)$ is difficult, it is helpful to first understand the key properties used to define these notions in Euclidean space. In general, we desire a compact representation of the convex hull of a set of points. Call this property (C):

(C): Given a finite set of points X, the convex hull $\mathcal{C}(X)$ of X has a finite description using simple convex objects.

Euclidean hulls. In Euclidean space, the "simple convex objects" are halfspaces, which satisfy two properties:

---

[1] In other words, computing the Frobenius distance between matrices.

[2] Compare this with even simple positively curved manifolds like the n-dimensional sphere, in which this is no longer true.

(P1) The complement of a closed halfspace is an open halfspace; both are convex. Their boundary, a hyperplane, is also convex.

(P2) Generically, d hyperplanes intersect at a single point and n hyperplanes partition $\mathbb{R}^d$ into $\Theta(n^d)$ regions.

These two properties can be used to construct the convex hull (satisfying property (C)) efficiently in two different ways. *Painting Segments:* Given a finite set $X_0 \subset \mathbb{R}^d$ we can paint segments between all $x_1, x_2 \in X_0$; the union of these segments yields a set X_1. We can recursively apply this procedure d times to generate $\langle X_2, \ldots, X_d \rangle$, where X_d is the convex hull of X_0 [5]. This is the convex combination of X_0, and the boundary is partitioned into a finite number of faces uniquely determined by painting segments among d-point subsets of X_0.

Intersection of Convex Families: The convex hull of $X \subset \mathbb{R}^d$ is the intersection of all halfspaces which contain X; we only need to consider the finite set of halfspaces supported by d points. We can also define the convex hull of X as the intersection of all balls which contain X. Again we only need to consider balls supported by d points; fixing this incidence, let their radius grow to infinity so they become halfspaces in the limit; see Figure 1(a).

Hyperbolic hulls. The first space we encounter as we move beyond Euclidean space towards P(n) is the hyperbolic space $\mathbb{H}^d$, which is a Riemannian manifold of constant negative curvature. It is convenient to embed $\mathbb{H}^d$ in the unit ball of $\mathbb{R}^d$ using well-known models, specifically the Klein model [8, I.6] of $\mathbb{H}^d$ in which geodesics are straight lines, and the Poincaré model [8, I.6] of $\mathbb{H}^d$ in which metric balls are Euclidean balls and geodesics are circular arcs normal to the boundary of the unit ball.

Since in the Klein model, geodesics are straight lines, the *painting segments* construction yields a convex hull in the same way as in Euclidean space; see Figure 1(b). Similarly, a *hyperbolic halfspace* can be written as the intersection of the unit ball with a Euclidean halfspace, and so a finite description of the convex hull can be obtained via the intersection of halfspaces; see Figure 1(c).

Hulls in P(n). Once we reach P(n), these concepts break down. There is no way, in general, to construct a halfspace (convex, and whose complement is convex) supported by d points; such an object is called a *totally geodesic submanifold* and might not pass
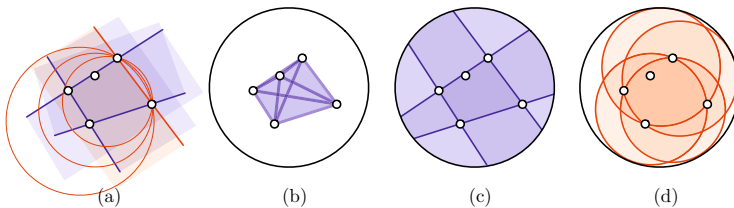

Fig. 1. Illustration of different constructions of hulls in $\mathbb{R}^2$ (a) and $\mathbb{H}^2$ under the Klein model (b,c) and Poincaré model (d). (a) Intersection of halfspaces, one halfspace as limit of ball supported by 2 points. (b) Painting segments, X_0 as circles, X_1 as segments, and X_2 shaded. (c) Intersection of hyperbolic-halfspaces, forming the convex hull. (d) Intersection of horoballs, forming the ball hull.

through any given set of d points. This rules out constructions via the intersection of convex families. Constructing a hull via painting of segments also does not work; the resulting process may not terminate in a finite number of steps, and the resulting object might be full-dimensional, and would not in general lend itself to a finite description.

There is however another way to approach the idea of halfspaces. Returning to $\mathbb{H}^d$ and the Poincaré model, consider a ball fixed at a point whose radius is allowed to grow to infinity. Such a ball is called a *horoball*, and is convex. In Euclidean space, this construction yields a halfspace passing through the fixed point, but in a curved space, the ball never completely "flattens" out in the sense of property (P1). Horoballs can be described finitely by a point and a tangent vector (in the same way a Euclidean halfspace can be described by a point and a normal). We can then describe the intersection of all horoballs containing X, which we call the *ball hull*; it is convex and contains the convex hull of X; see Figure 1(d). In the rest of this paper, we will focus our attention on horoballs and the ball hull.

1.2   Technical Overview

A key conceptual insight in this work is that the horoball acts functionally like a halfspace, and can be used as a replacement for halfspace in spaces (like $P(n)$) which do not in general admit halfspaces that span arbitrary sets of points. As justification for this insight, our main result is an algorithm for computing an approximate ball hull of a set of points in $P(n)$, where the approximation uses a generalized notion of *extent* defined analogously to how (hyperplane) extent is defined in $\mathbb{R}^d$. The construction itself follows the rough outline of approximate extent constructions in $\mathbb{R}^d$. In fact, we exploit the fact that $P(n)$ admits a decomposition into a collection of Euclidean subspaces, each "indexed" by an element of $SO(n)$, the group of $n \times n$ rotation matrices, and use a grid construction to cover $SO(n)$ with a net, followed by building convex hulls in each of the (finitely many) Euclidean subspaces induced by the net and combining them. We expect that this decomposition will be of independent interest.

We announce two other uses of horoballs, with details in the full version. We can define range spaces of horoballs, and we analyze shatter dimension and VC-dimension of $P(2)$. We also use these results to study center points in $P(n)$.

1.3   Related Work

The mathematics of Riemannian manifolds, Cartan-Hadamard manifolds, and $P(n)$ is well-understood; the book by Bridson and Haefliger [8] is an invaluable reference on metric spaces of nonpositive curvature, and Bhatia [5] provides a detailed study of $P(n)$ in particular. However, there are very few *algorithmic* results for problems in these spaces. To the best of our knowledge, the only prior work on algorithms for positive definite space are the work by Moakher [22] on mean shapes in positive definite space, the work by Fletcher and Joshi [15] on principal geodesic analysis in symmetric spaces, the robust median algorithms of Fletcher *et al* [16] for general manifolds (including $P(n)$ and $SO(n)$), and the generic approximation technique of Arnaudon and Nielsen [2] for the Riemannian 1-center.

Geometric algorithms in hyperbolic space are much more tractable. The Poincaré and Klein models of hyperbolic space preserve different properties of Euclidean space, and many algorithm carry over directly with no modifications. Leibon and Letscher [19] were the first to study basic geometric primitives in general Riemannian manifolds, constructing Voronoi diagrams and Delaunay triangulations for sufficiently dense point sets in these spaces. Their work was subsequently improved by Dyer *et al.* [13]. Eppstein [14] described hierarchical clustering algorithms in hyperbolic space.

δ-hyperbolic spaces [17] (metric spaces that "look" negatively curved without necessarily having a smooth notion of curvature) have also been studied. Krauthgamer and Lee [18] studied the nearest neighbor problem for points in δ-hyperbolic space. Chepoi *et al* [9,10] advanced this line of research, providing algorithms for computing the diameter and minimum enclosing ball of collections of points in δ-hyperbolic space. Work by Billera *et al.* [6] showed how to model the space of phylogenetic trees as a specific CAT(0) space [8, II.1]; work by Owen and Provan investigated how to efficiently compute geodesics in such a space [24].

2   Preliminaries

In this section we present the basic algebra and geometry needed to understand our technical results without additional outside references. $P(n)$ is the manifold consisting of symmetric positive-definite real matrices. As a manifold, it has a Euclidean tangent space $T_p P(n)$ at each point p, represented by the space of symmetric matrices $S(n)$. A point in the tangent space represents a vector tangent to a curve that passes through the point p. The velocity of a particle moving along such a curve, for example, would be represented by a vector in $T_p P(n)$ when it passes through p.

A *geodesic* through a point p is a special curve determined entirely by a tangent vector $A \in T_p P(n)$. This relationship between the tangent space and the manifold is realized by the exp map $\exp_p : S(n) \to P(n)$, defined as $\exp_p(A) = p e^{p^{-1}A}$, where e^X is just the matrix exponential. If $c(t)$ is a geodesic with tangent A at $c(0) = p$, then $c(t) = \exp_p(tA)$. For simplicity, we often assume that $p = I$ so $\exp_I(A) = e^A$. The exp map has a simple intuition: in Euclidean space, $\exp_p(\mathbf{u}) = p + \mathbf{u}$; that is, we just move from the point p to another point that is in the direction of $\mathbf{u}$, $\|\mathbf{u}\|$ units away.

We can then measure distance between two points on the manifold by finding a geodesic between them, solving for the unknown tangent vector, and measuring its length. The exp map is invertible, and its inverse is the log map, $\log_p : P(n) \to S(n)$, given by $\log_p(q) = p \log(p^{-1}q)$, where log is the inverse of the matrix exponential. $P(n)$ is also endowed with the special property that $\exp_p$ is invertible across the entire manifold, letting us measure distance between any two points. Because $P(n)$ is a *Riemannian* manifold, $T_p P(n)$ has an inner product $\langle A, B \rangle_p = \mathrm{tr}(p^{-1}A p^{-1}B)$, and $\|A\|_p = \sqrt{\langle A, A \rangle_p}$. The resulting metric is then $D(p,q) = \|\log_p(q)\| = \sqrt{\mathrm{tr}(\log(p^{-1}q)^2)}$. Usually we assume that $\|A\|_p = 1$, meaning that a geodesic's parameter t equals distance traveled.

As mentioned before, $P(n)$ is a Riemannian manifold of non-positive curvature. Its exp map is *surjective*, which enables us to talk about geodesics between any two points on the manifold. Therefore we need not concern ourselves with a radius of convexity; a convex subset of $P(n)$ need not be bounded. Another important property of $P(n)$ is that

it is *symmetric*. This means that there is always an isometry that moves a point p to a point q without altering the metric properties of the manifold, offering the equivalent of translation invariance. A good reference for the reader is [8, II.10].

Structure of P(2). Worth mentioning is the specific 3-dimensional manifold $P(2)$, which has a structure that we exploit frequently. $P(2)$ is isometric to $\mathbb{R} \times P(2)_1$, where $P(2)_1$ is the same as $\mathbb{H}^2$ with a factor of $1/\sqrt{2}$ on the metric. $P(2)_1$ is so named because it represents the submanifold of $P(2)$ containing all p.d. matrices of determinant 1. Intuitively, we can think of this as a "stack" of hyperbolic spaces, leading us further to a cylindrical representation of the space (see Figure 2). Decomposing a matrix p as $(r, p') = e^{r/2} \cdot p'$, we can represent the (log) determinant of p as r. Using a polar representation, we can break p' down further and realize the *anisotropy*, or ratio of eigenvalues, as a radial coordinate. Since the eigenvectors form a rotation matrix, we take the angle of this rotation (times 2) as the remaining coordinate.

2.1   Busemann Functions and Horoballs

In $\mathbb{R}^d$, the convex hull of a finite set can be described by a finite number of hyperplanes each supported by d points from the set. A hyperplane through a point may also be thought of as the limiting case of a sphere whose center has been moved away to infinity while a point at its surface remains fixed. This notion of "pulling away to infinity" can be formalized: given a geodesic ray $c(t) : \mathbb{R}^+ \to M$ on a Cartan-Hadamard manifold M, a *Busemann function* $b_c : M \to \mathbb{R}$ is defined $b_c : p \mapsto \lim_{t \to \infty} D(p, c(t)) - t$. An important property of b_c is that it is convex [8, II.8].

As an example, we can easily compute the Busemann function in $\mathbb{R}^n$ for a ray $c(t) = t\mathbf{u}$, where $\mathbf{u}$ is a unit vector. Since $\lim_{t \to \infty} \frac{1}{2t}(\|p - t\mathbf{u}\| + t) = 1$,

$$b_c(p) = \lim_{t \to \infty} \frac{1}{2t}\left(\|p - t\mathbf{u}\|^2 - t^2\right) = -\langle p, \mathbf{u} \rangle.$$

A *horoball* $B_r(b_c) \subset M$ is a sublevel set of b_c; that is, $B_r(b_c)$ is the set of all $p \in M$ such that $b_c(p) \le r$ (recall Figure 1). Since b_c is convex, any sublevel set of it is convex, and hence any horoball is convex. Continuing with the example of Euclidean space, horoballs are simply halfspaces: all $p \in \mathbb{R}^d$ such that $-\langle p, \mathbf{u} \rangle \le r$.

3   Ball Hulls

We now introduce our variant of the convex hull in $P(n)$, which we call the ball hull. For a subset $X \subset P(n)$, the *ball hull* $\mathcal{B}(X)$ is the intersection of all horoballs that also contain X:

$$\mathcal{B}(X) = \bigcap_{b_c, r} B_r(b_c), \quad X \subset B_r(b_c).$$

Properties of the ball hull. Recall that the ball hull can be seen as an alternate generalization of the Euclidean convex hull (i.e. via intersection of halfspaces) to $P(n)$. Furthermore, since it is the intersection of closed convex sets, it is itself guaranteed to be closed and convex (and therefore $\mathcal{C}(X) \subseteq \mathcal{B}(X)$). We can also show that it shares critical parts of its boundary with the convex hull (Theorem 1), but unfortunately, we cannot

represent it as a finite intersection of horoballs (Theorem 2). We state these theorems here, and defer proofs to the full version.

Theorem 1. *Every $x \in X$ (X finite) on the boundary of $\mathcal{B}(X)$ is also on the boundary of $\mathcal{C}(X)$ (i.e., $X \cap \partial\mathcal{B}(X) \subseteq X \cap \partial\mathcal{C}(X)$).*

Theorem 2. *In general, the ball hull cannot be described as the intersection of a finite set of horoballs.*

3.1 The ε-Ball Hull

Theorem 2 indicates that we cannot maintain a finite representation of a ball hull. However, as we show in this section, we can maintain a finite-sized *approximation* to the ball hull. Our approximation will be in terms of *extents*; intuitively, a set of horoballs approximates the ball hull if a geodesic traveling in any direction traverses approximately the same distance inside the ball hull as it does inside the approximate hull.

In Euclidean space, we can capture extent by measuring the distance between two parallel hyperplanes that sandwich the set. The analogue to this construction in $P(n)$ is the *horoextent*, the distance along a geodesic between two opposing horoballs. Let $c(t) = qe^{tq^{-1}A}$ be a geodesic, and $X \subset P(n)$. The *horoextent* $E_c(X)$ with respect to c is defined as:
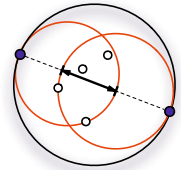
horoextent

$$E_c(X) = \left| \max_{p \in X} b_{c+}(p) + \max_{p \in X} b_{c-}(p) \right|,$$

where b_{c+} is the Busemann function created when we follow $c^+(t) = c(t)$ to infinity as normal, while b_{c-} is the Busemann function created when we follow the geodesic pointing in the opposite direction, $c^-(t) = qe^{tq^{-1}(-A)} = c(-t)$. Stated differently: $b_{c+}(p) = \lim_{t \to +\infty}(D(c(t),p) - t)$, and $b_{c-}(p) = \lim_{t \to -\infty}(D(c(t),p) + t)$. Observe that for any c, $E_c(X) = E_c(\mathcal{C}(X)) = E_c(\mathcal{B}(X))$.

In Euclidean space, the distance between parallel planes is a constant. In general, because of the effects of curvature, the distance between horoballs depends on the geodesic used. For instance in $P(n)$, horofunctions are nonlinear, so the distance between opposing horoballs is not constant. The width of the intersection of the opposing horoballs is taken along the geodesic c, and a geodesic is described by a point q and a direction A. We fix the point q so that we need only choose a uniform grid of directions A for our approximation.

An intersection of horoballs is called an ε-ball hull with origin q ($\mathcal{B}_{\varepsilon,q}(X)$) if for all geodesic rays c such that $c(0) = q$, $|E_c(\mathcal{B}_{\varepsilon,q}(X)) - E_c(X)| \leq \varepsilon$. For convenience, we assume that $I \in \mathcal{C}(X)$ (this assumption will be removed in the full version), and our origin $q = I$. Then we will refer to $\mathcal{B}_{\varepsilon,I}(X)$ as just an ε-ball hull $\mathcal{B}_\varepsilon(X)$.

We will use $D_X \leq \mathrm{diam}(X) = \max_{p,q \in X} D(p,q)$ in our bounds; see the full version for more precise definition, and note $\mathrm{diam}(X)$ is an intrinsic parameter of the data.

4   Constructing the ε-Ball Hull

Main result. In this section we construct a finite-sized ε-ball hull.

Theorem 3. *Let $\Gamma_n(\varepsilon, D_X) = (\sinh(D_X)/\varepsilon)^{n-1}$. For a set $X \subset P(n)$ of size N (for constant n), we can construct an ε-ball hull of size $O(\Gamma_n(\varepsilon, D_X) \cdot (D_X/\varepsilon)^{(n-1)/2})$ in time $O(\Gamma_n(\varepsilon, D_X) \cdot ((D_X/\varepsilon)^{n-3/2} + N))$. Furthermore, we can construct a coreset $Y \subset X$ of size $O(\Gamma_n(\varepsilon, D_X) \cdot (D_X/\varepsilon)^{(n-1)/2})$ whose $(\varepsilon/2)$-ball hull is an ε-ball hull of X.*

Proof overview. We make extensive use of a structural decomposition in our proof. But first it will be helpful to define a *flat*. Let us define a *subspace* of a manifold as the result of applying $\exp_p$ to each point of a subspace of the tangent space $T_pP(n)$. If the resulting submanifold is isometric to a Euclidean space, then it is called a *flat*. A flat has the important property of being convex (in general a subspace is not). One canonical example of a flat is the subspace of positive sorted diagonal matrices.

$P(n)$ can then be realized as the union of a set of n-dimensional flats, and the space can be parameterized by a rotation matrix $Q \in SO(n)$, one for each flat F. The intersection of these flats is the line of multiples of I. In $P(2)$, we can picture these flats as panes in a revolving door; see Figure 2(a).

We construct ε-ball hulls by discretizing $P(n)$ in two steps. First, we show that within a flat F (i.e., given a rotation $Q \in SO(n)$) we can find a finite set of minimal horoballs exactly, or we can use ε-kernel machinery [1] to approximate this structure. This is done by showing an equivalence between halfspaces in F and horoballs in $P(n)$ in Section 4.1. This result implies that computing all minimal horoballs with respect to a rotation Q is equivalent to computing a convex hull in Euclidean space.

Second, we show that instead of searching over the entire space of rotations $SO(n)$, we can discretize it into a finite set of rotations such that when we calculate the horoballs with respect to each of these rotations, the horoextents of the resulting ε-ball hull are not too far from those of the ball hull. In order to do this, we prove a Lipschitz bound for horofunctions (and hence horoextents) on the space of rotations.

Proving this theorem is quite technical. We first prove a Lipschitz bound in $P(2)$, where the space of rotations is a circle (as in Figure 2(b)). After providing a bound in $P(2)$ we decompose the distance between two rotations in $SO(n)$ into $\lfloor n/2 \rfloor$ angles defined by 2×2 submatrices in an $n \times n$ matrix. In this setting it is possible to apply the $P(2)$ Lipschitz bound $\lfloor n/2 \rfloor$ times to get the full bound. We present the proof for $P(2)$ in Section 4.2, and the generalization to $P(n)$ in Section 4.3. Finally, we combine these results in an algorithm in Section 4.4.

4.1 Decomposing P(n) into Flats

A critical operation associated with the decomposition (illustrated in Figure 2) is the *horospherical projection function* $\pi_F : P(n) \to F$ that maps a point $p \in P(n)$ to a point $\pi_F(p)$ in a n-dimensional flat F. For each Busemann function b_c there exists a flat F for which it is invariant under the associated projection π_F; that is, $b_c(p) = b_c(\pi_F(p))$ for all $p \in P(n)$. Using π_F for associated geodesic $c(t) = e^{tA}$ where $A \in S(n)$, the Busemann function $b_c : P(n) \to \mathbb{R}$ can be written [8, II.10]

$$b_c(p) = -\operatorname{tr}(A \log(\pi_F(p))).$$

It is irrelevant which point is chosen for the origin of the geodesic ray, so we usually assume that it is chosen in such a way that $b_c(I) = 0$ in $P(n)$.
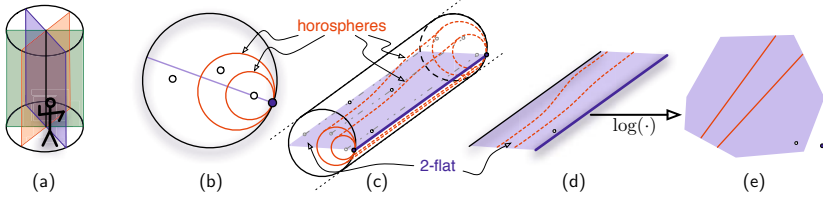
Fig. 2. (a) P(2) as revolving door, (b) projection of $X \subset$ P(2) onto $\det(x) = 1$, (c) $X \subset$ P(2), (d) a flat in P(2), (e) flat of P(2) under $\log(\cdot)$ map. Two horospheres drawn in views (b-d).

In P(2) it is convenient to visualize Busemann functions through horospheres. We can embed P(2) in $\mathbb{R}^3$ where the log of the determinant of elements grows along one axis. The orthogonal planes contain a model of hyperbolic space called the *Poincaré disk* that is modeled as a unit disk, with boundary at infinity represented by the unit circle. Thus the entire space can be seen as a cylinder, as shown in Figure 2(c). Within each cross section with constant determinant (Figure 2(b)), the horoballs are disks tangent to the boundary at infinity. Within each flat F (Figure 2(d)) under a $\log(\cdot)$ map (Figure 2(e)) the horoballs are halfspaces. The full version provides a more technical treatment of this decomposition and proofs of technical lemmas.

Rotation of Busemann functions. The following lemma describes how geodesics (and horofunctions) are transformed by a rotation. In particular, this allows us to pick a flat where computation of b_c is convenient, and rotate the point set by Q to compute b_c instead of attempting computation of $b_{c'}$ directly.

Lemma 1. *For $p \in$ P(n), rotation matrix Q, geodesics $c(t) = e^{tA}$ and $c'(t) = e^{tQAQ^T}$, then $b_{c'}(p) = b_c(Q^T p Q)$.*

Projection to k-flat. We now establish an equivalence between horoballs and halfspaces. That is, after we compute the projection of our point set, we can say that the point set X lies inside a horoball $B_r(b_c)$ if and only if its projection $\pi_F(X)$ lies inside a halfspace H_r of F (recall that F is isometric to a Euclidean space under log).

Lemma 2. *For any horoball $B_r(b_c)$, there is a halfspace $H_r \subset \log(F) \subset S(n)$ such that $\log(\pi_F(B_r(b_c))) = H_r$.*

Proof. If $b_c(p) \leq r$, $p \in$ P(n), and $c(t) = e^{tA}$, then $-\operatorname{tr}(A \log(\pi_F(p))) \leq r$. Since $\pi_F(p)$ is positive-definite, $\log(\pi_F(p))$ is symmetric. But $\operatorname{tr}((\cdot)(\cdot))$ defines an inner product on the Euclidean space of symmetric $n \times n$ matrices. Then the set of all Y such that $-\operatorname{tr}(AY) \leq r$ defines a halfspace H_r whose boundary is perpendicular to A. Furthermore, given a matrix v such that $b(vpv^T) \leq r$ (see the full version), we can compute $ve^Y v^T$ for every $Y \in H_r$, so every such Y maps back to an element of $B_r(b_c)$. □

4.2 A Lipschitz Bound in P(2)

To show our Lipschitz bound we analyze the deviation between two flats with similar directions. Since any two flats F and F' are identified with rotations Q and Q', we can move a point from F to F' simply by applying the rotation $Q^T Q'$, and measure the angle θ between the flats. If we consider a geodesic $c \subset F$ we apply $Q^T Q'$ to c to get c', then for any point $p \in$ P(n) we bound $|b_c(p) - b_{c'}(p)|$ as a function of θ. Technical proofs are in the full version.

Rotations in P(2). We start with some technical lemmas that describe the locus of rotating points in P(2).

Lemma 3. *Given a rotation matrix $Q \in SO(2)$ corresponding to an angle of $\theta/2$, Q acts on a point $p \in P(2)$ via QpQ^T as a rotation by θ about the (geodesic) axis $e^{tI} = e^t I$.*

By Lemma 3, as we apply a rotation to p, it moves in a circle. Because any rotation Q has determinant 1, $\det(QpQ^T) = \det(p)$. This leads to the following corollary:

Corollary 1. *In P(2), the radius of the circle that p travels on is $D(\sqrt{\det(p)}I, p)$. Such a circle lies entirely within a submanifold of constant determinant.*

In fact, any submanifold $P(2)_r$ of points with determinant equal to some $r \in \mathbb{R}^+$ is isometric to any other such submanifold $P(2)_s$ for $s \in \mathbb{R}^+$. This is easily seen by considering the distance function $\text{tr}(\log(p^{-1}q))$; the determinants of p and q will cancel. We pick a natural representative of these submanifolds, $P(2)_1$. This submanifold forms a complete metric space of its own that has special structure:

Lemma 4. *$P(2)_1$ has constant sectional curvature $-1/2$.*

To bound the error incurred by discretizing the space of directions, we need to understand the behavior of b_c as a function of a rotation Q. We show that the derivative of a geodesic is constant on $P(n)$.

Lemma 5. *For a geodesic ray $c(t) = e^{tA}$ where $\|A\| = 1$, then $\|\nabla b_c\| = 1$ at any point $p \in P(n)$.*

Lipschitz condition on Busemann functions in P(2).

Theorem 4. *Consider $Q \in SO(2)$ corresponding to $\theta/2$, $c(t) = e^{tA}$, and $c'(t) = e^{tQ^T AQ}$. Then for any $p \in X$*

$$|b_c(p) - b_{c'}(p)| \leq |\theta| \cdot \sqrt{2}\sinh\left(\frac{D_X}{\sqrt{2}}\right).$$

Proof. The derivative of a function f along a curve $\gamma(t)$ has the form $\langle \nabla f|_{\gamma(t)}, \gamma'(t) \rangle$, and has greatest magnitude when the tangent vector $\gamma'(t)$ to the curve and the gradient $\nabla f|_{\gamma(t)}$ are parallel. When this happens, the derivative reaches its maximum at $\|\nabla f|_{\gamma(t)}\| \cdot \|\gamma'(t)\|$. Since $\|\nabla b_c\| = 1$ anywhere by Lemma 5, the derivative of b_c along γ at $\gamma(t)$ is bounded by $\|\gamma'(t)\|$. We are interested in the case where $\gamma(\theta)$ is the circle in P(2) defined by tracing $Q(\theta/2)pQ(\theta/2)^T$ for all $-\pi < \theta \leq \pi$. By Corollary 1, we know that this circle has radius $D(\sqrt{\det(p)}I, p) \leq D(I, p) \leq D_X$ and lies entirely within a submanifold of constant determinant, which by Lemma 4 also has constant curvature $\kappa = -1/2$. This implies that

$$\|\gamma'(\theta)\| = \frac{1}{\sqrt{-\kappa}}\sinh(\sqrt{-\kappa}\,r) = \sqrt{2}\sinh\left(\frac{D(\sqrt{\det(p)}I, p)}{\sqrt{2}}\right) \leq \sqrt{2}\sinh\left(\frac{D_X}{\sqrt{2}}\right)$$

for any value of $\theta \in (-\pi, \pi]$ [8, I.6]. Then

$$|b_c(p) - b_{c'}(p)| = |b_c(p) - b_c(Q^T pQ)| \leq |\theta| \cdot \sqrt{2}\sinh\left(\frac{D_X}{\sqrt{2}}\right). \qquad \square$$

4.3   Generalizing to P(n)

Now to generalize to P(n) we need to decompose the projection operation $\pi_F(\cdot)$ and the rotation matrix Q. We can compute π_F recursively, and it turns out that this fact helps us to break down the analysis of rotations. Since we can decompose any rotation into a product of $\lfloor n/2 \rfloor$ 2×2 rotation matrices, decomposing the computation of π_F in a similar manner lets us build a Lipschitz condition for P(n). The full version formalizes rotation decompositions we use.

Theorem 5 (Lipschitz condition on Busemann functions in P(n)). *Consider a set $X \subset P(n)$, a rotation matrix $Q \in SO(n)$ corresponding to an angle $\theta/2$, geodesics $c(t) = e^{tA}$ and $c'(t) = e^{tQAQ^T}$. Then for any $p \in X$*

$$|b_c(p) - b_{c'}(p)| \le |\theta| \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \sqrt{2} \sinh\left(\frac{D_X}{\sqrt{2}}\right).$$

Proof. Every rotation Q may be decomposed into a product of rotations, relative to some orthonormal basis B; that is, $Q = B(Q_1 Q_2 \cdots Q_{k-1} Q_k) B^T$ where $k = \lfloor n/2 \rfloor$ and Q_i is a 2×2 subblock rotation corresponding to an angle $\theta_i/2$ with $|\theta_i| \le |\theta|$. Now applying Lemma 1, we can factor out B: $c'(t) = e^{tQAQ^T} = e^{tB(Q_1 \cdots Q_k)B^T AB(Q_k^T \cdots Q_1^T)B^T}$, and let $\hat{c}'(t) = e^{t(Q_1 \cdots Q_k)\hat{A}(Q_k^T \cdots Q_1^T)}$, where $\hat{A} = B^T AB$. This means that $b_{c'}(p) = b_{\hat{c}'}(B^T pB) = b_{\hat{c}'}(\hat{p})$ for $\hat{p} = B^T pB$. Also, since $c(t) = e^{tA} = e^{tB\hat{A}B^T}$, $b_c(p) = b_{\hat{c}}(\hat{p})$ for $\hat{c}(t) = e^{t\hat{A}}$. From this point we will omit the "hat" notation and just assume the change of basis.

Then $|b_c(p) - b_{c'}(p)| = \left| \sum_{i=1}^{k} (b_{c'}^{i-1}(p) - b_{c'}^i(p)) \right| \le \sum_{i=1}^{k} |b_{c'}^{i-1}(p) - b_{c'}^i(p)|,$

where $b_{c'}^0(p) = b_c(p)$ and $b_{c'}^i(p)$ is $b_c(p)$ with the first i rotations successively applied, so $b_{c'}^i(p) = b_c(Q_i^T \cdots Q_1^T p Q_1 \cdots Q_i)$. Then by Theorem 4 $|b_{c'}^{i-1}(p) - b_{c'}^i(p)| \le |\theta_i| \cdot \sqrt{2} \sinh\left(\frac{D_X}{\sqrt{2}}\right)$, and therefore, since for all i we have $|\theta_i| \le |\theta|$,

$$|b_c(p) - b_{c'}(p)| \le \left(\sum_{i=1}^{k} |\theta_i| \right) \cdot \sqrt{2} \sinh\left(\frac{D_X}{\sqrt{2}}\right) \le |\theta| \cdot \left\lfloor \frac{n}{2} \right\rfloor \cdot \sqrt{2} \sinh\left(\frac{D_X}{\sqrt{2}}\right). \quad \square$$

4.4   Algorithm

For $X \subset P(n)$ we can construct ε-ball hull as follows. We place a grid G_ε on $SO(n)$ so that for any $Q' \in SO(n)$, there is another $Q \in G_\varepsilon$ such that the angle between Q and Q' is at most $(\varepsilon/4)/(2\lfloor n/2 \rfloor \sqrt{2} \sinh(D_X/\sqrt{2}))$. For each $Q \in G_\varepsilon$, we consider $\pi_F(X)$, the projection of X into the associated n-flat F associated with Q. Within F, we build an $(\varepsilon/4D_X)$-kernel [1] K_F of $\log(\pi_F(X))$, and return the horoball associated with the hyperplane passing through each facet of $\mathcal{C}(K_F)$ in F, using the transformation specified in Lemma 2. (This step can be replaced with an exact convex hull of $\log(\pi_F(X))$.) This produces a coreset of X for extents, $K = \bigcup_F K_F$ for all F associated with a $Q \in G_\varepsilon$.

   To analyze this algorithm we can now consider any direction $Q' \in SO(n)$ and a horofunction $b_{c'}$ that lies in the associated flat F'. There must be another direction $Q \in G_\varepsilon$ such that the angle between Q and Q' is at most $(\varepsilon/4)/(2\lfloor n/2 \rfloor \sqrt{2} \sinh(D_X/\sqrt{2}))$. Let b_c be the similar horofunction to $b_{c'}$, except it lies in the flat F associated with Q. This ensures that for any point $p \in X$, we have $|b_{c'}(p) - b_c(p)| \le \varepsilon/4$. Also, for $p \in X$, there

is a $q \in K_F$ such that $b_c(p) - b_c(q) \leq (\varepsilon/4D_X) \cdot D_X = \varepsilon/4$. Thus $b_{c'}(p) - b_c(q) \leq \varepsilon/2$. Since $E_{c'}(X)$ depends on two points in X, and each point p changes at most $\varepsilon/2$ from $b_{c'}(p)$ to $b_c(q)$, we can argue that $|E_{c'}(X) - E_c(K)| \leq \varepsilon$. Since this holds for any direction $Q' \in \text{SO}(n)$, the returned set of horoballs defines an ε-ball hull.

Let $\Gamma_n(\varepsilon, D_X) = (\sinh(D_X)/\varepsilon)^{n-1}$. For constant n, the grid G_ε is size $O(\Gamma_n(\varepsilon, D_X))$. In each flat, the $(\varepsilon/4D_X)$-kernel is designed to have convex hull with $O((\varepsilon/D_X)^{(n-1)/2})$ facets [1,12], and can be computed in $O(N + (D_X/\varepsilon)^{n-3/2})$ time. Thus a ε-ball hull represented as the intersection of $O(\Gamma_n(\varepsilon, D_X) \cdot (D_X/\varepsilon)^{(n-1)/2})$ horoballs can be computed in $O(\Gamma_n(\varepsilon, D_X) \cdot ((D_X/\varepsilon)^{n-3/2} + N))$ time, proving Theorem 3.

References

1. Agarwal, P.K., Har-Peled, S., Varadarajan, K.R.: Approximating extent measures of points. JACM 51 (2004)
2. Arnaudon, M., Nielsen, F.: On Approximating the Riemannian 1-Center, arXiv:1101.4718v1 (2011)
3. Basser, P.J., Mattiello, J., LeBihan, D.: MR diffusion tensor spectroscopy and imaging. Biophys. J. 66(1), 259–267 (1994)
4. Belkin, M., Niyogi, P.: Semi-supervised learning on Riemannian manifolds. Machine Learning 56(1), 209–239 (2004)
5. Bhatia, R.: Positive Definite Matrices. Princeton University Press, Princeton (2006)
6. Billera, L., Holmes, S., Vogtmann, K.: Geometry of the Space of Phylogenetic Trees. Advances in Applied Mathematics 27(4), 733–767 (2001)
7. Bonnabel, S., Sepulchre, R.: Riemannian metric and geometric mean for positive semidefinite matrices of fixed rank. SIAM J. on Matr. Anal. & App. 31(3), 1055–1070 (2010)
8. Bridson, M.R., Haefliger, A.: Metric Spaces of Non-Positive Curvature. Springer, Heidelberg (2009)
9. Chepoi, V., Dragan, F., Estellon, B., Habib, M., Vaxès, Y.: Diameters, centers, and approximating trees of delta-hyperbolicgeodesic spaces and graphs. In: SoCG (2008)
10. Chepoi, V., Estellon, B.: Packing and covering δ-hyperbolic spaces by balls. In: Charikar, M., Jansen, K., Reingold, O., Rolim, J.D.P. (eds.) RANDOM 2007 and APPROX 2007. LNCS, vol. 4627, pp. 59–73. Springer, Heidelberg (2007)
11. Cowin, S.: The structure of the linear anisotropic elastic symmetries. Journal of the Mechanics and Physics of Solids 40, 1459–1471 (1992)
12. Dudley, R.M.: Metric entropy of some classes of sets with differentiable boundaries. Journal of Approximation Theory 10, 227–236 (1974)
13. Dyer, R., Zhang, H., Möller, T.: Surface sampling and the intrinsic Voronoi diagram. In: SGP, pp. 1393–1402 (2008)
14. Eppstein, D.: Squarepants in a tree: Sum of subtree clustering and hyperbolic pants decomposition. ACM Transactions on Algorithms (TALG) 5 (2009)
15. Fletcher, P.T., Joshi, S.: Principal geodesic analysis on symmetric spaces: Statistics of diffusion tensors. Com. Vis. & Math. Methods Med. Biomed. Im. Anal, 87–98 (2004)
16. Fletcher, P.T., Venkatasubramanian, S., Joshi, S.: The geometric median on Riemannian manifolds with application to robust atlas estimation. NeuroImage 45, S143–S152 (2009)

17. Gromov, M.: Hyperbolic groups. Essays in Group Theory 8, 75–263 (1987)
18. Krauthgamer, R., Lee, J.R.: Algorithms on negatively curved spaces. In: FOCS (2006)
19. Leibon, G., Letscher, D.: Delaunay triangulations and Voronoi diagrams for Riemannian manifolds. In: SoCG (2000)
20. Meyer, G., Bonnabel, S., Sepulchre, R.: Regression on fixed-rank positive semidefinite matrices: a Riemannian approach, arXiv:1006.1288 (2010)
21. Moakher, M.: A Differential Geometric Approach to the Geometric Mean of Symmetric Positive-Definite Matrices. SIAM J. on Matr. Anal. & App. 26 (2005)
22. Moakher, M., Batchelor, P.G.: Symmetric positive-definite matrices: From geometry to applications and visualization. In: Visualization and Processing of Tensor Fields (2006)
23. Nesterov, Y., Todd, M.: On the Riemannian geometry defined by self-concordant barriers and interior-point methods. FoCM 2(4), 333–361 (2008)
24. Owen, M., Provan, J.: A Fast Algorithm for Computing Geodesic Distances in Tree Space, arXiv:0907.3942 (2009)
25. Pennec, X., Fillard, P., Ayache, N.: A Riemannian framework for tensor computing. IJCV 66(1), 41–66 (2006)
26. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis, Cambridge (2004)
27. Smith, S.: Covariance, subspace, and intrinsic Cramér-Rao bounds. IEEE Transactions on Signal Processing 53(5), 1610–1630 (2005)
28. Vandereycken, B., Absil, P., Vandewalle, S.: A Riemannian geometry with complete geodesics for the set of positive semidefinite matrices of fixed rank. status: submitted (2010)

Enumerating Minimal Subset Feedback Vertex Sets⋆

Fedor V. Fomin[1], Pinar Heggernes[1], Dieter Kratsch[2], Charis Papadopoulos[3], and Yngve Villanger[1]

[1] Department of Informatics, University of Bergen, Norway
{fedor.fomin,pinar.heggernes,yngve.villanger}@ii.uib.no
[2] LITA, Université Paul Verlaine – Metz, France
kratsch@univ-metz.fr
[3] Department of Mathematics, University of Ioannina, Greece
charis@cs.uoi.gr

Abstract. The SUBSET FEEDBACK VERTEX SET problem takes as input a weighted graph G and a vertex subset S of G, and the task is to find a set of vertices of total minimum weight to be removed from G such that in the remaining graph no cycle contains a vertex of S. This problem is a generalization of two classical NP-complete problems: FEEDBACK VERTEX SET and MULTIWAY CUT. We show that it can be solved in time $O(1.8638^n)$ for input graphs on n vertices. To the best of our knowledge, no exact algorithm breaking the trivial $2^n n^{O(1)}$-time barrier has been known for SUBSET FEEDBACK VERTEX SET, even in the case of unweighted graphs. The mentioned running time is a consequence of the more general main result of this paper: we show that all minimal subset feedback vertex sets of a graph can be enumerated in $O(1.8638^n)$ time.

1 Introduction

Given a graph $G = (V, E)$ and a set $S \subseteq V$, a *subset feedback vertex set* of (G, S) is a set $X \subseteq V$ such that no cycle in $G[V \setminus X]$ contains a vertex of S. A subset feedback vertex set is *minimal* if no proper subset of it is a subset feedback vertex set. Given a *weighted* graph G with positive real weights on its vertices and S as input, the SUBSET FEEDBACK VERTEX SET problem is the problem of finding a subset feedback vertex set X of (G, S) such that the sum of weights of the vertices in X is minimized.

SUBSET FEEDBACK VERTEX is a generalization of several well-known problems. When $S = V$, it is equivalent to the classical NP-hard FEEDBACK VERTEX SET problem [10]. When $|S| = 1$, it generalizes the MULTIWAY CUT problem. Given a set $T \subseteq V$, called *terminals*, a *multiway cut* of (G, T) is a set of vertices whose removal from G disconnects every pair of terminals. Given a graph $G = (V, E)$, with weights on its vertices, and $T \subseteq V$, the MULTIWAY CUT problem is the problem of computing a multiway cut of total minimum weight. It

---

follows that this is a special case of SUBSET FEEDBACK VERTEX SET by adding a singleton vertex with a large weight to the set S and making it adjacent to all terminals in T. The *unweighted* versions of the three above mentioned problems are obtained when the weight of every vertex of the input graph is 1. For further results on variants of multiway cut problems see [2,11] and for connections between variants of the subset feedback vertex set problem and the multiway cut problems see also [4]. SUBSET FEEDBACK VERTEX SET was first studied by Even et. al. who obtained a constant factor approximation algorithm [5]. In this paper we are interested in an exact solution of SUBSET FEEDBACK VERTEX SET. This does not seem to have been studied before, whereas there are a series of exact results on FEEDBACK VERTEX SET. Razgon [13] gave the first improvement over the trivial exact algorithm for unweighted FEEDBACK VERTEX SET. This result has been improved by Fomin et. al. [6,9]. A minimum feedback vertex set in an unweighted graph can be computed in $O(1.7548^n)$ time [6]. Furthermore, all minimal feedback vertex sets can be enumerated in $O(1.8638^n)$ time [6]. The latter result implies that a minimum weight feedback vertex set can be computed in $O(1.8638^n)$ time. So far, this is the best known algorithm for FEEDBACK VERTEX SET. Fixed parameter tractability of SUBSET FEEDBACK VERTEX SET was raised as an open question in [1], and only recently it was proved to be fixed parameter tractable [4], whereas FEEDBACK VERTEX SET has long been known to be fixed parameter tractable [3,12,14].

In this paper, we show that SUBSET FEEDBACK VERTEX SET can be solved in time $O(1.8638^n)$. Prior to our result, no algorithm breaking the trivial $2^n n^{O(1)}$-time barrier has been known, even for the unweighted version of the problem. As our main result, we give an algorithm that enumerates all minimal subset feedback vertex sets of (G, S) and runs in $O(1.8638^n)$ time. Thus our running time matches the best-known algorithm for enumerating all minimal feedback vertex sets [6]. While the general branching approach for enumerating the subset feedback vertex sets is similar to the one enumerating the feedback vertex sets [6], we introduce and use here several non-trivial ideas and new techniques for the subset variant of the problem. As mentioned above, our enumeration algorithm can be trivially adapted to an algorithm computing a minimum weight subset feedback vertex set in time $O(1.8638^n)$. Furthermore, by making use of the reduction above, our algorithm can be used to enumerate all minimal multiway cuts within the same running time. As a consequence, we are also able to solve MULTIWAY CUT in time $O(1.8638^n)$. To our knowledge, this is the first non-trivial exact algorithm for the solution of this latter problem, even for its unweighted version. Some proofs has been moved due to space restrictions.

2   Preliminaries

All graphs in this paper are undirected and with weights on their vertices. All graphs are simple unless explicitly mentioned; in particular input graphs are always simple, but during the course of our algorithm, multiple edges are introduced due to contraction of edges. A graph is denoted by $G = (V, E)$ with vertex set V and edge set E. We use the convention that $n = |V|$ and $m = |E|$.

Each vertex $v \in V$ is assigned a weight that is a positive real number. For a vertex set $X \subseteq V$ the weight of X is the sum of the weights of all vertices in X, and the *subgraph of G induced by X* is denoted by $G[X]$. The *neighborhood* of a vertex v of G is $N(v) = \{x \mid vx \in E\}$. For $X \subseteq V$, $N(X) = \bigcup_{x \in X} N(x) \setminus X$. In this paper, we distinguish between paths (cycles) and induced paths (induced cycles). A path (cycle) of G is *induced* if there are no edges in G between non-consecutive vertices of the path (cycle). An edge of G is called a *bridge* if its removal increases the number of connected components. A *forest* is a graph that contains no cycles and a *tree* is a forest that is connected. The *contraction* of edge $\{u, v\}$ removes u and v from the graph, and replaces them with a new vertex that is incident with every edge that was incident with u or v. If we say that edge $\{u, v\}$ is *contracted to u*, then u takes the role of the new vertex after the contraction. Clearly, multiple edges might result from this operation.

Note that a *minimum weight* (or simply *minimum*) subset feedback vertex set is dependent on the weights of the vertices, whereas a *minimal* subset feedback vertex set is only dependent on the vertices and not their weights. Clearly, both in the weighted and the unweighted versions, a minimum subset feedback vertex set must be minimal.

3   Enumeration of All Minimal Subset Feedback Vertex Sets

Let $G = (V, E)$ be a graph and let $S \subseteq V$. In this section we give an algorithm that enumerates all minimal subset feedback vertex sets of (G, S).

We define an *S-forest* of G to be a vertex set $Y \subseteq V$ such that $G[Y]$ contains no cycle with a vertex from S. An S-forest Y is *maximal* if no proper superset of Y is an S-forest. Observe that X is a minimal subset feedback vertex set if and only if $Y = V \setminus X$ is a maximal S-forest. Thus, the problem of enumerating all minimal subset feedback vertex sets is equivalent to the problem of enumerating all maximal S-forests. Thus we present an algorithm enumerating all maximal S-forests of the input graph G which is equivalent to enumerating all subset feedback vertex sets of (G, S).

Our algorithm is a branching algorithm consisting of a sequence of reduction and branching rules. The running time of the algorithm is up to a polynomial factor proportional to the number of generated subproblems, or to the number of nodes of the branching tree. For more information on branching algorithms and Measure & Conquer analysis of such algorithms we refer to [8].

In our algorithm each subproblem corresponding to a leaf of the branching tree will define an S-forest, and each maximal S-forest will be defined by one leaf of the branching tree. Each of the reduction and branching rules will reduce the problem instance by making progress towards some S-forest.

To incorporate all information needed in the algorithm we use so-called red-blue S-forests. Given a set $B \subseteq V$ of *blue* vertices with $B \cap S = \emptyset$ and a set $R \subseteq V$ of *red* vertices with $R \subseteq S$, a maximal *red-blue S-forest* of G is a maximal S-forest Y of G such that $R \cup B \subseteq Y$. If the set $R \cup B$ of vertices

have the property that no two red vertices, or no two blue vertices, are adjacent, then we say that the red-blue coloring of these vertices is a *proper 2-coloring*. Let $RBF(G, S, R, B)$ be the set of all maximal red-blue S-forests in G. Hence a maximal S-forest Y is an *element* of $RBF(G, S, R, B)$ if $R \cup B \subseteq Y$. Observe that the problem of enumerating all maximal S-forests of G is equivalent to enumerating all elements of $RBF(G, S, \emptyset, \emptyset)$. We refer to the vertices of $V \setminus (R \cup B)$ as *non-colored*. Before proceeding with the description of the algorithm, we need the following observations concerning the set $RBF(G, S, R, B)$.

Observation 1. *Let $Y = R \cup B$ be an S-forest of G that is an element of $RBF(G, S, R, B)$. Let G' be the graph obtained from $G[Y]$ by contracting every edge whose endpoints have the same color, giving the resulting vertex that same color, and removing self loops and multiple edges. Then G' is a forest. Moreover, red and blue vertices form a proper 2-coloring of G'.*

Let Y be an S-forest of G and let $u \in V \setminus Y$. If $G[Y \cup \{u\}]$ contains an induced cycle C_u that contains u and some vertex of S, then we say that C_u is a *witness cycle* of u.

Observation 2. *Let Y be a maximal S-forest of G. Then every vertex $u \in V \setminus Y$ has a witness cycle C_u.*

We are ready to proceed with the description of the enumeration algorithm. The description of the algorithm is given by a sequence of reduction and branching rules. We always assume that the rules are performed in the order in which they are given (numbered), such that a rule is only applied if none of the previous rules can be applied.

Initially all vertices of G are non-colored. Vertices that are colored red or blue have already been decided to be in every maximal S-forest that is an element of $RBF(G, S, R, B)$. For a non-colored vertex v, we branch on two subproblems, and the cardinality of $RBF(G, S, R, B)$ is the sum of cardinalities of the sets of maximal S-forests that contain v and those that do not. The first set is represented by coloring vertex v red or blue, and second set is obtained by deleting v. This partitioning defines a naive branching, where a leaf is reached when there is at most one maximal S-forest in the set. We define the following two procedures, which take as input vertex v and $RBF(G, S, R, B)$.

> **Coloring** of vertex v:
> – **if $v \in S$ then** proceed with $RBF(G, S, R \cup \{v\}, B)$;
> – **if $v \notin S$ then** proceed with $RBF(G, S, R, B \cup \{v\})$.
>
> **Deletion** of vertex v:
> – proceed with $RBF(G[V \setminus \{v\}], S \setminus \{v\}, R, B)$.

After the description of each of the Rules 1-12, we argue that the rule is *sound*, which means that there is a one-to-one correspondence between the maximal S-forests in the problem instance and the maximal S-forests in the instances of the subproblem(s). We start to apply the rules on instance $RBF(G, S, \emptyset, \emptyset)$.

Rule 1. *If G has a vertex of degree at most 1 then remove this vertex from the graph.*

Rule 1 is sound because a vertex of degree zero or one does not belong to any cycle. Furthermore, when a vertex of degree zero or one is removed, every vertex that previously belonged to a cycle still belongs to a cycle and maintains degree at least two.

Note that removal of vertex v means that v belongs to every element of the set $RBF(G, S, R, B)$. We emphasize that there is a crucial difference to **Deletion** of vertex v which means that the non-colored vertex v belongs to no element of $RBF(G, S, R, B)$. Such a removal of a vertex belonging to every element of $RBF(G, S, R, B)$ is done in Rules 1, 4 and 5 and it necessitates the backtracking part of our algorithm to be explained later.

Rule 2. *If $R = \emptyset$, and $S \neq \emptyset$ then select an arbitrary non-colored vertex $v \in S$, and branch into two subproblems. One subproblem is obtained by applying **Deletion** of v and the other by **Coloring** of v.*

Rule 2 is sound for the following reason. Only vertices of S are colored red. Thus if $R = \emptyset$, all vertices of S are non-colored vertices. For every maximal S-forest Y, we have that either $v \in Y$ (corresponding to **Coloring** of v), or $v \notin Y$ (corresponding to **Deletion** of v).

After the application of Rule 2 there always exists a red vertex, unless $S = \emptyset$ when we reached a leaf of the branching tree. For many of the following rules we need to fix a particular vertex t of the S-forest $R \cup B$. We call it *pivot vertex* t. If no pivot vertex exists (at some step a pivot vertex might be deleted), we apply the following rule to select a new one.

Rule 3. *If there is no pivot vertex then select a red vertex as new pivot vertex t.*

The following reduction rule is to ensure (by making use of Observation 1) that the graph $G[R \cup B]$ induces a forest and that the current red-blue coloring is a proper 2-coloring of this forest.

Rule 4. *If there are two adjacent red vertices u, v, then contract edge $\{u, v\}$ to u to obtain a new graph G'. Let Z be the set of non-colored vertices that are adjacent to u via multiple edges in G'. If v was the pivot then use u as new pivot t. Proceed with problem instance $RBF(G' \setminus Z, S \setminus (\{v\} \cup Z), R \setminus \{v\}, B)$.*

Observe that Rule 4 corresponds to applying **Deletion** of w for every vertex w of Z. Let us argue why this rule is sound. If a vertex w belongs to Z, then because $u, v \in S$, we have that w cannot be in any S-forest of G. Thus applying **Deletion** of this vertex does not change the set of maximal S-forests. Finally, every cycle of length more than 3 in G corresponds to a cycle of length at least 3 in the reduced instance.

Rule 5. *If there are two adjacent blue vertices u, v, then contract edge $\{u, v\}$ to u to obtain a new graph G'. Let Z be the set of non-colored vertices of S that*

are adjacent to u via multiple edges in G'. We replace each set of multiple edges between u and a vertex of Z with a single edge. If $t \in \{u, v\}$ then use u as pivot t. New problem instance is $RBF(G' \setminus Z, S \setminus Z, R, B \setminus \{v\})$.

Observe that Rule 5 corresponds to applying **Deletion** of w for every vertex w of Z. No vertex of Z can be in an S-forest containing u and v. Thus applying **Deletion** of the vertices of Z is sound. As with the previous rule, every cycle of length more than 3 in G corresponds to a cycle of length at least 3 in the reduced instance. We conclude that Rule 5 is sound.

If none of Rules 1-5 can be applied to the current instance, and in particular Rules 4 and 5 cannot be applied, we can assume that $R \cup B$ induces a forest and that the red-blue coloring is a proper 2-coloring of this forest. We will call such a forest (tree) a red-blue forest (tree).

Rule 6. *If a non-colored vertex v has at least two distinct neighbors w_1, w_2 in the same connected component of $G[R \cup B]$, then apply **Deletion** of v.*

As we already mentioned, the connected component of $G[R \cup B]$ that contains w_1 and w_2 is a properly 2-colored red-blue tree T. Let $w_1, u_1, u_2, \ldots, u_p = w_2$, $p \geq 1$, be the unique induced path in T between w_1 and w_2. Then either w_1 or u_1 is a red vertex, and thus no element of $RBF(G, S, R, B)$ contains v. This shows that Rule 6 is sound.

Let T_t be the vertices of the connected component of $G[R \cup B]$ containing the pivot vertex t. Consider a non-colored vertex v adjacent to a vertex of the red-blue tree $G[T_t]$. Observe that v has exactly one neighbor w in T_t, by Rule 6. By Observation 2, every vertex u, which is not in a maximal S-forest Y, should have a witness cycle C_u such that all vertices of C_u except u are in Y. Hence every vertex $u \notin Y$ has at least two neighbors in Y. Since we cannot apply Rule 6 on vertex v, this implies that if v is not in Y, at least one of the vertices from $N(v) \setminus T_t$ is in Y.

For a non-colored vertex v adjacent to a vertex of T_t, we define vertex set $P(v)$ to be the set of non-colored vertices adjacent to v or reachable from v via induced red-blue paths in $G[V \setminus T_t]$. Let w be the unique neighbor of v in T_t. We define vertex set $PW(v)$ to be the subset of $P(v)$ consisting of every vertex x of $P(v)$ for which at least one of the following conditions holds:

P1 $\{w, v, x\} \cap S \neq \emptyset$,
P2 $x \notin N(w)$, or
P3 there exists an induced red-blue path from x to v in $G[V \setminus T_t]$ containing at least one red vertex.

See Fig. 1 for an example of sets $P(v)$ and $PW(v)$. The intuition behind the definition of $PW(v)$ is the following. If vertex v does not belong to any maximal S-forest Y of G, then there is a witness cycle C_v. This cycle C_v may pass through some connected components of $G[R \cup B]$ and some non-colored vertices. If we traverse C_v starting from v and avoiding T_t, then the first non-colored vertex we meet will be a vertex of $PW(v)$. Note that the vertex set $PW(v)$ can easily be computed in polynomial time.
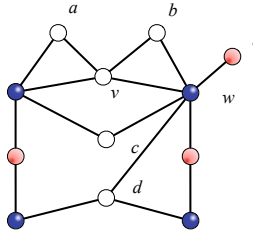
Fig. 1. Let $S \cap \{a, b, c, d, v\} = \emptyset$ and set $P(v) = \{a, b, c, d\}$. Vertex $a \in PW(v)$ by (P2), $d \in PW(v)$ by (P3). Vertices b and c do not belong to $PW(v)$.

Observation 3. *For every vertex $x \in PW(v) \cap N(T_t)$, there is an induced cycle containing x and v and at least one vertex of S.*

Observation 4. *Let v be a non-colored vertex adjacent to a vertex of T_t. If there is an induced cycle C in G that contains v and some vertex of S, then C contains also at least one vertex of $P(v)$.*

Lemma 5. *A witness cycle C_v of a vertex v contains a vertex of $PW(v)$.*

Proof. Let us assume that v is not contained in a maximal S-forest Y, and let C_v be a witness cycle for v. By Observation 4, C_v contains at least one vertex x of $P(v)$.

If $x \in PW(v)$, we are done with the proof. Otherwise, by Observation 4, $x \in P(v) \setminus PW(v)$. As a consequence, $v, w, x \notin S$, x is adjacent to w, and every induced red-blue path from v to x in $G[V \setminus T_t]$ contains only blue vertices.

Let us now trace the induced cycle C_v, starting from v on the path to x using only blue vertices. By definition, no vertex on the path from v to x is contained in S. Continue now in the same direction along C_v until a vertex of S is reached. The cycle has to return to v without passing through vertex w; otherwise the edge $\{x, w\}$ would be a chord of C_v contradicting the fact that C_v is an induced cycle. The path from x to v containing a vertex of S cannot be a red-blue path as this contradicts the definition of x. As a consequence, C_v contains a second vertex x' of $P(v) \setminus PW(v)$. Maximal S-forest Y contains w because it is colored blue, and $C_v \setminus Y = \{v\}$ due to the maximality. Now we have a contradiction since the path P from x to x' on C_v not containing v, contains at least one vertex of S, and because of the edges $\{x, w\}$ and $\{x', w\}$, the graph $G[P \cup \{w\}]$ induces a cycle containing a vertex of S.   □

The following rules depend on the cardinality of the set $PW(v)$. Rules 7 and 8 are sound due to Lemma 5.

Rule 7. *If $PW(v) = \emptyset$ then apply **Coloring** of v.*

Rule 8. *If $PW(v) = \{x\}$ then branch into two subproblem instances: one obtained by applying **Deletion** of v and then **Coloring** of x, and the other obtained by applying **Coloring** of v.*

Rule 9. *If $|PW(v)| \geq 2$ and $PW(v) \subseteq N(T_t)$ then branch into two subproblem instances: one obtained by applying **Coloring** of v and then **Deletion** of x to all vertices $x \in PW(v)$, and the other obtained by **Deletion** of v.*

To see that Rule 9 is sound, observe that vertex v is either colored, or deleted. By Observation 3, for each vertex $x \in PW(v)$ the induced subgraph $G[T_t \cup \{x, v\}]$ contains a cycle with a vertex of S and the non-colored vertices v and x. Thus either x or v has to be deleted, for every $x \in PW(v)$. When Rule 9 can not be applied, at least one of the vertices in $PW(v)$ is not contained in $N(T_t)$.

Rule 10. *If $PW(v) = \{x_1, x_2\}$ and $x_1 \notin N(T_t)$ then branch into three subproblem instances. The first one is obtained by applying **Coloring** of v. The second by **Deletion** of v and then **Coloring** of x_1. The third one by applying **Deletion** of v and x_1 and then **Coloring** of x_2.*

Let us remark that vertex v is either colored or deleted. If v is deleted then by Lemma 5, either x_1 or x_2 is contained in the witness cycle C_v. This shows that Rule 10 is sound.

Rule 11. *If $PW(v) = \{x_1, x_2, x_3\}$ and $x_1 \notin N(T_t)$ then branch into four subproblem instances. The first instance is obtained by applying **Coloring** of v. The second by **Deletion** of v and then **Coloring** of x_1. The third by **Deletion** of v and x_1 and then **Coloring** of x_2. The fourth by **Deletion** of v, x_1, x_2 and **Coloring** of x_3.*

Again, the soundness of this rule follows by Lemma 5.

Rule 12. *If $|PW(v)| \geq 4$ then create two problem instances: one obtained by applying **Coloring** of v, and the other obtained by applying **Deletion** of v.*

This rule is sound because v is either colored or deleted.

We call an instance *non-reducible* if none of Rules 1-12 can be applied to it. Such an instance corresponds to a leaf of the branching tree of our algorithm. The following property of non-reducible instances of the the red-blue S-forest problem is crucial for our arguments.

Lemma 6. *Let (G, S, B, R) be an instance. If none of the Rules 1–12 can be applied then $RBF(G, S, R, B)$ contains at most one maximal red-blue S-forest. Moreover, this forest can be computed in polynomial time.*

Proof. If $S = \emptyset$ then trivially the only maximal S-forest of G is V. Let us assume that $S \neq \emptyset$. With every rule we either remove a vertex, select a pivot vertex, color a vertex, delete a vertex or contract an edge. Rule 2 guarantees that the set of red vertices is not empty. Rule 3 ensures that a pivot vertex t is selected. Rules 1, 2 and 4–12 can be applied as long as there are non-colored neighbors of red-blue tree T_t. When the set $N(T_t)$ becomes empty then T_t is completely removed by Rule 1. Then the algorithm selects a new pivot vertex t and component T_t by making use of Rule 3. Thus the conditions that none of the rules can be applied and $S \neq \emptyset$, yield that $V = R \cup B$. But then the only possible maximal S-forest Y of $RBF(G, S, R, B)$ is $Y = R \cup B$. $\square$

We are finally in the position to describe the algorithm. The algorithm enumerates all elements of $RBF(G, S, \emptyset, \emptyset)$ by applying Rules 1–12 in priority of their numbering as long as possible. Let $\mathcal{F}$ be the set of all non-reducible instances produced by the application of the rules. These are the instances corresponding to the leaves of the branching tree. By Lemma 6, for each non-reducible instance of $\mathcal{F}$ there is at most one red-blue S-forest which can be computed in polynomial time. To enumerate all maximal S-forests of the input graph, we have to add to each S-forest of an instance of $\mathcal{F}$ all vertices which were possibly removed by applications of some of the rules on the unique path from the root of the branching tree to the corresponding leaf. This can be done in polynomial time by backtracking in the branching tree.

The correctness of the algorithm follows by Lemma 6 and the fact that each rule is sound. Thus each maximal S-forests of the input graph can be mapped to a private element of $\mathcal{F}$. In the next section we analyze the running time.

4   Running Time

With every rule we either remove a vertex, select a pivot vertex, color a vertex, delete a vertex, or contract an edge. Thus the height of the branching tree is $O(|V| + |E|)$. Hence, for every non-reducible instance, the backtracking part of the algorithm producing the corresponding maximal S-forest in G can be performed in polynomial time. Therefore, the running time of the algorithm, up to a polynomial multiplicative factor, is proportional to the number of non-reducible instances produced by reduction and branching rules.

In what follows, we upper bound the number of maximal S-forests of the input graph enumerated by the algorithm, or equivalently, the number of leaves in the corresponding branching tree. Rules 1, 3, 4, 5, 6, and 7 are reduction rules and generate only one problem instance. Thus they do not increase the number of leaves in the branching tree. Therefore we may restrict ourselves to the analyses of the branching Rules 2, 8, 9, 10, 11, and 12 .

Our proof combines induction with Measure & Conquer [7]. Let us first define a measure for any problem instance generated by the algorithm. All colored vertices have weight 0, non-colored vertices contained in $N(T_t)$ have weight 1, and non-colored vertices not contained in $N(T_t)$ have weight $1 + \alpha$. A problem instance $RBF(G, S, R, B)$ will be defined to have weight $|N(T_t)| + (1 + \alpha)|V \setminus (R \cup B \cup N(T_t))|$. Define $f(\mu)$ to be the maximum number of maximal red-blue S-forests for any instance $RBF(G, S, R, B)$ of weight μ where $\mu \geq 0$ is a real number.

The induction hypothesis is that $f(\mu) \leq x^\mu$ for $x = 1.49468$. Note that the number of possible measures of problem instances is finite, and thus induction is over a finite set.

Base case $\mu = 0$. Since no vertex has weight greater than 0, we have that all vertices are colored, and thus V is the unique maximal S-forest, implying $f(0) = 1$. By the induction hypothesis we assume that $f(k) \leq x^k$ for $k < \mu$, and we want to prove that $f(\mu) \leq x^\mu$. We prove this by showing that each rule reduces a problem instance of weight μ to one or more problem instances of

weight $\mu_1, \ldots, \mu_r$ where $\mu_i < \mu$ such that $f(\mu) \le \sum_{i=1}^{r} f(\mu_i) \le x^\mu$ if $f(\mu_i) \le x^{\mu_i}$ for $1 \le i \le r$. Before proceeding to the detailed analysis, we mention that the instance $RBF(G, S, \emptyset, \emptyset)$ has weight $n(1+\alpha)$, and the result will thus imply that $f(n(1 + \alpha)) \le 1.49468^{n(1+\alpha)} \le 1.8638^n$ for $\alpha = 1.5491$.

Rule 2. Vertex set $R = \emptyset$, so $t = \emptyset$ and $N(T_t)$ is defined as the empty set. As a consequence all non-colored vertices have weight $1 + \alpha$. In both new instances the weight of v is reduced from $1 + \alpha$ to zero. In the case when v is colored (Rule 3), we use v as vertex t, and due to the minimum degree two property by Rule 1 there are at least two neighbors with weights reduced by α. The two subproblem instances are **Deletion** of v: $\mu_1 \le \mu - 1 - \alpha$ and **Coloring** of v: $\mu_2 \le \mu - 1 - 3\alpha$, and we get that

$$f(\mu) \le f(\mu - 1 - \alpha) + f(\mu - 1 - 3\alpha) \le x^{\mu-1-\alpha} + x^{\mu-1-3\alpha} \le x^\mu.$$

Rule 8. In both cases the weight of vertex v is reduced from 1 to zero. If x is contained in $N(T_t)$ then it has weight 1, otherwise x has weight $1 + \alpha$. Consider first the case $x \in N(T_t)$. Since $x \in PW(v)$, we have by Observation 3 that $G[\{v\} \cup T_t \cup \{x\}]$ contains a cycle with a vertex of S. Hence either v, or x has to be deleted. If v is colored, then x is deleted by Rule 6 in order to break the cycle, and if v is deleted, then x is colored since it has to be in the witness cycle. We have for **Deletion** of v and **Coloring** of x: $\mu_1 \le \mu - 2$; for **Deletion** of x and **Coloring** of v: $\mu_2 \le \mu - 2$. Thus

$$f(\mu) \le f(\mu - 2) + f(\mu - 2) \le x^{\mu-2} + x^{\mu-2} \le x^\mu$$

If $x \notin N(T_t)$, then the weight of x is $1 + \alpha$, and we have for **Deletion** of v and **Coloring** of x: $\mu_1 \le \mu - 2 - \alpha$; for **Coloring** of v: $\mu_2 \le \mu - 1 - \alpha$, resulting in

$$f(\mu) \le f(\mu - 2 - \alpha) + f(\mu - 1 - \alpha) \le x^{\mu-2-\alpha} + x^{\mu-1-\alpha} \le x^\mu.$$

Rule 9. All vertices in $PW(v)$ have weight 1. Thus we have for **Coloring** of v and **Deletion** of $PW(v)$: $\mu_1 \le \mu - 1 - |PW(v)|$; for **Deletion** of v: $\mu_2 \le \mu - 1$. Since $|PW(v)| \ge 2$, we have that

$$f(\mu) \le f(\mu - 3) + f(\mu - 1) \le x^{\mu-3} + x^{\mu-1} \le x^\mu.$$

Rule 10. Vertex v has weight 1, x_1 has weight $1+\alpha$, and x_2 has weight 1 or $1+\alpha$. Consider first the case where x_2 has weight 1, meaning that $x_2 \in N(T_t)$. If v is colored, then x_2 is deleted by Rule 6 and Observation 3. We have for **Coloring** of v: $\mu_1 \le \mu - 2 - \alpha$; **Deletion** of v and **Coloring** of x_1: $\mu_2 \le \mu - 2 - \alpha$; and **Deletion** of v, x_1 and **Coloring** of x_2: $\mu_3 \le \mu - 3 - \alpha$. Thus

$$f(\mu) \le 2f(\mu - 2 - \alpha) + f(\mu - 3 - \alpha) \le 2x^{\mu-2-\alpha} + x^{\mu-3-\alpha} \le x^\mu.$$

If $x_2 \notin N(T_t)$, then it has weight $1 + \alpha$. We have for **Coloring** of v: $\mu_1 \le \mu - 1 - 2\alpha$; **Deletion** of v and **Coloring** of x_1: $\mu_2 \le \mu - 2 - \alpha$; and **Deletion** of v, x_1 and **Coloring** of x_2: $\mu_3 \le \mu - 3 - 2\alpha$. Therefore,

$$\begin{aligned} f(\mu) &\le f(\mu - 1 - 2\alpha) + f(\mu - 2 - \alpha) + f(\mu - 3 - 2\alpha) \\ &\le x^{\mu-1-2\alpha} + x^{\mu-2-\alpha} + x^{\mu-3-2\alpha} \le x^\mu. \end{aligned}$$

Rule 11. Let i be the number of vertices in $PW(v) \setminus N(T_t)$ and assume that $x_j \notin N(T_t)$ for $j \leq i$. The case $i = 0$ is covered by Rule 9. For $i = 1, 2$, we have for **Coloring** of v: $\mu_1 \leq \mu - 4 + i - i\alpha$; **Deletion** of v and **Coloring** of x_1: $\mu_2 \leq \mu - 2 - \alpha$; **Deletion** of v, x_1 and **Coloring** of x_2: $\mu_3 \leq \mu - 3 - i\alpha$; and **Deletion** of v, x_1, x_2 and **Coloring** of x_3: $\mu_4 \leq \mu - 4 - i\alpha$. In total

$$f(\mu) \leq f(\mu - 4 + i - i\alpha) + f(\mu - 2 - \alpha) + f(\mu - 3 - i\alpha) + f(\mu - 4 - i\alpha)$$
$$\leq x^{\mu-4+i-i\alpha} + x^{\mu-2-\alpha} + x^{\mu-3-i\alpha} + x^{\mu-4-i\alpha} \leq x^\mu.$$

For $i = 3$, we have for **Coloring** of v: $\mu_1 \leq \mu - 1 - 3\alpha$, for **Deletion** of v and **Coloring** of x_1: $\mu_2 \leq \mu - 2 - \alpha$, **Deletion** of v, x_1 and **Coloring** of x_2: $\mu_3 \leq \mu - 3 - 2\alpha$, and **Deletion** of v, x_1, x_2 and **Coloring** of x_3: $\mu_4 \leq \mu - 4 - 3\alpha$, and we get that

$$f(\mu) \leq f(\mu - 1 - 3\alpha) + f(\mu - 2 - \alpha) + f(\mu - 3 - 2\alpha) + f(\mu - 4 - 3\alpha)$$
$$\leq x^{\mu-1-3\alpha} + x^{\mu-2-\alpha} + x^{\mu-3-2\alpha} + x^{\mu-4-3\alpha} \leq x^\mu.$$

Rule 12. Let i be the number of vertices in $PW(v) \setminus N(T_t)$ and assume that $x_j \notin N(T_t)$ for $j \leq i$. The case where $i = 0$ is covered by Rule 9. For $i \geq 1$, we have for **Coloring** of v: $\mu_1 \leq \mu - (1 + |PW(v)|) + i - i\alpha$; and for **Deletion** of v: $\mu_2 \leq \mu - 1$. Since $|PW(v)| \geq 4$, we notice that the value is minimum when $i = 4$ and we get

$$f(\mu) \leq f(\mu - 1 - 4\alpha) + f(\mu - 1) \leq x^{\mu-1-4\alpha} + x^{\mu-1} \leq x^\mu.$$

We conclude the analysis of the running time of the algorithm with the following theorem, which is the main result of this paper.

Theorem 1. *The maximum number of maximal S-forests of a graph G on n vertices is at most 1.8638^n. The minimal subset feedback vertex sets of an input (G, S), where G is a graph on n vertices, can be enumerated in time $O(1.8638^n)$.*

Proof. Correctness and completeness follows from the arguments above. The number of leaves in the branching tree is at most $O^*(x^{(1+\alpha)n})$, and $1.49468^{1+0.5491}$ is strictly less than 1.8638. $\qquad\square$

5   Concluding Remarks

The running time of our algorithm matches the running time of the algorithm from [6] enumerating all minimal feedback vertex sets. Let us note that while the running times of the enumeration algorithms for both SUBSET FEEDBACK VERTEX SET and FEEDBACK VERTEX SET are the same, the number of minimal feedback vertex sets in a graph can be exponentially larger or smaller than the number of minimal subset feedback vertex sets.

We conclude with the following natural questions. Is it possible to show (see [6]) that there are graphs with 1.5926^n minimal subset feedback vertex sets? Can it be that our enumeration algorithm overestimates the maximum number

of minimal subset feedback vertex sets, and that this number is significantly smaller than 1.8638^n, say $O(1.6^n)$? Our enumeration algorithm can also be used to compute a *minimum* weight subset feedback vertex set in time $O(1.8638^n)$. It would be interesting to know whether a better running time can be obtained for the unweighted SUBSET FEEDBACK VERTEX SET problem.

References

1. Open problems from Dagstuhl seminar 09511. Dagstuhl Seminar 09511 (2009)
2. Calinescu, G.: Multiway cut. In: Encyclopedia of Algorithms, Springer, Heidelberg (2008)
3. Chen, J., Fomin, F.V., Liu, Y., Lu, S., Villanger, Y.: Improved algorithms for feedback vertex set problems. J. Comput. Syst. Sci. 74(7), 1188–1198 (2008)
4. Cygan, M., Pilipczuk, M., Pilipczuk, M., Wojtaszczyk, J.O.: Subset feedback vertex set is fixed parameter tractable. In: CoRR, abs/1004.2972 (2010)
5. Even, G., Naor, J., Zosin, L.: An 8-approximation algorithm for the subset feedback vertex set problem. SIAM J. Comput. 30(4), 1231–1252 (2000)
6. Fomin, F.V., Gaspers, S., Pyatkin, A.V., Razgon, I.: On the minimum feedback vertex set problem: Exact and enumeration algorithms. Algorithmica 52(2), 293–307 (2008)
7. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. J. ACM 56, Article 25(5) (2009)
8. Fomin, F.V., Kratsch, D.: Exact Exponential Algorithms. In: Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin (2010)
9. Fomin, F.V., Villanger, Y.: Finding induced subgraphs via minimal triangulations. In: Proceedings of STACS 2010, vol. 5, pp. 383–394. Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik (2010)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability. W. H. Freeman and Co., New York (1978)
11. Garg, N., Vazirani, V.V., Yannakakis, M.: Multiway cuts in node weighted graphs. J. Algorithms 50(1), 49–61 (2004)
12. Raman, V., Saurabh, S., Subramanian, C.R.: Faster fixed parameter tractable algorithms for finding feedback vertex sets. ACM Transactions on Algorithms 2(3), 403–415 (2006)
13. Razgon, I.: Exact computation of maximum induced forest. In: Arge, L., Freivalds, R. (eds.) SWAT 2006. LNCS, vol. 4059, pp. 160–171. Springer, Heidelberg (2006)
14. Thomassé, S.: A $4k^2$ kernel for feedback vertex set. ACM Transactions on Algorithms 6, Article 32(2) (2010)

Upper Bounds for Maximally Greedy Binary Search Trees

Kyle Fox

Department of Computer Science, University of Illinois, Urbana-Champaign
kylefox2@illinois.edu

Abstract. At SODA 2009, Demaine et al. presented a novel connection between binary search trees (BSTs) and subsets of points on the plane. This connection was independently discovered by Derryberry et al. As part of their results, Demaine et al. considered GREEDYFUTURE, an offline BST algorithm that greedily rearranges the search path to minimize the cost of future searches. They showed that GREEDYFUTURE is actually an online algorithm in their geometric view, and that there is a way to turn GREEDYFUTURE into an online BST algorithm with only a constant factor increase in total search cost. Demaine et al. conjectured this algorithm was dynamically optimal, but no upper bounds were given in their paper. We prove the first non-trivial upper bounds for the cost of search operations using GREEDYFUTURE including giving an access lemma similar to that found in Sleator and Tarjan's classic paper on splay trees.

1 Introduction

The ***dynamic optimality conjecture*** states that given a sequence of successful searches on an n-node binary search tree, the number of nodes accessed by splay trees is at most a constant times the number of node accesses and rotations performed by the optimal algorithm *for that sequence.* Sleator and Tarjan gave this conjecture in their paper on splay trees in which they showed $O(\log n)$ amortized performance as well as several other upper bounds [12]. Proving the dynamic optimality conjecture seems very difficult. There is no known polynomial time algorithm for finding an optimal BST in the offline setting where we know all searches in advance,[1] and this conjecture states that splaying is a simple solution to the online problem.

Until recently, there has been little progress made directly related to this conjecture. Wilber gave two lower bounds on the number of accesses needed for any given search sequence [16]. There are a handful of online BST algorithms that are $O(\log \log n)$-competitive [5,15,14,1], but no upper bound is known for the competitiveness of splay trees except the trivial $O(\log n)$.

---

[1] In fact, the exact optimization problem becomes NP-hard if we must access an arbitrary number of specified nodes during each search [4].

1.1   A Geometric View

Recently, Demaine et al. introduced a new way of conceptualizing BSTs using geometry [4]. A variant of this model was independently discovered by Derryberry et al. [6]. In the geometric view, BST node accesses are represented as points (x, y) where x denotes the rank of the accessed node and y represents which search accessed the node. A pair of points a and b in point set P are called *arborally satisfied* if they lie on the same horizontal or vertical line, or if the closed rectangle with corners a and b contains another point from P. The family of arborally satisfied point sets corresponds exactly to BST accesses when rotations upon accessed nodes are allowed [4].

By starting with a point set X that represents the points a BST must access to complete searches in a given search sequence S, we can describe an optimal BST algorithm for S as a minimum superset of X that is arborally satisfied [4]. This correspondence between BSTs and arborally satisfied supersets allows us to focus on algorithms strictly in the geometric view. Additionally, it is possible to show lower bounds for the BST model by showing the same for the geometric model. Demaine et al. take advantage of this fact to show a class of lower bounds that supersede the lower bounds of Wilber [16,4]. Further, it is possible to describe an *online* version of the arborally satisfied superset problem and transform solutions to this problem into online BST algorithms with at most a constant factor increase in cost [4].

1.2   Being Greedy

Demaine et al. also consider an offline BST algorithm they call GREEDYFUTURE, originally proposed by Lucas [10] and Munro [11]. GREEDYFUTURE only touches nodes on the search path, and then rearranges the search path in order to greedily minimize the time for upcoming searches.

The worst-case example known for the competitiveness of GREEDYFUTURE is a complete binary search tree with searches performed in bit-reversal order upon the leaves [11]. GREEDYFUTURE has an amortized cost of $\lg n$ per search on this sequence. The optimal algorithm rotates the leaves closer to the root and obtains an amortized cost of $\lg \frac{n}{2} + o(1)$. Given a search sequence of length m, let OPT be the total cost of the optimal algorithm for that sequence. Demaine et al. conjecture that GREEDYFUTURE is $O(1)$-competitive. In fact, the bit-reversal example suggests that the cost of GREEDYFUTURE is at most $OPT + m$; it appears optimal within an *additive term*.

Surprisingly, Demaine et al. showed that GREEDYFUTURE, an offline algorithm that uses very strong knowledge about the future, is actually an online algorithm in the geometric model [4]. Recall that online algorithms in the geometric model correspond to online algorithms in the BST model with essentially the same cost. If GREEDYFUTURE is actually an offline dynamically optimal BST algorithm as it appears to be, then there exists an *online* dynamically optimal BST algorithm.

1.3   Our Contributions

Despite the apparent optimality of the GREEDYFUTURE algorithm, nothing was known about its amortized behavior when Demaine et al. wrote their report. We provide the first theoretical evidence that GREEDYFUTURE is an optimal algorithm in the following forms:

- An access lemma similar to that used by Sleator and Tarjan for splay trees [12]. This lemma implies several upper bounds including $O(\log n)$ amortized performance.
- A sequential access theorem that states GREEDYFUTURE takes linear time to access all nodes in order starting from any arbitrary BST.

We heavily use the geometric model of Demaine et al. to prove the access lemma while focusing directly on BSTs to prove the sequential access theorem. It is our hope that these results will create further interest in studying GREEDYFUTURE as its structural properties seem well suited for further theoretical analysis (the proof of the sequential access theorem takes only a page). Additionally, the proof of the access lemma may provide additional insight into other algorithms running in the geometric model.

1.4   A Note on Independent Work

John Iacono and Mihai Pătrașcu have discovered a similar access lemma to that given here using different proof techniques from those shown below. The author learned about their work via personal correspondence with them and Erik Demaine well into performing the research contained in this report. Their results have never been published.

Additionally, the author became aware of work by Goyal and Gupta [8] after initially writing this report. They show GREEDYFUTURE has $O(\log n)$ amortized performance. This result appears in our paper as Corollary 2. As in our proof, they use the geometric model, but they do not use a potential function as we do to prove a more general access lemma.

2   Arboral and Geometric Models of BSTs

2.1   The Arboral Model

We will consider the same BST model used by Demaine et al. [4]. We consider only successful searches and not insertions or deletions. Let n and m be the number of elements in the search tree and the number of searches respectively. We assume the elements have distinct keys in $\{1, \ldots, n\}$.

Given a BST T_1, a subtree τ of T_1 containing the root, and a tree τ' on the same nodes as τ, we say T_1 can be **reconfigured** by an operation $\tau \to \tau'$ to another BST T_2 if T_2 is identical to T_1 except for τ being replaced by τ'. The cost of the reconfiguration is $|\tau| = |\tau'|$.

Given a search sequence $S = \langle s_1, s_2, \ldots, s_m \rangle$, we say a BST algorithm **executes** S by an execution $E = \langle T_0, \tau_1 \to \tau'_1, \ldots, \tau_m \to \tau'_m \rangle$ if all reconfigurations are performed on subtrees containing the root, and $s_i \in \tau_i$ for all i.

For $i = 1, 2, \ldots, m$, define T_i to be T_{i-1} with the reconfiguration $\tau_i \to \tau_i'$. The cost of execution E is $\sum_{i=1}^{m} |\tau_i|$.

As explained by Demaine et al. [4], this model is constant-factor equivalent to other reasonable BST models such as those by Wilber and Lucas [16,10].

2.2   The Geometric Model

We now turn our focus to the geometric model as given by Demaine et al. [4]. Define a **point** p to be a point in 2D with integer coordinates $(p.x, p.y)$ such that $1 \leq p.x \leq n$ and $1 \leq p.y \leq m$. Let $\Box ab$ denote the closed axis-aligned rectangle with corners a and b.

A pair of points (a, b) (or their induced rectangle $\Box ab$) is **arborally satisfied** with respect to a point set P if (1) a and b are orthogonally collinear (horizontally or vertically aligned), or (2) there is at least one point from $P \setminus \{a, b\}$ in $\Box ab$. A point set P is arborally satisfied if all pairs of points in P are arborally satisfied with respect to P. See Fig. 1 and Fig. 2.

As explained in [4], there is a one-to-one correspondence between BST executions and arborally satisfied sets of points. Let the **geometric view** of a BST execution E be the point set $P(E) = \{(x, y) | x \in \tau_y\}$. The point set $P(E)$ for any BST execution E is arborally satisfied [4]. Further, for any arborally satisfied point set X, there exists a BST execution E with $P(E) = X$ [4].

Let the **geometric view** of an access sequence S be the set of points $P(S) = \{(s_1, 1), (s_2, 2), \ldots, (s_m, m)\}$. The above facts suggest that finding an optimal BST algorithm for S is equivalent to finding a minimum cardinality arborally satisfied superset of S. Due to this equivalence with BSTs, we will refer to values in $\{1, \ldots, n\}$ as **elements**.

Naturally, we may want to use the geometric model to find dynamically optimal *online* BST algorithms. The **online arborally satisfied superset** (online ASS) problem is to design an algorithm that receives a sequence of points $\langle (s_1, 1), (s_2, 2), \ldots, (s_m, m) \rangle$ incrementally. After receiving the ith point (s_i, i), the algorithm must output a set P_i of points on the line $y = i$ such that $\{(s_1, 1), (s_2, 2), \ldots, (s_i, i)\} \cup P_1 \cup P_2 \cup \cdots \cup P_i$ is arborally satisfied. The cost of the algorithm is $m + \sum_{i=1}^{m} |P_i|$.

We say an online ASS algorithm performs a **search** at time i when it outputs the set P_i. Further, we say an online ASS algorithm **accesses** x at time i if (x, i) is included in the input set of points or in P_i. The (non-amortized) cost of a search at time i is $|P_i| + 1$.

Unfortunately, the algorithm used to create a BST execution from an arborally satisfied point set requires knowledge about points above the line $y = i$ to construct T_i [4]. We are not able to go directly from a solution to the online ASS problem to a solution for the online BST problem with exactly the same cost. However, this transformation is possible if we allow the cost of the BST algorithm to be at most a constant multiple of the ASS algorithm's cost [4].
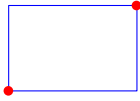
Fig. 1. An unsatisfied pair of points. The closed axis-aligned rectangle with corners defined by the pair is shown.
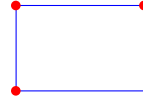

Fig. 2. An arborally satisfied superset of the same pair of points

3   GreedyFuture

We now turn our focus to describing the GREEDYFUTURE algorithm in more detail. Let $S = \langle s_1, \ldots, s_m \rangle$ be an arbitrary search sequence of length m. After every search, GREEDYFUTURE will rearrange the search path to minimize the cost of future searches.

More precisely, consider the ith search for the given sequence S. If $i = m$, then GREEDYFUTURE does not rearrange the search path. Otherwise, if s_{i+1} lies on the search path τ_i, GREEDYFUTURE makes s_{i+1} the root of τ_i'. If s_{i+1} does not lie along the search path, then GREEDYFUTURE makes the predecessor and successor of s_{i+1} *within* τ_i the root and root's right child of τ_i' (if the successor (predecessor) does not exist, then GREEDYFUTURE makes the predecessor (successor) the root and does not assign a right (left) child within τ_i'.) Now that it has fixed one or two nodes x_ℓ and x_r with $x_\ell < x_r$, GREEDYFUTURE recursively sets the remaining nodes of τ_i *less* than x_ℓ using the subsequence of $\langle s_{i+1}, \ldots, s_m \rangle$ containing nodes less than x_ℓ. It then sets the nodes of τ_i greater than x_r using the subsequence of $\langle s_{i+1}, \ldots, s_m \rangle$ containing nodes greater than x_r.

Taking a cue from Demaine et al., we will call the online geometric model of the algorithm GREEDYASS. Let $X = P(S)$ for some BST access sequence S. At each time i, GREEDYASS simply outputs the minimal set of points at $y = i$ needed to satisfy X up to $y \leq i$.

We note that the set of points needed to satisfy X up to $y \leq i$ is uniquely defined. For each unsatisfied rectangle formed with (s_i, i) in one corner, we add the other corner at $y = i$. We can also define GREEDYASS as an algorithm that sweeps right and left from the search node, accessing nodes that have increasingly greater last access times. See Fig. 3.

GREEDYASS, the online geometric view of GREEDYFUTURE, greatly reduces the complexity of predicting GREEDYFUTURE's behavior. By focusing our attention on this geometric algorithm, we proceed to prove several upper bounds on both algorithms' performance in the following section.

4   An Access Lemma and Its Corollaries

In their paper on splay trees, Sleator and Tarjan prove the *access lemma*, a very general expression detailing the amortized cost of a splay (and therefore search) operation [12]. They use this lemma to prove several upper bounds, including the entropy bound, the static finger bound, and the working set bound.
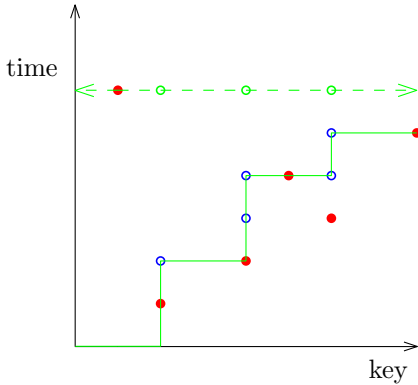
Fig. 3. (Left) A sample execution of GREEDYASS. Search elements are represented as solid disks. For the latest search, GREEDYASS sweeps right, placing points when the greatest last access time seen increases. The staircase represents these increasing last access times.
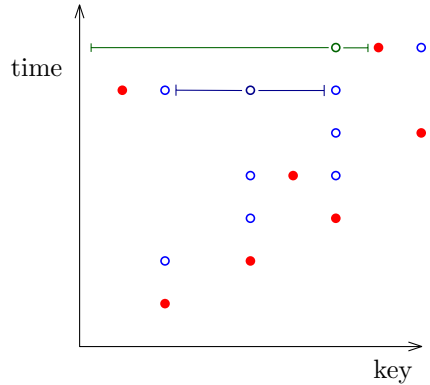
Fig. 4. (Right) Later in the same execution of GREEDYASS. The most recent neighborhoods for two of the elements are represented as line segments surrounding those elements. Observe that adding another search for anything within a neighborhood will result in accessing the corresponding element for that neighborhood.

Wang et al. prove a similar lemma for their multi-splay tree data structure to show $O(\log \log n)$-competitiveness and $O(\log n)$ amortized performance, and the version of the lemma given in Wang's Ph.D. thesis is used to prove the other distribution sensitive upper bounds listed above [15,14]. In this section, we provide a similar lemma for GREEDYASS and discuss its consequences.

4.1 Potentials and Neighborhoods

Fix a BST access sequence S and let $X = P(S)$. We consider the execution of GREEDYASS on X. Let $\rho(x, i)$ be the last access of x at or before time i. Formally, $\rho(x, i)$ is the y coordinate of the highest point on the closed ray from (x, i) to $(x, -\infty)$.

Let a be the greatest positive integer smaller than x such that $\rho(a, i) \geq \rho(x, i)$ (or let $a = 0$ if no such integer exists). The **left neighborhood** of x at time i is $\{a + 1, a + 2, \ldots, x - 1\}$ and denoted $\Gamma_\ell(x, i)$. The **right neighborhood** of x at time i is defined similarly and denoted $\Gamma_r(x, i)$. Finally, the **inclusive neighborhood** of x at time i is $\Gamma(x, i) = \Gamma_\ell(x, i) \cup \Gamma_r(x, i) \cup \{x\}$.

The inclusive neighborhood of x at time i contains precisely those keys whose appearance as s_{i+1} would prompt GREEDYASS to access x at time $i + 1$. Intuitively, the inclusive neighborhood is similar to a node's subtree in the arboral model. See Fig. 4.

Assign to each element $x \in \{1, \ldots, n\}$ a positive real weight $w(x)$. The **size** of x at time i is $\sigma(x, i) = \sum_{e \in \Gamma(x,i)} w(e)$. The **rank** of x at time i

is $r(x, i) = \lfloor \lg \sigma(x, i) \rfloor$. Finally, define a potential function $\Phi(i) = \sum_{x \in [n]} r(x, i)$ and let the **_amortized cost_** of a search at time i be $1 + |P_i| + \Phi(i) - \Phi(i - 1)$.

Lemma 1 (Access Lemma). *Let $W = \sum_{x \in [n]} w(x)$. The amortized cost of a search at time i is at most $5 + 6 \lfloor \lg W \rfloor - 6r(s_i, i - 1)$.*

4.2   Immediate Consequences

Before we proceed to prove Lemma 1, we will show several of its consequences. Recall that the equivalence between the arboral and geometric models mean these corollaries apply to both GREEDYASS and GREEDYFUTURE. The proofs of these corollaries mirror the proofs by Sleator and Tarjan for splay trees [12].

Corollary 2 (Balance Theorem). *The total cost of searching is $O((m + n) \times \log n)$.*

Corollary 3 (Static Optimality Theorem). *Let $t(x)$ be the number of times x appears in the search sequence S. If every element is searched at least once, the total cost of searching is $O\left(m + \sum_{x=1}^{n} t(x) \log (m/t(x))\right)$.*

Corollary 4 (Static Finger Theorem). *Fix some element f. The total cost of searching is $O(m + \sum_{i=1}^{m} \log(|s_i - f| + 1))$.*

Corollary 5 (Working Set Theorem). *Let $d(i)$ be the number of distinct elements in the search sequence S before s_i and since the last instance of s_i. If there are no earlier instances of s_i, then let $d(i) = i - 1$. The total cost of searching is $O(m + \sum_{i=1}^{m} \log(d(i) + 1))$.*

Note that Corollary 5 implies other upper bounds on GREEDYFUTURE's performance such as key-independent optimality [9].

4.3   Telescoping Rank Changes

We proceed to prove Lemma 1. First we observe the following.

Lemma 6. *Let x be any element* not *accessed during search i. Then we have $\Gamma(x, i - 1) = \Gamma(x, i)$.*

Proof: Assume without loss of generality that $x > s_i$. Let x_ℓ be the greatest element in $\{s_i, s_i + 1, \ldots, x - 1\}$ such that $\rho(x_\ell, i - 1) \geq \rho(x, i - 1)$. Element x_ℓ must exist, because GREEDYASS does not access x at time i. No elements in $\{x_\ell + 1, \ldots, x - 1\}$ are accessed at time i since they have smaller last access time than x_ℓ, so $\Gamma_\ell(x, i - 1) = \Gamma_\ell(x, i)$. Likewise, no elements in $\Gamma_r(x, i - 1)$ are accessed at time i since they have smaller last access time than x. The inclusive neighborhood of x (as well as its size and rank) remains unchanged by the search. □

Consider a search at time i. Lemma 6 immediately implies the amortized cost of the search is equal to

$$\sum_{x \in P_i \cup \{s_i\}} (1 + r(x,i) - r(x,i-1)). \qquad (1)$$

Suppose we access an element $x \neq s_i$. Assume $x > s_i$ without loss of generality. If it exists, let x_r be the least accessed element greater than x. We call x_r the **successor** of x. Observe that $\Gamma(x,i)$ contains a subset of the elements in $\{s_i + 1, \ldots, x_r - 1\}$ while $\Gamma(x_r, i-1)$ contains a superset of the elements in $\{s_i, \ldots, x_r\}$. This fact implies $\Gamma(x,i) \subset \Gamma(x_r, i-1)$ which in turn implies

$$\sigma(x,i) < \sigma(x_r, i-1) \text{ and } r(x,i) \leq r(x_r, i-1). \qquad (2)$$

If the second inequality is strict, then

$$1 + r(x,i) - r(x,i-1) \leq r(x_r, i-1) - r(x, i-1). \qquad (3)$$

Otherwise,

$$1 + r(x,i) - r(x,i-1) = 1 + r(x_r, i-1) - r(x, i-1). \qquad (4)$$

Call an accessed element $x > s_i$ a **stubborn element** if x has a successor x_r and $r(x,i) = r(x_r, i-1)$. From (1), (3), and (4) above, the amortized cost of accessing elements greater than s_i forms a telescoping sum and we derive the following lemma.

Lemma 7. *Let α be the number of elements greater than s_i that are stubborn and let $e_{r\ell}$ and e_{rr} be the least and greatest elements greater than s_i to be accessed. The amortized cost of accessing elements greater than s_i is*

$$1 + \alpha + r(e_{rr}, i) - r(e_{r\ell}, i-1).$$

4.4   Counting Stubborn Elements

The biggest technical challenge remaining is to upper bound the number of stubborn elements α. We have the following lemma.

Lemma 8. *The number of accessed elements greater than s_i which are stubborn is at most*

$$1 + 2\lfloor \lg W \rfloor - 2r(s_i, i-1)$$

Proof: Consider any stubborn element $x > s_i$ and its successor x_r. Let the **left size** of x at time i be $\sigma_\ell(x,i) = \sum_{e \in \Gamma_\ell(x,i)} w(e)$. Further, let the **left rank** of x at time i be $r_\ell(x,i) = \lfloor \lg(\sigma_\ell(x,i)) \rfloor$. By the definitions of stubborn elements and left sizes we see

$$\sigma(x,i) > \frac{1}{2}\sigma(x_r, i-1) > \frac{1}{2}\sigma_\ell(x_r, i-1). \qquad (5)$$

We note that for any accessed element v (stubborn or not) with $s_i < v < x$ we have

$$\sigma_\ell(v, i-1) < \frac{1}{2}\sigma_\ell(x_r, i-1) \tag{6}$$

by (5) since every element of $\Gamma_\ell(v, i-1)$ is in $\Gamma_\ell(x_r, i-1)$, but none of these elements are in $\Gamma(x, i)$ since the left neighborhood of x at time i cannot extend past v. Further,

$$\sigma_\ell(x, i-1) \geq \sigma(s_i, i-1) \tag{7}$$

since all weights are positive and every element in $\Gamma(s_i, i-1)$ is also in $\Gamma_\ell(x, i-1)$.

Let $z > s_i$ be the greatest stubborn element, and let z_r be its successor. We will inductively argue the number of stubborn elements is at most

$$1 + 2r_\ell(z_r, i-1) - 2r(s_i, i-1)$$

which is a stronger statement than that given in the lemma. The argument can be divided into two cases.

1. Suppose $\sigma_\ell(z_r, i-1) < 2\sigma(s_i, i-1)$. For any stubborn element v between s_i and z we have

$$\sigma_\ell(v, i-1) < \sigma(s_i, i-1)$$

by (6). There can be no such element v by (7), making z the only stubborn element. The total number of stubborn elements is

$$1 \leq 1 + 2r_\ell(z, i-1) - 2r(s_i, i-1)$$
$$\leq 1 + 2r_\ell(z_r, i-1) - 2r(s_i, i-1)$$

by (7) and the definition of left rank.

2. Now suppose $\sigma_\ell(z_r, i-1) \geq 2\sigma(s_i, i-1)$. Consider any stubborn element v with successor v_r such that $s_i < v < v_r < z$. Note that if a stubborn element exists with z as its successor, v cannot be this stubborn element. We have

$$\sigma_\ell(v_r, i-1) < \frac{1}{2}\sigma_\ell(z_r, i-1)$$

by (6). By induction on the left sizes of stubborn element successors greater than s_i, the successors of at most

$$1 + 2\left\lceil \lg\left(\frac{1}{2}\sigma_\ell(z_r, i-1)\right)\right\rceil - 2r(s_i, i-1)$$

stubborn elements can have this smaller left size. Counting z and the one other stubborn element that may exist with z as its successor, the total number of stubborn elements is at most

$$3 + 2\left\lceil \lg\left(\frac{1}{2}\sigma_\ell(z_r, i-1)\right)\right\rceil - 2r(s_i, i-1) = 1 + 2r_\ell(z_r, i-1) - 2r(s_i, i-1).$$

$\square$

4.5   Finishing the Proof

We now conclude the proof of Lemma 1.

Proof: By Lemma 6, the amortized cost of accessing s_i alone is

$$1 + r(s_i, i) - r(s_i, i - 1) \le 5 + 6 \lfloor \lg W \rfloor - 6r(s_i, i - 1)$$

so the lemma holds in this case.

If all other accessed elements are greater than s_i, let $e_{r\ell}$ and e_{rr} be the least and greatest of these elements. Observe $r(e_{r\ell}, i - 1) \ge r(s_i, i)$ and $r(e_{rr}, i) \le \lfloor \lg W \rfloor$. By Lemma 7 and Lemma 8, the total amortized cost of accessing elements is at most

$$
\begin{aligned}
3 + r(s_i, i) &- 3r(s_i, i - 1) + 2 \lfloor \lg W \rfloor + r(e_{rr}, i) - r(e_{r\ell}, i - 1) \\
&\le 3 + 3 \lfloor \lg W \rfloor - 3r(s_i, i - 1) \\
&\le 5 + 6 \lfloor \lg W \rfloor - 6r(s_i, i - 1)
\end{aligned}
$$

so the lemma holds in this case. It also holds in the symmetric case when all accessed elements are smaller than s_i.

Finally, consider the case when there are accessed elements both greater than and less than s_i. Let $e_{\ell\ell}$ and $e_{\ell r}$ be the least and greatest elements *less than* s_i. Observe $r(e_{\ell\ell}, i) \le \lfloor \lg W \rfloor$ and $r(e_{\ell r}, i - 1) \ge r(s_i, i - 1)$. By two applications of Lemma 7 and Lemma 8, the total amortized cost of the search is at most

$$
\begin{aligned}
5 + r(s_i, i) &- 5r(s_i, i - 1) + 4 \lfloor \lg W \rfloor + r(e_{rr}, i) - r(e_{r\ell}, i - 1) \\
&+ r(e_{\ell\ell}, i) - r(e_{\ell r}, i - 1) \\
&\le 5 + 6 \lfloor \lg W \rfloor - 6r(s_i, i - 1)
\end{aligned}
$$

□

5   A Sequential Access Theorem

The working set bound proven above shows that GREEDYFUTURE has good *temporal locality*. Accessing an element shortly after its last access guarantees a small amortized search time. Sleator and Tarjan conjectured that their splay trees also demonstrate good spatial locality properties in the form of the dynamic finger conjecture [12]. This conjecture was verified by Cole, et al. [3,2].

One special case of the dynamic finger theorem considered by Tarjan and others was the sequential access theorem [13,7,15,14]. We give a straightforward proof of the sequential access theorem when applied to GREEDYFUTURE. Note that this theorem requires focusing on an arbitrary fixed BST, so we do not use the geometric model in the proof.

Theorem 9 (Sequential Access Theorem). *Let $S = \langle 1, 2, \ldots, n \rangle$. Starting with an arbitrary BST T_0, the cost of running GREEDYFUTURE on search sequence S is $O(n)$.*

Let $T_0, T_1, \ldots, T_n$ be the sequence of search trees configured by GREEDYFUTURE. We make the following observations:

Lemma 10. *For all $i > 1$, either node i is the root of T_{i-1} or $i - 1$ is the root and i is the leftmost node of the root's right subtree.*

Proof: If i was accessed during the $i - 1$st search, then i is the root of T_{i-1}. Otherwise, $i - 1$ is the predecessor node of i on the search path. Therefore, $i - 1$ is the root of T_{i-1} and i is the leftmost node of the root's right subtree.   □

Lemma 11. *Node x is accessed at most once in any position other than the root or the root's right child.*

Proof: Consider node x and search i. Node x cannot be accessed if $x < i - 1$ according to Lemma 10. If x lies on the search path and $x \leq i + 1$ then either x becomes the root or x moves into the root's left subtree so that i or $i + 1$ can become the root.

Now suppose x lies along the search path and $x > i + 1$. Let x_ℓ be the least node strictly smaller than x that does not become the root. If x_ℓ does not exist, then x becomes the root's right child as either x is the successor of $i + 1$ on the search path, node $i + 1$ is on the search path and $x = i + 2$, or node $i + 1$ is on the search path and x is the successor of $i + 2$ on the search path. If x_ℓ does exist, then x_ℓ becomes the root's right child for one of the reasons listed above and x becomes a right descendent of x_ℓ.

Node x cannot be moved to the left subtree of the root's right child in all the cases above. Lemma 10 therefore implies x is accessed in the root's left subtree on the first search, x is accessed *once* in the left subtree of the root's right child, or x is never accessed anywhere other than as the root or root's right child.   □

We now conclude the proof of Theorem 9.

Proof: The cost of the first search is at most n. The costs of all subsequent searches is at most $2(n - 1) + n$ according to Lemma 11; at most $2(n - 1)$ node accesses occur at the root or root's right child, and at most n nodes are accessed exactly once in a position other than the root or the root's right child. The total cost of all searches is at most $4n - 2$.   □

6   Closing Remarks

The ultimate goal of this line of research is to prove GREEDYFUTURE or splay trees optimal, but showing other upper bounds may prove interesting. In particular, it would be interesting to see if some difficult to prove splay tree properties such as the dynamic finger bound have concise proofs when applied to GREEDYFUTURE. Another direction is to explore how GREEDYFUTURE may be modified to support insertions and deletions while still maintaining its small search cost.

References

1. Bose, P., Douïeb, K., Dujmović, V., Fagerberg, R.: An $O(\log \log n)$-competitive binary search tree with optimal worst-case access times. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 38–49. Springer, Heidelberg (2010)
2. Cole, R.: On the dynamic finger conjecture for splay trees. Part II: The proof. SIAM J. Comput. 30, 44–85 (2000)
3. Cole, R., Mishra, B., Schmidt, J., Siegel, A.: On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$-block sequences. SIAM J. Comput. 30, 1–43 (2000)
4. Demaine, E.D., Harmon, D., Iacono, J., Kane, D., Pătraşcu, M.: The geometry of binary search trees. In: Proc. 20th ACM/SIAM Symposium on Discrete Algorithms, pp. 496–505 (2009)
5. Demaine, E.D., Harmon, D., Iacono, J., Pătraşcu, M.: Dynamic optimality–almost. SIAM J. Comput. 37(1), 240–251 (2007)
6. Derryberry, J., Sleator, D.D., Wang, C.C.: A lower bound framework for binary search trees with rotations. Tech. Rep. CMU-CS-05-187. Carnegie Mellon University (2005)
7. Elmasry, A.: On the sequential access theorem and deque conjecture for splay trees. Theoretical Computer Science 314(3), 459–466 (2004)
8. Goyal, N., Gupta, M.: On dynamic optimality for binary search trees (2011), http://arxiv.org/abs/1102.4523
9. Iacono, J.: Key independent optimality. Algorithmica 42, 3–10 (2005)
10. Lucas, J.M.: Canonical forms for competitive binary search tree algorithms. Tech. Rep. DCS-TR-250. Rutgers University (1988)
11. Munro, J.I.: On the competitiveness of linear search. In: Paterson, M. (ed.) ESA 2000. LNCS, vol. 1879, pp. 338–345. Springer, Heidelberg (2000)
12. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the Association for Computing Machinery 32(3), 652–686 (1985)
13. Tarjan, R.E.: Sequential access in splay trees takes linear time. Combinatorica 5, 367–378 (1985)
14. Wang, C.C.: Multi-Splay Trees. Ph.D. thesis. Carnegie Mellon University (2006)
15. Wang, C.C., Derryberry, J., Sleator, D.D.: $O(\log \log n)$-competitive binary search trees. In: Proc. 17th Ann. ACM-SIAM Symp. Discrete Algorithms, pp. 374–383 (2006)
16. Wilber, R.E.: Lower bounds for accessing binary search trees with rotations. SIAM J. Comput. 18(1), 56–67 (1989)

On the Matter of Dynamic Optimality in an Extended Model for Tree Access Operations

Michael L. Fredman

Rutgers University, New Brunswick
`fredman@cs.rutgers.edu`

Abstract. The model of node access operations taking place in binary search trees subject to rotations is extended to unordered binary trees, as motivated by consideration of certain self-adjusting priority queues. Rotations in this extended model can be preceded by sibling subtree swaps. Whereas the Wilber lower bound for off-line computation extends to this model of computation – a precondition for the possibility of dynamic optimality – the goal of dynamic optimality is nonetheless demonstrated to be unattainable in this model.

1   Introduction

A model has recently been introduced (this author [1]) for node access operations in binary trees that generalizes the standard search tree model with rotations. As with search trees, distance from the root determines the immediate cost of an access request, and rotation steps can be utilized to restructure a tree in order to reduce the cost of subsequent access requests (with rotation count contributing to the total implementation cost). The new model departs from that of search trees by relaxing the search tree ordering constraint. This added freedom allows for an expanded repertoire of rotation operations; the two subtrees of the node promoted to the parent position may first be swapped, defining a non-standard rotation. Figure 1 illustrates the notion. We refer to this model of computation as the *unordered tree* model.
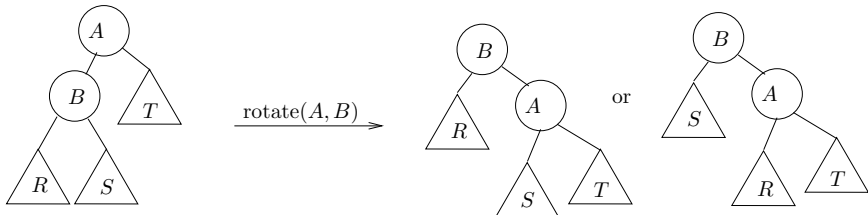

Fig. 1. Standard and non-standard rotations

The unordered tree model naturally arises as a means for viewing a class of self-adjusting priority queues such as the pairing heap [2]. The pertinent data structures can be reformulated as tournament trees: a set of data originates in

the leaves of a tree; each internal node stores the minimum value stored among its two children, so that the set minimum is found in the tree root. A replacement operation vacates the minimum value along the path from the root to its originating leaf, places a new (replacement) value in this leaf, and reestablishes the root value by recomputing the values stored in the nodes on the path previously vacated. The cost of the operation is the length of the path. The replacement operation, therefore, can be reinterpreted simply as a leaf access operation. Replacement operations provide a vehicle for merging multiple sorted lists, with each list supplying replacement values to a unique specified leaf. Tree restructuring to improve performance is an option. For this type of application, the search tree constraint, preserving left-to-right node ordering, is clearly unnecessary.

One form of tree restructuring available in this model of node access consists of a purely "zig-zig" style splay operation [2,3], treating any path as though all of its edges have consistent orientation (or equivalently, employing non-standard rotation steps when this is not the case). In fact, the two-pass pairing heap effectively functions this way when implementing alternating delete-min and insertion operations.

With its enhanced repertoire of rotation operations it is entirely plausible that the unordered tree model of computation is more powerful (offering greater efficiencies) than that of search trees, and indeed this is the case for off-line computations [1]. On the other hand, Wilber's lower bound for search trees [4], that for randomly generated access request sequences, $\log n$ access cost per item is required, even for off-line computations, extends to the unordered tree model [1]. If the Wilber lower bound did not extend, that would definitively establish an efficiency separation between on-line and off-line computations in the unordered tree model. Thus, the on-line versus off-line situations for search trees and unordered trees seem analogous, and one could, for instance, plausibly conjecture that zig-zig access operations provide dynamic optimality in the unordered tree model. Dynamic optimality asserts, with respect to a given on-line method for processing access requests, that the method processes any given request sequence as efficiently (to within a constant factor) as the optimal off-line processing of the given sequence. Originally formulated in the context of binary search trees [3], the *dynamic optimality conjecture* asserts that use of splay operations to implement access requests provides dynamic optimality.

Our main result, however, shows that dynamic optimality in the unordered tree model is not possible.

2   Technical Development

Definition. (*swap-equivalent* trees) Two trees are considered *swap-equivalent* provided that one can be obtained from the other by a sequence of swaps of sibling subtrees.

Observation A. Given two swap-equivalent trees T_1 and T_2 and a rotation ρ acting on T_1, there exists a rotation ρ' acting on T_2 such that the respective resulting trees are equivalent.

A consequence of this observation is that for off-line processing, an access request sequence, implemented commencing from T_1, can be accomplished at the same cost, commencing from T_2. In the sequel sibling subtree swaps will be extensively utilized (but not explicitly acknowledged) for descriptive convenience.

Following standard practice as set forth by Wilber [4], we impose the convention that any access request be accomplished by bringing the requested node to the root position (at some point during the implementation of the request), noting that this requirement reduces efficiency by at most a constant factor [4].

We make use of the following theorem [1], presented here in a slightly improved form (proof given in the Appendix).

Theorem 1. *Given an initial tree T upon which access request sequences are to be implemented, there exists a binary encoding of these sequences such that the encoding $e(\sigma)$ of the request sequence σ satisfies*

$$length(e(\sigma)) \leq length(\sigma) + 4 \cdot \mu(T, \sigma),$$

where $\mu(T, \sigma)$ is the minimum number of rotations that suffice to implement the request sequence off-line, starting with the tree T.

Additionally needed are the following constructs, terminology, and facts, obtained from the Appendix. Given any implementation of an access request sequence, there exists a rearrangement of its rotations that contains a prefix, referred to as a *root-sequence*, that brings the same nodes to the root position of the tree – and in the same order – as the original sequence of rotations. The root-sequence can be augmented with navigation steps yielding an *augmented* root-sequence, satisfying the following **properties** (Appendix: lemma 4 and subsequent discussion).

1. Navigation proceeds from the root position.
2. A single step of navigation transfers the position of navigation to a specified child of the node of current position. (We remark that the node at which navigation is positioned does not change when a rotation takes place, thereby providing a means by which the position of navigation can move toward the root position.) Any node at which navigation has been positioned is considered as having been *visited*.
3. A rotation of an edge can only take place after both of the nodes joined by the edge have been visited (Appendix: Observation B).
4. The number of navigation steps inserted into a root-sequence, to form an augmented root-sequence, is the same as the number of rotations it contains.

An *augmented root-sequence implementation* of an access request sequence refers to a sequence of rotations having the form of a root-sequence, that implements the given access request sequence, and which is also augmented with navigation steps as described above. We readily note that one can always be obtained without increasing rotation count.

2.1   Outline

We construct an adversary that generates access request sequences for a tree of size n, designed to elude efficient on-line implementation while admitting efficient off-line implementation. The construction is outlined as follows.

1. We first prove the existence of a permutation π over the first m integers, $m = \lceil \sqrt{n} \rceil$, such that for any arbitrary tree T of size n and any sequence of m distinct nodes $\sigma = x_1, \cdots, x_m$ in T, one of the request sequences, σ or $\pi(\sigma) = x_{\pi(1)}, \cdots, x_{\pi(m)}$, requires $\Omega(m \log m)$ rotations to implement off-line.
2. The adversary constructs a request sequence ζ that consists of a concatenation of rounds – as many as desired – each of length approximately n. The rounds are constructed so that each is independently implementable off-line with a linear number of rotations. Let T_1 and T_2 be specified trees on a common set of nodes. We proceed to define a (T_1, T_2)-round. Subdivide the symmetric order sequence $\tau = a_1, \cdots, a_n$ of the nodes that comprise T_1 into contiguous blocks of size m, omitting the last up to $m-1$ nodes, should n not be a multiple of m, and let τ_i, $1 \le i \le n/m$, be the ith block of τ. A (T_1, T_2)-round γ consists of a concatenation of $\lfloor n/m \rfloor$ request blocks, $\beta_1 \cdots \beta_{\lfloor n/m \rfloor}$, each β_i consisting of m access requests, with these blocks defined inductively: Given $i \ge 0$, and the previously defined blocks β_j, $1 \le j \le i$, let T_2^i be the tree generated by the on-line processing of the concatenation, $\beta_1 \cdots \beta_i$, initiated from the tree T_2 ($T_2^0 = T_2$). As follows from the preceding step of this outline, one of the sequences, τ_{i+1} or $\pi(\tau_{i+1})$, inherently requires $\Omega(m \log m)$ rotations to implement, initiated from T_2^i, and the adversary sets β_{i+1} to be that choice, τ_{i+1} or $\pi(\tau_{i+1})$, requiring that many rotations. Now a round of ζ consists of a (T_1, T_2)-round, where T_1 is the tree generated by a specified off-line implementation of the preceding rounds, and T_2 is the tree generated by the on-line implementation of the preceding rounds.
3. Given its construction, we immediately find, upon summing over the blocks of a given round, that its on-line implementation requires $\Omega(n \log n)$ rotations. It will be the case, however, that any given round can be implemented off-line with $O(n)$ rotations.

2.2   Existence of the Permutation π

Lemma 1. *Given an augmented root-sequence implementation σ of an access request sequence commencing from a tree T_o, the set of nodes that are visited comprise an embedded tree E_o within the tree T_o, that includes the root of T_o. Moreover, the implementation of the request sequence can be regarded as commencing from E_o, and implemented within this tree as it evolves.*

Proof: We argue by induction on j that with respect to the first j steps (counting both rotations and navigation steps) of σ, that (a) the nodes visited during during these steps induce a connected subgraph of T_o that includes its root, (b) any unvisited node that is a child of a visited node in the tree as it evolves, is also a child of some visited node (possibly different) in T_o, and (c) the visited nodes

induce a connected subgraph of the tree as it evolves (that includes its root). When j is 0, the visited subgraph consists of the root of T_o, and the assertion is immediate. Assume $j > 0$, and the claim holds after $j - 1$ steps of σ. If the next step is a navigation step to a child x of a visited node, then there is nothing to prove if x has been previously visited. Assuming otherwise, (c) holds trivially, and moreover, (b) of the induction hypothesis implies that x is a child (in T_o) of some previously visited node, so that (a) continues to hold. Furthermore, the subtree rooted at x will be as found in T_o: no node in this subtree could have already been visited, as this would be contrary to (c), and therefore no rotation at an edge within this subtree (or incident upon x) could have taken place since both nodes joined by the edge must be visited before such as rotation can take place, as asserted by property 3 (following the statement of theorem 1). Therefore the children of x will be as found in T_o, so that (b) continues to hold after this jth step. Now in the case that the jth step is a rotation, the set of visited nodes is unaltered, and moreover, the collective set of the unvisited children of visited nodes is the same as before the rotation (again following from property 3). Thus, (a) and (b) continue to hold, as does (c). The final sentence in the statement of the lemma follows from property 3. □

Lemma 2. *There exists a fixed permutation π over the first m integers that satisfies the following. Given an arbitrary sequence of m distinct nodes $x_1, \cdots, x_m$, belonging to an arbitrary tree T, at least one of the request sequences, $\sigma = x_1, \cdots, x_m$ or $\pi(\sigma) = x_{\pi(1)}, \cdots, x_{\pi(m)}$, requires $(m \log_2 m)/20$ rotations to implement (even) off-line, commencing from the tree T.*

Proof: First we show that the claimed statement holds provided that we impose the restriction that the size of T is bounded by $(m \log_2 m)/10$. Then we show that this restriction can be removed.

Assume for now that the trees acted upon by the request sequences have size bounded by $(m \log_2 m)/10$. We proceed to demonstrate the claimed existence of π as follows. Let $\mathcal{C}$ denote the set of configurations (α, s), where α is an unlabeled tree of size bounded by $(m \log_2 m)/10$, and s is a subset of m nodes in α. (We refer to a node in an unlabeled tree by its symmetric order position within the tree.) The size of $\mathcal{C}$ is bounded by $m^{m/5} \cdot (\log_2 m)^m$, as follows from the fact that there are fewer than 4^h unlabeled binary trees of size h (and using the inequality $\binom{a}{b} \leq (ea/b)^b$). Now map an arbitrary pair (r, g), where g is a tree of size bounded by $(m \log_2 m)/10$ and r is a request sequence of length m, that references distinct nodes within g, to a triple (δ, α, s), where $(\alpha, s) \in \mathcal{C}$, by choosing α to be g with its node labels removed, choosing s to be the subset of nodes in α corresponding to those that r references in g, and choosing δ to be the particular permutation of the nodes in s that the order of the requests in r induces. And let $\langle \delta, s \rangle$ denote the request sequence consisting of requests to the nodes in s, as ordered by δ. By extending the isomorphism between the trees g and α to implementations of the corresponding request sequences, r and $\langle \delta, s \rangle$, initiated from the respective trees, g and α, we immediately conclude that the optimal efficiency with which r can be implemented is uniquely determined by (δ, α, s), given that $(r, g) \rightarrow (\delta, \alpha, s)$. Accordingly, we are justified in confining our analysis to the setting of

request sequences $\langle \delta, s \rangle$ being implemented commencing from unlabeled trees α, with s a specified subset of nodes in α. Holding (α, s) fixed, we claim that theorem 1 then implies that the number of permutations δ for which $\langle \delta, s \rangle$ is implementable with fewer than $(m \log_2 m)/20$ rotations, commencing from α is bounded by $m^{m/4}$. Namely, from the bound $\ell = (m \log_2 m)/20$ on the number of rotations available for implementing $\langle \delta, s \rangle$, theorem 1 implies a bound of $4\ell + m \leq (m \log_2 m)/4$ on the length of the derived binary encodings of these access request sequences, which in turn imposes the bound $m^{m/4}$ on the corresponding number of permutations δ. Allowing (α, s) to vary, it follows that the total number κ of triples (δ, α, s), with $(\alpha, s) \in \mathcal{C}$, such that $\langle \delta, s \rangle$ is implementable with fewer than $(m \log_2 m)/20$ rotations, commencing from α, is bounded by $\text{size}(\mathcal{C}) \cdot m^{m/4}$, so that we have $\kappa \leq m^{m/5 + m/4} \cdot (\log_2 m)^m$.

Now we require a single permutation π such that whenever $\langle \delta, s \rangle$ is implementable with fewer than $(m \log_2 m)/20$ rotations, commencing from α, $\pi(\langle \delta, s \rangle)$ requires at least $(m \log m)/20$ rotations to implement. For each instance (δ, α, s) contributing to κ, at most $m^{m/4}$ candidates for π are eliminated, as again follows by application of theorem 1. (Those not eliminated are such that $\pi(\langle \delta, s \rangle)$ requires at least $(m \log m)/20$ rotations to implement, commencing from α.) Summing over the instances (δ, α, s) enumerated by κ, we conclude that at most $\kappa \cdot m^{m/4} < m^{7m/10} \cdot (\log_2 m)^m$ candidates for π are eliminated. As there are $m!$ potential candidates to choose from, a suitable choice exists (for m sufficiently large).

We show next that the size restriction on T, assumed above, can be removed. Assume otherwise, and let σ be such that both σ and $\pi(\sigma)$ have implementations, each involving fewer than $(m \log_2 m)/20$ rotations, in a tree T of size exceeding $(m \log_2 m)/10$, where π satisfies the lemma provided that T is of size bounded by $(m \log_2 m)/10$. Considering first σ, we identify an embedded tree T' in T (that includes the root of T), in which the implementation of σ can be confined, such that the size of T' is bounded by $(m \log_2 m)/20$. T' is obtained by application of lemma 1 with respect to an augmented root-sequence implementation of σ satisfying the requisite rotation bound. Since the number of navigation steps is bounded by the number of rotations in an augmented root-sequence (property 4, above), the size of the embedded tree defined by visited nodes in this particular instance is bounded by $(m \log_2 m)/20$. Similarly, there is an embedded tree T'', similarly bounded in size, in which the implementation of $\pi(\sigma)$ can be confined. As both T' and T'' contain the root of T, their union gives an embedded tree of size bounded by $(m \log_2 m)/10$, in which σ and $\pi(\sigma)$ can both be implemented, each with fewer than $(m \log_2 m)/20$ rotations. But this contradicts the choice of π. $\square$

2.3   Linear Off-Line Implementation

Lemma 3. *Any round of the request sequence ζ can be implemented off-line with $O(n)$ rotations.*

Proof: To facilitate this construction, the following library of transformation macros will be referenced.

Macro Library

Disclaimer: With respect to the transformation macros described below, the implementation given for a particular macro may result in a tree that is only swap-equivalent to what is described (which by Observation A suffices for the purpose of implementing access requests).

M-1 The effect of a rotation can be reversed by just one rotation (always standard).

M-2 If B is a child of A, then the other child pointer of A and a child pointer of B can be exchanged. This can be accomplished by rotating $A \to B$ and then rotating $B \to A$, with the standard–non-standard options appropriately chosen.

M-3 The ordering of two nodes on a path can be exchanged, each bringing along its own attached off-path subtree – accomplished by applying the (standard or non-standard) rotation that leaves the path-continuation subtree appropriately situated.

M-4 Two sibling nodes can exchange child pointers. This uses a rotation, an instance of M-2, and a rotation that reverses the first (see figure 5).

First, the tree T_1, as generated by the off-line implementation of the prior rounds, is transformed to a rightward linear list R (any child node is the right child of its parent) using $O(n)$ standard rotations.

Let ψ denote the symmetric order list of nodes of T_1 (which conforms to the list R). A round of ζ consists of requests that conform to contiguous blocks in ψ of length m, some of which are internally permuted by the permutation π, and referred to as *π-blocks*, and the remaining not internally reordered, referred to as *plain-blocks*.

Broadly outlined,

(a) the list R is split into two lists, the left one L consisting of the (to become) π-blocks, and the other consisting of the plain-blocks. (The left list constitutes the left subtree attached to the first node of the other list.) The splitting requires $O(n)$ rotations. Next,

(b) the permutation π is applied (block-wise) to the list L consisting of the (to become) π-blocks using $O(n)$ rotations. Finally,

(c) the splitting process is reversed to merge the π-blocks with the plain-blocks, to reform a single list that now conforms to the round of ζ, so that passing its nodes through the root position satisfies the request sequence given by the round.

We first describe the merging of two lists, noting that the splitting transformation is implemented by reversing the rotations (using transformation M-1) that implement merging. The general configuration during the merging process appears as a "Y", with the stem consisting of the merged result thus far formed, and two arms of the Y being residual lists still needing to be merged. One arm of the Y shifts relative to the other using transformation M-2 (being attached as

a subtree to successive nodes of the other arm), as the second arm contributes to the merged result. Then the second arm shifts relative to the first, etc. The two arms thus shift in tandem relative to one another, until both are consumed, forming a single list.

Next, we describe how the permutation π is applied to each of the blocks of the list L, which constitutes the left subtree attached to the root of T_1 as modified immediately following the splitting step (a). For descriptive ease, we speak of L as a stand-alone tree, separated from T_1. Let $x_{j,i}$ denote the ith node of the jth block (of size m) in L. The following outlines the execution. (View L as being strictly leftward, its nodes being linked using left-child links.)

1. L is demultiplexed to form m chains; the jth node of the ith chain P_i is given by $x_{j,i}$. The nodes belonging to a common chain P_i are linked through their respective right links, and had occupied positions in L belonging to a common residue class $(\bmod\, m)$ prior to the demultiplexing transformation. Upon completion of the demultiplexing process a list whose individual elements consist of whole chains, formed by linking the respective first elements of these chains through their left child links in the order $P_1, P_2, \cdots, P_m$ (with the first node of P_1 being the root), emerges. $O(\text{length}(L))$ rotations suffice for this step. Figure 2 shows the completion of this demultiplexing.

2. A reordering of the list of chains is performed, so that they are then linked through their respective first nodes in the order $P_{\pi(1)}, \cdots, P_{\pi(m)}$. A bubble sort can accomplish this step with $O(m^2) = O(n)$ applications of transformation M-3 to exchange neighboring chains (a less concisely described method would be far more efficient). Figure 3 shows the completed reordering.

3. The chains are now reassembled (multiplexed), using $O(\text{length}(L))$ rotations, to form a list consisting of the now-permuted π-blocks of L, with the order among these blocks being the same as before the permutation π had been applied.

Considering the multiplexing and demultiplexing steps, multiplexing (see figure 4) is accomplished by alternating *peeling* and *shifting* phases, one peeling phase and one shifting phase for each (permuted) block in L. A peeling phase strips the lead elements from their respective residual chains, while simultaneously reconstituting the list of residual chains; each chain now having one less element. Then during the shifting phase, the subtree consisting of the reconstituted list of chains is repeatedly shifted to be properly appended to the single chain under construction. Demultiplexing is accomplished by reversing the process of multiplexing via repeated application of transformation M-1.

A single step of a peeling phase is shown in figure 6, using the transformation M-4. A single step of the shifting phase is shown in figure 7, using the transformation M-2.   □

The preceding lemmas substantiate the outline of section 2.1, so that the following theorem has been established.

Theorem 2. *Dynamic optimality is not attainable in the unordered tree model.*

3   Some Remarks

The established separation between off-line implementation versus on-line implementation in this model leaves open the question of what constitutes an efficient and versatile on-line implementation. The above proof can be modified to show that for any given on-line method, there is another (seemingly preposterous) on-line method that will dominate the former with respect to certain specific access request sequences. The latter method, moreover, satisfies the property that its rotations restructure only the access path to an immediate request. Perhaps a *plausibility* constraint of some sort, requiring that certain performance guarantees such as static optimality [3] be satisfied (and maybe some additional conditions), would circumvent this phenomenon, so that a reasonably defined "optimal" on-line method can emerge.
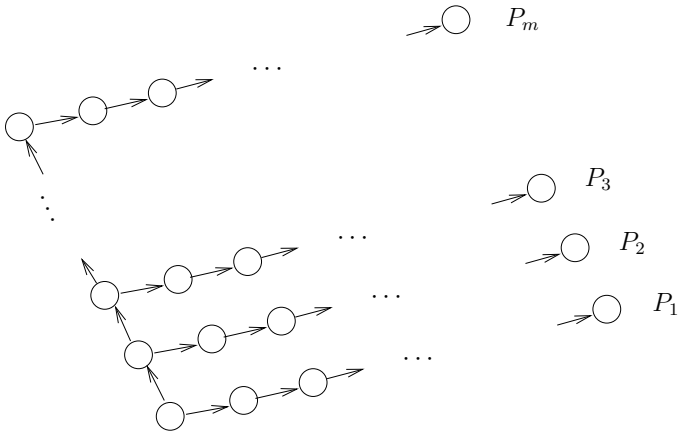
Fig. 2. After demultiplexing (Edges point from parent to child)
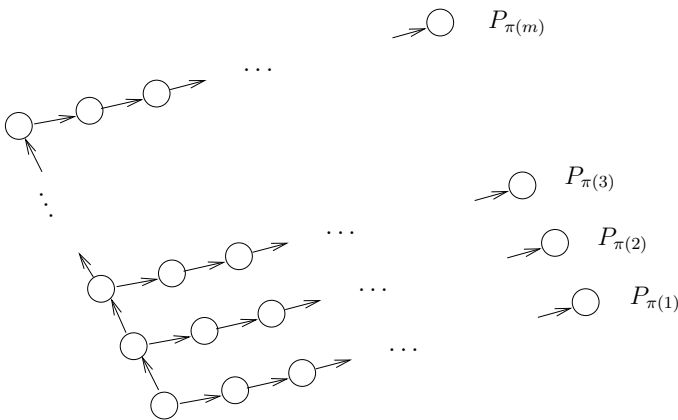
Fig. 3. After reordering chains
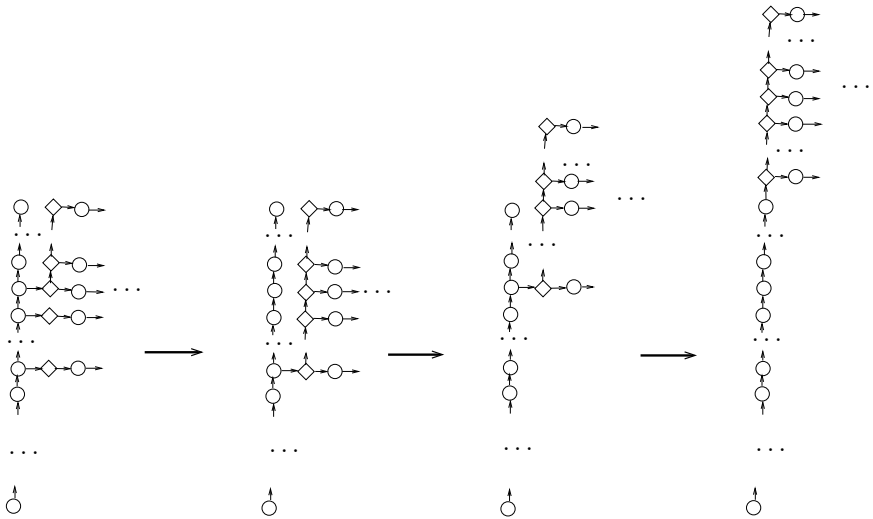
Fig. 4. Process of multiplexing: peeling and shifting

From left to right, the first diagram depicts a partially completed peeling phase. The second diagram depicts the completion of the peeling phase. The third diagram depicts a partially completed follow-on shifting phase, and the last diagram shows the completion of the shifting phase.
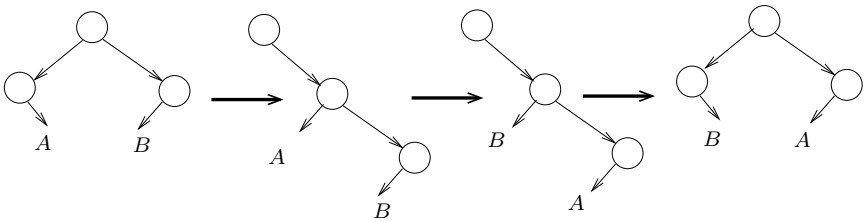

Fig. 5. Transformation M-4

From left to right, the first tree is transformed via one rotation into the second tree. The second is transformed via M-2 into the third, and a rotation transforms the third tree into the fourth, completing the M-4 transformation.
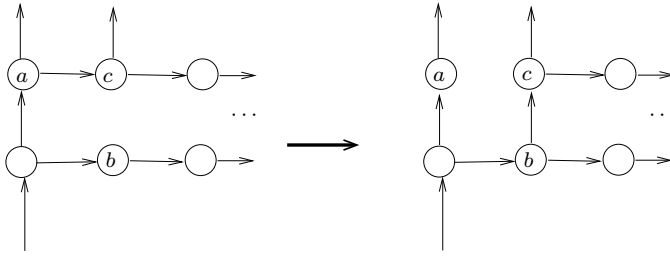
Fig. 6. A peeling step

A single step of peeling is applied to the tree on the left, to obtain
that on the right. This is accomplished via M-4, exchanging point-
ers from nodes a and b, so that c is pointed to from b instead of a.
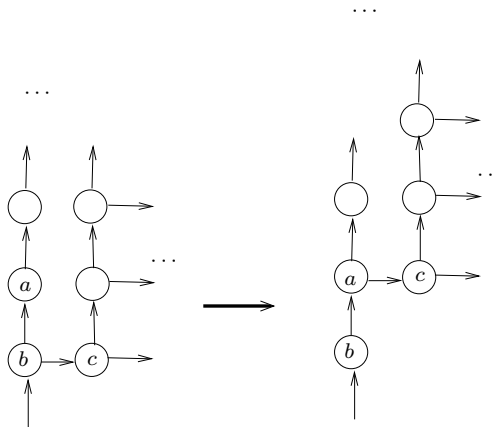The other pointer in the exchanged pair isn't shown.

$\dots$

Fig. 7. A shifting step

Transformation M-2 is applied to exchange pointers from nodes a
and b, so that c is pointed to from b instead of a. The other pointer
in the exchanged pair isn't shown.

References

1. Fredman, M.L.: Generalizing a theorem of Wilber on rotations in binary search trees
   to encompass unordered binary trees. Algorithmica (to appear)
2. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: a new
   form of self-adjusting heap. Algorithmica 1(1), 111–129 (1986)
3. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. JACM 32(3), 652–686
   (1985)
4. Wilber, R.: Lower bounds for accessing binary trees with rotations. SIAM J. on
   Computing 18(1), 56–67 (1989)

A   Appendix

We proceed to prove theorem 1, largely following the presentation given in [1], but with certain changes that are necessary for the main body of this paper, and also somewhat simplified. Our argument makes use of a *root sequence* construct. A sequence of rotations acting on an evolving tree is considered a root sequence provided that it satisfies the following recursively stated conditions:

1. A root sequence consists of zero or more blocks of rotations, referred to as *root sequence blocks*, with each block having exactly one rotation that changes the root node, that being its final rotation. Thus, the root node and the node sets of its two connected subtrees are static throughout the execution of the rotations of a single block, until the final rotation.
2. Let x be the node placed in the root position by the final rotation of a root sequence block, and let H be the subtree connected to the root that contains x (prior to the final rotation). Then all but the final rotation of the block are restricted to act upon the (evolving) subtree H.
3. All but the final rotation of a root sequence block recursively constitute a root sequence, but relative to the subtree H connected to the root that they are confined to.

We proceed to show first that a sequence of rotations R acting on an initial tree T can be rearranged to commence with a root sequence that brings nodes to the root position, exactly as had occurred during the implementation of R (including relative order).

A rearrangement of a sequence of rotations is considered *equivalent* to the original provided that (a) each rotation specifies an existing edge of the evolving tree (preserving the parent-child relation), and (b) the net transformation effected by the rearranged sequence is the same as that effected by the original sequence. We shall make use of the observation that two consecutive rotations commute provided that their respective edges are not incident upon a common node. All sequence rearrangements discussed in the sequel are understood to preserve equivalence, and will be verifiably so by design.

Lemma 4. *Given a sequence of rotations R, there exists a rearrangement R_1 of R having the form $R_1 = UV$ where U is a root sequence and V contains no rotations involving the node in the root position. Moreover, the sequence of nodes arriving at the root position of the evolving tree during the implementation of U (referred to as the sequence of* root-arrivals*), is identical to the sequence of root-arrivals for R.*

Proof: If R contains no rotations involving the root, then $V = R$. Otherwise, let $S = s_1, s_2, \cdots, s_m, r$ be the prefix of R, where r is the first rotation involving the root. Assume that the tree structure at the moment *immediately preceding* the rotation r appears as shown below, and that r rotates the edge (A, B).
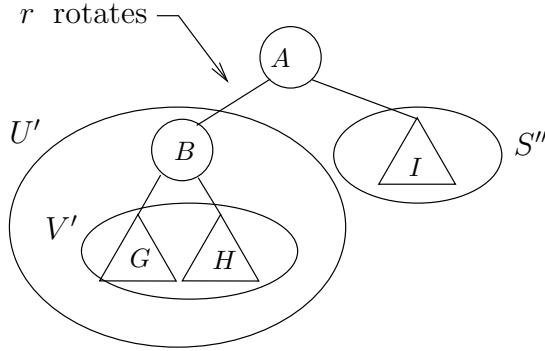
$$\underbrace{[S', S'']}_{S} r \Rightarrow S'S''r \Rightarrow U'V'S''r \Rightarrow \underbrace{U'r}_{\beta} V'S''$$

Fig. 8.

The above figure may prove helpful while reading the following description. To explain the notation used in the figure, $[X, Y]$ refers to a sequence formed of two subsequences X and Y interleaved in some arbitrary manner, and for sequences K and L, $K \Rightarrow L$ depicts K being rearranged as L. The encircled substructures are labeled with rotation sequences; the constituent rotations of a given label are confined to edges that join nodes belonging to the corresponding encircled node sets as the structure evolves. We note that until the rotation r takes place, the node sets of the two subtrees connected to A remain static. The subtree comprised of the node set that contains B is referred to here as the *has-B-subtree*. Now let $S' = t_1, \cdots, t_g$ be the subsequence of rotations among the s_j's that occur among edges in the has-B-subtree. The subsequence S'' of remaining rotations among the s_j's are on edges confined to nodes in the sibling subtree (that depicted as I in figure 8) as it evolves. Thus, we preserve sequence equivalence upon moving the S' subsequence of rotations ahead of the other rotations S'' (the first of the three rearrangements represented in the above figure). By induction these S' rotations can be rearranged (the second rearrangement in the figure) as a sequence $U'V'$, where U' forms a root sequence for this has-B-subtree (ultimately promoting B to its root position), and V' contains no rotations involving the root of the subtree; this root staying fixed as B over the course of V'. (Referring to figure 8, V' consists of rotations on edges confined to nodes in the subtrees G and H, as they evolve, whose respective node sets are established upon completion of U'.) Now the rotation r can be moved ahead of the rotations S'' (confined to the evolving subtree shown as I), and also ahead of the rotations in V' (confined to G and H), preserving sequence equivalence (the third rearrangement in the figure), since none of the these rotations that r passes over are incident upon either of its two nodes. Doing so creates a syntactically

valid first block, $\beta = U'r$, for a root sequence, with the rotation r bringing the same next node to the root position as had occurred during the implementation of R.

The rearrangement leading to β has not altered the order of root arrivals. By induction the remaining terms $\tilde{R}$ of R (consisting of all rotations that *now* follow r) can be rearranged to commence with a root sequence (respecting root-arrivals), that when appended to β yields the promised root sequence U (and the residual tail of this rearrangement of $\tilde{R}$, being devoid of any rotations that involve the root, provides the required V).   □

We now proceed with the encoding of a root sequence. Given a root sequence block (possibly embedded recursively in another block) we augment the block with a navigation process as follows: navigation is considered as being initiated from the root of the (sub)tree acted upon by the rotations of the block. A single navigational bit is placed at the beginning of the block to indicate from which subtree of the root its replacement node is drawn by the single rotation that concludes the block, and a single step of navigation is considered as taking place, with the position of navigation being transferred to the root of this subtree. By design, immediately before the rotation that concludes the block takes place, the position of navigation will be found at the node in the child position of the edge being rotated, and remains at that node as the rotation takes place, so that it will be positioned at the parent node end of the just rotated edge, and therefore once again at the root node of the tree acted upon by the block rotations. The recursive sub-blocks embedded in the given block are similarly augmented, and at the completion of a given sub-block, the position of navigation is appropriately located at the root node of the subtree to be acted upon by the follow-on sub-block – should there be one – and when there is no follow-on sub-block, the position of navigation is appropriately located at the child node end of the edge to be rotated by the final rotation of the parent block. (If there are no recursive sub-blocks the step of navigation placed at the beginning of a block positions navigation at the appropriate node for the ensuing single rotation of the block.) Generally, one bit of navigational information is provided for each rotation in a root sequence block; the bit entered at the beginning of the block is accounted for by the rotation that completes the block.

As there are only two possibilities for rotating an edge from the vantage of the node in the child position, i.e. whether the rotation is standard or non-standard, only one bit is required to specify a given rotation. We note that the edge affected by the rotation is uniquely determined, given the position of navigation at that point. To signify whether a bit is intended as descriptive of a step of navigation versus the form (standard or non-standard) of a rotation, we immediately precede it with an additional bit that serves this purpose. Thus we obtain a binary encoding of a root sequence having length 4 times that of the number of rotations in the sequence: given this encoding and a representation of the tree acted upon by the root sequence, the evolution of the tree, as determined by the associated root sequence, can be recovered. (Given the present position in the present configuration of T, the next pair of bits in an encoding provides the

next step of navigation, or the next rotation – as may be the case – so that the next configuration of T and position of navigation within T can be determined.) A further augmentation facilitates recovery (hence encoding) of an access request sequence: indicators are inserted in appropriate locations that specify that the node presently occupying the root position satisfies the pending access request. When the root satisfies the pending access request, and navigation is positioned at this root (as must be the case after the pertinent rotation), then placement of a single bit (otherwise) signifying that the next bit describes a rotation, between the node of current position and its parent – an impossibility in this positional context – serves to provide this signal.

Applying this encoding method to the root sequence constructed in lemma 4, derived from the least costly implementation of a given access request sequence, completes the proof of Theorem 1.

The following Observation is used in the main body of the paper.

Observation B. Let σ be an augmented root-sequence. Prior to any rotation in σ, the navigation process must at some point be positioned at each of the nodes joined by the edge undergoing rotation.

This is clear by consideration of the root-sequence sub-block that concludes with the rotation under consideration; navigation is positioned at the node in the parent position of the edge to be rotated as the block gets underway, and at the node in the child position immediately prior to the rotation.

Resilient and Low Stretch Routing through Embedding into Tree Metrics

Jie Gao and Dengpan Zhou

Department of Computer Science,
Stony Brook University,
Stony Brook, NY 11794, USA
{jgao,dpzhou}@cs.sunysb.edu

Abstract. Given a network, the simplest routing scheme is probably routing on a spanning tree. This method however does not provide good stretch — the route between two nodes can be much longer than their shortest distance, nor does it give good resilience — one node failure may disconnect quadratically many pairs. In this paper we use two trees to achieve both constant stretch and good resilience. Given a metric (e.g., as the shortest path metric of a given communication network), we build two *hierarchical well-separated trees* using randomization such that for any two nodes u, v, the shorter path of the two paths in the two respective trees gives a *constant* stretch of the metric distance of u, v, and the removal of any node only disconnect the routes between $O(1/n)$ fraction of all pairs. Both bounds are in expectation and hold true as long as the metric follows certain geometric growth rate (the number of nodes within distance r is a polynomial function of r), which holds for many realistic network settings such as wireless ad hoc networks and Internet backbone graphs. The algorithms have been implemented and tested on real data.

1 Introduction

This paper considers a fundamental problem of designing routing schemes that give low stretch and are resilient to node failures. We consider a metric (P, d) on n nodes (e.g., as the shortest path metric of a given network), in particular, metrics of bounded geometric growth as a popular family of metrics in the real world. The result we present in this paper is a routing structure, constructed in a distributed manner such that each node of P keeps routing information of size $O(\log n)$, the route discovered has constant stretch (e.g., a constant factor longer than the metric distance), and the routing structure is robust to node failures, where a single node failure will only disconnect $O(1/n)$ fraction of the routes between all possible pairs.

The technique we use in this paper is through embedding into tree metrics. Given a metric (P, d), the simplest way to route is probably by taking a spanning tree to guide message routing. This has a number of benefits, as a tree metric is a much simpler metric with many special features. For example, between any two vertices in a tree, there is a unique simple path connecting them, and the unique path can be found in a local manner by first traversing up the tree towards the root, and traversing down the tree at the lowest common ancestor. There is a simple labeling scheme such that one can use routing table of $O(\log n)$ bits at each node to support routing on a tree [3].

However, routing on a spanning tree of the metric has a number of problems, in particular, the *poor stretch* and *lack of resilience*. The path on a tree might be much longer than the metric distance. Take the shortest path metric of a cycle of n vertices, any spanning tree will separate some pair of vertices, adjacent on the cycle, by distance $n - 1$. That is, the distortion introduced by routing on a spanning tree is factor of $\Omega(n)$ of their true distance. A more serious problem of routing on a tree is due to the lack of robustness to node failures. If a node fails or decides not to cooperate and stops forwarding messages, the tree is broken into pieces and in the worst case quadratically many pairs have their paths disconnected.

In this paper we use embedding into tree metrics for efficient, scalable routing, but address the shortcomings regarding stretch and resilience. Instead of using one tree, we use simply two trees. The shorter one of the paths from two trees may have better stretch. Regarding node failures, if a node u fails and the path between two nodes x, y is disconnected as it goes through u, the path connecting x, y in the second tree hopefully does not contain u and still remains valid. We briefly elaborate our technical approach and then relate to prior work.

Our Results. The tree embedding we use follows from the embedding of a general metric into tree metrics with low distortion. Given a metric (P, d) we embed it to a hierarchically well-separated tree (HST). The leaf nodes of the HST are 1-to-1 mapped to nodes in P and internal nodes of the HST are also mapped to nodes of P although certain nodes may appear multiple times. The embedding of (P, d) into a tree metric necessarily introduces distortions. As discussed earlier, using a fixed tree one cannot avoid the worst case distortion of $\Omega(n)$. But if one build a tree, chosen randomly from a family of tree metrics, the *expected* distortion can be bounded by $O(\log n)$. Thus using this tree for routing one immediately obtains $O(\log n)$ stretch routing with low routing overhead. Approximating a metric with probabilistic hierarchical well-separated trees was first proposed by Bartal [6,5], with the motivation that many problems are easier to solve on a tree than on a general graph. Later, Fakcharoenphol *et al.* [11] improved the distortion to $O(\log n)$ for any n node metric and this is tight.

The results we prove in this paper are mainly in three pieces

- Using two HSTs, randomly constructed with independent seeds, we show that the stretch can be improved to a constant in expectation. That is, for any two nodes x, y, between the two paths in the two HSTs respectively, one of them is short and is at most a constant factor of the metric distance between x, y.
- Regarding the resilience of using one HST for routing, we show that for any node failure, the number of pairs with their routes on the HST disconnected is at most a fraction of $O(\log \Delta/n)$ of all pairs, where Δ is the aspect ratio of (P, d), defined as the furthest pair distance versus the closest pair distance. When Δ is polynomial in n the bound is as small as $O(\log n/n)$ but in the worst case when the aspect ratio is exponential the bound can be bad.
- Using two HSTs we substantially improve the routing resilience. We build two HSTs with random, independent seeds. In the case of a node failure, we show that the number of pairs with their routes on both HSTs disconnected is at most a fraction of $O(1/n)$ of all pairs, thus removing the factor of $O(\log \Delta)$ compared with the case of a single HST.

The results hold for metrics with 'geometric growth', that is, the number of nodes within distance r from any node grows as a polynomial function of r, not exponential (as in the case of a balanced binary tree). Such a family of metrics appears in many realistic settings, either due to physical constraints such as in wireless networks and VLSI layout networks, or due to geographical constraints such as in peer-to-peer overlay networks [25,22,24]. In the next section we introduce the rigorous definitions and elaborate the precise assumptions for each of the results.

Last remark that in the case that (P, d) is the shortest path metric of a given network G, there is a distributed algorithm [12] that constructs the HST with a total number of messages bounded by $O(n \log n)$. In addition, each node is given a label of size $O(\log n)$ such that one can route on an HST using only the node label information. Thus the entire scheme of using one or multiple HSTs for robust, low-stretch and efficient routing can be implemented in a completely de-centralized manner.

Prior Work. There is numerous prior work on routing. We only have the space to review some most relevant ones.

The traditional routing methods as used for the Internet are essentially shortest path routing. Essentially each node keeps a routing table of size $O(n)$ to save the next hop on the shortest path for each destination. This is equivalent to maintaining n shortest path trees, rooted on every node. From this perspective, our approach defines one or two global trees, rather than one tree per node. By doing so we can substantially reduce the size of the routing table from $O(n)$ to $O(\log n)$, while still keeping the routing stretch by a constant.

From a theoretical aspect, compact routing that minimizes the routing table size while achieving low stretch routing has been studied extensively [23,14]. There are two popular models in the literature, the *labeled routing model* (in which naming and routing schemes are jointly considered) [9,10,30] and *name-independent routing* (in which node IDs are independent of the routing schemes) [2,17]. Generally speaking, the theoretical results in compact routing in a graph whose shortest path metric has a constant doubling dimension are able to obtain, with polylogarithmic routing table size, $1 + \varepsilon$ stretch routing in the labeled routing scheme (see [8] and many others in the reference therein), and constant stretch factor routing in the name-independent routing scheme [17,1] (getting a stretch factor of $3 - \varepsilon$ will require linear routing table size [1]). The schemes here are all by centralized constructions and aim to get the best asymptotic bounds. Our focus of using tree embedding is to obtain practical routing solutions with theoretical guarantee. Further, the compact routing schemes above have no consideration of robustness to node failures.

Routing methods that can recover from node or link failures receive a lot of interests recently. There are many heuristic methods for Internet routing such as fast re-routing [28], Loop-free alternate (LFA) [4], O2 [7], DIV-R [26] and MARA [31]. But these methods have no theoretical guarantee. There are some previous work considering approximate shortest paths avoiding a failed vertex [21]. Path splicing [19] uses multiple shortest path trees with perturbed edge weights. When routing in one tree metric encounters a problem, the message is quickly routed on a different tree. Using a similar idea we can also use multiple HSTs to recover in-transit failures. The difference is that we do not keep separate shortest path trees rooted at each node, but rather use two

global trees. Thus our storage overhead is substantially better. Our simulation shows that we have roughly the same routing robustness, our stretch is a little higher but we substantially save on routing table size. A more recent related work is using multiple HSTs for information delivery to mobile users [20].

2 Preliminaries

Metrics with Geometric Growth. An important family of metrics is the metrics with 'geometric growth'. There are several related definitions. Given a metric (P, τ), let $B(p, r) = \{v \mid \tau(p, v) \leq r\}$ denote the radius r ball centered at p. In [16], a metric has bounded *expansion rate* (also called the KR-dimension, counting measure) k_1 if $|B(v, 2r)| \leq k_1|B(v, r)|$ for a constant k_1; and in [15], a metric has bounded *doubling dimension* k_2 if $B(v, 2r)$ is contained in the union of at most k_2 balls with radius r for a constant k; in [18,13], a metric has upper bounded growth rate *growth rate* k_3 if for every $p \in V$ and every $r \geq 1$, $|B(p, r)| \leq \rho r^{k_3}$, for a constant ρ and k_3. A few sensor network papers [27,32] consider a model when the growth rate is both upper and lower bounded, i.e., $\rho^- r^{k_4} \leq |B(p, r)| \leq \rho^+ r^{k_4}$ for a constant k_4, where $\rho^- \leq \rho^+$ are two constants. We denote the family of metrics with constant expansion rate, constant doubling dimension, constant upper bounded growth rate, and constant upper and lower bounded growth rate as $\mathcal{M}_{\text{expansion}}, \mathcal{M}_{\text{doubling}}, \mathcal{M}_{\text{growth}}^+, \mathcal{M}_{\text{growth}}$ respectively. It is not hard to see that $\mathcal{M}_{\text{growth}} \subseteq \mathcal{M}_{\text{expansion}} \subseteq \mathcal{M}_{\text{doubling}} \subseteq \mathcal{M}_{\text{growth}}^+$. See [15,13] for more discussions. In terms of the results in this paper the detailed definitions actually matter. In the following we will make it clear which definition is needed for each result.

Embedding into Tree Metrics. Given two metric spaces (X, d_X) and (Y, d_Y), an injective mapping $f : X \rightarrow Y$ is called an *embedding* of X into Y. We can scale up Y to make the embedding to be *non-contractive*, i.e., for any $u \neq v \in X$: $d_Y(f(u), f(v)) \geq d_X(u, v)$. We say Y *dominates* X. The distortion of the pair u, v is $\text{dist}_f(u, v) = \frac{d_Y(f(u), f(v))}{d_X(u, v)}$. The distortion of the embedding f is $\text{dist}(f) = \max_{u,v \in X} \text{dist}_f(u, v)$.

Given a metric (P, d), we embed it to a tree metric and use the tree metric to guide message routing. Ideally we want the route length to be close to the metric distance. As shown in the introduction, it is not possible to get distortion of $o(n)$ using a single tree. However, it is known that for any metric (P, d), one can use randomization such that the *expected* distortion is only $O(\log n)$. Such a tree is a type of a *hierarchical well-separated tree H*, as defined below.

Definition 1 (α-HST [5]). *A rooted weighted tree H is an α-HST if the weights of all edges between an internal node to its children are the same, all root-to-leaf paths have the same hop-distance, and the edge weights along any such path decrease by a factor of α as we go down the tree.*

In this paper we focus on 2-HST. The leaves of T are the vertices in P, and the internal nodes are Steiner nodes. Fakcharoenphol, Rao and Talwar [11] have shown that for any metric (P, d) one can find a family of trees such that a randomly selected metric from the family has expected distortion of $O(\log n)$, which is also tight.

Review of The FRT Algorithm [11]. Without loss of generality, we assume that the smallest distance between any two vertices in P is 1 and the diameter of P is Δ. The aspect ratio is also Δ. Assume $2^{\delta-1} < \Delta \le 2^\delta$. The FRT algorithm proceeds in a centralized manner by computing a hierarchical cut decomposition $D_0, D_1, \cdots, D_\delta$.

Definition 2 (Cut decomposition). *For a parameter r, an r-decomposition of a metric (P, d) is a partitioning of P into clusters, each centered at a vertex with radius r.*

Definition 3 (Hierarchical cut decomposition). *A hierarchical cut decomposition of (P, d) is a sequence of $\delta + 1$ nested cut decompositions $D_0, D_1, \cdots, D_\delta$ such that*

- $D_\delta = P$, i.e.*the trivial partition that puts all vertices in a single cluster.*
- D_i *is a 2^i-cut decomposition, and a refinement of D_{i+1}. That is, each cluster in D_{i+1} is further partitioned into clusters with radius 2^i.*

To find the hierarchical cut decomposition, one first chooses a random permutation $\pi : P \to \{1, 2, \cdots, n\}$ of the nodes. We use $\pi(i)$ to denote the node with rank i in the permutation. We also fix a value β chosen uniformly at random from the interval $[1, 2]$. For each i, compute D_i from D_{i+1} as follows. First set β_i to be $2^{i-1}\beta$. Let S be a cluster in D_{i+1}. Each vertex $u \in S$ is assigned to the first (according to π) vertex v within distance β_i. We also say that u nominates v. Each child cluster of S in D_i then consists of the set of vertices in S assigned to the same center. We denote the center of a cluster C by $\operatorname{center}(C)$. Note that all clusters in D_i have radius $2^{i-1} \le 2^{i-1}\beta \le 2^i$. Remark that a node can nominate a center outside of its current cluster in D_{i+1} and one node can be the center for multiple clusters.

An alternative view of the hierarchical cut decomposition is to define for each node u a δ-dimensional *signature vector* $S(u)$. The i-th element in the vector is the lowest rank node within distance $2^i\beta$. $S(u)_i = \arg\min_{v \in B(u, 2^i\beta)} \pi(v)$ where $B(p, r)$ is the collection of nodes within distance r from node p. A cluster at level i contains all the nodes with the same prefix $[1, i]$ of their signature vectors.

To turn the hierarchical cut decomposition to a 2-HST, the points of P are the leaf nodes of the HST and each internal node in the HST corresponds to a cluster of nodes in the hierarchical partitioning. The refined clusters in D_{i-1} of a cluster C in D_i are mapped to children of C. The root corresponds to D_0. We can also use the center u of a cluster C as the representative node of C in the HST. Thus the root of the HST has $\pi(1)$ as its representative node. Denote by P_i the centers of the clusters in D_i. P_i is the set of node that are 'nominated' by others at level i.

The HST has $\delta + 1$ levels, at $0, 1, \cdots, \delta$. The level i has a number of internal nodes in the HST corresponding to P_i. The edge weight connecting a cluster C in D_i to its children clusters in D_{i-1} is 2^i, i.e., greater than the radius of the cluster C. Clearly the HST metric dominates (V, d), as one only relaxes the distances. For any two nodes u, v, suppose that they are first separated in different clusters in the decomposition D_i, i.e., their lowest common ancestor in the HST is at level $i + 1$. In this case we have their distance on the tree to be $d_H(u, v) = 2\sum_{j=1}^i 2^j = 2^{i+2}$. Fakcharoenphol, Rao and Talwar [11] proved that $d_H(u, v) \le O(\log n)d(u, v)$, in expectation over all random choices of β and π. A distributed implementation of the algorithm is available in [12].

3 Constant Distortion Routing Using Two HSTs

Starting from this section we examine the properties of routing using *two* HSTs.

Constant Distortion Embedding in Two HSTs. For a given metric (P, d) with expansion rate k, we build two HSTs, H_1 and H_2 with independent, random seeds using the algorithm in [11]. For any two points u, v in P, we define the distance between them to be the minimum shortest path in the two trees. That is $d_H(u, v) = \min\{d_{H_1}(u, v), d_{H_2}(u, v)\}$.

Theorem 1. *For any metric (P, d) with expansion rate k and two HSTs H_1, H_2, there is a constant c such that for any two nodes $u, v \in P$,*

$$E[d_H(u, v)] = E[\min\{d_{H_1}(u, v), d_{H_2}(u, v)\}] \leq c \cdot k^4 \cdot d(u, v).$$

For two nodes $u, v \in P$, denote their lowest common ancestor (LCA) in H_i by $\text{LCA}_i(u, v)$, for $i = 1, 2$. And denote $\text{LCA}(u, v) = \min\{\text{LCA}_i(u, v), i = 1, 2\}$. Thus $d_H(u, v) = 2^{i+2}$ if $\text{LCA}(u, v)$ is at $i + 1$. Now we have $E[d_H(u, v)] = \sum_{i=0}^{\delta-1} \text{Prob}\{\text{LCA}(u, v) \text{ is at level } i + 1\} \cdot 2^{i+2}$. With the following Lemma that bounds the probability that $\text{LCA}(u, v)$ is at $i + 1$ (the proof is in the Appendix), we can prove the Theorem.

Lemma 1

$$Prob\{\text{LCA}(u, v) \text{ is at level } i + 1\} \leq \begin{cases} 0, & \text{if } 2^{i+2} < d(u, v); \\ 3k^4 \cdot d^2(u, v)/2^{2i-4}, & \text{if } 2^{i-2} \geq d(u, v). \end{cases}$$

Proof (Theorem 1). With the above lemma, we can prove Theorem 1 easily. Suppose j^* is the smallest i such that $2^{i+2} \geq d(u, v)$,

$$\begin{aligned} E[d_H(u, v)] &= \sum_{i=0}^{\delta} \text{Prob}\{\text{LCA}(u, v) \text{ is at level } i + 1\} \cdot 2^{i+2} \\ &\leq \sum_{i=j^*+4}^{\delta} [3k^4 \cdot \frac{d^2(u,v)}{2^{2i-4}}] \cdot 2^{i+2} + \sum_{i=j^*}^{j^*+3} 2^{i+2} \\ &\leq 2^7 \cdot 3k^4 \cdot d^2(u, v)/2^{j^*} + 14d(u, v) \leq (96k^4 + 14) \cdot d(u, v). \end{aligned}$$

Routing with Two HSTs. To route a message from a source to a destination node, we check each set of labels to see which tree gives a lower LCA (lowest common ancestor). That tree will provide a path with only constant stretch. We remark that the storage requirement for each node is very low, in the order of $O(\log n)$.

4 Resilience to Node Failures Using Two HSTs

In this section we show that using two trees, instead of one, can improve the routing robustness substantially. For a pair of node u, v, if the path connecting them is disconnected on the first tree, it is still possible that there is a path between them on the second tree. Thus one can switch to the second tree for a backup route and recover from sudden, unforseen failures instantaneously, akin to the path splicing idea [19].

Robustness of a single random HST. We first examine the properties of a single HST in terms of node failure. When a node u fails, any path on the HST that uses a cluster

with u as the center is disconnected. We examine how many such pairs there are. The worst case is that u is a center of a cluster near the root of the HST – this will leave big components and $\Omega(n^2)$ number of pairs disconnected. For example, if the node $\pi(1)$ fails. However, since the construction of the HST uses random permutations (assuming the adversary has no control over the choice of this random permutation, as in standard settings of randomized algorithms), a single node failure is unlikely to be near the root. The following theorem works for any metric (P, d) with constant doubling dimension.

Theorem 2. *Given a node u and an HST, the expected number of nodes within clusters with u as center is $O(\log \Delta)$, where Δ is the aspect ratio of the metric (P, d) with constant doubling dimension.*

Proof. Suppose a node x is within a cluster with u as the center, say this cluster is at level i. Then we know that $d(u, x) \leq \beta 2^i$ and u is the highest rank node in $B(x, \beta 2^i)$. Now, take $\ell_u(x)$ as the lowest level j such that $d(u, x) \leq \beta 2^i$. Clearly, $\ell_u(x) \leq i$. Thus $B(x, \beta 2^{\ell_u(x)}) \subseteq B(x, \beta 2^i)$. That is, u is the lowest rank node at level $\ell_u(x)$ as well. The probability for that to happen is $1/|B(x, \beta 2^{\ell_u(x)})|$. Thus the probability that x is inside a cluster with u as center is no greater than $1/|B(x, \beta 2^{\ell_u(x)})|$.

Now, the expected number of nodes within clusters with u as center, W, is,

$$W = \sum_x \text{Prob}\{x \text{ is in a cluster centered at } u\} \leq \sum_x 1/|B(x, \beta 2^{\ell_u(x)})|$$
$$= \sum_j \sum_{x \in B(u, \beta 2^j) \setminus B(u, \beta 2^{j-1})} 1/|B(x, \beta 2^j)| \leq \sum_j \sum_{x \in B(u, \beta 2^j)} 1/|B(x, \beta 2^j)|.$$

Now, recall that the metric (P, d) has constant doubling dimension γ. Thus we can cover the point set $B(u, \beta 2^j)$ by balls of radius $\beta 2^{j-1}$, denoted as sets $B_1, B_2, \cdots, B_m$, $m \leq 2^\gamma$. Since the points in B_j are within a ball with radius $\beta 2^{j-1}$, all the points within B_j are within distance $\beta 2^j$ of each other. That is, for a node $y \in B_i$, $B_i \subseteq B(y, \beta 2^j)$. Thus $|B_i| \leq |B(y, \beta 2^j)|$, where $y \in B_i$. Now we group the points of $B(u, \beta 2^j)$ first by the balls they belong to, and then take the summation over the balls.

$$W \leq \sum_j \sum_{x \in B(u, \beta 2^j)} 1/|B(x, \beta 2^j)| = \sum_j \sum_i^m \sum_{x \in B_i} 1/|B(x, \beta 2^j)|$$
$$\leq \sum_j \sum_i^m \sum_{x \in B_i} 1/|B_i| = \sum_j \sum_i^m |B_i| \cdot 1/|B_i| = \sum_j m \leq 2^\gamma \delta = O(\log \Delta).$$

The above lemma shows that the total number of pairs disconnected if one random node is removed is bounded by $O(n \log \Delta)$.

Robustness of Two Random HSTs. We now examine the robustness property of using two random HSTs and bound the number of pairs 'disconnected' in both trees, i.e., their routes by using both HSTs go through u. For this case we assume that (P, d) has both constant upper and lower bounded growth ratio. By using two trees we reduce the expected number of disconnected pairs from $O(n \log \Delta)$ to $O(n)$.

Theorem 3. *The number of pairs of nodes disconnected in two HSTs, constructed using independent random permutations, is a fraction of $O(1/n)$ of all pairs, for a metric (P, d) with both constant upper and lower bounded growth ratio.*

Proof. Take a pair of nodes x, y, the paths connecting the two in both trees are disconnected if and only if in each of the tree, exactly one node is in a cluster with u as center and another one is not in any cluster with u as center. Denote by $P_u(x)$ the probability

that x is in a cluster with u as the center. $P_u(x) \leq 1/|B(x, \beta 2^{\ell_u(x)})|$. Thus the expected number of pairs of nodes disconnected after node u is removed is,

$$W_2 = \sum_y \sum_x 4[P_u(x)]^2[1 - P_u(y)]^2 \leq \sum_y \sum_x 4[1/|B(x, \beta 2^{\ell_u(x)})|]^2$$
$$= 4n \sum_j \sum_{x \in B(u, \beta 2^j) \setminus B(u, \beta 2^{j-1})} 1/|B(x, \beta 2^j)|^2$$
$$= 4n \sum_j (|B(u, \beta 2^j)| - |B(u, \beta 2^{j-1})|)/|B(x, \beta 2^j)|^2.$$

If (P, d) has constant bounded growth ratio k, we know that $\rho^- \beta^k 2^{jk} \leq |B(x, \beta 2^j)| \leq \rho^+ \beta^k 2^{jk}$ for constants $\rho^- \leq \rho^+$. Thus

$$W_2 \leq 4n \sum_j [\rho^+ \beta^k 2^{jk}]/[\rho^- \beta^k 2^{jk}]^2 = 4n \sum_j \rho^+/(\rho^-)^2 \cdot 1/(\beta^k 2^{jk}) = O(n).$$

Robustness of Two HSTs with Reversed Rank. An alternative method to use two trees for robust routing is to construct the second tree to be as different as possible from the first tree. One idea is to build the second HST H_2 by using permutation π_2, as the reverse of the permutation π_1 used in H_1. As an immediate consequence of that, suppose x is in a cluster with u as the center in H_1, then x can not be inside any cluster with u as center in H_2. This is because the rank of x is greater than u in π_1, and the rank of x must be smaller than the rank of u in π_2. Thus x can never nominate u in H_2. This says that the set of nodes 'chopped off' by the failure of u in H_1 will not be chopped off in H_2, ensuring certain robustness of routing. We also evaluate this method by simulations and it performs no worse than the two random HSTs.

5   Simulations

This section evaluates our two HSTs mechanism in terms of path stretch and reliability against node or link failures. We run our simulation on two data sets. The first data set is a unit disk graph on a network of nodes deployed using perturbed grid model, a widely used model for wireless sensor networks. To be specific, the networks are generated by perturbing n nodes of the $\sqrt{n} \times \sqrt{n}$ grid in the $[0, 1]^2$ unit square, by 2D Gaussian noise of standard deviation $\frac{0.3}{\sqrt{n}}$, and connecting two resulting nodes if they are at most $\frac{2}{\sqrt{n}}$ apart. The average degree of the network generated in this way is about 5. The second data set is the Sprint backbone network topology inferred from Rocketfuel [29], which has 314 nodes and 972 edges.

We first study the routing stretch by using 1 HST and 2 HSTs respectively assuming no node or link failures. We also examine the number of pairs disconnected when using one HST and two HSTs respectively. For the two HSTs, we carry out simulations for both random HSTs and a pair of HSTs whose node rankings are in reverse of each other. Next, we compare our scheme with the path splicing approach [19] when link or node failures exist. We conducted two sets of experiments using two randomly constructed HSTs, and a pair of HSTs with reversed rank. For both methods, the next hop has a probability p to fail at each step. In case of a failure, we route a message using one HST or one splicing, and switch to another HST or splicing instance when we encounter a link or node failure on the next hop. The path stretch is computed only on the messages that reach the destination before TTL runs down to zero.
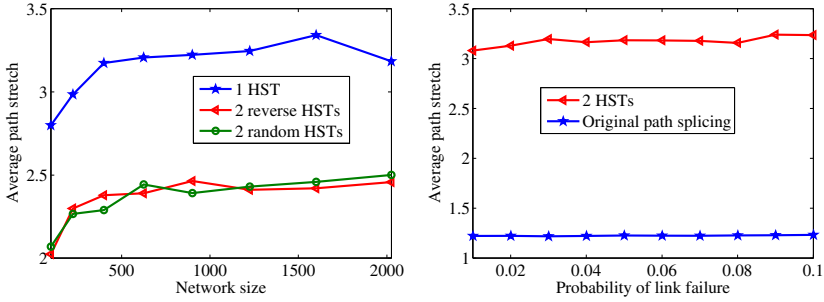
Fig. 1. [Left] Average path stretch using 2 HSTs v.s. 1 HST for the unit disk network (for each network size, we sample 20 different networks and take the average value); [Right] Path stretch using 2 HSTs v.s. path splicing on the Sprint topology, where each underlying link fails with probability p from 0.01 to 0.1.
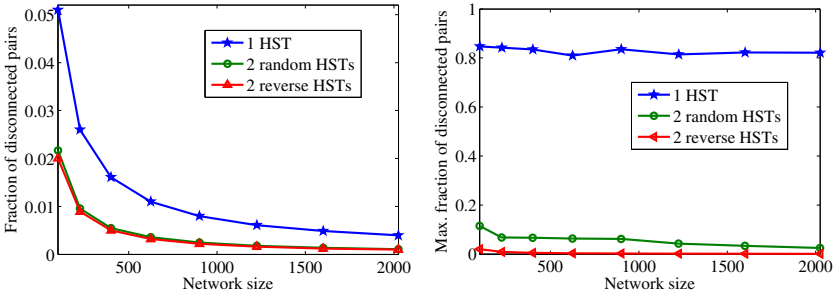

Fig. 2. By setting each node fail and removing all its adjacent edges from the network, we compute the fraction of disconnected pairs using 1 HST v.s. 2 HSTs. [Left]: average value. [Right]: maximum value. Results are the average for 20 unit disk networks for each network size.
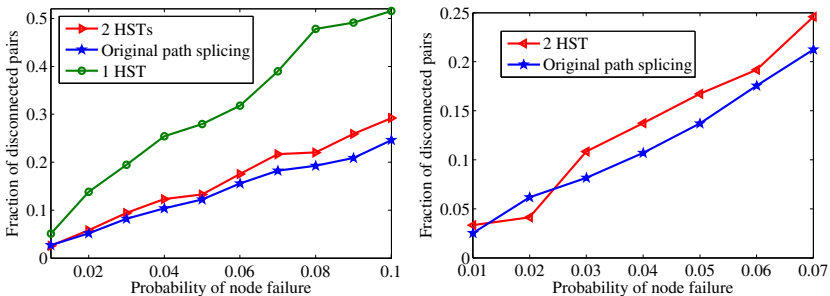
Fig. 3. The fraction of messages that are not delivered to the destination when next hop node fails with probability p. To avoid infinite loop, we set TTL to be $5n$. [left]: results on Sprint network. [Right]: average results on 50 unit disk graphs with 400 nodes.

Summary of simulation results. Our observations from these experiments are:

- *Small path stretch.* Without failure, the path stretch from two HSTs improves significantly over a single HST (Figure 1 [left]). In case of failures, using two HSTs gives worse stretch compared with path splicing, but reducing the routing table size significantly (Figure 1 [right]).
- *Extremely good resilience.* The maximum number of disconnected pairs using one HST can be bad, roughly 85% but using two HSTs the number drops to below 10% (Figure 2). Combining 2 HSTs with path splicing, our routing performance (i.e., the delivery rate), is nearly as good as using $2n$ spanning trees in the path splicing method (Figure 3).

References

1. Abraham, I., Gavoille, C., Goldberg, A.V., Malkhi, D.: Routing in networks with low doubling dimension. In: Proc. of the 26th International Conference on Distributed Computing Systems (ICDCS) (July 2006)
2. Abraham, I., Malkhi, D.: Name independent routing for growth bounded networks. In: SPAA 2005: Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 49–55 (2005)
3. Alstrup, S., Gavoille, C., Kaplan, H., Rauhe, T.: Nearest common ancestors: a survey and a new distributed algorithm. In: SPAA 2002: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 258–264 (2002)
4. Atlas, A., Zinin, A.: Basic specification for ip fast reroute: Loop-free alternates. In: IETF RFC 5286 (September 2008)
5. Bartal, Y.: Probabilistic approximation of metric spaces and its algorithmic applications. In: FOCS 1996: Proceedings of the 37th Annual Symposium on Foundations of Computer Science, p. 184 (1996)
6. Bartal, Y.: On approximating arbitrary metrices by tree metrics. In: STOC 1998: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, pp. 161–168 (1998)
7. Reichert, Y.G.C., Magedanz, T.: Two routing algorithms for failure protection in ip networks. In: Proc. ISCC (2005)
8. Chan, H.T.-H., Gupta, A., Maggs, B.M., Zhou, S.: On hierarchical routing in doubling metrics. In: SODA 2005: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 762–771 (2005)
9. Cowen, L.J.: Compact routing with minimum stretch. In: SODA 1999: Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 255–260 (1999)
10. Eilam, T., Gavoille, C., Peleg, D.: Compact routing schemes with low stretch factor. J. Algorithms 46(2), 97–114 (2003)
11. Fakcharoenphol, J., Rao, S., Talwar, K.: A tight bound on approximating arbitrary metrics by tree metrics. In: STOC 2003: Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing, pp. 448–455 (2003)
12. Gao, J., Guibas, L.J., Milosavljevic, N., Zhou, D.: Distributed resource management and matching in sensor networks. In: Proc. of the 8th International Symposium on Information Processing in Sensor Networks (IPSN 2009), pp. 97–108 (April 2009)
13. Gao, J., Zhang, L.: Tradeoffs between stretch factor and load balancing ratio in routing on growth restricted graphs. IEEE Transactions on Parallel and Distributed Computing 20(2), 171–179 (2009)

14. Gottlieb, L.-A., Roditty, L.: Improved algorithms for fully dynamic geometric spanners and geometric routing. In: SODA 2008: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (2008)
15. Gupta, A., Krauthgamer, R., Lee, J.R.: Bounded geometries, fractals, and low-distortion embeddings. In: FOCS 2003: Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 534–543 (2003)
16. Karger, D., Ruhl, M.: Find nearest neighbors in growth-restricted metrics. In: Proc. ACM Symposium on Theory of Computing, pp. 741–750 (2002)
17. Konjevod, G., Richa, A.W., Xia, D.: Optimal-stretch name-independent compact routing in doubling metrics. In: PODC 2006: Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, pp. 198–207 (2006)
18. Linial, N., London, E., Rabinovich, Y.: The geometry of graphs and some of its algorithmic applications. Combinatorica 15, 215–245 (1995)
19. Motiwala, M., Elmore, M., Feamster, N., Vempala, S.: Path splicing. SIGCOMM Comput. Commun. Rev. 38(4), 27–38 (2008)
20. Motskin, A., Downes, I., Kusy, B., Gnawali, O., Guibas, L.: Network Warehouses: Efficient Information Distribution to Mobile Users. In: Proc. of the 30th Annual IEEE Conference on Computer Communications (INFOCOM) (April 2011)
21. Neelesh Khanna, S.B.: Approximate shortest paths avoiding a failed vertex: Optimal size data structures for unweighted graphs. In: STACS, pp. 513–524 (2010)
22. Ng, E., Zhang, H.: Predicting Internet network distance with coordinates-based approaches. In: Proc. IEEE INFOCOM, pp. 170–179 (2002)
23. Peleg, D.: Distributed Computing: A Locality-Sensitive Approach. SIAM Monographs on Discrete Mathematics and Applications (2000)
24. Plaxton, C.G., Rajaraman, R., Richa, A.W.: Accessing nearby copies of replicated objects in a distributed environment. In: Proc. ACM Symposium on Parallel Algorithms and Architectures, pp. 311–320 (1997)
25. Raghavan, P., Thompson, C.D.: Provably good routing in graphs: regular arrays. In: Proceedings of the 17th Annual ACM Symposium on Theory of Computing, pp. 79–87 (1985)
26. Ray, K.-W.K.S., Guerin, R., Sofia, R.: Always acyclic distributed path computation. To appear in IEEE/ACM Transactions on Networking (2009)
27. Sarkar, R., Zhu, X., Gao, J.: Spatial distribution in routing table design for sensor networks. In: Proc. of the 28th Annual IEEE Conference on Computer Communications (INFOCOM 2009), Mini-Conference (April 2009)
28. Shand, M., Bryant, S.: Ip fast reroute framework. In: Internet Draft (June 2009)
29. Spring, N., Mahajan, R., Wetherall, D., Anderson, T.: Measuring isp topologies with rocketfuel. IEEE/ACM Trans. Netw. 12(1), 2–16 (2004)
30. Thorup, M., Zwick, U.: Compact routing schemes. In: SPAA 2001: Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 1–10 (2001)
31. Ohara, S.I.Y., Meter, R.V.: Mara: Maximum alternative routing algorithm. In: Proc. IEEE INFOCOM (2009)
32. Zhou, D., Gao, J.: Maintaining approximate minimum steiner tree and k-center for mobile agents in a sensor network. In: Proc. of the 29th Annual IEEE Conference on Computer Communications (INFOCOM 2010) (March 2010)

A Appendix

To prove Lemma 1, we first evaluate the probability that in one tree, say, H_1, the probability that u, v have a lowest common ancestor at level j, $1 \le j \le \delta$.

Lemma 2

$$\text{Prob}\{\text{LCA}_1(u, v) \text{ is at level } i + 1\}$$
$$\le \begin{cases} 0, & \text{if } 2^{i+1} < d(u, v); \\ k^2 \cdot d(u, v)/2^{i-2}, & \text{if } 2^{i-2} \ge d(u, v). \end{cases}$$

Proof. First, if $w = \text{LCA}_1(u, v)$ is at level $i + 1$, then $d(w, u) \le \beta_{i-1} \le 2^i$, $d(w, v) \le \beta_{i-1} \le 2^i$. By triangle inequality $d(u, v) \le d(u, w) + d(w, v) \le 2^{i+1}$. Thus in the first case of the lemma, the probability is 0. Suppose j^* is the smallest i such that $2^{i+2} \ge d(u, v)$. In the following we focus on the second case, i.e., $i \ge j^* + 4$.

If u, v belong to different clusters at level i, we say that the decomposition D_i *separates* u, v at level i. Thus $\text{LCA}_1(u, v)$ is at level $i + 1$ if and only if D_i separates u, v and $D_j(j > i)$ does not. Thus,

$$\text{Prob}\{\text{LCA}_1(u, v) \text{is at level } i + 1\} \le \text{Prob}\{D_i \text{ separates } (u, v)\}.$$

Take this level i such that D_i separates u, v. There is a node w such that one of u, v is first assigned to w and the other is not. We say that w *settles* the pair u, v at level i. Such a node w is unique, as once the pair u, v is settled it won't be settled again. Thus we will consider the union of the probability for each node w of P to possibly settle u, v. If w settles u, v and u is assigned to w, we say w *cuts* u out. Summarizing the above, we have $\text{Prob}\{D_i \text{ separates } (u, v)\} = \sum_w \text{Prob}\{w \text{ settles } u, v\} = \sum_w \text{Prob}\{w \text{ cuts } u \text{ out}\} + \sum_w \text{Prob}\{w \text{ cuts } v \text{ out}\}$.

Let K_i^u be the set of nodes in P within distance 2^i to node u, and let $k_i^u = |K_i^u|$. We rank the node in K_i^u with increasing order of distance from u: $w_1, w_2, \cdots, w_{k_i^u}$. For a node w_s to cut u out of the pair u, v at level i, it must satisfy the following conditions: (i) $d(u, w_s) \le \beta_i$. (ii) $d(v, w_s) > \beta_i$. (iii) w_s settles u, v. Thus β_i must lie in $[d(u, w_s), d(v, w_s)]$. But we have $d(v, w_s) \le d(v, u) + d(u, w_s)$ by triangle inequality. so the length of interval $[d(u, w_s), d(v, w_s)]$ is at most $d(u, v)$. Since we choose β_i uniformly from the range $[2^{i-1}, 2^i]$, the probability for β_i to fall into this interval is at most $d(u, v)/2^{i-1}$.

We also need to bound the probability that it is w_s that cut u out of the pair u, v, not others in K_i^u. First we note that the points that are very close to both u, v cannot possibly settle u, v. In fact, w_s must lie outside K_{i-2}^u for $i \ge j^* + 4$. Suppose otherwise, w_s is in K_{i-2}^u, and u is assigned to w_s, then v must be assigned to w_s too, by triangle inequality, $d(v, w_s) \le d(v, u) + d(u, w_s) \le 2^{i-2} + 2^{i-2} \le 2^{i-1} \le \beta_i$ (note that $i \ge j^* + 4$). Thus only those in $w_{k_{i-2}^u+1}, w_{k_{i-2}^u+2}, \cdots, w_{k_i^u}$ can separate u, v in level i. Since we have a random permutation on the node rank, the probability for w_s to be the first center assigned to u is at most $1/s$. Then the probability that u is cut out of the pair (u, v) at level i is bounded by $\sum_{s=k_{i-2}^u+1}^{k_i^u} \frac{1}{s} \cdot \frac{d(u, v)}{2^{i-1}} = \frac{d(u, v)}{2^{i-1}} \cdot (H_{k_i^u} - H_{k_{i-2}^u})$, where $H(m)$ is the harmonic function.

For a metric with expansion ratio k, we have $k_i^u \le k \cdot k_{i-1}^u \le k^2 \cdot k_{i-2}^u$. Then

$$H_{k_i^u} - H_{k_{i-2}^u} = \sum_{s=k_{i-2}^u+1}^{k_i^u} \frac{1}{s} < \sum_{s=k_{i-2}^u+1}^{k_i^u} \frac{1}{k_{i-2}^u} = \frac{k_i^u}{k_{i-2}^u} - 1 \le k^2.$$

Thus, we have $\text{Prob}\{D_i \text{ separates } (u,v)\} = d(u,v) \cdot \frac{k^2}{2^{i-2}}$, as required in the theorem.

Now we are ready to prove Lemma 1.

First, if $\text{LCA}(u,v)$ is at level $i+1$, then at least in one tree the lowest common ancestor is at level $i+1$, the probability of which is 0 if $d(u,v) < 2^{i+2}$, as shown in Lemma 2. In the following we focus on the second case when $2^{i-2} \ge d(u,v)$.

If $\text{LCA}(u,v)$ is at level $i+1$, the first time (smallest level) that u, v belong to different clusters is i in one tree and is $j \ge i$ in another tree. Denote by $P_1(i)$ and $P_2(i)$ the probability that $\text{LCA}_1(u,v)$ and $\text{LCA}_2(u,v)$ are at level $i+1$ respectively.

$$\begin{aligned}
&\text{Prob}\{\text{LCA}(u,v) \text{ is at level } i+1\} \\
&= P_1(i) \sum_{j=i+1}^{\delta} P_2(j) + P_2(i) \sum_{j=i+1}^{\delta} P_1(j) + P_1(i)P_2(i)
\end{aligned}$$

By using Lemma 2. Now we have

$$\begin{aligned}
&\text{Prob}\{\text{LCA}(u,v) \text{ is at level } i+1\} \\
&\le 2k^2 \tfrac{d(u,v)}{2^{i-2}} \sum_{j=i+1}^{\delta} [k^2 \tfrac{d(u,v)}{2^{j-2}}] + [k^2 \tfrac{d(u,v)}{2^{i-2}}][k^2 \tfrac{d(u,v)}{2^{i-2}}] \\
&= 3k^4 \cdot d^2(u,v)/2^{2i-4}.
\end{aligned}$$

Consistent Labeling of Rotating Maps

Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter

Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Germany

Abstract. Dynamic maps that allow continuous map rotations, e.g., on mobile devices, encounter new issues unseen in static map labeling before. We study the following dynamic map labeling problem: The input is a static, labeled map, i.e., a set P of points in the plane with attached non-overlapping horizontal rectangular labels. The goal is to find a *consistent* labeling of P under rotation that maximizes the number of visible labels for all rotation angles such that the labels remain horizontal while the map is rotated. A labeling is consistent if a single *active* interval of angles is selected for each label such that labels neither intersect each other nor occlude points in P at any rotation angle.

We first introduce a general model for labeling rotating maps and derive basic geometric properties of consistent solutions. We show NP-completeness of the active interval maximization problem even for unit-square labels. We then present a constant-factor approximation for this problem based on line stabbing, and refine it further into an EPTAS. Finally, we extend the EPTAS to the more general setting of rectangular labels of bounded size and aspect ratio.

1 Introduction

Dynamic maps, in which the user can navigate continuously through space, are becoming increasingly important in scientific and commercial GIS applications as well as in personal mapping applications. In particular GPS-equipped mobile devices offer various new possibilities for interactive, location-aware maps. A common principle in dynamic maps is that users can pan, rotate, and zoom the map view. Despite the popularity of several commercial and free applications, relatively little attention has been paid to provably good labeling algorithms for dynamic maps.

Been et al. [2] identified a set of consistency desiderata for dynamic map labeling. Labels should neither "jump" (suddenly change position or size) nor "pop" (appear and disappear more than once) during monotonous map navigation; moreover, the labeling should be a function of the selected map viewport and not depend on the user's navigation history. Previous work on the topic has focused solely on supporting zooming and/or panning of the map [2, 3, 12], whereas consistent labeling under map rotations has not been considered prior to this paper.

Most maps come with a natural orientation (usually the northern direction facing upward), but applications such as car or pedestrian navigation often rotate the map view dynamically to be always forward facing [6]. Still, the labels must remain horizontally aligned for best readability regardless of the actual rotation angle of the map. A basic requirement in static and dynamic label placement is that labels are pairwise disjoint, i.e., in general not all labels can be placed simultaneously. For labeling point features,
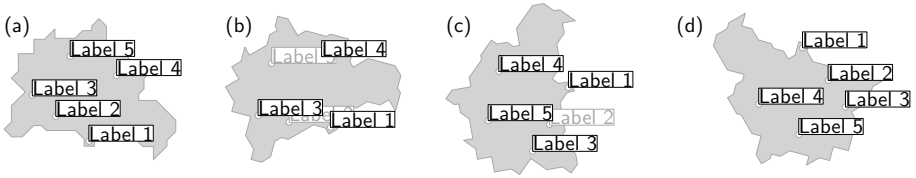
Fig. 1. Input map with five points (a) and three rotated views with some partially occluded labels (b)–(d)

it is further required that each label, usually modeled as a rectangle, touches the labeled point on its boundary. It is often not allowed that labels occlude the input point of another label. Figure 1 shows an example of a map that is rotated and labeled. The objective in map labeling is usually to place as many labels as possible. Translating this into the context of rotating maps means that, integrated over one full rotation from 0 to 2π, we want to maximize the number of visible labels. The consistency requirements of Been et al. [2] can immediately be applied for rotating maps.

Our Results. Initially, we define a model for rotating maps and show some basic properties of the different types of conflicts that may arise during rotation. Next, we prove that consistently labeling rotating maps is NP-complete, for the maximization of the total number of visible labels in one full rotation and NP-hard for the maximization of the visibility range of the least visible label. Finally, we present a new 1/4-approximation algorithm and an efficient polynomial-time approximation scheme (EPTAS) for unit-height rectangles. A PTAS is called *efficient* if its running time is $O(f(\varepsilon) \cdot \text{poly}(n))$. Both algorithms can be extended to the case of rectangular labels with the property that the ratio of the smallest and largest width, the ratio of the smallest and largest height, as well as the aspect ratio of every label is bounded by a constant, even if we allow the anchor point of each label to be an arbitrary point of the label. This applies to most practical scenarios where labels typically consist of few and relatively short lines of text.

Related Work. Most previous algorithmic research efforts on automated label placement cover *static* labeling models for point, line, or area features. For static point labeling, fixed-position models and slider models have been introduced [4,9], in which the label, represented by its bounding box, needs to touch the labeled point along its boundary. The label number maximization problem is NP-hard even for the simplest labeling models, whereas there are efficient algorithms for the decision problem that asks whether all points can be labeled in some of the simpler models (see, e.g., the discussion by Klau and Mutzel [8]). Approximation results [1,9], heuristics [14], and exact approaches [8] are known for many variants of the static label number maximization problem.

   In recent years, *dynamic* map labeling has emerged as a new research topic that gives rise to many unsolved algorithmic problems. Petzold et al. [13] used a preprocessing step to generate a reactive conflict graph that represents possible label overlaps for maps of all scales. For any fixed scale and map region, their method computes a conflict-free labeling using heuristics. Mote [11] presents another fast heuristic method

for dynamic conflict resolution in label placement that does not require preprocessing. The consistency desiderata of Been et al. [2] for dynamic labeling (no popping and jumping effects when panning and zooming), however, are not satisfied by either of the methods. Been et al. [3] showed NP-hardness of the label number maximization problem in the consistent labeling model and presented several approximation algorithms for the problem. Nöllenburg et al. [12] recently studied a dynamic version of the alternative boundary labeling model, in which labels are placed at the sides of the map and connected to their points by leaders. They presented an algorithm to precompute a data structure that represents an optimal one-sided labeling for all possible scales and thus allows continuous zooming and panning. None of the existing dynamic map labeling approaches supports map rotation.

2 Model

In this section we describe a general model for rotating maps with axis-aligned rectangular labels. Let M be a labeled input map, i.e., a set $P = \{p_1, \ldots, p_n\}$ of points in the plane together with a set $L = \{\ell_1, \ldots, \ell_n\}$ of pairwise disjoint, closed, and axis-aligned rectangular labels, where each point p_i is a point on the boundary $\partial \ell_i$ of its label ℓ_i. We say ℓ_i is *anchored* at p_i. As M is rotated, each label ℓ_i in L remains horizontally aligned and anchored at p_i. Thus, label intersections form and disappear during rotation of M. We take the following alternative perspective on the rotation of M. Rather than rotating the points, say clockwise, and keeping labels horizontal we may instead rotate each label around its anchor point counterclockwise and keep the set of points fixed. It is easy to see that both rotations are equivalent and yield exactly the same results.

A *rotation* of L is defined by a rotation angle $\alpha \in [0, 2\pi)$; a *rotation labeling* of M is a function $\phi \colon L \times [0, 2\pi) \to \{0, 1\}$ such that $\phi(\ell, \alpha) = 1$ if label ℓ is visible or *active* in the rotation of L by α, and $\phi(\ell, \alpha) = 0$ otherwise. We call a labeling ϕ *valid* if, for any rotation α, the set of labels $L(\alpha) = \{\ell \in L \mid \phi(\ell, \alpha) = 1\}$ consists of pairwise disjoint labels and no label in $L(\alpha)$ contains any point in P (other than its anchor point). We note that a valid labeling is not yet consistent in terms of the definition of Been et al. [2, 3]: given fixed anchor points, labels clearly do not jump and the labeling is independent of the rotation history, but labels may still *pop* during a full rotation from 0 to 2π, i.e., appear and disappear more than once. In order to avoid popping effects, each label may be active only in a single contiguous range of $[0, 2\pi)$, where ranges are circular ranges modulo 2π so that they may span the input rotation $\alpha = 0$. A valid labeling ϕ, in which for every label ℓ the set $A_\phi(\ell) = \{\alpha \in [0, 2\pi) \mid \phi(\ell, \alpha) = 1\}$ is a contiguous range modulo 2π, is called a *consistent* labeling. For a consistent labeling ϕ the set $A_\phi(\ell)$ is called the *active range* of ℓ. The *length* $|A_\phi(\ell)|$ of an active range $A_\phi(\ell)$ is defined as the length of the circular arc $\{(\cos \alpha, \sin \alpha) \mid \alpha \in A_\phi(\ell)\}$ on the unit circle.

The objective in static map labeling is usually to find a maximum subset of pairwise disjoint labels, i.e., to label as many points as possible. Generalizing this objective to rotating maps means that integrated over all rotations $\alpha \in [0, 2\pi)$ we want to display as many labels as possible. This corresponds to maximizing the sum $\sum_{\ell \in L} |A_\phi(\ell)|$ over all consistent labelings ϕ of M; we call this optimization problem MAXTOTAL. An alternative objective is to maximize over all consistent labelings ϕ the minimum length $\min_\ell |A_\phi(\ell)|$ of all active ranges; this problem is called MAXMIN.
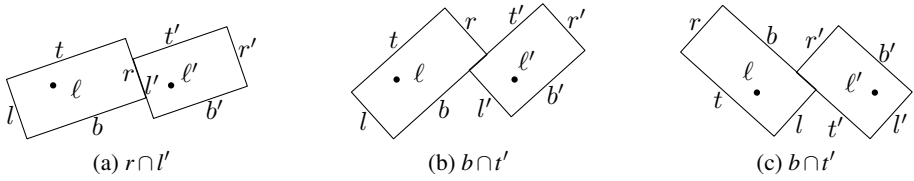
Fig. 2. Two labels ℓ and ℓ' and three of their eight possible boundary intersection events. Anchor points are marked as black dots.

3   Properties of Consistent Labelings

In this section we show basic properties of consistent labelings. If two labels ℓ and ℓ' intersect in a rotation of α they have a (regular) *conflict* at α, i.e., in a consistent labeling at most one of them can be active at α. The set $C(\ell,\ell') = \{\alpha \in [0,2\pi) \mid \ell \text{ and } \ell' \text{ are in conflict at } \alpha\}$ is called the *conflict set* of ℓ and ℓ'.

We show the following lemma in a more general model, in which the anchor point p of a label ℓ can be *any* point within ℓ and not necessarily a point on the boundary $\partial\ell$.

Lemma 1. *For any two labels ℓ and ℓ' with anchor points $p \in \ell$ and $p' \in \ell'$ the set $C(\ell,\ell')$ consists of at most four disjoint contiguous conflict ranges.*

Proof. The first observation is that due to the simultaneous rotation of all initially axis-parallel labels in L, ℓ and ℓ' remain "parallel" at any rotation angle α. Rotation is a continuous movement and hence any maximal contiguous conflict range in $C(\ell,\ell')$ must be a closed "interval" $[\alpha,\beta]$, where $0 \le \alpha, \beta < 2\pi$. Here we explicitly allow $\alpha > \beta$ by defining, in that case, $[\alpha,\beta] = [\alpha,2\pi) \cup [0,\beta]$. At a rotation of α (resp. β) the two labels ℓ and ℓ' intersect only on their boundary. Let l,r,t,b be the left, right, top, and bottom sides of ℓ and let l',r',t',b' be the left, right, top, and bottom sides of ℓ' (defined at a rotation of 0). Since ℓ and ℓ' are parallel, the only possible cases, in which they intersect on their boundary but not in their interior are $t \cap b'$, $b \cap t'$, $l \cap r'$, and $r \cap l'$. Each of those four cases may appear twice, once for each pair of opposite corners contained in the intersection. Figure 2 illustrates three of these eight boundary intersection events. Each of the conflicts defines a unique rotation angle and obviously at most four disjoint conflict ranges can be defined with these eight rotation angles as their endpoints. □

In the following we look more closely at the conditions under which the boundary intersection events (also called *conflict events*) occur and at the rotation angles defining them. Let h_t and h_b be the distances from p to t and b, respectively. Similarly, let w_l and w_b be the distances from p to l and r, respectively (see Figure 3). By h'_t, h'_b, w'_l, and w'_r we denote the corresponding values for label ℓ'. Finally, let d be the distance of the two anchor points p and p'. To improve readability of the following lemmas we define two functions $f_d(x) = \arcsin(x/d)$ and $g_d(x) = \arccos(x/d)$.

Lemma 2. *Let ℓ and ℓ' be two labels anchored at points p and p'. Then the conflict events in $C(\ell,\ell')$ are a subset of $\mathcal{C} = \{2\pi - f_d(h_t + h'_b), \pi + f_d(h_t + h'_b), f_d(h_b + h'_t), \pi - f_d(h_b + h'_t), 2\pi - g_d(w_r + w'_l), g_d(w_r + w'_l), \pi - g_d(w_l + w'_r), \pi + g_d(w_l + w'_r)\}$.*
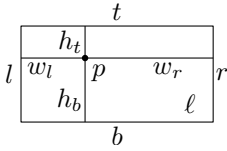
Fig. 3. Parameters of label ℓ anchored at p

(a) rotation of $2\pi - \alpha$
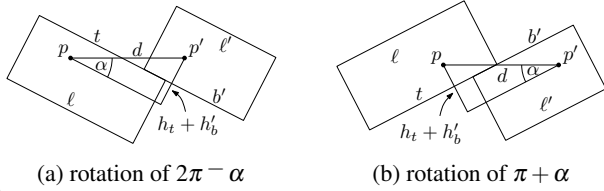
(b) rotation of $\pi + \alpha$

Fig. 4. Boundary intersection events for $t \cap b'$

Proof. Assume without loss of generality that p and p' lie on a horizontal line. First we show that the possible conflict events are precisely the rotation angles in $\mathcal{C}$. We start considering the intersection of the two sides t and b'. If there is a rotation angle under which t and b' intersect then we have the situation depicted in Figure 4 and by simple trigonometric reasoning the two rotation angles at which the conflict events occur are $2\pi - \arcsin((h_t + h_b')/d)$ and $\pi + \arcsin((h_t + h_b')/d)$. Obviously, we need $d \geq h_t + h_b'$. Furthermore, for the intersection in Figure 4a to be non-empty, we need $d^2 \leq (w_r + w_l')^2 + (h_t + h_b')^2$; similarly, for the intersection in Figure 4b, we need $d^2 \leq (w_l + w_r')^2 + (h_t + h_b')^2$.

From an analogous argument we obtain that the rotation angles under which b and t' intersect are $\arcsin((h_b + h_t')/d)$ and $\pi - \arcsin((h_b + h_t')/d)$. Clearly, we need $d \geq h_b + h_t'$. Furthermore, we need $d^2 \leq (w_r + w_l')^2 + (h_b + h_t')^2$ for the first intersection and $d^2 \leq (w_l + w_r')^2 + (h_b + h_t')^2$ for the second intersection to be non-empty under the above rotations.

The next case is the intersection of the two sides r and l'. Here the two rotation angles at which the conflict events occur are $2\pi - \arccos((w_r + w_l')/d)$ and $\arccos((w_r + w_l')/d)$. For the first conflict event we need $d^2 \leq (w_r + w_l')^2 + (h_t + h_b')^2$, and for the second we need $d^2 \leq (w_r + w_l')^2 + (h_b + h_t')^2$. For each of the intersections to be non-empty we additionally require that $d \geq w_r + w_l'$.

Similar reasoning for the final conflict events of $l \cap r'$ yields the rotation angles $\pi - \arccos((w_l + w_r')/d)$ and $\pi + \arccos((w_l + w_r')/d)$. The additional constraints are $d \geq w_l + w_r'$ for both events and $d^2 \leq (w_l + w_r')^2 + (h_b + h_t')^2$ for the first intersection and $d^2 \leq (w_l + w_r')^2 + (h_t + h_b')^2$. Thus, $\mathcal{C}$ contains all possible conflict events. $\qquad\square$

One of the requirements for a valid labeling is that no label may contain a point in P other than its anchor point. For each label ℓ this gives rise to a special class of conflict ranges, called *hard* conflict ranges, in which ℓ may never be active. The rotation angles at which hard conflicts start or end are called *hard* conflict events. Every angle that is a (hard) conflict event is called a *label event*. Obviously, every hard conflict is also a regular conflict. Regular conflicts that are not hard conflicts are also called *soft* conflicts. We note that by definition regular conflicts are symmetric, i.e., $C(\ell, \ell') = C(\ell', \ell)$, whereas hard conflicts are not symmetric. The next lemma characterizes the hard conflict ranges.

Lemma 3. *For a label ℓ anchored at point p and a point $q \neq p$ in P, the hard conflict events of ℓ and q are a subset of $\mathcal{H} = \{2\pi - f_d(h_t), \pi + f_d(h_t), f_d(h_b), \pi - f_d(h_b), 2\pi - g_d(w_r), g_d(w_r), \pi - g_d(w_l), \pi + g_d(w_l)\}$.*

Proof. We define a label of width and height 0 for q, i.e., we set $h'_t = h'_b = w'_l = w'_r = 0$. Then the result follows immediately from Lemma 2.   □

A simple way to visualize conflict ranges and hard conflict ranges is to mark, for each label ℓ anchored at p and each of its (hard) conflict ranges, the circular arcs on the circle centered at p and enclosing ℓ. Figure 5 shows an example.

In the following we show that the MAXTOTAL problem can be discretized in the sense that there exists an optimal solution whose active ranges are defined as intervals whose borders are label events. An active range *border* of a label ℓ is an angle α that is characterized by the property that the labeling ϕ is not constant in any ε-neighborhood of α. We call an active range where both borders are label events a *regular* active range.
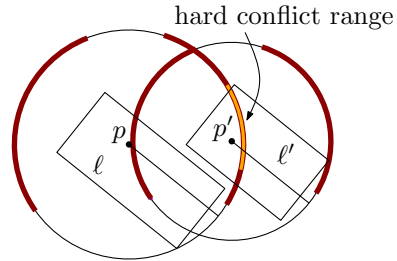

Fig. 5. Conflict ranges of two labels ℓ and ℓ' marked in bold on the enclosing circles

Lemma 4. *Given a labeled map M there is an optimal rotation labeling of M consisting of only regular active ranges.*

Proof. Let ϕ be an optimal labeling with a minimum number of active range borders that are no label events. Assume that there is at least one active range border β that is no label event. Let α and γ be the two adjacent active range borders of β, i.e., $\alpha < \beta < \gamma$, where α and γ are active range borders, but not necessarily label events. Then let L_l be the set of labels whose active ranges have left border β and let L_r be the set of labels whose active ranges have right border β. For ϕ to be optimal L_l and L_r must have the same cardinality since otherwise we could increase the active ranges of the larger set and decrease the active ranges of the smaller set by an $\varepsilon > 0$ and obtain a better labeling.

So define a new labeling ϕ' that is equal to ϕ except for the labels in L_l and L_r: define the left border of the active ranges of all labels in L_l and the right border of the active ranges of all labels in L_r as γ instead of β. Since $|L_l| = |L_r|$ we shrink and grow an equal number of active ranges by the same amount. Thus the two labelings ϕ and ϕ' have the same objective value $\sum_{\ell \in L} |A_\phi(\ell)| = \sum_{\ell \in L} |A_{\phi'}(\ell)|$. Because ϕ' uses as active range borders one non-label event less than ϕ this number was not minimum in ϕ—a contradiction. As a consequence ϕ has only label events as active range borders.   □

4   Complexity

In this section we show that finding an optimal solution for MAXTOTAL (and also MAXMIN) is NP-hard even if all labels are unit squares and their anchor points are their lower-left corners. We present a gadget proof reducing from the NP-complete problem planar 3-SAT [10]. Proofs of the lemmas in this section are found in the full version of the paper [5]. Before constructing the gadgets, we show a special property of unit-square labels.
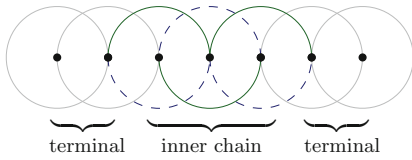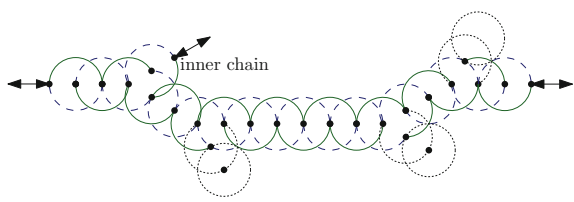
(a) A chain whose two different states are marked as full green and dashed blue arcs.
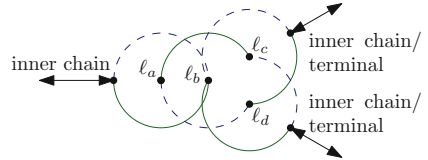
(b) A turn that splits one inner chain into two inner chains.
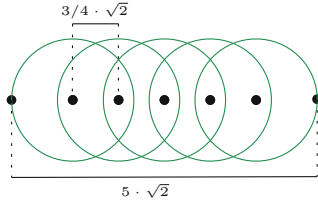
(c) Inverter.

(d) Literal Reader.

Fig. 6. Basic Building Blocks

Lemma 5. *If two unit-square labels ℓ and ℓ' whose anchor points are their lower-left corners have a conflict at a rotation angle α, then they have conflicts at all angles $\alpha + i \cdot \pi/2$ for $i \in \mathbb{Z}$.*

For every label ℓ we define the *outer circle* of ℓ as the circle of radius $\sqrt{2}$ centered at the anchor point of ℓ. Since the top-right corner of ℓ traces the outer circle we will use the locus of that corner to visualize active ranges or conflict ranges on the outer circle. Note that due to the fact that at the initial rotation of 0 the diagonal from the anchor point to the top-right corner of ℓ forms an angle of $\pi/4$ all marked ranges are actually offset by $\pi/4$.

4.1   Basic Building Blocks

Chain. A *chain* consists of at least four labels anchored at collinear points that are evenly spaced with distance $\sqrt{2}$. Hence, each point is placed on the outer circles of its neighbors. We call the first and last two labels of a chain *terminals* and the remaining part *inner chain*, see Figure 6a. We denote an assignment of active ranges to the labels as the *state* of the chain. The important observation is that in any optimal solution of MAXTOTAL an inner chain has only two different states, whereas terminals have multiple optimal states that are all equivalent for our purposes; see Figure 6a. In particular, in an optimal solution each label of an inner chain has an active range of length π and active ranges alternate between adjacent labels. We will use the two states of chains as a way to encode truth values in our reduction.

Lemma 6. *In any optimal solution, any label of an inner chain has an active range of length π. The active ranges of consecutive labels alternate between $(0, \pi)$ and $(\pi, 2\pi)$.*

Inverter. The second basic building block is an *inverter*. It consists of five collinear labels that are evenly spaced with distance $3/4 \cdot \sqrt{2}$ as depicted in Figure 6c. This means that the five labels together take up the same space as four labels in a usual inner chain. Similar to Lemma 6 the active ranges in an optimal solution also alternate. By replacing four labels of an inner chain with an inverter we can alter the parity of an inner chain.

Turn. The third building block is a *turn* that consists of four labels, see Figure 6b. The anchor points p_a and p_b are at distance $\sqrt{2}$ and the pairwise distances between p_b, p_c, and p_d are also $\sqrt{2}$ such that the whole structure is symmetric with respect to the line through p_a and p_b. The central point p_b is called *turn point*, and the two points p_c and p_d are called *outgoing points*. Due to the hard conflicts created by the four points we observe that the outer circle of p_b is divided into two ranges of length $5\pi/6$ and one range of length $\pi/3$. The outer circles of the outgoing points are divided into ranges of length π, $2\pi/3$, and $\pi/3$. The outer circle of p_a is divided into two ranges of length π. The outgoing points serve as connectors to terminals, inner chains, or further turns. Note, by coupling multiple turns we can divert an inner chain by any multiple of $30°$.

Lemma 7. *A turn has only two optimal states and allows to split an inner chain into two equivalent parts in an optimal solution.*

4.2   Gadgets of the Reduction

Variable Gadget. The variable gadget consists of an alternating sequence of two building blocks: horizontal chains and *literal readers*. A literal reader is a structure that allows us to split the truth value of a variable into one part running towards a clause and the part that continues the variable gadget, see Figure 6d. The literal reader consists of four turns, the first of which connects to a literal pipe and the other three are dummy turns needed to lead the variable gadget back to our grid. Note that some of the distances between anchor points in the literal reader need to be slightly less than $\sqrt{2}$ in order to reach a grid point at the end of the structure.

In order to encode truth values we define the state in which the first label of the first horizontal chain has active range $(0, \pi)$ as *true* and the state with active range $(\pi, 2\pi)$ as *false*.

Clause Gadget. The clause gadget consists of one inner and three outer labels, where the anchor points of the outer labels split the outer circle of the inner label into three equal parts of length $2\pi/3$, see Figure 7. Each outer label further connects to an incoming literal pipe and a terminal. These two connector labels are placed so that the outer circle of the outer label is split into two ranges of length $3\pi/4$ and one range of length $\pi/2$.

The general idea behind the clause gadget is as follows. The inner label obviously cannot have an active range larger than $2\pi/3$. Each outer label is placed in such a way that if it carries the value *false* it has a soft conflict with the inner label in one of the three possible active ranges of length $2\pi/3$. Hence, if all three labels transmit the value *false* then every possible active range of the inner label of length $2\pi/3$ is affected by a soft conflict. Consequently, its active range can be at most $\pi/2$.
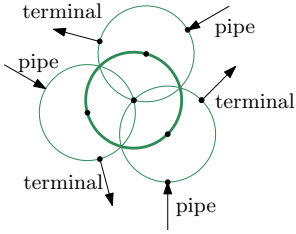
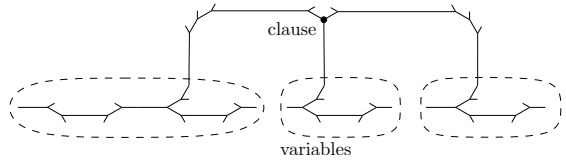**Fig. 7.** Clause gadget with one inner and three outer labels

Fig. 8. Sketch of the gadget placement for the reduction

On the other hand, if at least one of the pipes transmits *true*, the inner label can be assigned an active range of maximum length $2\pi/3$.

Lemma 8. *There must be a label in a clause or one of the incoming pipes with an active range of length at most $\pi/2$ if and only if all three literals of that clause evaluate to* false.

Pipes. Pipes propagate truth values of variable gadgets to clause gadgets. We use three different types of pipes, which we call *left arm*, *middle arm*, and *right arm*, depending on where the pipe attaches to the clause.

One end of each pipe attaches to a variable at the open outgoing label of a literal reader. Initially, the pipe leaves the variable gadget at an angle of $30°$. By using sequences of turns, we can route the pipes at any angle that is an integer multiple of $30°$. Thus we can make sure that for a clause above the variables the left arm enters the clause gadget at an angle of $150°$, the middle arm at an angle of $270°$, and the right arm at an angle of $30°$ with respect to the positive *x*-axis. For clauses below the variables the pipes are mirrored.

In order to transmit the correct truth value into the clause we first need to place the literal reader such that the turn point of the first turn corresponds to an even position in the variable chain. Next, for a positive literal we need a pipe of even length, whereas for a negative literal the pipe must have odd length. Note that we can always achieve the correct parity by making use of the inverter gadgets.

Gadget Placement. We place all variable gadgets on the same *y*-coordinate such that each anchor point of variable labels (except for literal readers) lies on integer *x*- and *y*-coordinates with respect to a grid of width and height $\sqrt{2}$. Clause gadgets and pipes lie below and above the variables and form three-legged "combs". The overall structure of the gadget arrangement is sketched in Figure 8.

Theorem 1. MAXTOTAL *is NP-complete.*

Proof. For a given planar 3-SAT formula φ we construct the MAXTOTAL instance as described above. For this instance we can compute the maximum possible sum K of active ranges assuming that each clause is satisfiable. By Lemma 8 every unsatisfied clause forces one label to have an active range of only $\pi/2$. Thus we know that φ is

satisfiable if and only if the MAXTOTAL instance has a total active range sum of at least K. Constructing and placing the gadgets can be done in polynomial time and space.

Due to Lemma 4 we can discretize the MAXTOTAL problem. Thus we can construct an oracle that guesses an active range assignment, which we can verify in polynomial time. So MAXTOTAL is in $\mathcal{NP}$.   □

We note that the same construction as for the NP-hardness of MAXTOTAL can also be applied to prove NP-hardness of MAXMIN. The maximally achievable minimum length of an active range for a satisfiable formula is $2\pi/3$, whereas for an unsatisfiable formula it is $\pi/2$ due to Lemma 8. This observation also yields that MAXMIN cannot be efficiently approximated within a factor of $3/4$.

Corollary 1. MAXMIN *is NP-hard and it has no efficient approximation algorithm with an approximation factor larger than* $3/4$ *unless* $\mathcal{P} = \mathcal{NP}$.

5   Approximation Algorithms

In the previous section we have established that MAXTOTAL is NP-complete. Unless $\mathcal{P} = \mathcal{NP}$ we cannot hope for an efficient exact algorithm to solve the problem. In the following we devise a $1/4$-approximation algorithm for MAXTOTAL and refine it to an EPTAS. For both algorithms we initially assume that labels are congruent unit-height rectangles with constant width $w \geq 1$ and that the anchor points are the lower-left corners of the labels. Let d be the length of the label's diagonal, i.e., $d = \sqrt{w^2 + 1}$.

Before we describe the algorithms we state four important properties that apply even to the more general labeling model, where anchor points are arbitrary points within the label or on its boundary, and where the ratio of the smallest and largest width and height, as well as the aspect ratio are bounded by constants: i) the number of anchor points contained in a square is proportional to its area, ii) the number of conflicts a label can have with other labels is bounded by a constant, iii) any two conflicting labels produce only $O(1)$ conflict regions, and finally, iv) there is an optimal MAXTOTAL solution where the borders of all active ranges are events.

Properties (i) and (ii) can easily be proved with a simple packing argument (see full version of the paper [5] for details). Property (iii) follows from property (ii) and Lemma 1. Property (iv) follows immediately from Lemma 4.

5.1   A $1/4$-Approximation for MAXTOTAL

The basis for our algorithm is the *line stabbing* or *shifting* technique by Hochbaum and Maass [7], which has been applied before to *static* labeling problems for (non-rotating) unit-height labels [1, 9]. Consider a grid G where each grid cell is a square with side length $2d$. We can address every grid cell by its row and column index. Now we can partition G into four subsets by deleting every other row and every other column with either even or odd parity. Within each of these subsets we have the property that any two grid cells have a distance of at least $2d$. Thus no two labels whose anchor points lie in different cells of the same subset can have a conflict. We say that a grid cell c *covers* a label ℓ if the anchor point of ℓ lies inside c. By property (i) only $O(1)$ labels are

covered by a single grid cell. Combining this with property (ii) we see that the number of conflicts of the labels covered by a single grid cell is constant. This implies that the number of events in that cell (cf. Lemma 4) is also constant.

The four different subsets of grid cells divide a MAXTOTAL instance into four subinstances, each of which decomposes into independent grid cells. If we solve all subsets optimally, at least one of the solutions is a 1/4-approximation for the initial instance due to the pigeon-hole principle.

Determining an optimal solution for the labels covered by a grid cell c works as follows. We compute, for the set of labels $L_c \subseteq L$ covered by c, the set E_c of label events. Due to Lemma 4 we know that there exists an optimal solution where all borders of active ranges are label events. Thus, to compute an optimal active range assignment for the labels in L_c we need to test all possible combinations of active ranges for all labels $\ell \in L_c$. For a single cell this requires only constant time.

We can precompute the non-empty grid cells by simple arithmetic operations on the coordinates of the anchor points and store those cells in a binary search tree. Since we have n anchor points there are at most n non-empty grid cells in the tree, and each of the cells holds a list of the covered anchor points. Building this data structure takes $O(n \log n)$ time and then optimally solving the active range assignment problem in the non-empty cells takes $O(n)$ time.

Theorem 2. *There exists an $O(n \log n)$-time algorithm that yields a $1/4$-approximation of* MAXTOTAL *for congruent unit-height rectangles with their lower-left corners as anchor points.*

5.2   An Efficient Polynomial-Time Approximation Scheme for MAXTOTAL

We extend the technique for the 1/4-approximation to achieve a $(1-\varepsilon)$-approximation. Let again G be a grid whose grid cells are squares of side length $2d$. For any integer k we can remove every k-th row and every k-th column of the grid cells, starting at two offsets i and j ($0 \leq i, j \leq k-1$). This yields collections of meta cells of side length $(k-1) \cdot 2d$ that are pairwise separated by a distance of at least $2d$ and thus independent. In total, we obtain k^2 such collections of meta cells.

For a given $\varepsilon \in (0,1)$ we set $k = \lceil 2/\varepsilon \rceil$. Let c be a meta cell for the given k and let again L_c be the set of labels covered by c, and E_c the set of label events for L_c. Then, by properties (i) and (ii), both $|L_c|$ and $|E_c|$ are $O(1/\varepsilon^2)$. Since we need to test all possible active ranges for all labels in L_c, it takes $O(2^{O(1/\varepsilon^2 \log 1/\varepsilon^2)})$ time to determine an optimal solution for the meta cell c.

For a given collection of disjoint meta cells we determine (as in Section 5.1) all $O(n)$ non-empty meta cells and store them in a binary search tree such that each cell holds a list of its covered anchor points. This requires again $O(n \log n)$ time. So for one collection of meta cells the time complexity for finding an optimal solution is $O(n 2^{O(1/\varepsilon^2 \log 1/\varepsilon^2)} + n \log n)$. There are k^2 such collections and, by the pigeon hole principle, the optimal solution for at least one of them is a $(1-\varepsilon)$-approximation of the original instance. This yields the following theorem.

Theorem 3. *There exists an EPTAS that computes a $(1-\varepsilon)$-approximation of* MAXTOTAL *for congruent unit-height rectangles with their lower-left corners as anchor points. Its time complexity is $O((n 2^{O(1/\varepsilon^2 \log 1/\varepsilon^2)} + n \log n)/\varepsilon^2)$.*

We note that this EPTAS basically relies on properties (i)–(iv) and that there is nothing special about congruent rectangles anchored at their lower-left corners. Hence we can generalize the algorithm to the more general labeling model, in which the ratio of the label heights, the ratio of the label widths, and the aspect ratios of all labels are bounded by constants. Furthermore, the anchor points are not required to be label corners; rather they can be any point on the boundary or in the interior of the labels. Finally, we can even ignore the distinction between hard and soft conflicts, i.e., allow that anchor points of non-active labels are occluded. Properties (i)–(iv) still hold in this general model. The only change in the EPTAS is to set the width and height of the grid cells to twice the maximum diameter of all labels in L.

Corollary 2. *There exists an EPTAS that computes a* $(1 - \varepsilon)$*-approximation of* MAX-TOTAL *in the general labeling model with rectangular labels of bounded height ratio, width ratio, and aspect ratio, whose anchor points are arbitrary points in the respective labels. The time complexity of the EPTAS is* $O((n2^{O(1/\varepsilon^2 \log 1/\varepsilon^2)} + n \log n)/\varepsilon^2)$.

References

1. Agarwal, P.K., van Kreveld, M., Suri, S.: Label placement by maximum independent set in rectangles. Comput. Geom. Theory Appl. 11, 209–218 (1998)
2. Been, K., Daiches, E., Yap, C.: Dynamic map labeling. IEEE Trans. Visual. and Comput. Graphics 12(5), 773–780 (2006)
3. Been, K., Nöllenburg, M., Poon, S.-H., Wolff, A.: Optimizing active ranges for consistent dynamic map labeling. Comput. Geom. Theory Appl. 43(3), 312–328 (2010)
4. Formann, M., Wagner, F.: A packing problem with applications to lettering of maps. In: Proc. 7th Annu. ACM Sympos. Comput. Geom. (SoCG 1991), pp. 281–288 (1991)
5. Gemsa, A., Nöllenburg, M., Rutter, I.: Consistent Labeling of Rotating Maps. ArXiv e-prints, arXiv:1104.5634 (April 2011)
6. Gervais, E., Nussbaum, D., Sack, J.-R.: DynaMap: a context aware dynamic map application. In: Proc. GISPlanet, Estoril, Lisbon, Portugal (2005)
7. Hochbaum, D.S., Maass, W.: Approximation schemes for covering and packing problems in image processing and VLSI. J. ACM 32(1), 130–136 (1985)
8. Klau, G.W., Mutzel, P.: Optimal labeling of point features in rectangular labeling models. Math. Programming (Series B), 435–458 (2003)
9. van Kreveld, M., Strijk, T., Wolff, A.: Point labeling with sliding labels. Comput. Geom. Theory Appl. 13, 21–47 (1999)
10. Lichtenstein, D.: Planar formulae and their uses. SIAM J. Comput. 11(2), 329–343 (1982)
11. Mote, K.D.: Fast point-feature label placement for dynamic visualizations. Inform. Visual. 6(4), 249–260 (2007)
12. Nöllenburg, M., Polishchuk, V., Sysikaski, M.: Dynamic one-sided boundary labeling. In: Proc. 18th ACM SIGSPATIAL Int'l Conf. Adv. Geo. Inform. Syst., pp. 310–319. ACM, New York (2010)
13. Petzold, I., Gröger, G., Plümer, L.: Fast screen map labeling—data-structures and algorithms. In: Proc. 23rd Int'l. Cartographic Conf. (ICC 2003), pp. 288–298 (2003)
14. Wagner, F., Wolff, A., Kapoor, V., Strijk, T.: Three rules suffice for good label placement. Algorithmica 30(2), 334–349 (2001)

Finding Longest Approximate Periodic Patterns

Beat Gfeller

IBM Research - Zurich, Switzerland
bgf@zurich.ibm.com

Abstract. Motivated by the task of finding approximate periodic patterns in real-world data, we consider the following problem: Given a sequence $\mathcal{S}$ of n numbers in increasing order, and $\alpha \in [0, 1]$, find a longest subsequence $\mathcal{T} = s_1, s_2, \ldots, s_k$ of numbers $s_i \in \mathcal{S}$, ordered as in $\mathcal{S}$, under the condition that $\max_{i=1,\ldots,k-1}\{s_{i+1} - s_i\} / \min_{i=1,\ldots,k-1}\{s_{i+1} - s_i\}$, called the *period ratio* of $\mathcal{T}$, is at most $1 + \alpha$. We give an exact algorithm with run time $O(n^3)$ for this problem. This bound is too high for large inputs in practice. Therefore, we describe an algorithm which approximates the longest periodic pattern present in the input in the following sense: Given constants α and ϵ, the algorithm computes a subsequence with period ratio at most $(1 + \alpha)(1 + \epsilon)$, whose length is greater or equal to the longest subsequence with period ratio at most $(1 + \alpha)$. This latter algorithm has a much smaller run time of $O(n^{1+\gamma})$, where $\gamma > 0$ is an arbitrarily small positive constant. As a byproduct which may be of independent interest, we show that an approximate variant of the well-known 3SUM problem can also be solved in $O(n^{1+\gamma} + T_{sort}(n))$ time, for any constant $\gamma > 0$, where $T_{sort}(n)$ is the time required to sort n numbers.

1 Introduction and Related Work

We study the problem of finding approximate arithmetic progressions in a sequence of numbers. This problem is motivated by the existence of many phenomena in the real world in which a particular type of event repeats periodically during a certain period of time. Examples of highly periodic events include road traffic peaks, load peaks on web servers, monitoring events in computer networks and many others. Finding periodic patterns in real-world data often leads to useful insights, because it sheds light on the structure of the data, and gives a basis to predict future events. Moreover, in some applications periodic patterns can point out a problem: In a computer network, for example, repeating error messages can indicate a misconfiguration, or even a security intrusion such as a port scan [4]. Note that such periodic patterns might only occur temporarily, and need not persist throughout the whole period of time covered by the event log. Since short (approximate) periodic patterns will appear in any sequence of random events that is long enough, a periodic pattern is more interesting the more repetitions it contains. Therefore, we are interested in finding a longest periodic pattern, which contains the largest number of repetitions of the same event.

In general, the input data we are given can be modeled as a sequence of events, each associated with a timestamp. Since we are only interested in the repetition of events of the same type, we can treat each type of event separately. Thus, the input consists of a sequence $\mathcal{S}$ of n distinct numbers $t_1, t_2, \ldots, t_n$ *in increasing order*, which are the times at which an event of a particular type has occurred. A periodic pattern then corresponds to an (approximate) arithmetic progression in this sequence. For a given sequence of numbers $s_1, s_2, \ldots, s_k$, $s_i \in \mathcal{S}$, we say that $\frac{\max_{i=1,\ldots,k-1}\{s_{i+1}-s_i\}}{\min_{j=1,\ldots,k-1}\{s_{j+1}-s_j\}}$ is the *period ratio* of the sequence, and that $\max_{i=1,\ldots,k-1}\{s_{i+1}-s_i\} - \min_{j=1,\ldots,k-1}\{s_{j+1}-s_j\}$ is its *period width*. We call the differences $s_{i+1} - s_i$ of consecutive numbers in the sequence *periods*. In most real-world applications, the timestamps which are given as input are imprecise. Hence, no exact arithmetic progression may be present in the input, and it is necessary to allow some slack in the periodic patterns.

If the approximate period of the patterns of interest are known, then the periodic patterns that we want to find can be formally defined as follows:

Definition 1 (absolute error periodic pattern). *Given $p_{\min} > 0$, $p_{\max} \geq p_{\min}$ and a sequence $\mathcal{S}$ of numbers in increasing order, an* absolute error periodic pattern *is a subsequence $s_1, s_2, \ldots, s_k$ of $\mathcal{S}$, such that the absolute difference between any two consecutive numbers s_i, s_{i+1} is at least $p_{\min}$ and at most $p_{\max}$.*

This type of pattern is useful if one is interested in a few particular periods: For example, in a sequence of log file entries, one might be interested only in events which occur (approximately) every hour, every day, every week, or every month. In this case, one can define an interval of acceptable distances for each period of interest: For hourly repetitions, for example, one could accept any distance between $p_{\min} = 55$ minutes and $p_{\max} = 65$ minutes.

For a given interval $[p_{\min}, p_{\max}]$, it is not difficult to compute the longest absolute error periodic pattern in linear time. We describe such an algorithm in Section 2, which forms the basis for our more involved algorithms.

Finding periodic patterns with arbitrary periods, where no period (or range of periods) of interest is given in advance, is more challenging. In this case, there are several choices for defining a valid pattern. We choose a definition which bounds the ratio between the longest and the shortest distance between consecutive events in the sequence. In contrast to an absolute error bound, this formalization has the natural property that it is scale invariant. Our definition is as follows:

Definition 2 (relative error periodic pattern). *Given $\alpha \in [0, 1]$ and a sequence $\mathcal{S}$ of numbers in increasing order, a* relative error periodic pattern *is a subsequence $s_1, s_2, \ldots, s_k$ of $\mathcal{S}$, for which there exists a number p such that the absolute difference between any two consecutive numbers in the subsequence is at least p and at most $p(1 + \alpha)$.*

This type of periodic pattern is the main focus of this paper. In Section 3.1, we give a simple $O(n^3)$ time algorithm to obtain, for a given α, the longest relative error pattern in a sequence of n numbers. This bound is too high for large inputs

in practice. Therefore, we explore approximate solutions, which approximate the longest periodic pattern that is present in the input in the following sense: Given constants α and ϵ, the algorithm computes a relative error periodic pattern which is at least as long as the longest one with period ratio up to $1+\alpha$, but the period ratio of the returned pattern can be up to $(1+\alpha)(1+\epsilon)$. In other words, we allow the period ratio of the output pattern to be slightly above $1+\alpha$. For example, if the given tolerance is $(1+\alpha) = 1.05$, we could set $\epsilon = 0.001$, and our algorithm would return a longest sequence with period ratio at most 1.05105. Assuming that the timestamps given as input are imprecise, one can choose ϵ small enough such that the increase of the period width caused by the algorithm is smaller than the error caused by the imprecision in the input. Thus, for practical purposes, the output of our approximate algorithm is almost as good as an exact solution. For this relaxed version of the problem, we are able to greatly reduce the run time to $O(n^{1+\gamma})$, for any constant $\gamma > 0$.

As a byproduct which may be of independent interest, our algorithm given in Section 3.3 can be modified to solve an approximate variant of the 3SUM problem in $O(n^{1+\gamma} + T_{sort}(n))$ time, for any constant $\gamma > 0$. We describe the details of these modifications in Section 4.

Related Work

Relatively few papers are closely related to this work. To the best of our knowledge, the problem of finding longest relative error periodic patterns has not been studied before. Ma and Hellerstein [4] also study the problem of finding approximate arithmetic sequences. However, they use an absolute tolerance for the variation of periods: In their definition, a pattern is valid only if its period width is at most δ. Moreover, their algorithms are heuristic, and cannot guarantee to find the longest pattern present in the input.

All maximal sequences of equally-spaced collinear points within a given point set can be found in $O(n^2)$ time [3]. It follows that the longest exact arithmetic progression in a sequence of numbers can also be found in $O(n^2)$ time. Motivated by an application in landmine detection, Robins et al. studied the problem of finding *approximate* sequences of equally-spaced collinear points in a given set of points in the plane [6], and gave an algorithm with run time $O(n^{5/2})$. Their notion of pattern is rather different from ours: In their definition, a pattern is valid only if there exists an imaginary exact sequence of equally-spaced collinear points, such that each point in the pattern is within a square of side-length ϵ of its corresponding imaginary point. Moreover, they postulate that any two points in the input have distance greater than 8ϵ.

Finding periodic patterns is a widely studied subject in the data mining community (see for example [2,5,7,8]). In most of these works, the input is given as a time series, where a measurement is taken at regular time intervals (say, hourly), and yields the set of events present at this moment. Periodic patterns have also been studied in bioinformatics (for example, tandem repeats in genomic sequences), where the input is a sequence of discrete symbols, and in astronomy (Lomb-Scargle periodograms), where the input is a real-valued timeseries. To

apply these previous methods to our problem, we would have to convert our input consisting of timestamped events into a sequence of discrete symbols, each indicating presense or absence of the event at a given time. However, this would vastly increase the input size, since the average time between consecutive events can be much larger than the granularity of the timestamps. The same reason prevents us from simply using the fast Fourier transform to solve our problem.

2   Optimal Algorithm for Absolute Error Periodic Patterns

In this section, we describe a linear time algorithm for computing the longest absolute error periodic pattern in a sequence S of numbers. We use dynamic programming: for each t_i in S, we compute the length of a longest absolute error periodic pattern which ends with t_i, denoted by $L[i]$. Note that the possible predecessors of t_i are all the numbers of S in $[t_i - p_{max}, t_i - p_{min}]$. A naive implementation using this idea would consider, for each t_i, all valid predecessors of t_i (i.e., all $t_j < t_i$ for which $t_i - t_j \in [p_{min}, p_{max}]$), and compose the longest periodic pattern ending at t_i by using the predecessor with the longest periodic pattern among these. This approach would have a run time of $O(n^2)$. In the following, we describe a solution with run time $O(n)$ which uses two additional observations:

Observation 1. *If we consider the numbers t_i for $i = 1, \ldots, n$, then the valid predecessors of t_i are contained in a sliding window which moves from left to right as we increase i. More precisely, let $pred_l(x)$ and $pred_r(x)$ denote the leftmost and rightmost valid predecessor of x. Then, for any $b > a$, we have: $pred_l(t_b) \geq pred_l(t_a)$ and $pred_r(t_b) \geq pred_r(t_a)$.*

Observation 2. *If two numbers t_a, t_b, with $a < b$, are both valid predecessors of the same t_i, and $L[a] \leq L[b]$, then the number t_a need not be considered as a predecessor of t_i, nor for any other number $t_j > t_i$. To see this, note that due to Observation 1, t_b will be a valid predecessor for any number t_j for which t_a is a valid predecessor.*

Our algorithm, which we call Algorithm 1, uses these two observations as follows: We go through all numbers from left to right, maintaining a list Pre of valid predecessors for the currently considered number t_i. This list only contains exactly those predecessors of t_i which cannot be ignored according to Observation 2. Hence, the predecessors in Pre are sorted in decreasing order of $L[\ ]$ *and* also ordered as in S. Therefore, the longest periodic pattern ending at t_i is always obtained using the first number t_q in this list as t_i's predecessor (if the list Pre is empty, the longest sequence ending at t_i consists only of t_i itself), and appending t_i to the longest periodic pattern ending at t_q. To update this list when moving from number t_i to t_{i+1}, we need to (i) delete the predecessors which were valid for t_i but no longer for t_{i+1} (unless they have already been removed due to Observation 2), and (ii) insert the numbers which are valid predecessors for t_{i+1} but were invalid for t_i. Before inserting such a number into

the list, we remove all predecessors which can be ignored because they are worse than the new solution (according to Observation 2). Note that since the list is sorted, all such entries can be found by traversing the list from right to left. We omit the pseudocode of this approach due to lack of space.

Lemma 1. *For any sequence $\mathcal{S}$ of n numbers, and bounds $p_{\min}$, $p_{\max}$, Algorithm 1 computes a longest absolute error periodic pattern in $\mathcal{S}$ in $O(n)$ time and space.*

Proof. The correctness of this algorithm follows from the above discussion. Moreover, its run time is linear in the input size because as we traverse $\mathcal{S}$ once from t_1 to t_n, every number t_i is inserted at the head of the list Pre at most once, and at most once deleted from its tail. The space complexity of the algorithm is also $O(n)$, since at any step in the computation, the list Pre contains at most n numbers.   □

3   Algorithms for Relative Error Periodic Patterns

This section is structured as follows. In Section 3.1, we describe an exact algorithm for finding a longest relative error periodic pattern. Then, we give a basic approximate algorithm for this problem in Section 3.2, on which we build in Section 3.3 to obtain a more efficient approximate algorithm.

3.1   An Exact Algorithm

A longest relative error periodic pattern with period ratio at most $1 + \alpha$ can be found using a relatively simple approach. There are at most $O(n^2)$ values for the smallest period $p_{\min}$ in the optimal pattern (all pairwise differences between the input numbers), so we can run Algorithm 1 for each of these values, using an upper bound of $p_{\max} := p_{\min}(1 + \alpha)$, to obtain the longest pattern with period ratio at most $1 + \alpha$. This approach clearly has run time $O(n^3)$.

3.2   A Basic Approximate Algorithm

We now describe an algorithm for finding relative error periodic patterns which is approximate in the following sense: Given constants α and ϵ, and an interval $[low, high]$ of periods to consider, it computes a relative error periodic pattern whose length is at least as long as the longest one with period ratio $1 + \alpha$, but the period ratio of the returned pattern can be up to $(1 + \alpha)(1 + \epsilon)$. In other words, the algorithm finds a solution comparable to the optimal one, except that the allowed relative error is increased by a fixed percentage. The run time of this algorithm is $O(n \log_{1+\epsilon}(high/low))$, and its space complexity is $O(n)$.

Basically, the idea of this algorithm, described in pseudocode in Algorithm 2, is to run Algorithm 1 for a sequence of different period intervals, whose period ratio is bounded by $(1 + \alpha)(1 + \epsilon)$. As a result, we find only periodic patterns whose period ratio is at most $(1 + \alpha)(1 + \epsilon)$. Moreover, the periodic pattern will be at least as long as the longest one with period ratio at most $1 + \alpha$:

Algorithm 2. relative error basic approximation algorithm

Input:

$\mathcal{S}$: the input numbers $t_1, \ldots, t_n$,

low, high: the (inclusive) bounds for the minimum and maximum period of
considered exact patterns,

α, and ϵ

1 for $i := 0, 1, 2, \ldots, \lceil \log_{1+\epsilon}(high/low) \rceil$ **do**

2 $\quad p_i := low(1+\epsilon)^i$

3 $\quad$ Run Algorithm 1 on $\mathcal{S}$ with $p_{\min} := p_i$, $p_{\max} := \min(p_i(1+\alpha)(1+\epsilon), high)$

4 return *the longest pattern found for any i*

Theorem 1. *Given constants $\alpha, \epsilon \geq 0$, and an interval $[low, high]$, suppose that
among sequences with minimum period $\geq low$ and maximum period $\leq high$, the
length of a longest relative error periodic pattern with period ratio at most $1+\alpha$ is
L. Algorithm 2 computes a relative error periodic pattern with length at least L,
whose period ratio is at most $(1+\alpha)(1+\epsilon)$, whose minimum period is at least low,
and whose maximum period is at most high. Its run time is $O(n \log_{1+\epsilon}(\frac{high}{low}))$,
and it requires $O(n)$ space.*

Proof. Consider any longest relative error periodic pattern $\mathcal{P}$ with period ra-
tio at most $1 + \alpha$, with all periods within $[low, high]$, and let $Pmin$ be its
minimum period, and $Pmax$ its maximum period. Consider the largest j for
which $p_j \leq Pmin$: Since $Pmax \leq Pmin(1 + \alpha)$, and $Pmin \leq p_j(1 + \epsilon)$, we
have $Pmax \leq p_j(1 + \alpha)(1 + \epsilon)$. Therefore, when we run Algorithm 1 for the
interval $[p_j, min(p_j(1 + \alpha)(1 + \epsilon), high)]$, the pattern $\mathcal{P}$ is contained in the al-
lowed period interval, and we will find a pattern at least as long as $\mathcal{P}$. By
construction, the period ratio of this pattern is at most $(1+\alpha)(1+\epsilon)$. Moreover,
its minimum period is at least $p_j \geq low$, and its maximum period is at most
$min(p_j(1+\alpha)(1+\epsilon), high) \leq high$. The time and space bounds are obvious. $\square$

Corollary 1. *Let $d_{\min}$ and $d_{\max}$ be the largest and the smallest difference be-
tween any two input numbers (note that these values are easy to compute in
$O(n)$ time). Running Algorithm 2 with $low := d_{\min}$ and $high := d_{\max}$ yields an
approximate relative error pattern in $O(n \log_{1+\epsilon}(\frac{d_{\max}}{d_{\min}}))$ time and $O(n)$ space.*

Note that if the ratio between the largest and the smallest difference of any
two numbers in the input is polynomially bounded, Algorithm 2 yields an ap-
proximate solution in $O(n \log n)$ time. In the following section, we describe an
algorithm which is efficient regardless of the ratio between smallest and largest
distance.

3.3   An Approximate $O(n^{1+\gamma})$ Time Algorithm

In this section, we describe a recursive algorithm for finding approximate relative
error patterns, which achieves a run time of $O(n^{1+\gamma})$, for any $\gamma > 0$.

Our approach is based on Algorithm 2, but we carefully select the period
intervals for which the algorithm is run, and not always apply the algorithm to

the entire input, but only to selected parts of it. To achieve this, we combine the two following basic observations:

- If there are two consecutive input numbers t_i and t_{i+1} whose absolute difference is x, then any pattern which contains at least one number from $t_1, \ldots, t_i$ *and* at least one number from $t_{i+1}, \ldots, t_n$ has a maximum period of at least x. Hence, we can find all periodic patterns whose maximum period is smaller than x by searching separately in the two parts $t_1, \ldots, t_i$ and $t_{i+1}, \ldots, t_n$.
- If some numbers $t_i, \ldots, t_j$ are all within an interval of width y (i.e., $t_j - t_i \leq y$), then any periodic pattern with minimum period at least y/ϵ contains at most one number from $t_i, \ldots, t_j$. Moreover, replacing such a number by any other from $t_i, \ldots, t_j$ will only increase the period ratio of such a pattern by a factor of at most $\frac{1+2\epsilon}{1-2\epsilon}$ (see Lemma 2). Hence, when searching for periodic patterns with minimum period at least y/ϵ, one can omit some numbers in the input if their mutual distance is smaller than y.

Lemma 2 (proof omitted). *Consider a periodic pattern $\mathcal{P}$, with minimum period $Pmin$ and maximum period $Pmax$. If we change each number in $\mathcal{P}$ by an absolute value of at most $\epsilon \cdot Pmin$, where $\epsilon < 1/2$, then we obtain a pattern $\mathcal{H}$ whose minimum period is $Pmin' \geq Pmin \cdot (1 - 2\epsilon)$, whose maximum period is $Pmax' \leq Pmax \cdot (1 + 2\epsilon)$, and whose period ratio is at most $\frac{Pmax}{Pmin} \frac{1+2\epsilon}{1-2\epsilon}$.*

Algorithm description. We now describe our algorithm, and give some intuition on how it works. For a detailed description of the algorithm in pseudocode, see Algorithm 3. For an overview of terms used to describe our algorithm, see Fig. 1. First, we select a number t' which separates the input into two parts, where the first part $t_1, \ldots, t'$ contains a constant fraction of the input numbers. To that end, we set $t' := t_{\lceil nq \rceil}$, where $q \in (0, 1)$ is a parameter that we choose later. We select a second number $t'' := t_{\lceil n(1-q) \rceil}$, such that $t'', \ldots, t_n$ also contains a constant fraction of the input numbers. To use both of the above observations, we consider $d := t'' - t'$: It is clear that there exist two consecutive numbers p', p'' in $t', \ldots, t''$ whose distance is at least $d/(n-1)$. For patterns with maximum period smaller than $d/(n-1)$, we can therefore search separately in $t_1, \ldots, p'$ and $p'', \ldots, t_n$. Furthermore, note that $t', \ldots, t''$ contains a constant fraction of all input numbers, all within an interval of length d. Thus, our second observation applies: For patterns with minimum period at least d/ϵ, we can omit all the numbers between t' and t'', because any such pattern contains at most one number from $t', \ldots, t''$, and replacing this number either by t' or by t'' will only increase the period ratio of the pattern by at most a factor of $\frac{1+2\epsilon}{1-2\epsilon}$. Thus, it suffices to search for these patterns in the sequence $t_1, \ldots, t', t'', \ldots, t_n$. It remains to search for patterns whose maximum period is at least $d/(n-1)$ and whose minimum period is smaller than d/ϵ. Fortunately, all these patterns have a minimum period at least $d/((n-1)(1+\alpha))$ and a maximum period at most $(1+\alpha)d/\epsilon$. Therefore, running Algorithm 2 on $t_1, \ldots, t_n$ finds the longest of these patterns present in the input in $O(n \log_{1+\epsilon}((n-1)(1+\alpha)/\epsilon)) = O(n \log_{1+\epsilon}(n/\epsilon))$ time (see Theorem 1). Since some numbers of the original input are no longer
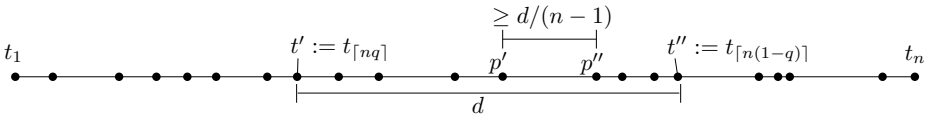
Fig. 1. Relative position of the different numbers selected by Algorithm 3

Algorithm 3. Average gap

Input: ϵ and the input numbers $t_1, t_2, \ldots, t_n$.
1. If $n < \frac{3}{1-2q}$, compute the longest periodic pattern in $t_1, \ldots, t_n$ whose period ratio is at most $(1+\alpha)\frac{1+2\epsilon}{1-2\epsilon}$ (using any exact algorithm, e.g. the one in Section 3.1). **return** *longest pattern found*
2. Find $t' := t_{\lceil nq \rceil}, t'' := t_{\lceil n(1-q) \rceil}$ and $d := t'' - t'$.
3. Find two consecutive numbers p' and p'' in $t', \ldots, t''$ whose distance is at least $d/(n-1)$.
4. Run Algorithm 2 on $t_1, \ldots, t_n$ with $low := d/((1+\alpha)(n-1)(1-2\epsilon))$, $high := d(1+\alpha)(1+2\epsilon)/\epsilon$, $\alpha' := (1+\alpha)\frac{1+2\epsilon}{1-2\epsilon} - 1$ and ϵ.
5. Run Algorithm 3 recursively on $t_1, \ldots, p'$.
6. Run Algorithm 3 recursively on $p'', \ldots, t_n$.
7. Run Algorithm 3 recursively on $t_1, \ldots, t', t'', \ldots, t_n$.
8. **return** the longest periodic pattern found in any of the steps 4–7.

---

present in the recursive calls, we actually need to use a slightly lower minimum period limit and a slightly larger maximum period limit. This will become clear in the correctness proof for Algorithm 3.

Note that by construction, the size of each part which is recursively searched is at most a fixed constant fraction of the input. This is important to achieve logarithmic recursion depth. Since the inputs of the recursive calls are not disjoint, the total run time is $\omega(n \log n)$. However, by decreasing q, the overlap can be made small while keeping the logarithmic recursion depth.

Correctness. To prove the correctness of Algorithm 3, we will consider a particular sequence of recursive calls for every periodic pattern $\mathcal{P}$ that is present in the original input $\mathcal{S}$. The following lemma helps us find a suitable call sequence:

Lemma 3. *Consider any relative error periodic pattern $\mathcal{P}$ with period ratio at most $1 + \alpha$. Let $Pmax$ be the maximum period of $\mathcal{P}$, and $Pmin$ its minimum period. During the execution of Algorithm 3, there is a recursive call such that:*

(A) For each number r in $\mathcal{P}$, there exists some number u in the input of the recursive call, such that the absolute difference between r and u is at most $\epsilon Pmin$, and

(B) $(d/(n-1) \leq Pmax \leq d(1+\alpha)/\epsilon)$ or $(n < \frac{3}{1-2q})$, where n and d are defined as in Algorithm 3.

Proof. We follow a particular path of recursive calls of Algorithm 3 to prove our claim. As we will show, there exists a path of recursive calls leading to a call satisfying the conditions stated in the lemma. To that end, we show that the first property (A) stated in the lemma is actually an invariant, in the sense that for any $\mathcal{P}$, we can select a particular call path for which (A) holds for all recursive calls along the path.

Initially, the invariant holds trivially because all numbers of $\mathcal{P}$ are present in the input. Moreover, whenever p' and p'' both lie to the right of $\mathcal{P}$'s last number, or both lie to the left of $\mathcal{P}$'s first number, we consider the call which fully contains $\mathcal{P}$ (i.e., either the call in step 5 or in step 6). The gap between p' and p'' is either to the left or to the right of $\mathcal{P}$ in this case. For any number r of $\mathcal{P}$ which is not in the call's input, let u be the closest number to r which is still in the input (if there are two such numbers, pick one arbitrarily). Since there are no numbers between u and r in the input, the gap p', p'' cannot lie between u and r, and thus the invariant holds.

Otherwise, $\mathcal{P}$ contains some numbers to the left of p' as well as some numbers to the right of p''. Thus, we have $Pmax \geq Pmin \geq d/(n-1)$ in this recursive call. If it holds additionally that $Pmax \leq d(1+\alpha)/\epsilon$, then we have proved the lemma because of the invariant. If $n < \frac{3}{1-2q}$, then again the lemma is proved thanks to the invariant. In the remaining case, $Pmax > d(1+\alpha)/\epsilon$, and $Pmin \geq Pmax/(1+\alpha) > d/\epsilon$. In this case, we continue the considered call path by following the recursive call in step 7 of Algorithm 3. To show that the invariant still holds in this case, we consider all the different possibilities of how it could be violated. Note that as long as all numbers of $\mathcal{P}$ are still in the input for step 7, the invariant holds trivially. We need to consider two additional cases: (i) If a number r of $\mathcal{P}$ is in the current input but is removed from the input in step 7 (i.e., $r \in t_{\lceil nq \rceil + 1}, \ldots, t_{\lceil n(1-q) \rceil - 1}$), then we need to show that there remains a number u in the input of the call in step 7 which differs from r by at most $\epsilon Pmin$. Recall that $d < \epsilon Pmin$. Thus, when r is removed, the two numbers t' and t'' which remain in the input are both within distance less than $\epsilon Pmin$ from r, and the invariant holds. (ii) If a number r of $\mathcal{P}$ is not in the current input, by the invariant there is a number in the current input which is within distance $\epsilon Pmin$ of r. Let u be the closest number to r in the current input. Note that if $u = t'$ or $u = t''$, u will not be removed from the input, and the invariant holds. If u is removed from the input in step 7 (i.e., $u \in t_{\lceil nq \rceil + 1}, \ldots, t_{\lceil n(1-q) \rceil - 1}$), we need to show that there exists another number in the input to the call in step 7 which is also within distance $\epsilon Pmin$ of r. In this case, clearly r must lie between t' and t'' (in the original input). Thus, r is within distance at most $\epsilon Pmin$ of both t' and t'', and hence the invariant holds.

While we are following a particular path as described above, the size of the input is guaranteed to decrease with each recursive call, because $n \geq \frac{3}{1-2q}$ ensures that $t', \ldots, t''$ contains at least three numbers. The path we follow thus eventually leads to a call satisfying the lemma, because (B) is satisfied as soon as n is small enough, and (A) is an invariant of the considered path. $\square$

Theorem 2. *Given constants α and $\delta \leq \alpha$, Algorithm 3 returns a relative error periodic pattern whose length is at least as long as the longest one with period ratio at most $1 + \alpha$, whose period ratio is at most $(1 + \alpha)(1 + \delta)$, if called with suitably chosen ϵ.*

Proof. Since we never insert any numbers in the input, but only remove some (for the recursive calls in steps 5–7), and by correctness of Algorithm 2 (Theorem 1), Algorithm 3 never outputs a pattern which is not present in the input. Thus, it suffices to prove that for every relative error periodic pattern $\mathcal{P}$ with period ratio at most $1 + \alpha$, our algorithm returns a pattern whose length is at least as long as $\mathcal{P}$, and whose period ratio is at most $(1 + \alpha)(1 + \delta)$. To this end, fix a particular $\mathcal{P}$. We focus on the recursive call defined by Lemma 3: We can replace each number in $\mathcal{P}$ by a different point within distance (in terms of absolute difference) less than $\epsilon Pmin$ which is still present in the input. Moreover, if we choose $\epsilon < 1/2$, then no two numbers of $\mathcal{P}$ will be replaced by the same number. Therefore, by replacing each number $\mathcal{P}$ by its closest number which is still in the input, we obtain a pattern $\mathcal{H}$ which is exactly as long as $\mathcal{P}$, and which consists only of numbers that are still present in the input. By Lemma 2, the period ratio of $\mathcal{H}$ is at most $\frac{Pmax}{Pmin}\frac{1+2\epsilon}{1-2\epsilon}$.

Recall that in this call, $Pmax \leq d(1 + \alpha)/\epsilon$, and $Pmax \geq Pmin \geq d(n - 1)$. Let $Pmin'$ and $Pmax'$ be the minimum and maximum period of $\mathcal{H}$. We have $Pmin' \geq Pmin(1 - 2\epsilon) \geq Pmax/(1 + \alpha) \cdot (1 - 2\epsilon) \geq d(1 - 2\epsilon)/((1 + \alpha)(n - 1))$ and $Pmax' \leq Pmax(1 + 2\epsilon) \leq d(1 + \alpha)(1 + 2\epsilon)/\epsilon$. These bounds match exactly the values for *low* and *high* in Algorithm 3. Moreover, we use $\alpha' := (1 + \alpha)\frac{1+2\epsilon}{1-2\epsilon} - 1$ for running Algorithm 2. Altogether, this ensures that $\mathcal{H}$ is a valid pattern, and thus the algorithm returns a pattern whose length is at least as long as $\mathcal{H}$ (and $\mathcal{P}$), and whose period ratio is at most $(1 + \alpha)\frac{1+2\epsilon}{1-2\epsilon}(1 + \epsilon)$.

Summarizing, we have that for any pattern $\mathcal{P}$ with period ratio at most $1 + \alpha$, Algorithm 3 outputs a pattern which is at least as long as $\mathcal{P}$, whose period ratio is at most $(1 + \alpha)\frac{1+2\epsilon}{1-2\epsilon}(1 + \epsilon)$. Clearly, this period ratio is at most $(1 + \alpha)(1 + \delta)$ for small enough ϵ, which concludes the proof. □

Run time analysis. The run time of Algorithm 3 depends on the parameter q. Indeed, by choosing the constant q sufficiently small, we obtain an exponent which is arbitrarily close to 1:

Theorem 3. *For any constant $\epsilon > 0$, Algorithm 3 has run time $O(n^{1+\gamma})$, where $\gamma > 0$ is an arbitrarily small constant.*

Proof. Let $T(n)$ denote the run time of Algorithm 3 on n numbers. Assume q is a fixed constant (its value will be determined later in the proof). Steps 1, 2 and 3 only require $O(n)$ time. Step 4 takes $O(n \log_{1+\epsilon}(\frac{d/\epsilon}{d/(n-1)})) = O(n \log_{1+\epsilon}(n/\epsilon))$ time (see Theorem 1). For the recursive calls in steps 5 and 6, we are guaranteed that $t_1, \ldots, p'$ contains at least nq numbers and at most $n(1 - q)$ numbers. Consequently, $p'', \ldots, t_n$ contains at least nq and at most $n(1 - q)$ numbers. Step 7 runs our algorithm on at most $2nq + 2$ numbers. Thus, we have the following recurrence relation for the run time:

$$T(n) = an\log_{1+\epsilon}(n/\epsilon) + T(x) + T(n-x) + T(2nq+2),$$

where $nq \le x \le n(1-q)$, and where a is some constant.

We prove by induction on n that $T(n) \le b \cdot n^c$ for some constants b and $c > 1$. The base case is trivial (for n below some constant, we can always find a suitable b). For the induction step, assume $T(i) \le b \cdot i^c$ for all $i < n$, and let n be large enough such that $nq > 1$ and $2nq + 2 < n$. Then, $T(n) \le an\log_{1+\epsilon}(n/\epsilon) + b(x^c + (n-x)^c) + b(2nq+2)^c$. Assuming $n \ge 2/q$, we have $T(n) \le an\log_{1+\epsilon}(n/\epsilon) + b(x^c + (n-x)^c) + b(3nq)^c$. Simple calculus shows that for $qn \le x \le n(1-q)$, the term $x^c + (n-x)^c$ is maximized when $x = nq$ (or, equivalently, when $x = n(1-q)$). Thus, we have $T(n) \le an\log_{1+\epsilon}(n/\epsilon) + bn^c((1-q)^c + q^c(1+3^c))$ $= bn^c\big((1-q)^c + q^c(1+3^c) + (a/b)n^{1-c}\log_{1+\epsilon}(n/\epsilon)\big)$. Note that if we set q such that $(1-q)^{c-1} = (1+3^c)q^{c-1}$ (which is true for $q := 1/(1 + (1+3^c)^{1/(c-1)}))$, we have $(1-q)^c + q^c(1+3^c) = (1-q)(1+3^c)q^{c-1} + q^c(1+3^c) = (1+3^c)q^{c-1}$. Substituting our chosen value for q, we have

$$(1+3^c)q^{c-1} = \frac{1+3^c}{(1+(1+3^c)^{1/(c-1)})^{c-1}} < \frac{1+3^c}{((1+3^c)^{1/(c-1)})^{c-1}} = 1.$$

Hence, for any $c > 1$, we have $(1-q)^c + q^c(1+3^c) + (a/b)n^{1-c}\log_{1+\epsilon}(n/\epsilon) < 1$ for some constant $q > 0$, for large enough n. This concludes the induction step. $\square$

4   Solving Approximate 3SUM in $O(n^{1+\gamma} + T_{sort}(n))$ Time

The well-known 3SUM problem can be defined as follows [1]: "Given three sets A, B, C of integers, each with cardinality at most n, is there a triple $a \in A$, $b \in B$, $c \in C$ such that $a + b = c$?" We briefly explain how Algorithm 3 can be modified to give an approximate answer to the 3SUM problem, as follows:

– It concludes that no triple $(a, b, c) \in A \times B \times C$ with $a + b = c$ exists, or
– it outputs a triple $(a, b, c) \in A \times B \times C$ with $a + b \in [c/(1+\epsilon), c(1+\epsilon)]$.

Although this algorithm does not give an approximation in the usual way, it approximates 3SUM in the following sense: (i) Whenever the exact algorithm would find an exact triple, our algorithm finds an approximate triple. (ii) Whenever our algorithm does not find an approximate triple, the exact algorithm would not find an exact triple either.

We first transform the given sets A, B, C as follows: $A' := \{2x | x \in A\}$, $B' := \{2x | x \in B\}$, $C' := C$. We thus need to either find a triple $(a', b', c') \in A' \times B' \times C'$ with $\frac{a'+b'}{2} \in [c'/(1+\epsilon), c'(1+\epsilon)]$, or to show that no triple exists with $\frac{a'+b'}{2} = c'$. Note that such a triple a', b', c' is an arithmetic progression of length three, with the constraints that it contains one number from each of A', B', C', and that the median of the three numbers is from C'. We solve the approximate 3SUM problem by modifying Algorithm 3 such that it returns a triple $(a', b', c') \in A' \times B' \times C'$ with $\frac{a'+b'}{2} \in [c'/(1+\epsilon), c'(1+\epsilon)]$ whenever an exact triple exists, as follows:

- The input set $\mathcal{S}$ is replaced by the three sets A', B', C', and we sort each set in $O(T_{sort}(n))$ time.
- In line 3 of Algorithm 2, we replace Algorithm 1 by an algorithm which determines whether any triple $(a', b', c') \in A' \times B' \times C'$ exists such that $|a' - c'| \in [p_{\min}, p_{\max}]$ and $|b' - c'| \in [p_{\min}, p_{\max}]$. There exists a simple linear time algorithm which achieves this. Hence, Algorithm 2, which is used in step 4 of Algorithm 3, then also works with the additional constraint mentioned above.
- In step 7 of Algorithm 3, we run Algorithm 3 recursively on the sequence $t_1, \ldots, t_{\lceil nq \rceil - 1}, \hat{t}_1, \ldots, \hat{t}_6, t_{\lceil n(1-q) \rceil + 1}, \ldots t_n$, where the sorted list $\hat{t}_1, \ldots, \hat{t}_6$ contains, from each set $X \in \{A', B', C'\}$, the number in $\{t', \ldots, t''\} \cap X$ closest to t' and the number in $\{t', \ldots, t''\} \cap X$ closest to t''.

It is not hard to verify that after these modifications of the algorithm, Lemma 3 and Theorem 2 also hold for approximate periodic patterns of length three with the mentioned additional constraints. Moreover, the run time analysis remains the same, except that step 7 of Algorithm 3 may run on up to $2nq + 6$ numbers (instead of up to $2nq + 2$), which does not affect the asymptotic time bound.

References

1. Gajentaan, A., Overmars, M.H.: On a class of $O(n^2)$ problems in computational geometry. Computational Geometry 5(3), 165–185 (1995)
2. Han, J., Dong, G., Yin, Y.: Efficient mining of partial periodic patterns in time series database. In: Proc. of the 15th International Conference on Data Engineering (ICDE 1999), pp. 106–115. IEEE Computer Society, Los Alamitos (1999)
3. Kahng, A.B., Robins, G.: Optimal algorithms for extracting spatial regularity in images. Pattern Recognition Letters 12(12), 757–764 (1991)
4. Ma, S., Hellerstein, J.L.: Mining partially periodic event patterns with unknown periods. In: Proc. of the 17th International Conference on Data Engineering (ICDE 2001), pp. 205–214. IEEE Computer Society, Los Alamitos (2001)
5. Rasheed, F., Alshalalfa, M., Alhajj, R.: Efficient Periodicity Mining in Time Series Databases Using Suffix Trees. IEEE Transactions on Knowledge and Data Engineering 99 (2010) (preprints)
6. Robins, G., Robinson, B.L.: Pattern Minefield Detection from Inexact Data. In: Proc. SPIE International Symposium on Aerospace/Defense Sensing and Dual-Use Photonics, vol. 2496, pp. 568–574 (1995)
7. Tanbeer, S., Ahmed, C., Jeong, B.-S., Lee, Y.-K.: Discovering periodic-frequent patterns in transactional databases. In: Theeramunkong, T., Kijsirikul, B., Cercone, N., Ho, T.-B. (eds.) PAKDD 2009. LNCS, vol. 5476, pp. 242–253. Springer, Heidelberg (2009)
8. Yang, J., Wang, W., Yu, P.S.: Mining asynchronous periodic patterns in time series data. IEEE Trans. on Knowl. and Data Eng. 15, 613–628 (2003)

A $(5/3 + \varepsilon)$-Approximation for Strip Packing[*]

Rolf Harren[1], Klaus Jansen[2], Lars Prädel[2], and Rob van Stee[1]

[1] Max-Planck-Institut für Informatik (MPII), Campus E1 4, 66123 Saarbrücken, Germany
{rharren,vanstee}@mpi-inf.mpg.de
[2] Universität Kiel, Institut für Informatik, Christian-Albrechts-Platz 4, 24118 Kiel, Germany
{kj,lap}@informatik.uni-kiel.de

Abstract. We study strip packing, which is one of the most classical two-dimensional packing problems: Given a collection of rectangles, the problem is to find a feasible orthogonal packing without rotations into a strip of width 1 and minimum height. In this paper we present an approximation algorithm for the strip packing problem with approximation ratio of $5/3 + \varepsilon$ for any $\varepsilon > 0$. This result significantly narrows the gap between the best known upper bounds of 2 by Schiermeyer and Steinberg and 1.9396 by Harren and van Stee and the lower bound of $3/2$.

Keywords: strip packing, rectangle packing, approximation algorithm, absolute worst-case ratio.

1   Introduction

Two-dimensional packing problems are classical in combinatorial optimization and continue to receive a lot of research interest [4,5,9,10,11,13,14]. One of the most important ones is the strip packing problem also known as the cutting stock problem: given a set of rectangles $I = \{r_1, \ldots, r_n\}$ of specified widths w_i and heights h_i, the problem is to find a feasible packing for I (i.e. an orthogonal arrangement where rectangles do not overlap and are not rotated) into a strip of width 1 and minimum height.

The strip packing problem has many practical applications in manufacturing, logistics, and computer science. In many manufacturing settings rectangular pieces need to be cut out of some sheet of raw material, while minimizing the waste. Scheduling independent tasks on a group of processors, each requiring a certain number of contiguous processors or memory allocation during a certain length of time, can also be modeled as a strip packing problem.

Since strip packing includes bin packing as a special case (when all heights are equal), the problem is strongly $\mathcal{NP}$-hard. Therefore, there is no efficient algorithm for constructing an optimal packing, unless $\mathcal{P} = \mathcal{NP}$. We focus on approximation algorithms with good performance guarantee. Let $A(I)$ be the objective value (in our case the height of the packing) generated by a polynomial-time algorithm A, and $\mathrm{OPT}(I)$ be the optimal value for an instance I. The approximation ratio of A is $\sup_I \frac{A(I)}{\mathrm{OPT}(I)}$

whereas the asymptotic approximation ratio of A is defined by $\limsup_{\mathrm{OPT}(I)\to\infty}$ $\frac{A(I)}{\mathrm{OPT}(I)}$. A problem admits a polynomial-time approximation scheme ($\mathcal{PTAS}$) if there is a family of algorithms $\{A_\varepsilon \mid \varepsilon > 0\}$ such that for any $\varepsilon > 0$ and any instance I, A_ε produces a $(1 + \varepsilon)$-approximate solution in time polynomial in the size of the input. A fully polynomial-time approximation scheme ($\mathcal{FPTAS}$) is a $\mathcal{PTAS}$ where additionally A_ε has run-time polynomial in $1/\varepsilon$ and the size of the input.

Results. The Bottom-Left algorithm by Baker et al. [2] has asymptotic approximation ratio equal to 3 when the rectangles are ordered by decreasing widths. Coffman et al. [6] provided the first algorithms with proven approximation ratios of 3 and 2.7, respectively. The approximation algorithm presented by Sleator [16] generates a packing of height $2\,\mathrm{OPT}(I) + h_{\max}(I)/2$. Since $h_{\max}(I) \leq \mathrm{OPT}(I)$ this implies an absolute approximation ratio of 2.5. This was independently improved by Schiermeyer [15] and Steinberg [17] with algorithms of approximation ratio 2.

In the asymptotic setting we consider instances with large optimal value. Here, the asymptotic performance ratio of the above algorithms was reduced to $4/3$ by Golan [7] and then to $5/4$ by Baker et al. [1]. An asymptotic $\mathcal{FPTAS}$ with additive constant of $\mathcal{O}(h_{\max}(I)/\varepsilon^2)$ was given by Kenyon & Rémila [14]. Jansen & Solis-Oba [11] found an asymptotic $\mathcal{PTAS}$ with additive constant of $h_{\max}(I)$.

On the negative side, since strip packing includes the bin packing problem as a special case, there is no algorithm with absolute ratio better than $3/2$ unless $\mathcal{P} = \mathcal{NP}$. After the work by Steinberg and Schiermeyer in 1994, there was no improvement on the best known approximation ratio until very recently. Jansen & Thöle [12] presented an approximation algorithm with approximation ratio $3/2 + \varepsilon$ for restricted instances where the widths are of the form i/m for $i \in \{1, \ldots, m\}$ and m is polynomially bounded in the number of items. Notice that the general version that we consider appears to be considerably more difficult. Recently, Harren & van Stee [9] were the first to break the barrier of 2 for the general problem and presented an algorithm with a ratio of 1.9396. Our main result is the following significant improvement.

Theorem 1. *For any $\varepsilon > 0$, there is an approximation algorithm A which produces a packing of a list I of n rectangles in a strip of width 1 and height $A(I)$ such that*

$$A(I) \leq \left(\frac{5}{3} + \varepsilon\right) \mathrm{OPT}(I).$$

Although our algorithm uses a $\mathcal{PTAS}$ as a subroutine and therefore has very high running time for small values of ε, this result brings us much closer to the lower bound of $3/2$ for this problem.

Techniques. The algorithm approximately guesses the optimal height of a given instance. In the main phase of the algorithm we use a recent result by Bansal et al. [3], a $\mathcal{PTAS}$ for the so-called rectangle-packing problem with area maximization (RPA). Given a set I of rectangles, the objective is to find a subset $I' \subseteq I$ of the rectangles and a packing of I' into a unit sized bin while maximizing the total area of I'. For the iteration close to the minimal height, the approximation scheme by Bansal et al. computes a

packing of a subset of the rectangles with total area at least $(1 - \delta)$ times the total area of all rectangles in I. After this step a set of unpacked rectangles with small total area remains. The main idea of our algorithm is to create a *hole* of depth 1/3 and width ε in the packing created by the $\mathcal{PTAS}$, and use this to pack the unpacked tall items (with height possibly very close to 1). (The other unpacked items account for the $+\varepsilon$ in our approximation ratio.) Finding a suitable location for such a hole and repacking the items which we have to move out of the hole account for the largest technical challenges of this paper. To achieve a packing of the whole input we carefully analyse the structure of the generated packing and use interesting and often intricate rearrangements of parts of the packing.

The techniques of this geometric analysis and the reorganization of the packing could be useful for several other geometric packing problems. Our reoptimization could also be helpful for related problems like scheduling parallel tasks (malleable and non-malleable), three-dimensional strip packing and strip packing in multiple strips. To achieve faster heuristics for strip packing, we could apply our techniques on different initial packings rather than using the $\mathcal{PTAS}$ from [3].

2 Overview of the Algorithm

Let $I = \{r_1, \ldots, r_n\}$ be the set of given rectangles, where $r_i = (w_i, h_i)$ is a rectangle with width w_i and height h_i. For a given packing P we denote the bottom left corner of an item r_i by (x_i, y_i) and its top right corner by (x'_i, y'_i), where $x'_i = x_i + w_i$ and $y'_i = y_i + h_i$. So the interior of rectangle r_i covers the area $(x_i, x'_i) \times (y_i, y'_i)$. It will be clear from the context to which packing P the coordinates refer.

Let $W_\delta = \{r_i \mid w_i > \delta\}$ be the set of so-called δ-*wide* items and let $H_\delta = \{r_i \mid h_i > \delta\}$ be the set of δ-*high* items. To simplify the presentation, we refer to the 1/2-wide items as *wide* items and to the 1/2-high items as *high* items. Let $W = W_{1/2}$ and $H = H_{1/2}$ be the sets of wide and high items, respectively.

For a set T of items, let $\mathcal{A}(T) = \sum_{i \in T} w_i h_i$ be the total area and let $h(T) = \sum_{r_i \in T} h_i$ and $w(T) = \sum_{r_i \in T} w_i$ be the total height and total width, respectively. Furthermore, let $w_{\max}(T) = \max_{r_i \in T} w_i$ and $h_{\max}(T) = \max_{r_i \in T} h_i$.

We use two important subroutines in our algorithms, namely Steinberg's algorithm [17] and a recent algorithm by Bansal et al. [3]. The algorithm by Steinberg allows us to pack a set T of rectangles into a container $R = (a, b)$, when the following conditions hold: $w_{\max}(T) \leq a$, $h_{\max}(T) \leq b$, and $2\mathcal{A}(T) \leq ab - (2w_{\max}(T) - a)_+ (2h_{\max}(T) - b)_+$, where $x_+ = \max(x, 0)$. Bansal, Caprara, Jansen, Prädel & Sviridenko [3] considered the problem of maximizing the total area packed into a unit-sized bin. Using a technical *Structural Lemma* they derived a $\mathcal{PTAS}$ for this problem: For any fixed $\delta > 0$, the $\mathcal{PTAS}$ returns a packing of $I' \subseteq I$ in a unit-sized bin such that $\mathcal{A}(I') \geq (1 - \delta)\mathrm{OPT}_{\max\,\mathrm{area}}(I)$, where $\mathrm{OPT}_{\max\,\mathrm{area}}(T)$ denotes the maximum area of items from T that can be packed into a unit-sized bin.

With similar methods as in [9], we can derive a statement on the existence of a structured packing of certain sets of wide and high items (a full description is also given in the full version of this paper [8]): For sets $H' \subseteq H$ and $W' \subseteq W \setminus H'$ of high and wide items with $\mathrm{OPT}(W \cup H) \leq 1$ we can derive a packing of $W' \cup H'$ into a

(a) PointI(x, y) (b) HLI$(x_1, x_2; y)$ (c) AI$(x_1, x_2; y_1, y_2)$ (d) AI(VPCE(x, y))
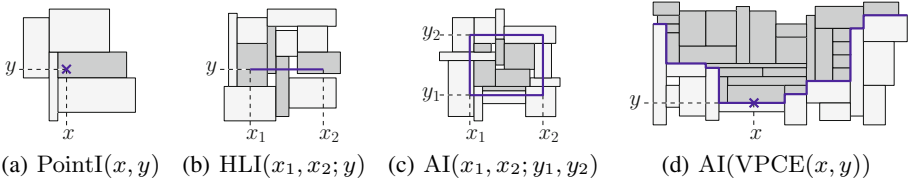
Fig. 1. Notations

strip of height at most $1 + h(W')/2$ such that the wide items are stacked in the bottom right of the strip and the high items are stacked above the wide items at the left side of the strip.

Modifying packings. Our methods involve modifying existing packings in order to insert some additional items. To describe these modifications or, more specifically, the items involved in these modifications, we introduce the following notations—see Figure 1. Let PointI(x, y) be the item that contains the point (x, y) (in its interior). We use the notation of *vertical line items* VLI$(x; y_1, y_2)$ and *horizontal line items* HLI$(x_1, x_2; y)$ as the items that contain any point of the given vertical or horizontal line in their interiors, respectively. Finally, we introduce two notations for items whose interiors are completely contained in a designated area, namely AI$(x_1, x_2; y_1, y_2)$ for items completely inside the respective rectangle and AI(p) for items completely above a given polygonal line p, where p is a staircase-function on $[0, 1]$.

To describe such a polygonal line p we define the *vertical polygonal chain extension* of a point (x, y) inside a given packing P as follows. Start at position (x, y) and move leftwards until hitting an item r_i. Then move upwards to the top of r_i, that is, up to position y_i'. Repeat the previous steps until hitting the left side of the strip. Then do the same thing to the right starting again at (x, y). We denote the polygonal chain that results from this process by VPCE(x, y). In addition, let VPCE$_{\text{left}}(x, y)$ and VPCE$_{\text{right}}(x, y)$ be the left and right parts of this polygonal chain, respectively. Another way to describe a polygonal line is by giving a sequence of points, which we denote as PL$((x_1, y_1), (x_2, y_2), \ldots)$.

Algorithm. We start now with the presentation of our algorithm. Let $\varepsilon < 1/(28 \cdot 151) = 1/4228$ throughout the paper. In a first step we scale the heights of all rectangles, so that an optimal solution fits into a unit square. In order to do this, we first apply Steinberg's algorithm to get a 2-approximation, and then *guess* the optimal height up to ε accuracy. Afterwards we scale the heights of all rectangles by this guess.

We will not make this explicit anymore but from now on only consider the case where we have guessed correctly, hence OPT$(I) \leq 1$ (for a full description of the proof, we refer to [9] or the full version of this paper). We only give a high-level version of the algorithm here. The phrases 'very close' and 'very wide' should be understood to depend on our accuracy parameter ε.

In the next phase we use some direct methods involving Steinberg's algorithm to solve instances I with $h(W_{1-130\varepsilon}) \geq 1/3$ or $w(H_{2/3}) \geq 27/28$, that is, special cases where many items have a width of almost 1, or almost all of the items are at least

2/3 high. Having this many high or wide items makes it much easier to pack all items without wasting much space. We present these methods in the full version of this paper, and focus on the important general case in the rest of the paper.

Algorithm 1. Turn the PTAS packing into a strip packing

Requirement: $\varepsilon < 1/4228$, $h(W_{1-130\varepsilon}) < 1/3$ and $w(H_{2/3}) < 27/28$
Input: packing P produced by the $\mathcal{PTAS}$ from [3] with an accuracy of $\delta := \varepsilon^2/2$.
1: Pack the remaining unpacked items into $C_1 = (\varepsilon, 1)$ and $C_2 = (1, \varepsilon)$.
2: **if** there is 1/3-high item with one side at position $x_1^* \in [\varepsilon, 1/2 - \varepsilon]$, and the total width of 2/3-high items to the left of x_1^* is at most $x_1^* - \varepsilon$ (or if these conditions hold for P mirrored over $x = 1/2$) **then**
3:    apply Algorithm 2 (Lemma 1), stop
4: **if** there is an item r_1 of height $h_1 \in [1/3, 2/3]$ and width $w_1 \in [\varepsilon, 1 - 2\varepsilon]$ and $y_1 \geq 1/3$ or $y_1' \leq 2/3$ **then**
5:    apply Algorithm 3 (Lemma 2), stop
6: Let r_ℓ be the rightmost 2/3-high item in AI$(0, 1/2 - \varepsilon; 0, 1)$ and let r_r be the leftmost 2/3-high item in AI$(1/2 + \varepsilon, 1; 0, 1)$.
      \\At this point, all vertical sides of 1/3-high items are to the left of r_ℓ,
      \\ to the right of r_r, or within ε distance of $x = 1/2$.
7: **if** there is no 1/3-high item in a particular horizontal interval of width ε very close to the side of the strip, and the total height of very wide items is less than 1/3, **then**
8:    create a strip for C_1 in this interval.
9: Apply specific methods based on which of the cases shown in Figure 2 occurs. (Figure 2 covers all the cases because of where the 1/3-high sides of items can be, given that the algorithm did not terminate yet.)
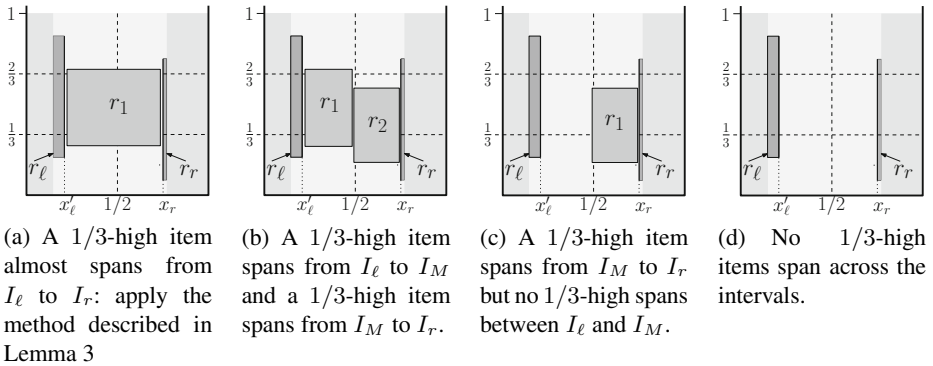

(a) A 1/3-high item almost spans from I_ℓ to I_r: apply the method described in Lemma 3

(b) A 1/3-high item spans from I_ℓ to I_M and a 1/3-high item spans from I_M to I_r.

(c) A 1/3-high item spans from I_M to I_r but no 1/3-high spans between I_ℓ and I_M.

(d) No 1/3-high items span across the intervals.

Fig. 2. Schematic illustration of the main cases if Lemma 1 and 2 are not applicable. The area to the left of r_ℓ and the area to the right of r_r is almost completely covered by 2/3-high items (and shown in darker shade). I_ℓ, I_r and I_M are horizontal intervals very close to r_ℓ, r_r and the middle of the strip.

For the general case, we first apply the $\mathcal{PTAS}$ from [3] to pack most of the items into a strip of height 1. Denote the resulting packing of $I' \subseteq I$ by P and let $R = I \setminus I'$ be the set of remaining items. By the $\mathcal{PTAS}$ from [3] we have $\mathcal{A}(R) \leq \varepsilon^2/2 \cdot \text{OPT}_{\max \text{area}}(I) = \varepsilon^2/2 \cdot \mathcal{A}(I) \leq \varepsilon^2/2$. Pack $R \cap H_{\varepsilon/2}$ into a container $C_1 = (\varepsilon, 1)$

(by forming a stack of the items of total width at most $\mathcal{A}(R)/(\varepsilon/2) \leq \varepsilon$) and pack $R \setminus H_{\varepsilon/2}$ with Steinberg's algorithm into a container $C_2 = (1, \varepsilon)$ (this is possible by Steinberg's algorithm [17] since $h_{\max}(R \setminus H_{\varepsilon/2}) \leq \varepsilon/2$, $w_{\max}(R \setminus H_{\varepsilon/2}) \leq 1$ and $2\mathcal{A}(R \setminus H_{\varepsilon/2}) \leq \varepsilon^2 < \varepsilon$).

We will now modify the packing P to free a gap of width ε and height 1 to insert the container C_1 while retaining a total packing height of at most $5/3$. This is the main part of our work. Afterwards, we pack C_2 above the entire packing, achieving a total height of at most $5/3 + \varepsilon$. The entire algorithm to modify the PTAS packing is given in Algorithm 1. The running time of the $\mathcal{PTAS}$ from [3] is polynomial in the number of items in the input (it is not explicitly stated in [3]). In the following sections that build upon packing P we only give the additional running time for modifying the packing.

It is not possible to present all methods for modifying the packing P, due to page limitations. Instead, we convey the main ideas and methods by presenting two selected parts of our algorithm in Sections 3 and 4 in detail. For the omitted parts we refer to the full version. We prove that we cover all the cases in Section 5.

3   Item of Height Greater than $1/3$

Lemma 1. *If the following conditions hold for P, namely*

3.1. *there is an item r_1 of height $h_1 > 1/3$ with one side at position $x_1^* \in [\varepsilon, 1/2 - \varepsilon]$, and*

3.2. *the total width of $2/3$-high items to the left of x_1^* is at most $x_1^* - \varepsilon$, that is $w(\mathrm{AI}(0, x_1^*; 0, 1) \cap H_{2/3}) \leq x_1^* - \varepsilon$,*

then we can derive a packing of I into a strip of height $5/3 + \varepsilon$ in additional time $\mathcal{O}(n \log n)$.

Note that Condition 3.1 leaves open whether x_1^* refers to the left or right side of r_1 as our method works for both cases. In particular, r_1 could be one of the $2/3$-high items from Condition 3.2.

Proof. Assume w.l.o.g. $y_1' > 2/3$ by otherwise mirroring the packing P over $y = 1/2$.

We lift up a part of the packing P in order to derive a gap of sufficient height to insert the container C_1. In this case we mirror the part of the packing that we lift up. See Figure 3 for an illustration.

Consider the contour C_{lift} defined by a horizontal line at height $y = y_1' - 1/3$ to the left of x_1^*, a vertical line at $x = x_1^*$ up to y_1' and a vertical polygonal chain extension to the right starting at the top of r_1. More formally, $C_{\mathrm{lift}} = \mathrm{PL}((0, y_1' - 1/3), (x_1^*, y_1' - 1/3), (x_1^*, y_1')) + \mathrm{VPCE}_{\mathrm{right}}(x_1^*, y_1')$, where the $+$-operator denotes the concatenation of polygonal lines (see thick line in Figure 3(a)). Let $T = \mathrm{AI}(C_{\mathrm{lift}})$ be the set of items that are completely above this contour.

Move up T by $2/3$ (and hereby move T completely above the previous packing since $y_1' > 2/3$ and thus $y_1' - 1/3 > 1/3$) and mirror T vertically, i.e., over $x = 1/2$. Let y_{bottom} be the height of C_{lift} at $x = 1/2$ (i.e. C_{lift} crosses the point $(1/2, y_{\mathrm{bottom}})$). By definition, C_{lift} is non-decreasing and no item intersects with C_{lift} to the right of x_1^*. Therefore, T is completely packed above $y = y_{\mathrm{bottom}} + 2/3$ on the left side of the strip,

i.e., for $x \leq 1/2$, and $P \setminus T$ does not exceed y_{bottom} between $x = x_1^*$ and $x = 1/2$. Thus between $x = x_1^*$ and $x = 1/2$ we have a gap of height at least $2/3$.

Let $B = \text{HLI}(0, x_1^*; y_1' - 1/3)$ be the set of items that intersect height $y = y_1' - 1/3$ to the left of x_1^* (see Figure 3(a)). Note that $r_1 \in B$, if x_1^* corresponds to the right side of r_1. Remove B from the packing, order the items by non-increasing order of height and build a top-left-aligned stack at height $y = y_{\text{bottom}} + 2/3$ and distance ε from the left side of the strip. Since we keep a slot of width ε to the left, the stack of B might exceed beyond x_1^*. This overhang does not cause an overlap of items because Condition 3.1 ensures that $x_1^* \leq 1/2 - \varepsilon$ and thus the packing of B does not exceed position $x = 1/2$ and Condition 3.2 ensures that the excessing items have height at most $2/3$ whereas the gap has height at least $2/3$.

Now pack the container C_1 top-aligned at height $y_{\text{bottom}} + 2/3$ directly at the left side of the strip. C_1 fits here since $y_{\text{bottom}} + 2/3 - (y_1' - 1/3) = 1 + y_{\text{bottom}} - y_1' \geq 1$. Finally, pack C_2 above the entire packing at height $y = 5/3$, resulting in a total packing height of $5/3 + \varepsilon$. $\qquad\square$
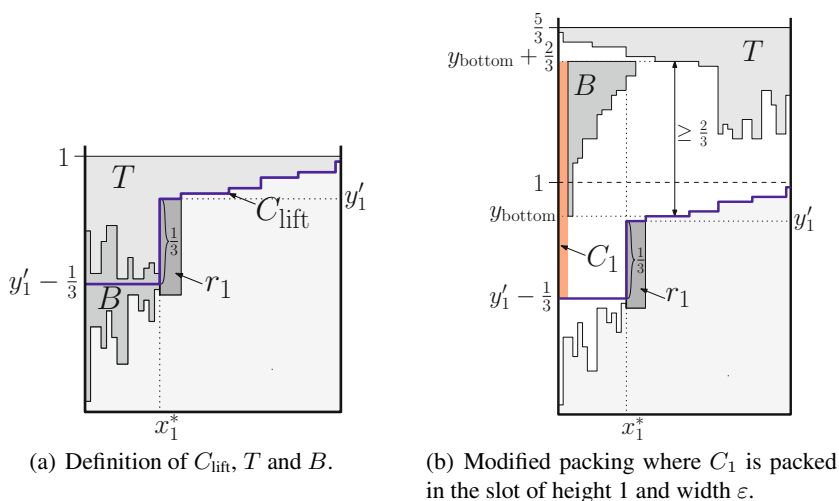

(a) Definition of C_{lift}, T and B.

(b) Modified packing where C_1 is packed in the slot of height 1 and width ε.

Fig. 3. Packing methods for Lemma 1

Algorithm 2. Edge of height greater than $1/3$

Requirement: Packing P that satisfies Conditions 3.1 and 3.2 and $y_1' > 2/3$.
1: Move up the items $T = \text{AI}(C_{\text{lift}})$ by $2/3$ and then mirror the packing of these items vertically at position $x = 1/2$.
2: Order the items of $B = \text{HLI}(0, x_1^*; y_1' - 1/3)$ by non-increasing order of height and pack them into a top-aligned stack at position $(\varepsilon, y_{\text{bottom}} + 2/3)$.
3: Pack C_1 top-aligned at position $(0, y_{\text{bottom}} + 2/3)$ and pack C_2 above the entire packing.

Note that Lemma 1 can symmetrically be applied for a $1/3$-high item with one side at position $x_1^* \in [1/2 + \varepsilon, 1 - \varepsilon]$ with $w(\text{AI}(x_1^*, 1; 0, 1) \cap H_{2/3}) \leq 1 - x_1^* - \varepsilon$ by mirroring P over $x = 1/2$.

4 One Special Big Item in P

Lemma 2. *If the following condition holds for P, namely*

  4.1. there is an item r_1 of height $h_1 \in [1/3, 2/3]$ and width $w_1 \in [\varepsilon, 1 - 2\varepsilon]$, and $y_1 \geq 1/3$ or $y_1' \leq 2/3$,

then we can derive a packing of I into a strip of height $5/3 + \varepsilon$ in additional time $\mathcal{O}(n)$.
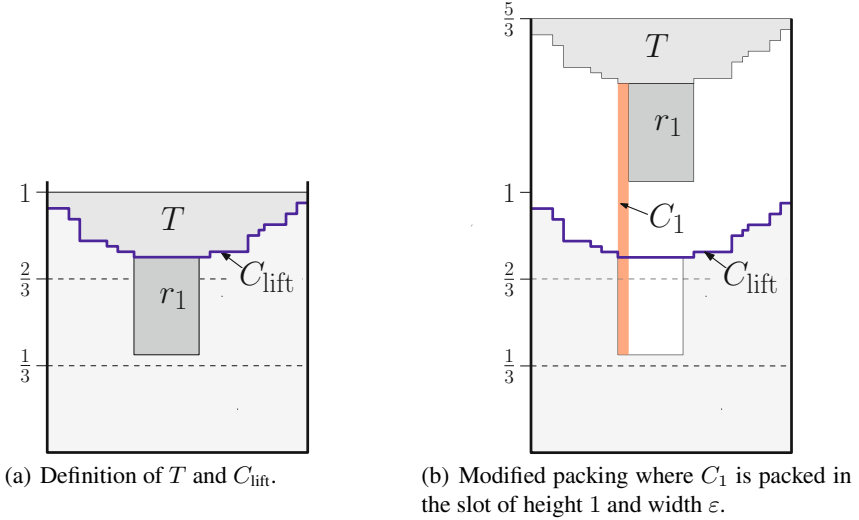

(a) Definition of T and C_{lift}.

(b) Modified packing where C_1 is packed in the slot of height 1 and width ε.

Fig. 4. Packing methods for Lemma 2

Algorithm 3. Rectangle of height $1/3$

Requirement: Packing P that satisfies Condition 4.1.
1: Define $C_{\text{lift}} := \text{VPCE}(x_1, y_1')$ and move up $T = \text{AI}(C_{\text{lift}})$ by $2/3$.
2: Move up r_1 by $2/3$ and then by ε to the right, i.e., pack r_1 at position $(x_1 + \varepsilon, y_1 + 2/3)$.
3: Pack C_1 into the slot vacated by r_1 and pack C_2 above the entire packing.

Proof. See Figure 4 for an illustration of the following proof. W.l.o.g. we assume that $y_1 \geq 1/3$, by otherwise mirroring the packing horizontally, i.e., over $y = 1/2$. Furthermore, we assume that $x_1' \leq 1 - \varepsilon$ since $w_1 \leq 1 - 2\varepsilon$ and otherwise mirror the packing vertically, i.e., over $x = 1/2$.

Define a vertical polygonal chain extension $C_{\text{lift}} = \text{VPCE}(x_1, y_1')$ starting on top of r_1 and let $T = \text{AI}(C_{\text{lift}})$. Move up the rectangles in T and the rectangle r_1 by $2/3$ and hereby move r_1 completely out of the previous packing, since $y_1 \geq 1/3$. Then move r_1 to the right by ε, this is possible, since $x_1' \leq 1 - \varepsilon$.

In the hole vacated by r_1 we have on the left side a free slot of width ε (since $w_1 \geq \varepsilon$ and since we moved r_1 to the right by ε) and height $2/3 + h_1 \geq 1$ (since we moved up T by $2/3$ and since $h_1 \geq 1/3$). Place C_1 in this slot and pack C_2 on top of the packing at height $5/3$.   □

Lemma 3. *Let $c_1 > 0$ be a constant. If the following conditions hold for P, namely*

  4.2. *there is an item r_1 of height $h_1 \in [1/3, 2/3]$ and width $w_1 \in [(4c_1 + 1)\varepsilon, 1]$ with $y_1 < 1/3$ and with $y_1' > 2/3$, and*
  4.3. *we have $w(H_{2/3}) \geq 1 - w_1 - c_1\varepsilon$,*

then we can derive a packing of I into a strip of height $5/3 + \varepsilon$ in additional time $O(n)$.

Proof. Since the height of r_1 is $h_1 \leq 2/3$ we can assume w.l.o.g. that r_1 does not intersect $y = 1/6$, i.e., $y_1 \geq 1/6$ (by otherwise mirroring over $y = 1/2$).

We want to line up all rectangles in the instance I of height greater than $h = \max(1/2, 1 - h_1)$ and the rectangle r_1 on the bottom of the strip. These rectangles fit there, since in any optimal solution they have to be placed next to each other (all rectangles of $H_h = \{r_i \mid h_i > h\}$ have to intersect the horizontal line at height $y = 1/2$ and no rectangle of H_h fits above r_1). Since $1 - h_1 \geq 1/3$, $H_{2/3}$ is included in the set H_h. See Figure 5 for an illustration of the following algorithm.
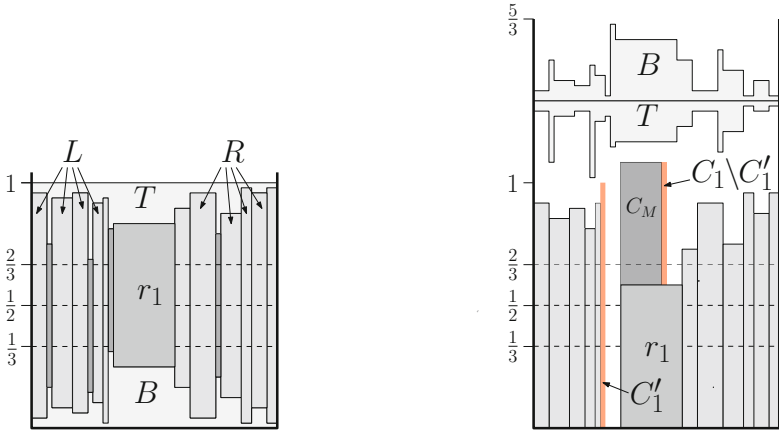
(a) Definition of T, B, L, R and M (dark items squeezed between L and R).

(b) Modified packing where C_1' is packed in the left-hand slot of height 1 and $C_1 \setminus C_1'$ is packed in the right-hand slot of height h and width ε.

Fig. 5. Packing methods for Lemma 3

Let $T = \text{AI}(0, 1; 2/3, 1)$ be the set of rectangles which lie completely above the horizontal line at height $y = 2/3$. We move up the rectangles in T by $1/3$ into the area $[0, 1] \times [1, 4/3]$. Now there is a free space of height at least $1/3$ above r_1.

Let $B = \text{AI}(0, 1; 0, 1/3)$ be the rectangles which lie completely below the horizontal line at height $y = 1/3$. We pack these items into a container $C_B = (1, 1/3)$ by preserving the packing of B and pack C_B at position $(0, 4/3)$, i.e., directly above T. Since by assumption r_1 does not intersect the horizontal line at height $y = 1/6$, there is a free space of height at least $1/6$ below r_1.

The remaining items of height smaller than h except r_1 have to intersect one of the horizontal lines at height $1/3$ or $2/3$ or lie completely between them. We denote these

rectangles by $M_1 = \text{HLI}(0, 1; 1/3) \setminus (H_h \cup \{r_1\})$, $M_2 = \text{HLI}(0, 1; 2/3) \setminus (H_h \cup \{r_1\} \cup M_1)$ and $M_3 = \text{AI}(0, 1; 1/3, 2/3)$. Since each rectangle in $H_{2/3}$ and r_1 intersects both of these lines, the rectangles in $M = M_1 \cup M_2 \cup M_3$ lie between them in slots of total width $c_1 \varepsilon$. Therefore, we can pack M_1 and M_2 each bottom-aligned into a container $(c_1 \varepsilon, h)$. Furthermore, the rectangles in M_3 fit into a container $(c_1 \varepsilon, 1/3)$ by pushing the packing of the slots together. In total we pack M into a container $C_M = (3 c_1 \varepsilon, h)$ and pack it aside for the moment.

After these steps we removed all rectangles of height at most h except r_1 out of the previous packing. All remaining items intersect the horizontal line at height $y = 1/2$. We line up the rectangles in $L = \text{HLI}(0, x_1; 1/2)$, i.e., the remaining rectangles on the left of r_1, bottom-aligned from left to right starting at position $(0, 0)$. The rectangles in $R = \text{HLI}(0, x_1'; 1/2)$ (the remaining rectangles on the right of r_1) are placed bottom-aligned from right to left starting at position $(1, 0)$. Now move r_1 down to the ground, i.e., pack r_1 at position $(x_1, 0)$. Above r_1 is a free space of height at least $1/2$, since we moved T up by $1/3$ and r_1 down by at least $1/6$. The free space has also height at least $1 - h_1$, since there is no item left above r_1 up to height 1. Hence, in total, this leaves us a free space of width $w_1 \geq (4 c_1 + 1)\varepsilon$ and height h. Denote this area by $X = [x, x'] \times [h_1, h_1 + h]$ with $x = x_1$ and $x' = x_1 + w_1$.

Move r_1 to the right by at most $c_1 \varepsilon$ until it touches the first rectangle in R, i.e., place r_1 at position $(1 - w(R) - w_1, 0)$. This reduces the width of the free area on top of r_1 to $X' = [x + c_1 \varepsilon, x'] \times [h_1, h_1 + h]$. Note, the width of X' is still at least $(3 c_1 + 1)\varepsilon$.

In the next step we reorganize the packing of C_1. Recall, that the rectangles in C_1 are placed bottom-aligned in that container. Let C_1' be the rectangles in C_1 of height larger than h. By removing C_1', we can resize the height of C_1 down to h. The resized container C_1 and the container C_M have both height h and total width at most $(3 c_1 + 1)\varepsilon$. Place them on top of r_1 in the area X'.

Then place the rectangles in C_1' into the free slot on the left side of r_1. They fit there, since in any optimal packing all rectangles of height greater than h in the instance and r_1 have to be placed next to each other (all rectangles of height greater than h have to intersect the horizontal line at height $y = 1/2$ and none of them fits above r_1). Finally, pack C_2 above the entire packing at height $5/3$.   □

5 Overview of the Cases

In this section we prove that our algorithm (stated on page 479) indeed covers all the cases. Recall that $\varepsilon < 1/(28 \cdot 151) = 1/4228$. Suppose

$$h(W_{1-130\varepsilon}) < 1/3 \qquad \text{and} \qquad (1)$$
$$w(H_{2/3}) < 27/28. \qquad\qquad (2)$$

Consider the intervals $I_\ell = [0, x_\ell' + \varepsilon]$, $I_M = [1/2 - \varepsilon, 1/2 + \varepsilon]$ and $I_r = [x_r - \varepsilon, 1]$, where x_ℓ' and x_r refer to the items defined in line 6 of the algorithm. From the inapplicability of Lemma 1 on items r_ℓ and r_r follows that the intervals I_ℓ and I_r are almost occupied with 2/3-high items. To be more precise we have $w(\text{AI}(0, x_\ell'; 0, 1) \cap H_{2/3}) \geq x_\ell' - \varepsilon$ and $w(\text{AI}(x_r, 1; 0, 1) \cap H_{2/3}) \geq 1 - x_r - \varepsilon$. Furthermore, the x-coordinates of the sides of all 1/3-high items are in I_ℓ, I_M or I_r, since otherwise we

could apply Lemma 1 on this rectangle. To put it in another way the rectangles in $H_{1/3}$ are either completely inside one of these intervals or span across one interval to another.

If the algorithm reaches line 6 it is not possible that a $2/3$-high item r_1 spans from I_ℓ to I_r, as otherwise we have $w(H_{2/3}) \geq w(\mathrm{AI}(0, x'_\ell; 0, 1) \cap H_{2/3}) + w(\mathrm{AI}(x_r, 1; 0, 1) \cap H_{2/3}) + w_1 \geq x'_\ell - \varepsilon + 1 - x_r - \varepsilon + x_r - x'_\ell - 2\varepsilon \geq 1 - 4\varepsilon > 27/28$ for $\varepsilon < 1/112$. The same holds if there were two $2/3$-high rectangles r_1, r_2, that span from I_ℓ to I_M and I_M to I_r, respectively $(w(H_{2/3}) \geq w(\mathrm{AI}(0, x'_\ell; 0, 1) \cap H_{2/3}) + w(\mathrm{AI}(x_r, 1; 0, 1) \cap H_{2/3}) + w_1 + w_2 \geq x'_\ell - \varepsilon + 1 - x_r - \varepsilon + x_r - x'_\ell - 4\varepsilon \geq 1 - 6\varepsilon > 27/28$ for $\varepsilon < 1/168)$.

If there is a $2/3$-high item r that intersects with $x = x'_\ell + \varepsilon$, i.e., r spans from I_ℓ to I_M, then we redefine r_ℓ as the rightmost $2/3$-high item in I_M, or $r_\ell = r$ if there is no $2/3$-high item completely in I_M. On the other hand, if there is a rectangle r that intersects with $x = x_r - \varepsilon$, i.e., r spans from I_M to I_r, then we redefine r_r as the leftmost $2/3$-high item completely in I_M, or $r_r = r$ if no $2/3$-high item is completely in I_M.

P now (after line 6 of the algorithm) has the following properties.

- The areas to the left of r_ℓ and to the right of r_r are almost completely covered by $2/3$-high items, i.e., $w(\mathrm{AI}(0, x'_\ell; 0, 1) \cap H_{2/3}) > x'_\ell - 4\varepsilon$ and $w(\mathrm{AI}(x_r, 1; 0, 1) \cap H_{2/3}) > 1 - x_r - 4\varepsilon$.
- The x-coordinates of the sides of all $1/3$-high items are in I_ℓ, I_M or I_r.
- We have $x_r - x'_\ell > 143\varepsilon$, since otherwise $w(H_{2/3}) \geq w(\mathrm{AI}(0, x'_\ell; 0, 1) \cap H_{2/3}) + w(\mathrm{AI}(x_r, 1; 0, 1) \cap H_{2/3}) \geq x'_\ell - 4\varepsilon + 1 - x_r - 4\varepsilon \geq 1 - 151\varepsilon \geq 27/28$ for an $\varepsilon < 1/(28 \cdot 151)$.

The first property follows from the inapplicability of Lemma 1 and the observation that only uncovered area of total width 3ε in $[x'_\ell, x'_\ell + \varepsilon]$ (for the now outdated value of x'_ℓ) and $[1/2 - \varepsilon, 1/2 + \varepsilon]$ can be added if we redefine r_ℓ and/or r_r.

The specific method that we apply in the next step depends on the existence of $1/3$-high items that span across the intervals I_ℓ, I_M and I_r. See Figure 2 for a schematic illustration of the following four cases (by the considerations above, all $1/3$-high items that span across the intervals have height at most $2/3$).

- A $1/3$-high item reaches close to r_ℓ and r_r—see Figure 2(a).
  In this case we assume that there is a $1/3$-high item r_1 that intersects with $x = x'_\ell + \varepsilon$ and with $x = x_r - \varepsilon$, i.e., that spans from I_ℓ to I_r. By Inequality (1) we have $w_1 \leq 1 - 130\varepsilon$ as $h_1 > 1/3$. Moreover, we have $w_1 \geq x_r - \varepsilon - x'_\ell - \varepsilon \geq 141\varepsilon$ (since $x_r - x'_\ell > 143\varepsilon$). Thus if $y_1 \geq 1/3$ or $y'_1 \leq 2/3$ we can apply the methods of Lemma 2. Otherwise, we can apply Lemma 3 with $c_1 = 10$ since $w_1 \geq x_r - \varepsilon - x'_\ell - \varepsilon \geq 141\varepsilon > (4c_1 + 1)\varepsilon$ (since $x_r - x'_\ell > 143\varepsilon$) and $w(H_{2/3}) \geq w(\mathrm{AI}(0, x'_\ell; 0, 1) \cap H_{2/3}) + w(\mathrm{AI}(x_r, 1; 0, 1) \cap H_{2/3}) \geq x'_\ell - 4\varepsilon + 1 - x_r - 4\varepsilon \geq 1 - w_1 - 10\varepsilon = 1 - w_1 - c_1\varepsilon$.

For space reasons we could not present the methods needed to solve the other cases. So we only state them here and refer to the full version of this paper for the required methods and for the details of the following remaining cases.

– Two $1/3$-high items lie between r_ℓ and r_r—see Figure 2(b).
– A $1/3$-high item reaches from the middle close to r_r but no $1/3$-high item reaches from r_ℓ to the middle (or vice versa)—see Figure 2(c).
– No $1/3$-high items span across the intervals—see Figure 2(d).

These four cases cover all possibilities and therefore our algorithm always outputs a packing into a strip of height at most $5/3 + 260\varepsilon/3$. Thus by scaling the solution we get an approximation ratio for the overall algorithm of $5/3 + 263\varepsilon/3$. By scaling ε appropriately we proved Theorem 1. The running time of the algorithm is $\mathcal{O}(T_{\mathcal{PTAS}} + (n \log^2 n)/\log \log n)$, where $T_{\mathcal{PTAS}}$ is the running time of the $\mathcal{PTAS}$ from [3].

References

1. Baker, B.S., Brown, D.J., Katseff, H.P.: A 5/4 algorithm for two-dimensional packing. Journal of Algorithms 2(4), 348–368 (1981)
2. Baker, B.S., Coffman Jr., E.G., Rivest, R.L.: Orthogonal packings in two dimensions. SIAM Journal on Computing 9(4), 846–855 (1980)
3. Bansal, N., Caprara, A., Jansen, K., Prädel, L., Sviridenko, M.: A structural lemma in 2-dimensional packing, and its implications on approximability. In: ISAAC: Proc. 20th International Symposium on Algorithms and Computation, pp. 77–86 (2009)
4. Bansal, N., Caprara, A., Sviridenko, M.: A new approximation method for set covering problems, with applications to multidimensional bin packing. SIAM Journal on Computing 39(4), 1256–1278 (2009)
5. Bansal, N., Correa, J.R., Kenyon, C., Sviridenko, M.: Bin packing in multiple dimensions: Inapproximability results and approximation schemes. Mathematics on Operation Research 31(1), 31–49 (2006)
6. Coffman Jr., E.G., Garey, M.R., Johnson, D.S., Tarjan, R.E.: Performance bounds for level-oriented two-dimensional packing algorithms. SIAM Journal on Computing 9(4), 808–826 (1980)
7. Golan, I.: Performance bounds for orthogonal oriented two-dimensional packing algorithms. SIAM Journal on Computing 10(3), 571–582 (1981)
8. Harren, R., Jansen, K., Prädel, L., van Stee, R.: A $(5/3 + \epsilon)$-approximation for strip packing. Technical Report 1105. University of Kiel (2011), http://www.informatik.uni-kiel.de/en/ifi/research/technical-reports/
9. Harren, R., van Stee, R.: Improved absolute approximation ratios for two-dimensional packing problems. In: Dinur, I., Jansen, K., Naor, J., Rolim, J. (eds.) APPROX 2009 and RANDOM 2009. LNCS, vol. 5687, pp. 177–189. Springer, Heidelberg (2009)
10. Jansen, K., Prädel, L., Schwarz, U.M.: Two for one: Tight approximation of 2d bin packing. In: WADS: Proc. Workshop on Algorithms and Data Structures, pp. 399–410 (2009)
11. Jansen, K., Solis-Oba, R.: Rectangle packing with one-dimensional resource augmentation. Discrete Optimization 6(3), 310–323 (2009)
12. Jansen, K., Thöle, R.: Approximation algorithms for scheduling parallel jobs: Breaking the approximation ratio of 2. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 234–245. Springer, Heidelberg (2008)
13. Jansen, K., Zhang, G.: Maximizing the total profit of rectangles packed into a rectangle. Algorithmica 47(3), 323–342 (2007)
14. Kenyon, C., Rémila, E.: A near optimal solution to a two-dimensional cutting stock problem. Mathematics of Operations Research 25(4), 645–656 (2000)

15. Schiermeyer, I.: Reverse-fit: A 2-optimal algorithm for packing rectangles. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 290–299. Springer, Heidelberg (1994)
16. Sleator, D.D.: A 2.5 times optimal algorithm for packing in two dimensions. Information Processing Letters 10(1), 37–40 (1980)
17. Steinberg, A.: A strip-packing algorithm with absolute performance bound 2. SIAM Journal on Computing 26(2), 401–409 (1997)

Reversing Longest Previous Factor Tables is Hard[*]

Jing He, Hongyu Liang, and Guang Yang

IInstitute for Interdisciplinary Information Sciences,
Tsinghua University, Beijing, China
{he-j08,lianghy08}@mails.tsinghua.edu.cn, 2006yangguang@gmail.com

Abstract. The Longest Previous Factor (LPF) table of a string s of
length n is a table of size n whose i^{th} element indicates the length of
the longest substring of s starting from position i that has appeared
previously in s. LPF tables facilitate the computing of the Lempel-Ziv
factorization of strings [21,22] which plays an important role in text com-
pression. An open question from Clément, Crochemore and Rindone [4]
asked whether the following problem (which we call the *reverse LPF
problem*) can be solved efficiently: Given a table W, decide whether it is
the LPF table of some string, and find such a string if so.

In this paper, we address this open question by proving that the re-
verse LPF problem is NP-hard. Thus, there is no polynomial time algo-
rithm for solving it unless $P = NP$. Complementing with this general
hardness result, we also design a linear-time online algorithm for the
reverse LPF problem over input tables whose elements are all 0 or 1.

1   Introduction

The concept of Longest Previous Factor tables is introduced by Crochemore
and Ilie [7]. Let s be a string of length n. The Longest Previous Factor (LPF)
table of s, denoted by LPF_s, is a table of length n defined as follows: For all
$1 \leq i \leq n$, the i-th element of LPF_s (denoted as $LPF_s[i]$) is the length of the
longest substring of s beginning at position i that appears previously in s. More
formally, we have

$$LPF_s[i] = \max\{k \mid \exists 1 \leq j < i, \ s[i, i+k-1] = s[j, j+k-1]\}, \qquad (1)$$

where $s[i, j]$ denotes the substring of s starting from position i and ending at
position j (and it means the empty string if $i > j$).

Using LPF tables one can easily compute the Lempel-Ziv factorization of a
string [21,22], which is of great importance in lossless data compression and
dictionary compression methods. Other applications of LPF tables include de-
tecting all runs [17], computing leftmost maximal periodicities [18] and testing
square freeness of a string [5].

Various types of table-like data structures are very useful in designing algorithms on strings, trees or sequences. The Border table is an essential part of the famous KMP algorithm for string-matching [16]. The Suffix table plays an important role in several string algorithms, including Boyer-Moore algorithm [3], Apostolico-Giancarlo algorithm [1,9] and Gusfield's algorithm [14]. The Prefix table, a dual notion of the Suffix table, also has many applications [6]. All these tables are computable in linear time [16,3,6]. Like others, the LPF table of a given string of length n can be computed in time $O(n)$ independent of the alphabet size. Crochemore and Ilie [7] gave two simple algorithms for computing LPF tables using the Suffix Array [19] and the Longest Common Prefix table.

We study in this paper the *reverse engineering problem* on Longest Previous Factor tables. The *reverse LPF problem* is defined as: Given an integer table W, test whether W is the LPF table of some string s, and return such a string if so. Clément, Crochemore and Rindone [4] raised the open question that whether the reverse LPF problem can be solved efficiently. We give a negative answer to this question (with the belief of $P \neq NP$), by proving that the reverse LPF problem is actually NP-hard. Our proof is by a reduction from the classical NP-complete problem 3-SAT. Since the reduction itself is quite complicated, we only show some intuitive ideas in this extended abstract. The technical details will appear in the full version of this paper.

Our result actually exhibits a natural widely-used string-related data structure for which the reverse problem cannot be solved efficiently. Previously this type of reverse problems have been considered by many authors, and they almost all proposed polynomial-time algorithms, or even linear-time algorithms, for the corresponding problems. Franek et. al. [11] and Duval et. al. [10] both gave a linear-time verification of Border tables, for unbounded alphabets and bounded alphabets respectively. Clément, Crochemore and Rindone [4] provided a linear-time algorithm for characterizing Prefix tables. Gawrychowski, Jeż and Jeż [13] proposed a linear-time algorithm for checking whether a given array is the failure function of the Knuth-Morris-Pratt algorithm (see [16]) for some string. Linear-time algorithms for reversing Suffix tables have been proposed by Bannai et al. [2] and Franek and Smyth [12]. The Parameterized Border tables are considered by I et al. [15] and an $O(n^{1.5})$ time algorithm is given. Crochemore et al. studied minimal- and maximal- cover arrays [8] and designed linear-time algorithms for reversing both arrays. To our knowledge, the only known hard reverse problem prior to ours is that of inferring a string from the set of its runs, due to Matsubara, Ishino and Shinohara [20].

Complementing with the intractability result, we also find a class of nontrivial integer tables, namely the tables whose elements are all 0 or 1, on which the reverse LPF problem can be solved in linear time. Our algorithm runs online; that is, assuming that the elements of the input table is fed to the algorithm one by one, our algorithm will maintain a string which is an LPF-string of the current input table, or a special signal indicating that such string does not exist. We also give a simple yet complete characterization of 0-1 LPF tables.

2   Preliminaries

2.1   Notations and Definitions

A string w with alphabet Σ is a sequence $(w[1], w[2], \cdots, w[n])$ with $w[i] \in \Sigma$ for all $1 \leq i \leq n$. Call n the *length* of w, denoted as $|w| = n$. We say $w[i]$ is the *i*-th *element* of w, or, $w[i]$ is at *position i* in w. Let $w_1 w_2$ be the concatenation of w_1 and w_2. Thus, we can write $(w[1], w[2], \ldots, w[n]) = w[1]w[2] \cdots w[n]$. Denote $w^k = \underbrace{ww \ldots w}_{k \text{ copies of } w}$ for $k \in \mathbb{N}$. A string of the form c^k, where c is a single character, is called a *unary string of c*, or simply a *unary string*. Given a string w, let $w[i, j] = w[i]w[i+1] \cdots w[j]$ denote the *substring* (or alternatively called *factor*) of w starting from position i to j. If w' is a factor of w, we say that w' *appears* in w. If w' starts from position $i' < i$, we say that w' *appears in w before position i*. We also say $w[i, j]$ *appears before* $w[i', j']$ if $i < i'$. If uv is a factor of w, we call u a *length-$|u|$ predecessor* of v in w, and call v a *length-$|v|$ successor* of u in w. A length-1 predecessor (resp. successor) is simply called a *predecessor* (resp. *successor*).

A string W is called an *integer table*, or simply a *table*, if its alphabet is a subset of the set of non-negative integers, i.e. $\Sigma(W) \subseteq \mathbb{N}$. An integer table W is called *LPF-reversible* if there exists a string w satisfying $LPF_w = W$, and such w is called an *LPF-string* of W.

2.2   Simple Properties of LPF Tables

Let w be a string of length n. Trivially $LPF_w[1] = 0$. It is also clear that $LPF_w[i] = 0$ if and only if $w[i]$ is a new character appeared in w (that is, $w[i] \notin \{w[j] \mid 1 \leq j < i\}$), and $LPF_w[i] \leq 1$ (for $i < n$) if and only if the length-2 factor $w[i, i+1]$ does not appear in w before position i. Furthermore, it holds that $LPF_w[i] \leq n - i + 1$ and $LPF_w[i] \geq LPF_w[i-1] - 1$ (we let $LPF_w[0] = 0$). Other important observations include:

- $LPF_w[i] \leq 1$, then $LPF_w[1, i] = LPF_{w[1,i]}$;
- If $LPF_w[i] > 1$, then for every $1 \leq j \leq i$, $LPF_{w[1,i]}[j] = \min(LPF_w[j], i - j + 1)$.

This statement enables us to "extract" a prefix from the whole table, which is very useful in our construction. The proof is omitted due to lack of space.

3   Reduction from 3-SAT

In this section we demonstrate the main theorem of this paper.

Theorem 1. *The problem of deciding whether a given integer table is LPF-reversible is NP-complete.*

This problem is the decision version of the reverse LPF problem, and is in NP since we can verify the validity of a witness (an LPF-string of the given table) in linear time [7]. Thus we only need to present a polynomial-time reduction from an NP-complete problem to it. We choose the classical problem 3-SAT.

Let $\mathcal{F}$ be a 3-CNF formula with variable set $\{X_i \mid 1 \leq i \leq n\}$ and clause set $\{C_j \mid 1 \leq j \leq m\}$. W.l.o.g. we assume $n \geq 3$ and $m \geq 3$. We wish to construct an integer table W such that W is LPF-reversible if and only if $\mathcal{F}$ is satisfiable. Our construction of W basically consists of three consecutive sections:

- **Variable Section**. In this part we add segments corresponding to the variables.
- **Assignment Section**. This part contains gadgets for encoding assignments of the variables.
- **Checking Section**. This part is used for checking whether the corresponding assignment satisfies the formula $\mathcal{F}$.

We will carefully design the three sections to ensure that any LPF-string of W corresponds to a satisfying assignment for $\mathcal{F}$, and vice versa.

3.1   Variable Section

In the variable section, we aim to construct $2n + 2$ characteristic segments for the following $2n+2$ literals and Boolean values: $\{X_i \mid 1 \leq i \leq n\} \cup \{\overline{X_i} \mid 1 \leq i \leq n\} \cup \{T, F\}$. For the simplicity of expressions, we denote $X_{n+i} = \overline{X_i}$, $X_{2n+1} = T$ and $X_{2n+2} = F$, and temporarily call all of them variables (instead of literals and Boolean values).

Since LPF tables contain only information about lengths of factors, a natural approach is to use different lengths to distinguish between variables. Our target is to use long unary substrings, of distinct characters and lengths, to encode different variables. Since what we are going to construct is the LPF table, it must contain some special structure that can "force" its LPF-string to contain long unary substrings.

To achieve this, we plant some highly overlapping structures in the LPF table. As an example, consider what happens if the LPF table contains three consecutive entries $100, 101, 102$. From the definition, these numbers mean the length of the longest previous factors for these three positions. But we can in fact learn more. We know that the three substrings, starting from these three positions with length $100, 101, 102$ respectively, have a common substring r of length 98 (that starts with the position of entry 102). This tells us that three copies of r have appeared before them. Moreover, these copies must start from distinct position. To see this, w.l.o.g. assume the two copies of r corresponding to 100 and 101 coincide with each other. Then by the definition of LPF tables, there exists a previous factor of length 102 for the entry 100, a contradiction! Thus, we have three copies of the same long string. In order to get our desired unary substring, it suffices to guarantee the existence of two copies of r such that their starting positions only differ by a small distance.

Now comes the specific constructions. We choose $2n+2$ *odd* integers $\{x_i \mid 1 \leq i \leq 2n+2\}$ as follows:

$$(\forall 1 \leq i \leq 2n)\ x_i = 8n + (6i - 2)m + 6i^2 - 6i - 9; \tag{2}$$

$$x_{2n+1} = 2(24n^2 + 12mn + 34n + 8m - 9) + 1; \tag{3}$$

$$x_{2n+2} = 2(24n^2 + 12mn + 58n + 14m + 3) + 1. \tag{4}$$

We call x_i the *characteristic length* of variable X_i. Their values given above are carefully chosen so that they have a "safe-but-not-too-long" distance from each other. Now we formally define W_{VS}, the variable section of W. It comprises $2n + 2$ substrings each of which encodes one of the variables.

Definition 1. *For every* $1 \leq i \leq 2n+2$, *let*

$$W_{VS}^{(i)} := (0, 0, x_i{-}1, x_i{-}2, \cdots, 1, 1^{12i-7}, 0, x_i, x_i{+}1, x_i{+}12i{-}5, x_i{+}12i{-}6, \cdots, 1, 1^{6i-1}). \tag{5}$$

(Recall that 1^k *means a unary string of 1 of length* k.)
Define the variable section W_{VS} *as:*

$$W_{VS} := (0,\ 1,\ W_{VS}^{(1)},\ W_{VS}^{(2)},\ \cdots,\ W_{VS}^{(2n+2)}).$$

The next lemma asserts that any LPF-string of W_{VS} has a "fixed" structure.

Lemma 1. W_{VS} *is LPF-reversible. Moreover, if* w_{VS} *is its LPF-string, then:*

$$w_{VS} = (a,\ a,\ w_{VS}^{(1)},\ w_{VS}^{(2)},\ \cdots,\ w_{VS}^{(2n+2)}),$$

where, for every $1 \leq i \leq 2n+2$,

$$w_{VS}^{(i)} = (b_i,\ c_i^{x_i},\ u_i,\ d_i,\ c_i^{x_i+2},\ u_i,\ d_i,\ c_i,\ v_i),$$

$(w_{VS}^{(i)}$ *corresponds to the factor* $W_{VS}^{(i)}$), *such that:*

(a) $\{a\} \cup \{b_j, c_j, d_j \mid 1 \leq j \leq 2n+2\}$ *is a set of* $6n + 7$ *distinct characters;*
(b) u_i *and* v_i *are strings that end up with the character "a" and do not contain new characters. Moreover, all length-2 factors of* u_i *are exactly those length-2 strings (with already appeared characters) that do not appear in* W_{VS} *before (the first occurrence of)* u_i; *similarly, all length-2 factors of* v_i *consists of precisely those length-2 strings (with old characters) that do not appear in* W_{VS} *before* v_i.

The rigorous proof is omitted from this short version, and here we just give some intuitions. The crux of the proof is to analyze, as stated before the explicit construction, the common substring r of length $x_i - 2$ resulted from the highly overlapping structures beginning with $(x_i, x_i + 1, x_i + 12i - 5)$. We wish to force the LPF-string to contain long unary substrings ($c_i^{x_i}$ and $c_i^{x_i+2}$ in $w_{VS}^{(i)}$, defined in Lemma 1), which will be regarded as the "encoding" of the corresponding variable. We hope to prove that there are two copies of r whose starting positions

differ only by 1. The choice of $\{x_i\}$ is to ensure that x_{i+1} is large enough so that such "long repeated pattern" cannot appear in the previous parts $w_{VS}^{(j)}, j < i$. Another gadget is simply the 0 before $(x_i, x_i + 1, x_i + 12i - 5)$. Since this 0 corresponds to a new character d_i, it ensures that, if all three copies of r start before d_i, then they also end before d_i. To see this, just notice that if one of them contains d_i, so do the other two; but then they must locate at the same position, contradicting the previous analysis. Furthermore, they must end before the 1^{12i-7} part, since an entry "1" in the LPF table means that the length-2 substring starting from it has not appeared before, but there are three copies of r at different positions! Therefore, if all three copies of r start before d_i, then they are restricted in a small range $(0, 0, x_i - 1, x_i - 2, \ldots, 1)$, which will lead to contradictions by simple arguments. Thus, at least one of the three r's appear after d_i. However, since they are "previous factors" of the substring starting from the position of $x_i + 12i - 5$, there are only two possible starting positions for them. Thus, a case-by-case investigation will help us find out the only possibility of their positions, which turns out to be exactly what we want. Finally, the 1^{6i-1} segment at the end of $w_{VS}^{(i)}$ serves as a "clean-up" procedure, which ensures that all the results for $w_{VS}^{(i)}$ similarly apply to $w_{VS}^{(i+1)}$, thus leading to an inductive proof of the lemma.

It should be noticed that this "proof pattern", of utilizing highly overlapping structures and carefully designed gadgets (for example, using new characters as "separators" that cannot be contained in the considered substring) to force the LPF-string to have a "good" form, is the most significant idea of the whole proof.

We end up this section by giving a formal definition of the *characteristic segment* of a variable, which encodes the corresponding variable as a special factor in the LPF-string. This concept will frequently appear in the following sections.

Definition 2. *A factor of W's LPF-string is called a* characteristic segment *of X_i, if it has the form $(\alpha, c_i^{x_i + \delta}, u_i[1])$ for some $\delta \geq 0$, where α is a new character that has not appeared before. (Recall that x_i is the characteristic length of X_i defined before, and u_i and c_i are defined in Lemma 1.)*

3.2   Assignment Section

The goal of the assignment section is to associate each variable X_i, $1 \leq i \leq 2n$, with a Boolean value T or F. In this section, we provisionally "forget" the relation between X_i and $X_{n+i}(= \overline{X_i})$, and just regard them as independent variables. The legality of the assignment will be checked in the next part, namely the checking section. Since "assignment" is the theme of this section, the two Boolean values T and F will be highlighted. Thus, for the sake of clearness, we write T and F as the subscripts corresponding to the two values, instead of using $2n + 1$ and $2n + 2$ as before. More formally, for $I \in \{x, b, c, d, u\}$, let I_T and I_F denote I_{2n+1} and I_{2n+2}, respectively. For example, $x_T = x_{2n+1}$ and $d_F = d_{2n+2}$. We also write $T = c_T(= c_{2n+1})$ and $F = c_F(= c_{2n+2})$ for convenience, i.e., we

do not distinguish between the two Boolean values and the characters associated with them. (Recall that c_i is the character used for encoding X_i.)

For every variable X_i, we first construct its characteristic segment, and then append to it a "value segment" of length y_i, with the hope that the corresponding part of the LPF-string will be forced to have the form $t_i^{y_i}$ where $t_i \in \{T, F\}$; t_i is then interpreted as the assigned Boolean value to X_i. Once we want to inquire for the value of X_i, we construct a special gadget to extract t_i. Due to some technique issues, the lengths of the value segments $\{y_i \mid 1 \le i \le 2n\}$ must be carefully selected. We choose them as:

$$(\forall 1 \le i \le 2n) \ y_i = 24n^2 + 12mn + 28n + 8m - 14 + 3i. \qquad (6)$$

We need some other definitions to make the statement below clearer. Given a string w and one of its unary substring s (recall that a unary string is one that has the form c^k), we say u is a *non-trivial predecessor* (resp. *non-trivial successor*) of s in w if u is a predecessor (resp. successor) of the maximal unary substring of w that contains s as a factor. For example, in the string $abcaaaade$, we say c is a non-trivial predecessor of aa, and d is a nontrivial successor of aaa.

We now define the assignment section of W.

Definition 3. *For every $1 \le i \le 2n$, let*

$$W_{AS}^{(i)} = (0, \ x_i+2, \ x_i+3, \ x_i+3, \ x_i+2, \ \cdots, \ 1, \ 0, \ y_i+1, \ y_i+1, \ y_i, \ y_i-1, \ \ldots, \ 1).$$

Define the assignment section W_{AS} *as*

$$W_{AS} := (W_{AS}^{(1)}, \ W_{AS}^{(2)}, \ \cdots, \ W_{AS}^{(2n)}).$$

Lemma 2. *Let W_{VS} and W_{AS} be defined as in Definitions 1 and 3. Then the string $W_{VS}W_{AS}$ is LPF-reversible. Furthermore, suppose $w_{VS}w_{AS}$ is an LPF-string of it with $|w_{VS}| = |W_{VS}|$ (and hence $|w_{AS}| = |W_{AS}|$). Then, w_{VS} has the same form as stated in Lemma 1, and*

$$w_{AS} = (w_{AS}^{(1)}, \ w_{AS}^{(2)}, \ \cdots, \ w_{AS}^{(2n)}),$$

where for every $1 \le i \le 2n$,

$$w_{AS}^{(i)} = (e_i, \ c_i^{x_i+4}, \ u_i[1], \ f_i, \ r_i, \ t_i^{y_i}, \ s_i),$$

($w_{AS}^{(i)}$ corresponds to the factor $W_{AS}^{(i)}$), such that:

(a) e_i, f_i are new characters never appeared before;
(b) $t_i \in \{T, F\}$;
(c) r_i and s_i are respectively non-trivial predecessor and successor of $t_i^{y_i}$ in w_{VS}.

The key point of the proof of Lemma 2 is to show that the substring $(0, \ x_i + 2, \ x_i + 3, \ x_i + 3, \ x_i + 2, \ \cdots, \ 1, \ 0)$ of $W_{AS}^{(i)}$ must correspond to $e_i c_i^{x_i+4} u_i[1]$ (a characteristic segment of X_i) in its LPF-string. This special structure enables

us to "extract" c_i, the encoding of the variable X_i, which is very useful since we need to deal with the variables everywhere in the whole reduction. The proof is omitted from this short version.

Up till now, we have successfully associated each variable X_i with a Boolean value t_i by appending a gadget string, which is forced to have the form $t_i^{y_i}$, to the characteristic segment of X_i. However, it is possible that both X_i and $\overline{X_i}$ are assigned with the same value, resulting in an infeasible assignment. We will deal with this issue in the checking section.

3.3 Checking Section

The preceding parts of our construction involved no logical connections between variables. The assignment section can encode any possible assignment to $\{X_i \mid 1 \leq i \leq 2n\}$, including those invalid ones (i.e. $X_i = X_{n+i}$ for some i). Moreover, $W_{VS}W_{AS}$ is always LPF-reversible regardless of whether $\mathcal{F}$ is satisfiable or not. The checking section is designed to handle these problems. Recall that $\{t_i \mid 1 \leq i \leq 2n\}$ is an assignment to $\{X_i \mid 1 \leq i \leq 2n\}$ induced by the assignment section W_{AS}. The checking section W_{CS} consists of 3 consecutive substrings $W_{CS1}, W_{CS2}, W_{CS3}$. We use W_{CS1} (resp. W_{CS2}) to guarantee that W is LPF-reversible implies at least one of t_i and t_{n+i} is F (resp. T). Hence, the first two parts enforce the assignment to be valid. The last part W_{CS3} is to ensure that for every clause C_j, at least one of its literals is assigned with T, if W is LPF-reversible. On the other hand, it is fairly easy to prove that the satisfiability of $\mathcal{F}$ implies the LPF-reversibility of W (just follow the LPF-strings defined in the lemmas). Thus, combining three parts together assures us that W is LPF-reversible if and only if $\mathcal{F}$ is satisfiable.

In fact, the constructions of the three parts have similar structures; their tasks are all to check if there is a certain value (T or F) among the assigned values of several (in fact 2 or 3) variables. Therefore, in this extended abstract we only provide the definition and results of W_{CS1}. The constructions of W_{CS2} and W_{CS3}, as well as the rigorous proofs, will appear in the full paper.

We shall briefly explain the (somewhat seemingly unnatural) technique used to guarantee at least one pre-specified value (say T) among some variables. We construct some gadgets in such a way that each of them implies the existence of a distinct, previously appeared character-pair (p, q), where p and q are, respectively, non-trivial predecessor and successor of a long unary string of T or F (that is, T^z or F^z for some large integer z). Thus, some number of gadgets imply the same number of such non-trivial (predecessor, successor)-pairs of T^z or F^z. We can actually calculate the number of such pairs in our construction till now. We then create some new successors by appending new characters to long unary strings of the values that we need to check; note that we don't know whether the values are T or F, but we want at least one of them to be T. By some careful design, we make sure that there are enough number of such pairs if and only if at least one of the new successors is a successor of T^z; in other words, at least one of the values we consider is T. Some further discussions are given after the statement of Lemma 3.

We now formally define the first part of our checking section.

Definition 4. *For each $i \in \{1, 2, \ldots, n\}$, define*

$$W_{CS1}^{(i)} := (0, \ x_i + 4, \ x_i + 5, \ x_i + 8 + z_i, \ x_i + 7 + z_i, \ \ldots, \ 1, \ 0,$$
$$0, \ x_{n+i} + 4, \ x_{n+i} + 5, \ x_{n+i} + 8 + z_i, \ x_{n+i} + 7 + z_i, \ \ldots, \ 1, \ 0,$$
$$(0, \ z_i + 1, \ z_i + 1, \ z_i, \ \cdots, \ 1)^{10}).$$

Define the first part of the checking section as:

$$W_{CS1} := (0, \ 0, \ x_F + 2, \ x_F + 3, \ x_F + 3, \ x_F + 2, \ \ldots, \ 1,$$
$$W_{CS1}^{(1)}, \ W_{CS1}^{(2)}, \ \cdots, \ W_{CS1}^{(n)}).$$

The substring $W_{CS}^{(i)}$ is also called a *block*, which is the minimum unit of a complete verification of the assignment for one variable. $W_{CS}^{(i)}$ is to check that at least one of X_i and $X_{n+i}(= \overline{X_i})$ is assigned with F.

To avoid the circumstances where successors introduced by former blocks interfere with the following blocks, we need to choose $\{z_i \mid 1 \leq i \leq 2n + m\}$ with enough distances as follows: (Here $\{z_i \mid n + 1 \leq i \leq 2n\}$ is used for the second part of checking section, and $\{z_i \mid 2n + 1 \leq i \leq 2n + m\}$ is used for the third one; recall that the 3-CNF formula $\mathcal{F}$ has m clauses.)

$$(\forall 1 \leq i \leq 2n + m) \ z_i = 24n^2 + 12mn + 20n + 4m - 13 + 4i.$$

Lemma 3. *If $W_{VS}W_{AS}W_{CS1}$ is LPF-reversible, then, for any LPF-string w of it, $w = w_{VS}w_{AS}w_{CS1}$ holds, where w_{VS} and w_{AS} are defined as in Lemmas 1 and 2, and w_{CS1} has the following form:*

$$w_{CS1} = (g_F, \ r_F, \ F^{x_F+4}, \ u_F[1], w_{CS1}^{(1)}, \ w_{CS1}^{(2)}, \ldots, \ w_{CS1}^{(n)}),$$

where, for every $1 \leq i \leq n$,

$$w_{CS1}^{(i)} = (g_i, \ c_i^{x_i+6}, \ u_i[1], \ f_i, \ r_i, \ t_i^{z_i+1}, \ h_i,$$
$$g_{n+i}, \ c_{n+i}^{x_{n+i}+6}, \ u_{n+i}[1], \ f_{n+i}, \ r_{n+i}, \ t_{n+i}^{z_i+1}, \ h_{n+i},$$
$$g_{i,1}, \ r_{i,1}, \ t_{i,1}^{z_i}, \ s_{i,1},$$
$$\vdots$$
$$g_{i,10}, \ r_{i,10}, \ t_{i,10}^{z_i}, \ s_{i,10}),$$

($w_{CS1}^{(i)}$ corresponds to the factor $W_{CS1}^{(i)}$), such that:

(a) $\{g_i, h_i \mid 1 \leq i \leq 2n\} \cup \{g_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq 10\} \cup \{g_F, r_F\}$ is a set of distinct new characters;

(b) $\forall 1 \leq i \leq n, \forall 1 \leq j \leq 10, \ t_{i,j} \in \{T, F\}$ and $r_{i,j}, s_{i,j}$ are previously appeared non-trivial predecessor and successor of $t_{i,j}^{z_i}$;

(c) $\forall 1 \leq i \leq n$, at least one of t_i and t_{n+i} must be F.

Conversely, if there exists a string $w = w_{VS}w_{AS}w_{CS1}$ of the above form, then $W_{VS}W_{AS}W_{CS1}$ is LPF-reversible and w is its LPF-string.

An important step in proving Lemma 3 is to show that any substring $(0, x_i + 4, x_i + 5, x_i + 8 + z_i, x_i + 7 + z_i, \ldots, 1, 0)$ of $W_{CS1}^{(i)}$ must correspond to $(g_i, c_i^{x_i+6}, u_i[1], f_i, r_i, t_i^{z_i+1}, h_i)$ in its LPF-string. This allows us to extract c_i as well as t_i, the assigned value of the variable X_i, which is important for later use.

We now give some intuitive ideas of how such scheme works. As claimed before, the values of X_i and $\overline{X_i}$ (that is, t_i and t_{n+i}) can be extracted out, and h_i, h_{n+i} correspond to non-trivial successors of $t_i^{z_i}$ and $t_{n+i}^{z_i}$, respectively. With similar analysis to the proof of Lemma 2, every component $(0, z_i + 1, z_i + 1, z_i, \cdots, 1)$ in $W_{CS1}^{(i)}$ corresponds to $g_{i,j} r_{i,j} t_{i,j}^{z_i} s_{i,j}$ in the LPF-string. Thus, $(r_{i,j}, s_{i,j})$ is a non-trivial (predecessor, successor)-pair of T^{z_i} or F^{z_i} that appeared before. We can also argue that these 10 pairs should be pairwise different, which implies the existence of 10 different non-trivial (predecessor, successor)-pairs of T^{z_i} or F^{z_i}. We calculate the number of such predecessors and successors that have appeared so far: T^{z_i} has b_T, d_T as its predecessors, and $u_T[1]$ as a successor; F^{z_i} has predecessors b_F, d_F, r_F and a successor $u_F[1]$; also, there are two successors h_i and h_{n+i} (of $t_i^{z_i}$ and $t_{n+i}^{z_i}$, respectively) yet to be determined whose successor they actually are. On the one hand, if both h_i, h_{n+i} are successors of T^{z_i}, there are $2 \times 3 + 3 \times 1 = 9$ (predecessor, successor)-pairs in all, contradicting the requirement of 10 pairs. On the other hand, it is easy to verify the existence of 10 or 11 such pairs when at least one of h_i and h_{n+i} is a successor of F^{z_i}. Thus, when $W_{VS}W_{AS}W_{CS1}$ is LPF-reversible, at least one of h_i and h_{n+i} should be a successor of F^{z_i}, implying that at least one of t_i and t_{n+i} should be F. The converse direction is only a matter of tedious verification.

Applying similar techniques to construct W_{CS2} and W_{CS3} completes our reduction. The details of the constructions of W_{CS2} and W_{CS3} are omitted due to lack of space, and will appear in the full paper. The final (minor) step for proving Theorem 1 is to show that the reduction runs in polynomial time, which directly follows from our construction. We can also strengthen Theorem 1 as follows by a padding argument.

Theorem 2. *For every constant $\epsilon > 0$, it is NP-complete to decide whether a table of length n with at most n^ϵ zeros is LPF-reversible.*

Note that we need an unbounded size alphabet in our hardness proof. It is interesting to see what happens if the alphabet size is bounded. Formally, we propose the following open question.

Question 1. Is it NP-complete to decide whether a table that contains at most k zeros is LPF-reversible, where k is a fixed integer?

4   LPF Tables with 0-1 Entries

An integer table whose elements are all 0 or 1 is called a *0-1 table.* In this section, we prove that the reverse LPF problem is solvable in linear time over

0-1 tables. Moreover, we give a complete characterization of LPF-reversible 0-1 tables (Theorem 3). Given a table W and an integer i, define

$$Num(W, i) := |\{j \mid W[j] = i, 1 \leq j \leq |W|\}|,$$

which is the number of elements in W that are equal to i.

Theorem 3. *A 0-1 table W of length n is LPF-reversible if and only if*

$$W[1] = 0 \ and \ (\forall 2 \leq i \leq n) \ i \leq (Num(W[1, i], 0))^2 + 1 \ . \tag{7}$$

Theorem 4. *There is a linear-time online algorithm for the reverse LPF problem on 0-1 tables.*

It is easy to see that Theorem 3 implies the decision part of Theorem 4, since we can test whether (7) holds or not in linear time simply using two counters. The construction of an LPF-string in linear time needs more effort.

The "only if" direction of Theorem 3 is easy. Suppose the input 0-1 table W of length n is LPF-reversible and let w be an LPF-string of W. Since $W[i] \leq 1$ for each i, we know from Section 2.2 that for every $i \in \{1, 2, \ldots, n-1\}$, the length-2 factor $W[i, i+1]$ does not appear in t before position i. Thus, $w[1, i]$ contains $i-1$ distinct length-2 factors, whereas the total number of such factors is $(Num(T[1, i], 0))^2$. We therefore have $i - 1 \leq (Num(W[1, i], 0))^2$ for each $2 \leq i \leq n$. Together with the trivial fact that $W[1] = 0$, the "only if" part of Theorem 3 follows.

The idea of proving the "if" direction is to show the equivalence between the LPF-reversibility of W and the existence of a "partial" Eulerian path with some constraints in a directed graph associated with W. Let $ZeroW = \{i \mid W[i] = 0, 1 \leq i \leq n\}$ and $m = |ZeroW|$. Assume $ZeroW = \{i_1, i_2, \ldots, i_m\}$ where $1 = i_1 < i_2 < \ldots < i_m$ (by Equation (7) we have $1 \in ZeroW$). Let G be a complete directed graph on the vertex set $V = \{1, 2, \ldots, m\}$ with all self-loops included. A (not necessarily simple) path in G is called a *partial Eulerian path* if it traverses each edge at most once. We can prove that the following two statements are equivalent:

1. W is LPF-reversible.
2. G has a partial Eulerian path P of length $n - 1$ (thus it contains n vertices), such that for every $1 \leq j \leq m$, the i_j-th vertex on P is j, and it has not been visited by P before.

The linear time online algorithm for the reverse LPF problem on 0-1 tables is just based on a linear time algorithm for finding such a partial Eulerian path. The detailed proofs of Theorems 3 and 4 will appear in the full version of this paper.

Acknowledgements

References

1. Apostolico, A., Giancarlo, R.: The Boyer-Moore-Galil string searching strategies revisited. SIAM J. Comput. 15(1), 98–105 (1986)
2. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M.: Inferring strings from graphs and arrays. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 208–217. Springer, Heidelberg (2003)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commin. ACM 20(10), 762–772 (1977)
4. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: STACS 2009, Freiburg, pp. 289–300 (2009)
5. Crochemore, M.: Transducers and repetitions. Theoret. Comput. Sci. 45(1), 63–86 (1986)
6. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press, Cambridge (2007)
7. Crochemore, M., Ilie, L.: Computing Longest Previous Factor in linear time and applications. Inf. Process. Lett. 106(2), 75–80 (2008)
8. Crochemore, M., Iliopoulos, C.S., Pissis, S.P., Tischler, G.: Cover Array string reconstruction. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 251–259. Springer, Heidelberg (2010)
9. Crochemore, M., Lecroq, T.: Tight bounds on the complexity of the Apostolico-Giancarlo algorithm. Inf. Process. Lett. 63(4), 195–203 (1997)
10. Duval, J.-P., Lecroq, T., Lefebvre, A.: Efficient validation and construction of border arrays. In: Proceedings of 11th Mons Days of Theoretical Computer Science, Rennes, France, pp. 179–189 (2006)
11. Franek, F., Gao, S., Lu, W., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a Border array in linear time. J. Combinatorial Math. and Combinatorial Computing 42, 223–236 (2002)
12. Franek, F., Smyth, W.F.: Reconstructing a Suffix Array. International Journal of Foundations of Computer Science 17(6), 1281–1295 (2006)
13. Gawrychowski, P., Jeż, A., Jeż, Ł.: Validating the Knuth-Morris-Pratt failure function, fast and online. In: Ablayev, F., Mayr, E.W. (eds.) CSR 2010. LNCS, vol. 6072, pp. 132–143. Springer, Heidelberg (2010)
14. Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, Cambridge (1997)
15. I., T., Inenaga, S., Bannai, H., Takeda, M.: Verifying a Parameterized Border Array in $O(n^{1.5})$ time. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 238–250. Springer, Heidelberg (2010)
16. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. 6(1), 323–350 (1977)
17. Kolpakov, R., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: FOCS 1999, pp. 596–604. IEEE Computer Society Press, New York (1999)
18. Main, M.G.: Detecting leftmost maximal periodicities. Discrete Applied Math. 25, 145–153 (1989)
19. Manber, U., Myers, G.: Suffix arrays: a new method for on-line search. SIAM J. Comput. 22(5), 935–948 (1993)
20. Matsubara, W., Ishino, A., Shinohara, A.: Inferring strings from runs. In: Prague Stringology Conference 2010, pp. 150–160 (2010)
21. Ziv, J., Lempel, A.: A Universal algorithm for sequential data compression. IEEE Trans. Inform. Theory 23, 337–342 (1977)
22. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inform. Theory 24, 530–536 (1978)

Space Efficient Data Structures for Dynamic Orthogonal Range Counting[*]

Meng He and J. Ian Munro

Cheriton School of Computer Science, University of Waterloo, Canada
{mhe,imunro}@uwaterloo.ca

Abstract. We present a linear-space data structure that maintains a dynamic set of n points with coordinates of real numbers on the plane to support orthogonal range counting, as well as insertions and deletions, in $O((\frac{\lg n}{\lg \lg n})^2)$ time. This provides faster support for updates than previous results with the same bounds on space cost and query time. We also obtain two other new results by considering the same problem for points on a $U \times U$ grid, and by designing the first succinct data structures for a dynamic integer sequence to support range counting.

1 Introduction

The two-dimensional ***orthogonal range counting*** problem is a fundamental problem in computational geometry. In this problem, we store a set, P, of n points in a data structure so that given a query rectangle R, the number of points in $P \cap R$ can be computed efficiently. This problem has applications in many areas of computer science, including databases and computer graphics, and thus has been studied extensively [4,11,13,12,3]. Among previous previous, Chazelle [4] designed a linear-space data structure for points with real coordinates to support orthogonal range counting in $O(\lg n)$ time[1], while the linear-space data structure of JáJá *et al.* [11] provides $O(\frac{\lg n}{\lg \lg n})$-time support for integer coordinates.

Researchers have also considered the orthogonal range counting problem in dynamic settings. The goal is to maintain a dynamic set, P, of n points to support orthogonal range counting, while allowing points to be inserted into and deleted from P. Chazelle [4] designed a linear-space data structure that supports orthogonal range counting, insertions and deletions in $O(\lg^2 n)$ time. Nekrich [12] designed another data structure of linear space with improved query time. With his data structure, a range counting query can be answered in $O((\frac{\lg n}{\lg \lg n})^2)$ time, matching the lower bound proved by Pătraşcu [13] under the group model, but it takes $O(\lg^{4+\epsilon} n)$ amortized time to insert or delete a point. Thus in this paper, we consider the problem of designing a linear-space data structure that matches Nekrich's query time while providing faster support for updates.

In addition to considering points on the plane, we also define range counting over a dynamic sequence $S[1..n]$ of integers from $[1..\sigma]$: given a range, $[i_1..i_2]$,

---

[*] This work was supported by NSERC and the Canada Research Chairs program.
[1] $\lg n$ denotes $\log_2 n$.

of indices and a range, $[v_1..v_2]$, of values, a range counting query returns the number of entries of $S[i_1..i_2]$ that store integers from $[v_1..v_2]$.[2] We are interested in designing **succinct data structures** to represent S. Succinct data structures were first proposed by Jacobson [10] to encode bit vectors, trees and planar graphs using space close to the information-theoretic lower bound, while supporting efficient navigation operations in them. As succinct data structures provide solutions to modern applications that process large data sets, they have been studied extensively [14,6,5,7,3,8].

1.1   Our Results

Under the word RAM model with word size $w = \Omega(\lg n)$, we present the following results:

1. A linear-space data structure that maintains a dynamic set, P, of n points on the plane to answers an orthogonal range counting query in $O((\frac{\lg n}{\lg \lg n})^2)$ worst-case time. A point can be inserted into or deleted from P in $O((\frac{\lg n}{\lg \lg n})^2)$ amortized time. This improves the result of Nekrich [12]. The point coordinates are real numbers, and we only require that any two coordinates can be compared in constant time.
2. A linear-space data structure that maintains a dynamic set, P, of n points on a $U \times U$ grid to provide $O(\frac{\lg n \lg U}{(\lg \lg n)^2})$ worst-case time support for orthogonal range counting, insertions and deletions. Note that for large data sets in which $\lg n = \Theta(\lg U)$, the query and update times are both $O((\frac{\lg n}{\lg \lg n})^2)$ in the worst case.
3. A succinct representation of a dynamic sequence $S[1..n]$ of integers from $[1..\sigma]$ in $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits[3] to support range counting, insertion and deletion in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. When $\sigma = O(\texttt{polylog}(n))$, all the operations can be supported in $O(\frac{\lg n}{\lg \lg n})$ time. This is the first dynamic succinct data structure that supports range counting.

2   Data Structures for Range Sum

In this section, we present two data structures that are used in our solutions to dynamic orthogonal range counting. Both data structures represent a two-dimensional array $A[1..r, 1..c]$ of numbers to support **range sum** queries that return the sum of the elements in a rectangular subarray of A. More precisely, a range sum query, $\texttt{range\_sum}(A, i_1, j_1, i_2, j_2)$, evaluates $\sum_{u=i_1}^{i_2} \sum_{v=j_1}^{j_2} A[u, v]$, i.e. the sum of the numbers stored in the subarray $A[i_1..i_2, j_1..j_2]$, where $1 \leq i_1 \leq i_2 \leq r$ and $1 \leq j_1 \leq j_2 \leq c$.

We define a special form of range sum queries as **dominance sum** queries. These return the sum of the elements stored in subarrays of the form $A[1..i, 1..j]$,

---

[2] $[i..j]$ denotes a range of integers, while $[i, j]$, $[i, j)$, etc. denote ranges of real values.
[3] H_0 denotes the zeroth empirical entropy of S, which is $\lg \sigma$ in the worst case.

where $1 \leq i \leq r$ and $1 \leq j \leq c$. In other words, we define the operator `dominance_sum`(A, i, j) to be `range_sum`$(A, 1, i, 1, j)$. It is clear that any range sum query can be answered using at most four dominance sum queries.

Our data structures also support update operations to the array A, namely the following three operations:

- `modify`(A, i, j, δ), which sets $A[i, j]$ to $A[i, j] + \delta$ (restrictions on δ will apply);
- `insert`(A, j), which inserts a 0 between $A[i, j-1]$ and $A[i, j]$ for all $1 \leq i \leq r$, thus increasing the number of columns of A by one;
- `delete`(A, j), which deletes $A[i, j]$ for all $1 \leq i \leq r$, decreasing the number of columns of A by one, and to perform this operation, $A[i, j] = 0$ must hold for all $1 \leq i \leq r$.

The two data structures presented in this section solve the dynamic range sum problem under different restrictions on the input data and updates, in order to achieve desired time and space bounds of our range counting structures.

2.1   Range Sum in a Small Two-Dimensional Array

We now design a data structure for a small two-dimensional array $A[1..r, 1..c]$ to support range sum. Let n be a positive integer such that $w = \Omega(\lg n)$, where w is the word size of the RAM model. We require that $rc = O(\lg^{\lambda} n)$ for any constant $\lambda \in (0, 1)$, and that each entry of A stores an nonnegative, b-bit integer where $b = O(\lg n)$. This data structure supports `modify`(A, i, j, δ), where $|\delta| \leq \lg n$, but it does not support `insert` or `delete`.

Our data structure is a generalization of data structure of Raman *et al.* [14] on supporting prefix-sum queries on a small one-dimensional array: the dominance sum query is a two-dimensional version of prefix sum. It is not hard to adapt the approach of Raman *et al.* to 2D and represent A in $O(\lg^{1+\lambda} n)$ bits to support `range_sum` in $O(1)$ worst-case time and `modify` in $O(1)$ amortized time with the help of a universal table, but deamortization is interesting and nontrivial. We first present the following data structure that provides $O(1)$ amortized time support for queries and updates:

Lemma 1. *The array A described above can be represented using $O(\lg^{1+\lambda} n)$ bits to support* `range_sum` *in $O(1)$ worst-case time and* `modify`(A, i, j, δ), *where $|\delta| \leq \lg n$, in $O(1)$ amortized time. This data structure requires a precomputed universal table of size $O(n^{\lambda'})$ bits for any fixed constant $\lambda' \in (0, 1)$.*

Proof. In addition to storing A, we construct and maintain a two-dimensional array $B[1..r, 1..c]$, in which $B[i, j]$ stores $\sum_{u=1}^{i} \sum_{v=1}^{j} A[u, v]$, i.e. the result of `dominance_sum`(A, i, j). We however cannot always keep B up-to-date under updates, so we allow B to get slightly "out of date". More precisely, B is not changed each time `modify` is performed; instead, after every rc `modify` operations, we reconstruct B from A to make B up-to-date. Since it takes $O(rc)$ time to reconstruct B, the amortized cost is $O(1)$ per `modify` operation.

As mentioned before, to support the `range_sum` operation, it suffices to provide support for `dominance_sum`. In order to answer dominance sum queries correctly

using B, we maintain another two-dimensional array $C[1..r, 1..c]$, whose content is set to all zeros each time we construct B. Otherwise, after an operation $\texttt{modify}(A, i, j, \delta)$ is performed, we set $C[i, j] \leftarrow C[i, j] + \delta$. Thus we have:

$$\texttt{dominance\_sum}(A, i, j) = B[i, j] + \sum_{u=1}^{i} \sum_{v=1}^{j} C[u, v] \qquad (1)$$

To use the above identity to compute $\texttt{dominance\_sum}(A, i, j)$ in constant time, it suffices to compute $\sum_{u=1}^{i} \sum_{v=1}^{j} C[u, v]$ in constant time. Since we set $C[i, j]$ to 0 after every rc $\texttt{modify}$ operations and we require $|\delta| \leq \lg n$, we have $|C[i, j]| \leq rc \lg n = O(\lg^{1+\lambda} n)$. Hence each entry of C can be encoded in $O(\lg \lg n)$ bits. Thus array C can be encoded in $O(\lg^{\lambda} n \lg \lg n) = o(\lg n)$ bits, which allows us to build a $O(n^{\lambda'})$-bit precomputed table to perform the above computation in constant time. It is clear that $\texttt{modify}$ can be supported in $O(1)$ amortized time, and the arrays A, B and C can be encoded in $O(\lg^{1+\lambda} n)$ bits in total.   □

To eliminate amortization, we design the following approach:

1. We construct a new table C' after rc $\texttt{modify}$ operations have been performed since the table C was created, i.e. after the values of C have been changed rc times. Initially, all entries of C' are zeroes.
2. After we create C', for the next rc $\texttt{modify}$ operations, if the operation is $\texttt{modify}(A, i, j, \delta)$, we set $C'[i, j] \leftarrow C'[i, j] + \delta$ without changing the content of C. We use the following identity instead of Identity 1 to answer queries:

$$\texttt{dominance\_sum}(A, i, j) = B[i, j] + \sum_{u=1}^{i} \sum_{v=1}^{j} C[u, v] + \sum_{u=1}^{i} \sum_{v=1}^{j} C'[u, v] \qquad (2)$$

3. We also maintain a pointer called **refresh pointer** that moves from $B[1, 1]$ to $B[r, c]$ in row-major order. When we create the table C', the refresh pointer points to $B[1, 1]$. After each $\texttt{modify}$, we move the pointer by one position. Right before we move the pointer that points to $B[i, j]$, we perform the following process of **refreshing** $B[i, j]$:
   (a) Set $B[i, j] \leftarrow B[i, j] + C[i, j]$;
   (b) If $i < r$, set $C[i + 1, j] \leftarrow C[i + 1, j] + C[i, j]$;
   (c) If $j < c$, set $C[i, j + 1] \leftarrow C[i, j + 1] + C[i, j]$;
   (d) If $i < r$ and $j < c$, set $C[i + 1, j + 1] \leftarrow C[i + 1, j + 1] - C[i, j]$;
   (e) Set $C[i, j] \leftarrow 0$.
4. After we refresh $B[r, c]$, rc $\texttt{modify}$ operations have been performed since we created C'. At this time, all the entries of C are zeroes. We then deallocate C, rename C' by C. Note that at this time, rc $\texttt{modify}$ operations have already been performed on the new array C (when it was named C'), so it is time to go back to step 1, create a new table C', and repeat this process.

In the above approach, $\texttt{modify}$ clearly takes $O(1)$ worst-case time, and A, B, C and C' can be encoded in $O(\lg^{1+\lambda} n)$ bits. To show the correctness of the above

process, it is not hard to see that Identity 2 always holds. Finally, we need argue that the right-hand side of Identity 2 can be evaluated in constant time. The term $\sum_{u=1}^{i} \sum_{v=1}^{j} C'[u,v]$ can be evaluated in constant time using the precomputed universal table. However, it is not clear whether $\sum_{u=1}^{i} \sum_{v=1}^{j} C[u,v]$ can still be evaluated in constant time using this table: Because of the refresh process, it is not trivial to show that each entry of C can still be encoded in $O(\lg \lg n)$ bits. For this we first present these two easy-to-prove lemmas (we omit their proofs):

Lemma 2. *For any integers $i \in [1,m]$ and $j \in [1,d]$, a refresh process does not change the value of $\sum_{u=1}^{i} \sum_{v=1}^{j} C[u,v]$ unless this process refreshes $B[i,j]$.*

Lemma 3. *Let $C^*[u,v]$ be the value of $C[u,v]$ when the table C' is created. Then immediately before we refresh $B[i,j]$, the value of $C[i,j]$ is $\sum_{u=1}^{i} \sum_{v=1}^{j} C^*[u,v]$.*

We can now show that each entry of C can be encoded in $O(\lg \lg n)$ bits:

Lemma 4. *The absolute value of any entry, $C[i,j]$, of C never exceeds $4rc \lg n$.*

Proof. We prove this lemma for $i > 1$ and $j > 1$; the other cases can be handled similarly. When we create the table C' and start to refresh the entries of B, rc `modify` operations have been performed since C was created (recall that initially C was named C'). Hence when we start to refresh the entries of B, the absolute value of $C[i,j]$ is at most $rc \lg n$. When we refresh $B[i,j]$, we set $C[i,j]$ to 0 and never changes its value till we deallocate C. Before $B[i,j]$ is refreshed, the value of $C[i,j]$ changes at most three times: (i) when we refresh $B[i-1,j-1]$; (ii) when we refresh $B[i-1,j]$; and (iii) when we refresh $B[i,j-1]$. In (i), we set $C[i,j] \leftarrow C[i,j] - C[i-1,j-1]$ before we set $C[i-1,j-1]$ to 0. By Lemma 3, the absolute value of $C[i-1,j-1]$ before we set it to 0 is at most $rc \lg n$. Hence the absolute value of $C[i,j]$ does not exceed $2rc \lg n$ after (i). By similar reasoning, we can show that the absolute values of $C[i,j]$ do not exceed $3rc \lg n$ and $4rc \lg n$ after (ii) and (iii), respectively.[4]   □

Our result in this section immediately follows from Lemma 4:

Lemma 5. *Let n be a positive integer such that $w = \Omega(\lg n)$, where w is the word size of the RAM model. A two-dimensional array $A[1..r, 1..c]$ of non-negative, b-bit integers, where $b = O(\lg n)$ and $rc = \lg^{\lambda} n$ for any constant $\lambda \in (0,1)$, can be represented using $O(\lg^{1+\lambda} n)$ bits to support `range_sum` and `modify`(A,i,j,δ), where $|\delta| \leq \lg n$, in $O(1)$ worst-case time. This data structure can be constructed in $O(rc)$ time, and it requires a precomputed universal table of size $O(n^{\lambda'})$ bits for any fixed constant $\lambda' \in (0,1)$.*

---

[4] With greater care, we can show that the absolute value of any element of C never exceeds $rc \lg n$ using the identity in Lemma 3, although this would not affect the time/space bounds in Lemma 5.

2.2   Range Sum in a Narrow Two-Dimensional Array

Our second data structure for dynamic range sum requires the array $A[1..r, 1..c]$ to be "narrow", i.e. $r = O(\lg^\gamma c)$ for a fixed constant $\gamma \in (0, 1)$. Dominance sum queries on this data structure can be viewed as a 2-dimensional versions of prefix sum queries in the Collections of Searchable Partial Sums (CSPSI) problem defined by González and Navarro [7]. Our data structure in based on the solution to the CSPSI problem given by He and Munro [8,9], and the main change is to use Lemma 5 to encode information encoded as small 2D arrays. We have the following result:

Lemma 6. *Let $A[1..r][1..c]$ be a two-dimensional array of nonnegative integers, where $r = O(\lg^\gamma c)$ for any constant $\gamma \in (0, 1)$, and each integer of A is encoded in $b = O(w)$ bits, where $w = \Omega(\lg c)$ is the word size of the RAM model. A can be represented using $O(rcb+w)$ bits to support* `range_sum`, `search`, `modify`(C, i, j, δ) *where $|\delta| \leq \lg c$,* `insert` *and* `delete` *in $O(\frac{\lg c}{\lg \lg c})$ time with a $O(c \lg c)$ bit buffer.*

3   Range Counting in Integer Sequences

A basic building block for many succinct data structures [6,5,7,3,8] is a highly space-efficient representation of a sequence $S[1..n]$ of integers from $[1..\sigma]$ to support the fast evaluation of `rank` and `select`[5]. Under dynamic settings, the following operations are considered:

- `access`(S, i), which returns $S[i]$;
- `rank`$_\alpha(S, i)$, which returns the number of occurrences of integer α in $S[1..i]$;
- `select`$_\alpha(S, i)$, which returns the position of the ith occurrence of integer α in the string S;
- `insert`$_\alpha(S, i)$, which inserts integer α between $S[i-1]$ and $S[i]$;
- `delete`(S, i), which deletes $S[i]$ from S.

He and Munro [8] designed a succinct representation of S to support the above operations in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. In this section, we extend their results to support range counting on integer sequences. We are interested in the operation `range_count`(S, p_1, p_2, v_1, v_2), which returns the number of entries in $S[p_1..p_2]$ whose values are in the range $[v_1..v_2]$.

3.1   Sequences of Small Integers

We first consider the case in which $\sigma = O(\lg^\rho n)$ for any constant $\rho \in (0, 1)$. In our approach, we encode S using a B-tree as in [8]. Each leaf of this B-tree contains a **_superblock_** that has at most $2L$ bits, where $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. Entries of S are stored in superblocks. A two-dimensional array $F[1..\sigma, 1..t]$ is constructed,

---

[5] Many papers define S as a string of characters over alphabet $[1..\sigma]$, which is equivalent to our definition. We choose to define S as a sequence of integers as it seems more natural to introduce range counting on integers.

where t denotes the number of superblocks. An entry $F[\alpha, i]$ stores the number of occurrences of integer α in superblock i. The array F is encoded using Lemma 6. We defer the details of our algorithms and data structures to the full version of our paper, and only present our results here. We first present our result on representing dynamic sequences of small integers to support range counting:

Lemma 7. *Under the word RAM model with word size $w = \Omega(\lg n)$, a sequence $S[1..n]$ of integers from universe $[1..\sigma]$, where $\sigma = O(\lg^\rho n)$ for any constant $\rho \in (0, 1)$, can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* access, rank, select, range_count, insert *and* delete *in $O(\frac{\lg n}{\lg \lg n})$ time.*

We also have the following lemma to show that a batch of update operations performed on a contiguous subsequence S can be supported efficiently:

Lemma 8. *Let S be a sequence represented by Lemma 7. Consider a batch of m update operations performed on subsequence $S[a..a + m - 1]$, in which the ith operation changes the value of $S[a+i-1]$. If $m > 5L/\lg \sigma$, then the above batch of operations can be performed in $O(m)$ time.*

3.2   General Integer Sequences

To generalized our result on sequences of small integers to general integer sequences, we combine the techniques of Lemma 7 with generalized wavelet trees proposed by Ferragina *et al.* [6]. Similar ideas were used by He and Munro [8] to support rank and select operations on dynamic sequences, and by Bose *et al.* [3] for static orthogonal range search structures on a grid. Here we apply these techniques on range counting in dynamic settings:

Theorem 1. *Under the word RAM model with word size $w = \Omega(\lg n)$, a sequence $S[1..n]$ of integers from $[1..\sigma]$ can be represented using $nH_0 + O(\frac{n \lg \sigma \lg \lg n}{\sqrt{\lg n}}) + O(w)$ bits to support* access, rank, select, range_count, insert *and* delete *operations in $O(\frac{\lg n}{\lg \lg n}(\frac{\lg \sigma}{\lg \lg n} + 1))$ time. When $\sigma = O(\text{polylog}(n))$, all these operations can be supported in $O(\frac{\lg n}{\lg \lg n})$ time.*

4   Range Counting in Planar Point Sets

We now consider orthogonal range counting over a dynamic set of n points on the plane. In Section 4.1, we consider a special case in which each point is on a fixed $U \times U$ grid, i.e. each x or y-coordinate is an integer from universe $[1..U]$, while in Section 4.2, points on the plane have arbitrary (real) coordinates as long as any two coordinates can be compared in constant time.

4.1   Range Counting on a $U \times U$ Grid

Our orthogonal range counting structure for a dynamic set of n points on a $U \times U$ grid is based on our data structure supporting range counting over an integer sequence. The key idea is to convert coordinates in one dimension, say, the x-coordinates, to rank space.

Theorem 2. *Under the word RAM model with word size $w = \Omega(\lg n)$, there is an $O(n)$ word data structure that can maintain a dynamic set, P, of n points on a $U \times U$ grid to answer orthogonal range counting queries in $O(\frac{\lg n \lg U}{(\lg \lg n)^2})$ time. A point can be inserted to or deleted from P in $O(\frac{\lg n \lg U}{(\lg \lg n)^2})$ time.*

Proof. Let P_x be the set of x-coordinates. Without loss of generality, we assume that x-coordinates are distinct. We construct an augmented red-black tree T_x to represent P_x: For each node v in T_x, we store additional information that encodes the number of nodes in the subtree rooted at v. With such information, given a value x, we can easily find out, in $O(\lg n)$ time, the number of elements in P_x that are less than or equal to x. Maintaining such information does not slow down the $O(\lg n)$-time support for insertions and deletions of values in P_x. This is because each insertion or deletion requires at most 3 tree rotations, so we need only update the information of subtree size for the constant number of nodes directly involved in the rotations and their $O(\lg n)$ ancestors.

We construct a sequence $S[1..n]$ of integers from $[1..U]$, in which $S[i] = u$ if and only if the point in P with the ith smallest x-coordinate has y-coordinate u. We represent S using Theorem 1. Since T_x maps x-coordinates to rank space, it is easy to use T_x and S to support query and update. □

4.2   Range Counting for General Point Sets

For general point sets, we can still use the augmented red-black tree designed in the proof of Theorem 2 to map the set of x-coordinates to the rank space, since this tree structure does not require any assumptions on the values stored. Handling the other dimension, however, is challenging: We cannot simply use a generalized wavelet tree, which is the main building block of the representation of the sequence S used in the proof of Theorem 2. This is because a (generalized) wavelet tree has, up to now, only been used to handle data of a two-dimensional nature for which the range of values in at least one dimension is fixed [6,5,3,7,8], such as sequences of integers from a fixed range in Theorem 1. To overcome this difficulty, our main strategy is to combine the notion of range trees [2] with generalized wavelet trees. Our work is the first that combines these two powerful data structures.

Let P_x and P_y denote the set of x and y-coordinates of the points in P, respectively. Without loss of generality, we assume that the values in P_x are distinct, and so are the values in P_y. We construct the following data structures:

1. An augmented red-black tree, T_x, that represents the set P_x, as described in the proof of Theorem 1. Recall that this structure supports the computation of the number of values in P_x that are less than or equal to a given value.
2. As amortizing a rebuilding cost to insertions or deletions will be crucial, we use a weight-balanced B-tree [1], T_y. This is constructed over P_y, with branching factor $d = \Theta(\lg^\epsilon n)$ for a fixed constant $\epsilon \in (0, 1)$ and leaf parameter 1. Hence each internal node has at least $d/4$ and at most $4d$ children, except the root for which the lower bound on degree does not apply. Each leaf represents a range $[y, y')$, where y and y' are in P_y, and y' is the immediate successor of y. The

(contiguous) range represented by an internal node is the union of the ranges represented by its children. The levels of T_y are numbered $0, 1, 2, \cdots$, starting from the root level. We store the tree structure of T_y together with the start and end values of the range represented by each node.

3. Next we use ideas analogous to those of generalized wavelet trees [6]. A sequence $L_\ell[1..n]$ of integers from $[1..4d]$ is constructed for each level ℓ except the leaf level, which is encoded using Lemma 7: For each internal node v at level ℓ of T_y, we construct a sequence, S_v of integers from $[1..4d]$. Each entry of S_v corresponds to a point in P whose y-coordinate is in the range represented by v, and $S_v[i]$ corresponds to, among all the points with y-coordinates within the range represented by v, the one with the ith smallest x-coordinate. $S_v[i]$ does not store this point directly. Instead, $S_v[i]$ stores j if the y-coordinate of the corresponding point in P is within the range represented by the jth child of v. We further concatenate all the sequences constructed for the nodes, from left to right, at level ℓ to get the sequence L_ℓ. It is important to understand that, for the top level of T_y, the entries of L_0 correspond to points in P ordered by x-coordinates, but as we move down the tree T_y, the ordering gradually changes: The entries of $L_1, L_2, \cdots$ do not correspond to points ordered by x-coordinates, and at the bottom level, the leaves correspond to points ordered by y-coordinates.

To analyze the space cost of our data structures, it is clear that T_x and T_y use linear space. Our third set of data structures consist of $O(\frac{\lg n}{\lg \lg n})$ subsequences, each storing n integers from $[1..4d]$. By Lemma 7, they occupy $O(n \lg d + w) \times O(\frac{\lg n}{\lg \lg n}) = O(n \lg n + w \times \frac{\lg n}{\lg \lg n})$ bits in total, where w is the size of a word. This space cost is $O(n)$ words. We now use these data structures to support queries:

Lemma 9. *Under the word RAM model with word size $w = \Omega(\lg n)$, the above data structures support orthogonal range counting in $O((\frac{\lg n}{\lg \lg n})^2)$ time.*

Proof. We first give an overview of our algorithm for orthogonal range counting. Let $R = [x_1, x_2] \times [y_1, y_2]$ be the query rectangle. We use T_x to find two x-coordinates x_1' and x_2' in P_x that are the immediate successor of x_1 and the immediate predecessor of x_2, respectively (if a value is present in P_x, we define its immediate predecessor/successor to be itself). We then perform a top-down traversal in T_y to locate the (up to two) leaves that represent ranges containing y_1 and y_2. During this traversal, at each level ℓ of T_y, at most two nodes are visited. For a node v visited at level ℓ, we answer a range counting query `range_count`$(S_v, i_v, j_v, c_v, d_v)$, where $S_v[i_v..j_v]$ is the longest contiguous subsequence of S_v whose corresponding points in P have x-coordinates in the range $[x_1', x_2']$, and the children of v representing ranges that are entirely within $[y_1..y_2]$ are children $c_v, c_v + 1, \cdots, d_v$ (child i refers to the ith child). The sum of the results of the above range queries at all levels is the number of points in $N \cap R$.

To show how to perform the above process, we first observe that for the root r of T_y, i_r and j_r are the numbers of values in P_x that are less than or equal to x_1' and x_2', respectively, which can be computed using T_x in $O(\lg n)$ time.

To compute c_r and d_r, we can perform binary searches on the up to $4d$ ranges represented by the children of r, which takes $O(\lg \lg n)$ time. The binary searches also tell us which child/children of r represent ranges that contain y_1 and y_2, and we continue the top-down traversal by descending into these nodes.

It now suffices to show, for each node v visited at each level ℓ, how to locate the start and end positions of S_v in L_ℓ, how to compute i_v, j_v, c_v and d_v, and which child/children of v we shall visit at level $\ell + 1$. Let u be the parent of v, and we assume that v is the cth child of u. Let s be the start position of S_u in $L_{\ell-1}$, which was computed when we visited u. We observe that, to compute the start and end positions of S_v, it suffices to compute the numbers of entries of S_u that are in the range $[1..c-1]$ and $[1..c]$, respectively. Thus the start and end positions of S_v in L_ℓ are $s + \mathtt{range\_count}(S_u, 1, |S_u|, 1, c-1) + 1$ and $s + \mathtt{range\_count}(S_u, 1, |S_u|, 1, c)$, respectively. Positions i_v and j_v can also be computed by performing operations on S_u using the identities $i_v = \mathtt{rank}_c(S_u, i_u - 1) + 1$ and $j_v = \mathtt{rank}_c(S_u, j_u)$. Finally, to compute c_v and d_v, and to determine the child/children of v that we visit at level $\ell + 1$, we perform binary searches on the ranges represented by the at most $4d$ children of v using $O(\lg d) = O(\lg \lg n)$ time. Since we perform a constant number of $\mathtt{rank}$, $\mathtt{select}$ and $\mathtt{range\_count}$ operations on a sequence of small integers at each level of T_y, and there are $O(\frac{\lg n}{\lg \lg n})$ levels, we can answer an orthogonal range counting query over P in $O((\frac{\lg n}{\lg \lg n})^2)$ time. $\square$

Finally, we support update operations to achieve our main result:

Theorem 3. *Under the word RAM model with word size $w = \Omega(\lg n)$, there is a data structure using $O(n)$ words of structural information plus space for the coordinates that can maintain a dynamic set, P, of n points on the plane to answer orthogonal range counting queries in $O((\frac{\lg n}{\lg \lg n})^2)$ worst-case time. A point can be inserted to or deleted from P in $O((\frac{\lg n}{\lg \lg n})^2)$ amortized time.*

Proof. To support update operations, we only show how to insert a point into P; deletions can be handled similarly. To insert a point with coordinates $< x, y >$ into P, we first insert x into T_x in $O(\lg n)$ time. Inserting y into T_y will either cause a leaf of T_y to split into two, or create a new leaf that is either the leftmost leaf or the rightmost. Without loss of generality, we assume that a leaf is split. Then this leaf can be located by performing a top-down traversal, similar to the process required to support range counting. Let q be the parent of this leaf, and let ℓ' be the level number of the level right above the leaf level. Then the start and end positions of S_q in $L_{\ell'}$ can also be located in the above process, using $O((\frac{\lg n}{\lg \lg n})^2)$ time in total. We first consider the case that the split of this leaf will not cause q to split. In this case, since q has one more child, we insert one more entry into S_q for the new child, and increase the values of at most $|S_q|$ entries in S_q by 1. This can be done by performing at most $|S_q|$ insertions and deletions over S_q, which costs $O(d \times \frac{\lg n}{\lg \lg n}) = O(\frac{\lg^{1+\epsilon} n}{\lg \lg n})$ time. Since this insertion also causes the length of the sequence S_v for the ancestor, v, of q at each level to increase by 1, one integer is inserted into each string L_ℓ for $\ell = 0, 1, \cdots, \ell' - 1$. The exact position where we should perform the insertion can be determined using

tree T_x for L_0, and by performing one `rank` operation on $L_0, L_1, \cdots, L_{\ell'-2}$ for $L_1, L_2, \cdots, L_{\ell'-1}$, respectively, during the top-down traversal. The exact value to be inserted to each L_ℓ is an appropriate child number. So far we have spent $O((\frac{\lg n}{\lg \lg n})^2)$ time in total.

Each insertion may however cause a number of internal nodes to split. Let v be an internal node that is to split, and let v_1 and v_2 be the two nodes that v is to be split into, where v_1 is a left sibling of v_2. Let f be the number of the level that contains v. Then the splitting of v requires us to replace the substring, S_v, of L_f by two substrings S_{v_1} and S_{v_2}. Since points corresponding to these substrings are sorted by their x-coordinates, this is essentially a process that splits one sorted list into two sorted lists. Thus, we can perform a linear scan on S_v, and perform one insertion and one deletion for each entry of S_v. This costs $O(|S_v| \times \frac{\lg n}{\lg \lg n})$ time. This would have been messy, but fortunately the following two invariants of weight-balanced B-trees allow us to bound the above time in the amortized sense: First, if v is k levels above the leaf level, then $|S_v| < 2d^k$. Second, after the split of node v, at least $d^k/2$ insertions have to be performed below v before it splits again. Hence we can amortize the above $O(|S_v| \times \frac{\lg n}{\lg \lg n}) = O(2d^k \times \frac{\lg n}{\lg \lg n})$ cost over $d^k/2$ insertions, which is $O(\frac{\lg n}{\lg \lg n})$ per insertion.

Let u be the parent of v. The above split may also increase the values of up to $|S_u|$ entires of S_u by 1, which cost $O(|S_u| \times \frac{\lg n}{\lg \lg n})$ time. By the same argument as above, we have $|S_u| < 2d^{k+1}$, and we can amortize the above $O(|S_u| \times \frac{\lg n}{\lg \lg n}) = O(2d^{k+1} \times \frac{\lg n}{\lg \lg n})$ cost over $d^k/2$ insertions, which is $O(\frac{d \lg n}{\lg \lg n})$ per insertion. Since the insertion into a leaf may cause its $O(\frac{\lg n}{\lg \lg n})$ ancestors to split, and each split charges $O(\frac{d \lg n}{\lg \lg n})$ amortized cost for this insertion, splitting these $O(\frac{\lg n}{\lg \lg n})$ internal nodes after an insertion requires $O(\frac{\lg^{2+\epsilon} n}{(\lg \lg n)^2}) = O(\lg^{2+\epsilon} n)$ amortized time.

To further speed up the process in the previous paragraph, we observe that the bottleneck is the $O(|S_u| \times \frac{\lg n}{\lg \lg n})$ time required to change the values of up to $|S_u|$ entires of S_u after v is split. Since these entries are all in S_u, which is a contiguous subsequence of L_{f-1}, we apply Lemma 8. There is one more technical detail: this lemma requires that $|S_u| > 5L/\lg(4d)$ where $L = \lceil \frac{\lceil \lg n \rceil^2}{\lg \lceil \lg n \rceil} \rceil$. By an invariant maintained by a weight-balanced B-tree, $|S_u| > d^{k+1}/2$, since u is $k+1$ levels above the leaf level. Hence $|S_u| > 5L/\lg(4d)$ is true for all $k > \log_{d/2}(5L/\lg(4d))$, and the floor of the right-hand side is a constant number. Let k_0 denote this constant. Hence if v is up to k_0 levels above the leaf level, we use the approach in the previous paragraph to update S_u. Since each insertions can only cause a constant number of internal nodes that are up to k_0 levels above the leaf level to split, this incurs $O(\frac{k_0 d \lg n}{\lg \lg n}) = O(\lg^{1+\epsilon} n)$ amortized cost per insertion. If v is more than k_0 levels above the leaf level, then we use Lemma 8 to update L_{f-1} in $O(|S_u|)$ time. By the analysis in the previous paragraph, the cost of splitting v can be amortized over $d^k/2$ insertions, which is $O(d)$ per insertion. Since each insertions can potentially cause $O(\frac{\lg n}{\lg \lg n})$ nodes that are more than k_0 levels above the leaf level to split, this also incurs $O(\frac{d \lg n}{\lg \lg n}) = O(\lg^{1+\epsilon} n)$ amortized

cost per insertion. Therefore, each insertion can be supported in $O((\frac{\lg n}{\lg \lg n})^2) + O(\lg^{1+\epsilon} n) = O((\frac{\lg n}{\lg \lg n})^2)$ amortized time.

We finish our proof by pointing out that the succinct global rebuilding approach of He and Munro [8] can be applied here to handle the change of the value of $\lceil \lg n \rceil$, which affects the choices of the value for d. $\qquad \square$

5   Concluding Remarks

We have presented three new dynamic range counting structures, and to obtain these results, we designed two data structures for range sum queries, which are of independent interest. We have also developed new techniques. Our approach of deamortization on a two-dimensional array in Section 2.1 is interesting. Our attempt on combining wavelet trees and range trees in Section 4.2 is the first that combines these two very powerful data structures, and we expect to use the same strategy to solve other problems.

References

1. Arge, L., Vitter, J.S.: Optimal external memory interval management. SIAM J. Comput. 32(6), 1488–1508 (2003)
2. Bentley, J.L.: Multidimensional divide-and-conquer. Commun. ACM 23(4), 214–229 (1980)
3. Bose, P., He, M., Maheshwari, A., Morin, P.: Succinct orthogonal range search structures on a grid with applications to text indexing. In: Dehne, F., Gavrilova, M., Sack, J.-R., Tóth, C.D. (eds.) WADS 2009. LNCS, vol. 5664, pp. 98–109. Springer, Heidelberg (2009)
4. Chazelle, B.: A functional approach to data structures and its use in multidimensional searching. SIAM Journal on Computing 17(3), 427–462 (1988)
5. Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. Journal of the ACM 57(1) (2009)
6. Ferragina, P., Manzini, G., Mäkinen, V., Navarro, G.: Compressed representations of sequences and full-text indexes. ACM Transactions on Algorithms 3(2) (2007)
7. González, R., Navarro, G.: Rank/select on dynamic compressed sequences and applications. Theoretical Computer Science 410(43), 4414–4422 (2009)
8. He, M., Munro, J.I.: Succinct representations of dynamic strings. In: SPIRE, pp. 334–346 (2010)
9. He, M., Munro, J.I.: Succinct representations of dynamic strings. CoRR abs/1005.4652 (2010)
10. Jacobson, G.: Space-efficient static trees and graphs. In: FOCS, pp. 549–554 (1989)
11. JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 558–568. Springer, Heidelberg (2004)
12. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Computational Geometry: Theory and Applications 42(4), 342–351 (2009)
13. Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: STOC, pp. 40–46 (2007)
14. Raman, R., Raman, V., Rao, S.S.: Succinct dynamic data structures. In: Dehne, F., Sack, J.-R., Tamassia, R. (eds.) WADS 2001. LNCS, vol. 2125, pp. 426–437. Springer, Heidelberg (2001)

Searching in Dynamic Tree-Like Partial Orders

Brent Heeringa[1,⋆], Marius Cătălin Iordan[2,⋆⋆], and Louis Theran[3,⋆⋆⋆]

[1] Dept. of Computer Science, Williams College
heeringa@cs.williams.edu
[2] Dept. of Computer Science, Stanford University
mci@cs.stanford.edu
[3] Dept. of Mathematics, Temple University
theran@temple.edu

Abstract. We give the first data structure for the problem of maintaining a dynamic set of n elements drawn from a partially ordered universe described by a tree. We define the LINE-LEAF TREE, a linear-sized data structure that supports the operations: *insert; delete; test membership; and predecessor*. The performance of our data structure is within an $O(\log w)$-factor of optimal. Here $w \leq n$ is the width of the partial-order—a natural obstacle in searching a partial order.

1 Introduction

A fundamental problem in data structures is maintaining an ordered set S of n items drawn from a universe $\mathcal{U}$ of size $M \gg n$. For a totally ordered $\mathcal{U}$, the dictionary operations: *insert*; *delete*; *test membership*; and *predecessor* are all supported in $O(\log n)$ time and $O(n)$ space in the comparison model via balanced binary search trees. Here we consider the relaxed problem where $\mathcal{U}$ is partially ordered and give the first data structure for maintaining a subset of a universe equipped with a partial order that can be described by a tree.

As a motivating example, consider an email user that has stockpiled years of messages into a series of hierarchical folders. When searching for an old message, filing away a new message, or removing an impertinent message, the user must navigate the hierarchy. Suppose the goal is to minimize, in the worst-case, the number of folders the user must consider in order to find the correct location in which to retrieve, save, or delete the message. Unless the directory structure is completely balanced, an optimal search does not necessarily start at the top—it might be better to start farther down the hierarchy if the majority of messages lie in a sub-folder. If we model the hierarchy as a rooted, oriented tree and treat the question "is message x contained somewhere in folder y?" as our comparison, then maintaing an optimal search strategy for the hierarchy is equivalent to maintaining a *dynamic* partially ordered set under insertions and deletions.
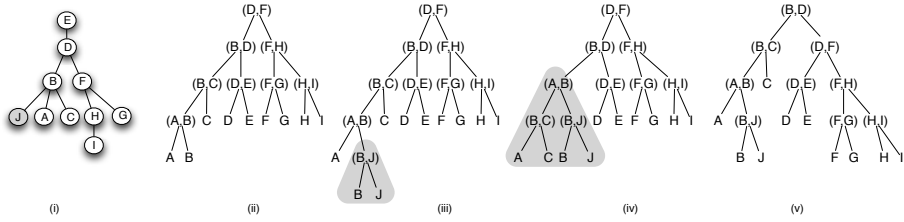
Fig. 1. (i) A partially ordered set $\{A, B, C, D, E, F, G, H, I, J\}$. A downward path from node X to node Y implies $X \prec Y$. Note that, for example, $E \prec F$ and G and I are incomparable. (ii) An optimal search tree for the set $\{A, B, \ldots, I\}$. For any query (X, Y) an answer of X means descend left and an answer of Y means descend right. (iii) After adding the element J, a standard search tree would add a new query (B, J) below (A, B) which creates an imbalance. (iv) The search tree after a rotation; the subtree highlighted in grey is not a correct search tree for the partial order (i). (v) An optimal search tree for the set $\{A, B, \ldots, J\}$.

Related Work. The problem of searching in trees and partial orders has recently received considerable attention. Motivating this research are practical problems in filesystem synchronization, software testing and information retrieval [1]. However, all of this work is on the *static* version of the problem. In this case, the set S is fixed and a search tree for S does not support the insertion or deletion of elements. For example, when S is totally ordered, the optimal minimum-height solution is a standard binary search tree. In contrast to the totally ordered case, finding a minimum height static search tree for an arbitrary partial order is NP-hard [2]. Because of this, most recent work has focused on partial orders that can be described by rooted, oriented trees. These are called *tree-like* partial orders in the literature. For tree-like partial orders, one can find a minimum height search tree in linear time [3,4,5]. In contrast, the weighted version of the tree-like problem (where the elements have weights and the goal is to minimize the *average* height of the search tree) is NP-hard [6] although there is a constant-factor approximation [7]. Most of these results operate in the edge query model which we review in Sec. 2.

Daskalakis et al. have recently studied the problem of *sorting* partial orders [8,9] and, in [9], ask for analogues of balanced binary search trees for dynamic partially ordered sets. We are the first to address this question.

Rotations do not preserve partial orders. Traditional data structures for dynamic ordered sets (*e.g.*, red black trees, AVL trees) appear to rely on the total order of the data. All these data structures use binary tree rotations as the fundamental operations; applied in an unrestricted manner, rotations *require* a totally ordered universe. For example, consider Figure 1 (ii) which gives an optimal search tree for the elements $\{A, B, \ldots, I\}$ depicted in the partial order of Figure 1 (i). If we insert node J (colored grey) then we must add a new test (B, J) below (A, B) which creates the sub-optimal search tree depicted in Figure 1 (iii). Using traditional rotations yields the search tree given in Figure 1 (iv) which

does not respect the partial order; the leaf marked C should appear under the right child of test (A, B). Figure 1 (v) denotes a correct optimal search for the set $\{A, B, \ldots, J\}$. The key observation is that, if we imagine the leaves of a binary search tree for a total order partitioning the real line, rotations preserve the order of the leaves, but not any kind of subtree relations on them. As a consequence, blindly applying rotations to a search tree for the static problem does not yield a viable dynamic data structure. To sidestep this problem, we will, in essence, decompose the tree-like partial order into totally ordered chains and totally incomparable stars.

Our Techniques and Contributions. We define the LINE-LEAF TREE, the first data structure that supports the fundamental dictionary operations for a *dynamic* set $S \subseteq \mathcal{U}$ of n elements drawn from a universe equipped with a *partial order* $\preceq$ described by a rooted, oriented tree.

Our dynamic data structure is based on a static construction algorithm that takes as input the *Hasse diagram* induced by $\preceq$ on S and in $O(n)$ time and space produces a LINE-LEAF TREE for S. The Hasse diagram H_S for S is the directed graph that has as its vertices the elements of S and a directed edge from x to y if and only if $x \prec y$ and no z exists such that $x \prec z \prec y$. We build the LINE-LEAF TREE inductively via a natural contraction process which starts with H_S and, ignoring the edge orientations, repeatedly performs the following two steps until there is a single node:

1. Contract paths of degree-two nodes into balanced binary search trees (which we can binary search efficiently); and
2. Contract leaves into linear search structures associated with their parents (since the children of an interior node are mutually incomparable).

One of these steps always applies in our setting since H_S is a rooted, oriented tree. We give an example of each step of the construction in Figure 2. We show that the contraction process yields a search tree that is provably within an $O(\log w)$-factor of the minimum-height static search tree for S. The parameter w is the *width* of S—the size of the largest subset of mutually incomparable elements of S—which represents a natural obstacle when searching a partial order. We also show that our analysis is tight. Our construction algorithm and analysis appear in Section 3.

To make the LINE-LEAF TREE *fully dynamic*, in Section 4 we give procedures to update it under insertions and deletions. All the operations, take $O(\log w) \cdot OPT$ comparisons and RAM operations where OPT is the height of a minimum-height static search tree for S. Additionally, *insertion* requires only $O(h)$ comparisons, where h is the height of the LINE-LEAF TREE being updated. (The non-restructuring operations *test membership* and *predecessor* also require at most $O(h)$ comparisons since the LINE-LEAF TREE is a search tree). Because w is a property of S, in the dynamic setting it changes under insertions and deletions. However, the LINE-LEAF TREE maintains the $O(\log w) \cdot OPT$ height bound *at all times*. This means it is well-defined to speak of the $O(\log w) \cdot OPT$ upper bound without mentioning S.
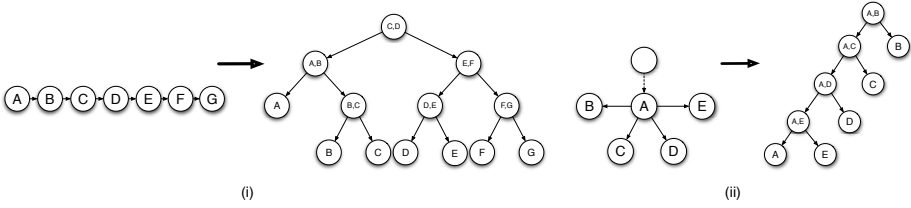
Fig. 2. Examples of (i) a line contraction where we build a balanced binary search tree from a path and (ii) a leaf contraction where we build a linear search tree from the leaves of a node.

The insertion and deletion algorithms maintain the invariant that the updated LINE-LEAF TREE is structurally equivalent to the one that we would have produced had the static construction algorithm been applied to the updated set S. In fact, the heart of insertion and deletion is *correcting* the contraction process to maintain this invariant. The key structural property of a LINE-LEAF TREE— one that is not shared by constructions for optimal search trees in the static setting—is that its sub-structures essentially represent either paths or stars in S, allowing for updates that make only local changes to each component search structure. The $O(\log w)$-factor is the price we pay for the additional flexibility. The dynamic operations, while conceptually simple, are surprisingly delicate. We devote detailed attention to them in the full version of this paper [10].

In Section 5 we provide empirical results on both random and real-world data that show the LINE-LEAF TREE is, in practice, competitive with the static optimal search tree.

2   Models and Definitions

Let $\mathcal{U}$ be a finite set of M elements and let $\preceq$ be a partial order, so the pair $(\mathcal{U}, \preceq)$ forms a *partially ordered set*. We assume the answers to $\preceq$-queries are provided by an oracle. (Daskalakis, et al. [8] provide a space-efficient data structure to answer $\preceq$-queries in $O(1)$ time.)

In keeping with previous work, we say that $\mathcal{U}$ is *tree-like* if $H_\mathcal{U}$ forms a rooted, oriented tree. Throughout the rest of this paper, we assume that $\mathcal{U}$ is tree-like and refer to the vertices of $H_\mathcal{U}$ and the elements of $\mathcal{U}$ interchangeably. For convenience, we add a dummy minimal element ν to $\mathcal{U}$. Since any search tree for a set $S \subseteq \mathcal{U}$ embeds with one extra comparison into a corresponding search tree for $S \cup \{\nu\}$, we assume from now on that ν is always present in S. This ensures that the Hasse diagram for S is always connected.

Given these assumptions it is easy to see that tree-like partial orders have the following properties:

Property 1. Any subset S of a tree-like universe $\mathcal{U}$ is also tree-like.

Property 2. Every non-root element in a tree-like partially ordered set $S \subseteq \mathcal{U}$ has exactly one predecessor in H_S.
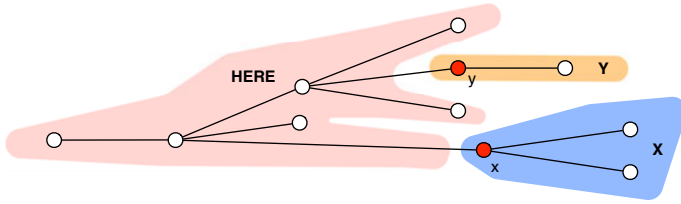
Fig. 3. Given two nodes x and y in S and a third node $u \in \mathcal{U}$, a dynamic edge query on (x, y) with respect to u can answer (i) Y, in which case u falls somewhere in the shaded area labelled Y; (ii) X, in which case u falls somewhere in the shaded area labelled X; or (iii) HERE, in which case u falls somewhere in the shaded area labelled HERE. Notice that if (x, y) forms an actual edge then the query reduces to a standard edge query.

Let T_S be the rooted, oriented tree corresponding to the Hasse diagram for S. We extend edge queries to *dynamic edge queries* by allowing queries on arbitrary pairs of nodes in T_S instead of just edges in T_S.

Definition 1 (Dynamic Edge-Queries). *Let u be an element in $\mathcal{U}$ and x and y be nodes in T_S. Let $S' = S \cup \{u\}$ and consider the edges (x, x') and (y, y') bookending the unique path from x to y in $T_{S'}$. Define $T_{S'}^x$, $T_{S'}^y$ and $T_{S'}^{\text{HERE}}$ to be the three connected components of $T_{S'} \setminus \{(x, x'), (y, y')\}$ containing x, y, and neither x nor y, respectively. A dynamic edge query on (x, y) with respect to u has one of the following three answers:*

1. X*: if $u \in T_{S'}^x$ (u equals or is closer to x)*
2. Y*: if $u \in T_{S'}^y$ (u equals or is closer to y)*
3. HERE*: if $u \in T_{S'}^{\text{HERE}}$ (u falls between, but is not equal to either, x or y)*

Figure 3 gives an example of a dynamic edge query. Any dynamic edge query can be simulated by $O(1)$ standard comparisons when H_S is tree-like. This is not the case for more general orientations of H_S and an additional data structure is required to implement either our algorithms or algorithms of [3,4]. Thus, for a tree-like S, the height of an optimal search tree in the dynamic edge query model and the height of an optimal decision tree for S in the comparison model are always within a small constant factor of each other. For the rest of the paper, we will often drop *dynamic* and refer to *dynamic edge queries* simply as *edge queries*.

3   Line-Leaf Tree Construction and Analysis

We build a LINE-LEAF TREE $\mathcal{T}$ inductively via a contraction process on T_S. Each contraction step builds a *component search structure* of the LINE-LEAF TREE. These component search structures are either linear search trees or balanced binary search trees. A linear search tree $LST(x)$ is a sequence of dynamic edge queries, all of the form (x, y) where $y \in S$, that ends with the node x. A balanced binary search tree $BST(x, y)$ for a path of contiguous degree-2 nodes between, but not including, x and y is a tree that *binary searches* the path using edge queries.

Let $T_0 = T_S$. If the contraction process takes m iterations total, then the final result is a single node which we label $\mathcal{T} = T_{2m}$. In general, let T_{2i-1} be the partial order tree after the line contraction of iteration i and T_{2i} be the partial order tree after the leaf contraction of iteration i where $i \geq 1$. We now show how to construct a Line-Leaf Tree for a fixed tree-like set S.

Base Cases. Associate an empty balanced binary search tree $BST(x, y)$ with every actual edge (x, y) in T_0. Associate a linear search tree $LST(x)$ with every node x in T_0. Initially, $LST(x)$ contains just the node itself.

Line Contraction. Consider the line contraction step of iteration $i \geq 1$: If $x_2, \ldots, x_{t-1}$ is a path of contiguous degree-2 nodes in $T_{2(i-1)}$ bounded on each side by non-degree-2 nodes x_1 and x_t respectively, we contract this path into a balanced binary search tree $BST(x_1, x_t)$ over the nodes $x_2, \ldots, x_{t-1}$. The result of the path contraction is an edge labeled (x_1, x_t). This edge yields a dynamic edge query.

Leaf Contraction. Consider the leaf contraction step of iteration $i \geq 1$: If $y_1, \ldots, y_t$ are all degree-1 nodes in T_{2i-1} adjacent to a node x in T_{2i-1}, we contract them into the linear search tree $LST(x)$ associated with x. Each node y_j contracted into x adds a dynamic edge query (x, y_j) to $LST(x)$. If nodes were already contracted into $LST(x)$ from a previous iteration, we add the new edge queries to the front (top) of the LST.

After m iterations we are left with $\mathcal{T} = T_{2m}$ which is a single node. This node is the root of the Line-Leaf Tree.

Searching a Line-Leaf Tree for an element u is tantamount to searching the component search structures. A search begins with $LST(x)$ where x is the root of $\mathcal{T}$. Searching $LST(x)$ with respect to u serially questions the edge queries in the sequence. Starting with the first edge query, if (x, y) answers X then we move onto the next query (x, z) in the sequence. If the query answers HERE then we proceed by searching for u in $BST(x, y)$. If it answers Y, then we proceed by searching for u in $LST(y)$. If there are no more edge queries left in $LST(x)$, then we return the actual element x. When searching $BST(x, y)$, if we ever receive a HERE response to the edge query (a, b), we proceed by searching for u in $BST(a, b)$. That is, we leave the current BST and search in a new BST. If the binary search concludes with a node x, then we proceed by searching $LST(x)$. Searching an empty BST returns NIL.

Implementation Details. The Line-Leaf Tree is an index into H_S but not a replacement for H_S. That is, we maintain a separate DAG data structure for H_S across insertions and deletions into S. This allows us, for example, to easily identify the predecessor and successors of a node $x \in S$ once we've used the Line-Leaf Tree to find x in H_S. The edges of H_S also play an essential role in the implementation of the Line-Leaf Tree. Namely, an edge query (x, y) is actually two pointers: $\lambda_1(x, y)$ which points to the edge (x, a) and $\lambda_2(x, y)$ which points to the edge (b, y). Here (x, a) and (b, y) are the actual edges bookending the undirected path between x and y in T_S. This allows us to take an actual edge (x, a) in memory, rename x to w, and indirectly update all edge queries (x, z)

to (w, z) in constant time. Here the path from z to x runs through a. Note that we are not touching the pointers involved in each edge query (x, z), but rather, the actual edge in memory to which the edge query is pointing.

Edge queries are created through line contractions so when we create the binary search tree $BST(x, y)$ for the path $x, a, \ldots, b, y$, we let $\lambda_1(x, y) = \lambda_1(x, a)$ and $\lambda_2(x, y) = \lambda_2(b, y)$. We assume that every edge query (x, y) corresponding to an actual edge (x', y') has $\lambda_1(x, y) = \lambda_2(x, y) = (x', y')$.

Node Properties. We associate two properties with each node in S. The *round* of a node x is the iteration i where x was contracted into either an LST or a BST. We say $\text{ROUND}(x) = i$. The *type* of a node represents the step where the node was contracted. If node x was *line contracted*, we say $\text{TYPE}(x) = \text{LINE}$, otherwise we say $\text{TYPE}(x) = \text{LEAF}$.

In addition to ROUND and TYPE, we assume that both the linear and binary search structures provide a PARENT method that operates in time proportional to the height of the respective data structure and yields either a node (in the case of a leaf contraction) or an edge query (in the case of a line contraction). More specifically, if node x is leaf contracted into $LST(a)$ then $\text{PARENT}(x) = a$. If node x is line contracted into $BST(a, b)$ then $\text{PARENT}(x) = (a, b)$. We emphasize that the PARENT operation here refers to the LINE-LEAF TREE and not T_S. Collectively, the ROUND, TYPE, and PARENT of a node help us recreate the contraction process when inserting or removing a node from S.

Approximation Ratio. The following theorem gives the main properties of the static construction.

Theorem 1. *The worst-case height of a* LINE-LEAF TREE $\mathcal{T}$ *for a tree-like* S *is* $\Theta(\log w) \cdot OPT$ *where* w *is the width of* S *and* OPT *is the height of an optimal search tree for* S. *In addition, given* H_S, $\mathcal{T}$ *can be built in* $O(n)$ *time and space.*

Proof. We prove the upper bound here and leave the tight example to the full version [10]. We begin with some lower bounds on OPT.

Claim. $OPT \geq \max\{\Delta(S), \log n, \log D, \log w\}$ where $\Delta(S)$ is the maximum degree of a node in T_S, n is the size of S, D is the diameter of T_S and w is the width of S.

Proof. Let x be a node of highest degree $\Delta(S)$ in T_S. Then, to find x in the T_S we require at least $\Delta(S)$ queries, one for each edge adjacent to x [11]. This implies $OPT \geq \Delta(S)$. Also, since querying any edge reduces the problem space left to search by at most a half, we have $OPT \geq \log n$. Because n is an upper bound on both the width w of S and D, the diameter of T_S we obtain the final two lower bounds. □

Recall that the width w of S is the number of leaves in T_S. Each round in the contraction process reduces the number of remaining leaves by at least half: round i starts with a tree T_{2i} on n_i nodes with w_i leaves. A line-contraction produces a tree T_{2i+1}, still with w_i leaves. Because T_{2i+1} is full, the number

of nodes neighboring a leaf is at most $w_i/2$. Round i completes with a leaf contraction that removes all w_i leaves, producing T_{2i+2}. As every leaf in T_{2i+2} corresponds to an internal node of T_{2i+1} adjacent to a leaf, T_{2i+2} has at most $w_i/2$ leaves. It follows that the number of rounds is at most $\log w$. The length of any root-to-leaf path is bounded in terms of the number of rounds. The following lemma follows from the construction.

Lemma 1. *On any root-to-leaf path in the* Line-Leaf Tree *there is at most one BST and one LST for each iteration i of the construction algorithm.*

For each LST we perform at most $\Delta(S)$ queries. In each BST we ask at most $O(\log D)$ questions. By the previous lemma, since we search at most one BST and one LST for each iteration i of the contraction process and since there at most $\log w$ iterations, it follows that the height of the Line-Leaf Tree is bounded above by: $(\Delta(S) + O(\log D)) \log w = O(\log w) \cdot OPT$.

We now prove the time and space bounds. Consider the line contraction step at iteration i: we traverse $T_{2(i-1)}$, labeling paths of contiguous degree-2 nodes and then traverse the tree again and form balanced BSTs over all the paths. Since constructing balanced BSTs is a linear time operation, we can perform a complete line contraction step in time proportional to the size of size of $T_{2(i-1)}$. Now consider the leaf contraction step at iteration i: We add each leaf in T_{2i-1} to the LST corresponding to its remaining neighbor. This operation is also linear in the size of T_{2i-1}. Since we know the size of T_{2i} is halved after each iteration, starting with n nodes in T_0, the total number of operations performed is $\sum_{i=0}^{\log n} O(\frac{n}{2^i}) = O(n)$.

Given that the construction takes at most $O(n)$ time, the resulting data structure occupies at most $O(n)$ space.   □

4   Operations

Test Membership. To test whether an element $A \in \mathcal{U}$ appears in $\mathcal{T}$, we search for A in $LST(x)$ where x is the root of $\mathcal{T}$. The search ends when we reach a terminal node. The only terminal nodes in the Line-Leaf Tree are either leaves representing the elements of S or Nil (which are empty BSTs). So, if we find A in $\mathcal{T}$ then test membership returns True, otherwise it returns False. Given that test membership follows a root-to-leaf path in $\mathcal{T}$, the previous discussion constitutes a proof of the following theorem.

Theorem 2. Test Membership *is correct and takes $O(h)$ time.*

Predecessor. Property 1 guarantees that each node $A \in \mathcal{U}$ has exactly one predecessor in S. Finding the predecessor of A in S is similar to test member-ship. We search $\mathcal{T}$ until we find either A or Nil. Traditionally if A appears in a set then it is its own predecessor, so, in the first case we simply return A. In the latter case, A is not in $\mathcal{T}$ and Nil corresponds to an empty binary search tree $BST(y, z)$ for the actual edge (y, z) where, say, $y \prec z$. We know that A

falls between y and z (and potentially between y and some other nodes) so y is the predecessor of A. We return y. Given that PREDECESSOR also follows a root-to-leaf path in $\mathcal{T}$, the previous discussion yields a proof of the following theorem.

Theorem 3. PREDECESSOR *is correct and takes* $O(h)$ *time.*

Insert. Let $A \notin S$ be the node we wish to insert in $\mathcal{T}$ and let $S' = S \cup \{A\}$. Our goal is to transform $\mathcal{T}$ into $\mathcal{T}'$ where $\mathcal{T}'$ is the search tree produced by the contraction process when started on $T_{S'}$. We refer to this transformation as *correcting* the LINE-LEAF TREE and divide INSERT into three corrective steps: *local correction, down correction,* and *up correction.* Local correction repairs the contraction process on $\mathcal{T}$ for elements of S that appear near A at some point during the contraction process. Down correction repairs $\mathcal{T}$ for nodes with round at most ROUND(A). Up correction repairs $\mathcal{T}$ for nodes with round at least ROUND(A). Our primary result is the following theorem.

Theorem 4. INSERT *is correct and takes* $O(h)$ *time.*

A full proof of Theorem 4 appears in the full version [10]. Here we give a detailed outline of the insertion procedure. Let X be a node such that $LST(X)$ has t edge queries $(X, Y_1) \ldots (X, Y_t)$ sorted in descending order by ROUND(Y_i). That is, Y_1 is the last node leaf-contracted into $LST(X)$, Y_t is the first node leaf-contracted into $LST(X)$ and Y_i is the $(t - i + 1)^{th}$ node contracted into $LST(X)$. Define $\rho_i(X) = Y_i$ and $\mu_i(X) = $ ROUND(Y_i). If $i > t$, then let $\mu_i(X) = 0$.

Local Correction. We start by finding the predecessor of A in T_S. Call this node B. In $H_{S'}$, A potentially falls between B and any number of $children(B)$. Thus, A may replace B as the parent of a set of nodes $D \subseteq children(B)$. We use D to identify two other sets of nodes C and L. The set C represents nodes that, in T_S, were leaf-contracted into B in the direction of some edge (B, D_j) where $D_j \in D$. The set L represents nodes that were involved in the contraction process of B itself. Depending on TYPE(B), the composition of L falls into one of the following two cases:

1. if TYPE(B) = LINE then let PARENT(B) = (E, F). Let D_E and D_F be the two neighbors of B on the path from E to F. If D_E and D_F are in D then $L = \{E, F\}$. If only D_E is in D, then $L = \{E\}$. If only D_F is in D, then $L = \{F\}$. Otherwise, $L = \emptyset$.
2. If TYPE(B) = LEAF then let PARENT(B) = E. Let D_E be the neighbor of B on the path $B \ldots E$. Let $L = \{E\}$ if D_E is in D and let $L = \emptyset$ otherwise.

If C and L are both empty, then A appears as a leaf in $T_{S'}$ and ROUND(A) = 1. In this case, we only need to correct $\mathcal{T}$ upward since the addition of A does not affect nodes contracted in earlier rounds. However, if either C or L is non-empty, then A is an interior node in $T_{S'}$ and A essentially acts as B to the stolen nodes in C. Thus, for every edge query (B, C_i) where $C_i \in C$, we remove (B, C_i) from

$LST(B)$ and insert it into $LST(A)$. In addition, we create a new edge (B, A) and add it to H_S which yields $H_{S'}$. This ends local correction.

Removing edge queries from $LST(B)$ and inserting them into $LST(A)$ may cause changes in the contraction process that reverberate upward and downward in the Line-Leaf Tree. Let $P = A$ and $Q = B$ when $\text{ROUND}(A) \leq \text{ROUND}(B)$ and let $P = B$ and $Q = A$ otherwise. Broadly, there are two interesting cases. If $\mu_1(P) \neq \mu_2(P)$ then P was potentially line contracted between $\rho_1(P)$ and Q at some earlier round. If this is the case then we must correct the contraction process *downward* on $BST(\rho_1(P), P)$ and $BST(P, Q)$. Likewise, when $\mu_1(P) = \mu_2(P)$ then $\text{ROUND}(Q)$ might increase, which in turn may affect later rounds of the contraction process. If this is the case then we must correct the contraction process *upward* on Q.

Down Correction. Here we know that P was line contracted between $\rho_1(P)$ and Q at some earlier round. The main idea of Down Correct is to float P down to the BST created in the same round as P. We do this by examining the rounds when $BST(\rho_1(P), P)$ and $BST(P, Q)$ were created and recursively calling Down Correct until we arrive at the BST with correct round.

Up Correction. In this case, we know that P increases the round of Q by one which can affect the contraction process for nodes contracted in later rounds. If Q was leaf-contracted into E (*i.e.*, $\text{TYPE}(Q) = \text{LEAF}$ and $\text{PARENT}(Q) = E$) then P replaces Q in the edge query (Q, E) since Q is now line-contracted between P and E in the iteration before. If Q was line-contracted into $BST(E, F)$ (*i.e*, $\text{TYPE}(Q) = \text{LINE}$ and $\text{PARENT}(Q) = (E, F)$) then $BST(E, F)$ is now split into $BST(E, Q)$ and $BST(Q, F)$. The interesting case is when, in $\mathcal{T}$, E was leaf-contracted into F. In $\mathcal{T}'$, the edge query (E, Q) now appears in $LST(Q)$ and we're in a position to recursively correct the contraction process upwards with Q and F replacing P and Q respectively in the recursive call.

Delete. Deletion removes a node A from a Line-Leaf Tree $\mathcal{T}$ assuming A appears in $\mathcal{T}$. As with insertion, the goal is to repair $\mathcal{T}$ so that it mimics $\mathcal{T}'$ where $\mathcal{T}'$ is the result of running the contraction process on $T_{S'}$ where $S' = S \setminus \{A\}$. Deletion is a somewhat simpler operation than insertion. This is because when we delete A, *all* of the successors of A become successors of A's predecessor B. If A outlasted B in the new contraction process, then B essentially plays the role of A in $\mathcal{T}'$. If B outlasted A, then its role does not change. The only problem is that B no longer has A as a neighbor which may create problems with nodes contracted later in the process. Repairing these problems is the technical crux of deletion. A thorough description of deletion, as well as a proof of the following Theorem also appear in the full version [10].

Theorem 5. Delete *is correct and takes* $O(\log w) \cdot OPT$ *time.*

5   Empirical Results

To conclude, we compare the height of a Line-Leaf Tree to the height of an optimal static search tree in two experimental settings: random tree-like partial
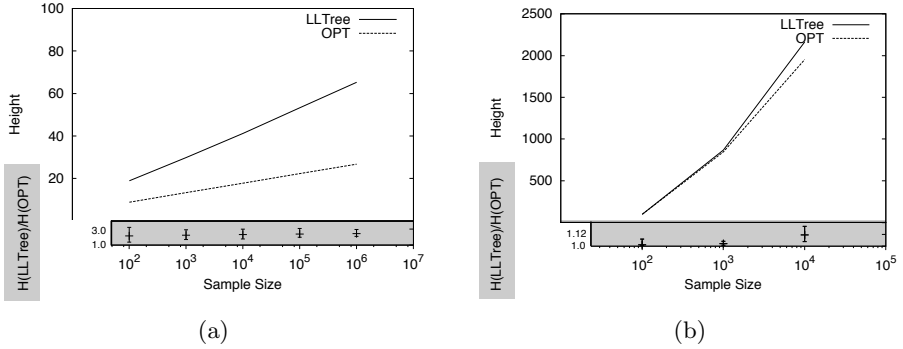
Fig. 4. Results comparing the height of the Line-Leaf Tree to the optimal static search search tree on (a) random tree-like partial orders; and (b) a large portion of the UNIX filesystem. The non-shaded areas show the average height of both the Line-Leaf Tree and optimal static algorithm. The shaded area shows their ratio (as well as the min and max values over the 1000 iterations).

orders and the UNIX directory structure. For these experiments, we consider the height of a search tree to be the maximum number of edge queries performed on any root-to-leaf path. So any dynamic edge query in a Line-Leaf Tree counts as two edge queries in our experiments.

In the first experiment, we examine tree-like partial orders of increasing size n. For each n, we independently sample 1000 partial-orders uniformly at random from all tree-like partial orders with n nodes [12] (this distribution give a tree of height $\theta(\log n)$, w.h.p. [13,14,15]).

The non-shaded area of Figure 4 (a) shows the heights of the Line-Leaf Tree and the optimal static tree averaged over the samples. The important thing to note is that both appear to grow linearly in $\log n$. We suspect that the differing slopes come mainly from the overheard of dynamic edge queries, and we conjecture that the Line-Leaf Tree performs within a small constant factor of OPT with high probability in the uniform tree-like model. The shaded area of Figure 4 (a) shows the average, minimum, and maximum approximation ratio over the samples.

Although the first experiment shows that the Line-Leaf Tree is competitive with the optimal static tree on *average* tree-like partial orders, it may be that, in practice, tree-like partial orders are distributed non-uniformly. Thus, for our second experiment, we took the `/usr` directory of an Ubuntu 10.04 Linux distribution as our universe $\mathcal{U}$ and independently sampled 1000 sets of size $n = 100$, $n = 1000$, and $n = 10000$ from $\mathcal{U}$ respectively. The `/usr` directory contains 23,328 nodes, of which 17,340 are leaves. The largest directory is `/usr/share/doc` which contains 1551 files. The height of `/usr` is 12. We believe that this directory is somewhat representative of the use cases found in our motivation. As with our first experiment, the shaded area in Figure 4 (b) shows the ratio of the height of the Line-Leaf Tree to the height of the optimal static search tree, averaged over all 1000 samples for each sample size. The non-shaded area shows the actual

heights averaged over the samples. The LINE-LEAF TREE is again very competitive with the optimal static search tree, performing at most a small constant factor more queries than the optimal search tree.

References

1. Ben-Asher, Y., Farchi, E., Newman, I.: Optimal search in trees. SIAM J. Comput. 28(6), 2090–2102 (1999)
2. Carmo, R., Donadelli, J., Kohayakawa, Y., Laber, E.S.: Searching in random partially ordered sets. Theor. Comput. Sci. 321(1), 41–57 (2004)
3. Mozes, S., Onak, K., Weimann, O.: Finding an optimal tree searching strategy in linear time. In: SODA 2008: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1096–1105. Society for Industrial and Applied Mathematics, Philadelphia (2008)
4. Onak, K., Parys, P.: Generalization of binary search: Searching in trees and forest-like partial orders. In: FOCS 2006: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, pp. 379–388. IEEE Computer Society, Washington, DC, USA (2006)
5. Dereniowski, D.: Edge ranking and searching in partial orders. Discrete Appl. Math. 156(13), 2493–2500 (2008)
6. Jacobs, T., Cicalese, F., Laber, E.S., Molinaro, M.: On the complexity of searching in trees: Average-case minimization. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6198, pp. 527–539. Springer, Heidelberg (2010)
7. Laber, E., Molinaro, M.: An approximation algorithm for binary searching in trees. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 459–471. Springer, Heidelberg (2008)
8. Daskalakis, C., Karp, R.M., Mossel, E., Riesenfeld, S., Verbin, E.: Sorting and selection in posets. In: SODA 2009: Proceedings of the Nineteenth Annual ACM-SIAM SODA, pp. 392–401. SIAM, Philadelphia (2009)
9. Daskalakis, C., Karp, R.M., Mossel, E., Riesenfeld, S., Verbin, E.: Sorting and selection in posets. CoRR abs/0707.1532 (2007)
10. Heeringa, B., Iordan, M.C., Theran, L.: Searching in dynamic tree-like partial orders. CoRR abs/1010.1316 (2010)
11. Laber, E., Nogueira, L.T.: Fast searching in trees. Electronic Notes in Discrete Mathematics 7, 1–4 (2001)
12. Meir, A., Moon, J.W.: On the altitude of nodes in random trees. Canadian Journal of Mathematics 30, 997–1015 (1978)
13. Bergeron, F., Flajolet, P., Salvy, B.: Varieties of increasing trees. In: Raoult, J.-C. (ed.) CAAP 1992. LNCS, vol. 581, pp. 24–48. Springer, Heidelberg (1992)
14. Drmota, M.: The height of increasing trees. Annals of Combinatorics 12, 373–402 (2009), doi:10.1007/s00026-009-0009-x
15. Grimmett, G.R.: Random labelled trees and their branching networks. J. Austral. Math. Soc. Ser. A 30(2), 229–237 (1980/1981)

Counting Plane Graphs: Flippability and Its Applications⋆

Michael Hoffmann[1], Micha Sharir[2,3], Adam Sheffer[2],
Csaba D. Tóth[4], and Emo Welzl[1]

[1] Institute of Theoretical Computer Science, ETH Zürich,
CH-8092 Zürich, Switzerland
{hoffmann,welzl}@inf.ethz.ch
[2] Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel
{michas,sheffera}@tau.ac.il
[3] Courant Institute of Mathematical Sciences, New York University,
New York, NY 10012, USA
[4] Department of Mathematics and Statistics, University of Calgary,
Calgary, AB, Canada
cdtoth@ucalgary.ca

Abstract. We generalize the notions of flippable and simultaneously flippable edges in a triangulation of a set S of points in the plane into so called *pseudo-simultaneously flippable edges*.

We prove a worst-case tight lower bound for the number of pseudo-simultaneously flippable edges in a triangulation in terms of the number of vertices. We use this bound for deriving new upper bounds for the maximal number of crossing-free straight-edge graphs that can be embedded on any fixed set of N points in the plane. We obtain new upper bounds for the number of spanning trees and forests as well. Specifically, let $\mathsf{tr}(N)$ denote the maximum number of triangulations on a set of N points in the plane. Then we show (using the known bound $\mathsf{tr}(N) < 30^N$) that any N-element point set admits at most $6.9283^N \cdot \mathsf{tr}(N) < 207.85^N$ crossing-free straight-edge graphs, $O(4.8795^N) \cdot \mathsf{tr}(N) = O(146.39^N)$ spanning trees, and $O(5.4723^N) \cdot \mathsf{tr}(N) = O(164.17^N)$ forests. We also obtain upper bounds for the number of crossing-free straight-edge graphs that have fewer than cN or more than cN edges, for a constant parameter c, in terms of c and N.

1   Introduction

A *crossing-free straight-edge graph* G is an embedding of a graph in the plane such that the vertices are mapped to a set S of points in the plane and the edges are pairwise non-crossing line segments between pairs of points in S. In this paper, we fix a labeled set S of points in the plane, and we only consider planar graphs that admit a straight-edge embedding with vertex set S. (In an *unlabeled* point set, isomorphic triangulations are considered identical and counted as a single triangulation.)

The first exponential bound on the number of crossing-free straight-edge graphs was proved by Ajtai *et al.* [2] back in 1982. Namely, they proved that no more than 10^{13N} such graphs can be embedded over any specific set of N points. Since then, progressively more reasonable bounds have been derived, all depending on the bound for the maximal number of *triangulations* that can be embedded over any specific point set. Obtaining sharper bounds on the number of such graphs is also a major theme of the present paper.

A triangulation of a set S of N points in the plane is a maximal crossing-free straight-edge graph on S (that is, no additional straight edges can be inserted without crossing some of the existing edges). Triangulations are an important geometric construct which is used in many algorithmic applications, and are also an interesting object of study in discrete and combinatorial geometry (recent comprehensive surveys can be found in [6,13]).

Improving the bound on the maximum number of triangulations that any set of N points in the plane can have has been a major research theme during the past 30 years. The initial upper bound 10^{13N} of [2] has been steadily improved in several paper (e.g., see [7,21,23]), culminating with the current record of 30^N due to Sharir and Sheffer [22]. Other papers have studied lower bounds on the maximal number of triangulations (e.g., [1,8]), and upper or lower bounds on the number of other kinds of planar graphs (e.g., [4,5,19,20]).

For a set S of points in the plane, we denote by $\mathcal{T}(S)$ the set of all triangulations of S, and put $\mathsf{tr}(S) := |\mathcal{T}(S)|$. Similarly, we denote by $\mathcal{P}(S)$ the set of all crossing-free straight-edge graphs on S, and put $\mathsf{pg}(S) := |\mathcal{P}(S)|$. We also let $\mathsf{tr}(N)$ and $\mathsf{pg}(N)$ denote, respectively, the maximum values of $\mathsf{tr}(S)$ and of $\mathsf{pg}(S)$, over all sets S of N points in the plane.

Let S be a set of N points in the plane. Every crossing-free straight-edge graph in $\mathcal{P}(S)$ is contained in at least one triangulation in $\mathcal{T}(S)$. Additionally, since a triangulation has fewer than $3N$ edges, every triangulation $T \in \mathcal{T}(S)$ contains fewer than $2^{3N} = 8^N$ crossing-free straight-edge graphs. This immediately implies $\mathsf{pg}(S) < 8^N \cdot \mathsf{tr}(S)$. However, this inequality seems rather weak since it potentially counts some crossing-free straight-edge graphs many times. More formally, given a graph $G \in \mathcal{P}(S)$ contained in x distinct triangulations of S, we say that G has a *support* of x, and write $\mathsf{supp}(G) = x$. Thus, every graph $G \in \mathcal{P}(S)$ will be counted $\mathsf{supp}(G)$ times in the preceding inequality.

Recently, Razen, Snoeyink, and Welzl [18] managed to break the 8^N barrier by overcoming the above inefficiency. However, they obtained only a slight improvement, with the bound $\mathsf{pg}(S) = O\left(7.9792^N\right) \cdot \mathsf{tr}(S)$. Using a careful analysis

of the supports of graphs in $\mathcal{P}(S)$, we give a more significant improvement with an upper bound of $6.9283^N \cdot \mathsf{tr}(N)$. Combining this bound with the recent bound $\mathsf{tr}(S) < 30^N$ [22], we get $\mathsf{pg}(N) < 207.85^N$. We provide similar bounds for the numbers of crossing-free straight-edge spanning trees and forests (i.e, cycle-free graphs). Table 1 summarizes these results[1].

Table 1. Upper and lower bounds for the number of several types of crossing-free straight-edge graphs on a set of N points in the plane. By plane graphs, we mean all crossing-free straight-edge graphs over a specific point set. Our new bounds are in the right two columns.

Graph type	Lower bound	Previous upper bound	New upper bound	In the form $a^N \cdot \mathsf{tr}(N)$
Plane Graphs	$\Omega(41.18^N)$ [1]	$O(239.4^N)$ [18,22]	$\mathbf{207.85^N}$	$\mathbf{6.9283^N \cdot tr(N)}$
Spanning Trees	$\Omega(11.97^N)$ [8]	$O(158.6^N)$ [5,22]	$\mathbf{O(146.39^N)}$	$\mathbf{O(4.8795^N) \cdot tr(N)}$
Forests	$\Omega(12.23^N)$ [8]	$O(194.7^N)$ [5,22]	$\mathbf{O(164.17^N)}$	$\mathbf{O(5.4723^N) \cdot tr(N)}$

We also derive similar bounds for the number of crossing-free straight-edge graphs that can be embedded on a fixed set S and that have at most $c|S|$ edges, and for the number of such graphs with at least $c|S|$ edges, for $0 < c < 3$. More precisely, the former bound applies for $0 < c \le 5/4$, and the latter bound applies for $7/4 \le c < 3$. For both cases we obtain the bound[2]

$$O^* \left(\left(\frac{5^{5/2}}{8(c+t-1/2)^{c+t-1/2}(3-c-t)^{3-c-t}(2t)^t(1/2-t)^{1/2-t}} \right)^N \cdot \mathsf{tr}(S) \right) \ ,$$

where $t = \frac{1}{2} \left(\sqrt{(7/2)^2 + 3c + c^2} - 5/2 - c \right)$.

Triangulations: Notations and simple facts. We only consider point sets S in general position, that is, no three points in S are collinear. For upper bounds on the number of graphs, this involves no loss of generality, because the number of graphs can only increase if collinear points are slightly perturbed into general position.

Every triangulation of S contains the edges of the convex hull of S, and the remaining edges of the triangulation decompose the interior of the convex hull into triangular faces. Assume that S contains N points, h of which are on the convex hull and the remaining $n = N - h$ points are interior to the hull (we use this notation throughout). By Euler's formula, every triangulation of S has $3n + 2h - 3$ edges (h *hull* edges, common to all triangulations, and $3n+h-3$ *interior* edges, each adjacent to two triangles), and $2n+h-2$ bounded (triangular) faces.

---

[1] Up-to-date bounds for these and for other families of graphs can be found in `http://www.cs.tau.ac.il/~sheffera/counting/PlaneGraphs.html` (version of February 2011).

[2] In the notations $O^*()$, $\Theta^*()$, and $\Omega^*()$, we neglect polynomial factors.
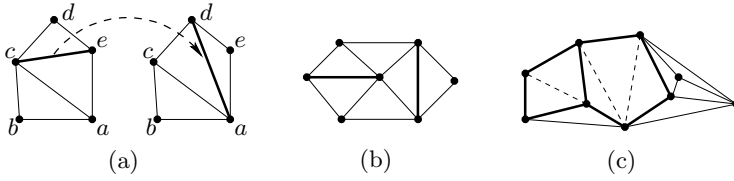
Fig. 1. (a) The edge *ce* can be flipped to the edge *ad*. (b) The two bold edges are simultaneously flippable. (c) Interior-disjoint convex quadrilateral and convex pentagon in a triangulation.

Edge Flips. Edge flips are simple operations that replace one or several edges of a triangulation with new edges and produce a new triangulation. As we will see in Sect. 2, edge flips are instrumental for counting various classes of subgraphs in triangulations. In the next few paragraphs, we review previous results on edge flips, and propose a new type of edge flip. We say that an interior edge in a triangulation of S is *flippable*, if its two adjacent triangles form a convex quadrilateral. A flippable edge can be *flipped*, that is, removed from the graph of the triangulation and replaced by the other diagonal of the corresponding quadrilateral, thereby obtaining a new triangulation of S. An edge flip operation is depicted in Fig. 1(a), where the edge *ce* is flipped to the edge *ad*. Already in 1936, Wagner [27] has shown that any *unlabeled non-embedded* triangulation T can be transformed into any other triangulation T' (with the same number of vertices) through a series of edge-flips (here one uses a more abstract notion of an edge flip). When we deal with a pair of triangulations over a specific common (labeled) set S of points in the plane, there always exists such a sequence of $O(|S|^2)$ flips, and this bound is tight in the worst case (e.g., see [3,16]). Moreover, there are algorithms that perform such sequences of flips to obtain some "optimal" triangulation (typically, the Delaunay triangulation; see [9] for example), which, as a by-product, provide an edge-flip sequence between any specified pair of triangulations of S.

How many flippable edges can a single triangulation have? Given a triangulation T, we denote by $\mathsf{flip}(T)$ the number of flippable edges in T. Hurtado, Noy, and Urrutia [16] proved the following lower bound.

Lemma 1. [16] *For any triangulation T over a set of N points in the plane, $\mathsf{flip}(T) \geq N/2 - 2$. Moreover, there are triangulations (of specific point sets of arbitrarily large size) for which this bound is tight.*
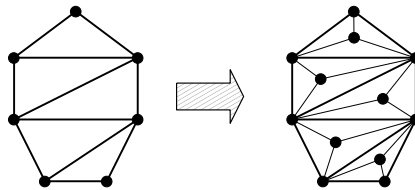

Fig. 2. Constructing a triangulation with $N/2 - 2$ flippable edges

To obtain a triangulation with exactly $N/2 - 2$ flippable edges, start with a convex polygon with $N/2 + 1$ vertices, triangulate it in some arbitrary manner, insert a new point into each of the $N/2 - 1$ resulting bounded triangles, and connect each new point p to the three hull vertices that form the triangle containing p. Such a construction is depicted in Fig. 2. The resulting graph is a triangulation with N vertices and exactly $N/2 - 2$ flippable edges, namely the chords of the initial triangulation.

Simultaneous Flippability. We say that two flippable edges e and e' of a triangulation T are *simultaneously flippable* if no triangle of T is incident to both edges; equivalently, the quadrilaterals corresponding to e and e' are interior-disjoint. See Fig. 1(b) for an illustration. Notice that flipping an edge e cannot affect the flippability of any edge simultaneously flippable with e. Given a triangulation T, let $\mathsf{flip}_s(T)$ denote the size of the largest subset of edges of T, such that every pair of edges in the subset are simultaneously flippable. The following lemma is taken from Souvaine *et al.* [24].

Lemma 2. [24] *For any triangulation T over a set of N points in the plane,* $\mathsf{flip}_s(T) \geq (N - 4)/5$.

Galtier et al. [10] show that this bound is tight in the worst case, by presenting a specific triangulation in which at most $(N - 4)/5$ edges are simultaneously flippable.

Pseudo-simultaneously flippable edge sets. A set of simultaneously flippable edges in a triangulation T can be considered as the set of diagonals of a collection of interior-disjoint convex quadrilaterals. We consider a more liberal definition of simultaneously flippable edges, by taking, within a fixed triangulation T, the diagonals of a set of interior-disjoint convex polygons, each with at least four edges (so that the boundary edges of these polygons belong to T). Consider such a collection of convex polygons $Q_1, \ldots, Q_m$, where Q_i has $k_i \geq 4$ edges, for $i = 1, \ldots, m$. We can then retriangulate each Q_i independently, to obtain many different triangulations. Specifically, each Q_i can be triangulated in $C_{k_i - 2}$ ways, where C_j is the j-th *Catalan number* (see, e.g., [25, Section 5.3]). Hence, we can get $M = \prod_{i=1}^{m} C_{k_i - 2}$ different triangulations in this way. In particular, if a graph $G \subseteq T$ (namely, a graph all of whose edges are edges of T) does not contain any diagonal of any Q_i (it may contain boundary edges though) then G is a subgraph of (at least) M distinct triangulations. An example is depicted in Fig. 1(c), where by "flipping" (or rather, redrawing) the diagonals of the highlighted convex quadrilateral and pentagon, we can get $C_2 \cdot C_3 = 2 \cdot 5 = 10$ different triangulations. We say that a set of interior edges in a triangulation is *pseudo-simultaneously flippable* (*ps-flippable* for short) if after the deletion of these edges every bounded face of the remaining graph is convex, and there are no vertices of degree 0. Notice that all three notions of flippability are defined within a fixed triangulation T of S (although each of them gives a recipe for generating many other triangulations).

Table 2. Bounds for minimum numbers of the various types of flippable edges in a triangulation of N points. All of these bounds are tight in the worst case.

Edge Type	Lower bound
Flippable	$N/2 - 2$ [16]
Simultaneously flippable	$N/5 - 4/5$ [10,24]
Ps-flippable	$\max\{N/2 - 2, h - 3\}$

We derive a lower bound on the size of the largest set of ps-flippable edges in a triangulation, and show that this bound is tight in the worst case. Specifically, we have the following *ps-flippability lemma*:

Lemma 3. *Let S be a set of N points in the plane, and let T be a triangulation of S. Then T contains a set of at least $\max\{N/2 - 2, h - 3\}$ ps-flippable edges. This bound cannot be improved.*

Lemma 3 is the major technical contribution of this paper. See the full version of this paper [14] for its proof. Table 2 summarizes the bounds for minimum numbers of the various types of flippable edges in a triangulation.

In Sect. 2, we use Lemma 3 to derive several upper bounds on the numbers of crossing-free straight-edge graphs of various kinds embedded as crossing-free straight-edge graphs on a fixed set S.
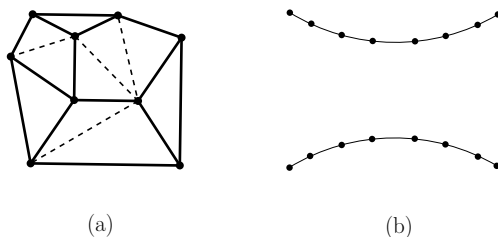

(a)   (b)

Fig. 3. (a) A convex decomposition of S. When completing it into a triangulation, the added (dashed) diagonals form a set of ps-flippable edges. This is one of the $C_2 \cdot C_2 \cdot C_3 = 20$ possible completions. (b) A double chain configuration with 16 vertices.

Convex decompositions. Ps-flippable edges are closely related to the notion of *convex decompositions* of a point set S. These are crossing-free straight-edge graphs on S such that (i) they include all the hull edges, (ii) each of their bounded faces is a convex polygon, and (iii) no point of S is isolated. See Fig. 3(a) for an illustration. Urrutia [26] asked what is the minimum number of faces that can always be achieved in a convex decomposition of any set of N points in the plane. Hosono [15] proved that every planar set of N points admits a convex decomposition with at most $\lceil \frac{7}{5}(N+2) \rceil$ (bounded) faces. For every $N \geq 4$, García-Lopez and Nicolás [11] constructed N-element point sets that do not admit a convex decomposition with fewer than $\frac{12}{11}N - 2$ faces. By Euler's formula, if a connected crossing-free straight-edge graph has N vertices and e edges, then

it has $e - N + 2$ faces (including the exterior face). It follows that for convex decompositions, minimizing the number of faces is equivalent to minimizing the number of edges. (For convex decompositions contained in a given triangulation, this is also equivalent to maximizing the number of removed edges, which form a set of ps-flippable edges.)

Lemma 3 directly implies the following corollary. (The bound that it gives is weaker than the bound in [15], but it holds for every triangulation.)

Corollary 1. *Let S be a set of N points in the plane, so that its convex hull has h vertices, and let T be a triangulation of S. Then T contains a convex decomposition of S with at most $\frac{3}{2}N - h \leq \frac{3}{2}N - 3$ convex faces and at most $\frac{5}{2}N - h - 1 \leq \frac{5}{2}N - 4$ edges. Moreover, there exist point sets S of arbitrarily large size, and triangulations $T \in \mathcal{T}(S)$ for which these bounds are tight.*

Notation. Here are some additional notations that we use.

For a triangulation T and an integer $i \geq 3$, let $v_i(T)$ denote the number of interior vertices of degree i in T.

Given two crossing-free straight-edge graphs G and H over the same point set S, we write $G \subseteq H$ provided every edge in G is also an edge in H.

Similarly to the case of edges, the *hull vertices* (resp., *interior vertices*) of a set S of points in the plane are those that are part of the boundary of the convex hull of S (resp., not part of the convex hull boundary).

2   Applications of Ps-Flippable Edges

In this section we apply the ps-flippability lemma (Lemma 3) to obtain several improved bounds on the number of crossing-free straight-edge graphs of various kinds on a fixed set of points in the plane.

2.1   The Ratio between the Number of Crossing-Free Straight-Edge Graphs and the Number of Triangulations

Let S be a set of N points in the plane. Recall that the relation $\mathsf{pg}(S) < 8^N \cdot \mathsf{tr}(S)$ can be obtained by noticing that every triangulation of S contains at most 8^N crossing-free straight-edge graphs. Also, recall that this inequality is rather weak, since every graph $G \in \mathcal{P}(S)$ will be counted $\mathsf{supp}(G)$ times. We overcome this inefficiency, using a technique that relies on the ps-flippability lemma.

Theorem 1. *For every set S of N points in the plane, h of which are on the convex hull,*

$$\mathsf{pg}(S) \leq \begin{cases} \frac{(4\sqrt{3})^N}{2^h} \cdot \mathsf{tr}(S) < \frac{6.9283^N}{2^h} \cdot \mathsf{tr}(S), & \text{for } h \leq N/2 \ , \\ 8^N \left(3/8\right)^h \cdot \mathsf{tr}(S), & \text{for } h > N/2 \ . \end{cases}$$

Proof. The exact value of $\mathsf{pg}(S)$ is easily seen to be

$$\mathsf{pg}(S) = \sum_{T \in \mathcal{T}(S)} \sum_{\substack{G \in \mathcal{P}(S) \\ G \subseteq T}} \frac{1}{\mathsf{supp}(G)} \quad , \tag{1}$$

because every graph G appears $\mathsf{supp}(G)$ times in the sum, and thus contributes a total of $\mathsf{supp}(G) \cdot \frac{1}{\mathsf{supp}(G)} = 1$. We obtain an upper bound on this sum as follows. Consider a graph $G \in \mathcal{P}(S)$ and a triangulation $T \in \mathcal{T}(S)$, such that $G \subseteq T$. By Lemma 3, there is a set F of $t = \max(N/2 - 2, h - 3)$ ps-flippable edges in T. Let $F_{\bar{G}}$ denote the set of edges that are in F but *not* in G, and put $j = |F_{\bar{G}}|$. Removing the edges of $F_{\bar{G}}$ from T yields a convex decomposition of S which still contains G and whose non-triangular interior faces have a total of j missing diagonals. Suppose that there are m such faces, with $j_1, j_2, \ldots, j_m$ diagonals respectively, where $\sum_{k=1}^{m} j_k = j$. Then these faces can be triangulated in $\prod_{k=1}^{m} C_{j_k+1}$ ways, and each of the resulting triangulations contains G. Since $C_{i+1} \geq 2^i$ holds for any $i \geq 1$, we have $\mathsf{supp}(G) \geq 2^j$. (Equality occurs if and only if all the non-triangular faces of $T \setminus F_{\bar{G}}$ are quadrilaterals.)

Next, we estimate the number of subgraphs $G \subseteq T$ for which the set $F_{\bar{G}}$ is of size j. Denote by E the set of edges of T that are not in F, and assume that the convex hull of S has h vertices. Since there are $3N - 3 - h$ edges in any triangulation of S, $|E| \leq 3N - 3 - h - t$. To obtain a graph G for which $|F_{\bar{G}}| = j$, we choose any subset of edges from E, and any j edges from F (the j edges of F that will not belong to G). Therefore, the number of such subgraphs is at most $2^{3N-h-t-3} \cdot \binom{t}{j}$. We can thus rewrite (1) to obtain

$$\mathsf{pg}(S) \leq \sum_{T \in \mathcal{T}(S)} \sum_{j=0}^{t} 2^{3N-h-t-3} \cdot \binom{t}{j} \cdot \frac{1}{2^j}$$

$$= \mathsf{tr}(S) \cdot 2^{3N-h-t-3} \sum_{j=0}^{t} \binom{t}{j} \frac{1}{2^j} = \mathsf{tr}(S) \cdot 2^{3N-h-t-3} \cdot (3/2)^t \quad .$$

If $t = N/2 - 2$, we get $\mathsf{pg}(S) < \mathsf{tr}(S) \cdot \frac{(4\sqrt{3})^N}{2^h} < \frac{6.9283^N}{2^h} \cdot \mathsf{tr}(S)$. If $t = h - 3$, we have $\mathsf{pg}(S) \leq \mathsf{tr}(S) \cdot 2^{3N-2h} \cdot (3/2)^h = \mathsf{tr}(S) \cdot 8^N \cdot (3/8)^h$. To complete the proof, we note that $N/2 - 2 > h - 3$ when $h < n + 2$, or $h < N/2 + 1$. □

For a lower bound on $\mathsf{pg}(S)/\mathsf{tr}(S)$, we consider the *double chain* configurations, presented in [12] (and depicted in Fig. 3(b)). It is shown in [12] that, when S is a double chain configuration, $\mathsf{tr}(S) = \Theta^*(8^N)$ and $\mathsf{pg}(S) = \Theta^*(39.8^N)$ (the upper bound for $\mathsf{pg}(S)$ appears in [1]). Thus, we have $\mathsf{pg}(S) = \Theta^*(4.95^N) \cdot \mathsf{tr}(S)$ (for this set $h = 4$, so h has no effect on the asymptotic bound of Theorem 1).

Recall the notations $\mathsf{tr}(N) = \max_{|S|=N} \mathsf{tr}(S)$ and $\mathsf{pg}(N) = \max_{|S|=N} \mathsf{pg}(S)$. Combining the bound $\mathsf{tr}(N) < 30^N$ [22], with the first bound of Theorem 1 implies $\mathsf{pg}(N) < 207.849^N$. The bound improves significantly as h gets larger.

2.2   The Number of Spanning Trees and Forests

Spanning Trees. For a set S of N points in the plane, we denote by $\mathcal{ST}(S)$ the set of all crossing-free straight-edge spanning trees of S, and put $\mathsf{st}(S) := |\mathcal{ST}(S)|$. Moreover, we let $\mathsf{st}(N) = \max_{|S|=N} \mathsf{st}(S)$.

Buchin and Schulz [5] have recently shown that every crossing-free straight-edge graph contains $O\left(5.2852^N\right)$ spanning trees, improving upon the earlier bound of $5.\bar{3}^N$ due to Ribó Mor and Rote [19,20]. We thus get $\mathsf{st}(S) = O\left(5.2852^N\right) \cdot \mathsf{tr}(S)$ for every set S of N points in the plane. The bound from [5] cannot be improved much further, since there are triangulations with at least 5.0295^N spanning trees [19,20]. However, the ratio between $\mathsf{st}(S)$ and $\mathsf{tr}(S)$ can be improved beyond that bound, by exploiting the fact that, as in the case of general graphs, some spanning trees may get multiply counted in many triangulations.

We now derive such an improved ratio by using ps-flippable edges. The proof goes along the same lines of the proof of Theorem 1.

Theorem 2. *For every set S of N points in the plane, $\mathsf{st}(S) = O\left(4.8795^N\right) \cdot \mathsf{tr}(S)$.*

Proof.   We have $\mathsf{st}(S) = \displaystyle\sum_{T \in \mathcal{T}(S)} \sum_{\substack{\tau \in \mathcal{ST}(S) \\ \tau \subset T}} \frac{1}{\mathsf{supp}(\tau)}$. Consider a spanning tree $\tau \in \mathcal{ST}(S)$ and a triangulation $T \in \mathcal{T}(S)$, such that $\tau \subset T$. As in Theorem 1, let F be a set of $N/2 - 2$ ps-flippable edges in T. Also, let $F_{\bar{\tau}}$ denote the set of edges that are in F but *not* in τ, and put $j = |F_{\bar{\tau}}|$. Thus, as argued earlier, $\mathsf{supp}(\tau) \geq 2^j$.

Next, we estimate the number of spanning trees $\tau \subset T$ for which the set $F_{\bar{\tau}}$ is of size j. First, there are $\binom{|F|}{j} < \binom{N/2}{j}$ ways to choose the j edges of F that τ does not use. Then τ uses $N/2 - 2 - j$ edges of F, and its other $N/2 + j + 1$ edges have to be chosen from the complementary set E, as in the preceding proof. Since $|E| < 5N/2 - h$, there are fewer than $\binom{N/2}{j} \cdot \binom{5N/2 - h}{N/2 + j + 1}$ spanning trees $\tau \subset T$ with $|F_{\bar{\tau}}| = j$. Notice that we may count many graphs that are not trees, but this does not affect the validity of the upper bound. However, when j is large, it is better to use the bound $O\left(5.2852^N\right)$ from [5] instead.

We thus get, for a threshold parameter $a < 0.5$ that we will set in a moment,

$$\mathsf{st}(S) < \sum_{T \in \mathcal{T}(S)} \left(\sum_{j=0}^{aN} \binom{N/2}{j} \cdot \binom{5N/2 - h}{N/2 + j + 1} \cdot \frac{1}{2^j} + \sum_{j=aN+1}^{N/2} O\left(5.2852^N\right) \cdot \frac{1}{2^j} \right) .$$

The terms in the first sum over j increase when $a \leq 0.5$, so the sum is at most $N/2$ times its last term. Using Stirling's formula and ignoring the effect of h, we get that for $a \approx 0.1152$, the last term in the first sum is $\Theta^*\left(5.2852^N / 2^{aN}\right) = O\left(4.8795^N\right)$. Since this also bounds the second sum, we get

$$\mathsf{st}(S) < \sum_{T \in \mathcal{T}(S)} O\left(4.8795^N\right) = O\left(4.8795^N\right) \cdot \mathsf{tr}(S) .$$

(a was optimized numerically.) □

Combining the bound just obtained with $\mathsf{tr}(N) < 30^N$ [22] implies

Corollary 2. $\mathsf{st}(N) = O\left(146.385^N\right).$

This improves all previous bounds, the smallest of which is $O(158.6^N)$ [5,22].

Forests. For a set S of N points in the plane, we denote by $\mathcal{F}(S)$ the set of all crossing-free straight-edge forests (i.e., cycle-free graphs) of S, and put $\mathsf{f}(S) := |\mathcal{F}(S)|$. Moreover, we let $\mathsf{f}(N) = \max_{|S|=N} \mathsf{f}(S)$. Buchin and Schulz [5] have recently shown that every crossing-free straight-edge graph contains $O\left(6.4884^N\right)$ forests (improving the simpler bound $O^*(6.75^N)$, noted below). Using this bound, we obtain

Theorem 3. *For every set S of N points in the plane,* $\mathsf{f}(S) = O\left(5.4723^N\right) \cdot \mathsf{tr}(S).$

The proof is a simple adaptation of the proof of Theorem 2. For more details, see the full version of the paper [14]. As in the previous cases, we can combine this with the bound $\mathsf{tr}(N) < 30^N$ [22] to obtain

Corollary 3. $\mathsf{f}(N) = O\left(164.169^N\right).$

This should be compared with the best previous upper bound $O(194.7^N)$ [5,22].

2.3   The Number of Crossing-Free Straight-Edge Graphs with a Bounded Number of Edges

In this subsection we derive an upper bound for the number of crossing-free straight-edge graphs on a set S of N points in the plane, with some constraints on the number of edges. Specifically, we bound the number of crossing-free straight-edge graphs with at most cN edges (for $0 < c \leq 5/4$) and the number of crossing-free straight-edge graphs with at least cN edges (for $7/4 \leq c < 3$).

For a set S of N points in the plane and a constant $0 < c \leq 3$, we denote by $\mathcal{P}_c(S)$ (resp., $\bar{\mathcal{P}}_c(S)$) the set of all crossing-free straight-edge graphs of S with at most cN edges (resp., at least cN edges), and put $\mathsf{pg}_c(S) := |\mathcal{P}_c(S)|$ (resp., $\bar{\mathsf{pg}}_c(S) := |\bar{\mathcal{P}}_c(S)|$).

For $c \leq 1.5$, we can obtain a trivial upper bound for $\mathsf{pg}_c(S)$ by bounding the maximal number of such crossing-free straight-edge graphs that any fixed triangulation of S can contain. Using Stirling's formula, we have

$$\mathsf{pg}_c(S) < \mathsf{tr}(S) \cdot \left(\binom{3N}{0} + \binom{3N}{1} + \cdots + \binom{3N}{cN} \right)$$

$$= O^* \left(\mathsf{tr}(S) \cdot \binom{3N}{cN} \right) = O^* \left(\mathsf{tr}(S) \cdot \left(\frac{27}{c^c \cdot (3-c)^{3-c}} \right)^N \right) .$$

For example, the above implies that there are at most $O^*\left(6.75^N\right) \cdot \mathsf{tr}(S)$ crossing-free straight-edge graphs with at most N edges, over every set S of N points in the plane. In particular, this is also an upper bound on the number of crossing-free straight-edge forests on S (a bound already observed in [1] and mentioned

above), or of spanning trees, or of spanning cycles. Of course, better bounds exist for these three special cases, as demonstrated earlier in this paper for the first two bounds. A bound for the number of graphs with a least $c \geq 1.5$ edges can be obtained symmetrically.

We now present a theorem that improves these trivial bounds. For the proof, together with a couple of applications, see the full version [14].

Theorem 4. *For every set S of N points in the plane and $0 < c \leq 5/4$,*

$$\frac{\mathsf{pg_c}(S)}{\mathsf{tr}(S)} = O^* \left(\left(\frac{5^{5/2}}{8(c + t - 1/2)^{c+t-1/2}(3 - c - t)^{3-c-t}(2t)^t(1/2 - t)^{1/2-t}} \right)^N \right) ,$$

where $t = \left(\sqrt{(7/2)^2 + 3c + c^2} - 5/2 - c \right)/2$. The same upper bound also applies to $\bar{\mathsf{pg}}_c(S)/\mathsf{tr}(S)$ when $7/4 \leq c < 3$.

3   Conclusion

In this paper we have introduced the notion of pseudo-simultaneously flippable edges in triangulations, have shown that many such edges always exist, and have used them to obtain several refined bounds on the number of crossing-free straight-edge graphs on a fixed (labeled) set of N points in the plane. The paper raises several open problems and directions for future research.

One such question is whether it is possible to further extend the notion of ps-flippability. For example, one could consider, within a fixed triangulation T, the set of diagonals of a collection of pairwise interior-disjoint simple, but not necessarily convex, polygons. The number of such diagonals is likely to be larger than the size of the maximal set of ps-flippable edges, but it not clear how large is the number of triangulations that can be obtained by redrawing diagonals.

We are currently working on two extensions to this work. The first extends our techniques to the cases of crossing-free straight-edge perfect matchings and spanning (Hamiltonian) cycles. This is done within the linear-algebra framework introduced by Kasteleyn (see [4,17]). The second work studies charging schemes in which the charge is moved across certain objects belonging to different crossing-free straight-edge graphs over the same point set. This cross-graph charging scheme allows us to obtain bounds that do not depend on the current upper bound for $\mathsf{tr}(N)$ (or depend on $\mathsf{tr}(N)$ in a non-linear fashion).

References

1. Aichholzer, O., Hackl, T., Huemer, Hurtado, F., Krasser, H., Vogtenhuber, B.: On the number of plane geometric graphs. Graph. Comb. 23(1), 67–84 (2007)
2. Ajtai, M., Chvátal, V., Newborn, M.M., Szemerédi, E.: Crossing-free subgraphs. Annals Discrete Math. 12, 9–12 (1982)
3. Bose, P., Hurtado, F.: Flips in planar graphs. Comput. Geom. Theory Appl. 42(1), 60–80 (2009)

4. Buchin, K., Knauer, C., Kriegel, K., Schulz, A., Seidel, R.: On the number of cycles in planar graphs. In: Lin, G. (ed.) COCOON 2007. LNCS, vol. 4598, pp. 97–107. Springer, Heidelberg (2007)
5. Buchin, K., Schulz, A.: On the number of spanning trees a planar graph can have. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6346, pp. 110–121. Springer, Heidelberg (2010)
6. De Loera, J.A., Rambau, J., Santos, F.: Triangulations: Structures for Algorithms and Applications. Springer, Berlin (2010)
7. Denny, M.O., Sohler, C.A.: Encoding a triangulation as a permutation of its point set. In: Proc. 9th Canadian Conf. on Computational Geometry, pp. 39–43 (1997)
8. Dumitrescu, A., Schulz, A., Sheffer, A., Tóth, C.D.: Bounds on the maximum multiplicity of some common geometric graphs. In: Proc. 28th International Symposium on Theoretical Aspects of Computer Science, pp. 637–648 (2011)
9. Fortune, S.: Voronoi diagrams and Delaunay triangulations. In: Du, D.A., Hwang, F.K. (eds.) Euclidean Geometry and Computers, pp. 193–233. World Scientific Publishing Co., New York (1992)
10. Galtier, J., Hurtado, F., Noy, M., Pérennes, S., Urrutia, J.: Simultaneous edge flipping in triangulations. Internat. J. Comput. Geom. Appl. 13(2), 113–133 (2003)
11. García-Lopez, J., Nicolás, M.: Planar point sets with large minimum convex partitions. In: Proc. 22nd Euro. Workshop Comput. Geom., Delphi, pp. 51–54 (2006)
12. García, A., Noy, M., Tejel, J.: Lower bounds on the number of crossing-free subgraphs of K_N. Comput. Geom. Theory Appl. 16(4), 211–221 (2000)
13. Hjelle, Ø., Dæhlen, M.: Triangulations and Applications. Springer, Berlin (2009)
14. Hoffmann, M., Sharir, M., Sheffer, A., Tóth, C.D., Welzl, E.: Counting Plane Graphs: Flippability and its Applications, arXiv:1012.0591
15. Hosono, K.: On convex decompositions of a planar point set. Discrete Math. 309, 1714–1717 (2009)
16. Hurtado, F., Noy, M., Urrutia, J.: Flipping edges in triangulations. Discrete Comput. Geom. 22, 333–346 (1999)
17. Lovász, L., Plummer, M.: Matching theory. North Holland, Amsterdam (1986)
18. Razen, A., Snoeyink, J., Welzl, E.: Number of crossing-free geometric graphs vs. triangulations. Electronic Notes in Discrete Math. 31, 195–200 (2008)
19. Ribó, A.: Realizations and Counting Problems for Planar Structures: Trees and Linkages, Polytopes and Polyominos, Ph.D. thesis. Freie Universität Berlin (2005)
20. Rote, G.: The number of spanning trees in a planar graph. Oberwolfach Reports 2, 969–973 (2005)
21. Santos, F., Seidel, R.: A better upper bound on the number of triangulations of a planar point set. J. Combinat. Theory Ser. A 102(1), 186–193 (2003)
22. Sharir, M., Sheffer, A.: Counting triangulations of planar point sets. Electr. J. Comb. 18(1) (2011), arXiv:0911.3352v2
23. Sharir, M., Welzl, E.: Random triangulations of planar point sets. In: Proc. 17th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 860–869 (2006)
24. Souvaine, D.L., Tóth, C.D., Winslow, A.: Simultaneously flippable edges in triangulations. In: Proc. XIV Spanish Meeting on Comput. Geom. (to appear, 2011)
25. Stanley, R.P.: Enumerative Combinatorics, vol. 2. Cambridge University Press, Cambridge (1999)
26. Urrutia, J.: Open problem session. In: Proc. 10th Canadian Conference on Computational Geometry. McGill University, Montréal (1998)
27. Wagner, K.: Bemerkungen zum Vierfarbenproblem. J. Deutsch. Math.-Verein. 46, 26–32 (1936)

Geometric Computations on Indecisive Points[*]

Allan Jørgensen[1], Maarten Löffler[2], and Jeff M. Phillips[3]

[1] MADALGO
[2] University of California, Irvine
[3] University of Utah

Abstract. We study computing with indecisive point sets. Such points have spatial uncertainty where the true location is one of a finite number of possible locations. This data arises from probing distributions a few times or when the location is one of a few locations from a known database. In particular, we study computing distributions of geometric functions such as the radius of the smallest enclosing ball and the diameter. Surprisingly, we can compute the distribution of the radius of the smallest enclosing ball exactly in polynomial time, but computing the same distribution for the diameter is #P-hard. We generalize our polynomial-time algorithm to all LP-type problems. We also utilize our indecisive framework to deterministically and approximately compute on a more general class of uncertain data where the location of each point is given by a probability distribution.

1 Introduction

We consider uncertain data point sets where each element of the set is not known exactly, but rather is represented by a finite set of candidate elements, possibly weighted, describing the finite set of possible true locations of the data point. The weight of a candidate location governs the probability that the data point is at that particular location. We call a point under this representation an *indecisive point*. Given indecisive input points we study computing full probability distributions (paramount for downstream analysis) over the value of some geometric query function, such as the radius of the smallest enclosing ball.

Indecisive points appear naturally in many applications. They play an important role in databases [7,1,6,5,14], machine learning [3], and sensor networks [18] where a limited number of probes from a certain data set are gathered, each potentially representing the true location of a data point. Alternatively, data points may be obtained using imprecise measurements or are the result of inexact earlier computations.

We can generalize the classification of indecisive points to when the true location of each data point is described by a probability distribution. We call these points *uncertain points*. In addition to indecisive points, this general class also

---

includes for instance multivariate normal distributions and all points within a unit disk. More broadly, these data points could represent any (uncertain) geometric object, such as a hyperplane or disc; but since these objects can usually be dualized to points, our exposition will focus on points.

Related work on uncertain points. One of the earliest models for uncertain points was a simple circular region [8], where each point has an infinite set of possible locations: all locations inside a given unit disk. This model has received considerable attention since [2,9] and can be extended to more complicated regions [16]. Most work in these models focuses on computing the minimum or maximum possible values of a query.

The full model of uncertain points, where each point's location is represented by a probability distribution, has received much less attention in the algorithms community, mainly because its generality is difficult to handle and exact solutions seem impossible for all but the most simple questions. Löffler and Phillips [11] study simple randomized algorithms.

There has also been a recent flurry of activity in the database community [7] on problems such as indexing [14], clustering [6], and histogram building [5]. However, the results with detailed algorithmic analysis generally focus on one-dimensional data; furthermore, they often only return the expected value or the most likely answer instead of calculating a full distribution.

Contribution. We study several geometric measures on sets of indecisive points in Section 2. We compute an exact distribution over the values that these measures can take, not just an expected or most-likely value. Surprisingly, while for some measures we present polynomial time algorithms (e.g. for the radius of the smallest enclosing ball), other seemingly very similar measures either are #P-Hard (e.g. the diameter) or have a solution size exponential in the number of indecisive input points (e.g. the area of the convex hull). In particular, we show that the family of problems which admit polynomial-time solutions includes *all* LP-type problems [13] with constant combinatorial dimension. #P-hardness results for indecisive data have been shown before [7], but the separation has not been understood as precisely, nor from a geometric perspective.

In Section 3 we extend the above polynomial-time algorithms to uncertain points. We describe detailed results for data points endowed with multivariate normal distributions representing their location. We deterministically reduce uncertain points to indecisive points by creating ε-samples (aka ε-approximations) of their distributions; however, this is not as straightforward as it may seem. It requires a structural theorem (Theorem 4) describing a special range space which can account for the dependence among the distributions, dependence that is created by the measure being evaluated. This is required even when the distributions themselves are independent because once an ε-sample has been fixed for one uncertain point, the other ε-samples need to account for the interaction of those fixed points with the measure. All together, these results build important structure required to transition between sets of continuous and discrete distributions with bounded error, and may be of independent interest.

These results provide a deterministic alternative to some of the results in [11]. The determinism in these algorithms is important when there can be no probability of failure, the answer needs to be identical each time it is computed, or when each indecisive point has a small constant number of possible locations. For space, several proofs and extensions are deferred to a full version.

2   Exact Computations on Indecisive Point Sets

In this section, we assume that we are given a set of n indecisive points, that is, a collection $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ of n sets containing k points each, so $Q_i = \{q_{i1}, q_{i2}, \ldots, q_{ik}\}$. We say that a set P of n points is a *support* from $\mathcal{Q}$ if it contains exactly one point from each set Q_i, that is, if $P = \{p_1, p_2, \ldots, p_n\}$ with $p_i \in Q_i$. In this case we also write $P \Subset \mathcal{Q}$.

When given a set of indecisive points, we assume that each indecisive point is at each of its locations with a fixed probability. Often these probabilities are equal, but for completeness, we describe our algorithms for when each point has a distinct probability of being the true position. For each $Q_i \in \mathcal{Q}$ and each $q_{ij} \in Q_i$, let $w(q_{ij})$ be a positive weight. We enforce $\sum_{j=1}^{k} w(q_{ij}) = k$, and $w(q_{ij})/k$ is the probability that q_{ij} is the true position of Q_i.

We let f denote the function we are interested in computing on $\mathcal{Q}$, that is, f takes a set of n points as input and computes a single real number as output. Since $\mathcal{Q}$ is indecisive, the value of f is not fixed, rather we are interested in the distribution of the possible values. We show that for some measures, we can compute this distribution in polynomial time, while for others it is #P-hard or the solution itself has size exponential in n.

2.1   Polynomial Time Algorithms

We are interested in the distribution of the value $f(P)$ for each support $P \Subset \mathcal{Q}$. Since there are k^n possible supports, in general we cannot hope to do anything faster than that without making additional assumptions about f. Define $\tilde{f}(\mathcal{Q}, r)$ as the fraction (measured by weight) of supports of $\mathcal{Q}$ for which f gives a value smaller than or equal to r. We start with a simple example and then generalize. In this version, for simplicity, we assume general position and that k^n can be described by $O(1)$ words.

Smallest enclosing disk. Consider the problem where f measures the radius of the smallest enclosing disk of a support and let all weights be uniform so $w(q_{i,j}) = 1$ for all i and j.

Evaluating $\tilde{f}(\mathcal{Q}, r)$ in time polynomial in n and k is not completely trivial since there are k^n possible supports. However, we can make use of the fact that each smallest enclosing disk is in fact defined by a set of at most 3 points that lie on the boundary of the disk. For each support $P \Subset \mathcal{Q}$ we define $B_P \subseteq P$ to be this set of at most 3 points, which we call the *basis* for P. Bases have the property that $f(P) = f(B_P)$.

Now, to avoid having to test an exponential number of supports, we define a *potential basis* to be a set of at most 3 points in $\mathcal{Q}$ such that each point is from a different Q_i. Clearly, there are less than $(nk)^3$ possible potential bases, and each support $P \Subset \mathcal{Q}$ has one as its basis. Now, we only need to count for each potential basis the number of supports it represents. Counting the number of samples that have a certain basis is easy for the smallest enclosing circle. Given a basis B, we count for each indecisive point Q that does not contribute a point to B itself how many of its members lie inside the smallest enclosing circle of B, and then we multiply these numbers.

Now, for each potential basis B we have two values: the number of supports that have B as their basis, and the value $f(B)$. We can sort these $O((nk)^3)$ pairs on the value of f, and the result provides us with the required distribution. We spend $O(nk)$ time per potential basis for counting the points inside and $O(n)$ time for multiplying these values, so combined with $O((nk)^3)$ potential bases this gives $O((nk)^4)$ total time.

Theorem 1. *Let $\mathcal{Q}$ be a set of n sets of k points. In $O((nk)^4)$ time, we can compute a data structure of $O((nk)^3)$ size that can tell us in $O(\log(nk))$ time for any value r how many supports of $P \Subset \mathcal{Q}$ satisfy $f(P) \leq r$.*

LP-type problems. The approach described above also works for measures $f : \mathcal{Q} \to \mathbb{R}$ other than the smallest enclosing disk. In particular, it works for LP-type problems [13] that have constant combinatorial dimension. An *LP-type problem* provides a set of constraints H and a function $\omega : 2^H \to \mathbb{R}$ with the following two properties:

MONOTONICITY: For any $F \subseteq G \subseteq H$, $\omega(F) \leq \omega(G)$.
LOCALITY: For any $F \subseteq G \subseteq H$ with $\omega(F) = \omega(G)$ and an $h \in H$ such that $\omega(G \cup h) > \omega(G)$ implies that $\omega(F \cup h) > \omega(F)$.

A *basis* for an LP-type problem is a subset $B \subset H$ such that $\omega(B') < \omega(B)$ for all proper subsets B' of B. And we say that B is a basis for a subset $G \subseteq H$ if $B \subseteq G$, $\omega(B) = \omega(G)$ and B is a basis. A constraint $h \in H$ *violates* a basis B if $w(B \cup h) > w(B)$. The radius of the smallest enclosing ball is an LP-type problem (where the points are the constraints and $\omega(\cdot) = f(\cdot)$) as are linear programming and many other geometric problems. Let the maximum cardinality of any basis be the *combinatorial dimension* of a problem.

For our algorithm to run efficiently, we assume that our LP-type problem has available the following algorithmic primitive, which is often assumed for LP-type problems with constant combinatorial dimension [13]. For a subset $G \subset H$ where B is known to be the basis of G and a constraint $h \in H$, a *violation test* determines in $O(1)$ time if $\omega(B \cup h) > \omega(B)$; i.e., if h violates B. More specifically, given an efficient violation test, we can ensure a stronger algorithmic primitive. A *full violation test* is given a subset $G \subset H$ with known basis B and a constraint $h \in H$ and determines in $O(1)$ time if $\omega(B) < \omega(G \cup h)$. This follows because we can test in $O(1)$ time if $\omega(B) < \omega(B \cup h)$; MONOTONICITY implies that $\omega(B) < \omega(B \cup h)$ only if $\omega(B) < \omega(B \cup h) \leq \omega(G \cup h)$, and LOCALITY

implies that $\omega(B) = \omega(B \cup h)$ only if $\omega(B) = \omega(G) = \omega(G \cup h)$. Thus we can test if h violates G by considering just B and h, but if either MONOTONICITY or LOCALITY fail for our problem we cannot.

We now adapt our algorithm to LP-type problems where elements of each Q_i are potential constraints and the ranking function is f. When the combinatorial dimension is a constant β, we need to consider only $O((nk)^\beta)$ bases, which will describe all possible supports.

The full violation test implies that given a basis B, we can measure the sum of probabilities of all supports of $\mathcal{Q}$ that have B as their basis in $O(nk)$ time. For each indecisive point Q such that $B \cap Q = \emptyset$, we sum the probabilities of all elements of Q that do not violate B. The product of these probabilities times the product of the probabilities of the elements in the basis, gives the probability of B being the true basis. See Algorithm 2.1 where the indicator function applied $1(f(B \cup \{q_j\}) = f(B))$ returns 1 if q_j does not violate B and 0 otherwise. It runs in $O((nk)^{\beta+1})$ time.

Algorithm 2.1. Construct Probability Distribution for $f(\mathcal{Q})$

1: **for** all potential bases $B \subset P \Subset \mathcal{Q}$ **do**
2:     **for** $i = 1$ **to** n **do**
3:         **if** there is a j such that $q_{ij} \in B$ **then**
4:             Set $w_i = w(q_{ij})$.
5:         **else**
6:             Set $w_i = \sum_{j=1}^{k} w(q_{ij}) 1(f(B \cup \{q_j\}) = f(B))$.
7:     Store a point with value $f(B)$ and weight $(1/k^n) \prod_i w_i$.

As with the special case of smallest enclosing disk, we can create a distribution over the values of f given an indecisive point set $\mathcal{Q}$. For each basis B we calculate $\mu(B)$, the summed probability of all supports that have basis B, and $f(B)$. We can then sort these pairs according to the value as f again. For any query value r, we can retrieve $\tilde{f}(\mathcal{Q}, r)$ in $O(\log(nk))$ time and it takes $O(n)$ time to describe (because of its long length).

Theorem 2. *Given a set $\mathcal{Q}$ of n indecisive point sets of size k each, and given an LP-type problem $f : \mathcal{Q} \to \mathbb{R}$ with combinatorial dimension β, we can create the distribution of f over $\mathcal{Q}$ in $O((nk)^{\beta+1})$ time. The size of the distribution is $O(n(nk)^\beta)$.*

If we assume general position of $\mathcal{Q}$ relative to f, then we can often slightly improve the runtime needed to calculate $\mu(B)$ using range searching data structures. However, to deal with degeneracies, we may need to spend $O(nk)$ time per basis, regardless.

If we are content with an approximation of the distribution rather than an exact representation, then it is often possible to drastically reduce the storage and runtime. This requires the definition of ε-*quantizations* [11], which is delayed until Section 3 where it is discussed in more detail.

Measures that fit in this framework for points in $\mathbb{R}^d$ include smallest enclosing axis-aligned rectangle (measured either by area or perimeter) ($\beta = 2d$), smallest enclosing ball in the L_1, L_2, or L_∞ metric ($\beta = d+1$), directional width of a set of points ($\beta = 2$), and, after dualizing, linear programming ($\beta = d$).

2.2   Hardness Results

In this section, we examine some extent measures that do not fit in the above framework. First, diameter does not satisfy the LOCALITY property, and hence we cannot efficiently perform the full violation test. We show that a decision variant of diameter is #P-Hard, even in the plane, and thus (under the assumption that #P $\neq$ P), there is no polynomial time solution. Second, the area of the convex hull does not have a constant combinatorial dimension, thus we can show the resulting distribution may have exponential size.

Diameter. The *diameter* of a set of points in the plane is the largest distance between any two points. We will show that the counting problem of computing $\tilde{f}(\mathcal{Q}, r)$ is #P-hard when f denotes the diameter.

Problem 1. PLANAR-DIAM: Given a parameter d and a set $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ of n sets, each consisting of k points in the plane, how many supports $P \Subset \mathcal{Q}$ have $f(P) \leq d$?

We will now prove that Problem 1 is #P-hard. Our proof has three steps. We first show a special version of #2SAT has a polynomial reduction from Monotone #2SAT, which is #P-complete [15]. Then, given an instance of this special version of #2SAT, we construct a graph with weighted edges on which the diameter problem is equivalent to this #2SAT instance. Finally, we show the graph can be embedded as a straight-line graph in the plane as an instance of PLANAR-DIAM.

Let 3CLAUSE-#2SAT be the problem of counting the number of solutions to a 2SAT formula, where each variable occurs in at most three clauses, and each variable is either in exactly one clause or is negated in exactly one clause. Thus, each distinct literal appears in at most two clauses.

Lemma 1. *Monotone #2SAT has a polynomial reduction to 3CLAUSE-#2SAT.*

We convert this problem into a graph problem by, for each variable x_i, creating a set $Q_i = \{p_i^+, p_i^-\}$ of two points. Let $\mathcal{Q} = \bigcup_i Q_i$. Truth assignments of variables correspond to a support as follows. If x_i is set TRUE, then the support includes p_i^+, otherwise the support includes p_i^-. We define a distance function f between points, so that the distance is greater than d (long) if the corresponding literals are in a clause, and less than d (short) otherwise. If we consider the graph formed by only long edges, we make two observations. First, the maximum degree is 2, since each literal is in at most two clauses. Second, there are no cycles since a literal is only in two clauses if in one clause the other variable is negated, and negated variables are in only one clause. These two properties imply we can use the following construction to show that the PLANAR-DIAM problem is as hard as counting Monotone #2SAT solutions, which is #P-complete.

Lemma 2. *An instance of PLANAR-DIAM reduced from 3CLAUSE-#2SAT can be embedded so $\mathcal{Q} \subset \mathbb{R}^2$.*

Proof. Consider an instance of 3CLAUSE-#2SAT where there are n variables, and thus the corresponding graph has n sets $\{Q_i\}_{i=1}^n$. We construct a sequence Γ of $n' \in [2n, 4n]$ points. It contains all points from $\mathcal{Q}$ and a set of at most as many dummy points. First organize a sequence Γ' so if two points q and p have a long edge, then they are consecutive. Now for any pair of consecutive points in Γ' which do not have a long edge, insert a dummy point between them to form the sequence Γ. Also place a dummy point at the end of Γ.

We place all points on a circle C of diameter $d/\cos(\pi/n')$, see Figure 1. We first place all points on a semicircle of C according to the order of Γ, so each consecutive points are π/n' radians apart. Then for every other point (i.e. the points with an even index in the ordering Γ) we replace it with its antipodal point on C, so no two points are within $2\pi/n'$ radians of each other. Finally we remove all dummy points. This completes the embedding of $\mathcal{Q}$, we now need to show that only points with long edges are further than d apart.

We can now argue that only vertices which were consecutive in Γ are further than d apart, the remainder are closer than d. Consider a vertex p and a circle C_p of radius d centered at p. Let p' be the antipodal point of p on C. C_p intersects C at two points, at $2\pi/n'$ radians in either direction from p'. Thus only points within $2\pi/n'$ radians of p' are further than a distance d from p. This set includes only those points which are adjacent to p in Γ, which can only include points which should have a long edge, by construction. $\square$

Fig. 1. Embedded points are solid, at center of circles of radius d. Dummy points hollow. Long edges are drawn.

Combining Lemmas 1 and 2:

Theorem 3. *PLANAR-DIAM is #P-hard.*

Convex hull. Our LP-type framework also does not work for any properties of the convex hull (e.g. area or perimeter) because it does not have constant combinatorial dimension; a basis could have size n. In fact, the complexity of the distribution describing the convex hull may be $\Omega(k^n)$, since if all points in $\mathcal{Q}$ lie on or near a circle, then every support $P \Subset \mathcal{Q}$ may be its own basis of size n, and have a different value $f(P)$.
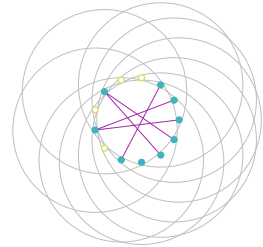
3   Approximate Computations on Uncertain Points

Perhaps the most general way to model an imprecise point is by providing a full probability distribution over $\mathbb{R}^d$; all popular simpler models can be seen as special cases of this. However, probability distributions can be difficult to handle, specifically, it is often impossible to do exact computations on them. In this section we show how the results from Section 2 can be used to approximately

answer questions about uncertain points by representing each distribution by a discrete point set, resulting in a set of indecisive points.

In this section, we are given a set $\mathcal{X} = \{X_1, X_2, X_3, \ldots, X_n\}$ of n independent random variables over the universe $\mathbb{R}^d$, together with a set $\mathcal{M} = \{\mu_1, \mu_2, \mu_3, \ldots, \mu_n\}$ of n probability distributions that govern the variables, that is, $X_i \sim \mu_i$. We call a set of points $P = \{p_1, p_2, p_3, \ldots, p_n\}$ a *support* of $\mathcal{X}$, and because of the independence we have probability $Pr[\mathcal{X} = P] = \prod_i Pr[X_i = p_i]$.

As before, we are interested in functions f that measure something about a point set. We now define $\hat{f}(\mathcal{X}, r)$ as the probability that a support P drawn from $\mathcal{M}$ satisfies $f(P) \leq r$. In most cases, we cannot evaluate this function exactly, but previous work [11] describes a Monte Carlo algorithm for approximating $\hat{f}(\mathcal{X}, r)$. Here we show how to make this approximate construction deterministic.

To approximate $\hat{f}(\mathcal{X}, r)$, we construct an ε-*quantization* [11]. Let $g : \mathbb{R} \to \mathbb{R}^+$ be a distribution so $\int_{\mathbb{R}} g(x)\ dx = 1$, and let G be its integral so $G(t) = \int_{-\infty}^{t} g(x)\ dx$. Then $G : \mathbb{R} \to [0, 1]$ is a cumulative density function. Also, let R be a set of points in $\mathbb{R}$, that is, a set of values. Then R induces a function H_R where $H_R(v) = \frac{|\{r \in R \mid r \leq v\}|}{|R|}$, that is, $H_R(v)$ describes the fraction of points in R with value at most v. We say that R is an ε-quantization of g if H_R approximates G within ε, that is, for all values v we have $|H_R(v) - G(v)| \leq \varepsilon$. Note that we can apply techniques from [11] to deterministically reduce the exact distributions created in Section 2 to ε-quantizations of size $O(1/\varepsilon)$.

The main strategy will be to replace each distribution μ_i by a discrete point set Q_i, such that the uniform distribution over Q_i is "not too far" from μ_i (Q_i is not the most obvious ε-sample of μ_i). Then we apply the algorithms from Section 2 to the resulting set of point sets. Finally, we argue that the result is in fact an ε-quantization of the distribution we are interested in, and we show how to simplify the output in order to decrease the space complexity for the data structure, without increasing the approximation factor too much.

Range spaces and ε-samples. Before we describe the algorithms, we need to formally define range spaces and ε-samples. Given a set of elements Y let $\mathcal{A} \subset 2^Y$ be a family of subsets of Y. For instance, if Y is a point set, $\mathcal{A}$ could be all subsets defined by intersection with a ball. A pair $T = (Y, \mathcal{A})$ is called a *range space*.

We say a set of ranges $\mathcal{A}$ *shatters* a set Y if for every subset $Y' \subseteq Y$ there exists some $A \in \mathcal{A}$ such that $Y' = Y \cap A$. The size of the largest subset $Y' \subseteq Y$ that $\mathcal{A}$ shatters is the *VC-dimension* [17] of $T = (Y, \mathcal{A})$, denoted ν_T.

Let $\mu : Y \to \mathbb{R}^+$ be a measure on Y. For discrete sets Y μ is cardinality, for continuous sets Y μ is a Lebesgue measure. An ε-*sample* (often referred to by the generic term ε-approximation) of a range space $T = (Y, \mathcal{A})$ is a subset $S \subset Y$ such that $\forall A \in \mathcal{A} : \left| \frac{\mu(A \cap S)}{\mu(S)} - \frac{\mu(A \cap Y)}{\mu(Y)} \right| \leq \varepsilon$. A random subset $S \subset Y$ of size $O((1/\varepsilon^2)(\nu_T + \log(1/\delta)))$ is an ε-sample with probability at least $1 - \delta$ [17,10]. For a range space $T = (Y, \mathcal{A})$ with Y discrete and $\mu(Y) = n$, there are also deterministic algorithms to generate ε-samples of size $O((\nu_T/\varepsilon^2) \log(1/\varepsilon))$ in time $O(\nu_T^{3\nu_T} n((1/\varepsilon^2) \log(\nu_T/\varepsilon))^{\nu_T})$ [4]. Or when the ranges $\mathcal{A}$ are defined by the

intersection of k oriented slabs (i.e. axis-aligned rectangles with $k = d$), then an ε-sample of size $O((k/\varepsilon) \log^{2k}(1/\varepsilon))$ can be deterministically constructed in $O((n/\varepsilon^3) \log^{6k}(1/\varepsilon))$ time [12].

In the continuous setting, we can think of each point $y \in Y$ as representing $\mu(y)$ points, and for simplicity represent a weighted range space as $(\mu, \mathcal{A})$ when the domain of the function μ is implicitly known to be Y (often $\mathbb{R}^2$ or $\mathbb{R}^d$). Phillips [12] studies ε-samples for such weighted range spaces with ranges defined as intersection of k intervals; the full version extends this.

General approach (KEY CONCEPTS). Given a distribution $\mu_i : \mathbb{R}^2 \to \mathbb{R}^+$ describing uncertain point X_i and a function f of bounded combinatorial dimension β defined on a support of $\mathcal{X}$, we can describe a straightforward range space $T_i = (\mu_i, \mathcal{A}_f)$, where $\mathcal{A}_f$ is the set of ranges corresponding to the bases of f (e.g., when f measures the radius of the smallest enclosing ball, $\mathcal{A}_f$ would be the set of all balls). More formally, $\mathcal{A}_f$ is the set of subsets of $\mathbb{R}^d$ defined as follows: for every set of β points which define a basis B for f, $\mathcal{A}_f$ contains a range A that contains all points p such that $f(B) = f(B \cup \{p\})$. However, taking ε-samples from each T_i is *not* sufficient to create sets Q_i such that $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ so for all r we have $|\tilde{f}(\mathcal{Q}, r) - \hat{f}(\mathcal{X}, r)| \leq \varepsilon$.

$\hat{f}(\mathcal{X}, r)$ is a complicated joint probability depending on the n distributions and f, and the n straightforward ε-samples do not contain enough information to decompose this joint probability. The required ε-sample of each μ_i should model μ_i in relation to f and any instantiated point q_i representing μ_j for $i \neq j$. The following crucial definition allows for the range space to depend on any $n-1$ points, including the possible locations of each uncertain point.

Let $\mathcal{A}_{f,n}$ describe a family of Lebesgue-measurable sets defined by $n-1$ points $Z \subset \mathbb{R}^d$ and a value w. Specifically, $A(Z, w) \in \mathcal{A}_{f,n}$ is the set of points $\{p \in \mathbb{R}^d \mid f(Z \cup p) \leq w\}$. We describe examples of $\mathcal{A}_{f,n}$ in detail shortly, but first we state the key theorem using this definition. Its proof, delayed until after examples of $\mathcal{A}_{f,n}$, will make clear how $(\mu_i, \mathcal{A}_{f,n})$ exactly encapsulates the right guarantees to approximate $\hat{f}(\mathcal{X}, r)$, and thus why $(\mu_i, \mathcal{A}_f)$ does not.

Theorem 4. *Let* $\mathcal{X} = \{X_1, \ldots, X_n\}$ *be a set of uncertain points where each* $X_i \sim \mu_i$. *For a function* f, *let* Q_i *be an* ε'-*sample of* $(\mu_i, \mathcal{A}_{f,n})$ *and let* $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$. *Then for any* r, $\left| \hat{f}(\mathcal{X}, r) - \tilde{f}(\mathcal{Q}, r) \right| \leq \varepsilon' n$.

Smallest axis-aligned bounding box by perimeter. Given a set of points $P \subset \mathbb{R}^2$, let $f(P)$ represent the perimeter of the smallest axis-aligned box that contains P. Let each μ_i be a bivariate normal distribution with constant variance. Solving $f(P)$ is an LP-type problem with combinatorial dimension $\beta = 4$, and as such, we can describe the basis B of a set P as the points with minimum and maximum x- and y-coordinates. Given any additional point p, the perimeter of size ρ can only be increased to a value w by expanding the range of x-coordinates, y-coordinates, or both. As such, the region of $\mathbb{R}^2$ described by a range $A(P, w) \in \mathcal{A}_{f,n}$ is defined with respect to the bounding box of P from an edge increasing the x-width or y-width by $(w - \rho)/2$, or from a corner extending so the sum of the x and y deviation is $(w - \rho)/2$. See Figure 2(Left).

Since any such shape defining a range $A(P,w) \in \mathcal{A}_{f,n}$ can be described as the intersection of $k = 4$ slabs along fixed axis (at $0°$, $45°$, $90°$, and $135°$), we can construct an (ε/n)-sample Q_i of $(\mu_i, \mathcal{A}_{f,n})$ of size $k = O((n/\varepsilon)\log^8(n/\varepsilon))$ in $O((n^6/\varepsilon^6)\log^{27}(n/\varepsilon))$ time [12]. From Theorem 4, it follows that for $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ and any r we have $\left|\hat{f}(\mathcal{X}, r) - \tilde{f}(\mathcal{Q}, r)\right| \le \varepsilon$.

We can then apply Theorem 2 to build an ε-quantization of $f(\mathcal{X})$ in $O((nk)^5) = O(((n^2/\varepsilon)\log^8(n/\varepsilon))^5) = O((n^{10}/\varepsilon^5)\log^{40}(n/\varepsilon))$ time. The size can be reduced to $O(1/\varepsilon)$ within that time bound.

Corollary 1. *Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a set of indecisive points where each $X_i \sim \mu_i$ is bivariate normal with constant variance. Let f measure the perimeter of the smallest enclosing axis-aligned bounding box. We can create an ε-quantization of $f(\mathcal{X})$ in $O((n^{10}/\varepsilon^5)\log^{40}(n/\varepsilon))$ time of size $O(1/\varepsilon)$.*

Smallest enclosing disk. Given a set of points $P \subset \mathbb{R}^2$, let $f(P)$ represent the radius of the smallest enclosing disk of P. Let each μ_i be a bivariate normal distribution with constant variance. Solving $f(P)$ is an LP-type problem with combinatorial dimension $\beta = 3$, and the basis B of P generically consists of either 3 points which lie on the boundary of the smallest enclosing disk, or 2 points which are antipodal on the smallest enclosing disk. However, given an additional point $p \in \mathbb{R}^2$, the new basis B_p is either B or it is p along with 1 or 2 points which lie on the convex hull of P.

We can start by examining all pairs of points $p_i, p_j \in P$ and the two disks of radius w whose boundary circles pass through them. If one such disk $D_{i,j}$ contains P, then $D_{i,j} \subset A(P,w) \in \mathcal{A}_{f,|P|+1}$. For this to hold, p_i and p_j must lie on the convex hull of P and no point that lies between them on the convex hull can contribute to such a disk. Thus there are $O(n)$ such disks. We also need to examine the disks created where p and one other point $p_i \in P$ are antipodal. The boundary of the union of all such disks which contain P is described by part of a circle of radius $2w$ centered at some $p_i \in P$. Again, for such a disk B_i to describe a part of the boundary of $A(P,w)$, the point p_i must lie on the convex hull of P. The circular arc defining this boundary will only connect two disks $D_{i,j}$ and $D_{k,i}$ because it will intersect with the boundary of B_j and B_k within these disks, respectively. An example of $A(P,w)$ is shown in Figure 2(Middle).
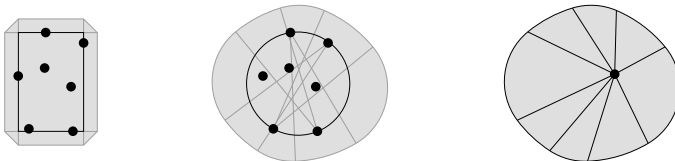

Fig. 2. Left: A shape from $\mathcal{A}_{f,n}$ for axis-aligned bounding box, measured by perimeter. Middle: A shape from $\mathcal{A}_{f,n}$ for smallest enclosing ball using the L_2 metric. The curves are circular arcs of two different radii. Right: The same shape divided into wedges from $\mathcal{W}_{f,n}$.

Unfortunately, the range space $(\mathbb{R}^2, \mathcal{A}_{f,n})$ has VC-dimension $O(n)$; it has $O(n)$ circular boundary arcs. So, creating an ε-sample of $T_i = (\mu_i, \mathcal{A}_{f,n})$ would take time exponential in n. However, we can decompose any range $A(P, w) \in \mathcal{A}_{f,n}$ into at most $2n$ "wedges." We choose one point y inside the convex hull of P. For each circular arc on the boundary of $A(P, w)$ we create a wedge by coning that boundary arc to y. Let $\mathcal{W}_f$ describe all wedge shaped ranges. Then $S = (\mathbb{R}^2, \mathcal{W}_f)$ has VC-dimension ν_S at most 9 since it is the intersection of 3 ranges (two half-spaces and one disk) that can each have VC-dimension 3. We can then create Q_i, an $(\varepsilon/2n^2)$-sample of $S_i = (\mu_i, \mathcal{W}_f)$, of size $k = O((n^4/\varepsilon^2)\log(n/\varepsilon))$ in $O((n^2/\varepsilon)^{5+2\cdot9}\log^{2+9}(n/\varepsilon)) = O((n^{46}/\varepsilon^{23})\log^{11}(n/\varepsilon))$ time (proof in full version). It follows that Q_i is an (ε/n)-sample of $T_i = (\mu_i, \mathcal{A}_{f,n})$, since any range $A(Z, w) \in \mathcal{A}_{f,n}$ can be decomposed into at most $2n$ wedges, each of which has counting error at most $\varepsilon/2n$, thus the total counting error is at most ε.

Invoking Theorem 4, it follows that $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$, for any r we have $\left| \hat{f}(\mathcal{X}, r) - \tilde{f}(\mathcal{Q}, r) \right| \leq \varepsilon$. We can then apply Theorem 2 to build an ε-quantization of $f(\mathcal{X})$ in $O((nk)^4) = O((n^{20}/\varepsilon^8)\log^4(n/\varepsilon))$ time. This is dominated by the time for creating the n (ε/n^2)-samples, even though we only need to build one and then translate and scale to the rest. Again, the size can be reduced to $O(1/\varepsilon)$ within that time bound.

Corollary 2. *Let $\mathcal{X} = \{X_1, \ldots, X_n\}$ be a set of indecisive points where each $X_i \sim \mu_i$ is bivariate normal with constant variance. Let f measure the radius of the smallest enclosing disk. We can create an ε-quantization of $f(\mathcal{X})$ in $O((n^{46}/\varepsilon^{23})\log^{11}(n/\varepsilon))$ time of size $O(1/\varepsilon)$.*

Now that we have seen two concrete examples, we prove Theorem 4.

Proof of Theorem 4. When each X_i is drawn from a distribution μ_i, then we can write $\hat{f}(\mathcal{X}, r)$ as the probability that $f(\mathcal{X}) \leq r$ as follows. Let $1(\cdot)$ be the indicator function, i.e., it is 1 when the condition is true and 0 otherwise.

$$\hat{f}(\mathcal{X}, r) = \int_{p_1} \mu_1(p_1) \ldots \int_{p_n} \mu_n(p_n) 1(f(\{p_1, p_2, \ldots, p_n\}) \leq r) \, dp_n dp_{n-1} \ldots dp_1$$

Consider the inner most integral $\int_{p_n} \mu_n(p_n) 1(f(\{p_1, p_2, \ldots, p_n\}) \leq r) \, dp_n$, where $\{p_1, p_2 \ldots, p_{n-1}\}$ are fixed. The $1(\cdot) = 1$ when $f(\{p_1, p_2, \ldots, p_{n-1}, p_n\}) \leq r$, and hence p_n is contained in a shape $A(\{p_1, \ldots, p_{n-1}\}, r) \in \mathcal{A}_{f,n}$. Thus if we have an ε'-sample Q_n for $(\mu_n, \mathcal{A}_{f,n})$, then we can guarantee that

$$\int_{p_n} \mu_n(p_n) 1(f(\{p_1, \ldots, p_n\}) \leq r) \, dp_n \leq \frac{1}{|Q_n|} \sum_{p_n \in Q_n} 1(f(\{p_1, \ldots, p_n\}) \leq r) + \varepsilon'.$$

We can then move the ε' outside and change the order of the integrals to write:

$$\hat{f}(\mathcal{X}, r) \leq \frac{1}{|Q_n|} \sum_{p_n \in Q_n} \left(\int_{p_1} \mu_1(p_1) .. \int_{p_{n-1}} \mu_{n-1}(p_{n-1}) 1(f(\{p_1, .., p_n\}) \leq r) dp_{n-1}..dp_1 \right) + \varepsilon'.$$

Repeating this procedure n times, where $\mathcal{Q} = \bigcup_i Q_i$, we get:

$$\hat{f}(\mathcal{X}, r) \leq \left(\prod_{i=1}^{n} \frac{1}{|Q_i|} \right) \sum_{p_1 \in Q_1} \cdots \sum_{p_n \in Q_n} 1(f(\{p_1, \ldots, p_n\}) \leq r) + \varepsilon' n. = \tilde{f}(\mathcal{Q}, r) + \varepsilon' n.$$

Similarly we can achieve a symmetric lower bound for $\hat{f}(\mathcal{X}, r)$.   □

References

1. Agrawal, P., Benjelloun, O., Sarma, A.D., Hayworth, C., Nabar, S., Sugihara, T., Widom, J.: Trio: A system for data, uncertainty, and lineage. In: PODS (2006)
2. Bandyopadhyay, D., Snoeyink, J.: Almost-Delaunay simplices: Nearest neighbor relations for imprecise points. In: SODA (2004)
3. Bi, J., Zhang, T.: Support vector classification with input data uncertainty. In: NIPS (2004)
4. Chazelle, B., Matousek, J.: On linear-time deterministic algorithms for optimization problems in fixed dimensions. Journal of Algorithms 21, 579–597 (1996)
5. Cormode, G., Garafalakis, M.: Histograms and wavelets of probabilitic data. In: ICDE (2009)
6. Cormode, G., McGregor, A.: Approximation algorithms for clustering uncertain data. In: PODS (2008)
7. Dalvi, N., Suciu, D.: Efficient query evaluation on probabilitic databases. The VLDB Journal 16, 523–544 (2007)
8. Guibas, L.J., Salesin, D., Stolfi, J.: Epsilon geometry: building robust algorithms from imprecise computations. In: SoCG (1989)
9. Held, M., Mitchell, J.S.B.: Triangulating input-constrained planar point sets. Information Processing Letters 109(1) (2008)
10. Li, Y., Long, P.M., Srinivasan, A.: Improved bounds on the samples complexity of learning. Journal of Computer and System Sciences 62, 516–527 (2001)
11. Löffler, M., Phillips, J.M.: Shape fitting on point sets with probability distributions. In: Fiat, A., Sanders, P. (eds.) ESA 2009. LNCS, vol. 5757, pp. 313–324. Springer, Heidelberg (2009)
12. Phillips, J.M.: Algorithms for ϵ-approximations of terrains. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 447–458. Springer, Heidelberg (2008)
13. Sharir, M., Welzl, E.: A combinatorial bound for linear programming and related problems. In: STACS (1992)
14. Tao, Y., Cheng, R., Xiao, X., Ngai, W.K., Kao, B., Prabhakar, S.: Indexing multi-dimensional uncertain data with arbitrary probability density functions. In: VLDB (2005)
15. Valiant, L.G.: The complexity of enumeration and reliability problems. SIAM Journal on Computing 8, 410–421 (1979)
16. van Kreveld, M., Löffler, M.: Largest bounding box, smallest diameter, and related problems on imprecise points. Computational Geometry: Theory and Applications 43, 419–433 (2010)
17. Vapnik, V., Chervonenkis, A.: On the uniform convergence of relative frequencies of events to their probabilities. Theory of Probability and its Applications 16, 264–280 (1971)
18. Zou, Y., Chakrabarty, K.: Uncertainty-aware and coverage-oriented deployment of sensor networks. Journal of Parallel and Distributed Computing (2004)

Closest Pair and the Post Office Problem for Stochastic Points

Pegah Kamousi[1], Timothy M. Chan[2], and Subhash Suri[1]

[1] Computer Science, UC Santa Barbara, CA 93106
[2] Computer Science, University of Waterloo, Ontario N2L3G1

Abstract. Given a (master) set M of n points in d-dimensional Euclidean space, consider drawing a random subset that includes each point $m_i \in M$ with an independent probability p_i. How difficult is it to compute elementary statistics about the closest pair of points in such a subset? For instance, what is the *probability* that the distance between the closest pair of points in the random subset is no more than ℓ, for a given value ℓ? Or, can we preprocess the master set M such that given a query point q, we can efficiently estimate the *expected* distance from q to its nearest neighbor in the random subset? We obtain hardness results and approximation algorithms for stochastic problems of this kind.

1 Introduction

Many years ago, Knuth [12] posed the now classic *post-office* problem, namely, given a set of points in the plane, find the one closest to a query point q. The problem, which is fundamental and arises as a basic building block of numerous computational geometry algorithms and data structures [7], is reasonably well-understood in small dimensions. In this paper, we consider a *stochastic* version of the problem in which each post office may be *closed* with certain probability. In other words, a given set of points M in d dimensions includes the locations of all the post offices but on a typical day each post office $m_i \in M$ is only open with an independent probability p_i. The set of points M together with their probabilities form a probability distribution where each point m_i is included in a random subset of points with probability p_i. Thus, given a query points q, we now must talk about the *expected* distance from q to its closest neighbor in M. Similarly, instead of simply computing the closest pair of points in a set, we must ask: how *likely* is it that the closest pair of points are no more than ℓ apart?

In this paper, we study the complexity of such elementary proximity problems and establish both upper and lower bounds. In particular, we have a finite set of points M in a d-dimensional Euclidean space, which constitutes our *master* set of points, and hence the mnemonic M. Each member m_i of M has probability p_i of being present and probability $1 - p_i$ of being absent. (Following the post-office analogy, the ith post office is open with probability p_i and closed otherwise.) These probabilities are independent, but otherwise arbitrary, and lead to a sample space of 2^n possible subsets, where a sample subset includes the ith point with independent probability p_i. We achieve the following results.

1. It is NP-Hard to compute the probability that the closest pair of points have distance at most a value ℓ, even for dimension 2 under the L_∞ norm.

2. In the *linearly-separable* and *bichromatic* planar case, the closest pair probability can be computed in polynomial time under the L_∞ norm.

3. Without the linear separability, even the bichromatic case is NP-Hard under the L_2 or L_∞ norm.

4. Even with linear separability and L_∞ norm, the bichromatic case becomes NP-Hard in dimension $d \geq 3$.

5. We give a linear-space data structure with $O(\log n)$ query time to compute the expected distance of a given query point to its $(1+\varepsilon)$-approximate nearest neighbor when the dimension d is a constant.

Related Work

A number of researchers have recently begun to explore geometric computing over *probabilistic* data [1,2,14,19]. These studies are fundamentally different from the classical geometric probability that deals with properties of random point sets drawn from some infinite sets, such as points in unit square [11]. Instead, the recent work in computational geometry is concerned with worst-case set of objects and worst-case probabilities or behavior. In particular, the recent work of Agarwal [1,2] deals with the database problem of skyline computation using a multiple-universe model. The work of van Kreveld and Löffler [14,19] deals with objects whose locations are randomly distributed in some *uncertainty* regions. Unlike these results, our focus is not on the locational uncertainty but rather on the discrete probabilities which which each point may appear.

2   The Stochastic Closest Pair Problem

We begin with the complexity of the stochastic closest pair problem, which asks for the probability that the closest pair has distance at most a given bound ℓ. We will show that this basic problem is intractable, via reduction from the problem of *counting vertex covers* in planar unit disk graphs (UDGs). In order to show that even the *bichromatic* closest pair problem is hard, we also prove that a corresponding vertex cover counting problem is hard for a bichromatic version of the unit disk graphs.

2.1   Counting Vertex Covers in Unit Disk Graphs

The problem of counting the vertex covers in a graph [13,16,18] is the following. Given a graph $G = (V, E)$, how many subsets $S \subseteq V$ constitute a vertex cover, where S is a vertex cover of G if for each $uv \in E$, either $u \in S$ or $v \in S$. (We note that we are counting vertex covers and *not* minimum vertex covers.) This problem is known to be #P-hard even for planar bipartite graphs with maximum degree 4 [16]. Our reduction will use *unit disk graphs* (UDG), which are the

intersection graphs of n equal-sized circles in the plane: each node corresponds to a circle, and there is an edge between two nodes if the corresponding circles intersect. We will first prove that the minimum vertex cover problem is hard for planar unit disk graphs of maximum degree 3 (3-planar UDG), using which we then prove that counting the vertex covers is also hard for 3-planar UDGs. We will then extend this result to a special class of planar unit disk graphs, which we call the *rectilinear unit disk graphs* (to be defined later). The first step in the proof is the following well-known lemma of Valiant [17].

Lemma 1. *A planar graph $G = (V, E)$ with maximum degree 4 can be embedded in the plane using $O(|V|)$ area in such a way that its nodes are at integer coordinates and its edges are drawn so that they are made up of line segments of the form $x = i$ or $y = j$, for integers i and j.*

Throughout this section, we assume that the disks defining unit disk graphs have radius 1. We use the notation $d(P, Q)$ for the L_2 distance between two sets P and Q. (When using the L_∞ or L_1 norms, we will use $d_\infty(P, Q)$ and $d_1(P, Q)$.)

Lemma 2. *The minimum vertex cover problem is NP-hard for planar unit disk graphs of maximum degree 3.*

Proof. The reduction is from minimum vertex cover for planar graphs with maximum degree three [8]. Let $G = (V, E)$ be an instance of such graphs. We embed G in the plane according to Lemma 1. Let $G_r = (V_r, E_r)$ be the graph obtained from this embedding by replacing each edge $uv \in E$ by a path consisting of even number $2k_{uv}$ of intermediate nodes, each at distance ≤ 1 from the previous one, in such a way that the resulting graph is a unit disk graph. The value k_{uv} depends on the L_1 distance between u and v in G_r. (To avoid unwanted edges between the intermediate nodes on different edges, some scaling may be required, but it is easy to see that this is always possible.)

It is not hard to see that G has a vertex cover of size $\leq k$ if and only if G_r has a vertex cover of size $\leq k + \sum_{uv \in E} k_{uv}$. But G_r is a 3-planar UDG, which shows that the 3-planar UDG vertex cover is NP-hard.   □

The graph G_r above is a unit disk graph with maximum degree 3, which can be embedded in the plane such that the length of each edge is $\geq 2/3$, and the edges lie on the integer grid lines. This is possible by placing the $2k_{uv}$ intermediate nodes on integer grid points if $d_1(u, v)$ is an odd number, or uniformly distributing $d_1(u, v)$ intermediate points on uv in case $d_1(u, v)$ is even (in the extreme case where $d_1(u, v) = 2$, we place 2 nodes on uv and obtain 3 edges of length $2/3$). Let us call a unit disk graph that admits such embedding a *rectilinear unit disk graph*. We have the following corollary of Lemma 2.

Corollary 1. *The minimum vertex cover problem is NP-hard for rectilinear unit disk graphs.*

Theorem 1. *It is NP-hard to count the vertex covers in a rectilinear unit disk graph. Moreover, the number of vertex covers cannot be approximated to any multiplicative factor in polynomial time assuming $P \neq NP$.*

Proof. We will prove the inapproximability, which shows the hardness as well. Let $G = (V, E)$ be a rectilinear UDG. Suppose we have an α-approximation algorithm for counting the vertex covers in G, i.e., if $c(G)$ is the number of vertex covers, the algorithm outputs a value $\tilde{c}$ such that $(1/\alpha)c(G) \leq \tilde{c} \leq \alpha c(G)$.

Let G_p be the stochastic graph obtained from G by assigning the probability $p = 1/(2^n \alpha^2)$ of being present to each of the nodes in G. Since this probability is the same for all the nodes, an α-approximation algorithm for counting the vertex covers in G readily provides an α-approximation algorithm for computing the probability that a random subset of nodes in G_p is a vertex cover. Let $\Pr(G_p)$ denote this probability, and $\tilde{r}$ be an α-approximation to $\Pr(G_p)$.

The key observation is that $\tilde{r} \geq p^k$ if and only if G has a vertex cover of size k or less. To see this, suppose G has a vertex cover C of size k or less. Then the probability that a random subset of nodes of G_p is a vertex cover is at least p^k, i.e., the probability that all the nodes in C are present. In this case, $\tilde{r} \geq p^k/\alpha$. Otherwise, at least $k+1$ nodes must be present to constitute a vertex cover, which happens with probability at most $2^{|V|}p^{k+1} < p^k/\alpha^2$. In this case $\tilde{r} < p^k/\alpha$. Corollary 1, however, shows that the minimum vertex cover problem is hard for G, and therefore $\Pr(G_p)$ cannot be approximated to any factor α in polynomial time assuming $P \neq NP$. This completes the proof. $\square$

2.2 Bichromatic Unit Disk Graphs

We introduce the class of *bichromatic unit disk graphs* as the graphs defined over a set of points in the plane, each colored as blue or red, with an edge between a red and a blue pair if and only if their distance is ≤ 1. (We do not put edges between nodes of the same color regardless of the distance between them.) We will show that counting the vertex covers is NP-hard for bichromatic UDGs. We will need the following lemma from [15]. Remember that a Vandermonde matrix M is in the form $\mathbf{M_{ij}} = (\mu_i^j, 0 \leq i, j \leq n)$ (or its transpose) for a given sequence of numbers $\mu_0, \ldots, \mu_n$. (See [9, §5.1].)

Lemma 3. *Suppose we have v_i and b_i, $i = 0, \ldots, n$, related by the equation $v_i = \sum_{j=0}^{n} a_{ij} b_i, j = 0, \ldots, n$. Further suppose that the matrix of the coefficients (a_{ij}) is Vandermonde, with parameters $\mu_0, \ldots, \mu_n$ which are distinct. Then given the values $v_0, \ldots, v_n$, we can obtain the values $b_0, \ldots, b_n$ in time polynomial in n.*

Now consider the gadget $\mathcal{H}$ in Fig. 1 (a), which consists of l paths between u and v, for a given l. Let $G = (V, E)$ be an instance of a rectilinear UDG. Let $G' = (V', E')$ be the graph obtained from G by replacing each edge $uv \in E$ with the graph $\mathcal{H}$. We color u, v and the b_i's red, and the remaining nodes blue.

Lemma 4. *The graph G' is a bichromatic unit disk graph.*

Proof. First we embed the original graph G in the plane with edges of length $\geq 2/3$ lying on integer grid lines (this is possible by definition of G). We will then scale the grid by a factor of 3.5, so that for all $uv \in E$, $2 < d_2(u, v) \leq 3.5$. Next we embed the graph $\mathcal{H}$ on each edge $uv \in E$ such that $d(a_i, b_i) = d(b_i, c_i) = 1$,

while $d(u, a_i) = d(v, c_i) \leq 1$, for $i = 1, \ldots, l$. This is always possible since $2 < d_2(u, v) \leq 3.5$.

It is easy to see that the distance between two nodes of different colors from two different rows is always greater than 1, and therefore there won't be any edges between two different rows. Moreover, there should not be any connections between two different gadgets placed on two edges. Given that we can scale the initial embedding as desired, the only case to worry about is for two orthogonal edges. Considering Fig. 1 (b), it is easy to see that each two nodes of different colors in two different gadgets are at distance > 1. This completes the proof. $\square$
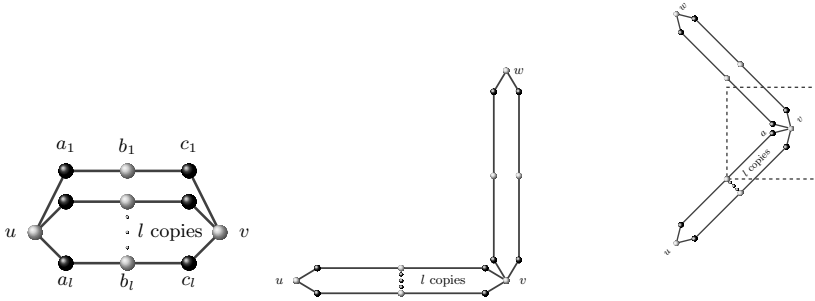
Fig. 1. (a) The gadget $\mathcal{H}$ (b) Two orthogonal gadgets (c) Two rotated gadgets

Finally we arrive at the following theorem.

Theorem 2. *It is* NP-hard *to count the number of vertex covers in a bichromatic unit disk graph even if the distances are measured in the L_∞ metric.*

Proof. The reduction is from counting the vertex covers in rectilinear unit disk graphs. It is inspired by a technique in [15]. Let $G = (V, E)$ be an instance of a rectilinear UDG. Let $G'(l) = (V'(l), E'(l))$ be the graph obtained from G by replacing each edge $uv \in E$ with the gadget $\mathcal{H}$ in Fig. 1 (a). By Lemma 4, $G'(l)$ is a bipartite unit disk graph (note that when $l = 0$, the graph $G'(l)$ has no edges at all). Let $N = \binom{m+2}{2}$, where $m = |E|$. We will show that by counting the vertex covers in $G'(l), l = 0, \ldots, N - 1$, we can effectively count the vertex covers in G.

In the graph $\mathcal{H}$, the number of vertex covers containing neither u nor v is 2^l since any vertex cover necessarily contains all the a_i's and c_i's ($i = 1, \ldots, l$), while b_i's may or may not be included. Moreover, the number of vertex covers containing one of u and v is 3^l, and the number of vertex covers containing both u and v is 5^l. (To see this, notice that when u is included and v is not, all the c_i's are necessarily included, and to cover the remaining edges on the i-th path, we need either a_i, b_i, or both. On the other hand, when both u and v are included, we would have 5 choices to cover the remaining two edges on each path.)

Let A_{ijk} be the number of subsets of nodes of the original graph, G, by which exactly i edges from G have none of their endpoints covered, j edges have exactly

one endpoint, and k edges have both of their endpoint covered. Then if $c'(l)$ is the number of vertex covers of $G'(l)$, we have

$$c'(l) = \sum_{i+j+k=m,\ i,j,k\geq 0} A_{ijk}(2^l)^i(3^l)^j(5^l)^k = \sum_{i+j+k=m,\ i,j,k\geq 0} A_{ijk}(2^i 3^j 5^k)^l. \quad (1)$$

The value we are interested in is $\sum_{j+k=m} A_{0jk}$, which is the number of vertex covers of G. Let $\mathbf{B} = (b_{ql})$ be the $N \times N$ matrix defined as

$$b_{ql} = (2^{i_q} 3^{j_q} 5^{k_q})^l \quad q = 1,\ldots,N, \quad l = 0,\ldots,N-1,$$

where (i_q, j_q, k_q) are all the triples summing up to m. Then $\mathbf{B}$ is a Vandermonde matrix and the values μ_q are distinct for $q = 1, \cdots, N$ since $\mu_q = 2^{i_q}, 3^{j_q}, 5^{k_q} = 2^{i_r}, 3^{j_r}, 5^{k_r} = \mu_r$ if and only if $i_q = i_r, j_q = j_r$ and $k_q = k_r$. By Lemma 3, we can solve (1) to obtain $\{A_{ijk}\}_{i+j+k=m,\ i,j,k\geq 0}$, and therefore also $\sum_{j+k=m} A_{0jk}$. We conclude that by counting the vertex covers in bichromatic unit disk graphs, we can count the vertex covers in rectilinear unit disk graphs. But Theorem 1 shows that this problem is hard.

Finally, we will show that the problem remains hard when we consider the distances under the L_∞ norm. Consider Fig. 1 (c), which illustrates a simple rotation of $G'(l)$ by the angle $\pi/2$. Consider the L_∞ ball around the point a. This ball does not include any point of a different color which is not connected to a in $G'(l)$. It is easy to see that the same applies to all other points in the graph, the connectivity structure of the graph remains unchanged, and so does the number of vertex covers. This completes the proof. $\qquad\square$

2.3 Complexity of the Stochastic Closest Pair Problem

We are now ready to prove the following result.

Theorem 3. *Given a set M of points in the plane, where each point $m_i \in M$ is present with probability p_i, it is NP-hard to compute the probability that the L_2 or L_∞ distance between the closest pair is $\leq \ell$ for a given value ℓ.*

Proof. In a unit disk graph $G = (V, E)$ defined over the full set M of points, a random subset S of nodes is a vertex cover if and only if in the complement of that subset, no two nodes are at distance ≤ 1. (In other words, all the edges are covered by S.) Therefore, computing the probability that a random subset of nodes is a vertex cover in G amounts to computing the probability that the closest pair of present points in a random subset S are at distance > 1. But as discussed in Theorem 1, counting the vertex covers in a unit disk graph is NP-hard. The fact that Theorem 1 applies to rectilinear unit disk graphs readily proves that the problem remains hard for the L_∞ metric.

The next theorem, which considers the bichromatic version of this problem, is based on the same argument as above along with Theorem 2.

Theorem 4. *Given a set R of red and a set B of blue points in the plane, where each point $r_i \in R$ is only present with an independent, rational probability p_i, and each point $b_i \in B$ is present with probability q_i, it is* NP-hard *to compute the probability that the closest L_2 or L_∞ distance between a bichromatic pair of present points is less that a given value ℓ.*

3   Linearly Separable Point Sets under the L_∞ Norm

In the following, we show that when the red points are linearly separable from the blue points by a vertical or a horizontal line, the stochastic bichromatic closest pair problem under L_∞ distances can be solved in polynomial time. We only describe the algorithm for the points separable by a vertical line, noting that all the arguments can be adapted to the case of a horizontal line.

Let $U = \{u_1, \ldots, u_n\}$ be the set of red points on one side, and $V = \{v_1, \ldots, v_m\}$ the set of blue points on the other side of a line. Each point $u_i \in U$ is present with probability p_i, while each point $v_j \in V$ is present with probability q_j. We sort the red points by x-coordinate (from right to left), and the blue points by y-coordinate (top-down). Let $R[i, j, k]$ be the region defined by the intersection of the halfplanes $x \leq 0$, $x \geq x(u_i) - 1$, $y \geq y(v_j)$ and $y \leq y(v_k)$, for $y(v_j) < y(v_k)$ (Fig. 2 (a)), where $x(u_i)$ and $y(v_j)$ denote the x-coordinate of the i-th red point and the y-coordinate of the j-th blue point, respectively. By abuse of notation, we will also use $R[i, j, k]$ to refer to the set of (blue) points inside this region.

Let $P[i, j, k]$ denote the probability that the subset $U_i = \{u_1, u_2, \ldots, u_i\}$ of red points does **not** have a neighbor within distance ≤ 1 in $R[i, j, k]$. The value we are interested in is $P[n, m, 1]$, which is the probability that the closest pair distance is > 1. We fill up the table for $P[i, j, k]$ values using dynamic programming, row by row starting from u_1 (the rightmost red point).
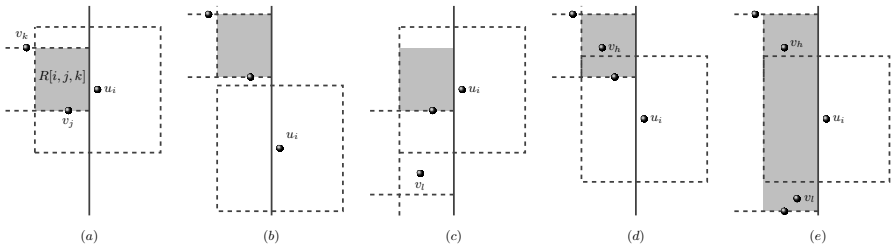
Fig. 2. Different configurations of $R[i, j, k]$ and $B(u_i)$

Let $B(u_i)$ be the L_∞ ball of radius 1 around u_i. In the computation of the entry $P[i, j, k]$, there are 4 cases:

1. $B(u_i)$ contains the region $R[i, j, k]$ (Fig. 2 (a)). In this case

$$P[i, j, k] = p_i \prod_{v_t \in B(u_i)} (1 - q_t) + (1 - p_i) \cdot P[i - 1, j, k].$$

2. $B(u_i)$ does not intersect with $R[i, j, k]$ (Fig. 2 (b)). In this case, $P[i, j, k] = P[i - 1, j, k]$.
3. $B(u_i)$ partially intersects with $R[i, j, k]$. If $y(u_i) - 1 < y(v_k)$ (Fig. 2 (c)),

$$P[i, j, k] = p_i \prod_{v_t \in B(u_i) \cap R[i,j,k]} (1 - q_t) \cdot P[i - 1, j, l] + (1 - p_i) \cdot P[i - 1, j, k],$$

where v_l is the highest blue point in $R[i, j, k]$ but outside $B(u_i)$.
If $y(u_i) + 1 < y(v_k))$ (Fig. 2 (d)), then

$$P[i, j, k] = p_i \prod_{v_t \in B(u_i) \cap R[i,j,k]} (1 - q_t) \cdot P[i - 1, h, k] + (1 - p_i) \cdot P[i - 1, j, k],$$

where v_h is the lowest blue point in $R[i, j, k]$ but outside $B(u_i)$.
4. $B(u_i)$ is contained in $R[i, j, k]$ (Fig. 2 (e)). In this case

$$P[i, j, k] = (1 - p_i) \cdot P[i - 1, j, k] + p_i \prod_{v_t \in B(u_i) \cap R[i,j,k]} (1 - q_t) \cdot P[i - 1, j, l] \cdot P[i - 1, h, k],$$

where v_l and v_h are defined as before. The subtlety is that the two events of U_{i-1} having no close neighbor in $R[i - 1, j, l]$, and U_{i-1} having no close neighbor in $R[i - 1, h, k]$ are independent. Therefore we can multiply the corresponding probabilities. The reason is that all the points in U_{i-1} that potentially have a close neighbor in $R[i - 1, j, l]$ must necessarily lie below the line $y = y(u_i)$, while those potentially close to a point in $R[i - 1, h, k]$ must lie above that line. The two sets are therefore disjoint.

The base case $(R[1, j, k], j, k \in \{1, \ldots, m\})$ can be easily computed. The size of the table is $O(n^3)$. The values $\prod_{v_t \in B(u_i) \cap R[i,j,k]} (1 - q_t)$ can be precomputed in $O(n^2)$ for each point v_i (by a sweep-line approach for example). This brings the computation time of each table entry down to a constant, and the running time of the whole algorithm to $O(n^3)$. This assumes a nonstandard RAM model of computation where each arithmetic operation on large numbers takes unit time.
Next Theorem considers the problem for $d > 2$.

Theorem 5. *Given a set R of red and a set B of blue points in a Euclidean space of dimension $d > 2$, each being present with an independent, rational probability, it is* NP-hard *to compute the probability that the L_∞ distance between the closest pair of bichromatic points is less than a given value r, even when the two sets are linearly separable by a hyperplane orthogonal to some axis.*

Proof. Let $d_\infty(R, B)$ be the minimum L_∞ distance between all the bichromatic pairs. It is always possible to make the two sets linearly separable in $d = 3$ by lifting all the blue (or red) points from the plane by a small value $\epsilon < d_\infty(R, B)$. This does not change the L_∞ distance of any pair of points. Therefore, an algorithm for solving the problem for linearly separable sets in $d > 2$, is essentially an algorithm for the stochastic bichromatic closest pair problem, which is NP-hard by Theorem 4. This completes the proof. □

4   Stochastic Approximate Nearest Neighbor Queries

Given a stochastic set M of points in a d-dimensional Euclidean space, and a query point q, what is the expected (L_2) distance of q to the closest present point of M? In this section we target this problem, and design a data structure for approximating the expected value of $d(S, q) = \min_{p \in S} d(p, q)$ with respect to a random subset S of M, assuming that d is a constant. (Technically, at least one point needs to be assigned probability 1 to ensure that the expected value is finite; alternatively, we can consider the expectation conditioned on the event that $d(S, q)$ is upper-bounded by a fixed value.) We obtain a linear-space data structure with $O(\log n)$ query time. Although our method is based on known techniques for approximate nearest neighbor search (namely, balanced quadtrees and shifting [3,4,5]), a nontrivial adaptation of these techniques is required to solve the stochastic version of the problem.

4.1   Approximation via a Modified Distance Function $\widetilde{\ell}$

As before, we are given a set M of points in a d-dimensional Euclidean space and each point is only *present* with an independent probability. Assume that the points lie in the universe $\{0, \ldots, 2^w - 1\}^d$. Fix an odd integer $k = \Theta(1/\varepsilon)$. Shift all points in M by the vector $(j2^w/k, j2^w/k, \ldots, j2^w/k)$ for a randomly chosen $j \in \{0, \ldots, k - 1\}$.

A *quadtree box* is a box of the form $[i_1 2^\ell, (i_1 + 1)2^\ell) \times \cdots \times [i_d 2^\ell, (i_d + 1)2^\ell)$ for natural numbers $\ell, i_1, \ldots, i_d$. Given points p and q, let $\mathcal{D}(p, q)$ be the side length of the smallest quadtree box containing p and q. Let $B_s(p)$ be the quadtree box of side length $\lfloor\lfloor s \rfloor\rfloor$ containing p, where $\lfloor\lfloor s \rfloor\rfloor$ denotes the largest power of 2 smaller than s. Let $c_s(p)$ denote the center of $B_s(p)$. Let $[X]$ be 1 if X is true, and 0 otherwise.

Definition 1. (a) *Define* $\ell(p, q) = d(B_s(p), B_s(q)) + 2\sqrt{d}s$ *with* $s = \varepsilon^2 \mathcal{D}(p, q)$. *Let* $\ell(S, q) = \min_{p \in S} \ell(p, q)$.

(b) r *is said to be* q-good *if the ball centered at* $c_{\varepsilon^2 r}(q)$ *of radius* $2r$ *is contained in* $B_{12kr}(q)$.

(c) *Define* $\widetilde{\ell}(S, q) = [\ell(S, q)$ *is* q-good$] \cdot \ell(S, q)$.

Lemma 5. (a) $\ell(S, q) \geq d(S, q)$. *Furthermore, if* $\ell(S, q)$ *is* q-good, then $\ell(S, q) \leq (1 + O(\varepsilon))d(S, q)$.

(b) $\ell(S, q)$ *is* q-good *for all but at most* d *choices of the random index* j.

(c) $\widetilde{\ell}(S, q) \leq (1 + O(\varepsilon))d(S, q)$ *always, and* $\mathbb{E}_j[\widetilde{\ell}(S, q)] \geq (1 - O(\varepsilon))d(S, q)$.

Proof. Let $p^*, p \in S$ satisfy $d(S, q) = d(p^*, q) = r^*$ and $\ell(S, q) = \ell(p, q) = r$.

The first part of (a) follows since $\ell(p, q) \geq d(p, q)$. For the second part of (a), suppose that r is q-good. Since $d(p^*, q) \leq d(p, q) \leq \ell(p, q) = r$, we have $d(p^*, c_{\varepsilon^2 r}(q)) < 2r$, implying $\mathcal{D}(p^*, q) \leq 12kr$. Then $r = \ell(p, q) \leq \ell(p^*, q) \leq d(p^*, q) + O(\varepsilon^2 \mathcal{D}(p^*, q)) \leq r^* + O(\varepsilon r)$, and so $r \leq (1 + O(\varepsilon))r^*$.

For (b), we use [6, Lemma 2.2], which shows that the following property holds for all but at most d choices of j: the ball centered at q with radius $3r^*$ is

contained in a quadtree box with side length at most $12kr^*$. By this property, $\mathcal{D}(p^*, q) \leq 12kr^*$, and so $r = \ell(p, q) \leq \ell(p^*, q) \leq d(p^*, q) + O(\varepsilon^2 \mathcal{D}(p^*, q)) = (1 + O(\varepsilon))r^*$. Then the ball centered at $c_{\varepsilon^2 r}(q)$ of radius $2r$ is contained in the ball centered at q of radius $(2 + O(\varepsilon^2))r < 3r^*$, and is thus contained in $B_{12kr^*}(q)$.

(c) follows from (a) and (b), since $1 - d/k \geq 1 - O(\varepsilon)$ (and $d(S, q)$ does not depend on j). $\qquad\square$

By (c), $\mathbb{E}_j[\mathbb{E}_S[\widetilde{\ell}(S, q)]]$ approximates $\mathbb{E}_S[d(S, q)]$ to within factor $1 \pm O(\varepsilon)$. It suffices to give an exact algorithm for computing $\mathbb{E}_S[\widetilde{\ell}(S, q)]$ for a query point q for a fixed j; we can then return the average, over all k choices of j.

4.2 The Data Structure: A BBD Tree

We use a version of Arya et al.'s balanced box decomposition (BBD) tree [3]. We form a binary tree T of height $O(\log n)$, where each node stores a cell, the root's cell is the entire universe, a node's cell is equal to the disjoint union of the two children's cells, and each leaf's cell contains $\Theta(1)$ points of M. Every cell B is a difference of a quadtree box (the *outer box*) and a union of $O(1)$ quadtree boxes (the *holes*). Such a tree can be constructed by forming the compressed quadtree and repeatedly taking centroids, as described by Arya et al. (in the original BBD tree, each cell has at most 1 hole and may not be perfect hypercubes). We will store $O(1/\varepsilon^{O(1)})$ amount of extra information (various expectation and probability values) at each node. The total space is $O(n/\varepsilon^{O(1)})$.

4.3 An Exact Query Algorithm for $\widetilde{\ell}$

In this section, we describe the algorithm for estimating $\mathbb{E}_S[\widetilde{\ell}(S, q)]$, given a query point q. First we extend the definition of $\widetilde{\ell}$ slightly: let $\widetilde{\ell}(S, q, r_0) = [\ell(S, q) \leq r_0] \cdot [\ell(S, q)$ is q-good$] \cdot \ell(S, q)$.

Consider a cell B of T and a query point $q \in B$. Let $R(B^c, q)$ denote the set of all possible values for $\ell(p, q)$ over points p in B^c, the complement of B. We solve the following extension of the query problem (all probabilities and expectations are with respect to the random subset S):

Problem 1. For every $r_0 \in R(B^c, q)$, compute the values $\Pr[\ell(S \cap B, q) > r_0]$ and $\mathbb{E}[\widetilde{\ell}(S \cap B, q, r_0)]$.

It suffices to compute these values for $r_0 \leq \sqrt{d}|B|$, where $|B|$ denotes the maximum side length of B, since they don't change as r_0 increases beyond $\sqrt{d}|B|$.

Lemma 6. *The number of elements in $R(B^c, q)$ that are below $\sqrt{d}|B|$ is $O(1/\varepsilon^{2d})$.*

Proof. If p is inside a hole H of B, then $\mathcal{D}(p, q) \geq |H|$, so we can consider a grid of side length $\Theta(\varepsilon^2|H|)$ and round p to one of the $O(1/\varepsilon^{2d})$ grid points without affecting the value of $\ell(p, q)$.

If p is outside the outer box of B, then $\mathcal{D}(p, q) \geq |B|$, so we can round p using a grid of side length $\Theta(\varepsilon^2|B|)$. In this case the number of grid points for $d(p, q) \leq \ell(p, q) \leq \sqrt{d}|B|$ is $O(1/\varepsilon^{2d})$ as well. $\qquad\square$

We now describe the query algorithm. The base case when B is a leaf is trivial. Let B_1 and B_2 be the children cells of B. Without loss of generality, assume that $q \in B_2$ (i.e., $q \notin B_1$). We apply the following formulas, based on the fact that $\ell(S \cap B, q) = \min\{\ell(S \cap B_1, q), \ell(S \cap B_2, q)\}$ and that $S \cap B_1$ and $S \cap B_2$ are independent:

$$\Pr[\ell(S \cap B, q) > r_0] = \Pr[\ell(S \cap B_1, q) > r_0] \cdot \Pr[\ell(S \cap B_2, q) > r_0]; \tag{2}$$

$$\mathbb{E}[\widetilde{\ell}(S \cap B, q, r_0)]$$
$$= \sum_{r \leq \sqrt{d}|B_2|} \Pr[\ell(S \cap B_1, q) = r] \cdot \mathbb{E}[\widetilde{\ell}(S \cap B_2, q, \min\{r, r_0\})] + \tag{3}$$

$$\sum_{r \leq \sqrt{d}|B_2|} \Pr[\ell(S \cap B_1, q) = r] \cdot \Pr[\ell(S \cap B_2, q) > r] \cdot [r < r_0] \cdot [r \text{ is } q\text{-good}] \cdot r \tag{4}$$

$$+ \Pr[\ell(S \cap B_1, q) > \sqrt{d}|B_2|] \cdot \mathbb{E}[\widetilde{\ell}(S \cap B_2, q, r_0)] \tag{5}$$

$$+ \mathbb{E}\left[[\ell(S \cap B_1, q) > \sqrt{d}|B_2|] \cdot \widetilde{\ell}(S \cap B_1, q, r_0)\right] \cdot \Pr[S \cap B_2 = \emptyset]. \tag{6}$$

(3) and (5) cover the case when $\ell(S \cap B_2, q) \leq \ell(S \cap B_1, q)$, and (4) and (6) cover the case when $\ell(S \cap B_1, q) < \ell(S \cap B_2, q)$. For (5), note that $\ell(S \cap B_2, q) \leq r_0$ already implies $S \cap B_2 \neq \emptyset$ and $\ell(S \cap B_2, q) \leq \sqrt{d}|B_2|$.

By recursively querying B_2, we can compute all probability and expectation expressions concerning $S \cap B_2$ in (2)–(6). Note that $r_0 \in R(B^c, q) \subseteq R(B_2^c, q)$, and in the sums (3) and (4), it suffices to consider $r \in R(B_2^c, q)$ since $S \cap B_1 \subset B_2^c$. In particular, the number of terms with $r \leq \sqrt{d}|B_2|$ is $O(1/\varepsilon^{2d})$, as already explained. For the probability and expectation expressions concerning $S \cap B_1$, we examine two cases:

- Suppose that q is inside a hole H of B_1. For all $p \in B_1$, $\mathcal{D}(p, q) \geq |H|$ and $\ell(p, q) \geq \Omega(\varepsilon^2 |H|)$, so we can consider a grid of side length $\Theta(\varepsilon^4 |H|)$ and round q to one of the $O(1/\varepsilon^{4d})$ grid points without affecting the value of $\ell(p, q)$, nor affecting whether $\ell(p, q)$ is q-good. Thus, all expressions concerning $S \cap B_1$ remain unchanged after rounding q. We can precompute these $O(1/\varepsilon^{O(1)})$ values for all grid points q (in $O(n/\varepsilon^{O(1)})$ time) and store them in the tree T.
- Suppose that q is outside the outer box of B_1. For all $p \in B_1$, $\mathcal{D}(p, q) \geq |B_1|$, so we can consider a grid of side length $\Theta(\varepsilon^2 |B_1|)$ and round each point $p \in M \cap B_1$ to one of the $O(1/\varepsilon^{2d})$ grid points without affecting the value of $\ell(p, q)$. Duplicate points can be condensed to a single point by combining their probabilities; we can precompute these $O(1/\varepsilon^{2d})$ probability values (in $O(n)$ time) and store them in the tree T. We can then evaluate all expressions concerning $S \cap B_1$ for any given q by brute force in $O(1/\varepsilon^{O(1)})$ time.

Since the height of T is $O(\log n)$, this recursive query algorithm runs in time $O((1/\varepsilon^{O(1)}) \log n)$. Therefore we arrive at the main result of this section.

Theorem 6. *Given a stochastic set of n points in a constant dimension d, we can build an $O(n/\varepsilon^{O(1)})$-space data structure in $O((1/\varepsilon^{O(1)})n \log n)$ time, so*

that for any query point, we can compute a $(1 + \varepsilon)$-factor approximation to the expected nearest neighbor distance in $O((1/\varepsilon^{O(1)}) \log n)$ time.

Acknowledgment

References

1. Afshani, P., Agarwal, P.K., Arge, L., Larsen, K.G., Phillips, J.M.: (Approximate) uncertain skylines. In: ICDT, pp. 186–196 (2011)
2. Agarwal, P.K., Cheng, S.-W., Tao, Y., Yi, K.: Indexing uncertain data. In: PODS, pp. 137–146 (2009)
3. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. J. ACM 45, 891–923 (1998)
4. Chan, T.M.: Approximate Nearest Neighbor Queries Revisited. Discrete and Computational Geometry 20, 359–373 (1998)
5. Chan, T.M.: Closest-point problems simplified on the RAM. In: Proc. SODA, pp. 472–473 (2002)
6. Chan, T.M.: Polynomial-time approximation schemes for packing and piercing fat objects. J. Algorithms 46, 178–189 (2003)
7. De Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: Computational geometry: algorithms and applications. Springer, Heidelberg (2008)
8. Garey, M.R., Johnson, D.S.: The Rectilinear Steiner Tree Problem is NP-Complete. SIAM Journal on Applied Mathematics 32(4), 826–834 (1977)
9. Hardy, G.H., Polya, G., Littlewood, J.E.: Inequalities. Cambridge Press, New York (1952)
10. Kamousi, P., Chan, T., Suri, S.: Stochastic Minimum Spanning Trees in Euclidean Spaces. In: Proc. SoCG (2011) (to appear)
11. Klain, D.A., Rota, G.: Introduction to Geometric Probability, Cambridge (1997)
12. Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley, Reading (1973)
13. Lin, M.-S., Chen, Y.-J.: Counting the number of vertex covers in a trapezoid graph. Inf. Process. Lett. 109, 1187–1192 (2009)
14. Löffler, M., van Kreveld, M.J.: Largest and Smallest Convex Hulls for Imprecise Points. Algorithmica 56(2), 235–269 (2010)
15. Provan, J.S., Ball, M.O.: The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected. SIAM J. Comput. 12(4), 777–788 (1983)
16. Vadhan, S.P.: The Complexity of Counting in Sparse, Regular, and Planar Graphs. SIAM Journal on Computing 31, 398–427 (1997)
17. Valiant, L.: Universality Considerations in VLSI Circuits. IEEE Trans. Computers 30, 135–140 (1981)
18. Valiant, L.G.: The Complexity of Enumeration and Reliability Problems. SIAM Journal on Computing 8(3), 410–421 (1979)
19. van Kreveld, M.J., Löffler, M., Mitchell, J.S.B.: Preprocessing Imprecise Points and Splitting Triangulations. SIAM J. Comput. 39(7), 2990–3000 (2010)

Competitive Search in Symmetric Trees

David Kirkpatrick[1] and Sandra Zilles[2]

[1] Department of Computer Science, University of British Columbia, Canada
`kirk@cs.ubc.ca`
[2] Department of Computer Science, University of Regina, Canada
`zilles@cs.uregina.ca`

Abstract. We consider the problem of searching for one of possibly many goals situated at unknown nodes in an unknown tree $\mathcal{T}$. We formulate a universal search strategy and analyse the competitiveness of its average (over all presentations of $\mathcal{T}$) total search cost with respect to strategies that are informed concerning the number and location of goals in $\mathcal{T}$. Our results generalize earlier work on the multi-list traversal problem, which itself generalizes the well-studied m-lane cow-path problem. Like these earlier works our results have applications in areas beyond geometric search problems, including the design of hybrid algorithms and the minimization of expected completion time for Las Vegas algorithms.

1   Introduction

The *m-lane cow-path problem* specifies a sequence of m rays (*lanes*) of unbounded length incident on a common origin (*crossroad*). A goal (*pasture*) lies at some unknown distance d from the origin along some (unknown) ray. The objective is to formulate a provably good strategy (minimizing the total search cost) for an agent (*cow*) to reach the goal, starting from the origin.

The cow-path problem is a special instance of a family of problems called *search games*, in which a searcher tries to minimize the time needed to find a hidden goal. For a detailed study of search games, the reader is referred to [1]. The cow-path problem itself has been studied in several variations, including directionally dependent traversal costs, turnaround penalties, shortcuts and dead-ends [4,6,11,12,15]. It has also been analysed in terms of worst-case and average-case competitive ratio (using d as a benchmark), as well as in a game-theoretic framework [3,8,16,17,18].

Essentially the same ideas as those used in solving the cow-path problem have been used in the synthesis of deterministic and randomized hybrid algorithms with (near) optimal competitive ratios [2,7]. Given are a number of *basic* algorithms each of which might (or might not) be useful in solving some problem. The goal is to synthesize a hybrid algorithm from these basic components by some kind of dovetailing process. Memory limitations may restrict the number of processes that can be suspended at any given time (the alternative being a complete restart with successively larger computation bounds).

More recently, the cow-path problem has been generalized in a new and fundamentally different direction. The *multi-list traversal problem* [10] assumes that *every ray* leads to a goal, and the objective is to minimize the total search cost in finding a goal on

at least one path. (Conventional one-goal cow-path problems correspond to the special case in which all goals but one are located arbitrarily far from the origin). Essentially the same problem has been studied by McGregor *et al.* [14] as an "oil searching problem", where the objective is to maximize the number of goals (wells) discovered for a specified budget. Even earlier, similar results were presented by Luby *et al.* [13] for the problem of minimizing the expected execution time of Las Vegas algorithms (viewed as an infinite sequence of deterministic algorithms with unknown completion times.)

The m-lane cow paths problem, the multi-list traversal problem, and its variants can all be thought of as search-with-backtracking problems, in which backtracking always brings the algorithm back to the origin of search, from where a new path can be chosen or search in a previously visited path can be resumed. In many real-world search problems, it is often the case that part of the search effort invested into one search path eases the search along another path. Backtracking would then allow the search algorithm to return to a fork part-way along the current path and to search along a new path branching from the current one (without repeating the search effort to reach the fork from the origin). The simplest search domain allowing this kind of backtracking is a tree.

Motivated by the desire to understand the limitations of oblivious backtracking algorithms, we consider a generalization of the multi-list traversal problem in which the search domain is an unknown unbounded fully-symmetric tree $\mathcal{T}$ with goals at one or more nodes. Fleischer *et al.* [5] considered search problems on trees as part of a more general study of algorithms that minimize the optimal search ratio: essentially the worst case ratio of the cost of finding a goal in a search domain to the length of the shortest path to that goal. For our competitive analysis we compare uninformed algorithms to those that know $\mathcal{T}$, including the locations of all goals, but not the specific presentation of $\mathcal{T}$ (i.e. the ordering of the children at each of its internal nodes). In fact, McGregor *et al.* [14] already introduced a generalization of their oil-searching problem to symmetric trees in an appendix to their paper. Unfortunately, their algorithm exhibits a rather poor competitive ratio for general symmetric trees, motivating a more in-depth treatment of the symmetric tree search problem. Note that while it is possible to study backtracking in asymmetric trees (or more general graphs), it is natural to restrict attention to search domains in which all search paths are equivalent up to relabeling: as McGregor *et al.* [14] point out, asymmetries serve to amplify the knowledge (concerning goal locations) of informed algorithms, making competitive analysis simultaneously more difficult and less meaningful.

1.1 Symmetric Tree Traversal

In many respects our treatment of search in symmetric trees parallels and generalizes earlier work on the multi-list search problem. Where previously an algorithm would be evaluated with respect to possible input presentations ranging over *all possible permutations of a multi-set of list lengths*, we are now interested in inputs that correspond to presentations of some fixed symmetric tree. Thus an *instance* of our *symmetric tree traversal problem* is an unbounded rooted unordered fully-symmetric[1] tree $\mathcal{T}$, one or more nodes of which are distinguished as *goal* nodes, called *goals* for short. We assume,

---

[1] All nodes at the same level ℓ have the same number of children d_ℓ.

without loss of generality, that the path from the root to any goal does not contain any other goal. We denote by $\Pi(\mathcal{T})$ the set of all *presentations* of the problem instance $\mathcal{T}$. Each such presentation is an ordering of $\mathcal{T}$, i.e. for each internal node x of $\mathcal{T}$, a bijection from the set $\{1, \ldots, d_x\}$ to the edges joining x to the d_x children of x. In this way, every presentation of $\mathcal{T}$ assigns to every node x, and in particular every goal, in $\mathcal{T}$ a labeled path from the root to x. We interpret the concatenation of labels on this path as the *index* of x in the given presentation.

We assume that in general algorithms take an arbitrary presentation π of $\mathcal{T}$ as input, and know nothing about the number or location of goals in $\mathcal{T}$. Algorithms proceed in a stepwise fashion. In the first step the root node is explored, and in every subsequent step a specified child of some previously explored node is explored, continuing until some goal node is reached. We denote by $\text{search\_cost}(\mathcal{A}, \pi)$ the total search cost (number of explored nodes) of algorithm $\mathcal{A}$ on input presentation π.[2] We analyse this search cost of algorithms (both deterministic and randomized) for specific problem instances $\mathcal{T}$ in both the worst and average cases (over all presentations of $\mathcal{T}$). For worst-case behaviour we can think of an adversary choosing the least favorable presentation of $\mathcal{T}$, knowing the search strategy of the algorithm. We view randomized (Las Vegas) algorithms as probability distributions over deterministic algorithms; in this case we are interested in *expected* search cost.

For the purpose of competitive analysis we contrast general *uninformed* algorithms with several *informed* variants that are only required to behave correctly on problem instances that satisfy certain constraints on the number or location of the goals. A *instance-informed* algorithm knows the problem instance, i.e. the location of goals in $\mathcal{T}$, but not their index in the given input presentation. A *level-count-informed* algorithm knows the number of goals at each level of $\mathcal{T}$, but not their location. A *cost-informed* algorithm knows an upper bound on the worst-case search cost that is realizable by the best instance-informed algorithm for the given instance.

We start by restricting our attention to the case where $\mathcal{T}$ is a full binary tree (i.e. $d_\ell = 2$, at every level). Section 2 considers the situation where all goals are known to lie on one fixed level of $\mathcal{T}$, and results are developed for both the full search cost as well as the search cost restricted to the goal level. These results are extended, in Section 3, to the general situation where goals may appear on multiple levels. Finally, the restriction to binary trees is relaxed in Section 4. (Most of the proofs in this last section are most easily understood as elaborations of the proofs of corresponding results for binary trees; full details of these proofs are presented in [19].)

In general, our oblivious search algorithms not only significantly improve the search bounds provided by the tree-searching algorithm of McGregor *et al.* [14], but they also are arguably close to optimal in the competitive ratio of their search cost with that of non-oblivious counterparts. For example, for binary trees with k goals on one fixed level h, our algorithm guarantees an average search cost that is within a factor h of that achievable by any algorithm that is only required to perform efficiently on presentations of one fixed tree. In the same situation, the strategy proposed in [14] is only claimed to

---

[2] Our results apply equally well when the cost of backtracking is taken into account, *i.e.*, when the search cost includes the cost of re-visiting nodes.

have a corresponding competitive ratio which is bounded by the square of the number of nodes in the tree!

2   The Case Where All Goals Are Known to Lie at the Same Level

In the multi-list traversal problem the best uninformed strategy employs a non-uniform interleaving (dubbed "hyperbolic dovetailing" in [10]) of strategies each of which searches all lists uniformly to some fixed depth. Motivated by that, we first consider the case where all goals are known to lie at some fixed level h. In this case, it does not make any sense for an algorithm to explore paths in $\mathcal{T}$ to a level more or less than h. Therefore we initially consider $\mathcal{T}$ to be truncated at level h and count just the number of *probes* an algorithm makes of nodes at the leaf level h, ignoring the cost associated with reaching those nodes. In this restricted setting, a level-count-informed algorithm knows the number k of goals at level h in $\mathcal{T}$, but not their location. We denote by $\mathrm{probe\_cost}(\mathcal{A}, \pi)$ the total number of nodes on level h explored by algorithm $\mathcal{A}$ on input presentation π.

Since every presentation of the full binary tree $\mathcal{T}$ of height h fixes, for each of its $2^h - 1$ internal nodes x, one of two possible labelings on the pair of edges leading to the children of x , we have the following:

Observation 1. *If $\mathcal{T}$ is a full binary tree of height h then $|\Pi(\mathcal{T})| = 2^{2^h - 1}$.*

2.1   Worst-Case Probe Cost

It is clear that an arbitrary uninformed probing algorithm will never need to make more than $2^h - k + 1$ probes at level h, when faced with a problem instance $\mathcal{T}$ with exactly k goals at level h. On the other hand, an adversary can force this many probes by any fixed (even count-informed) algorithm by choosing a suitable problem instance $\mathcal{T}$ with exactly k goals at level h and a suitable presentation $\pi \in \Pi(\mathcal{T})$. Thus,

Observation 2. *For every deterministic level-count-informed algorithm $\mathcal{A}$, there exists a problem instance $\mathcal{T}$ with exactly k goals at level h such that*
$\max_{\pi \in \Pi(\mathcal{T})} \mathrm{probe\_cost}(\mathcal{A}, \pi) = 2^h - k + 1$.

As we observe next, fully informed probing algorithms can, at least for some problem instances, have significantly lower worst-case probe cost. In the next section, we show that similar reductions are always achievable if we measure instead the average or expected probe cost.

Observation 3. *There exists a deterministic instance-informed algorithm $\mathcal{A}$ and, for every $j \geq 0$, a problem instance $\mathcal{T}_j$ with exactly 2^j goals at level h, such that* $\max_{\pi \in \Pi(\mathcal{T}_j)} \mathrm{probe\_cost}(\mathcal{A}, \pi) \leq 2^{h-j}$.

Proof. If tree $\mathcal{T}_j$ has goals at all 2^j leaves of some subtree rooted at an internal node at level $h - j$, then it suffices to probe one leaf in each of the 2^{h-j} subtrees rooted at level $h - j$, in any presentation of $\mathcal{T}_j$. As a second example, if $\mathcal{T}_j$ has one goal in each of its 2^j subtrees rooted at internal nodes at level j, it suffices to explore all 2^{h-j} leaves in any one of these subtrees, in any presentation of $\mathcal{T}_j$. □

It follows from Theorem 4 below that instances like $\mathcal{T}_j$ above are the *least* complex, in terms of their worst-case probe cost, for fully informed algorithms. As Theorem 5 and its corollary demonstrate, the *most* complex such instances have a significantly higher worst-case probe cost.

Theorem 4. *For every deterministic instance-informed algorithm* $\mathcal{A}$, *and every problem instance* $\mathcal{T}$ *with exactly* k *goals at level* h,
$\max_{\pi \in \Pi(\mathcal{T})} \text{probe\_cost}(\mathcal{A}, \pi) \geq 2^h/k$.

Proof. For any node x in $\mathcal{T}$ and any index i of a fixed probe location at level h, x is assigned index i in exactly $\frac{2^{2^h-1}}{2^h}$ presentations of $\mathcal{T}$, since any presentation that maps x to a fixed probe location fixes the labels associated with the h edges on the path to that goal, and only those edges. Thus, for any i, there are exactly $k\frac{2^{2^h-1}}{2^h}$ presentations that assign one of k goals to the probe with location index i. It follows that any deterministic algorithm that uses fewer than $\frac{2^h}{k}$ probes at level h fails to detect a goal for at least one presentation of $\mathcal{T}$. □

Theorem 5. *For any* r, $0 \leq r \leq h$, *there exists a problem instance* $\mathcal{T}_{r,h}$ *with* $k = \sum_{j=r}^{h} \binom{h}{j}$ *goals at level* h, *such that for every deterministic instance-informed algorithm* $\mathcal{A}$, $\max_{\pi \in \Pi(\mathcal{T}_{r,h})} \text{probe\_cost}(\mathcal{A}, \pi) \geq 2^r$.

Proof. (Sketch) The tree $\mathcal{T}_{r,h}$ is defined recursively for $0 \leq r \leq h$: (i) $\mathcal{T}_{0,h}$ is the complete tree with 2^h leaves, all of which are goals; (ii) $\mathcal{T}_{h,h}$ is the complete tree with 2^h leaves, exactly one of which is a goal; and (iii) $\mathcal{T}_{r,h}$ is the complete tree whose root has subtrees $\mathcal{T}_{r,h-1}$ and $\mathcal{T}_{r-1,h-1}$, when $0 < r < h$.

One can show, by induction on r and h, that (i) $\mathcal{T}_{r,h}$ has $k = \sum_{j=r}^{h} \binom{h}{j}$ goals at level h and (ii) for any set of fewer than 2^r probes in $\mathcal{T}_{r,h}$ there is a presentation of $\mathcal{T}_{r,h}$ for which no probe detects a goal. (See [19] for details.) □

2.2 Average and Expected-Case Probe Cost

Theorem 4 extends to average case behaviour of fully informed algorithms:

Theorem 6. *For every deterministic instance-informed algorithm* $\mathcal{A}$, *and every problem instance* $\mathcal{T}$ *with exactly* k *goals at level* h,
$\text{avg}_{\pi \in \Pi(\mathcal{T})} \text{probe\_cost}(\mathcal{A}, \pi) \geq 2^{h-2}/k$.

Proof. As shown in the proof of Theorem 4, for any i, there are exactly $k\frac{2^{2^h-1}}{2^h}$ presentations that assign one of k goals to the location index i. Thus, any deterministic algorithm using fewer than $\frac{2^{h-1}}{k}$ probes at level h fails to detect a goal in at least half of the presentations of $\mathcal{T}$. Hence every deterministic algorithm uses at least $\frac{2^{h-1}}{k}$ probes at level h on at least half of its input presentations. □

Theorem 6 can be strengthened to apply to the expected case behaviour of randomized instance-informed algorithms $\mathbf{A}$, by viewing $\mathbf{A}$ as a probability distribution over deterministic algorithms in the standard way (see [19] for details).

Theorem 7. *For every randomized instance-informed algorithm* **A**, *and every problem instance* $\mathcal{T}$ *with exactly* k *goals at level* h,
$\mathrm{avg}_{\pi \in \Pi(\mathcal{T})} E[\mathrm{probe\_cost}(\mathbf{A}, \pi)] \geq 2^{h-2}/k$.

The following theorem, whose proof embodies the central idea of our general oblivious tree-searching strategy, shows that the lower bound of Theorem 6 is realizable to within a constant factor, even by an uninformed algorithm.

Theorem 8. *There is a deterministic uninformed algorithm* $\mathcal{A}_0$ *such that, for every problem instance* $\mathcal{T}$ *with exactly* k *goals at level* h,
$\mathrm{avg}_{\pi \in \Pi(\mathcal{T})} \mathrm{probe\_cost}(\mathcal{A}_0, \pi) \leq 2^{h+2}/k$.

Proof. For any $r, 0 \leq r \leq h$, we can interpret an arbitrary presentation of $\mathcal{T}$ as a *bottom tree* $\mathcal{T}'$, consisting of all nodes of $\mathcal{T}$ at level at most r, together with 2^r *top* trees, each with 2^{h-r} leaves.

The algorithm $\mathcal{A}_0$ proceeds in rounds: at the completion of round $r \geq 0$, exactly one leaf in each of the 2^r trees rooted at nodes on level r has been probed. The algorithm terminates if a goal is discovered in at least one of its probe locations. The total number of probes in round r is just $2^r - 2^{r-1} = 2^{r-1}$.

We count the fraction Φ_r of presentations of $\mathcal{T}$ for which algorithm $\mathcal{A}_0$ terminates by the end of round r. Each goal resides in one of the 2^r top trees with 2^{h-r} leaves, and coincides with the probed leaf in that tree in exactly $\frac{1}{2^{h-r}}$ of the presentations of that top tree. Thus each individual goal is probed in $\frac{1}{2^{h-r}}$ of the presentations of $\mathcal{T}$, by the end of round r.

Of course, some presentations map two or more goals to probe positions. So to count Φ_r we number the goals arbitrarily and, for $1 \leq i \leq k$, we count, among the presentations of $\mathcal{T}$ that map none of the first $i-1$ goals to a probe position, the fraction f_i that map the i-th goal to a probe position. Clearly, $\Phi_r = \sum_{1 \leq i \leq k} f_i \cdot [\prod_{1 \leq j < i}(1 - f_j)]$. Furthermore, $f_i \geq \frac{1}{2^{h-r}}$, where equality holds just when none of the first $i-1$ goals occupy the same top tree as the i-th goal.

If we define $F_x = \sum_{x \leq i \leq k} f_i \cdot [\prod_{x \leq j < i}(1 - f_j)]$, for $1 \leq x \leq k$, then $F_k = f_k$ and, for $1 \leq x < k$, $F_x = f_x + (1 - f_x)F_{x+1}$. It is straightforward to confirm by induction that $F_x \geq 1 - (1 - \frac{1}{2^{h-r}})^{k-x+1}$. Thus $\Phi_r = F_1 \geq 1 - (1 - \frac{1}{2^{h-r}})^k > 1 - (\frac{1}{e})^{k/2^{h-r}}$.

Now if $2^{h-j} \leq k < 2^{h+1-j}$, then at most $(\frac{1}{e})^{k/2^{h-j-i}} \leq (\frac{1}{e})^{2^i}$ of the presentations of $\mathcal{T}$ have not terminated after $r = j + i$ rounds. Hence the average, over all presentations of $\mathcal{T}$, of the number of probes of algorithm $\mathcal{A}_0$ is at most
$2^j + \sum_{i \geq 1}(2^{j+i-1}(\frac{1}{e})^{2^{i-1}}) < 2^j(1 + \sum_{s \geq 1}(s(\frac{1}{e})^s)) < 2^j(1 + \frac{e}{(e-1)^2}) < 4\frac{2^h}{k}$. $\square$

Remark 1. Choosing $k = 2^{h-1}$ in Theorem 8 and $r = h/2$ in Theorem 5 demonstrates a large gap between the average and worst-case behaviours of deterministic instance-informed algorithms. Specifically, the problem instance $\mathcal{T}_{h/2,h}$ with 2^{h-1} goals at level h has the property that algorithm $\mathcal{A}_0$ has average probe cost of at most 8, whereas every deterministic instance-informed algorithm requires at least $2^{h/2}$ probes in the worst case.

Remark 2. It is easy to see that the total additional search cost in round r of Algorithm $\mathcal{A}_0$ is $2^{r-1}(h - r + 1)$. Thus if $2^{h-j} \leq k < 2^{h+1-j}$ the proof above implies that the

average total search cost is at most $2^j(h-j) + \sum_{i \geq 1}(2^{j+i-1}(h-j-i+1)(\frac{1}{e})^{2^{i-1}}) < 2^j(h-j)(1 + \sum_{s \geq 1}(s(\frac{1}{e})^s)) = O((h-j)\frac{2^h}{k}) = O(\frac{2^h}{k}(1+\lg k))$.

By simply randomizing the given presentation before running algorithm $\mathcal{A}_0$ the average-case bound of Theorem 8 can be realized as the worst-case expected cost, providing a tight complement to the lower bound of Theorem 7:

Corollary 9. *There is a randomized uninformed algorithm $\mathbf{A}_1$ such that, for every problem instance $\mathcal{T}$ with exactly k goals at level h,*
$\max_{\pi \in \Pi(\mathcal{T})} E[\text{probe\_cost}(\mathbf{A}_1, \pi)] \leq 2^{h+2}/k$.

2.3 Taking Full Search Cost into Consideration

As noted above, the algorithm $\mathcal{A}_0$ outlined in Section 2.2 has probe cost $O(\frac{2^h}{k})$ but total search cost $O(\frac{2^h}{k}(1+\lg k))$. For some problem instances, e.g., the tree $\mathcal{T}_j$ (described in Theorem 3) with goals at its leftmost $k = 2^j$ leaves, even fully informed algorithms require average total search cost $\Omega(\frac{2^h}{k}(1+\lg k))$, since at least one probe must be made in at least half of the top-level trees of size k, or the algorithm will fail on at least half of the permutations. Hence this additional $\lg k$ factor is unavoidable in some cases, even when $k = o(2^h)$.

Nevertheless, we have not been able to formulate a notion of intrinsic total search cost that would permit a tighter general competitive bound than that given by the following:

Theorem 10. *The uninformed algorithm $\mathcal{A}_0$ has the property that, for every problem instance $\mathcal{T}$, $\text{avg}_{\pi \in \Pi(\mathcal{T})}\text{search\_cost}(\mathcal{A}_0, \pi) = O(c_{\text{inf}}(\mathcal{T}) \cdot (h+1-\lg(c_{\text{inf}}(\mathcal{T}))))$, where $c_{\text{inf}}(\mathcal{T})$ denotes the minimum, over all informed algorithms $\mathcal{B}$, of $\text{avg}_{\pi \in \Pi(\mathcal{T})}$ probe_cost$(\mathcal{B}, \pi)$.*

Proof. Suppose that input $\mathcal{T}$ has k goals. By Theorem 6, $c_{\text{inf}}(\mathcal{T})$ is $\Omega(2^h/k)$. Furthermore, it is easy to see from the proof of Theorem 8 that the average, over all presentations $\pi \in \Pi(\mathcal{T})$, of the total search cost of $\mathcal{A}_0$ on presentation π is
$O(\frac{2^h}{k}(1 + h - \lg(\frac{2^h}{k}))) = O(c_{\text{inf}}(\mathcal{T}) \cdot (h+1-\lg(c_{\text{inf}}(\mathcal{T}))))$.   □

Following Corollary 9, it is easy to see that the competitive bound in Theorem 10 holds for the expected search cost of Algorithm $\mathbf{A}_1$ as well. This should be contrasted with the $O(c_{\text{inf}}(\mathcal{T}) \cdot 4^h)$ bound, given by Theorem 23 of McGregor *et al.* [14], for the expected cost of their uninformed search strategy in this same situation.

3 The Case Where Goals May Appear on Many Different Levels

To this point we have assumed that all problem instances have the property that all goals lie on one fixed level h. In this section we develop a dovetailing strategy that allows us to relax this assumption.

We have already noted that the uninformed algorithm $\mathcal{A}_0$ described in Theorem 8 is competitive (in terms of expected total search cost), to within a factor of at most h,

with the best fully informed algorithm, for input instances all of whose goals lie on level h. For more general instances, we first generalize Theorem 6, establishing a lower bound on the intrinsic total expected search cost, and then show how algorithm $\mathcal{A}_0$ can be modified to minimize its competitive ratio with this bound. We then argue that the competitive ratio achieved by this modified uninformed algorithm cannot be improved, by more than a logarithmic factor, even by an algorithm that is cost-informed (that is, is constrained only to work correctly for problem instances of a known bounded intrinsic cost).

Theorem 11. *For every deterministic instance-informed algorithm $\mathcal{A}$, and every problem instance $\mathcal{T}$ with exactly k_t goals at level t,*
$$\text{avg}_{\pi \in \Pi(\mathcal{T})}\text{search\_cost}(\mathcal{A}, \pi) \geq \min_{t \geq 0}\{t + \tfrac{2^t}{2k_t}\}/2.$$

Proof. Let $\mathcal{T}$ be any problem instance with exactly k_t goals at level t and let $\mathcal{A}$ be any informed goal-searching algorithm. Suppose $\mathcal{A}$ makes p_t probes at level t, and let $m = \max\{t \mid p_t > 0\}$ and $p = \sum_{t \geq 0} p_t$. We consider the fraction of the presentations of $\mathcal{T}$ that take some goal to some probe location. We can restrict our attention to the 2^{2^m-1} presentations of $\mathcal{T}$ truncated at level m. By the argument in Theorem 4, at most $p_t k_t \frac{2^{2^m-1}}{2^t}$ presentations take a goal on level t to a probe on level t. Thus at most a fraction $\sum_{t=0}^{m} \frac{p_t k_t}{2^t} \leq p / \min_{t \leq m}\{\frac{2^t}{k_t}\}$ of the presentations of $\mathcal{T}$ take some goal to some probe location. It follows that if $p < \min_{t \leq m}\{\frac{2^t}{k_t}\}/2$ then $\mathcal{A}$ fails to detect a goal for at least half of the presentations of $\mathcal{T}$. Thus, any deterministic algorithm must make at least $\min_{m \geq 0} \max\{m, \min_{t \leq m}\{\frac{2^t}{k_t}\}/2\} = \min_{t \geq 0} \max\{t, \frac{2^t}{2k_t}\} \geq \min_{t \geq 0}\{t + \frac{2^t}{2k_t}\}/2$ probes on at least half of the presentations of $\mathcal{T}$. $\square$

Algorithm $\mathcal{A}_0$, as described in the proof of Theorem 8, makes 2^r equally spaced probes, for increasing values of r, at one fixed level h, at a total cost of $2^r(h-r+1)$. To spread the cost equitably among levels we formulate a modification $\mathcal{A}_2$ of algorithm $\mathcal{A}_0$ that, for increasing values of r, probes all 2^r nodes at level r, and makes 2^{r-i} equally spaced probes at all 2^i levels in the interval $(r-2+2^i, r-2+2^{i+1}]$, for $1 \leq i < r$.

Algorithm $\mathcal{A}_2$ effectively simulates algorithm $\mathcal{A}_0$, for all values of h. The total cost of algorithm $\mathcal{A}_2$, up to a fixed value r_0 of the parameter r, is $(r_0 + 1)2^{r_0}$. Let $t_0 = \arg\min_{t \geq 0}\{(t+1)2^t/k_t\}$. Then, from the proof of Theorem 8, we know that the fraction of presentations for which algorithm $\mathcal{A}_2$ requires more than $2^j 2^{t_0}/k_{t_0}$ probes on level t_0 before hitting a goal is less than $(\frac{1}{e})^{2^j}$. It follows that the average number of probes made on level t_0 before hitting a goal on that level is $O(2^{t_0}/k_{t_0})$ and the average total search cost of algorithm $\mathcal{A}_2$ is $O((r_0 + 1)2^{r_0+1})$, provided $2^{r_0} \geq (t_0 - r_0 + 1)2^{t_0}/k_{t_0}$.

We summarize this result in the following:

Theorem 12. *The uninformed algorithm $\mathcal{A}_2$ has the property that, for every problem instance $\mathcal{T}$ with exactly k_t goals at level t,*
$$\text{avg}_{\pi \in \Pi(\mathcal{T})}\text{search\_cost}(\mathcal{A}_2, \pi) \leq \min_{t \geq 0}\{(t+1)\tfrac{2^t}{k_t}\} \cdot \lg(\min_{t \geq 0}\{(t+1)\tfrac{2^t}{k_t}\}).$$

When $k_{t_0} = 2^{t_0}/t_0$ and $k_t = 0$, when $t \neq t_0$, the ratio of the $O(t_0^2 \lg t_0)$ average search cost of Algorithm $\mathcal{A}_2$ (given by Theorem 12) and the $\Omega(t_0)$ lower bound on the

same cost for any *instance-informed* algorithm (given by Theorem 11), is maximized. It turns out that at least a quadratic cost inflation is unavoidable, even for cost-informed algorithms:

Theorem 13. *For every cost $c \geq 0$, there is a family $\mathcal{F}$ of problem instances, each member of which can be searched with worst-case total search cost at most c by some fully informed deterministic search algorithm, such that any cost-informed search algorithm $\mathcal{A}$ must have average, over all input presentations, total search cost at least $\Omega(c^2)$, on at least half of the instances in the family.*

Proof. (Sketch) $\mathcal{F}$ includes instances $\mathcal{T}_i$ with $2^{i+1}/(c-i)$ goals equally-spaced on level i. For each such instance $(c-i)/2$ probes at level i (and at most c total search cost) suffices in the worst case, by a instance-informed algorithm (cf. Theorem 3), and $(c-i)/8$ probes at level i are necessary on average (by Theorem 6).   □

4   General Symmetric Trees

To this point we have restricted our attention to full binary trees. Not surprisingly, all of our results generalize to arbitrary symmetric trees. There are some subtleties, however, arising both from nodes with just one child, which can be used to form trees whose number of leaves is significantly smaller than the number of internal nodes, and nodes with a very large number of children, which complicate our round-based algorithms. In the remainder of this section, we outline our generalized results.

We denote by $D_{i,j}$ the expression $\prod_{\ell=i}^{j} d_\ell$, where d_ℓ, recall, denotes the number of children of all internal nodes at level ℓ. Clearly, the number of nodes at level h is now $D_{0,h-1}$, and Observation 1 generalizes to the following:

Observation 14. *If $\mathcal{T}$ is a general symmetric tree of height h then $|\Pi(\mathcal{T})| = \prod_{j=0}^{h-1} d_j^{D_{0,j-1}}$.*

Using this, Theorems 4, 6 and 7 generalize directly to arbitrary symmetric trees, with 2^h replaced by $D_{0,h-1}$. Theorem 8 generalizes in the same way, by a relatively straightforward modification of algorithm $\mathcal{A}_0$:

Theorem 15. *There is a deterministic uninformed algorithm $\mathcal{A}_0$ such that, for every problem instance $\mathcal{T}$ with exactly k goals at level h,*
$\mathrm{avg}_{\pi \in \Pi(\mathcal{T})}\mathrm{probe\_cost}(\mathcal{A}_0, \pi) = O(D_{0,h-1}/k)$.

The next theorem gives a generalization of Theorem 11. It should be noted that our analysis presented here sacrifices comprehensiveness for brevity; it is possible to tighten the analysis to better exploit the situation where the degrees on many successive levels are all one (giving rise to subtrees whose number of leaves is far exceeded by their number of internal nodes).

Theorem 16. *For every deterministic instance-informed algorithm $\mathcal{A}$, and every problem instance $\mathcal{T}$ with exactly k_t goals at level t,*
$\mathrm{avg}_{\pi \in \Pi(\mathcal{T})}\mathrm{search\_cost}(\mathcal{A}, \pi) = \Omega(\min_{t \geq 0}\{t + D_{0,t-1}/k_t\})$.

Next, we give a generalization of Theorem 12. We begin by describing algorithm $\mathcal{A}_3$, the general tree variant of binary tree search algorithm $\mathcal{A}_2$. We dovetail, as in Theorem 12, but in rounds that are partitioned into sub-rounds. Let $\sigma_r = \sum_{0 \leq j \leq r} D_{0,j-1}$, the total number of nodes of $\mathcal{T}$ on levels 0 through r. After round $r \geq 0$, the tree $\mathcal{T}$ has been completely searched up to level r, at a cost of σ_r. In addition, for $0 \leq j < \lg D_{0,r-1}$, $D_{0,r-1}/2^j$ nodes on all levels in the interval $(r-1+\sigma_r/D_{0,r-1}2^j, r-1+\sigma_r/D_{0,r-1}2^{j+1}]$ have been searched, at an additional total cost of $\sigma_r \lg D_{0,r-1}$.

More generally, after sub-round s of round r, $s+1$ of the d_{r-1} children of each node on level $r-1$ have been probed, at a cost of $\sigma_{r-1} + (s+1)D_{0,r-2}$. In addition, for $0 \leq j < \lg((s+1)D_{0,r-2})$, $(s+1)D_{0,r-2}/2^j$ nodes on all levels in the interval $(r-2+(\sigma_{r-1}+(s+1)D_{0,r-2})/((s+1)D_{0,r-2})2^j, r-2+(\sigma_{r-1}+(s+1)D_{0,r-2})/((s+1)D_{0,r-2})2^{j+1}]$ have been searched, at an additional total cost of $(\sigma_{r-1}+(s+1)D_{0,r-2}) \lg((s+1)D_{0,r-2})$.

Theorem 17. *The uninformed algorithm $\mathcal{A}_3$ has the property that, for every problem instance $\mathcal{T}$ with exactly k_t goals at level t, $\text{avg}_{\pi \in \Pi(\mathcal{T})}\text{search\_cost}(\mathcal{A}_3, \pi)$*
$= O(\min_{t \geq 0}\{(t+1)\frac{D_{0,t-1}}{k_t}\} \cdot \lg(\min_{t \geq 0}\{(t+1)\frac{D_{0,t-1}}{k_t}\}))$.

Contrasting Theorems 16 and 17, we obtain competitive bounds comparable to those achieved in the case of binary trees; of course, the competitive limit captured by Theorem 13 still applies.

Acknowledgements

References

1. Alpern, S., Gal, S.: The Theory of Search Games and Rendezvous. Kluwer Academic Publishers, Dordrecht (2003)
2. Azar, Y., Broder, A.Z., Manasse, M.S.: On-line choice of on-line algorithms. In: Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 432–440 (1993)
3. Baeza-Yates, R.A., Culberson, J.C., Rawlins, G.J.E.: Searching in the plane. Inform. Comput. 106(2), 234–252 (1993)
4. Demaine, E., Fekete, S., Gal, S.: Online searching with turn cost. Theoret. Comput. Sci. 361, 342–355 (2006)
5. Fleischer, R., Kamphans, T., Klein, R., Langetepe, E., Trippen, G.: Competitive online approximation of the optimal search ratio. In: Proc. 12th Annual European Symposium on Algorithms, pp. 335–346 (2004)
6. Kao, M.-Y., Littman, M.L.: Algorithms for informed cows. In: AAAI 1997 Workshop on On-Line Search (1997)
7. Kao, M.-Y., Ma, Y., Sipser, M., Yin, Y.: Optimal constructions of hybrid algorithms. J. Algorithms 29(1), 142–164 (1998)
8. Kao, M.-Y., Reif, J.H., Tate, S.R.: Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. Inform. Comput. 131(1), 63–79 (1996)

9. Kenyon, C.: Best-fit bin-packing with random order. In: Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 359–364 (1996)
10. Kirkpatrick, D.: Hyperbolic dovetailing. In: Proc. 17th Annual European Symposium on Algorithms, pp. 516–527 (2009)
11. Koutsoupias, E., Papadimitriou, C., Yannakakis, M.: Searching a fixed graph. In: Proc. 23rd International Colloquium on Automata, Languages and Programming, pp. 280–289 (1996)
12. Lopez-Ortiz, A., Schuierer, S.: The ultimate strategy to search on ï rays. Theoret. Comput. Sci. 261, 267–295 (2001)
13. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. In: Proc. Second Israel Symposium on Theory of Computing and Systems, Jerusalem, pp. 128–133 (June 1993)
14. McGregor, A., Onak, K., Panigrahy, R.: The oil searching problem. In: Proc. 17th Annual European Symposium on Algorithms, pp. 504–515 (2009)
15. Papadimitriou, C.H., Yannakakis, M.: Shortest paths without a map. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 610–620. Springer, Heidelberg (1989)
16. Schönhage, A.: Adaptive raising strategies optimizing relative efficiency. In: Proc. 30th International Colloquium on Automata, Languages and Programming, pp. 611–623 (2003)
17. Schuierer, S.: Lower bounds in on-line geometric searching. Comp. Geom. 18, 37–53 (2001)
18. Schuierer, S.: A lower bound for randomized searching on ï rays. In: Klein, R., Six, H.-W., Wegner, L. (eds.) Computer Science in Perspective. LNCS, vol. 2598, pp. 264–277. Springer, Heidelberg (2003)
19. `http://www2.cs.uregina.ca/~zilles/kirkpatrickZ11b.pdf`

Multiple-Source Single-Sink Maximum Flow in Directed Planar Graphs in $O(\text{diameter} \cdot n \log n)$ Time

Philip N. Klein[*] and Shay Mozes[**]

Brown University, Providence, RI, USA
{klein,shay}@cs.brown.edu

Abstract. We develop a new technique for computing maximum flow in directed planar graphs with multiple sources and a single sink that significantly deviates from previously known techniques for flow problems. This gives rise to an $O(\text{diameter} \cdot n \log n)$ algorithm for the problem.

1   Introduction

The study of maximum flow in planar graphs has a long history. In 1956, Ford and Fulkerson introduced the max st-flow problem, gave a generic augmenting-path algorithm, and also gave a particular augmenting-path algorithm for the case of a planar graph where s and t are on the same face (that face is traditionally designated to be the infinite face). Researchers have since published many algorithmic results proving running-time bounds on max st-flow for (a) planar graphs where s and t are on the same face, (b) undirected planar graphs where s and t are arbitrary, and (c) directed planar graphs where s and t are arbitrary. The best bounds known are (a) $O(n)$ [5], (b) $O(n \log \log n)$ [6], and (c) $O(n \log n)$ [2], where n is the number of nodes in the graph.

   This paper is concerned with the maximum flow problem in the presence of multiple sources. In max-flow applied to general graphs, multiple sources presents no problem: one can reduce the problem to the single-source case by introducing an artificial source and connecting it to all the sources. However, as Miller and Naor [10] pointed out, this reduction violates planarity unless all the sources are on the same face to begin with. Miller and Naor raise the question of computing a maximum flow in a planar graph with multiple sources and multiple sinks. Until recently, the best known algorithm for computing multiple-source max-flow in a planar graph is to use the reduction in conjunction with a max-flow algorithm for general graphs. That is, no planarity-exploiting algorithm was known for the problem. A few months after developing the technique described in this paper we developed with collaborators an algorithm for the more general problem of maximum flow in planar graphs with multiple sources and sinks [3] which runs in $O(n \log^3 n)$ time and uses a recursive approach. Given these recent

---

developments, the algorithm presented here is mostly interesting for the new technique developed.

In this paper we present an alternative algorithm for the maximum flow problem with multiple sources and a single sink in planar graphs that runs in $O(\text{diameter} \cdot n \log n)$ time. The diameter of a graph is defined as the maximum over all pairs of nodes of the minimum number of edges in a path connecting the pair. Essentially, we start with a non-feasible flow that dominates a maximum flow, and convert it into a feasible maximum preflow by eliminating negative-length cycles in the dual graph. The main algorithmic tool is a modification of an algorithm of Klein [9] for finding multiple-source shortest paths in planar graphs with nonnegative lengths; our modification identifies and eliminates negative-length cycles. This approach is significantly different than all previously known maximum-flow algorithms. While the relation between flow in the primal graph and shortest paths in the dual graph has been used in many algorithms for planar flow, considering fundamentally non-feasible flows and handling negative cycles is novel. We believe that this is an interesting algorithmic technique and are hopeful it will be useful beyond the current context.

1.1   Applications

Schrijver [12] has written about the history of the maximum-flow problem. Ford and Fulkerson, who worked at RAND, were apparently motivated by a classified memo of Harris and Ross on interdiction of the Soviet railroad system. That memo, which was declassified, contains a diagram of a planar network that models the Soviet railroad system and has multiple sources and a single sink.

A more realistic motivation comes from selecting multiple nonoverlapping regions in a planar structure. Consider, for example, the following image-segmentation problem. A grid is given in which each vertex represents a pixel, and edges connect orthogonally adjacent pixels. Each edge is assigned a cost such that the edge between two similar pixels has higher cost than that between two very different pixels. In addition, each pixel is assigned a weight. High weight reflects a high likelihood that the pixel belongs to the foreground; a low-weight pixel is more likely to belong to the background.

The goal is to find a partition of the pixels into foreground and background to minimize the sum

> weight of *background* pixels
>
>   + cost of edges between *foreground* pixels and *background* pixels

subject to the constraints that, for each component K of foreground pixels, the boundary of K forms a closed curve in the planar dual that surrounds all of K (essentially that the component is simply connected).

This problem can be reduced to multiple-source, single-sink max-flow in a planar graph (in fact, essentially the grid). For each pixel vertex v, a new vertex v', designated a source, is introduced and connected only to v. Then the sink is connected to the pixels at the outer boundary of the grid. See [4] for similar applications in computer vision.

1.2   Related Work

Most of the algorithms for computing maximum flow in general (i.e., non-planar) graphs build a maximum flow by starting from the zero flow and iteratively pushing flow without violating arc capacities. Traditional augmenting path algorithms, as well as more modern blocking flow algorithms, push flow from the source to the sink at each iteration, thus maintaining a feasible flow (i.e., a flow assignment that respects capacities and obeys conservation at non-terminals) at all times. Push-relabel algorithms relax the conservation requirement and maintain a feasible preflow rather than a feasible flow. However, none of these algorithms maintains a flow assignment that violates arc capacities.

There are algorithms for maximum flow in planar graphs that do use flow assignments that violate capacities [11,10]. However, these violations are not fundamental in the sense that the flow does respect capacities up to a circulation (a flow with no sources or sinks). In other words, the flow may over-saturate some arcs, but no cut is over-saturated. We call such flows *quasi-feasible* flows.

The value of a flow that does over-saturate some cuts is higher than that of a maximum flow. This situation can be identified by detecting a negative-length cycle in the dual of the residual graph. One of the algorithms in [10] uses this property in a parametric search for the value of the maximum flow. When a quasi-feasible flow with maximum value is found, it is converted into a feasible one. This approach is not suitable for dealing with multiple sources because the size of the search space grows exponentially with the number of sources.

Our approach is the first to use and handle fundamentally infeasible flows. Instead of interpreting the existence of negative cycles as a witness that a given flow should not be used to obtain a maximum feasible flow, we use the negative cycles to direct us in transforming a fundamentally non-feasible flow into a maximum feasible flow. A negative-length cycle whose length is $-c$ corresponds to a cut that is over saturated by c units of flow. This implies that the flow should be decreased by pushing c units of flow back from the sink across that cut.

2   Preliminaries

In this section we provide basic definitions and notions that are useful in presenting the algorithm. Additional definitions and known facts that are relevant to the proof of correctness and to the analysis are presented later on.

We assume the reader is familiar with the basic definitions of planar embedded graphs and their duals (cf. [2]). Let $G = \langle V, A \rangle$ be a planar embedded graph with node-set V and arc-set A. For notational simplicity, we assume here and henceforth that G is connected and has no parallel edges and no self-loops. For each arc a in the arc-set A, we define two oppositely directed darts, one in the same orientation as a (which we sometimes identify with a) and one in the opposite orientation. We define rev($\cdot$) to be the function that takes each dart to the corresponding dart in the opposite direction. It is notationally convenient to equate the edges, arcs and darts of G with the edges, arcs and darts of the dual G^*. It is well-known that contracting an edge that is not a self-loop corresponds
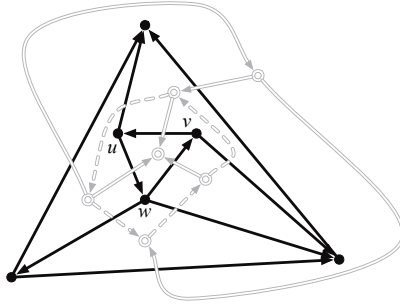
Fig. 1. A primal graph (black) and its dual (gray). The cut $\boldsymbol{\delta}(\{u, v, w\})$ in the primal corresponds to a counterclockwise dual cycle (dashed arcs).

to deleting it in the dual, and that a set of darts forms a simple directed cycle in G iff it forms a simple directed cut in the dual G^* [13]; see Fig. 1.

Given a length assignment length$(\cdot)$ on the darts, we extend it to sets D of darts by length$(D) = \sum_{d \in D}$ length(d). For a set X of nodes, let $\boldsymbol{\delta}(X)$ denote the set of darts crossing the cut $(X, V - X)$. Namely, $\boldsymbol{\delta}(X) = \{d : \text{tail}(d) \in X, \text{head}(d) \notin X\}$. Let T be a rooted spanning tree of G^*. For a node $v \in G^*$, let $T[v]$ denote the unique root-to-v path in T. The *reduced length of d with respect to T* is defined by

$$\text{length}_T(d) = \text{length}(d) + \text{length}(T[\text{tail}_{G^*}(d)]) - \text{length}(T[\text{head}_{G^*}(d)]) \quad (1)$$

The edges of G not in T form a spanning tree τ of G. A dart d is *unrelaxed* if $\text{length}_T(d) < 0$. Note that, by definition, only darts not in T can be unrelaxed. A *leafmost unrelaxed* dart is an unrelaxed dart d of τ such that no proper descendant of d in τ is unrelaxed. For a dart d not in T, the *elementary cycle of d with respect to T in G^** is the cycle composed of d and the unique path in T between the endpoints of d.

2.1   Flow

Let $S \subset V$ be a set vertices called sources, and let $t \notin S$ be vertex called sink.

A *flow assignment* $f(\cdot)$ in G is a real-valued function on the darts of G satisfying *antisymmetry*:

$$f(\text{rev}(d)) = -f(d)$$

A *capacity assignment* $c(\cdot)$ is a real-valued function on darts. A flow assignment $f(\cdot)$ is *feasible* or *respects capacities* if, for every dart d, $f(d) \leq c(d)$. Note that, by antisymmetry, $f(d) \leq c(d)$ implies $f(\text{rev}(d)) \geq -c(d)$. Thus a negative capacity on a dart acts as a lower bound on the flow on the reverse dart.

For a given flow assignment $f(\cdot)$, the *net inflow* (or just *inflow*) node v is $\text{inflow}_f(v) = \sum_{d \in A:\text{head}(d)=v} f(d)$.[1] The *outflow* of v is $\text{outflow}_f(v) = -\text{inflow}_f(v)$.

---

[1] An equivalent definition, in terms of arcs, is $\text{inflow}_f(v) = \sum_{a \in A:\text{head}(a)=v} f(a) - \sum_{a \in A:\text{tail}(a)=v} f(a)$.

The *value* of $f(\cdot)$ is the inflow at the sink, inflow$_f(t)$. A flow assignment $f(\cdot)$ is said to *obey conservation* at node v if inflow$_f(v) = 0$. A flow assignment is a *circulation* if it obeys conservation at all nodes. A flow assignment is a *flow* if it obeys conservation at every node other than the sources and sink. It is a *preflow* if for every node other than the sources, inflow$_f(v) \geq 0$.

For two flow assignments f, f', the *addition* $f + f'$ is the flow that assigns $f(d) + f'(d)$ to every dart d. A flow assignment f is a *quasi-feasible flow* if there exists a circulation ϕ such that $f + \phi$ is a feasible flow. This concept is not new, but it is so central to our algorithm that we introduce a name for it.

The *residual graph* of G with respect to a flow assignment $f(\cdot)$ is the graph G_f with the same arc-set, node-set and sink, and with capacity assignment $c_f(\cdot)$ defined as follows. For every dart d, $c_f(d) = c(d) - f(d)$.

Given a feasible preflow f_+ in a planar graph, there exists an $O(n \log n)$-time algorithm that converts f_+ into a feasible flow f with the same value (cf. [7]). In fact, this can be done in linear time by first canceling flow cycles using the technique of Kaplan and Nussbaum [8], and then by sending any excess flow from back to the sources in topological sort order.

2.2   Quasi-Feasible Flows and Negative-Length Dual Cycles

Miller and Naor prove that f is a quasi-feasible flow in G if and only if G_f^* contains no negative-length cycles. Intuitively, a negative-length cycle in the dual corresponds to a primal cut whose residual capacity is negative. That is, the corresponding cut is over-saturated. Since a circulation obeys conservation at all nodes it does not change the total flow across any cut. Therefore, there exists no circulation whose addition to f would make it feasible. Conversely, they show that if G_f^* has no negative-length cycles, then shortest path distances from any arbitrary node in the dual define a feasible circulation in the primal.

3   The Algorithm

We describe an algorithm that, given a graph G with n nodes, a sink t incident to the infinite face f_∞, and multiple sources, computes a maximum flow from the sources to t in time $O(\text{diameter} \cdot n \log n)$, where diameter is the diameter of the face-vertex incidence graph of G. Initially, each dart d has a non-negative capacity, which we denote by length(d) since we will interpret it as a length in the dual. During the execution of the algorithm, the length assignment length$(\cdot)$ is modified. Even though the algorithm does not explicitly maintain a flow at all times, we will refer throughout the paper to the flow pushed by the algorithm. At any given point in the execution of the algorithm we can interpret the lengths of darts in the dual as their residual capacities in the primal. By the flow pushed by the algorithm, we mean the flow that would induce these residual capacities.

The algorithm starts by pushing an infeasible flow that saturates $\delta(s)$ from every source s to t (Line 4). It then starts to reduce that flow in order to make it feasible. This is done by using a spanning tree τ of G and a spanning tree

---

Algorithm 1. Multiple-source single-sink maximum flow (G, S, t, c_0)

Input: planar directed graph G with capacities c_0, source set S, sink t incident to f_∞
Output: a maximum feasible flow f

1: length$(d) := c_0(d)$ for every dart d
2: initialize spanning tree T of G^* rooted at f_∞ using right-first-search
3: let τ be the spanning tree of G consisting of edges not in T, and root τ at t
4: **for** each source $s \in S$
5:    **for** each dart d on the s-to-t path in τ
6:       length$(d) :=$ length$(d) -$ length$(\delta(\{s\}))$
7:       length$(\text{rev}(d)) :=$ length$(\text{rev}(d)) +$ length$(\delta(\{s\}))$
8: **while** there exist unrelaxed darts in G^*
9:    let $\hat{d}$ be an unrelaxed dart that is leafmost in τ
10:   **if** $\hat{d}$ is not a back-edge in T **then**   //*perform a pivot*
11:      remove from T the parent edge of head$_{G^*}(\hat{d})$ and insert $\hat{d}$ into T
12:   **else**   // *fix a negative cycle by pushing back flow*
13:      let C denote the elementary cycle of $\hat{d}$ with respect to T in G^*
14:      **for** each dart d of the $\hat{d}$-to-t path of darts in the primal spanning tree τ

$$\text{length}(d) := \text{length}(d) + |\text{length}(C)|$$

$$\text{length}(\text{rev}(d)) := \text{length}(\text{rev}(d)) - |\text{length}(C)|$$

15:      **for** every dart d strictly enclosed by C
16:         $f(d) := c_0(d) - \text{length}_T(d)$
17:         in G, contract d //*in G^*, delete d*
18:   $f(d) := c_0(d) - \text{length}_T(d)$ for every dart d
19: convert the preflow f into a flow.

T of G^* such that each edge is in exactly one of these trees. The algorithm repeatedly identifies a negative cycle C in G^*, which corresponds in G to an over-saturated cut. Line 14 decreases the lengths of darts on a primal path in τ that starts at the sink t and ends at some node v (a face in G^*) that is enclosed by C. We call such a path a *pushback path*. This change corresponds to pushing flow back from the sink to v along the pushback path, making the over-saturated cut exactly saturated. We call the negative-turned-zero-length cycle C a *processed cycle*. Processed cycles enclose no negative cycles. This implies that there exists a feasible preflow that saturates the cut corresponding to a processed cycle (see Lemma 5). The algorithm records that preflow (Line 16) and contracts the source-side of the cut into a single node referred to as a *super-source*.

When no negative length cycles are left in the contracted graph, the flow pushed by the algorithm is quasi-feasible. That is, the flow pushed by the algorithm is equivalent, up to a circulation, to a feasible flow in the contracted graph. Combining this feasible flow in the contracted graph and the preflows recorded at the times cycles were processed yields a maximum feasible preflow for the original uncontracted graph. In a final step, this feasible preflow is converted into a feasible flow.

We now describe in more detail how negative cycles are identified and processed by the algorithm. The algorithm maintains a spanning tree T of G^* rooted at the infinite face f_∞ of G, and a spanning tree τ of G, rooted at the sink t. The tree τ consists of the edges not in T. The algorithm tries to transform T into a shortest-path tree by pivoting into T unrelaxed darts according to some particular order (line 9). However, if an unrelaxed dart $\hat{d}$ happens to be a back-edge[2] in T, the corresponding elementary cycle C is a negative-length cycle. To process C, flow is pushed from t to head($\hat{d}$) along the t-to-$\hat{d}$ path in τ (line 14). The amount of flow the algorithm pushes is $|\text{length}(C)|$, so after C is processed its length is zero, and $\hat{d}$ is no longer unrelaxed. The algorithm then records a feasible preflow for C, deletes the interior of C, and proceeds to find the next unrelaxed dart. See Fig. 2 for an illustration. When all darts are relaxed, T is a shortest-path tree, which implies no more negative-length cycles exist.
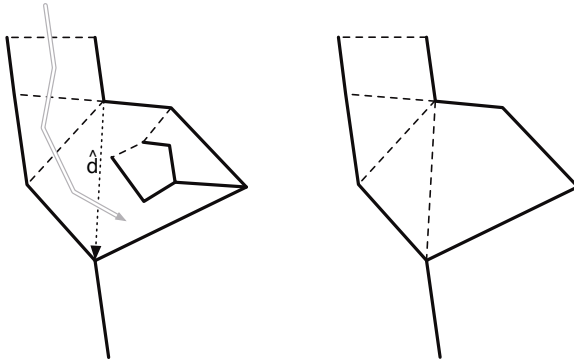
Fig. 2. Identifying and processing a negative cycle. On the left, the tree T is shown in solid. Non-tree edges are dashed. The unrelaxed dart $\hat{d}$ is dotted. The elementary cycle of $\hat{d}$ w.r.t. T has negative length. The primal pushback path is double-lined gray. After the negative cycle is processed, its interior is deleted (on the right).

To control the number of pivots we initialize T to have a property called *right-shortness*, and show it is preserved by the algorithm and that it implies that the number of pivots of any dart is bounded by the diameter of the graph. We later discuss how data structures enable the iterations to be performed efficiently.

4   Correctness and Analysis

We refer the reader to the full version of this paper[3] for complete proofs of some of the lemmas in this section as well as for the precise definitions of the following (standard) terms that are used in the sequel: clockwise and counterclockwise, being left-of and right-of, reduced lengths, and winding numbers. We begin with

---

[2] A non-tree dart d is a back edge of T if head(d) is an ancestor in T of tail(d).
[3] Available on the authors' website and at http://arxiv.org/abs/1104.4728

a couple of basic lemmas. Consider the sink t as the infinite face of G^*. By our conventions, any clockwise (counterclockwise) cycle C in G^* corresponds to a primal cut $(X, V - X)$ such that $t \in X$ ($t \notin X$); see Fig. 1.

Lemma 1. *Consider the sink t as the infinite face of G^*. The length of any clockwise (counterclockwise) dual cycle does not decrease (increase) when flow is pushed to t. The length of any clockwise (counterclockwise) dual cycle does not increase (decrease) when flow is pushed from t.*

The following is a restatement of a theorem of Miller and Naor [10].

Lemma 2. *Let G be a planar graph. Let c_0 be a capacity function on the darts of G. Let f be a flow assignment. Define the length of a dart d to be its residual capacity $c_0(d) - f(d)$. If f is quasi-feasible then $f'(d) = c_0(d) - length_T(d)$ is a feasible flow assignment, where T is a shortest-path tree in G^*. Furthermore, $f' = f + \phi$ for some circulation ϕ.*

Correctness. To prove the correctness of the algorithm, we first show that an elementary cycle w.r.t. a back-edge is indeed a negative-length cycle.

Lemma 3. *Let $\hat{d}$ be an unrelaxed back-edge w.r.t. T. The length of the elementary cycle of $\hat{d}$ w.r.t. T is negative.*

Proof. Consider the price function induced by from-root distances in T. Every tree dart whose tail is closer to the root than its head has zero reduced length. The reduced length of an unrelaxed dart is negative. Since $\hat{d}$ is a back edge w.r.t. T, its elementary cycle C uses darts of T whose length is zero. Therefore the reduced length of C equals the reduced length of just $\hat{d}$, which is negative. The lemma follows since the length and reduced length of any cycle are the same.

Lemma 4. *Let C be the negative-length cycle defined in line 13. After C is processed in line 14, $length(C) = 0$ and $\hat{d}$ is relaxed. Furthermore, the following invariants hold just before line 14 is executed:*

1. *the flow pushed by the algorithm satisfies flow conservation at every node other than the sources (including super-sources) and the sink.*
2. *the outflow at the sources is non-negative.*
3. *there are no clockwise negative-length cycles.*

The proof is omitted. The idea is that initially the invariants hold since flow is pushed from the sources to the sink in the initialization step. Subsequently, the invariants are maintained since whenever a negative-length cycle is processed, flow is pushed back into that cycle to make the corresponding cut exactly saturated. Then the interior of the cycle is deleted.

We now prove properties of the flow computed by the algorithm. The following lemma characterizes the flow recorded in line 16. Intuitively, this shows that it is a saturating feasible flow for the cut corresponding to the processed cycle.

Lemma 5. *Let C be a cycle currently being processed. Let $(X, V - X)$ be the corresponding cut, where $t \notin X$. Let $\boldsymbol{\delta}_c$ be the set of darts crossing the cut. The flow assignment f computed in the loop in Line 16 satisfies:*

1. *$f(d) \leq c_0(d)$ for all darts whose endpoints are both in X.*
2. *every node in X except sources and tails of darts of $\boldsymbol{\delta}_c$ satisfies conservation.*
3. *for every $d' \in \boldsymbol{\delta}_c$, $\sum_{d:head(d)=tail(d')} f(d) \geq \sum_{d \in \delta_c : tail(d) = tail(d')} c_0(d)$*

The proof is omitted. The main idea is that at the time C is processed it encloses no negative-length cycles. Furthermore, the tree T is a shortest-path tree for the interior of C at that time. Since the length of the cycle is adjusted to be zero, the corresponding cut is exactly saturated, so by the relation of quasi-feasible flows, the distances in T define a feasible flow in which that cut is saturated.

Lemma 6. *The following invariant holds. In G^* every source is enclosed by some zero-length cycle that encloses no negative-length cycles.*

The proof is omitted. The idea is that the initialization guarantees the invariant holds initially. It is preserved since a processed cycle has length zero and its interior is deleted and replaces with a super-source.   The following lemma, whose proof is omitted, is an easy consequence of Lemma 6.

Lemma 7. *The following invariant holds. There exists no feasible flow f' s.t. $inflow_{f'}(t)$ is greater than $inflow_f(t)$, where f is the flow pushed by the algorithm.*

Lemma 8. *The flow f computed in Line 18 is a maximum feasible flow in the contracted graph G w.r.t. the capacities c_0.*

Proof. The flow pushed by the algorithm satisfies conservation by Lemma 4. By Lemma 7, there is no feasible flow of greater value. Since there are no unrelaxed darts, T is a shortest-path tree in G^* and G^* contains no negative cycles. Therefore, the flow pushed by the algorithm is quasi-feasible. This shows that the conditions of Lemma 2 are satisfied. It follows that f computed in Line 18 satisfies conservation, respects the capacities c_0 and has maximum value.

Lemma 9. *The flow assignment $f(d)$ is a feasible maximum preflow in the (uncontracted) graph G w.r.t. the capacities c_0.*

Proof. f is well defined since each dart is assigned a value exactly once; in Line 16 at the time it is contracted, or in Line 18 if it was never contracted. By Lemma 8 and by part 1 of Lemma 5, $f(d) \leq c_0(d)$ for all darts d, which shows feasibility. By Lemma 8 and by part 2 of Lemma 5, flow is conserved everywhere except at the sources, the sink, and nodes that are tails of darts of processed cycles. However, for a node v that is the tail of some dart of a processed cycle, $\sum_{d:head(d)=v} f(d) \geq 0$. This is true by part 3 of Lemma 5, and since $f(d) \leq c_0(d)$ for any dart. $f(\cdot)$ is therefore a preflow. Finally, the value of $f(\cdot)$ is maximum by Lemma 8.

Lemma 9 completes the proof of correctness since line 19 converts the maximum feasible preflow into a feasible flow of the same value.

Efficient implementation. The dual tree T is represented by a table parentD[·] that, for each nonroot node v, stores the dart parentD[v] of T whose head in G^* is v. The primal tree τ is represented using a dynamic-tree data structure such as self-adjusting top-trees [1]. Each node and each edge of τ is represented by a node in the top-tree. Each node of the top-tree that represents an edge e of τ has two weights, $w_L(e)$ and $w_R(e)$. The values of these weights are the reduced lengths of the two darts of e, the one oriented towards leaves and the one oriented towards the root.[4] The weights are represented so as to support an operation that, given a node x of the top-tree and an amount Δ, adds Δ to $w_R(e)$ and subtracts Δ from $w_L(e)$ for all edges e in the x-to-root path in τ.

This representation allows each of the following operations be implemented in $O(\log n)$ time: lines 9, 13, 11, 16, and 17, the loop of line 5, and the loop of line 14.[5]

Running-Time Analysis. Lines 16 and 17 are executed at most once per edge. To analyze the running time it therefore suffices to bound the number of pivots in line 11 and the number of negative cycles encountered by the algorithm. Note that every negative length cycle strictly encloses at least one edge. This is because the length of any cycle that encloses just a single source is initially set to zero, and since the length of the cycle that encloses just a single super-source is set to zero when the corresponding negative cycle is processed and contracted.

Since the edges strictly enclosed by a processed cycle are deleted, the number of processed cycles is bounded by the number of edges, which is $O(n)$.

It remains to bound the number of pivots. We will prove that at the tree T satisfies a property called *right-shortness*. This property implies that the number of times a given dart pivots into T is bounded by the diameter of the graph.

Definition 1. *[9] A tree T is right-short if for all nodes $v \in T$ there is no simple root-to-v path P that is: as short as $T[v]$ and strictly right of $T[v]$.*

Since T is initialized using right first search, initially, for every node v there is no simple path in G^* that is strictly right of $T[v]$. Therefore, initially, T is right-short. The algorithm changes T in two ways; either by making a pivot (line 11) or by processing a negative-length cycle (line 14).

Lemma 10. *[9] leafmost dart relaxation (line 11) preserves right-shortness.*

Lemma 11. *Right-shortness is preserved when processing the counterclockwise negative-length cycle C in line 14.*

The proof is omitted. The main idea is that the changes in lengths of darts in line 14 correspond to pushing back flow from the sink t. By Lemma 1, the length of any counterclockwise cycle does not decrease. Since lengths of darts of T are not affected by line 14, this implies that right-shortness is preserved. We have thus established that

---

[4] Note that the length of the the cycle C in line 13 is exactly the reduced length of $\hat{d}$.

[5] The whole initialization in the loop of line 4 can instead be carried out in linear time by working up from the leaves towards the root of τ.

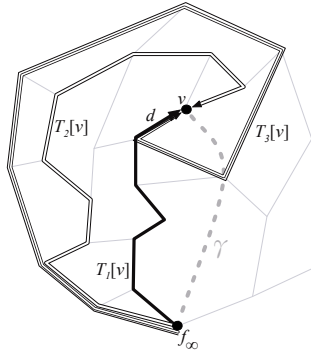**Corollary 1.** *T is right-short at all times.*

Fig. 3. Bounding the number of pivots of dart d whose head is a node v of G^*. Three tree paths to v at times $t_1 < t_2 < t_3$ are shown. $T_3[v]$ (triple-lined) is left of $T_2[v]$ (double-lined) and $T_2[v]$ is left of $T_1[v]$ (black) . The dart d is in $T_1[v]$ and $T_3[v]$, but not in $T_2[v]$. Therefore, it must have pivoted out of T and into T between time t_1 and t_3. The curve γ (dashed) visits only nodes of G^*. The winding number of $T_1[v]$ about γ is 0. The winding number of $T_3[v]$ must be greater than that of T_1 (1 in this example).

Consider a dart d of G^* with head v. Let γ be an arbitrary root(T)-to-v curve on the sphere. Let $T_1[v]$ and $T_2[v]$ denote the tree path to v at two distinct times in the execution of the algorithm. Since right-shortness is preserved throughout the algorithm and since with every pivot length$(T[v])$ may only decrease, if $T_2[v]$ occurs later in the execution than $T_1[v]$, then $T_2[v]$ is left of $T_1[v]$. It follows that the winding number of $T[v]$ about γ between any two occurrences of d as the pivot dart in line 11 must increase. See Fig. 3. Therefore, if we let T_0 denote the initial tree T, and T_t denote the tree T at the latest time node v appears in G^*, then the number of times d may appear as the pivot dart in line 11 is bounded by the difference of the winding numbers of $T_t[v]$ and $T_0[v]$ about γ. Since γ is arbitrary, we may choose it to intersect the embedding of G^* only at nodes, and to further require that it visit the minimum possible number of nodes of G^*. This number is bounded by the diameter of the face-vertex incidence graph of G, which is bounded by the minimum of the diameter of G and the diameter of G^*. Since both T_0 and T are simple, the absolute value of their winding number about γ is trivially bounded by the number of nodes γ visits. Therefore, the total number of pivots is bounded by $\sum_d \text{diameter} = O(\text{diameter} \cdot n)$. The total running time of the algorithm is thus bounded by $O(\text{diameter} \cdot n \log n)$.

References

1. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Maintaining information in fully dynamic trees with top trees. ACM Transactions on Algorithms 1(2), 243–264 (2005)

2. Borradaile, G., Klein, P.N.: An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. Journal of the ACM 56(2) (2009)
3. Borradaile, G., Klein, P.N., Mozes, S., Nussbaum, Y., Wulff-Nilsen, C.: Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time (2011) (submitted)
4. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. IEEE Transactions on Pattern Analysis and Machine Intelligence 26, 1124–1137 (2004)
5. Henzinger, M.R., Klein, P.N., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. Journal of Computer and System Sciences 55(1), 3–23 (1997)
6. Italiano, G.F., Nussbaum, Y., Sankowski, P., Wulff-Nilsen, C.: Improved algorithms for min cut and max flow in undirected planar graphs. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (to appear, 2011)
7. Johnson, D.B., Venkatesan, S.: Using divide and conquer to find flows in directed planar networks in $O(n^{3/2} \log n)$ time. In: Proceedings of the 20th Annual Allerton Conference on Communication, Control, and Computing, pp. 898–905 (1982)
8. Kaplan, H., Nussbaum, Y.: Maximum flow in directed planar graphs with vertex capacities. In: Proceedings of the 17th European Symposium on Algorithms, pp. 397–407 (2009)
9. Klein, P.N.: Multiple-source shortest paths in planar graphs. In: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 146–155 (2005)
10. Miller, G.L., Naor, J.: Flow in planar graphs with multiple sources and sinks. SIAM Journal on Computing 24(5), 1002–1017 (1995); preliminary version in FOCS 1989
11. Reif, J.: Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. SIAM Journal on Computing 12, 71–81 (1983)
12. Schrijver, A.: On the history of the transportation and maximum flow problems. Mathematical Programming 91(3), 437–445 (2002)
13. Whitney, H.: Planar graphs. Fundamenta Mathematicae 21, 73–84 (1933)

Planar Subgraphs without Low-Degree Nodes

Evangelos Kranakis[1], Oscar Morales Ponce[1], and Jukka Suomela[2]

[1] School of Computer Science, Carleton University, Ottawa, Canada
kranakis@scs.carleton.ca, omponce@connect.carleton.ca
[2] Helsinki Institute for Information Technology HIIT, University of Helsinki, Finland
jukka.suomela@cs.helsinki.fi

Abstract. We study the following problem: given a geometric graph $\mathcal{G}$ and an integer k, determine if $\mathcal{G}$ has a planar spanning subgraph (with the original embedding and straight-line edges) such that all nodes have degree at least k. If $\mathcal{G}$ is a unit disk graph, the problem is trivial to solve for $k = 1$. We show that even the slightest deviation from the trivial case (e.g., quasi unit disk graphs or $k = 2$) leads to NP-hard problems.

1 Introduction

We study the problem of finding planar subgraphs that do not have low-degree nodes. More precisely, given a geometric graph $\mathcal{G} = (V, E)$ and an integer k, we want to determine if $\mathcal{G}$ has a *planar spanning subgraph* with the original embedding and straight-line edges such that *all nodes have degree at least k*.

This is a natural prerequisite in many problems related to communication networks. For example, if the answer is *no*, then certainly we cannot find a k-vertex connected or k-edge connected planar spanning subgraph, either. Moreover, if the answer is *no*, then we know that the domatic number (the maximum number of disjoint dominating sets) of any planar spanning subgraph is at most k. On the positive side, if the answer is *yes* for $k = 1$, then we can find a planar spanning subgraph that has domatic number at least 2. That is, we can partition the nodes in two disjoint dominating sets; for example, in a monitoring application, the two sets of nodes can take turns in order to conserve energy.

The problem is easy to solve if $\mathcal{G}$ is a complete graph and $k \leq 2$: any triangulation of V is a planar graph of degree at least 2, and hence the answer is *yes* iff we have at least $k + 1$ nodes. The case of complete graphs and $k = 3$ requires more thought, but it turns out that there is a planar spanning subgraph of minimum degree 3 iff we have at least 4 nodes and they are not in a convex position [2,11]. Hence the problem can be solved in polynomial time for complete graphs and $k \leq 3$. On the other hand, for $k \geq 6$ the answer is always *no*: any planar graph contains a node of degree at most 5.

The problem is also easy to solve if $\mathcal{G}$ is a connected unit disk graph (see below for the definitions) and $k = 1$: the answer is *yes* if there are at least two nodes, since the Euclidean minimum spanning tree in $\mathcal{G}$ is planar. In this work we investigate what happens if we slightly deviate from the trivial case of a unit disk graph and $k = 1$. In particular, can we solve the problem efficiently in quasi

unit disk graphs [3], which are relaxations of unit disk graphs? Or can we solve the problem in other simple families of geometric graphs such as graphs with orthogonal edges? And what happens if $k = 2$ or $k = 3$? Surprisingly, it turns out that even the slightest deviations lead to NP-complete decision problems.

Related Work. To our knowledge, this problem has not been studied before, but there are related problems that have been considered in prior work.

In a *non-geometric* setting (i.e., the embedding of the nodes is not fixed), finding a planar spanning subgraph is trivial: any spanning tree is planar. However, finding *large* planar subgraphs is hard. For example, deciding if there is a planar subgraph with a certain number of edges is a classical NP-complete problem [5, problem GT27]. We refer to the survey by Liebers [8] for many variants of the theme.

Our focus is on a *geometric* setting: the input is a graph with a fixed embedding of the nodes, edges are straight lines, and a subgraph is planar if its edges do not cross each other. With these definitions, finding a planar spanning subgraph is no longer trivial: for example, an arbitrary spanning tree may have crossing edges. However, in *unit disk graphs* the problem is easy to solve: the Euclidean minimum spanning tree is planar. Indeed, a spanning tree is also connected, and many algorithms have been proposed to obtain a *connected* planar spanning subgraph of a unit disk graph – see, e.g., Gabriel and Sokal [4] and Toussaint [12]. However, none of these algorithms guarantee that the minimum degree is greater than one, and more general settings such as quasi unit disk graphs have not been extensively studied.

Another closely related problem is *connectivity augmentation* in which new edges are added to a graph to increase the connectivity. Abellanas et al. [1], Rutter and Wolff [9], and Tóth [10] consider geometric planar graphs and add edges to obtain 2-edge connected planar graphs, and Tóth and Valtr [11] and Al-Jubeh et al. [2] consider the problem of augmenting a geometric planar graph into a 3-edge connected planar graph. However, in these results the newly added edges are of unbounded length. Recently, Kranakis et al. [6] proved that a geometric planar graph can be augmented to a 2-edge connected planar graph with edges of length at most three times the largest edge in the graph.

Notation and Preliminaries. A *geometric graph* (straight line graph) $\mathcal{G} = (V, E)$ is a graph where each $v \in V$ is a point in the Euclidean plane and each $e \in E$ is a straight line segment joining two distinct points. Throughout this paper we assume that the points are in general position (no three points being collinear). A geometric graph is *planar* if its edges do not cross each other except at their endpoints, and it is *orthogonal* if each edge is either horizontal or vertical. A graph is *k-vertex connected* if it remains connected after the removal of any $k-1$ vertices, and it is *k-edge connected* if it remains connected after the removal of any $k - 1$ edges. The degree of node v in graph $\mathcal{G}$ is denoted by $\deg(\mathcal{G}, v)$.

We define the family qUDG(s) of *quasi unit disk graphs* with parameter $s \leq 1$ as follows. A geometric graph $\mathcal{G} = (V, E)$ is in qUDG(s) if the following holds for all $u, v \in V$: if the distance between u and v is at most s, then there is an edge

Table 1. Computational complexity of finding a planar spanning subgraph that has a given minimum degree in different families of geometric graphs. NP-C = NP-complete.

Degree	Complete graphs	UDG	qUDG$(1-\epsilon)$	Orthogonal graphs	Reference
≥ 1	in P	in P	NP-C if $\epsilon > 0$	NP-C	§3
≥ 2	in P	NP-C	NP-C if $\epsilon \geq 0$	(open)	§4
≥ 3	in P [11]	NP-C	NP-C if $\epsilon \geq 0$	(open)	§5

Table 2. Computational complexity of the augmentation problem

Degree	Graph family	Complexity	Reference
≥ 1	UDG	in P	
≥ 2	UDG	NP-C if $\alpha < \sqrt{5}/2$	§4
	UDG	in P if $\alpha \geq 3$	[6]
	UDG, strip of height h	in P if $\alpha \geq \sqrt{1+h^2}$	§6
≥ 3	UDG	NP-C if $\alpha < \sqrt{5}/2$	§5

$\{u, v\} \in E$, and if the distance between u and v is larger than 1, then there is no edge between u and v. The family UDG of *unit disk graphs* is equal to qUDG(1); in a unit disk graph, the locations of the nodes determine the set of edges.

If $\mathcal{G} \in$ UDG and $\alpha \geq 1$ is a real number, then we define the geometric αth power of a geometric graph $\mathcal{G}$, denoted by $\mathcal{G}^{(\alpha)}$, as the graph obtained from $\mathcal{G}$ by adding all edges between vertices of Euclidean distance at most α.

Contributions. Our results are summarized in Tables 1 and 2.

In Section 3 we study the case of minimum degree 1. While the problem is trivial to solve in UDG, we show that it is NP-complete in qUDG$(1-\epsilon)$ for any positive constant ϵ. We also show that the problem is NP-complete in orthogonal graphs. The case of $k = 2$ is investigated in Section 4, and the case of $k = 3$ in Section 5. In both cases it turns out that the problem is NP-complete in UDG.

As the strict decision problem turns out to be NP-complete in UDG if $k \in \{2, 3\}$, it is natural to ask whether we can solve the *augmentation* version of the problem with parameter $\alpha \geq 1$: either (i) prove that a given $\mathcal{G} \in$ UDG does not have a planar spanning subgraph of minimum degree k, or (ii) prove that $\mathcal{G}^{(\alpha)}$ has a planar subgraph of minimum degree k. The case of $\alpha = 1$ is equivalent to the original decision problem. Our results in Sections 4 and 5 show that the case of $k \in \{2, 3\}$ and $\alpha < \sqrt{5}/2$ is NP-hard, while prior work [6] implies that the case of $k = 2$ and $\alpha = 3$ can be solved in polynomial time. In Section 6 we present an algorithm that solves the augmentation problem in *narrow strips* of height h in polynomial time for $k = 2$ and $\alpha = \sqrt{1+h^2}$.

The main new technique that we introduce in this work is the concept of *planar circuit networks*. In Section 2 we show that the problem of choosing an *orientation* of such a network is NP-complete, and Sections 3–5 demonstrate

that the orientation problem on planar circuit networks serves as a useful starting point in NP-completeness proofs that are related to planar subgraphs.

2   Orientation Problem on Planar Circuit Networks

A *planar circuit network* is a planar geometric graph $\mathcal{C} = (V_\mathcal{C}, E_\mathcal{C})$ with the following properties:

(i) The node set $V_\mathcal{C} = T_\mathcal{C} \cup S_\mathcal{C} \cup U_\mathcal{C}$ consists of three disjoint subsets: *terminals* $T_\mathcal{C}$, *switches* $S_\mathcal{C}$, and *users* $U_\mathcal{C}$.

(ii) The edge set $E_\mathcal{C} = P_\mathcal{C} \cup W_\mathcal{C}$ consists of two disjoint subsets: *ports* $P_\mathcal{C}$ and *wires* $W_\mathcal{C}$. Each port is *labeled* with either 1, 2, or 3; the set of ports with label x is denoted by $P_\mathcal{C}(x)$.

(iii) Each wire joins a pair of terminals. Each port joins a terminal and a non-terminal.

(iv) Each terminal is incident to exactly 2 edges, and at least one of them is a wire.

(v) Each switch or user is incident to exactly 3 edges, and all of them are ports with different labels. That is, for each $v \in S_\mathcal{C} \cup U_\mathcal{C}$ and $x \in \{1, 2, 3\}$, there is exactly one edge in $P_\mathcal{C}(x)$ that is incident to v.

Refer to Fig. 1 for an illustration; we draw terminals as black dots, switches as trapezoids with the port 1 on the short side, and users as squares. In what follows, we use the notation $v(x)$ to refer to the unique port with label $x \in \{1, 2, 3\}$ that is incident to $v \in S_\mathcal{C} \cup U_\mathcal{C}$.

We can partition a circuit network into *components* that are connected to each other by terminals. More precisely, a component consists of (i) a wire and two terminals or (ii) a user or a switch, three ports, and three terminals. Note that each terminal is contained in exactly two components. However, if a is a wire, switch, or user, then there is a unique component $\mathcal{C}[a]$ that contains a; in that case, we use the notation $a \rightarrow\!\bullet\, t$ to denote that t is a terminal in the component $\mathcal{C}[a]$. We write $a \xrightarrow{x}\!\bullet\, t$ if the port that leads from switch or user a to terminal t has label x.

Orientation Problem. An *orientation* of a planar circuit network $\mathcal{C}$ assigns a direction to each edge $e \in E_\mathcal{C}$. If $a \rightarrow\!\bullet\, t$, and t has indegree 1 in the component
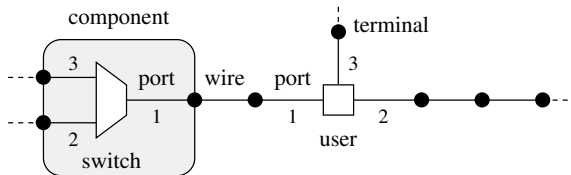
Fig. 1. A circuit network
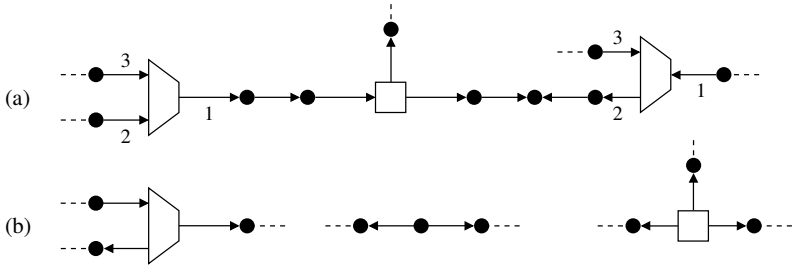
Fig. 2. (a) Valid and (b) invalid orientations of a circuit network

$\mathcal{C}[a]$, then we write $a \rightarrow\!\bullet\, t$; otherwise $a \leftarrow\!\bullet\, t$. If $a \xrightarrow{x}\!\bullet\, t$, we use notation $a \xrightarrow{x}\!\!\!\rightarrow\!\bullet\, t$ or $a \leftarrow\!\!\!\xrightarrow{x}\!\bullet\, t$.

A *valid orientation* of a component $\mathcal{C}[a]$ satisfies the following requirements (the first one is trivially satisfied but is listed here for reference):

(i) If a is a wire, then there is at least one terminal t with $a \leftarrow\!\bullet\, t$.
(ii) If a is a user, then there is at least one terminal t with $a \leftarrow\!\bullet\, t$.
(iii) If a is a switch with $s \xrightarrow{2}\!\!\!\rightarrow\!\bullet\, t_2$ or $s \xrightarrow{3}\!\!\!\rightarrow\!\bullet\, t_3$, then $s \leftarrow\!\!\!\xleftarrow{1}\!\bullet\, t_1$.

A valid orientation of a circuit network $\mathcal{C}$ satisfies the following additional requirement:

(iv) Each terminal has indegree at least one.

See Fig. 2 for an illustration. Not all circuit networks have valid orientations; in the *orientation problem* the task is to decide if a given planar circuit network $\mathcal{C}$ has a valid orientation.

The following lemma shows that we can replace a path of wires by a single wire and vice versa, without affecting the essential properties of the problem.

Lemma 1. *Let $\mathcal{C}$ be a planar circuit network, and let $e = \{u, v\} \in W_{\mathcal{C}}$ be a wire in $\mathcal{C}$. Construct another planar circuit network $\mathcal{C}'$ by repeatedly subdividing e; that is, we replace e by a path P that consists of wires and terminals. Then if we are given a valid orientation of $\mathcal{C}$, we can find in polynomial time a valid orientation of $\mathcal{C}'$, and vice versa.*

Proof. Clearly if we are given a valid orientation of $\mathcal{C}$, we can construct a valid orientation of $\mathcal{C}'$ as well. Now assume that we are given a valid orientation of $\mathcal{C}'$. Since all internal nodes of P have indegree at least 1, there must be at least one endpoint of P that has indegree 0. W.l.o.g., we can assume that u is an endpoint with indegree 0. Then we can orient the wire e in $\mathcal{C}$ from u to v. The orientations of all other edges are inherited from $\mathcal{C}'$.   □

We will use the following theorem in reductions throughout this work.

Theorem 1. *The orientation problem on planar circuit networks is NP-complete.*

Proof. By a reduction from planar 3SAT; see the extended version of this work [7].

3   Degree One

In this section we study the problem of deciding if a given geometric graph has a planar spanning subgraph. The problem is trivial in UDG – a minimum spanning tree is a planar spanning subgraph. We will show that the problem is NP-complete in quasi unit disk graphs and in orthogonal graphs.

Theorem 2. *The following problem is NP-complete for any $\epsilon > 0$: given a graph $\mathcal{G} \in \mathsf{qUDG}(1 - \epsilon)$, decide if $\mathcal{G}$ has a planar spanning subgraph.*

To prove the theorem, assume that we are given an $\epsilon > 0$ and a planar circuit network $\mathcal{C}$. We will show how to construct a graph $\mathcal{G} \in \mathsf{qUDG}(1 - \epsilon)$ with the following property: $\mathcal{G}$ has a planar spanning subgraph if and only if $\mathcal{C}$ has a valid orientation. The claim then follows by Theorem 1.

In our construction of $\mathcal{G}$, we replace each component $\mathcal{C}[a]$ of $\mathcal{C}$ by a gadget $\mathcal{G}[a]$ that *implements* the component. In the construction, we will have two kinds of nodes in $\mathcal{G}$: *black nodes* are identified with the terminals of $\mathcal{C}$, while *grey nodes* are internal to a gadget. A component with a wire is replaced with the gadget of Fig. 3a, a component with a switch is replaced with the gadget of Fig. 3b, and a component with a switch is replaced with the gadget of Fig. 3c. Note that each black node is shared by exactly two gadgets.

To guarantee that the gadgets are in $\mathsf{qUDG}(1 - \epsilon)$, we must choose appropriate dimensions. Moreover, two gadgets must not be placed too close to each other; for example, an internal node of one gadget cannot be within distance $1 - \epsilon$ from the internal node of another gadget. Hence we cannot directly replace the components by gadgets in an arbitrary embedding of $\mathcal{C}$. However, we can always find an appropriate embedding by moving the components around and by exploiting the flexibility provided by Lemma 1; see Fig. 3d for an illustration
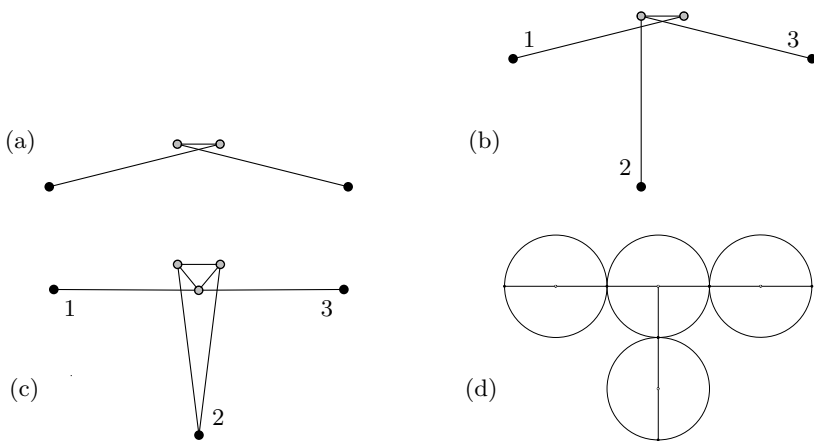
Fig. 3. Gadgets for $\mathsf{qUDG}(1 - \epsilon)$ and minimum degree 1: (a) wire, (b) switch, (c) user. (d) Embedding of a switch, three ports, and three wires

of the embedding in the neighborhood of a switch. Note that we can use a polynomially-sized grid to embed the nodes, as we have some flexibility in the choice of the exact coordinates: for example, in the wire gadget, the distance between the two terminals can be chosen from the range $(2 - \epsilon, 2)$.

Now we proceed to relate the orientations of $\mathcal{C}$ and the planar spanning subgraphs of $\mathcal{G}$. Let us first focus on a component $\mathcal{C}[a]$ and the gadget $\mathcal{G}[a]$ that implements it. We say that a subgraph $\mathcal{G}'[a]$ of $\mathcal{G}[a]$ is *internally good* if $\mathcal{G}'[a]$ is planar and each grey node of $\mathcal{G}[a]$ has degree at least one in $\mathcal{G}'[a]$. Note that if we have a planar spanning subgraph $\mathcal{G}'$ of $\mathcal{G}$, then $\mathcal{G}'$ restricted to $\mathcal{G}[a]$ is internally good. The key observation is summarized in the following lemma.

Lemma 2. *Given a valid orientation of $\mathcal{C}[a]$, we can find an internally good subgraph $\mathcal{G}'[a]$ such that $a \rightarrowtail t$ implies $\deg(\mathcal{G}'[a], t) \geq 1$. Conversely, given an internally good subgraph $\mathcal{G}'[a]$, we can find a valid orientation of $\mathcal{C}[a]$ such that $a \rightarrow\bullet\ t$ and $\deg(\mathcal{G}'[a], t) \geq 1$ implies $a \rightarrowtail t$.*

Proof. A straightforward case analysis. □

Now if we are given a valid orientation of $\mathcal{C}$, we can apply Lemma 2 to each component $\mathcal{C}[a]$ to find an internally good subgraph $\mathcal{G}'[a]$ for each $\mathcal{G}[a]$; the union of $\mathcal{G}'[a]$ forms a subgraph $\mathcal{G}'$ of $\mathcal{G}$. By construction, $\mathcal{G}'$ is planar and each grey node has degree at least one. It remains to be shown that each black node has degree at least one. To verify this, consider a terminal t. In a valid orientation, there is a component $\mathcal{C}[a]$ such that $a \rightarrowtail t$. Hence in $\mathcal{G}'[a]$ we have $\deg(\mathcal{G}'[a], t) \geq 1$, and therefore also $\deg(\mathcal{G}', t) \geq 1$. Hence $\mathcal{G}'$ is a planar spanning subgraph of $\mathcal{G}$.

Conversely, if we are given a planar spanning subgraph $\mathcal{G}'$ of $\mathcal{G}$, we can apply Lemma 2 to each component $\mathcal{G}[a]$ to orient $\mathcal{C}$. We will have a valid orientation for each component; it remains to be shown that each terminal has indegree at least one. To verify this, consider a terminal t. Since $\deg(\mathcal{G}', t) \geq 1$ we have a component $\mathcal{C}[a]$ such that $\deg(\mathcal{G}'[a], t) \geq 1$, and hence a valid orientation with $a \rightarrowtail t$. Hence we have a valid orientation of $\mathcal{C}$. This concludes the proof of Theorem 2.

Theorem 3. *The following problem is NP-complete: given an orthogonal graph $\mathcal{G}$, decide if it has a planar spanning subgraph.*

Proof. The structure of the proof is identical to the proof of Theorem 2. We are only using a different set of gadgets: see Fig. 4. □

4   Degree Two

In this section we study the case of planar spanning subgraphs that have degree at least 2. It turns out that finding such subgraphs is NP-hard even in the case of UDG.

In what follows, we prove a stronger result by considering a variant in which we are allowed to *augment* the graph by adding edges of length $\alpha \geq 1$. In the
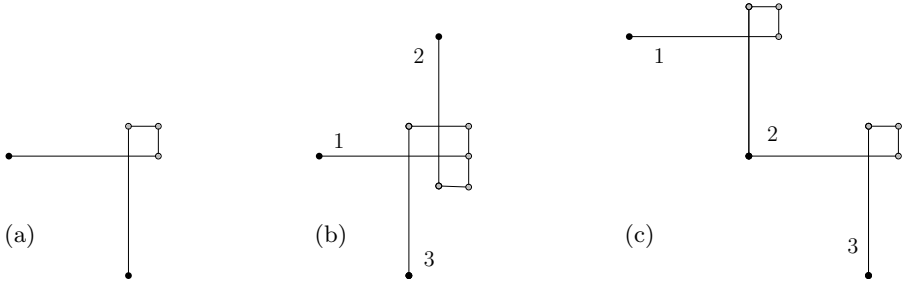
Fig. 4. Gadgets for orthogonal graphs: (a) wire, (b) switch, (c) user

augmentation problem we are allowed to return the answer that *"the original graph does not have a planar spanning subgraph with minimum degree at least 2"* or *"I do not know about the original graph, but if I first add some edges of length α, I can construct a planar spanning subgraph with minimum degree at least 2".* Obviously, the case of $\alpha = 1$ is equivalent to the original decision problem.

If $\alpha = 3$, the problem can be solved in polynomial time by using techniques from prior work [6]: If we are given a graph $\mathcal{G} \in \mathsf{UDG}$, we can consider each connected component of $\mathcal{G}$ separately. If a connected component contains fewer than 3 nodes, then we know that $\mathcal{G}$ does not have a planar subgraph with minimum degree 2. Otherwise we can first find a spanning tree in each component, and then augment the trees by adding non-crossing edges of length at most 3 so that each connected component becomes 2-edge connected. In particular, after augmentation, the graph is planar and each node has minimum degree at least 2.

In what follows, we prove that the problem is NP-complete if $\alpha < \sqrt{5}/2 \approx$ 1.118. We do this by formulating the problem as a promise problem.

Theorem 4. *The following promise problem is NP-complete for any $1 \leq \alpha < \sqrt{5}/2$: given a graph $\mathcal{G} \in \mathsf{UDG}$, decide whether (i) there is no planar spanning subgraph of $\mathcal{G}$ such that each node has degree at least 2, or (ii) there is a planar spanning subgraph of $\mathcal{G}^{(\alpha)}$ such that each node has degree at least 2.*

Proof. The structure of the proof is similar to the proof of Theorem 2. For the sake of brevity, we only list the differences.

In Theorem 2 we constructed a graph $\mathcal{G} \in \mathsf{qUDG}(1 - \epsilon)$; this time we construct a graph $\mathcal{G} \in \mathsf{UDG}$. Moreover, our construction satisfies $\mathcal{G} = \mathcal{G}^{(\alpha)}$, that is, the augmentation does not change the graph at all, and hence the augmentation problem is exactly as difficult as deciding if $\mathcal{G}$ has a planar spanning subgraph with minimum degree at least two.

We use a new set of gadgets; see Fig. 5. In addition to the gadgets that correspond to the components of the circuit network, we also have a gadget for each terminal. It can be verified that the distance between any pair of non-adjacent nodes within a gadget is larger than α, and that there is an embedding that preserves this property.
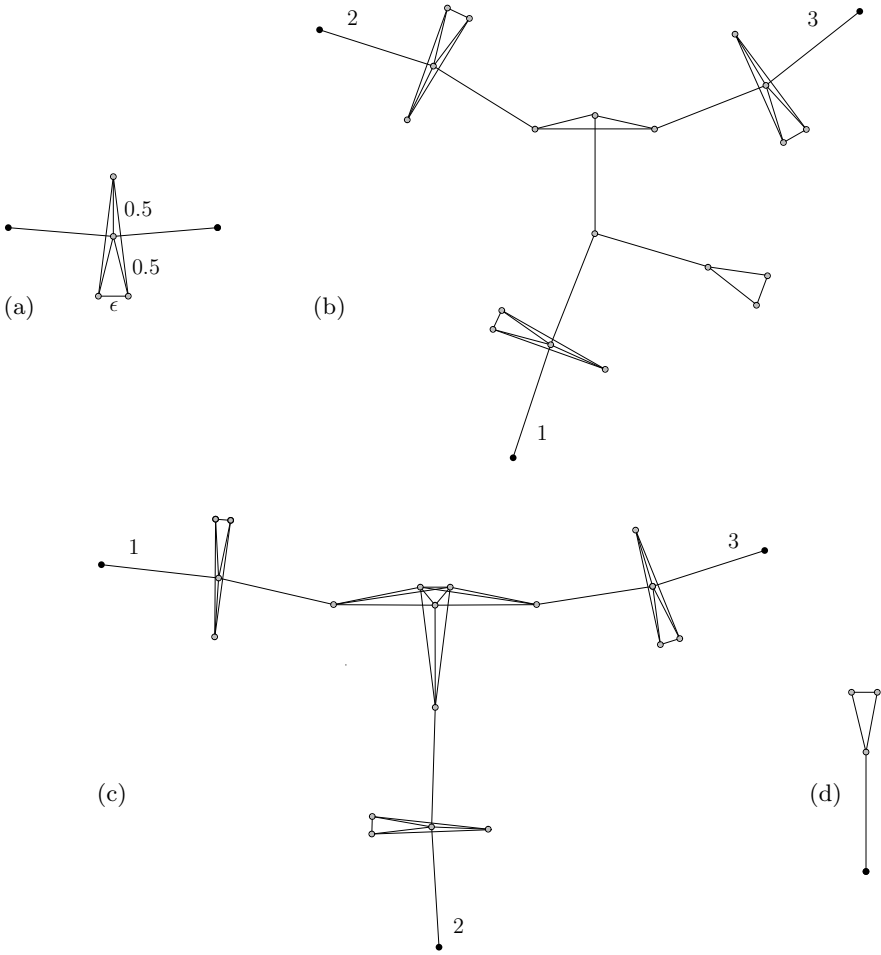
Fig. 5. Gadgets for UDG, minimum degree 2: (a) wire, (b) switch, (c) user, (d) terminal

As we are finding a subgraph of minimum degree 2, we change the definition of an internally good subgraph accordingly: each grey node must have degree at least 2. With this change, Lemma 2 holds verbatim.

The key difference with the proof of Theorem 2 is that we have to deal with the terminal gadgets. If we are given a valid orientation of $\mathcal{C}$, we can apply Lemma 2 to each component $\mathcal{C}[a]$ to find an internally good subgraph $\mathcal{G}'[a]$ for each $\mathcal{G}[a]$. The union of the subgraphs $\mathcal{G}'[a]$ and *all terminal gadgets* forms a subgraph $\mathcal{G}'$ of $\mathcal{G}$. By construction, $\mathcal{G}'$ is planar and each grey node has degree at least 2. Now consider a terminal t. In a valid orientation, there is a component $\mathcal{C}[a]$ such that $a \rightarrowtail t$, and we have $\deg(\mathcal{G}'[a], t) \geq 1$. Furthermore, t is incident to exactly one edge from the terminal gadget. In summary, $\deg(\mathcal{G}', t) \geq 2$. Hence $\mathcal{G}'$ is a planar spanning subgraph of $\mathcal{G}$ and all nodes have degree at least 2.

Conversely, given a planar spanning subgraph $\mathcal{G}'$ with minimum degree 2, we can verify that we can construct a valid orientation of $\mathcal{C}$: each terminal gadget contributes only one edge, and hence for each terminal t we must have $a \rightarrowtail t$ such that $\deg(\mathcal{G}'[a], t) \geq 1$. This concludes the proof.   □

5   Degree Three

In this section, we consider the case of planar spanning subgraphs with minimum degree at least $k = 3$. This case turns out to be similar to that of $k = 2$ in Section 4. Again, we can prove that the problem is NP-complete, and even the augmentation version of the problem is NP-complete.

Theorem 5. *The following promise problem is NP-complete for any $1 \leq \alpha < \sqrt{5}/2$: given a graph $\mathcal{G} \in \mathsf{UDG}$, decide whether (i) there is no planar spanning subgraph of $\mathcal{G}$ such that each node has degree at least 2, or (ii) there is a planar spanning subgraph of $\mathcal{G}^{(\alpha)}$ such that each node has degree at least 2.*

Proof. The proof is similar to Theorem 2. The key differences are as follows: We have a new set of gadgets, see Fig. 6. We do not have any terminal gadgets. The definition of an internally good subgraph is changed accordingly: each grey node has degree at least 3. Finally, Lemma 2 is modified as follows.

Lemma 3. *Given a valid orientation of $\mathcal{C}[a]$, we can find an internally good subgraph $\mathcal{G}'[a]$ such that $a \rightarrow\bullet t$ implies $\deg(\mathcal{G}'[a], t) \geq 1$ and $a \rightarrowtail t$ implies $\deg(\mathcal{G}'[a], t)$*
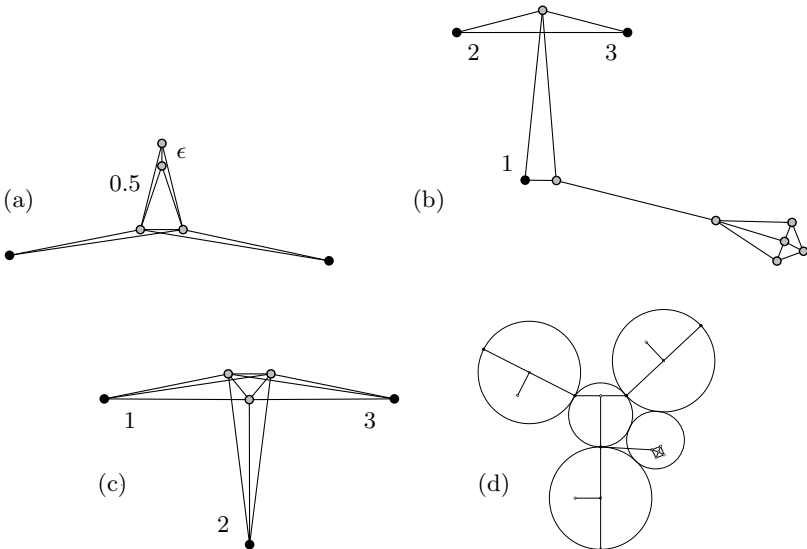
Fig. 6. Gadgets for UDG and minimum degree 3: (a) wire, (b) switch, (c) user. (d) Embedding of a switch and three wires.

≥ 2. *Conversely, given an internally good subgraph $\mathcal{G}'[a]$, we can find a valid orientation of $\mathcal{C}[a]$ such that $a \rightarrow\!\bullet\, t$ and $\deg(\mathcal{G}'[a], t) \geq 2$ implies $a \rightarrow\!\!\bullet\, t$.*

Now if we are given a valid orientation of $\mathcal{C}$, we can apply Lemma 3 to each component in order to construct a planar subgraph $\mathcal{G}'$ of $\mathcal{G}$. For each terminal t there are $a \rightarrow\!\!\bullet\, t$ and $b \rightarrow\!\bullet\, t$ with $a \neq b$. Now $\deg(\mathcal{G}', t) = \deg(\mathcal{G}'[a], t) + \deg(\mathcal{G}'[b], t) \geq 2 + 1 = 3$. The converse case is similar. $\qquad\square$

6 Augmenting with Bounded Length Edges

As we saw in Section 4, the augmentation problem for the case of minimum degree $k = 2$ is NP-complete in UDG if we are allowed to augment by adding edges of length $\alpha < \sqrt{5}/2$, and it can be solved in polynomial time if $\alpha = 3$. In this section we present an algorithm that solves the augmentation problem for $\alpha = \sqrt{1 + h^2}$ in narrow strips of height h.

Theorem 6. *The following problem can be solved in polynomial time: given a graph $\mathcal{G} \in \mathsf{UDG}$ in which all nodes are inside a strip of height h, either (i) show that there is no planar spanning subgraph of $\mathcal{G}$ such that each node has degree at least 2, or (ii) find a planar spanning subgraph of $\mathcal{G}^{(\alpha)}$ for $\alpha = \sqrt{1 + h^2}$ such that each node has degree at least 2.*

Proof. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the nodes of $\mathcal{G}$ ordered by their x-coordinates, and let x_i be the x-coordinate of node v_i; that is, we have $x_i \leq x_{i+1}$ for all $i < n$. We say that $V_{s,t} = \{v_s, v_{s+1}, \ldots, v_t\}$ is a *section* if

- $x_i + 1 \geq x_{i+1}$ for all $s \leq i < t$,
- $s = 1$ or $x_{s-1} + 1 < x_s$, and
- $t = n$ or $x_t + 1 < x_{t+1}$.

Put otherwise, a section is a maximal set of nodes such that the x coordinates are separated by at most 1 unit. Note that the subgraph $\mathcal{G}_{s,t}$ induced by section $V_{s,t}$ consists of one or more connected components of $\mathcal{G}$.

Now consider each subgraph $\mathcal{G}_{s,t}$ one by one; we will either conclude that we have case (i), or we will show how find a planar subgraph $\mathcal{G}'_{s,t}$ of $\mathcal{G}^{(\alpha)}_{s,t}$:

- If $t - s \leq 5$, we have a constant-size subproblem that we can solve by brute force: either decide that we have case (i), or construct a planar subgraph $\mathcal{G}'_{s,t}$.
- If $x_s + 1 < x_{s+2}$, then the degree of v_s in $\mathcal{G}$ is at most one, and we have case (i). Similarly, if $x_{t-2} + 1 < x_t$, then the degree of v_t in $\mathcal{G}$ is at most one, and we have case (i).
- Otherwise we can construct $\mathcal{G}'_{s,t}$ as follows: the edge set of $\mathcal{G}'_{s,t}$ consists of $\{v_s, v_{s+2}\}$, $\{v_{t-2}, v_t\}$, and $\{v_i, v_{i+1}\}$ for all $s \leq i < t$; each of these has length at most α, and they are nonintersecting.

If we do not have case (i), we can find a planar subgraph of $\mathcal{G}^{(\alpha)}$ as a union of the subgraphs $\mathcal{G}'_{s,t}$. $\qquad\square$

7 Conclusions

In this work we have studied the problem of finding a planar spanning subgraph with minimum degree k in different families of geometric graphs. One of the main discoveries is the existence of a very sharp threshold in the computational complexity of such problems: the case of $k = 1$ and unit disk graphs is trivial, while a minor deviation from $k = 1$ to $k = 2$, or from unit disk graphs to quasi unit disk graphs makes the problem NP-complete. A major open problem is a full characterization of the complexity of the augmentation problem: for which values of the parameter α can the problem be solved in polynomial time?

References

1. Abellanas, M., García, A., Hurtado, F., Tejel, J., Urrutia, J.: Augmenting the connectivity of geometric graphs. Computational Geometry: Theory and Applications 40(3), 220–230 (2008)
2. Al-Jubeh, M., Ishaque, M., Rédei, K., Souvaine, D.L., Tóth, C.D.: Tri-edge-connectivity augmentation for planar straight line graphs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 902–912. Springer, Heidelberg (2009)
3. Barrière, L., Fraigniaud, P., Narayanan, L., Opatrny, J.: Robust position-based routing in wireless ad hoc networks with irregular transmission ranges. Wireless Communications and Mobile Computing 3(2), 141–153 (2003)
4. Gabriel, K.R., Sokal, R.R.: A new statistical approach to geographic variation analysis. Systematic Zoology 18(3), 259–278 (1969)
5. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. WH Freeman and Company, New York (1979)
6. Kranakis, E., Krizanc, D., Ponce, O.M., Stacho, L.: Bounded length, 2-edge augmentation of geometric planar graphs. In: Wu, W., Daescu, O. (eds.) COCOA 2010, Part I. LNCS, vol. 6508, pp. 385–397. Springer, Heidelberg (2010)
7. Kranakis, E., Morales Ponce, O., Suomela, J.: Planar subgraphs without low-degree notes (2011), `http://www.iki.fi/jukka.suomela/low-degree`
8. Liebers, A.: Planarizing graphs—a survey and annotated bibliography. Journal of Graph Algorithms and Applications 5(1), 1–74 (2001)
9. Rutter, I., Wolff, A.: Augmenting the connectivity of planar and geometric graphs. Electronic Notes in Discrete Mathematics 31, 53–56 (2008)
10. Tóth, C.D.: Connectivity augmentation in plane straight line graphs. Electronic Notes in Discrete Mathematics 31, 49–52 (2008)
11. Tóth, C.D., Valtr, P.: Augmenting the edge connectivity of planar straight line graphs to three. In: Proc. 13th Spanish Meeting on Computational Geometry, EGC 2009 (2009)
12. Toussaint, G.T.: The relative neighbourhood graph of a finite planar set. Pattern Recognition 12(4), 261–268 (1980)

Constructing Orthogonal de Bruijn Sequences

Yaw-Ling Lin[1,*], Charles Ward[2], Bharat Jain[2], and Steven Skiena[2,**]

[1] Dept. of Comp. Sci. and Info. Eng., Providence University
yllin@pu.edu.tw
[2] Dept. of Comp. Sci., Stony Brook University
charlesbward@gmail.com, {bkjain,skiena}@cs.sunysb.edu

Abstract. A (σ, k)-de Bruijn sequence is a minimum length string on an alphabet set of size σ which contains all σ^k k-mers exactly once. Motivated by an application in synthetic biology, we say a given collection of de Bruijn sequences are *orthogonal* if no two of them contain the same $(k + 1)$-mer; that is, the length of their longest common substring is k.

In this paper, we show how to construct large collections of orthogonal de Bruijn sequences. In particular, we prove that there are at least $\lfloor \sigma/2 \rfloor$ mutually-orthogonal order-k de Bruijn sequences on alphabets of size σ for all k. Based on this approach, we present a heuristic which proves capable of efficiently constructing optimal collections of mutually-orthogonal sequences for small values of σ and k, which supports our conjecture that $\sigma - 1$ mutually-orthogonal de Bruijn sequences exist for all σ and k.

Keywords: de Bruijn graphs, de Bruijn sequences, orthogonal sequences, Eulerian cycles, DNA synthesis.

1   Introduction

New technologies create new questions about classical objects. Here we introduce a natural class of combinatorial sequence design problems in response to rapid advances in DNA synthesis technology.

An exciting new field of *synthetic biology* is emerging with the goal of designing novel organisms at the genetic level. DNA sequencing technology can be thought of as reading DNA molecules, so as to describe them as strings on $\{A, C, G, T\}$ for computational analysis. DNA *synthesis* is the inverse operation, where one can take any desired string and construct DNA molecules to specification with exactly the given sequence. Indeed, commercial vendors such as GeneArt and Blue Heron today charge under 40 cents per base, or only a few thousand dollars to synthesize virus-length sequences, and prices are rapidly dropping [4,6]. In May 2010, Venter's group announced the first synthetic life form, a feat requiring synthesizing a 1.08 megabase bacterial chromosome [8]. The advent of cheap

synthesis will have many exciting new applications throughout the life sciences (including our own work on synthetic vaccines [5,16,15]): the freedom to design new sequences to specification leads to a variety of new algorithmic problems on sequences.

We are particularly concerned with the problem of designing novel sequences encoding large collections of target patterns. A collaboration with microbiologists Bruce Futcher (Stony Brook University) and Lucas Carey (Weizmann Institute) revealed the need for several distinct sequence designs each containing representative binding sites for a large set of transcription factors. Three distinct objectives of these designs needed to be satisfied:

1. *Minimum Length* – Since synthesis cost increases as a function of length, these sequence designs must be as short as possible.
2. *Variability* – Several equivalent designs were needed, in order to control for the possibility of innoportunely inserting larger signal patterns, and to provide additional confirmatory evidence of the findings.
3. *Orthogonality* – The final goal is that each pair of our designed sequences avoid common (or reverse-complement) sequences to minimize the chances of cross-hybridization. Such molecular interactions could lead to a variety of problems, including blocking the very binding sites the molecules were specifically designed to include.

A particularly relevant class of DNA sequences contain occurrences of all 4^k patterns of length k. Simple concatenation would yield a string of length $k4^k$, but considerable length reduction is possible by overlapping patterns. Indeed, *de Bruijn sequences* [7,11] yield optimal circular strings of length 4^k containing each pattern exactly once. These can be linearized, as in the case of the string $AACAGATCCGCTGGTTA$ containing all 16 two-base nucleotide sequences (as shown in Figure 1(left)). Such compression can have a big impact on synthesis costs: realizing all 6-mers using the concatenation design would require 24,576 bases, as opposed to the linearized de Bruijn sequence of only 4,101 bases. Further, the number of distinct de Bruijn sequence of length (order) k and alphabet σ grow exponentially as a function of k and σ, addressing the second criteria.

In this paper we deal with the third criteria: orthogonality. In particular, we define two order-k de Bruijn sequences as *orthogonal* if they contain no $(k+1)$-mer in common – even though they both *must* contain *all* k-mers in common. A set S of de Bruijn sequences are *mutually orthogonal* if each pair is orthogonal, meaning that no $(k+1)$-mer occurs more than once in S. For example, the following three sequences:

```
AAACAAGCACCAGACGAGTCATACTCCCGCCTAGGCGTGATCTGTAATTTGCTTCGGTTATGGGAA
AAAGAATGAGGATAGTATCGACAGCGGGTGGCATTGTCTACGCTCAACCCTGCCGTTCCACTTTAA
AAATAACTATTACATCACGTAGATGTTTCTTGACCTCGCAGTGCGAAGGGCTGGTCCGGAGCCCAA
```

each contain all 4^3 possible DNA triplets exactly once, but have no 4-base sequence in common.

In this paper we show how to construct large sets of orthogonal de Bruijn sequences. In particular:

- *Small σ* – We prove that a pair of orthogonal de Bruijn sequences exist for all orders $k \geq 1$ for $\sigma \geq 3$, and further that this is the smallest alphabet which permits orthogonal de Bruijn sequences. Similarly, a pair of orthogonal sequences always exist on the related Kautz graphs for $\sigma \geq 4$.
- *Large σ* – Through an algorithmic construction, we prove that there are at least $\lfloor \sigma/2 \rfloor$ mutually orthogonal order-k de Bruijn sequences for all $k \geq 1$. Since there can be at most $\sigma - 1$ mutually orthogonal sequences, this family is at least half as large as optimal. Further our result generalizes to constructing edge-pair disjoint Eulerian cycles of σ-regular directed graphs, and hence can be used to construct large sets of orthogonal sequences of difference types (such as Kautz sequences [13]).

  We conjecture that $\sigma - 1$ mutually-orthogonal de Bruijn sequences exist for all $k \geq 1$, $\sigma \geq 3$. However, significantly improving our result seems intimately connected with a well-known conjecture on Hamiltonian decompositions of line graphs [1], and hence appears difficult.
- *Optimal constructions for fixed σ and k* – The technique employed in our combinatorial result suggests a heuristic procedure for constructing even larger sets of orthogonal sequences. We couple this heuristic with exhaustive search to construct maximal sets of sequences for all reachable k on $\sigma \leq 26$. These constructions support our conjecture that $\sigma - 1$ mutually-orthogonal sequences exist for all k.

  We have made these constructions available at *http://www.algorithm.cs.sunysb.edu/orthogonal*, along with the programs that identified them, where we anticipate they will suffice for all near-term DNA synthesis applications. Indeed, a proposal is now pending to synthesize our three mutually-orthogonal $\sigma = 4$, $k = 6$ sequence designs as plasmids within a host such as yeast or *E. coli*, where they can be easily grown as needed for new experiments without additional costly synthesis. These plasmids will be made available to the research community, where we anticipate they will find a wide variety of applications.
- *Kautz Sequences* – Many of our approaches in de Bruijn graphs can be applied on the Kautz graphs as well. In particular, we find that it is possible to find $\sigma - 1$ orthogonal Kautz Sequences on alphabet size with $\sigma > 3$, for a modest size of k. These results suggest that it is possible to find Hamiltonian decompositions of Kautz graphs with $\sigma > 3$, a very nice property for a network topology. Further, we also discover a simple deduction on counting the number of different Kautz sequences, independent of previous approaches using spectral graph theory [18].
- *Specialized Constructions* – Kása [12] conjectures that there are orthogonal de Bruijn sequences of size $\sigma - 1$ for $\sigma \geq 3, k \geq 2$, that can be constructed from one de Bruijn sequence s and series of morphism operations. We implement tools for constructing orthogonal de Bruijn sequences of this kind for relatively small σ and k.

  In this vein, we propose a conjectured form of orthogonal de Bruijn sequence sets of size $\sigma - 1$. These sets have the property of being uniquely described by a single input sequence. These sequences seem relatively abundant

for small values of σ and k, but grow increasing more scarce as the problem size increases. We show that such sequences exist for σ up to 9 when $k = 2$, and up to $k = 6$ when $\sigma = 4$.

This paper is organized as follows. Section 2 introduces previous work on constructing de Bruijn sequences. Analytical results are presented in Section 3. Our heuristic construction and experimental results are presented in Section 4. Open problems, particularly related to d-orthogonal sequences, are described in Section 5.

2   Preliminaries

The de Bruijn sequences are well studied combinatorial objects. There are a number of useful characterizations for the construction of de Bruijn sequences, including as an Eulerian Tour of the de Bruijn Graph [7,11] and as the output of a feedback shift register [10,17]. We will focus on the former characterization here, and review a number of useful known properties of the de Bruijn sequences.

Definition 1. *A de Bruijn graph of order k on an alphabet of size σ, denoted by $G_{\sigma,k}$, is a directed graph with vertices representing all σ^{k-1} sequences of length $(k-1)$, and σ^k edges joining pairs of vertices whose sequences have the maximum possible overlap (length $k-2$). See, for example, Figure 1(middle).*

The standard method for constructing de Bruijn sequences is to construct an Eulerian cycle of this graph. Every Eulerian cycle on the de Bruijn graph $G_{\sigma,k}$ defines a distinct, minimum length de Bruijn sequence containing all σ^k k-mers exactly once. Note that this Eulerian cycle approach can be applied to any pattern set whose overlap graph contains vertices of equal in- and out-degree.

The de Bruijn graph $G_{\sigma,2}$ is isomorphic to the directed complete graph $\hat{K}_\sigma^*$ (with σ self loops). Every Eulerian cycle of $G_{\sigma,k}$ defines a distinct, minimum length de Bruijn (circular) sequence containing all σ^k k-mers exactly once. We
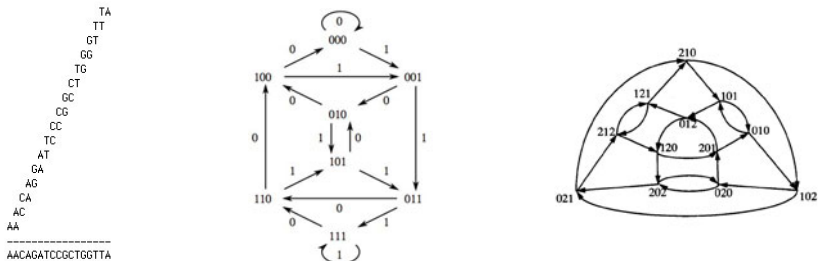
Fig. 1. (left) An order-2, $\sigma = 4$ de Bruijn sequence representing the shortest DNA sequence containing all 16 distinct bi-nucleotide patterns. (middle) The order-4, $\sigma = 2$ de Bruijn graph, Eulerian tours of which correspond to de Bruijn sequences. (right) The order-4, $\sigma = 3$ Kautz graph, discussed in section 2.1.

say a given collection of de Bruijn sequences are *orthogonal* if any two sequences maximally differ in sequence composition; that is, their longest common substring is length k. As an example, the collection $\{001122021, 002211012\}$ is an orthogonal collection of de Bruijn sequences on $G_{3,2}$; while $\{0011223313032021, 0023212203113301, 0033221101312302\}$ is an orthogonal collection of de Bruijn sequences on $G_{4,2}$.

The *directed line graph* $L(G)$ of a directed graph $G = (V, E)$ contains a vertex x for each edge $(u, v) \in G$, and directed edges $(x, y) \in L(G)$ iff $x = (u, v)$ and $y = (v, w)$, where $u, v, w \in V$ and $x, y \in E$.

Observation 2. *$G_{\sigma,k+1}$ is the directed line graph of $G_{\sigma,k}$. Then any Eulerian cycle in $G_{\sigma,k}$ corresponds to a Hamiltonian cycle in $G_{\sigma,k+1}$.*

Note that two orthogonal de Bruijn sequences correspond to two Eulerian cycles in $G_{\sigma,k}$ such that these two cycles do not share any pair of consecutively incident edges. It follows directly that:

Observation 3. *An orthogonal collection of de Bruijn sequences on $G_{\sigma,k}$ corresponds to a set of Eulerian cycles in $G_{\sigma,k}$ such that no two cycles share any common adjacent pair of edges. By the line graph property of Observation 2, these correspond to a set of edge-disjoint Hamiltonian cycles in $G_{\sigma,k+1}$.*

Let us define $\Omega(\sigma, k)$ to be the maximum size of all orthogonal collections of de Bruijn sequences on $G_{\sigma,k}$. Observation 2 tells us that:

Corollary 4. *The size of an orthogonal collection of de Bruijn sequences on $G_{\sigma,k}$ can not exceed $\sigma - 1$. That is, $\Omega(\sigma, k) \leq \sigma - 1$.*

Proof. An orthogonal collection of de Bruijn sequences on $G_{\sigma,k}$ corresponds to a set of edge-disjoint Hamiltonian cycles in $G_{\sigma,k+1}$. However, the minimum in/out degree of any vertex in $G_{\sigma,k+1}$ is $\sigma - 1$ plus a self-loop. As the self-loop cannot contribute towards an additional edge-disjoint cycle, there can be at most $\sigma - 1$ mutually edge-disjoint Hamiltonian cycles on $G_{\sigma,k+1}$, and therefore at most $\sigma - 1$ orthogonal de Bruijn sequences on $G_{\sigma,k}$. ☐

It is conjectured [3,12] that $\Omega(\sigma, k) = \sigma - 1$, for $k \geq 2$. In the case of $k = 1$, note that an orthogonal collection of de Bruijn sequences over $G_{\sigma,1}$, actually corresponds to a Hamiltonian decomposition of the complete directed graph K_σ^* (*without* self-loops). It is known that all K_n^* admit a (directed) Hamiltonian decomposition of size $n - 1$, *except* for the case of K_4^* and K_6^* [1]. These results directly lead to $\Omega(\sigma, 1) = \sigma - 1$ when $\sigma \neq 4$ or 6 ($\Omega(4, 1) = 2$ and $\Omega(6, 1) = 4$). The conjecture remains plausible for the case of $k \geq 2$, and we will later give compelling experimental evidence for this.

Subsequent to submitting this work for publication, we discovered Fleischner and Jackson's [9] result on compatible Eulerian circuits which implies our Theorem 8. Rowley and Bose [20,19] use feedback shift registers [10,17] to obtain $\sigma - 1$ orthogonal sequences when σ is a power of 2, and σ orthogonal sequences on a *modified* de Bruijn graph when σ is the power of a prime.

2.1   Kautz Graphs

A Kautz graph [13] is a labeled graph, very similar to the de Bruijn graph. Like the de Bruijn graph, vertices of $\mathbf{Kz}_{\sigma,k}$ are labeled by strings of length k over an alphabet Σ with σ letters, but with the additional restriction that every two consecutive letters in the string must be different. As in the de Bruijn graph, there is a directed edge from a vertex u to another vertex v if it is possible to transform the string of u into the string of v by removing the first letter and appending a letter to it. See figure 1(right) for an example of a Kautz graph.

For a fixed degree and number of vertices, the Kautz graph has the smallest diameter of any possible directed graph; furthermore, a degree-δ Kautz graph has δ disjoint paths from any node u to any other node v. These properties suggest that the Kautz graph can be a nice candidate of the network topology for connecting processors in interconnection networks [2]. Note that Kautz graphs have in-degree equal to out-degree, both being $\sigma - 1$, for each node. It thus follows that all Kautz graphs are Eulerian.

As with the de Bruijn sequences, a *Kautz sequence* naturally corresponds to a Euler path in the underlying Kautz graph. Thus, a Kautz sequence is a sequence of minimal length that produces all possible length-$(k + 1)$ sequences, but with the restriction that every two consecutive letters in the sequences must be different. A set of three mutually orthogonal Kautz sequences for $\sigma = 4$ and $k = 3$ are:

$$012023030213203123130103232121020213101$$
$$012130202123230132101023103131320303201$$
$$012302320203213132312120101303102103011$$

Observation 5. $\mathbf{Kz}_{\sigma,k+1}$, *is the directed line graph of* $\mathbf{Kz}_{\sigma,k}$. *Thus, an Eulerian cycle in* $\mathbf{Kz}_{\sigma,k}$, *corresponds to a Hamiltonian cycle in* $\mathbf{Kz}_{\sigma,k+1}$.

Therefore, both Kautz and de Bruijn graphs can be considered as families of iterated line graphs. $\mathbf{Kz}_{\sigma,k}$ are obtained from $k-1$ iterated line graph operations on K_σ^*, while $G_{\sigma,k}$ are the $(k-2)$-th iterated line graph from $\hat{K}_\sigma^*$ (with self-loops.)

3   Analytical Results

To be Eulerian, each vertex of a directed graph must have the same in-degree as out-degree. Furthermore, an Euler tour in G corresponds to a pairing of each in-edge to its out-edge for each vertex $v \in G$. Such an edge-pairing defines a *perfect matching* between input edges to output edges of v. We call such an edge-pairing an (edge) *wiring* of v. Two wirings of a vertex are *disjoint* if the corresponding matchings are edge-disjoint. Note that an Euler tour defines a specific wiring for each vertex in G; however, a set of arbitrary wirings for vertices of G usually ends up with several disconnected (edge) cycles. As a restatement of Observation 3, we have

Observation 6. *Two Euler tours in* G *are orthogonal to each other if and only if the two induced wirings for each vertex of* G *are disjoint.*

Kőnig [23] proved that an r-regular bipartite graph can be decomposed into r edge-disjoint perfect matchings, implying that it is hopeful to find an orthogonal collection of Euler tour with size δ in an Eulerian graph with minimum in/out-degree δ. However, it is easily verified that we *cannot* find two orthogonal Euler tours in the directed complete graph K_3^* (without self-loops), whose minimum in/out-degree is $\delta = 2$. In contrast, the directed complete graph $\hat{K}_3^*$ *with* 3 self-loops, has minimum degree 3, but there exist two orthogonal Euler tours in $\hat{K}_3^*$. We now show that there exist at least two orthogonal Euler tours in digraph G whenever $\delta \geq 3$:

Theorem 7. *There exists an orthogonal collection of Euler tours of size 2 in any Eulerian digraph G with minimum degree at least 3. Hence, there exists an orthogonal collection of de Bruijn sequences on $G_{\sigma,k}$ with size 2 if $\sigma \geq 3$. That is, $\Omega(\sigma, k) \geq 2$ if $\sigma \geq 3$*

Proof. Let C be an arbitrary Euler tour of G, defining a specific wiring for each vertex in G. We can rewire each vertex v in G such that the new wiring is disjoint and still forms an Euler tour. Note that the initial wiring of v partition edges of G into δ disjoint nonempty paths, namely $\{P_1, P_2, \ldots, P_\delta\}$, with $C = (P_1 P_2 \cdots P_\delta)$ in circular order. Let a_i (b_i) denote the first (last) edge of P_i. Note that the vertex v wires b_i to $a_{1+i \bmod \delta}$. It is easily verified that the newly constructed wiring of v by connecting b_i to $a_{2+i \bmod \delta}$ produces a disjoint Euler tour $(P_\delta P_{\delta-1} \cdots P_2 P_1)$. Note that the argument fails for $\delta = 2$ where $(P_1 P_2) = (P_2 P_1)$. Figure 2 shows this diagrammatically for an example for $\sigma = 3$. $\qquad\square$

The idea of rewiring the edge-disjoint perfect matchings in Theorem 7 can be extended so that we can find more orthogonal Euler tours, if possible, by repeatedly rewiring the edge matching while maintaining the one connected Euler tour:
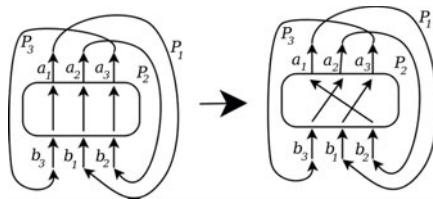
Fig. 2. An example of rewiring a vertex with $\sigma = 3$ for the proof of Theorem 7

Theorem 8. *Given an Eulerian digraph G with minimum degree δ, there exists an orthogonal collection of Euler tours in G of size $\lfloor \delta/2 \rfloor$. Therefore, there exists an orthogonal collection of de Bruijn sequences of $G_{\sigma,k}$ with size at least $\lfloor \sigma/2 \rfloor$; that is, $\lfloor \sigma/2 \rfloor \leq \Omega(\sigma, k) \leq \sigma - 1$.*

Proof. Given an orthogonal collection of m Euler tours on G, $\{C_1, C_2, \ldots, C_m\}, 1 \leq m < \lfloor \delta/2 \rfloor$, we will show that it is always possible

to rewire each vertex $v \in G$ such that the new wiring is disjoint and maintains an Euler tour.

Observe that these m disjoint Euler tours induce m forbidden bipartite matchings between the input and output edges for each vertex for the rewiring in the $(m+1)$-st sequence. Similar to the analysis of Theorem 7, take an arbitrary Euler tour with respect to a vertex v, partition edges of $G_{\sigma,k}$ into δ disjoint nonempty paths, namely $\{P_1, P_2, \ldots, P_\delta\}$, with the Eulerian tour $C = (P_1 P_2 \cdots P_\delta)$. Let a_i (b_i) denote the first (last) edge of P_i, and thus the original wiring of v wires b_i to $a_{1+i \bmod \delta}$, as, again, in Figure 2.

We define the *permissible wiring graph* of v as the digraph W_v with vertex set $\{P_1, P_2, \ldots, P_\delta\}$ & edge set $\{(P_i, P_j)| (b_i, a_j)$ is **not** an edge of $C_x, 1 \le x \le m\}$. That is, two path vertices are connected by an edge if they have not been wired together in a previous tour. We now wish to find a Hamiltonian path on this graph, as it will correspond to a wiring which is both permissible and preserves an Eulerian tour of the graph.

Observe that both the in-degree and out-degree of each vertex of the wiring graph are at least $\delta - 1 - m \ge \delta/2$. By the Ghouila-Houri theorem [23], a directed graph D with n vertices has a Hamiltonian cycle if both the in-degree and out-degree of every vertex of D are at least $n/2$. Thus, it follows that our wiring graph has a Hamilton cycle, and thus every vertex v can be successfully rewired, maintaining an Euler tour in G, while having the property that the edge-matching in v is orthogonal to those of the previous m tours. $\square$

3.1   Special Orthogonal Families

Kása [12] observed small samples of orthogonal de Bruijn sequences following a special pattern. In particular, a de Bruijn sequence can be transformed into another by means of a *morphism* function, μ, where $\mu(0) = 0, \mu(\sigma - 1) = 1$, and $\mu(i) = 1 + i$ for $1 \le i \le \sigma - 2$. Kása conjectured that for $\sigma \le 3, k \le 2$, sets of orthogonal de Bruijn sequences of size $\sigma - 1$ can be constructed from one de Bruijn sequence s and morphism operations $\nu^i(s), 1 \le i \le \sigma - 2$. For example, the following are a set of Kása's orthogonal de Bruijn sequences on $\sigma = 4, k = 2$,

$$00102113230331220$$
$$00203221310112330$$
$$00301332120223110$$

We implement tools for constructing orthogonal de Bruijn sequences of this kind under small σ and k. Kása's orthogonal de Bruijn sequences collection is interesting since it provides a simple construction of $\sigma - 1$ orthogonal de Bruijn sequences. In terms of routing in interconnection networks, this corresponds to a set of disjoint Hamiltonian cycles in the line graph, de Bruijn graph $G_{\sigma,k+1}$. Unfortunately, verifying and/or disproving Kása's conjecture is not easy mostly because of the abundance of de Bruijn sequences to be tested, and we were unable to find additional empirical evidence to support or disprove the conjecture.

We also define a new class of orthogonal de Bruijn sequences. Specifically, we note that there seem to exist sets of orthogonal de Bruijn sequences $s_1, \ldots, s_{\sigma-1}$

with the property that, for each k-mer X which is followed in s_i by the symbol y (i.e., yielding the $k+1$-mer Xy), in sequence s_{i+1} the k-mer X will be followed by the symbol $y + 1(\mathrm{mod}\ \sigma)$. For example, the sequences below have this property:

$$00102231121320330$$
$$00232103011331220$$
$$00313020123322110$$

These sets have the property of being uniquely determined by a single starting sequence. We have no guarantee that any de Bruijn sequence will yield such a set of sequences, although if they exist, the construction itself guarantees their orthogonality. If a randomly-generated de Bruijn sequence has this property, construction of such a set of orthogonal sequences is immediate. Unfortunately, the probability that a random de Bruijn sequence will have this property seems to decrease exponentially with increasing σ and k. Through extensive computational experiments, we have verified the existence of such sets of sequences up to $\sigma = 9$ for $k = 2$ and up to $k = 6$ for $\sigma = 4$.

3.2   Counting Eulerians

The matrix-tree theorem for graphs (see [21,22,23]), states that the number of Eulerian circuits in a labeled Eulerian digraph G is equal to

$$\varepsilon(G) = T \cdot \Pi_{v \in G}(deg(v) - 1)!$$

where T is the number of (directed) spanning trees rooted at any particular vertex of G. Note that the number of directed spanning trees in an Eulerian digraph does not depend on the vertex where it is rooted. Plugging in the terms $T = \sigma^{\sigma^{k-1}-k}$, $deg(v) = \sigma$, and $|V(G)| = \sigma^{k-1}$, it follows that number of Eulerian circuits in $G_{\sigma,k}$ is $[(\sigma - 1)!]^{\sigma^{k-1}} \cdot \sigma^{\sigma^{k-1}-k}$ or $(\sigma!)^{\sigma^{k-1}}/\sigma^k$.

To count the number of Eulerian circuits in Kautz graphs, and thus the number of Kautz sequences, van Aardenne-Ehrenfest's formula [22] again can be applied here. The interesting part is to calculate the number of rooted directed spanning tree on Kautz graphs. Here we use an induction on k; the inductive base is easily derived from Cayley's formula $T(\mathbf{Kz}_{\sigma,1}) = \sigma^{\sigma-1}$. The inductive hypothesis can be derived from Knuth's line graph spanning tree recursion [14]:

$$T(L(G)) = T(G)\Pi_{v \in G} \deg^+(v)^{\deg^-(v)-1}$$

where $T(G)$ denote the number of spanning trees on G; $L(G)$ is the line graph of G. It follows that $T(\mathbf{Kz}_{\sigma,k+1}) = T(\mathbf{Kz}_{\sigma,k}) \cdot (\sigma - 1)^{(\sigma-2)\sigma(\sigma-1)^{k-1}}$; note that k is decreased by 1 at deducing the final form. It follows that number of Eulerian circuits in $\mathbf{Kz}_{\sigma,k}$ is $\sigma^{\sigma-2}[(\sigma-1)!]^{\sigma(\sigma-1)^{k-1}}/(\sigma-1)^{\sigma+k-1}$. The formula can also be derived by the graph spectral theory in characteristic polynomial and permanent of the arc-graph [18]. The abundance of Kautz sequences suggests an orthogonal collection of Kautz sequences with large size.

4   Heuristic Construction of Orthogonal Sequences

The algorithm described in Theorem 8 gives no fewer than $\lfloor \sigma/2 \rfloor$ orthogonal de Bruijn sequences. In order to find the $\sigma - 1 - \lfloor \sigma/2 \rfloor$ orthogonal sequences conjectured to remain, we extend the idea used in the proof of Theorem 8. Note that the algorithm, in visiting every vertex v of G, expects the wiring graph of v to be Hamiltonian connected. Failing to satisfy the condition, the algorithm stops and reports the certified orthogonal sequences. However, it is possible (and generally the case, beyond the first $\sigma/2$ sequences) that an orthogonal sequence can only be found by rewiring two or more vertices simultaneously, while fixing any one of these vertices alone does *not* render an orthogonal sequence.

Thus, the refined approach rewires one vertex at a time, according to the temporarily fixed matches of other vertices of the graph, only this time, instead of trying to connect *all* paths incident to the vertex in one step, the algorithm picks a *good* wiring according to a heuristic, and continues to proceed to other vertices. Hopefully, in later stages of the algorithm, some vertex can be rewired and lead to an disjoint Eulerian cycle, rendering another orthogonal sequence.

To justify a *good* wiring, recall that the wiring graph of v, a digraph W_v, in essence, defines the *permissible connectivity* between the paths $Q = \{P_1, P_2, \ldots, P_\sigma\}$ incident to vertex v. Furthermore, a permissible (perfect) matching between σ pairs of in-edges and out-edges of v connects the paths of Q, forming disjoint cycles made of Q. Originally, the idea is to find a matching that forms a single loop. Here we define a good match is the one that *maximizes the length of the smallest cycle*. A good wiring makes sure that more disconnected vertices on the smallest cycle have a higher chance being *fixed* in the later stage. In order to find an orthogonal de Bruijn sequence, no disjoint cycle left on the graph is allowed. We have experimented with other heuristics, such as maximizing the longest cycle; however, these approaches do not successfully lead to finding all $\sigma - 1$ orthogonal de Bruijn sequences as efficiently.

This algorithm allows us to efficiently find de Bruijn sequences up to useful values of σ and k; the running times for our algorithm finding $\sigma - 1$ orthogonal de Bruijn sequences for various values of σ and k are illustrated in Figure 3.

It is straightforward to apply our algorithm to find orthogonal Eulerians in Kautz graphs as well. When applied to the Kautz graph, thus, we obtain orthogonal Kautz sequences. The first $\sigma - 2$ Kautz sequences are found by the algorithm in time somewhat less than that of the equivalent de Bruijn sequences; however, the algorithm frequently fails to find the $(\sigma - 1)$st Kautz sequence, because the final sequence is entirely determined by the preceding sequences. Figure 3 presents runtimes for finding $\sigma - 1$ orthogonal Kautz sequences for realizable values of σ and k.

Based on the empirical results of this program, we conjecture that $\Omega(\mathbf{Kz}_{\sigma,k}) = \sigma - 1$ for all $\sigma \geq 4$, and, moreover, that this is true for $\sigma = 3$ when $k \geq 5$. A Python implementation of this algorithm, as well as sets of orthogonal de Bruijn and Kautz sequences, are available at our website *http://www.algorithm.cs.sunysb.edu/orthogonal.*
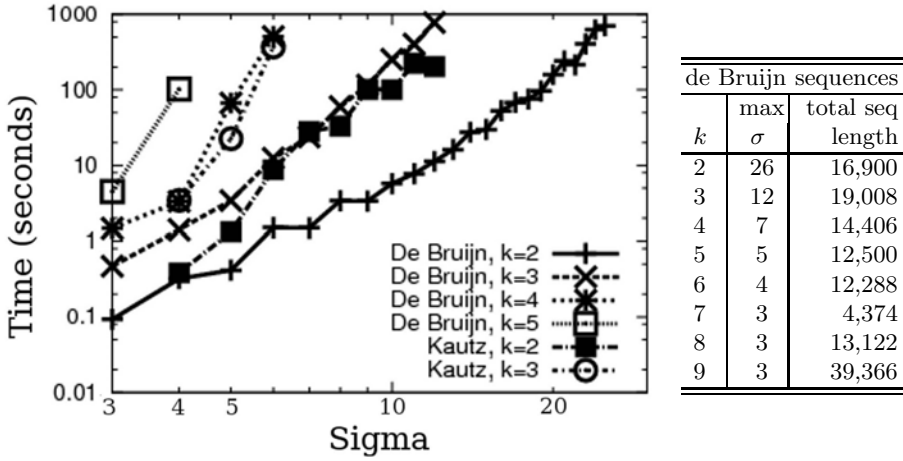
de Bruijn sequences		
	max	total seq
k	σ	length
2	26	16,900
3	12	19,008
4	7	14,406
5	5	12,500
6	4	12,288
7	3	4,374
8	3	13,122
9	3	39,366

Fig. 3. (left)Run-time of our algorithm for various values of σ and k, finding both orthogonal de Bruijn and Kautz sequences. (right) Largest achievable value of σ for each fixed k using our heuristic, given a small running time limit (1000 seconds).

5   Open Problems

We have proven that large families of orthogonal de Bruijn sequences of any order exist for all $\sigma \geq 3$. Further, we give optimal constructions for a large number of finite cases, particular those of interest in synthetic biology.

Several open problems remain. We say that two order-k de Bruijn sequences are d-orthogonal if they contain no $(k + d)$-mer in common, generalizing the notion beyond $d = 1$. The most compelling problem concerns tightening the bounds on the number of d-orthogonal (σ,k) de Bruijn sequences as a function of d, σ, and k. In particular, we know very little for $d > 1$.

We also envision a second class of diverse de Bruijn sequence families, designed to identify motifs from fragment length assays (e.g. electrophoresis) instead of sequencing. Suppose we seek to identify the cutter sequence of a specific restriction enzyme. If we design two sequences such that the resulting fragment lengths are distinct for each possible cutter, the cutter identity follows directly from observation. The applications for these sequences are more speculative than those comprising the body of this paper, but the algorithmics appear interesting.

References

1. Bermond, J.-C.: Hamiltonian decompositions of graphs, directed graphs and hypergraphs. Ann. Discrete Math. 3, 21–28 (1978); Présentéau Cambridge Combinatorial Conf., Advances in Graph Theory , Trinity College, Cambridge, England (1977)
2. Bermond, J.-C., Darrot, E., Delmas, O., Perennes, S.: Hamilton circuits in the directed wrapped butterfly network. Discrete Applied Mathematics 84(1), 21–42 (1998)

3. Bond, J., Iványi, A.: Modelling of interconnection networks using de bruijn graphs. In: Iványi, A. (ed.) Third Conference of Program Designer, Budapest (1987)
4. Bugl, H., Danner, J.P., Molinari, R.J., Mulligan, J.T., Park, H.-O., Reichert, B., Roth, D.A., Wagner, R., Budowle, B., Scripp, R.M., Smith, J.A.L., Steele, S.J., Church, G., Endy, D.: DNA synthesis and biological security. Nature Biotechnology 25, 627–629 (2007)
5. Coleman, J.R., Papamichial, D., Futcher, B., Skiena, S., Mueller, S., Wimmer, E.: Virus attenuation by genome-scale changes in codon-pair bias. Science 320, 1784–1787 (2008)
6. Czar, M.J., Anderson, J.C., Bader, J.S., Peccoud, J.: Gene synthesis demystified. Trends in Biotechnology 27(2), 63–72 (2009)
7. de Bruijn, N.G.: A combinatorial problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
8. Gibson, D., et al.: Creation of a bacterial cell controlled by a chemically synthesized genome. Science (2010), doi:10.1125./science.1190719
9. Fleischner, H., Jackson, B.: Compatible euler tours in eulerian digraphs. In: Cycles and Rays, Proceeding Colloquium Montreal, 1987. ATO ASI Ser. C, pp. 95–100. Kluwer Academic Publishers, Dordrecht (1990)
10. Golomb, S.W.: Shift Register Sequences. Holden-Day (1967)
11. Good, I.J.: Normal recurring decimals. J. London Math. Soc. 21, 167–172 (1946)
12. Kása, Z.: On arc-disjoint hamiltonian cycles in de Bruijn graphs. CoRR abs/1003.1520 (2010)
13. Kautz, W.H.: Bounds on directed (d,k) graphs. In: Theory of Cellular Logic Networks and Machines, AFCKL-68-0668 Final Rep., vol. 24, pp. 20–28 (1968)
14. Knuth, D.E.: Oriented subtrees of an arc digraph. Journal of Combinatorial Theory 3, 309–314 (1967)
15. Montes, P., Memelli, H., Ward, C., Kim, J., Mitchell, J., Skiena, S.: Optimizing restriction site placement for synthetic genomes. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 323–337. Springer, Heidelberg (2010)
16. Mueller, S., Coleman, R., Papamichail, D., Ward, C., Nimnual, A., Futcher, B., Skiena, S., Wimmer, E.: Live attenuated influenza vaccines by computer-aided rational design. Nature Biotechnology 28 (2010)
17. Ronse, C.: Feedback Shift Registers. Springer, Berlin (1984)
18. Rosenfeld, V.R.: Enumerating Kautz sequences. Kragujevac Journal of Mathematics 24, 19–41 (2002)
19. Rowley, R., Bose, B.: Edge-disjoint Hamiltonian cycles in de Bruijn networks. In: Distributed Memory Computing Conference, pp. 707–709 (1991)
20. Rowley, R., Bose, B.: On the number of arc-disjoint Hamiltonian circuits in the de Bruijn graph. Parallel Processing Letters 3(4), 375–380 (1993)
21. Tutte, W.T.: The dissection of equilateral triangles into equilateral triangles. Mathematical Proceedings of the Cambridge Philosophical Society 44, 463–482 (1948)
22. van Aardenne-Ehrenfest, T., de Bruijn, N.G.: Circuits and trees in oriented linear graphs. Simon Stevin: Wisen Natuurkundig Tijdschrift 28, 203–217 (1951)
23. West, D.: Introduction to Graph Theory, 2nd edn. Prentice-Hall, Englewood Cliffs (2000)

A Fast Algorithm for Three-Dimensional Layers of Maxima Problem

Yakov Nekrich

Department of Computer Science, University of Bonn
yasha@cs.uni-bonn.de

Abstract. We show that the three-dimensional layers-of-maxima problem can be solved in $o(n \log n)$ time in the word RAM model. Our algorithm runs in $O(n(\log \log n)^3)$ deterministic time or $O(n(\log \log n)^2)$ expected time and uses $O(n)$ space. We also describe a deterministic algorithm that uses optimal $O(n)$ space and solves the three-dimensional layers-of-maxima problem in $O(n \log n)$ time in the pointer machine model.

1 Introduction

A point p *dominates* a point q if each coordinate of p is larger than or equals to the corresponding coordinate of q. A point p is a *maximum point* in a set S if no point of S dominates p. The maxima set of S is the set of all maximum points in S. In the *layers-of-maxima* problem we assign points of a set S to layers S_i, $i \geq 1$, according to the dominance relation: The first layer of S is defined as the maxima set of S, the layer 2 of S is the maxima set of $S \setminus S_1$, and the i-th layer of S is the maxima set of $S \setminus (\cup_{j=1}^{i-1} S_j)$. In this paper we show that the three-dimensional layers-of-maxima problem can be solved in $o(n \log n)$ time.

Previous and Related Work. The algorithm of Kung, Luccio, and Preparata [23] finds the maxima set of a set S in $O(n \log n)$ time for $d = 2$ or $d = 3$ dimensions and $O(n \log^{d-2} n)$ time for $d \geq 4$ dimensions. The algorithm of Gabow, Bentley, and Tarjan [16] finds the maxima set in $O(n \log^{d-3} n \log \log n)$ time for $d \geq 4$ dimensions. Very recently, Chan, Larsen, and Pǎtraşcu [11] described a randomized algorithm that solves the d-dimensional maxima problem (i.e., finds the maxima set) for $d \geq 4$ in $O(n \log^{d-3} n)$ time. Numerous works are devoted to variants of the maxima problem in different computational models and settings: In [8], the authors describe a solution for the three-dimensional maxima problem in the cache-oblivious model. Output-sensitive algorithms and algorithms that find the maxima for a random set of points are described in [7,13,18,22]. The two-dimensional problem of maintaining the maxima set under insertions and deletions is considered in [21]; the problem of maintaining the maxima set for moving points is considered in [15].

The general layers-of-maxima problem appears to be more difficult than the problem of finding the maxima set. The three-dimensional layers-of-maxima problem can be solved in $O(n \log n \log \log n)$ time [1] using dynamic fractional

cascading [24]. The algorithm of Buchsbaum and Goodrich [9] runs in $O(n \log n)$ time and uses $O(n \log n \log \log n)$ space. Giyora and Kaplan [17] described a data structure for point location in a dynamic set of horizontal segments and showed how it can be combined with the approach of [9] to solve the three-dimensional layers-of-maxima problem in $O(n \log n)$ time and $O(n)$ space.

The $O(n \log n)$ time is optimal even if we want to find the maxima set in two dimensions [23] provided that we work in the infinite-precision computation model in which input values, i.e. point coordinates, can be manipulated with algebraic operations and compared. On the other hand, it is well known that it is possible to achieve $o(n \log n)$ time (resp. $o(\log n)$ time for searching in a data structure) for many one-dimensional as well as for some multi-dimensional problems and data structures in other computational models. For instance, the grid model, that assumes all coordinates to be integers in the range $[1, U]$ for a parameter U, was extensively studied in computational geometry. Examples of problems that can be solved efficiently in the grid model are orthogonal range reporting queries [26] and point location queries in a two- and three-dimensional rectangular subdivisons [5]. In fact, we can use standard techniques to show that these queries can be answered in $o(\log n)$ time when all coordinates are arbitrary integers. Recently, a number of other important geometric problems was shown to be solvable in $o(n \log n)$ time (resp. in $o(\log n)$ time) in the word RAM model. An incomplete list[1] includes Voronoi diagrams and three-dimensional convex hulls in $O(n \cdot 2^{O(\sqrt{\log \log n})})$ time [12], two-dimensional point location in $O(\log n / \log \log n)$ time [27,10], and dynamic convex hull in $O(\log n / \log \log n)$ time [14]. Results for the word RAM model are important because they help us better understand the structure and relative complexity of different problems and demonstrate how geometric information can be analyzed in algorithmically useful ways.

Our Results. In this paper we show that the three-dimensional layers-of-maxima problem can be solved in $O(n(\log \log n)^3)$ deterministic time and $O(n)$ space in the word RAM model. If randomization is allowed, our algorithm runs in $O(n(\log \log n)^2)$ expected time. For comparison, the fastest known deterministic linear space sorting algorithm runs in $O(n \log \log n)$ time [19]. Our result is valid in the word RAM computation model, but the time-consuming operations, such as multiplications, are only used during the pre-processing step when we sort points by coordinates (see section 2). For instance, if all points are on the $n \times n \times n$ grid, then our algorithm uses exactly the same model as [26] or [5].

We also describe an algorithm that uses $O(n)$ space and solves the three-dimensional layers-of-maxima problem in optimal $O(n \log n)$ time in the pointer machine model [28]. The result of Giyora and Kaplan [17] that achieved the same space and time bounds is valid only in the RAM model. Thus we present the first algorithm that solves the three-dimensional layers-of-maxima problem in optimal time and space in the pointer machine model.

---

[1] We note that problems in this list are more difficult than the layers-of-maxima problem because in our case we process a set of axis-parallel segments.

Overview. Our solution, as well as the previous results, is based on the sweep plane algorithm of [9] described in section 2. The sweep plane algorithm assigns points to layers by answering for each $p \in S$ a point location query in a dynamically maintained staircase subdivision. We observe that general data structures for point location in a set of horizontal segments cannot be used to obtain an $o(n \log n)$ time solution. Even in the word RAM model, no dynamic data structure that supports both queries and updates in $o(\log n)$ time is known. Moreover, by the lower bound of [2] any data structure for a dynamic set of horizontal segments needs $\Omega(\log n / \log \log n)$ time to answer a point location query. We achieve a significantly better result using the methods described below.

In section 3 we describe the data structure for point location in a staircase subdivision that supports queries in $O((\log \log n)^3)$ time and updates[2] in poly-logarithmic time per segment. This result may be of interest on its own.

The data structure of section 3 is not sufficient to obtain the desired runtime and space usage mainly due to high costs of update operations. To reduce the update time and space usage, we construct auxiliary staircases $\mathcal{B}_i$, such that: 1. the total number of segments in $\mathcal{B}_i$ and the total number of updates is $O(n/d)$ for a parameter $d = \log^{O(1)} n$; 2. locating a point p among staircases $\mathcal{B}_i$ gives us an approximate location of p among the original staircases $\mathcal{M}_i$ (up to $O(d)$ staircases). An efficient method for maintaining staircases $\mathcal{B}_i$, described in section 4, is the most technically challenging part of our construction. In section 5 we show how the data structure of section 3 can be combined with the auxiliary staircases approach to obtain an $O(n(\log \log n)^3)$ time algorithm. We also sketch how the same approach enables us to obtain an $O(n \log n)$ time and $O(n)$ space algorithm in the pointer machine model.

2   Sweep Plane Algorithm

Our algorithm is based on the three-dimensional sweep method that is also used in [9]. We move the plane parallel to the xy plane[3] from $z = +\infty$ to $z = 0$ and maintain the following invariant: when the z-coordinate of the plane equals v all points p with $p.z \geq v$ are assigned to their layers of maxima. Here and further $p.x$, $p.y$, and $p.z$ denote the x-,- y-, and z-coordinates of a point p. Let $S_i(v)$ be the set of points q that belong to the i-th layer of maxima such that $q.z > v$; let $P_i(v)$ denote the projection of $S_i(v)$ on the sweep plane, $P_i(v) = \{\pi(p) \,|\, p \in S_i(v)\}$ where $\pi(p)$ denotes the projection of a point p on the xy-plane. For each value of v maximal points of $P_i(v)$ form a staircase $\mathcal{M}_i$; see Fig. 1. When the z-coordinate of the sweep plane is changed from $v + 1$ to v, we assign all points with $p.z = v$ to their layers of maxima. If $\pi(p)$, such that $p.z = v$, is dominated by a point from $P_i(v + 1)$, then p belongs to the j-th layer of maxima and $j > i$. If $\pi(p)$, such that $p.z = v$, dominates a point on $P_k(v + 1)$, then p belongs to the j-th layer of maxima and $j \leq k$. We observe that $\pi(p)$ dominates $P_i(v + 1)$ if and only if the

---

[2] We will describe update operations supported by our data structure in sections 2 and 3.

[3] We assume that all points have positive coordinates.

staircase $\mathcal{M}_i$ is dominated by p, i.e., the vertical ray shot from p in $-y$ direction passes through $\mathcal{M}_i$. Hence, the point p belongs to the layer i, such that $\pi(p)$ is between the staircase $\mathcal{M}_{i-1}$ and the staircase $\mathcal{M}_i$. This means that we can assign a point to its layer by answering a point location query in a staircase subdivision. When all p with $p.z = v$ are assigned to their layers, staircases are updated.

Thus to solve the layers of maxima problem, we examine points in the descending order of their z-coordinates. For each v, such that there is at least one p with $p.z = v$, we proceed as follows: for every p with $p.z = v$ operation $\texttt{locate}(p)$ identifies the staircase $\mathcal{M}_i$ immediately below $\pi(p)$. If the first staircase below $\pi(p)$ has index i ($\pi(p)$ may also lie on $\mathcal{M}_i$), then p is assigned to the i-th layer of maxima; if $\pi(p)$ is below the lowest staircase $\mathcal{M}_j$, then p is assigned to the new layer $j+1$. When all points with $p.z = v$ are assigned to their layers, the staircases are updated. All points p such that $p.z = v$ are examined in the ascending order of their x-coordinates. If a point p with $p.z = v$ is assigned to layer i, we perform operation $\texttt{replace}(p,i)$ that removes all points of $\mathcal{M}_i$ dominated by p and inserts p into $\mathcal{M}_i$. If the staircase i does not exist, then instead of $\texttt{replace}(p,i)$ we perform the operation $\texttt{new}(p,i)$; $\texttt{new}(p,i)$ creates a new staircase $\mathcal{M}_i$ that consists of one horizontal segment h with left endpoint $(0, p.y)$ and right endpoint $\pi(p)$ and one vertical segment t with upper endpoint $\pi(p)$ and lower endpoint $(0, p.x)$. See Fig. 1 for an example.
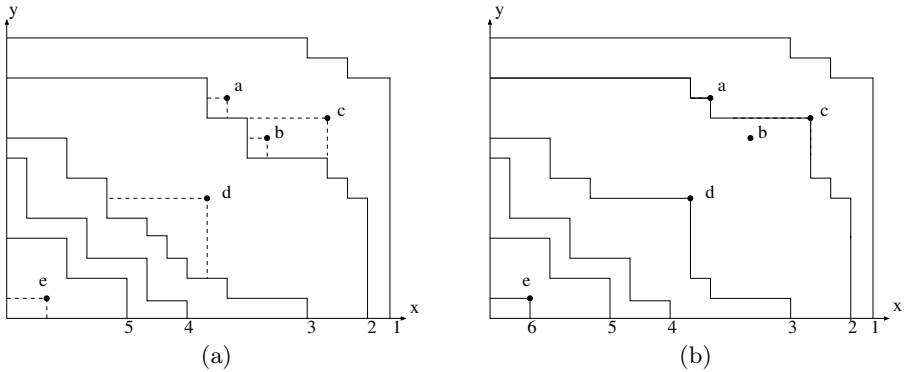
Fig. 1. Points a, b, c, d, and e have the same z-coordinate. (a) Points a, b and c are assigned to layer 2, d is assigned to layer 3, and e is assigned to a new layer 6. (b) Staircases after operations $\texttt{replace}(a,2)$, $\texttt{replace}(b,2)$, $\texttt{replace}(c,2)$, $\texttt{replace}(d,3)$, and $\texttt{new}(e)$. Observe that b is not the endpoint of a segment in the staircase $\mathcal{M}_2$ after updates.

We can reduce the general layers of maxima problem to the problem in the universe of size $O(n)$ using the reduction to rank space technique [26,16]. The rank of an element $e \in S$ is defined as the number of elements in S that are smaller than e: $\texttt{rank}(e, S) = |\{a \in S \,|\, a < e\}|$; clearly, $\texttt{rank}(e, S) \le |S|$. For a point $p = (p.x, p.y, p.z), p \in S$, let $\tau(p) = (\texttt{rank}(p.x, S_x) + 1, \texttt{rank}(p.y, S_y) + 1, \texttt{rank}(p.z, S_z) + 1)$. Let $S' = \{\tau(p) \,|\, p \in S\}$. Coordinates of all points in S'

belong to range $[1, n]$. A point p dominates a point q if and only if $\mathtt{rank}(p.x, S_x) \geq \mathtt{rank}(q.x, S_x)$, $\mathtt{rank}(p.y, S_y) \geq \mathtt{rank}(q.y, S_y)$, and $\mathtt{rank}(p.z, S_z) \geq \mathtt{rank}(q.z, S_z)$ where S_x, S_y, S_z are sets of x-, y-, and z-coordinates of points in S. Hence if a point $p' \in S'$ is assigned to the i-th layer of maxima of S', then $\tau^{-1}(p')$ belongs to the i-th layer of maxima of S. We can find ranks of $x-$, $y-$, and $z-$coordinates of every point by sorting S_x, S_y, and S_z. Using the sorting algorithm of [19], S_x, S_y, and S_z can be sorted in $O(n \log \log n)$ time and $O(n)$ space. Thus the layers of maxima problem can be reduced to the special case when all point coordinates are bounded by $O(n)$ in $O(n \log \log n)$ time.

3   Fast Queries, Slow Updates

In this section we describe a data structure that supports $\mathtt{locate}(q)$ in $O((\log \log n)^3)$ time and update operations $\mathtt{replace}(q, i)$ and $\mathtt{new}(q, i)$ in $O(\log n (\log \log n)^2)$ time per segment. We will store horizontal segments of all staircases in a data structure that supports *ray shooting queries*: given a query point q identify the first segment s crossed by a vertical ray that is shot from q in $-y$ direction; in this case we will say that the segment s precedes q (or s is the predecessor segment of q). In the rest of this paper, segments will denote horizontal segments. Identifying the segment that precedes q is (almost) equivalent to answering a query $\mathtt{locate}(q)$. Operation $\mathtt{replace}(q, i)$ corresponds to a deletion of all horizontal segments dominated by q and an insertion of at most two horizontal segments, see Fig 1. Operation $\mathtt{new}(q, i)$ corresponds to an insertion of a new segment.

Our data structure is a binary tree on x-coordinates and segments are stored in one-dimensional secondary structures in tree nodes. The main idea of our approach is to achieve fast query time by binary search of the root-to-leaf path: using properties of staircases, we can determine in $O((\log \log n)^2)$ time whether the predecessor segment of a point q is stored in the ancestor of a node v or in the descendant of a node v for any node v on the path from the root to $q.x$. Our approach is similar to the data structure of [5], but we need additional techniques to support updates.

For a horizontal segment s, we denote by $\mathtt{start}(s)$ and $\mathtt{end}(s)$ the x-coordinates of its left and right endpoints respectively; we denote by $y(s)$ the y-coordinate of all points of s. An integer $e \in S$ precedes (follows) an integer x in S if e is the largest (smallest) element in S, such that $e \leq x$ ($e \geq x$). Let H be a set of segments and let H_y be the set of y-coordinates of segments in H. We say that $s \in H$ precedes (follows) an integer e if the y-coordinate of s precedes (follows) e in H_y. Thus a segment that precedes a point q is a segment that precedes $q.y$ in the set of all segments that intersect the vertical line $x = q.x$.

We construct a balanced binary tree $\mathcal{T}$ of height $\log n$ on the set of all possible x-coordinates, i.e., n leaves of $\mathcal{T}$ correspond to integers in $[1, n]$. The range of a node v is the interval $\mathtt{rng}(v) = [\mathtt{left}(v), \mathtt{right}(v)]$ where $\mathtt{left}(v)$ and $\mathtt{right}(v)$ are leftmost and rightmost leaf descendants of v.

We say that a segment s spans a node v if $\texttt{start}(s) < \texttt{left}(v) < \texttt{right}(v) < \texttt{end}(s)$; a segment r belongs to a node v if $\texttt{left}(v) < \texttt{start}(s) < \texttt{end}(s) < \texttt{right}(v)$. A segment s l-cuts a node v if s intersects the vertical line $x = \texttt{left}(v)$, but s does not span v, i.e., $\texttt{start}(s) \leq \texttt{left}(v)$ and $\texttt{end}(s) < \texttt{right}(v)$; a segment s r-cuts a node v if s intersects the vertical line $x = \texttt{right}(v)$ but s does not span v, i.e., $\texttt{start}(s) > \texttt{left}(v)$ and $\texttt{end}(s) \geq \texttt{right}(v)$. A segment s such that $[\texttt{start}(s), \texttt{end}(s)] \cap \texttt{rng}(v) \neq \emptyset$ either cuts v, or spans v, or belongs to v. We store y-coordinates of all segments that l-cut (r-cut) a node v in a data structure $\mathcal{L}_v$ ($\mathcal{R}_v$). Using exponential trees [4], we can implement $\mathcal{L}_v$ and $\mathcal{R}_v$ in linear space, so that one-dimensional searching (i.e. predecessor and successor queries) is supported in $O((\log \log n)^2)$ time. Since a segment cuts $O(\log n)$ nodes (at most two nodes on each tree level), all $\mathcal{L}_v$ and $\mathcal{R}_v$ use $O(n \log n)$ space. We denote by $\texttt{index}(s)$ the index of the staircase $\mathcal{M}_i$ that contains s, i.e., $s \in \mathcal{M}_{\texttt{index}(s)}$. The following simple properties are important for the search procedure:

Fact 1. *Suppose that an arbitrary vertical line cuts staircases $\mathcal{M}_i$ and $\mathcal{M}_j$, $i < j$, in points p and q respectively. Then $p.y > q.y$ because staircases do not cross.*

Fact 2. *For any two points p and q on a staircase $\mathcal{M}_i$, if $p.x < q.x$, then $p.y \geq q.y$*

Fact 3. *Given a staircase $\mathcal{M}_i$ and a point p, we can determine whether $\mathcal{M}_i$ is below or above p and find the segment $s \in \mathcal{M}_i$ such that $p.x \in [\texttt{start}(s), \texttt{end}(s)]$ in $O((\log \log n)^2)$ time. The data structure D_i that supports such queries uses linear space and supports finger updates in $O(1)$ time.*

Proof: The data structure D_i contains x-coordinates of all segment endpoints of $\mathcal{M}_i$. D_i is implemented as an exponential tree so that it uses $O(n)$ space. Using D_i we can identify $s \in \mathcal{M}_i$ such that $p.x \in [\texttt{start}(s), \texttt{end}(s)]$ in $O((\log \log n)^2)$ time; $\mathcal{M}_i$ is below p if and only if s is below p. $\square$

Using Fact 3 we can determine whether a segment s precedes a point q in $O((\log \log n)^2)$ time: Suppose that s belongs to a staircase $\mathcal{M}_i$. Then s is the predecessor segment of q iff $q.x \in [\texttt{start}(s), \texttt{end}(s)]$, $q.y \geq y(s)$ and the staircase $\mathcal{M}_{i-1}$ is above q.

We can use these properties and data structures $\mathcal{L}_v$ and $\mathcal{R}_v$ to determine whether a segment b that precedes a point q spans a node v, belongs to a node v, or cuts a node v. If the segment b we are looking for spans v, then it cuts an ancestor of v; if that segment belongs to v, then it cuts a descendant of v. Hence, we can apply binary search and find in $O(\log \log n)$ iterations the node f such that the predecessor segment of q cuts f. Observe that in some situations there may be no staircase $\mathcal{M}_i$ below q, see[25] for an example. To deal with such situations, we insert a dummy segment s_d with left endpoint $(1, 0)$ and right endpoint $(n, 0)$; we set $\texttt{index}(s_d) = +\infty$ and store s_d in the data structure $\mathcal{L}_{v_0}$ where v_0 is the root of $\mathcal{T}$.

Let l_x be the leaf in which the predecessor of $q.x$ is stored. We will use variables l, u and v to guide the search for the node f. Initially we set $l = l_x$ and u is the
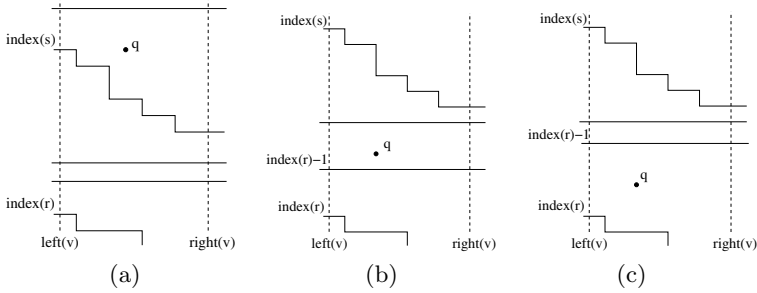
Fig. 2. Search procedure in a node v. Staircases are denoted by their indexes. Figures (a) and (b) correspond to cases **1** and **2** respectively. The case when the predecessor segment belongs to $\mathcal{M}_{\texttt{index}(r)}$ is shown on Fig. (c).

root of $\mathcal{T}$. We set v to be the middle node between u and l: if the path between u and l consists of h edges, then the path from u to v consists of $\lfloor h/2 \rfloor$ edges and v is an ancestor of l.

Let r and s denote the segments in $\mathcal{L}_v$ that precede and follow $q.y$. If there is no segment s in $\mathcal{L}_v$ with $y(s) > q.y$, then we set $s = NULL$. If there is no segment r in $\mathcal{L}_v$ with $y(r) \leq q.y$, then we set $r = NULL$. We can find both r and s in $O((\log \log n)^2)$ time. If the segment $r \neq NULL$, we check whether the staircase $\mathcal{M}_{\texttt{index}(r)}$ contains the predecessor segment of q; by Fact 3, this can be done in $O((\log \log n)^2)$ time. If $\mathcal{M}_{\texttt{index}(r)}$ contains the predecessor segment of q, the search is completed. Otherwise, the staircase $\mathcal{M}_{\texttt{index}(r)-1}$ is below q or $r = NULL$. In this case we find the segment r' that precedes $q.y$ in $\mathcal{R}_v$. If r' is not the predecessor segment of q or $r' = NULL$, then the predecessor segment of q either spans v or belongs to v. We distinguish between the following two cases:

1. The segment $s \neq NULL$ and the staircase that contains s is below q. By Fact 1, a vertical line $x = q.x$ will cross the staircase of s before it will cross a staircase $\mathcal{M}_i$, $i > \texttt{index}(s)$. Hence, a segment that spans v and belongs to the staircase $\mathcal{M}_i$, $i > \texttt{index}(s)$, cannot be the predecessor segment of q. If a segment t spans v and $\texttt{index}(t) < \texttt{index}(s)$, then the y-coordinate of t is larger than the y-coordinate of s by Fact 1. Since $y(t) > y(s)$ and $y(s) > q.y$, the segment t is above q. Thus no segment that spans v can be the predecessor of q.

2. The staircase that contains s is above q or $s = NULL$. If r exists, the staircase $\mathcal{M}_{\texttt{index}(r)-1}$ is below q. Hence, the predecessor segment of q belongs to a staircase[4] $\mathcal{M}_i$, $\texttt{index}(s) < i \leq \texttt{index}(r) - 1$. Since each staircase $\mathcal{M}_i$, $\texttt{index}(s) < i \leq \texttt{index}(r) - 1$, contains a segment that spans v, the predecessor segment of s is a segment that spans v. If r does not exist, then every segment below the point q spans the node v. Hence, the predecessor segment of s spans v. See Fig. 2 for an example.

If the predecessor segment spans v, we search for f among ancestors of v; if the predecessor segment belongs to v, we search for f among descendants of v.

---

[4] To simplify the description, we assume that $\texttt{index}(s) = 0$ if $s = NULL$.

Hence, we set $l = v$ in case 2, and we set $u = v$ in case 1. Then, we set v to be the middle node between u and l and examine the new node v. Since we examine $O(\log \log n)$ nodes and spend $O((\log \log n)^2)$ time in each node, the total query time is $O((\log \log n)^3)$.

If the predecessor segment is the dummy segment s_d, then there is no horizontal segment of any $\mathcal{M}_i$ below q. In this case we must identify the staircase to the left of $q.x$. Let m_i denote the rightmost point on the staircase $\mathcal{M}_i$, i.e., m_i is a point on $\mathcal{M}_i$ such that $m_i.y = 0$. Then q is between staircases $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$, such that $m_i.x < q.x < m_{i-1}.x$. We can find m_i in $O((\log \log n)^2)$ time.

When a segment s is deleted, we delete it from the corresponding data structure D_i. We also delete s from all data structures $\mathcal{L}_v$ and $\mathcal{R}_w$ for all nodes v and w, such that s l-cuts v (respectively r-cuts w). Since a segment cuts $O(\log n)$ nodes and exponential trees support updates in $O((\log \log n)^2)$ time, a deletion takes $O(\log n(\log \log n)^2)$ time. Insertions are supported in the same way[5]. Operation $\texttt{new}(q, l)$ is implemented by inserting a segment with endpoints $(0, q.y)$ and $(q.x, q.y)$ into $\mathcal{T}$, incrementing by one the number of staircases l, and creating the new data structure D_l. To implement $\texttt{replace}(q, i)$ we delete the segments "covered" by q from $\mathcal{T}$ and D_i and insert the new segment (or two new segments) into $\mathcal{T}$ and D_i.

Lemma 1. *We can store n horizontal staircase segments with endpoints on $n \times n$ grid in a $O(n \log n)$ space data structure that answers ray shooting queries in $O((\log \log n)^3)$ time and supports operation $\texttt{replace}(q, i)$ in $O(m \log n(\log \log n)^2)$ time where m is the number of segments inserted into and deleted from the staircase $\mathcal{M}_i$, and operation $\texttt{new}(q)$ in $O(\log n(\log \log n)^2)$ time.*

The data structure of Lemma 1 is deterministic. We can further improve the query time if randomization is allowed.

Fact 4. *Given a staircase $\mathcal{M}_i$ and a point p, we can determine whether $\mathcal{M}_i$ is below or above p and find the segment $s \in \mathcal{M}_i$ such that $p.x \in [\texttt{start}(s), \texttt{end}(s)]$ in $O((\log \log n))$ time. The data structure D_i that supports such queries uses linear space and supports finger updates in $O(1)$ expected time.*

Proof: The data structure is the same as in the proof of Fact 3, but we use the y-fast tree data structure [29] instead of the exponential tree.   □

Lemma 2. *We can store n horizontal staircase segments with endpoints on $n \times n$ grid in a $O(n \log n)$ space data structure that answers ray shooting queries in $O((\log \log n)^2)$ time and supports operation $\texttt{replace}(q, i)$ in $O(m \log n \log \log n)$ expected time where m is the number of segments inserted into and deleted from the staircase $\mathcal{M}_i$, and operation $\texttt{new}(q)$ in $O(\log n \log \log n)$ expected time.*

Proof: Our data structure is the same as in the proof of Lemma 1. But we implement D_i using Fact 4. Data structures $\mathcal{L}_v$ and $\mathcal{R}_v$ are implemented using

---

[5] The update time can be slightly improved using fractional cascading and similar techniques, but this is not necessary for our presentation.

the y-fast tree [29]. Hence, the search procedure spends $O(\log \log n)$ time in each node of $\mathcal{T}$ and a query is answered in $O((\log \log n)^2)$ time.   □

Although this is not necessary for further presentation, we can prove a similar result for the case when all segment endpoints are on a $U \times U$ grid; the query time is $O(\log \log U + (\log \log n)^3)$ and the update time is $O(\log^3 n (\log \log n)^2)$ per segment. We refer to the full version of this paper [25] for a proof of this result.

4   Additional Staircases

The algorithm in the previous section needs $O(n \log n (\log \log n)^2)$ time to construct the layers of maxima: n ray shooting queries can be performed in $O(n(\log \log n)^3)$ time, but $O(n)$ update operations take $O(n \log n (\log \log n)^2)$ time. To speed-up the algorithm and improve the space usage, we reduce the number of updates and the number of segments in the data structure of Lemma 1 to $O(n/\log^2 n)$.

Let $\mathcal{D}$ denote the data structure of Lemma 1. We construct and maintain a new sequence of staircases $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m$, where $m \leq n/d$ and the parameter d will be specified later. All horizontal segments of $\mathcal{B}_1, \ldots, \mathcal{B}_m$ are stored in $\mathcal{D}$. The new staircases satisfy the following conditions:

1. There are $O(\frac{n}{d})$ horizontal segments in all staircases $\mathcal{B}_i$
2. $\mathcal{D}$ is updated $O(\frac{n}{d})$ times during the execution of the sweep plane algorithm.
3. For any point q and for any i, if q is between $\mathcal{B}_{i-1}$ and $\mathcal{B}_i$, then q is situated between $\mathcal{M}_k$ and $\mathcal{M}_{k+1}$ for $(i - 3/2)d \leq k \leq (i + 1/2)d$.

Conditions 1 and 2 imply that the data structure $\mathcal{D}$ uses $O(n)$ space and all updates of $\mathcal{D}$ take $O(n)$ time if $d \geq \log n (\log \log n)^2$. Condition 3 means that we can use staircases $\mathcal{B}_i$ to guide the search among $\mathcal{M}_k$: we first identify the index i, such that the query point q is between $\mathcal{B}_{i+1}$ and $\mathcal{B}_i$, and then locate q in $\mathcal{M}_{(i-3/2)d}, \ldots, \mathcal{M}_{(i+1/2)d}$. It is not difficult to construct $\mathcal{B}_i$ that satisfy conditions 1 and 3. The challenging part is maintaining the staircases $\mathcal{B}_i$ with a small number of updates.

Lemma 3. *The total number of inserted and deleted segments in all $\mathcal{B}_i$ is $O(\frac{n}{d})$. The number of segments stored in $\mathcal{B}_i$ is $O(\frac{n}{d})$.*

We describe how staircases can be maintained and prove Lemma 3 in the full version of this paper [25].

5   Efficient Algorithms for the Layers-of-Maxima Problem

Word RAM Model. To conclude the description of our main algorithm, we need the following simple.

Lemma 4. *Using a $O(m)$ space data structure, we can locate a point in a group of d staircases $\mathcal{M}_j, \mathcal{M}_{j+1}, \ldots, \mathcal{M}_{j+d}$ in $O(\log d \cdot (\log \log m)^2)$ time, where m is the number of segments in $\mathcal{M}_j, \mathcal{M}_{j+1}, \ldots, \mathcal{M}_{j+d}$. An operation* replace$(q, i)$ *is supported in $O((\log \log m)^2 + m_q)$ time, where m_q is the number of inserted and deleted segments in the staircase $\mathcal{M}_i$, $j \leq i \leq i + d$.*

Proof: We can use Fact 3 to determine whether a staircase is above or below a staircase $\mathcal{M}_k$ for any $j \leq k \leq j + d$. Hence, we can locate a point in $O(\log d \cdot (\log \log m)^2)$ time by a binary search among d staircases. □

We set $d = \log^2 n$. The data structure F_i contains all segments of staircases $\mathcal{M}_{(i-1)d+1}, \mathcal{M}_{(i-1)d+2}, \ldots, \mathcal{M}_{id}$ for $i = 1, 2, \ldots, j$, where $j = \lfloor l/d \rfloor$ and l is the highest index of a staircase; the data structure F_{j+1} contains all segments of staircases $\mathcal{M}_{jd+1}, \ldots, \mathcal{M}_l$. We can locate a point q in each F_i in $O((\log \log n)^3)$ time by Lemma 4. Since each staircase belongs to one data structure, all F_i use $O(n)$ space. We also maintain additional staircases $\mathcal{B}_i$ as described in section 4. All segments of all staircases $\mathcal{B}_i$ are stored in the data structure $\mathcal{D}$ of Lemma 1; since $\mathcal{D}$ contains $O(n/d)$ segments, the space usage of $\mathcal{D}$ is $O(n)$.

Now we can describe how operations locate, replace, new can be implemented in $O((\log \log n)^3)$ time per segment.

-   locate(q): We find the index k, such that q is between $\mathcal{B}_{k-1}$ and $\mathcal{B}_k$ in $O((\log \log n)^3)$ time. As described in section 4, q is between $\mathcal{M}_{kd+g}$ and $\mathcal{M}_{(k-1)d-g}$. Hence, we can use data structures F_{k+1}, F_k, and F_{k-1} to identify j such that q is between $\mathcal{M}_j$ and $\mathcal{M}_{j+1}$. Searching F_{k+1}, F_k, and F_{k-1} takes $O((\log \log n)^3)$ time, and the total time for locate(q) is $O((\log \log n)^3)$.
-   replace(q, i): let m_q be the number of inserted and deleted segments. The data structure $F_{\lfloor i/d \rfloor}$ can be updated in $O(m_q + (\log \log n)^2)$ time. We may also have to update $\mathcal{B}_{\lfloor i/d \rfloor}$, $\mathcal{B}_{\lfloor i/d \rfloor + 1}$, and the data structure $\mathcal{D}$.
-   new(q, l): If $l = kd + 1$ for some k, a new data structure F_{k+1} is created. We add the horizontal segment of the new staircase into the data structure F_{k+1}. If $l = kd$, we create a new staircase $\mathcal{B}_k$ and add the segments of $\mathcal{B}_k$ into the data structure $\mathcal{D}$.

There are $O(n/d)$ update operations on the data structure $\mathcal{D}$ that can be performed in $O((n/d) \log n (\log \log n)^2) = O(n)$ time. If we ignore the time to update $\mathcal{D}$, then replace(q, i) takes $O(m_q (\log \log n)^2)$ time and new(q, l) takes $O((\log \log n)^2)$ time. Since $\sum_{q \in S} m_q = O(n)$ and new(q, l) is performed at most n times, the algorithm runs in $O(n(\log \log n)^3)$ time. We thus obtain the main result of this paper.

Theorem 1. *The three-dimensional layers-of-maxima problem can be solved in $O(n(\log \log n)^3)$ deterministic time in the word RAM model. The space usage of the algorithm is $O(n)$.*

If we use Fact 4 instead of Fact 3 in the proof of Lemma 4 and Lemma 2 instead of Lemma 1 in the proof of Theorem 1, we obtain a slightly better randomized algorithm.

Theorem 2. *The three-dimensional layers-of-maxima problem can be solved in* $O(n(\log \log n)^2)$ *expected time. The space usage of the algorithm is* $O(n)$.

Pointer Machine Model. We can apply the idea of additional staircases to obtain an $O(n \log n)$ algorithm in the pointer machine model. This time, we set $d = \log n$ and maintain additional staircases $\mathcal{B}_i$ as described in section 4. Horizontal segments of all $\mathcal{B}_i$ are stored in the data structure $\mathcal{D}$ of Giyora and Kaplan [17] that uses $O(m \log^\varepsilon m)$ space and supports queries and updates in $O(\log m)$ and $O(\log^{1+\varepsilon} m)$ time respectively, where m is the number of segments in all $\mathcal{B}_i$ and ε is an arbitrarily small positive constant. Using dynamic fractional cascading [24], we can implement F_i so that F_i uses linear space and answers queries in $O(\log n + \log \log n \log d) = O(\log n)$ time. Updates are supported in $O(\log n)$ time; details will be given in the full version of this paper. Using $\mathcal{D}$ and F_i, we can implement the sweep plane algorithm in the same way as described in the first part of this section. The space usage of all data structures F_i is $O(n)$, and all updates of F_i take $O(n \log n)$ time. By Lemma 3, the data structure $\mathcal{D}$ is updated $O(n/\log n)$ times; hence all updates of $\mathcal{D}$ take $O(n \log n)$ time. The space usage of $\mathcal{D}$ is $O(m \log^\varepsilon m) = O(n)$. Each new point is located by answering one query to $\mathcal{D}$ and at most three queries to F_i; hence, a new point is assigned to its layer of maxima in $O(\log n)$ time.

Theorem 3. *A three-dimensional layers-of-maxima problem can be solved in* $O(n \log n)$ *time in the pointer machine model. The space usage of the algorithm is* $O(n)$.

Acknowledgment

References

1. Agarwal, P.K.: Personal communication
2. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked Ancestor Problems. In: Proc. FOCS 1998, pp. 534–544 (1998)
3. Atallah, M.J., Goodrich, M.T., Ramaiyer, K.: Biased Finger Trees and Three-dimensional Layers of Maxima. In: Proc. SoCG 1994, pp. 150–159 (1994)
4. Andersson, A., Thorup, M.: Dynamic Ordered Sets with Exponential Search Trees. J. ACM 54, Article No. 13 (2007)
5. de Berg, M., van Kreveld, M.J., Snoeyink, J.: Two- and Three-Dimensional Point Location in Rectangular Subdivisions. J. Algorithms 18, 256–277 (1995)
6. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two Simplified Algorithms for Maintaining Order in a List. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002)
7. Bentley, J.L., Clarkson, K.L., Levine, D.B.: Fast Linear Expected-Time Algorithms for Computing Maxima and Convex Hulls. In: Proc. SODA 1990, pp. 179–187 (1990)

8. Brodal, G.S., Fagerberg, R.: Cache Oblivious Distribution Sweeping. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 426–438. Springer, Heidelberg (2002)
9. Buchsbaum, A.L., Goodrich, M.T.: Three-Dimensional Layers of Maxima. Algorithmica 39, 275–286 (2004)
10. Chan, T.M.: Point Location in o(log n) Time, Voronoi Diagrams in o(n log n) Time, and Other Transdichotomous Results in Computational Geometry. In: Proc. FOCS 2006, pp. 333–344 (2006)
11. Chan, T.M., Larsen, K., Pătraşcu, M.: Orthogonal Range Searching on the RAM, Revisited (to be published in SoCG 2011)
12. Chan, T.M., Pătraşcu, M.: Voronoi Diagrams in $n2^{o(\sqrt{lglgn})}$ time. In: Proc. STOC 2007, pp. 31–39 (2007)
13. Clarkson, K.L.: More Output-Sensitive Geometric Algorithms. In: Proc. FOCS 1994, pp. 695–702 (1994)
14. Demaine, E.D., Pătraşcu, M.: Tight Bounds for Dynamic Convex Hull Queries (Again). In: Proc. SoCG 2007, pp. 354–363 (2007)
15. Franciosa, P.G., Gaibisso, C., Talamo, M.: An Optimal Algorithm for the Maxima Set Problem for Data in Motion. In: Proc. CG 1992, pp. 17–21 (1992)
16. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and Related Techniques for Geometry Problems. In: Proc. STOC 1984, pp. 135–143 (1984)
17. Giyora, Y., Kaplan, H.: Optimal Dynamic Vertical Ray Shooting in Rectilinear Planar Subdivisions. ACM Transactions on Algorithms 5 (2009)
18. Golin, M.J.: A Provably Fast Linear-Expected-Time Maxima-Finding Algorithm. Algorithmica 11, 501–524 (1994)
19. Han, Y.: Deterministic Sorting in $O(nloglogn)$ time and linear space. J. Algorithms 50, 96–105 (2004)
20. Itai, A., Konheim, A.G., Rodeh, M.: A Sparse Table Implementation of Priority Queues. In: Proc. ICALP 1981, pp. 417–431 (1981)
21. Kapoor, S.: Dynamic Maintenance of Maximas of 2-d Point Sets. In: Proc. SoCG 1994, pp. 140–149 (1994)
22. Kirkpatrick, D.G., Seidel, R.: Output-Size Sensitive Algorithms for Finding Maximal Vectors. In: Proc. SoCG 1985, pp. 89–96 (1985)
23. Kung, H.T., Luccio, F., Preparata, F.P.: On Finding the Maxima of a Set of Vectors. J. ACM 22, 469–476 (1975)
24. Mehlhorn, K., Näher, S.: Dynamic Fractional Cascading. Algorithmica 5, 215–241 (1990)
25. Nekrich, Y.: A Fast Algorithm for Three-Dimensional Layers of Maxima Problem. In: CoRR, abs/1007.1593 (2010)
26. Overmars, M.H.: Efficient Data Structures for Range Searching on a Grid. J. Algorithms 9(2), 254–275 (1988)
27. Pătraşcu, M.: Planar Point Location in Sublogarithmic Time. In: Proc. FOCS 2006, pp. 325–332 (2006)
28. Tarjan, R.E.: A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets. J. Comput. Syst. Sci. 18(2), 110–127 (1979)
29. Willard, D.E.: Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. Information Processing Letters 17(2), 81–84 (1983)
30. Willard, D.E.: A Density Control Algorithm for Doing Insertions and Deletions in a Sequentially Ordered File in Good Worst-Case Time. Information and Computation 97, 150–204 (1992)

Succinct 2D Dictionary Matching with No Slowdown

Shoshana Neuburger[1,*] and Dina Sokol[2,**]

[1] Department of Computer Science, The Graduate Center of the City University of New York, New York, NY, 10016
shoshana@sci.brooklyn.cuny.edu
[2] Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, NY, 11210
sokol@sci.brooklyn.cuny.edu

Abstract. The dictionary matching problem seeks all locations in a given text that match any of the patterns in a given dictionary. Efficient algorithms for dictionary matching scan the text once, searching for all patterns simultaneously. This paper presents the first 2-dimensional dictionary matching algorithm that operates in small space and linear time. Given d patterns, $D = \{P_1, \ldots, P_d\}$, each of size $m \times m$, and a text T of size $n \times n$, our algorithm finds all occurrences of P_i, $1 \leq i \leq d$, in T. The preprocessing stores the dictionary in entropy compressed form, in $|D|H_k(D) + O(|D|)$ bits. Our algorithm uses $O(dm \log dm)$ bits of extra space. The time complexity of our algorithm is linear $O(|D| + |T|)$.

1 Introduction

Dictionary matching is the problem of searching a given text for all occurrences of any pattern from a given set of patterns. A search for specific phrases in a book, virus detection software, network intrusion detection, searching a DNA sequence for a set of motifs, and image identification, are all applications of dictionary matching. In this work we are concerned with efficiently solving the 2-dimensional dictionary matching problem in *small space*. The motivation for this is that there are many scenarios, such as on mobile and satellite devices, where storage capacity is limited. The added constraint of performing efficient dictionary matching using little extra space is a challenging and practical problem.

Linear-time *single* pattern matching algorithms in both one and two dimensions have achieved impressively small space complexities. For 1D data, we have pattern matching algorithms that require only constant extra space, [10, 8, 18, 11]. For 2D pattern matching, Crochemore et al. present a linear time algorithm that works with *log* extra space for pattern preprocessing and $O(1)$ extra space

to scan the text [7]. Such an algorithm can be trivially extended to perform dictionary matching but its runtime would depend on the number of patterns in the dictionary. The goal of an efficient dictionary matching algorithm is to scan the text once so that its running time depends only on the size of the text and not on the size of the patterns sought.

Concurrently searching for a set of patterns within limited working space presents a greater challenge than searching for a single pattern in small space. Much effort has recently been devoted to solving 1-dimensional dictionary matching in small space [6, 14, 15, 3, 13]. The most recent result of Hon et al. [13] has essentially closed this problem, with a linear-time algorithm that uses $|D'|H_k(D') + O(|D'|)$ bits of space, given a dictionary D' of 1D patterns. H_k, i.e. the kth order empirical entropy of a string, describes the minimum number of bits that are needed to encode each symbol of the string within context, and it is often used to demonstrate that storage space meets the information-theoretic lower bounds of data.

In this paper, our objective is to extend succinct 1D dictionary matching to the two-dimensional setting, in a way similar to the Bird and Baker (BB) extension of the Aho-Corasick 1D dictionary matching algorithm (AC). It turns out that this problem is not trivial, due to the necessity to label each position of the text. However, using new techniques of dynamic dueling, we indeed achieve a linear time algorithm that solves the *Small-Space 2D Dictionary Matching Problem*.

Specifically, given a dictionary D of d patterns, $D = \{P_1, \ldots, P_d\}$, each of size $m \times m$, and a text T of size $n \times n$, our algorithm finds all occurrences of P_i, $1 \leq i \leq d$, in T. During the preprocessing stage, the patterns are stored in entropy compressed form, in $|D|H_k(D) + O(|D|)$ bits. $H_k(D)$ denotes the kth order empirical entropy of the string formed by concatenating all the patterns in D, row by row. The preprocessing is completed in $O(|D|)$ time using $O(dm \log dm)$ bits of extra space. Then, the text is searched in $O(|T|)$ time using $O(dm \log dm)$ bits of extra space. For ease of exposition, we discuss patterns that are all of size $m \times m$, however, our algorithm generalizes to patterns that are the same size in only one dimension, and the complexity would depend on the size of the largest dimension. As in [3] and [13], the alphabet can be non-constant in size.

In the next section we give an overview of Bird/Baker [5, 2] and Hon et al. [13] and outline how it is possible to combine these algorithms to yield a small space 2-dimensional dictionary matching algorithm for certain types of patterns. In Section 3 we introduce a distinction between highly periodic patterns and non-periodic patterns, and summarize the algorithm for the periodic case. In section 4 we deal with the non-periodic case, introducing new space-saving techniques including dynamic dueling. We conclude with open problems in Section 5.

2   Overview

The first linear-time 2D pattern matching algorithm was developed independently by Bird [5] and Baker [2]. Although the BB algorithm was initially presented for a single pattern, it is easily extended to perform dictionary matching

by replacing the KMP automaton with an AC automaton. In Algorithm 1 we outline the dictionary matching version of BB.

Algorithm 1. Bird/Baker algorithm for 2D Dictionary Matching

1. Preprocess Pattern:
   a) Form Aho-Corasick automaton of pattern rows, called AC1.
   Let ℓ denote the number of states in AC1.
   b) Name pattern rows using AC1, and store a 1D pattern of names for each pattern in D, called D'.
   c) Construct AC automaton of D', called AC2.
   Let ℓ' denote the number of states in AC2.
2. Row Matching:
   Run Aho-Corasick on each text row using AC1.
   This labels positions at which a pattern row ends.
3. Column Matching:
   Run Aho-Corasick on named text columns using AC2.
   Output pattern occurrences.

The basic concept used here is the translation of the 2D patterns into 1D patterns using naming. Rows of each pattern are perceived as metacharacters and named so that distinct rows receive different names. The text is named in a similar fashion and 1D dictionary matching is performed over the text columns and the patterns of names. The linear time complexity of the algorithm depends on the assumption that the label of each state fits into a single word of RAM.

Space Complexity of BB: We use ℓ to denote the number of states in the Aho-Corasick automaton of D, AC1. Note that $\ell \leq |D| = dm^2$. $O(\ell \log \ell)$ bits of space are needed to store AC1, and labeling all text locations uses $O(n^2 \log dm)$ bits of space. Overall, the BB algorithm uses $O(\ell \log \ell + n^2 \log dm)$ bits of space.

Our objective is to improve upon this space requirement. As a first attempt to conserve space, we replace the traditional AC algorithm in Step 1 of BB with the compressed AC automaton of Hon et al. [13]. The algorithm of [13] builds upon the work of Belazzougui [3] which encodes the three functions of the AC automaton (*goto, report, failure*) separately and in different ways. The space complexity is reduced from 0th order empirical entropy to kth order empirical entropy by employing the XBW transform [9] to store the *goto* function. The details of these papers are very interesting, but for our purposes, we can actually use their algorithm as a black box replacement for the AC automata in both Steps 1a and 1c of the BB algorithm.

To reduce the algorithm's working space, we work with small overlapping text blocks of size $3m/2 \times 3m/2$. This way, we can replace the $O(n^2 \log dm)$ bits of space used to label the text in Step 3 with $O(m^2 \log dm)$ bits, relating the working space to the size of the dictionary, rather than the size of the entire text.

Theorem 1. *We can solve the 2D dictionary matching problem in linear $O(dm^2 + n^2)$ time and $\ell H_k(D) + \ell' H_k(D') + O(\ell + m^2 \log dm)$ bits of space.*

Proof. Since the algorithm of Hon et al. [13] has no slowdown, replacing the AC automata in BB with compressed AC automata preserves the linear time complexity. The space used by the preprocessing is: $\ell H_k(D) + O(\ell) + \ell' H_k(D') + O(\ell')$ bits. The compressed AC1 automaton uses $\ell H_k(D) + O(\ell)$ bits of space and it replaces the original dictionary, while the compressed AC2 automaton uses $\ell' H_k(D') + O(\ell')$ *extra* bits of space. Text scanning uses $O(m^2 \log dm)$ extra bits of space to label each location of a text block.   □

Although this is an improvement over the space required by the uncompressed version of BB, we would like to improve on this further. Our aim is to reduce the working space to $O(dm \log dm)$ bits, thus completely eliminating the dependence of the working space on the size of the given text. Yet, note that this constraint still allows us to store $O(1)$ information per pattern row to linearize the dictionary in the preprocessing. In addition, we will have the ability to store $O(1)$ information about each pattern per text row to allow linearity in text scanning.

The following corollary restates Theorem 1 in terms of $O(dm \log dm)$ for the case of a dictionary with many patterns. It also omits the term $\ell' H_k(D') + O(\ell')$, since $\ell' H_k(D') + O(\ell') = O(dm \log dm)$.

Corollary 1. *If $d > m$, we can solve the 2D dictionary matching problem in linear $O(dm^2 + n^2)$ time and $\ell H_k(D) + O(\ell) + O(dm \log dm)$ bits of space.*

The rest of this paper deals with the case in which the number of patterns is smaller than the dimension of the patterns, i.e., $d = o(m)$. For this case, we cannot label each text location and therefore the Bird and Baker algorithm cannot be applied trivially. We present several clever space-saving tricks to preserve the spirit of Bird and Baker's algorithm without incurring the necessary storage overhead.

3   Preliminaries

A string S is *primitive* if it cannot be expressed in the form $S = u^j$, for $j > 1$ and a prefix u of S. String S is *periodic* in u if $S = u'u^j$ where u' is a suffix of u, u is primitive, and $j \geq 2$. A periodic string p can be expressed as $u'u^j$ for one unique primitive u. We refer to u as "the period" of p. Depending on the context, u can refer to either the string u or the period size $|u|$.

There are two types of patterns, and each one presents its own difficulty. In the first type, which we call Case 1 patterns, all rows are periodic, with periods $\leq m/4$. The difficulty in this case is that many overlapping occurrences can appear in the text in close proximity to each other, and we can easily have more candidates than the working space we allow. The second type, Case 2 patterns, have at least one aperiodic row or one row whose period is larger than $m/4$. Here, each pattern can occur only $O(1)$ times in a text block. Since several patterns can overlap each other in both directions, a difficulty arises in the text scanning stage. We do not allow the time to verify different candidates separately, nor do we allow space to keep track of the possible overlaps for different patterns.

In the initial preprocessing step, we divide the patterns into two groups based on 1D periodicity. For Case 1 patterns, the algorithm presented by the authors [17] for LZ-compressed texts can be adapted here to solve the general 2D dictionary matching problem. Since every row of every pattern is periodic with period $\leq m/4$, we have the following.

Observation 1. *Any text row that is included in a pattern occurrence must have exactly one maximal periodic substring of length $\geq m$.*

Hence, we can run the Main and Lorentz algorithm [16] on the text rows, using location m as the 'center.' Once text rows are labeled, we use our innovative naming technique [17] based on Lyndon words to name both pattern rows and text rows. We then run the traditional AC automaton on the 1D patterns of names and 1D text of names to identify candidates for pattern occurrences. Verification of actual occurrences proceeds using similar data structures.

Lemma 1. *[17] 2D dictionary matching for Case 1 patterns can be done in $O(dm^2 + n^2)$ time and $\ell H_k(D) + O(\ell) + O(dm \log m)$ bits of space.*

For Case 2 patterns, we can use the aperiodic row to filter the text. This simplifies the identification of an initial set of candidates. Yet, verification in one pass over the text presents a difficulty. In dictionary matching, different candidates represent different patterns, and it is infeasible to compute and store information about the relationship between all patterns. In the next section we present an approach to verify a set of candidate positions for a *set of patterns* in linear time using a new technique called dynamic dueling.

4   The Algorithm

Recall that we focus on the case in which the number of patterns in the dictionary is smaller than the dimension of a pattern, i.e. $d < m$. We further assume that each pattern has at least one aperiodic row. The case of a pattern having a row that is periodic with period size between $m/4$ and $m/2$ will add only a small constant to the stated complexities.

The difficulty in applying BB is that we do not have sufficient space to label all text positions. However, we can initially filter the text block using the aperiodic row. Thus, the text scanning stage first identifies a small set of positions that are candidates for pattern occurrences. Then, in a second pass over the text block, we verify which candidates are actually pattern occurrences.

4.1   Pattern Preprocessing

1. Construct (compressed) AC automaton of first aperiodic row of each pattern. Store row number of each of these rows within the patterns.
2. Form a compressed AC automaton of the pattern rows.
3. Construct witness tree of pattern rows and preprocess for LCA.
4. Name pattern rows. Index the 1D patterns of names in a suffix tree.

In the first step, we form an AC automaton of one aperiodic row of each pattern, say, the first aperiodic row of each pattern. This will allow us to filter the text and limit the number of potential pattern occurrences to consider. Since we use only one row from each pattern, using a compressed version of the AC automaton is optional.

In the second step, the pattern rows are named as in BB to form a 1D dictionary of patterns. Here we use a compressed AC automaton of the pattern rows. An example of two patterns and their 1D representation is shown in Fig. 1.

Pattern 1

a	b	b	b		1
a	b	b	b		1
a	a	b	c		2
a	b	b	b		1

Pattern 2

a	b	b	b		1
a	a	b	c		2
c	b	c	a		3
b	b	a	a		4
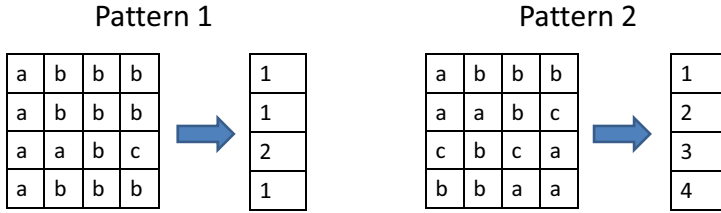
Fig. 1. Two linearized 2D patterns with their 1D names

Another necessary data structure is the witness tree introduced in [17] and summarized in Appendix A. A witness tree is used to store pairwise distinctions between different patterns, or pattern rows, of the same length. A witness tree provides a witness between pattern rows in constant time if it is preprocessed for Lowest Common Ancestor (LCA).

Preprocessing proceeds by indexing the 1D patterns of names. We form a suffix tree of the 1D patterns to allow efficient computation of longest common prefix (LCP) queries between substrings of the 1D patterns.

Lemma 2. *The pattern preprocessing stage for Case 2 patterns completes in $O(dm^2)$ time and $O(dm \log m)$ extra bits of space.*

Proof. The AC automaton of the first non-periodic row of each pattern is constructed in $O(dm)$ time and is stored in $O(dm \log m)$ bits, in its uncompressed form. A compressed AC automaton of all pattern rows occupies $\ell H_k(D) + O(\ell)$ bits of space and can then become the sole representation of the dictionary [13]. The witness tree occupies $O(dm \log m)$ bits of space [17]. A rooted tree can be preprocessed in linear time and space to answer LCA queries in $O(1)$ time [12, 4]. The patterns are converted to a 1D representation in $O(dm^2)$ time. A suffix tree of the 1D dictionary of names can be constructed and stored in linear time and space, e.g. [19]. □

4.2   Text Scanning

The text scanning stage has three steps.

1. Identify candidates in text block with 1D dictionary matching of a non-periodic row of each pattern.
2. Duel to eliminate inconsistent candidates within each column.
3. Verify pattern occurrences at surviving candidate positions.

Step 1. Identify Candidates. We identify a limited set of candidates in the text block using 1D dictionary matching on the first aperiodic row of each pattern. There can be only one occurrence of any non-periodic pattern row in a text block row. Each occurrence of an aperiodic pattern row demarcates a candidate, at most d per row. In total, there can be up to dm candidates in a text block, with candidates for several distinct 1D patterns on a single row of text. If the same aperiodic row occurs in several patterns, several candidates can occur at the same text position, but candidates are still limited to d per row.

We run the Aho-Corasick algorithm over the text block, row by row, to find up to dm candidates. Then we update each candidate to reflect the position at which we expect a pattern to begin. This is done by subtracting the row number of the selected aperiodic row from the row number of its found location in the text block.

Complexity of Step 1: 1D dictionary matching on a text block takes $O(m^2)$ time with the AC method[1]. Marking the positions at which patterns can begin is done in constant time per candidate found; overall, this requires $O(dm) = o(m^2)$ time. The AC algorithm uses extra space proportional to the dictionary, which is $O(dm \log m)$ bits of space for this limited set of pattern rows. The dm candidates can also be stored in $O(dm \log m)$ bits of space.

Step 2. Eliminate Vertically Inconsistent Candidates. We call a pair of candidate patterns *consistent* if all positions of overlap match. Vertically consistent candidates are two candidates that appear in the same column, and have a suffix/prefix match in their 1D representations. In order to verify candidates in a single pass over the text, we take advantage of the fact that overlapping segments of consistent candidates can be verified simultaneously.

We eliminate inconsistent candidates with a dueling technique inspired by the 2D *single* pattern matching algorithm of Amir et al. [1]. In the single pattern matching algorithm, duels are performed between candidates for the same pattern. In dictionary matching, we perform duels between candidates for *different* patterns.

In general, the dueling paradigm requires that a witness, i.e. position of mismatch in the overlap, be precomputed and stored for all possible overlaps. If no witness exists for a given distance, then such candidates are consistent. During a duel, the text location is compared to the witness, killing one or more of the candidates involved in the duel. To date, the dueling paradigm has not been applied to dictionary matching since it is prohibitive to precompute and store witnesses for all possible overlaps of all candidate patterns in a set of patterns. However, here we show an innovative way of performing 2D duels for a set of patterns.

---

[1] Hashing techniques achieve linear time complexity in the AC algorithm.

For our purposes, we need only eliminate vertically inconsistent candidates. Thus, we introduce *dynamic dueling* between two candidates in a given column. In dynamic dueling, no witness locations are computed in advance. We are given two candidate patterns and their locations, candidate A at location (i, j) in the text and candidate B at location (k, j) in the text, $i < k$. Since all of our candidates are in an $m/2 \times m/2$ square, we know that there is overlap between the two candidates.

A duel consists of two steps. In the first step, the 1D representation of names is used for A and B, denoted by A' and B'. An LCP query between the suffix $k - i + 1$ of A' against B' returns the number of overlapping rows that match. If this number is $\geq i + m - k$ then the two candidates are consistent. Otherwise, we are given a "row-witness," i.e. the LCP points to the first row at which the patterns differ. In the second step of the duel, we use the witness tree to locate the position of mismatch between the two different pattern rows, and we use that position to eliminate one or both candidates.

To demonstrate how a witness is found and the duel is performed, we return to the patterns in Fig. 1. Assume two candidates exist; directly below a candidate for Pattern1, we have a candidate for Pattern2. The LCP of *121*, (second suffix of linearized Pattern1) and *1234* (linearized Pattern2) is 2. Since $2 < 3$, the LCP query reveals that the patterns are inconsistent, and that a witness exists between the fourth row of Pattern1 (name 1) and the third row of Pattern2 (name 3). We then procure a witness from the witness tree shown in Fig. 3 by taking the LCA of the leaves that represent names 1 and 3. The result of this query shows that the first position is a point of distinction between names 1 and 3. If the text has an 'a' at that position, Pattern1 survives the duel. Otherwise, if the character is a 'c', Pattern2 survives the duel. If neither 'a' nor 'c' occur at the text location, both candidates are eliminated.
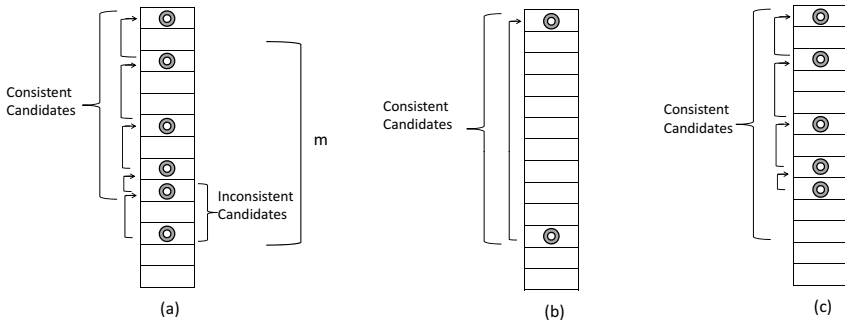
Fig. 2. (a) Duel between vertically inconsistent candidates in a column. (b) Surviving candidates if the lower candidate wins the duel. (c) Surviving candidates if the upper candidate wins the duel.

Lemma 3. *A duel between two candidate patterns A and B in a given column j of the text can be performed in constant time.*

Proof. The suffix tree constructed in Step 4 of the pattern preprocessing answers LCP queries in the 1D patterns of names in $O(1)$ time. The witness tree gives a position of a row-witness in $O(1)$ time, and retrieving the text character to perform the actual duel takes constant time. □

Verification begins by dueling top-down between candidates within each column. Since consistency is a transitive relation, if the lower candidate is killed, this does not affect the consistent candidates above it in the same column. However, if the lower candidate survives, it triggers the elimination of all candidates within m rows above the witness row. Pointers link consecutive candidates in each column. This way, a duel eliminates the set of consistent candidates that are within range of the mismatch. This is shown in Fig. 2. Note that the same method can be used when two (or more) candidates occur at a single text position.

Complexity of Step 2: Step 2 begins with at most dm candidate positions. Each candidate is involved in exactly one duel, and is either killed or survives. If a candidate survives, it may be visited exactly one more time to be eliminated by a duel beneath it. Since a duel is done in constant time, by Lemma 3, this step completes in $O(dm)$ time. Recall that $d < m$. Hence, the time for Step 2 is $O(m^2)$.

Step 3. Verify Surviving Candidates After eliminating vertically inconsistent candidates, we verify pattern occurrences in a single scan of the text block. We process one text block row at a time to conserve space. Before scanning the current text block row, we label the positions at which we expect to find a pattern row. This is done by merging the labels from the previous row with the list of candidates that begin on the new row. If a new candidate is introduced in a column that already has a label, we keep only the label of the lower candidate. This is permissible since the label must be from a consistent candidate in the same column. Thus, each position in the text has at most one label.

The text block row is then scanned sequentially, to mark actual occurrences of pattern rows. This is done by running AC on the text row with the compressed AC automaton of all pattern rows. The lists of expected row names and actual row names are then compared sequentially. If every expected row name appears in the text block row, the candidate list remains unchanged. If an expected row name does not appear, a candidate is eliminated. The pointers that connect candidates are used to eliminate candidates in the same column that also include the label that was not found.

After all rows are verified in this manner, all surviving candidates in the text are pattern occurrences of their respective patterns.

Complexity of Step 3: When a text block row is verified, we mark each position at which a pattern row (1D name) is expected to begin. This list is limited by $m/2$ due to the vertical consistency of the candidates. We also mark actual pattern row occurrences in the text row which are again no more than

$m/2$ due to distinctness of the row names. Thus, the space complexity for Step 3 is $O(m)$. The time complexity is also linear, since AC is run on the row in linear time, and then two sorted lists of pattern row names are merged. Over all $3m/2$ rows in the text block, the complexity of Step 3 is $O(m^2)$.

Lemma 4. *The algorithm for 2D dictionary matching, when pattern rows are not highly periodic and $d < m$, completes in $O(n^2)$ time and $O(dm \log m)$ bits of space, in addition to $\ell H_k(D) + O(\ell)$ bits of space to store the compressed AC automaton of the dictionary.*

Proof. This follows from Lemma 2 and the complexities of Steps 1, 2, and 3.   □

Theorem 2. *Our algorithm for 2D dictionary matching completes in $O(dm^2 + n^2)$ time and $O(dm \log dm)$ bits of extra space.*

Proof. For $d > m$, this is stated in Corollary 1 of Theorem 1. For $d \leq m$, the patterns are split into groups according to periodicity. Case 1 patterns are proven in Lemma 1, and Case 2 patterns are proven in Lemma 4.   □

5   Conclusion

We have developed the first linear-time small-space 2D dictionary matching algorithm. We work with a dictionary of d patterns, each of size $m \times m$. After preprocessing the dictionary in small space, and storing the dictionary in a compressed self-index, our algorithm processes the text in linear time. That is, it uses $O(n^2)$ time to search a 2D text that is $O(n^2)$ in size. Yet, our algorithm requires only $O(dm \log dm)$ bits of extra space.

Our algorithm is suitable for patterns that are all the same size in at least one dimension. This property is carried over from the Bird/Baker solution. Small space 2D dictionary matching for *rectangular patterns* remains an open problem. Other interesting variations of small-space dictionary matching include the approximate versions of the problem in which one or more changes occur either in the patterns or in the text.

Acknowledgments

References

[1] Amir, A., Benson, G., Farach, M.: An alphabet independent approach to two-dimensional pattern matching. SICOMP: SIAM Journal on Computing 23 (1994)
[2] Baker, T.J.: A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comp. (7), 533–541 (1978)

[3] Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 88–100. Springer, Heidelberg (2010)

[4] Bender, M.A., Farach-Colton, M.: The lca problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)

[5] Bird, R.S.: Two dimensional pattern matching. Information Processing Letters 6(5), 168–170 (1977)

[6] Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Transactions on Algorithms 3(2) (2007)

[7] Crochemore, M., Gasieniec, L., Plandowski, W., Rytter, W.: Two-dimensional pattern matching in linear time and small space. In: STACS: Annual Symposium on Theoretical Aspects of Computer Science (1995)

[8] Crochemore, M., Perrin, D.: Two-way string-matching. J. ACM 38(3), 650–674 (1991)

[9] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Structuring labeled trees for optimal succinctness, and beyond. In: FOCS 2005, pp. 184–196 (2005)

[10] Galil, Z., Seiferas, J.: Time-space-optimal string matching (preliminary report). In: STOC 1981: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, pp. 106–113. ACM, New York (1981)

[11] Gąsieniec, L., Kolpakov, R.: Real-time string matching in sublinear space. In: Sahinalp, S.C., Muthukrishnan, S.M., Dogrusoz, U. (eds.) CPM 2004. LNCS, vol. 3109, pp. 117–129. Springer, Heidelberg (2004)

[12] Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SICOMP: SIAM Journal on Computing 13 (1984)

[13] Hon, W.-K., Ku, T.-H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. In: SPIRE, pp. 191–200 (2010)

[14] Hon, W.-K., Lam, T.W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed index for dictionary matching. In: DCC, pp. 23–32 (2008)

[15] Hon, W.-K., Lam, T., Shah, R., Tam, S.-L., Vitter, J.S.: Succinct index for dynamic dictionary matching. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1034–1043. Springer, Heidelberg (2009)

[16] Main, M.G., Lorentz, R.J.: An O(n log n) algorithm for finding all repetitions in a string. ALGORITHMS: Journal of Algorithms 5 (1984)

[17] Neuburger, S., Sokol, D.: Small-space 2D compressed dictionary matching. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 27–39. Springer, Heidelberg (2010)

[18] Rytter, W.: On maximal suffixes and constant-space linear-time versions of kmp algorithm. Theor. Comput. Sci. 299(1-3), 763–774 (2003)

[19] Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)

A   Witness Tree

Given a set S of k strings, each of length m, a witness tree can be constructed to name these strings in linear time and $O(k)$ space [17]. Furthermore, the witness tree can provide the answer to the following query in constant time.

Query: For any two strings $s, s' \in S$, return a position of mismatch between s and s' if $s \neq s'$, otherwise return $m + 1$.

Witness Tree
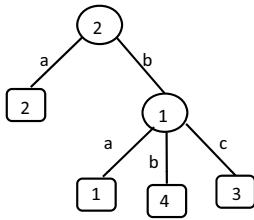
Fig. 3. A witness tree for several strings of length 4

For completeness, we review the construction of the witness tree. We omit all proofs, which have been included in [17].

Components of witness tree:

- *Internal node:* position of a character mismatch. The position is an integer $\in [1, m]$.
- *Edge:* labeled with a character in the alphabet Σ. Two edges emanating from a node must have different labels.
- *Leaf:* A name (i.e an integer $\in [1, k]$). Identical strings receive identical names.

Construction of the witness tree begins by choosing any two strings in S and sequentially comparing them. When a mismatch is found, comparison halts and a node is added to the witness tree to represent this witness. Each successive string is compared to the witnesses stored in the tree to identify to which name, if any, the string belongs. Characters of a new string are examined in the order dictated by traversal of the witness tree, possibly out of sequence. If traversal halts at an internal node, the string receives a new name. Otherwise, traversal halts at a leaf, and the new string is sequentially compared to the string represented by the leaf as done with the first two strings.

As an example, we explain how the string *bbaa* is named 4 using the witness tree of Fig. 3. Since the root represents position 2, the first comparison finds a 'b' as the second character in *bbaa*. Then, we look at position 1 and find 'b'. Since this character is not a child of the current node, a new leaf is created to represent the name 4.

Proprocessing the witness tree to allow Lowest Common Ancestor (LCA) queries on its leaves allows us to answer the above witness query between any two strings in S in constant time.

PTAS for Densest k-Subgraph in Interval Graphs

Tim Nonner

IBM Research - Zurich
`tno@zurich.ibm.com`

Abstract. Given an interval graph and integer k, we consider the problem of finding a subgraph of size k with a maximum number of induced edges, called *densest k-subgraph problem in interval graphs*. It has been shown that this problem is NP-hard even for chordal graphs [17], and there is probably no PTAS for general graphs [12]. However, the exact complexity status for interval graphs is a long-standing open problem [17], and the best known approximation result is a 3-approximation algorithm [16]. We shed light on the approximation complexity of finding a densest k-subgraph in interval graphs by presenting a polynomial-time approximation scheme (PTAS), that is, we show that there is an $(1+\epsilon)$-approximation algorithm for any $\epsilon > 0$, which is the first such approximation scheme for the densest k-subgraph problem in an important graph class without any further restrictions.

1 Introduction

The *densest k-subgraph problem* is defined as follows: given a graph with n vertices and an integer $k \leq n$, find a subgraph with k vertices that maximizes the number of induced edges. The NP-hardness is a direct consequence of the well-known fact that finding a maximum clique is NP-hard. This contrasts to the problem of finding an arbitrary-sized subgraph of maximum density, i.e., a subgraph with maximum average degree, which is polynomially solvable [14]. The first approximation algorithm for the densest k-subgraph problem with an approximation guarantee of $\mathcal{O}(n^{0.3885})$ was given by Kortsarz and Peleg [13]. Feige et al. [8] improved this factor to $\mathcal{O}(n^\delta)$ for some $\delta < 1/3$. Until recently, when Bhaskara et al. [5] presented an $\mathcal{O}(n^{1/4+\epsilon})$-approximation algorithm for any $\epsilon > 0$, it has been a long-standing open problem whether the factor $\mathcal{O}(n^{1/3})$ can be significantly beaten. Ashahiro et al. [3] showed that a simple greedy strategy yields an approximation ratio of $\mathcal{O}(n/k)$, and Feige and Langberg [7] showed that n/k is achievable using semidefinite programming. On the other hand, using a random sampling technique, Arora et al. [2] presented a polynomial-time approximation scheme (PTAS) for dense graphs with $k = \Omega(n)$, that is, if the number of edges is $\Omega(n^2)$ and $k = \Omega(n)$, then there is an $(1+\epsilon)$-approximation algorithm for any $\epsilon > 0$. However, there is probably no PTAS for general graphs [12].

These weak approximation guarantees for general graphs motivated the study of special graph classes, like perfect graphs, chordal graphs, and interval graphs.

Unfortunately, although a maximum clique of a perfect graph can be found in polynomial time [10], finding a densest k-subgraph remains NP-hard even for chordal graphs and bipartite graphs [17], two important subclasses of perfect graphs. However, the complexity status of finding a densest k-subgraph in interval graphs, a subclass of chordal graphs, has been a prominent open problem over the last three decades [17]. In an *interval graph*, each vertex corresponds to an interval, and two vertices are connected via an edge if their corresponding intervals overlap. Such a geometric structure occurs frequently in scheduling, VLSI-design, and biology, see for instance [9]. This unknown complexity status gave rise to the search for approximation results. Liazi et al. [16] presented a 3-approximation algorithm for interval graphs and chordal graphs, and there is moreover a PTAS if the clique graph is a star [15], i.e., if the intersection graph of the maximal cliques is a star. Recently, Chen et al. [6] showed that a large family of intersection graphs, including interval graphs, admit constant factor approximation algorithms. Finally, Backer and Keil [4] gave a 3/2-approximation algorithm for proper interval graphs [4], a subclass of interval graphs where no interval is allowed to contain another one.

Contributions. We significantly improve upon the known approximation results by presenting a PTAS for finding a densest k-subgraph in interval graphs. It is worth mentioning here that this is the first PTAS for an import graph class without any further restrictions, since so far only PTASs for dense graphs with $k = \Omega(n)$ [2] and a quite restricted subclass of chordal graphs [15] are known. We conjecture that finding a densest k-subgraph in interval graphs is NP-hard, but proving this (or giving a polynomial time algorithm instead) remains a challenging open problem [17].

Technique and outline. If k is constant, then we can easily find a densest k-subgraph in polynomial time by simply enumerating all $\binom{n}{k} = \mathcal{O}(n^k)$ subgraphs of size k. Hence, the difficulty of the problem stems from the fact that k is part of the input. Now, let V be the vertices of the input interval graph, and let $V^* \subseteq V$ with $|V^*| = k$ be a subset of vertices that induce a densest k-subgraph. Consider some clique $C \subseteq V$, and let $C^* = C \cap V^*$ be the subclique of C contained in V^*. Assume then that we already know the vertices $V^* \backslash C$, and hence we only need to pick some $k - |V^* \backslash C|$ vertices from C that maximize the number of induced edges in combination with $V^* \backslash C$. Clearly, C^* will solve this simplified problem. However, we might need to enumerate $\binom{|C|}{|C^*|}$ subcliques of C in order to find C^* (or another subclique of similar quality), which is not possible in polynomial time if $|C^*|$ is not constant. Therefore, we show in Section 4 that, by losing an $(1-\epsilon)$-factor in the number of induced edges for an arbitrary small $\epsilon > 0$, it suffices to only consider a polynomial number of subcliques (Lemma 4). To extend this search space restriction to the whole interval graph, we first need to introduce the notion of a sequence representation in Section 3 which allows us to decompose any densest k-subgraph into a sequence of cliques. Finally, we present a backward dynamic program in Section 5 that finds a densest k-subgraph in this restricted

search space in polynomial time. The principle of reducing the search space in order to make it treatable by dynamic programming in polynomial time has been succesfully applied during the last decades [11,1] to obtain approximation schemes. However, finding the right reduction and dynamic program is a highly problem-specific challenge.

2 Preliminaries

Let $G = (V, E)$ be an *interval graph*, i.e., each vertex in V corresponds to an interval, and two vertices are connected via an edge in E if their corresponding intervals overlap. Hence, the we can also think of the vertices in V as intervals. For an interval subset $V' \subseteq V$, let $E(V')$ denote the number of edges of the subgraph of G induced by V', i.e., the number of overlaps between intervals in V'. Hence, our goal is to find an interval set $V^* \subseteq V$ of size k that maximizes $E(V^*)$. We also refer to such a subset as a *densest k-interval subset*. Let OPT $:= E(V^*)$ denote this maximal number of overlaps. Finally, for two disjoint interval sets $V', V'' \subseteq V$, let $E(V', V'')$ denote the number of edges of the bipartite subgraph of G induced by these two interval sets, i.e., the number of overlaps between intervals in V' and intervals in V''. Observe that a *clique* $C \subseteq V$ is a set of intervals with $\cap_{I \in C} I \neq \emptyset$.

Let $l_I \in \mathbb{R}$ and $r_I \in \mathbb{R}$ denote the left and right endpoint of an interval $I \in V$, respectively, and assume that all endpoints of intervals are distinct, which can be easily ensured without changing the overlap structure. This also ensures that all intervals are distinct. We write $I_1 < I_2$ for two intervals $I_1, I_2 \in V$ if $r_{I_1} < l_{I_2}$. In this case, we also say that the pair I_1, I_2 is *consecutive*. Analogously, we call a sequence of intervals $I_1, I_2, \ldots, I_s \in V$ with $I_1 < I_2 < \ldots < I_s$ a *consecutive interval sequence*. Finally, for an interval $I \in V$, let $V_I \subseteq V$ be the set of all intervals $I' \in V$ with $I \subseteq I'$.

Lemma 1. *For any densest k-interval subset $V^* \subseteq V$, we may assume for each interval $I \in V^*$ that $V_I \subseteq V^*$.*

Proof. Assume for contradiction that there is an interval $I \in V^*$ and another interval $I' \in V_I \setminus V^*$. In this case, since $I \subset I'$, we could replace interval I by interval I' without modifying the size of V^* and without decreasing $E(V^*)$. Therefore, iterating this scheme terminates and gives us a densest k-interval subset V^* that satisfies the property from the claim. $\square$

3 Sequence Representations

For a consecutive interval pair $I_1, I_2 \in V$, let $C_{I_1 I_2}$ denote the clique of all intervals $I \in V$ with $l_{I_1} < l_I < r_{I_1}$ and $r_{I_1} < r_I < r_{I_2}$. Moreover, let $C'_{I_1 I_2}$ denote the clique of all intervals $I \in V$ with $r_{I_1} < l_I < r_{I_2}$ and $r_{I_2} \leq r_I$. Note that $I_2 \in C'_{I_1 I_2}$. We illustrate the cliques $C_{I_1 I_2}$ and $C'_{I_1 I_2}$ in Figure 1.
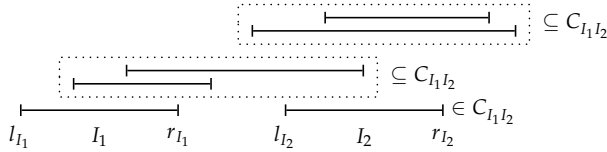
Fig. 1. Example intervals in the cliques $C_{I_1 I_2}$ and $C'_{I_1 I_2}$

To avoid case distinctions, we add a dummy interval I_∞ to V with $I < I_\infty$ for any other interval $I \in V \backslash \{I_\infty\}$. Our goal is then to find a densest $(k+1)$-interval subset $V^* \subseteq V$ with $I_\infty \in V^*$. Since I_∞ does not overlap with any other interval in V, this slightly modified problem is equivalent to our original problem of simply finding a densest k-interval subset. Specifically, removing I_∞ from a solution $V' \subseteq V$ for this modified problem yields a solution for our original problem with the same number of interval overlaps $E(V')$. Hence, OPT denotes the maximal number of interval overlaps in both cases.

We say that an interval subset $V' \subseteq V$ admits a *sequence representation* if there is a consecutive interval sequence $I_1, I_2, \ldots, I_s$ and a sequence of cliques $Q_1, Q_2, \ldots, Q_{s-1}$ with $Q_i \subseteq C_{I_i I_{i+1}}$ for each $1 \le i < s$ such that $V' = \cup_{i=1}^s V_{I_i} \cup \cup_{i=1}^{s-1} Q_i$. In words, for each interval $I \in V'$, we either have that there is an index $1 \le i \le s$ with $I_i \subseteq I$, or there is an index $1 \le i < s$ with $I \in Q_i$. We schematically depict the overlap structure of the cliques $Q_1, Q_2, \ldots, Q_{s-1}$ in Figure 2. If $I_\infty \in V'$, note that it must then hold for any such sequence representation that $I_s = I_\infty$.
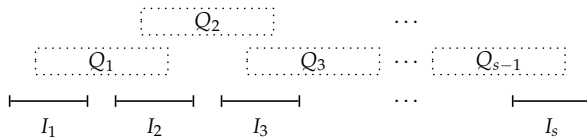
Fig. 2. Overlap structure of the cliques $Q_1, Q_2, \ldots, Q_{s-1}$

The following lemma says that we only have to consider interval subsets that admit a sequence representation.

Lemma 2. *We may assume that any densest $(k+1)$-interval subset $V^* \subseteq V$ with $I_\infty \in V^*$ admits a sequence representation.*

Proof. Given a densest $(k+1)$-interval subset $V^* \subseteq V$ with $I_\infty \in V^*$, we inductively construct the claimed sequence representation. During each iteration, we want to conserve the invariant that the currently constructed interval sequence $I_1, I_2, \ldots, I_i$ and cliques $Q_1, Q_2, \ldots, Q_{i-1}$ are a sequence representation of $\{I \in V^* \mid l_I \le l_{I_i}\}$. Since we extend this interval sequence by one interval

in each iteration, this already gives that we end up with the claimed sequence representation $I_1, I_2, \ldots, I_s$ and $Q_1, Q_2, \ldots, Q_{s-1}$ of V^*. Note that we obtain that finally $I_s = I_\infty$.

To start this inductive construction, let $I_1 \in V^*$ be the interval with leftmost left endpoint l_{I_1} subject to the constraint that there is no interval $I \in V^*$ with $I \subset I_1$. Recall that we know from Lemma 1 that we may assume that $V_{I_1} \subseteq V^*$. Hence, to see that the invariant described above holds, we only have to show that also $\{I \in V^* \mid l_I < l_{I_1}\} \subseteq V_{I_1}$. Recall here the assumption that all left endpoints of intervals are distinct, and thus $I = I_1$ if $l_I = l_{I_1}$. Now, assume for contradiction that there is an interval $I \in V^*$ with $l_I < l_{I_1}$ and $I \notin V_{I_1}$. Hence, we obtain that $r_I < r_{I_1}$. Let then $I' \in V^*$ be an interval with the properties that $I' \subseteq I$ and there is no interval $I'' \in V^*$ with $I'' \subset I'$. If $l_{I'} > l_{I_1}$, then $I' \subset I_1$, which contradicts the selection of I_1. On the other hand, if $l_{I'} < l_{I_1}$, then I_1 is not an interval with leftmost left endpoint, which contradicts the selection of I_1 as well. This shows that the invariant holds after the first iteration. Now assume that, for some $i \geq 1$, we have a sequence representation $I_1, I_2, \ldots, I_i$ and $Q_1, Q_2, \ldots, Q_{i-1}$ of $\{I \in V^* \mid l_I \leq l_{I_i}\}$. If $I_i = I_\infty$, then we are done. Otherwise, let $I_{i+1} \in V^*$ be the interval with leftmost left endpoint $l_{I_{i+1}}$ subject to the constraints that $I_i < I_{i+1}$ and there is no interval $I \in V^*$ with $I \subset I_{i+1}$. Moreover, define $Q_i := V^* \cap C_{I_i I_{i+1}}$. Using nearly the same arguments as for the first iteration shows that the invariant is conserved during each iteration. $\square$

4   Simple Sequence Representations

Consider some fixed but arbitrary small $\epsilon > 0$, and assume that $1/\epsilon$ is integral and $2/\epsilon + 2 \geq 4$. Moreover, let $C \subseteq V$ be some clique, and let $I'_1, I'_2, \ldots, I'_u$ with $r_{I'_1} > r_{I'_2} > \ldots > r_{I'_u}$ be an ordering of the intervals in C according to their right endpoints. Recall here the assumption that all interval endpoints are distinct. We will show how the following *inputs* define a subclique $Q \subseteq C$:

(1) an integer v with $2 \leq v \leq 2/\epsilon + 2$,
(2) an integer sequence $j_1, j_2, \ldots, j_v$ with $j_1 = 1 < j_2 < \ldots < j_v = u + 1$,
(3) an integer sequence $h_1, h_2, \ldots, h_{v-1}$ with $0 \leq h_t \leq j_{t+1} - j_t$ for each $1 \leq t < v$.

We define Q as follows: for each $1 \leq t < v$, Q contains exactly the h_t intervals in $\{I'_{j_t}, I'_{j_t+1}, \ldots, I'_{j_{t+1}-1}\}$ with leftmost left endpoints. Thus, any such combination of inputs defines a subclique $Q \subseteq C$. Let then $P_\epsilon(C)$ be the set of all subcliques constructed in this way for all possible such inputs. Since $|C| = u \leq n$, we immediately obtain the polynomial bound $|P_\epsilon(C)| \leq (2/\epsilon + 1) \cdot n^{2/\epsilon} \cdot n^{2/\epsilon+1}$.

Example. Consider the clique C of intervals $I'_1, I'_2, \ldots, I'_7$ depicted in Figure 3 which are ordered according to their right endpoints. Then, for the inputs $v = 4$, $j_1 = 1 < j_2 = 4 < j_3 = 7 < j_4 = 8$, $h_1 = h_2 = 2$, and $h_3 = 0$, we add the set $Q = \{I'_1, I'_3, I'_5, I'_6\}$ to $P_\epsilon(C)$, which are the intervals with solid lines in Figure 3. To see this, note that I'_1 and I'_3 are the h_1 intervals in $\{I'_1, I'_2, I'_3\}$ with leftmost left endpoints, and I'_5 and I'_6 are the h_2 intervals in $\{I'_4, I'_5, I'_6\}$ with leftmost left endpoints. Since $h_3 = 0$, Q does not contain the single interval in $\{I'_7\}$.
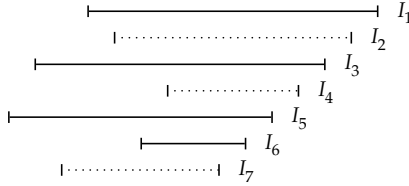
Fig. 3. Example construction of a clique $Q \in P_\epsilon(C)$

Let $V' \subseteq V$ be an interval subset with a sequence representation $I_1, I_2, \ldots, I_s$ and $Q_1, Q_2, \ldots, Q_{s-1}$. We then say that V' admits a *simple sequence representation* if additionally $Q_i \in P_\epsilon(C_{I_i I_{i+1}})$ for each $1 \le i < s$. The following lemma is critical for the correctness of the PTAS, since it allows us to trade the size of the search space for accuracy.

Lemma 3. *There is an interval subset $V' \subseteq V$ with $|V'| = k+1$, $I_\infty \in V'$, and $E(V') \ge (1 - 8\epsilon)\text{OPT}$ that admits a simple sequence representation.*

To prove Lemma 3, we need one preliminary lemma.

Lemma 4. *Consider an interval subset $V' \subseteq V$ with a sequence representation $I_1, I_2, \ldots, I_s$ and $Q_1, Q_2, \ldots, Q_{s-1}$. Then, for any $1 \le i < s$, there exists a clique $Q \in P_\epsilon(C_{I_i I_{i+1}})$ with $|Q| = |Q_i|$ such that replacing the clique Q_i by Q decreases $E(V')$ by at most $\epsilon |Q_i||Q'_i|$, where $Q'_i := V' \cap C'_{I_i I_{i+1}}$. Moreover, if $|Q_i| = 1$, then $E(V')$ does not decrease at all.*

Proof. We abbreviate $C = C_{I_i I_{i+1}}$ and $C' = C'_{I_i I_{i+1}}$ throughout this proof, and let $I'_1, I'_2, \ldots, I'_u$ with $r_{I'_1} > r_{I'_2} > \ldots > r_{I'_u}$ be an ordering of the intervals in C according to their right endpoints. First, consider the case that $|Q_i| = 1$. In this case, there is one input that defines a set $Q \in P_\epsilon(C)$ with exactly $Q = Q_i$. Specifically, if $Q_i = \{I'_t\}$ for $t < u$, then we use the input $j_1 = 1 < j_2 = t < j_3 = t + 1 < j_4 = u + 1$, $h_1 = h_3 = 0$, and $h_2 = 1$. On the other hand, if $Q_i = \{I'_u\}$, then we use the input $j_1 = 1 < j_2 = u < j_3 = u + 1$, $h_1 = 0$, and $h_2 = 1$. This proves the second part of the claim. Hence, we may assume that $|Q_i| > 1$ in what follows.

To define the set $Q \in P_\epsilon(C)$ for the first part of the claim, we have to define the respective inputs of Q used during the construction of $P_\epsilon(C)$. To this end, for each $1 \le j \le u$, let $b_j := E(\{I'_j\}, Q'_i)$ denote the number of intervals in Q'_i which overlap with I'_j. Consequently, since the intervals $I'_1, I'_2, \ldots I'_u \in C$ are ordered according to their right endpoints and all intervals in $Q'_i \subseteq C'$ are right of these intervals as schematically depicted in Figure 1, we obtain that $|Q'_i| \ge b_1 \ge b_2 \ge \ldots \ge b_u$. Using this, we now inductively construct a sequence $j_1, j_2, \ldots, j_v$ with $j_1 = 1 < j_2 < \ldots < j_v = u + 1$ as follows:

Since $j_1 = 1$, the start of this inductive construction is well-defined. Now assume that we have already defined some integers $j_1, j_2, \ldots, j_t$. Then, if $b_{j_t} - b_{j_t+1} > \epsilon|Q_i'|$, set $j_{t+1} := j_t + 1$. Otherwise, let $j_{t+1} > j_t$ be the maximal index such that still $b_{j_t} - b_{j_{t+1}} \leq \epsilon|Q_i'|$. If this index is the last index u, then set $j_{t+1} := u + 1$ instead, and terminate this inductive construction. This gives the integer sequence $j_1, j_2, \ldots, j_v$.

Observe that it holds for any $1 \leq t \leq v - 2$ that $b_{j_t} - b_{j_{t+2}} > \epsilon|Q_i'|$. Consequently, since $b_1 \leq |Q_i'|$, we obtain that $v \leq 2/\epsilon + 2$. Moreover, for each $1 \leq t < v$, we either have that $j_{t+1} = j_t + 1$ or $b_{j_t} - b_{j_{t+1}} \leq \epsilon|Q_i'|$.

Example. Assume that C is the clique from Figure 3, and moreover assume that $\epsilon|Q_i'| = 5$, $b_1 = 10$, $b_2 = 8$, $b_3 = 6$, $b_4 = 6$, $b_5 = 0$, $b_6 = 0$, and $b_7 = 0$. In any case, we have $j_1 = 1$. Consequently, it holds that $j_2 = 4$ is the maximal index such that still $b_{j_1} - b_{j_2} = 4 \leq \epsilon|Q_i'|$. Next, since $b_{j_2} - b_{j_2+1} = 6 > \epsilon|Q_i'|$, we set $j_3 := j_2 + 1$. Finally, $j_4 = 7$ is the maximal index such that still $b_{j_3} - b_{j_4} = 0 < \epsilon|Q_i'|$. However, since this is the last index, we set $j_4 := 8$ instead.

Finally, for each $1 \leq t < v$, define $C_t := \{I_{j_t}', I_{j_t+1}', \ldots, I_{j_{t+1}-1}'\}$, and let $h_t := |Q_i \cap C_t|$. Hence, $\sum_{t=1}^{v-1} h_t = |Q_i|$. The sequences $j_1, j_2, \ldots, j_v$ and $h_1, h_2, \ldots, h_{v-1}$ are the inputs required to define the claimed set $Q \in P_\epsilon(C)$. Because of the definition of the sequence $h_1, h_2, \ldots, h_{v-1}$, we immediately obtain $|Q| = |Q_i|$. For each $1 \leq t < v$, observe that the construction of Q implies that $Q \cap C_t$ contains the h_t intervals in C_t with leftmost left endpoints.

We still have to show that the interval set Q constructed above has the claimed property. First, since $|Q| = |Q_i|$, we have $E(Q) = E(Q_i) = E(V' \cap C)$. Therefore, to bound the decrease of $E(V')$ due to replacing Q_i by Q, we only need to show that $E(Q, V' \backslash C) \geq E(Q_i, V' \backslash C) - \epsilon|Q_i||Q_i'|$. To this end, we partition $V' \backslash C$ into four pairwise disjoint parts:

$$
\begin{aligned}
V^< &:= \{I \in V' \backslash C \mid r_I < r_{I_i}\}, \\
V^= &:= \{I \in V' \backslash C \mid l_I < r_{I_i} \leq r_I\}, \\
V^> &:= \{I \in V' \backslash C \mid r_{I_i} < l_I < r_{I_{i+1}}\}, \\
V^\gg &:= \{I \in V' \backslash C \mid l_I > r_{I_{i+1}}\}.
\end{aligned}
$$

Since these sets are pairwise disjoint, we may consider them separately:

Case $V^<$: Consider some index $1 \leq t < v$, and recall that $Q \cap C_t$ contains the h_t intervals in C_t with leftmost left endpoints. On the other hand, $Q_i \cap C_t$ also contains h_t from C_t, but these are not necessarily the ones with leftmost left endpoints. Consequently, since $V^<$ are the intervals left of r_{I_i} and it holds for any interval $I \in C_t$ that $r_{I_i} \in I$, this shows that $E(Q \cap C_t, V^<) \geq E(Q_i \cap C_t, V^<)$. Combining this for all indices $1 \leq t < v$ finally gives that $E(Q, V^<) \geq E(Q_i, V^<)$.

Case $V^=$: Since $|Q| = |Q_i|$ and $r_{I_i} \in I$ for each interval $I \in C$, we have that $E(Q, V^=) = E(Q_i, V^=)$.

Case $V^{\gg}$: Since no interval $V^{\gg}$ overlaps with an interval in C, we trivially obtain $E(Q, V^{\gg}) = 0 = E(Q_i, V^{\gg})$. Therefore, combining this case with the last two cases, we see that the next case is the only case where $E(V')$ might decrease.

Case $V^{>}$: Since the consecutive interval pair I_i, I_{i+1} is part of a sequence representation of V', there is no interval $I \in V'$ with $r_{I_i} < l_I < r_{I_{i+1}}$ and $r_I < r_{I_{i+1}}$. Therefore, we obtain that $V^{>} = Q'_i := V' \cap C'$. Hence, we only need to upper bound $E(Q_i, Q'_i) - E(Q, Q'_i) = \sum_{t=1}^{v-1} a_t$, where, for each $1 \le t < v$, we define $a_t := E(Q_i \cap C_t, Q'_i) - E(Q \cap C_t, Q'_i)$. Now note that, for each $1 \le t < v$, each interval in C_t overlaps with at least $b_{j_{t+1}}$ and at most b_{j_t} intervals in Q'_i. On the other hand, the sets $Q_i \cap C_t$ and $Q \cap C_t$ both contain exactly h_t intervals. Combining this, we find that $a_t \le h_t(b_{j_t} - b_{j_{t+1}})$, where we define $b_{j_v} := 0$. Therefore, we obtain that

$$\sum_{t=1}^{v-1} a_t = \sum_{t:j_{t+1}>j_t+1} a_t \le \sum_{t:j_{t+1}>j_t+1} h_t(b_{j_t} - b_{j_{t+1}})$$
$$\le \epsilon|Q'_i| \sum_{t:j_{t+1}>j_t+1} h_t \le \epsilon|Q'_i||Q_i|.$$

The equality in the first line is due to the fact that if $j_{t+1} = j_t + 1$, then trivially $Q_i \cap C_t = Q \cap C_t$, since $|C_t| = 1$, and consequently $a_t = 0$. Moreover, the first inequality in the second line is due to the earlier observation that either $j_{t+1} = j_t + 1$ or $b_{j_t} - b_{j_{t+1}} \le \epsilon|Q'_i|$. Hence, by combining all these arguments, we conclude that $E(Q, V^{>}) \ge E(Q_i, V^{>}) - \epsilon|Q_i||Q'_i|$.

Combining all four cases proves the claim.   □

Proof (Lemma 3). Consider a densest $(k+1)$-interval subset $V^* \subseteq V$ with $I_\infty \in V^*$ and $E(V^*) = \text{OPT}$. We know from Lemma 2 that we may assume that V^* admits a sequence representation $I_1, I_2, \ldots, I_s$ and $Q_1, Q_2, \ldots, Q_{s-1}$. Moreover, Lemma 4 implies that, for each $1 \le i < s$, we can replace Q_i by a clique $Q \in P_\epsilon(C_{I_i I_{i+1}})$ with $|Q| = |Q_i|$ such that $E(V^*)$ decreases by at most $\epsilon|Q_i||Q'_i|$, and if $|Q_i| = 1$, then $E(V^*)$ does not decrease at all. Iterating these replacements yields the claimed interval subset V'. We will argue in the following paragraph that these replacements can indeed be done iteratively without amplifying these decreases.

Assume that, for some $1 \le i < s$, we have already replaced Q_t for each $t > i$ as described above. However, we obtain that the only such replacement that might affect the next replacement of Q_i by some clique $Q \in P_\epsilon(C_{I_i I_{i+1}})$ is the one with $t = i+1$. Therefore, consider the clique $Q' \in P_\epsilon(C_{I_{i+1} I_{i+2}})$ that replaced Q_{i+1}. Because we always select intervals with leftmost left endpoints during the construction of Q' and $|Q_{i+1}| = |Q'|$, we can identify each interval $I \in Q_{i+1}$ with an interval $I' \in Q'$ with $l_{I'} \le l_I$. Hence, since $Q \subseteq C_{I_i I_{i+1}}$ is left of $C_{I_{i+1} I_{i+2}}$, we even have that $E(Q, Q') \ge E(Q, Q_{i+1})$. Consequently, replacing Q_i by Q will still decrease $E(V^*)$ by at most $\epsilon|Q_i||Q'_i|$, even if we replace Q_{i+1} by Q' first.

To bound the total decrease of $E(V^*)$, observe that

$$E(V^*) - E(V') \leq \sum_{i:|Q_i|>1} \epsilon |Q_i||Q_i'|$$

$$\leq \epsilon \left(\sum_{i=1}^{s-1} 2|Q_i|(|Q_i|-1) + \sum_{i=1}^{s-1} 2|Q_i'|(|Q_i'|-1) \right)$$

$$= \epsilon \left(\sum_{i=1}^{s-1} 4E(Q_i) + \sum_{i=1}^{s-1} 4E(Q_i') \right) \leq \epsilon 8 E(V^*)$$

which completes the proof of the lemma. The first line is due to Lemma 4 and the arguments above. The second line is due to the simple observation that $ab \leq 2(a(a-1)+b(b-1))$ for any pair of integers $a > 1$ and $b \geq 1$. Moreover, the third line uses the fact that $E(C) = \binom{|C|}{2} = |C|(|C|-1)/2$ for any clique $C \subseteq V$. Finally, the fourth line is due to the fact that the cliques $Q_1, Q_2, \ldots, Q_{s-1} \subseteq V^*$ are pairwise disjoint, and hence $\sum_{i=1}^{s-1} E(Q_i) \leq E(V^*)$. The same holds for the cliques $Q_1', Q_2', \ldots, Q_{s-1}' \subseteq V^*$. □

5   Dynamic Programming

We have a dynamic programming array Π with integral entries of the form $\Pi(h, s, I_{s-1}, I_s, Q_{s-1})$, where h is an integer with $0 \leq h \leq k+1$, s is an integer with $2 \leq s \leq n$, $I_{s-1}, I_s \in V$ are two consecutive intervals, and $Q_{s-1} \in P_\epsilon(C_{I_{s-1}I_s})$. The indexing of I_{s-1}, I_s, and Q_{s-1} with s is just for convenience. Our goal is to fill this array such that $\Pi(h, s, I_{s-1}, I_s, Q_{s-1}) = E(V')$ for an interval subset $V' \subseteq V$ with $|V'| = h$ that maximizes $E(V')$ subject to the constraint that V' admits a simple sequence representation $I_1, I_2, \ldots, I_s$ and $Q_1, Q_2, \ldots, Q_{s-1}$. Hence, only the last parts in this sequence representation are defined by the entry. We can bound the size of Π by $n^4 \cdot \max_{I_{s-1} < I_s} |P_\epsilon(C_{I_{s-1}I_s})|$. The second part of this product is simply the maximal size of $P_\epsilon(C_{I_{s-1}I_s})$ for any consecutive interval pair I_{s-1}, I_s. Consequently, since we already know that $P_\epsilon(C_{I_{s-1}I_s})$ has polynomial size for any such interval pair $I_{s-1}, I_s \in V$, we immediately obtain that the array Π has polynomial size as well.

We initialize Π by filling all entries of the form $\Pi(h, 2, I_1, I_2, Q_1)$ with $|Q_1 \cup V_{I_1} \cup V_{I_2}| = h$. Specifically, for any integer h with $0 \leq h \leq k+1$, any consecutive interval pair $I_1, I_2 \in V$, and any interval set $Q_1 \in P_\epsilon(I_1, I_2)$ with $|Q_1 \cup V_{I_1} \cup V_{I_2}| = h$, we set $\Pi(h, 2, I_1, I_2, Q_1) := E(Q_1 \cup V_{I_1} \cup V_{I_2})$. All other entries of Π are initialized as $-\infty$.

To define a recurrence relation, assume that we have already filled all entries of the form $\Pi(h, s-1, I_{s-2}, I_{s-1}, Q_{s-2})$, and now we want to use them to fill all entries of the form $\Pi(h, s, I_{s-1}, I_s, Q_{s-1})$, i.e., we want to increase the sequence length s by one. To this end, we can use the following recurrence relation:

$$\Pi(h, s, I_{s-1}, I_s, Q_{s-1}) =$$

$$\max_{I_{s-2} < I_{s-1}, Q_{s-2} \in P_\epsilon(C_{I_{s-2}I_{s-1}})} \left\{ \Pi(h - |Q_{s-1}| - |V_{I_s} \backslash V_{I_{s-1}}|, s-1, I_{s-2}, I_{s-1}, Q_{s-2}) \right.$$

$$\left. + E(Q_{s-2} \cup V_{I_{s-1}}, Q_{s-1} \cup V_{I_s} \backslash V_{I_{s-1}}) + E(Q_{s-1} \cup V_{I_s} \backslash V_{I_{s-1}}) \right\}$$

In words, we take the maximum over all intervals $I_{s-2} \in V$ with $I_{s-2} < I_{s-1}$ and all cliques $Q_{s-2} \in P_\epsilon(C_{I_{s-2}I_{s-1}})$. Since $P_\epsilon(C_{I_{s-1}I_s})$ has polynomial size as already used above, this recurrence relation can be clearly implemented in polynomial time. Hence, in combination with the size of Π listed above, we find that it takes polynomial time to fill Π.

To see the correctness of this recurrence relation, consider an interval subset $V' \subseteq V$ that realizes an entry of the form $\Pi(h, s, I_{s-1}, I_s, Q_{s-1})$, and let $I_1, I_2, \ldots, I_s$ and $Q_1, Q_2, \ldots, Q_{s-1}$ be a simple sequence representation of V'. Let then $V'' \subseteq V'$ be the subset with the shorter simple sequence representation $I_1, I_2, \ldots, I_{s-1}$ and $Q_1, Q_2, \ldots, Q_{s-2}$. Hence, we have that $V'' = V' \backslash (Q_{s-1} \cup V_{I_s} \backslash V_{I_{s-1}})$. Now observe that, as schematically illustrated in Figure 4, the only intervals in V'' which might overlap with an interval in $V' \backslash V'' = Q_{s-1} \cup V_{I_s} \backslash V_{I_{s-1}}$ are the ones in $Q_{s-2} \cup V_{I_{s-1}}$. Consequently, we obtain the decomposition $E(V') = E(V'') + E(Q_{s-2} \cup V_{I_{s-1}}, Q_{s-1} \cup V_{I_s} \backslash V_{I_{s-1}}) + E(Q_{s-1} \cup V_{I_s} \backslash V_{I_{s-1}})$ as used in the recurrence relation. Thus, since V' minimizes $E(V')$, we have that also V'' minimizes $E(V'')$, since we could otherwise improve V' by replacing the intervals V'' by another interval subset. This shows that $E(V'')$ realizes the entry $\Pi(h - |Q_{s-1}| - |V_{I_s} \backslash V_{I_{s-1}}|, s-1, I_{s-2}, I_{s-1}, Q_{s-2})$. Combining these facts shows the correctness of the recurrence relation.
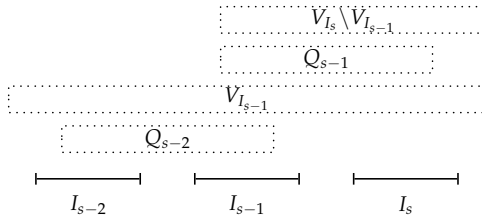
Fig. 4. Overlap structure during a recurrence relation

Theorem 1. *There is a PTAS for finding a densest k-subgraph in interval graphs.*

Proof. Use the dynamic programming approach explained above to compute all entries $\Pi(h, s, I_{s-1}, I_s, Q_{s-1})$ with $h = k + 1$ and $I_s = I_\infty$, and let V' be the interval set that realizes an optimal such entry, i.e., one that maximizes $E(V') = \Pi(h, s, I_{s-1}, I_s, Q_{s-1})$. By the definition of Π, we obtain that V' is a

densest $(k + 1)$-interval subset with $I_\infty \in V'$ subject to the constraint that V' admits a simple sequence representation. Consequently, by Lemma 3, we find that $E(V' \backslash \{I_\infty\}) = E(V') \geq (1 - 8\epsilon)$OPT. This proves the claim.   □

References

1. Arora, S.: Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. Journal of the ACM 45(5), 753–782 (1998)
2. Arora, S., Karger, D.R., Karpinski, M.: Polynomial time approximation schemes for dense instances of np-hard problems. J. Comput. Syst. Sci. 58(1), 193–210 (1999)
3. Asahiro, Y., Iwama, K., Tamaki, H., Tokuyama, T.: Greedily finding a dense subgraph. J. Algorithms 34(2), 203–221 (2000)
4. Backer, J., Keil, J.M.: Constant factor approximation algorithms for the densest k-subgraph problem on proper interval graphs and bipartite permutation graphs. Inf. Process. Lett. 110(16), 635–638 (2010)
5. Bhaskara, A., Charikar, M., Chlamtac, E., Feige, U., Vijayaraghavan, A.: Detecting high log-densities: an $O(n^{1/4})$-approximation for densest k-subgraph. In: Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC 2010), pp. 201–210 (2010)
6. Chen, D.Z., Fleischer, R., Li, J.: Densest k-subgraph approximation on intersection graphs. In: Jansen, K., Solis-Oba, R. (eds.) WAOA 2010. LNCS, vol. 6534, pp. 83–93. Springer, Heidelberg (2011)
7. Feige, U., Langberg, M.: Approximation algorithms for maximization problems arising in graph partitioning. J. Algorithms 41(2), 174–211 (2001)
8. Feige, U., Peleg, D., Kortsarz, G.: The dense k-subgraph problem. Algorithmica 29(3), 410–421 (2001)
9. Golumbic, M.C., Trenk, A.N.: Tolerance Graphs. Cambridge University Press, Cambridge (2004)
10. Gröschel, M., Lovász, L., Schrijver, A.: Geometric Algorithms and Combinatorial Optimization. Springer, Heidelberg (1988)
11. Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problems. Journal of the ACM 22 (1975)
12. Khot, S.: Ruling out ptas for graph min-bisection, dense k-subgraph, and bipartite clique. SIAM J. Comput. 36(4), 1025–1071 (2006)
13. Kortsarz, G., Peleg, D.: On choosing a dense subgraph. In: Proceedings of the 34th Annual Symposium on Foundations of Computer Science (FOCS 1993), pp. 692–701 (1993)
14. Lawler, E.L.: Combinatorial optimization - networks and matroids. Holt, Rinehart and Winston, New York (1976)
15. Liazi, M., Milis, I., Pascual, F., Zissimopoulos, V.: The densest k-subgraph problem on clique graphs. J. Comb. Optim. 14(4), 465–474 (2007)
16. Liazi, M., Milis, I., Zissimopoulos, V.: A constant approximation algorithm for the densest k-subgraph problem on chordal graphs. Inf. Process. Lett. 108(1), 29–32 (2008)
17. Perl, Y., Corneil, D.G.: Clustering and domination in perfect graphs. Discrete Applied Mathematics 9(1), 27–39 (1984)

Improved Distance Queries in Planar Graphs

Yahav Nussbaum[*]

The Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel
`yahav.nussbaum@cs.tau.ac.il`

Abstract. There are several known data structures that answers distance queries between two arbitrary vertices in a planar graph. The tradeoff is among preprocessing time, storage space and query time. In this paper we present three data structures that answer such queries, each with its own advantage over previous data structures. The first one improves the query time of data structures of linear space. The second improves the preprocessing time of data structures with a space bound of $O(n^{4/3})$ or higher while matching the best known query time. The third data structure improves the query time for a similar range of space bounds, at the expense of a longer preprocessing time. The techniques that we use include modifying the parameters of planar graph decompositions, combining the different advantages of existing data structures, and using the Monge property for finding minimum elements of matrices.

1 Introduction

There are several known data structures that answer distance queries in planar graphs. We survey them below. All of these data structures use the following basic idea. They split the graph into *pieces*, where each piece is connected to the rest of the graph only through its *boundary vertices*. Then, every path can go from one piece to another only through these boundary vertices. The different data structures find different efficient ways to store or compute the distance between two boundary vertices or between a boundary vertex and a non-boundary vertex.

Frederickson [10] gave the first data structures that answers distance queries in planar graphs fast. He gave a data structure of linear size and $O(n \log n)$ preprocessing time that finds the shortest path tree rooted at any vertex in $O(n)$ time, where n is the number of vertices in the graph. This leads also to an $O(n^2)$ time algorithm to compute all-pairs shortest-paths in a planar graph, and implies a distance query data structure of size $O(n^2)$ with $O(1)$ query time. Feuerstein and Marchetti-Spaccamela [9] modified the data structure of [10] and showed how to decrease the time of a distance query by increasing the preprocessing time. They do not provide an analysis of their data structure in terms of preprocessing time, storage space, and query time, but they show the total running time of k queries, which is $O(nk + n \log n)$, $O(n^{4/3}k^{1/3})$, $O(n^{5/3})$, $O(n\sqrt{k})$ for $k \leq \sqrt{n}$, $\sqrt{n} \leq k \leq n$, $n \leq k \leq n^{4/3}$, $n^{4/3} \leq k \leq n^2$, respectively. This solution actually

consists of three different data structures for the three cases $k \leq \sqrt{n}$, $\sqrt{n} < k \leq n$ and $n < k$, where the data structure for the first case is the one of [10].

Henzinger, Klein, Rao and Subramanian [11] gave an $O(n)$ time algorithm for the single-source shortest path problem. This implies a trivial distance query data structure, which uses the algorithm, and takes $O(n)$ space and query time.

Djidjev [6] gave three data structures. We will use the specific section number in [6] – §3, §4, or §5, to refer to each one of them. The first one [6, (§3)] works for $S \in [n^{3/2}, n^2]$ and has size $O(S)$, $O(S)$ preprocessing time, and $O(n^2/S)$ query time. The same data structure was also presented by Arikati et al. [3]. This data structure is similar to the two data structures of [9], but takes advantage of the algorithm of [11] to get a better preprocessing time. The second [6, (§4)] works for $S \in [n, n^{3/2}]$ and has size $O(S)$, $O(n\sqrt{S})$ preprocessing time, and $O(n^2/S)$ query time. The third data structure [6, (§5)] works for $S \in [n^{4/3}, n^2]$, has size $O(S)$, $O(n\sqrt{S})$ preprocessing time, and $O(n \log(n/\sqrt{S})/\sqrt{S})$ query time. Chen and Xu [5] presented a data structure with the same time and space bounds.[1]

Fakcharoenphol and Rao [8] gave a data structure with $O(n \log n)$ space, $O(n \log^3 n)$ preprocessing time and $O(\sqrt{n} \log^2 n)$ query time. Klein [13] improved the preprocessing time of this data structure to $O(n \log^2 n)$.

Cabello [4] presented a data structure of $O(S)$ space and construction time for $S \in [n^{4/3} \log^{1/3} n, n^2]$, and $O(n \log^{3/2}(n)/\sqrt{S})$ query time. This data structure answers k queries in a total of $O(n^{4/3} \log^{1/3} n + k^{2/3} n^{2/3} \log n)$ time. If the queries are known in advance, the algorithm of [4] avoids storing the entire structure, and uses only $O(n + k)$ space.

In this paper we present three new data structures for the problem:

- Section 3: A data structure with $O(n)$ space, $O(n \log n)$ preprocessing time, and $O(n^{1/2+\varepsilon})$ query time, for any constant $\varepsilon > 0$. This data structure has the best known query time achievable with linear space. This also improves the total running time for answering k distance queries when k is $O(n^{1/2-\varepsilon})$ and $\omega(\log n)$, and if we limit ourselves to data structures with $O(n + k)$ space then the upper bound on the range of k grows to $O(n^{5/6-\varepsilon})$. The data structure is based on the data structure of [8], by combing the *recursive decomposition* of [8] with *r-decomposition* as in [10]. As the data structure of [8], our data structure also generalizes to graphs embedded in a surface of bounded genus.
- Section 4: For $S \in [n^{4/3}, n^2]$, a data structure with $O(S)$ space, $O(S \log n)$ preprocessing time, and $O(n \log(n/\sqrt{S})/\sqrt{S})$ query time. This data structure matches the query time of [5,6, (§5)], which is the best previously known for this range of storage space, with a better preprocessing time for $S = o(n^2/\log^2 n)$. We obtain this data structure by combining a preprocessing algorithm similar to [4] with a data structure similar to [6, (§5)].

---

[1] Djidjev [6, (§5)] presents his result for $S \in [n^{4/3}, n^{3/2}]$ with $O(n \log(n)/\sqrt{S})$ query time, however the same data structure works within the bounds stated here. Chen and Xu [5] do not bound the time and space of the data structure in terms of S, the bounds here are derived by setting $r = n^2/S$ in the bounds that appear below Lemma 28 (page 477) of [5].

Table 1. Comparison of distance query data structures for planar graphs. Time bounds are expressed as a function of the storage space. The data structures are ordered by decreasing storage space and then by decreasing query time.

Reference	Storage space $O(S)$	Query time	Preprocessing time
[10,11]	$S = n^2$	$O(1)$	$O(n^2)$
[3,6, (§3)]	$S \in [n^{3/2}, n^2]$	$O(n^2/S)$	$O(S)$
[4]	$S \in [n^{4/3} \log^{1/3} n, n^2]$	$O(n \log^{3/2}(n)/\sqrt{S})$	$O(S)$
[5,6, (§5)]	$S \in [n^{4/3}, n^2]$	$O(n \log(n/\sqrt{S})/\sqrt{S})$	$O(n\sqrt{S})$
Section 4	$S \in [n^{4/3}, n^2]$	$O(n \log(n/\sqrt{S})/\sqrt{S})$	$O(S \log n)$
Section 5	$S \in [n^{4/3}, n^2]$	$O(n/\sqrt{S})$	$O((S^{3/2}/\sqrt{n}) \log^2 n)$
[6, (§4)]	$S \in [n, n^{3/2}]$	$O(n^2/S)$	$O(n\sqrt{S})$
[8]+[13]	$S = n \log n$	$O(\sqrt{n} \log^2 n)$	$O(n \log^2 n)$
[10]	$S = n$	$O(n)$	$O(n \log n)$
[11]	$S = n$	$O(n)$	—
Section 3	$S = n$	$O(n^{1/2+\varepsilon})$	$O(n \log n)$

 – Section 5: For $S \in [n^{4/3}, n^2]$, a data structure with $O(S)$ space, requir-
   ing $O((S^{3/2}/\sqrt{n}) \log^2 n)$ preprocessing time, and $O(n/\sqrt{S})$ query time. This
   data structure improves the query time for the same range of storage space
   as the previous structure, but with a longer preprocessing time. We obtain
   the fast query time with an efficient minimum search in a Monge matrix.

The different data structures are summarized in Table 1.

2   Preliminaries

We consider a directed simple planar graph G. We let $n = |V(G)|$, and by Euler's formula $|E(G)| = O(n)$. We assume that G is given with a fixed planar embedding, in other words it is a *plane graph*. Without loss of generality we assume that G is a triangulated, bounded degree graph; this assumption is required by algorithms that we use and are described in Sect. 2.1 and 2.2. We assume that G is connected, as we can handle each connected component separately.

Every edge in $E(G)$ has a non-negative *length*. The length of a *path* is the sum of lengths of its edges. The *distance* from a vertex u to a vertex v is the minimum length of a path from u to v. With additional $O(n \log^2 n/ \log \log n)$ preprocessing time we can allow negative edge lengths as well, using a *reduced length* technique, see [8,14,18] for details.

Let F, H be subgraphs of G. We write $d_H(u,v)$ to denote the distance from u to v in H. The graph $F \cap H$ is the subgraph induced by $E(F) \cap E(H)$. For short we denote $|H| = |V(H)|$.

2.1   Decomposition

A *decomposition* of a planar graph G is a set of subgraphs of G, such that each edge is in exactly one subgraph and each vertex of G is in at least one subgraph. Each of the subgraphs which define the decomposition is called a *piece*.

A vertex $v \in V(B)$ is a *boundary vertex* of the piece B, if it is incident to some edge not in $E(B)$, and it is an *internal vertex* otherwise. The set of all boundary vertices of B is the *boundary* of B, denoted by ∂B. A *hole* is a face of B (including the external face) that is not a face of G. For a hole H we denote by H also the subgraph of G inside H. A *boundary walk* of B is a facial walk of B around a hole H. For a piece B with hole H we denote $\partial B[H] = \partial B \cap V(H)$.

All distance query data structures mentioned in the introduction decompose the planar graph. They take advantage of the fact that a path can go from one piece to another only through boundary vertices.

A *recursive decomposition* [8] is obtained by starting with G itself being the only piece in level 0 of the decomposition. At each level, we split each piece B with $|B|$ vertices and $|\partial B|$ boundary vertices that has more than one edge into two pieces, each with at most $2|B|/3$ vertices and at most $2|\partial B|/3 + O(\sqrt{|B|})$ boundary vertices. We require that the boundary vertices of a piece B are also boundary vertices of the subpieces of B. A property of this decomposition is that each piece B has $O(\sqrt{|B|})$ boundary vertices.

An *r-decomposition* [10] is a decomposition of the graph into $O(n/r)$ pieces, each of size at most r with $O(\sqrt{r})$ boundary vertices.

Fakcharoenphol and Rao [8] showed how to find a recursive decomposition of G, such that each piece is connected and has at most a constant number of holes. They use these two properties for their distance algorithm. The construction of the decomposition takes $O(n \log n)$ time using $O(n \log n)$ space, and is done by recursively applying the separator algorithm of Miller [16]. Frederickson [10] showed how to find an r-decomposition in $O(n \log n)$ time and $O(n)$ space by recursively applying the separator algorithm of Lipton and Tarjan [15]. Thus, an r-decomposition is a limited type of recursive decomposition where we stop the recursion earlier (when we get to pieces of size r), and do not store all the levels of the recursion (we store only the leaves). Cabello [4] combined the two constructions of [8] and [10] (using [16] instead of [15]) and constructed an r-decomposition with the properties that the number of holes per piece is bounded by a constant, and that each piece is connected.

In Sect. 3 we use a combination of recursive decomposition and r-decomposition – we decompose the graph recursively, but we decompose each piece into $O(n/r)$ pieces instead of two. In Sect. 4 we use r-decomposition. In Sect. 5 we use r-decomposition as well, there we take advantage of the fact that the construction of an r-decomposition is the same as of a recursive decomposition, which was stopped earlier.

2.2 The Dense Distance Graph

Fakcharoenphol and Rao [8] define the *dense distance graph* of a recursive decomposition. For each piece B in the recursive decomposition they add a piece to the dense distance graph that contains the vertices of ∂B and for every $u, v \in \partial B$ an edge from u to v of length $d_B(u, v)$. The multiple-source shortest paths algorithm of Klein [13] finds k distances where the sources of all of them are on the same face in $O((k + n) \log n)$ time. Therefore, using [13] it takes $O(|B| \log |B|)$

time to find the part of the dense distance graph that corresponds to a piece B (recall that $|\partial B| = O(\sqrt{|B|})$ and B has a constant number of holes). It thus takes $O(n \log^2 n)$ time to construct the dense distance graph over all pieces of the recursive decomposition.

Every single edge defines a piece in the base of the recursive decomposition, so it is clear that the distance from u to v in the dense distance graph is the same as the distance between these two vertices in the original graph. Fakcharoenphol and Rao noticed that in order to find the distance from u to v we do not have to search the entire dense distance graph, but that it suffices to consider only edges that correspond to shortest paths between boundary vertices in a limited number of pieces. The pieces are these containing either u or v, and their siblings in the recursive decomposition. There are $O(\log n)$ such pieces with a total of $O(\sqrt{n})$ boundary vertices. Fakcharoenphol and Rao gave an implementation of Dijkstra's algorithm that runs over a subgraph of the dense distance graph with q vertices, defined by a subset of the pieces in the recursive decomposition, in $O(q \log^2 n)$ time. This gives the $O(\sqrt{n} \log^2 n)$ query time of their data structure.

We use dense distance graphs in two of our data structures (Sect. 3 and 5). In both cases it is on a variant of recursive decomposition, as discussed above.

2.3   The Monge Property

A $p \times q$ matrix M satisfies the *Monge property* if for every two rows $i \le k$ and two columns $j \le \ell$, M satisfies $M_{ij} + M_{k\ell} \le M_{i\ell} + M_{kj}$. We can find the minimum element of M using the SMAWK algorithm [1]. If we do not store M explicitly, but are able to retrieve each entry in $O(1)$ time this takes $O(p + q)$ time. Note that if we add a constant to an entire row or to an entire column of a matrix with the Monge property, then the property remains.

Consider two disjoint sets X and Y of consecutive boundary vertices on a boundary walk of some piece B. Rank the vertices of X from x_1 to $x_{|X|}$ according to their order around the boundary walk, and rank the vertices of Y from y_1 to $y_{|Y|}$ according to their order in the opposite direction around the boundary walk. For $i \le k$ and $j \le \ell$, the shortest path from x_i to y_ℓ inside B and the shortest path from x_k to y_j inside B must cross each other. Let w be a vertex common to both paths. Then, $d_B(x_i, y_j) + d_B(x_k, y_\ell) \le d_B(x_i, w) + d_B(w, y_j) + d_B(x_k, w) + d_B(w, y_\ell) \le d_B(x_i, y_\ell) + d_B(x_k, y_j)$ (see Fig. 1). Therefore, the matrix M such that $M_{ij} = d_B(x_i, y_j)$ has the Monge property. The Monge property was first used explicitly for distance queries in planar graphs by [8].

A *partial matrix* is a matrix that may have some blank entries. In a *falling staircase matrix* the non-blank entries are consecutive in each row starting not before the first non-blank entry of the previous row and ending at the end of the row, *inverse falling staircase matrix* is defined similarly by exchanging the positions of the non-blanks and the blanks (see Fig. 2). Aggarwal and Klawe [2] presented a technique that allows us to find the minimum of an (inverse) falling staircase matrix whose non-blank entries satisfy the Monge property in $O(q + p)$ time by filling the blanks with appropriate values and using the SMAWK [1] algorithm.
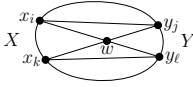
Fig. 1. The distances from X to Y satisfy the Monge property

Fig. 2. A falling staircase matrix (left) and an inverse falling staircase matrix (right)

In Sect. 5 we use this tool for finding the minimum of two staircase matrices whose non-blank entries satisfy the Monge property.

3   Linear-Space Data Structure

In this section we present a data structure with linear space, almost linear preprocessing time, and query time faster than any previous data structure of linear space. We generalize the data structure of Fakcharoenphol and Rao [8] by combining a recursive decomposition of the graph with r-decomposition. This is similar to the way that Mozes and Wulff-Nilsen [18] improved the shortest path algorithm of Klein, Mozes and Weimann [14]. Mozes and Sommer [17] have independently obtained a similar result.

We find an r-decomposition of G into $p = O(n/r)$ pieces, for some $p \in [2, n]$, and then we recursively decompose each piece into p subpieces, until we get to pieces with a single edge. The depth of the decomposition is $O(\log n / \log p)$ where at level i we have p^i pieces, each of size $O(n/p^i)$ and with $O(\sqrt{n/p^i})$ boundary vertices. Constructing this recursive decomposition takes $O(n \log n \frac{\log n}{\log p})$ time. An alternative way to describe this decomposition is to perform a recursive decomposition on G while storing only levels $k \log p$ for $k = 0, \ldots, \lfloor \log n / \log p \rfloor$ of the recursion tree and the leaves of the recursion (the pieces containing single edges).

We compute the dense distance graph for the recursive decomposition, in the same way as in the data structure of [8]. That is, we compute the distance between every pair of boundary vertices in each piece. Using the algorithm of Klein [13] this takes $O(n \log n)$ time for each level, and a total of $O(n \log n \frac{\log n}{\log p})$ time. The size of dense distance graph over our recursive decomposition is $O(n \frac{\log n}{\log p})$.

When a distance query from u to v arrives, we use the Dijkstra implementation of [8] to answer it. We run the algorithm on the subgraph of the dense distance graph that includes all the pieces that contain either u or v, and the $p-1$ siblings in the recursive decomposition of each such piece. We require the sibling pieces because the shortest path can get out of a piece B into a sibling of B without getting out of any piece that contains B. Therefore, the number of boundary vertices involved in each distance query is $O(\sum_{i=1}^{\log n / \log p} p \sqrt{n/p^i}) = O(p\sqrt{n})$. Hence the query time using the algorithm of [8] is $O(p\sqrt{n} \log^2 n)$.

We conclude that for a planar graph with n vertices and any $p \in [2, n]$, we can construct in $O(n \log^2 n / \log p)$ time a data structure of size $O(n \log n / \log p)$

that computes the distance between any two vertices in $O(p\sqrt{n}\log^2 n)$ time. If we set $p = 2$ we get exactly the data structure of [8]. If we set $p = n^\delta$ for a constant $0 < \delta < \varepsilon$ we get:

Theorem 1. *For a planar graph with n vertices and any constant $\varepsilon > 0$, we can construct in $O(n \log n)$ time a data structure of size $O(n)$ that computes the distance between any two vertices in $O(n^{1/2+\varepsilon})$ time.*

The total time for k distance queries is $O(n \log n + kn^{1/2+\varepsilon})$ and the required space is $O(n + k)$. This improves the fastest time for k distance queries for $k = O(n^{1/2-\varepsilon})$, or $k = O(n^{5/6-\varepsilon})$ if we consider only data structures of $O(n+k)$ space, and $k = \omega(\log n)$ simultaneously.

Fakcharoenphol and Rao [8] noted that Smith suggested that their algorithm can be generalized to graphs of bounded genus. If a graph G with bounded vertex degree is embedded in an orientable surface of genus g, then [7,12] showed how to find a *planarizing set* of $O(\sqrt{ng})$ edges whose removal from the graph makes the graph planar, in $O(n + g)$ time. We use the planarizing set for the first decomposition of the graph, and combine the Dijkstra implementation of [8] with standard implementation using a heap for the topmost pieces in the recursion. We get that the bounds of Theorem 1 apply also to graphs embedded in an orientable surface of a fixed genus.

4   Improved Preprocessing Time for $S \in [n^{4/3}, n^2]$

In this section we present a data structure that matches the space-query time tradeoff of the data structures of Djidjev [6, (§5)] and Chen and Xu [5] with the preprocessing time of the data structure of Cabello [4]. Our data structure combines parts from the data structures of [6, (§5)] and of [4].

First, we construct an r-decomposition of G in $O(n \log n)$ time, for some parameter $r \in (0, n)$. For each piece B our data structure has three parts:

(i) The distances $d_G(u, v)$ and $d_G(v, u)$ for every $u \in \partial B$ and $v \in V(B)$.
(ii) A data structure that reports $d_B(u, v)$ in $O(\sqrt{r})$ time for $u, v \in V(B)$.
(iii) For each hole H of B we store $d_H(u, v)$ for every $u \in \partial B[H]$ and $v \in V(H)$ such that v is a boundary vertex of some piece contained in H.

Part (i) is from the data structure of Cabello [4]. The construction of this part requires $O(n \log n + r^{3/2})$ time and $O(n + r^{3/2})$ space per piece [4]. Part (ii) was used both by Djidjev [6, (§5)] and by Cabello [4]. This is the data structure of [3,6, (§3)] with $S = r^{3/2}$, its construction takes $O(r^{3/2})$ time and space per piece. Part (iii) is from the data structure of [6, (§5)], but we construct it more efficiently. We find the distances for this part using the multiple-source shortest paths algorithm of Klein [13] for every boundary walk. The required space per piece for part (iii) is $O(n)$ and the preprocessing time is $O(n \log n)$.

Since there are $O(n/r)$ pieces, each with a constant number of holes and $O(\sqrt{r})$ boundary vertices, constructing the three parts for the entire graph takes a total of $O((n^2/r) \log n + n\sqrt{r})$ time and $O(n^2/r + n\sqrt{r})$ space.

Let u, v be a query pair. We use the data structure of this section to find $d_G(u, v)$ in $O(\sqrt{r} \log r)$ time. If u and v are in the same piece then we find the distance from u to v using parts (i) and (ii) of the data structure with the query algorithm of [4] in $O(\sqrt{r})$ time (see details in Sect. 5.2 below). If u and v are in different pieces then we find the distance using parts (i) and (iii) with the query algorithm of [6, (§5)] in $O(\sqrt{r} \log r)$ time.

We conclude that for a planar graph with n vertices and any $r \in (0, n)$, we can construct in $O((n^2/r) \log n + n\sqrt{r})$ time a data structure of size $O(n^2/r + n\sqrt{r})$ that computes the distance between any two vertices in $O(\sqrt{r} \log r)$ time. The sum $n^2/r + n\sqrt{r}$ minimizes at $n^{4/3}$, and for $r = n^2/S$ we get:

Theorem 2. *For a planar graph with n vertices and $S \in [n^{4/3}, n^2]$, we can construct in $O(S \log n)$ time a data structure of size $O(S)$ that computes the distance between any two vertices in $O(n \log(n/\sqrt{S})/\sqrt{S})$ time.*

5   Improved Query Time for $S \in [n^{4/3}, n^2]$

In this section we present a data structure with an improved query time, for the same range of space bounds as in the previous section. In return, the preprocessing time is higher. For this purpose we use minimum search in Monge matrices. While previous planar distance data structures have taken advantage of the Monge property before, this is the first to use fast minimum search in a Monge matrix with the SMAWK algorithm [1].

Again, we construct an r-decomposition of G. Assume that we want to find the distance from a vertex u to a vertex v that are in two different pieces. Let B and B' be the different pieces that contain u and v respectively, let H and H' be the holes of B and B' that contain v and u respectively, and let $X = \partial B[H]$ and $Y = \partial B'[H']$. Let $J = H \cap H'$ be the subgraph of G contained both in H and in H'. We assume without lost of generality that J contains the infinite face.

The shortest path from u to v must contain a vertex $x \in X$ and a vertex $y \in Y$ (it is possible that $x = y$). We assume that there is no internal vertex of B or B' between x and y in this path, since otherwise we can replace x with a later vertex. Therefore, $d_G(u, v) = \min_{x \in X, y \in Y} \{d_G(u, x) + d_J(x, y) + d_G(y, v)\}$.

Our goal then is to find x, y that minimize $d_G(u, x) + d_J(x, y) + d_G(y, v)$. For a particular order of X and of Y, which we specify below, let M be the matrix such that $M_{ij} = d_J(x_i, y_j)$, and N be the matrix such that $N_{ij} = d_G(u, x_i) + d_J(x_i, y_j) + d_G(y_j, v)$. We show how to order the members of X and Y such that M decomposes into two staircase matrices, each with the Monge property. Since $d_G(u, x_i)$ is fixed for a fixed x_i, and $d_G(y_j, v)$ is fixed for a fixed y_j, then N also consists of two staircase matrices with the Monge property. Thus we can use the algorithm of Aggarwal and Klawe [2] to find the minimum entry of N, which is the desired distance.

For every $x \in X$ we define the *leftmost shortest path from x to Y*, denoted by $L(x)$ as follows. We add to the embedding of J a vertex u' inside B and connect it with an edge to x, and a vertex v' inside B' and connect with an edge every vertex of Y to v' (recall the internal vertices of B and B' are not in J). We set

the length of all new edges to be 0. An edge $e = (w, z) \in E(J)$ is called *tight* if the length of e is equal to $d_J(u', z) - d_J(u', w)$, that is if e is on some shortest path from u' to z. We remove all non-tight edges from the graph, and perform a *left-first search* from u' until we find v' (i.e. we perform a depth-first search from u', and visit the edges outgoing from a specific vertex according to their left-to-right order, see also [13]). Let $L(x)$ be the path we obtain by removing the first and the last edges of the leftmost path we found from u' to v', and let $\ell(x) \in Y$ be the last vertex of $L(x)$. Note that $L(x)$ is a shortest path from x to $\ell(x)$. The reason we added u' is to decide between two paths that diverge at x itself, the reason we added v' is to decide between two paths such that one is a prefix of the other. Note that $L(x)$ may contain more than one vertex of Y. Moreover, even if $x \in X \cap Y$ it is not necessarily true that $\ell(x) = x$.

Fix some arbitrary vertex of X to be x_1 and rank the other vertices of X in a clockwise order. Let $y_{|Y|} = \ell(x_1)$, and rank the vertices of Y in a counterclockwise order. Let $P = L(x_1)$. Since P is the leftmost shortest path from x_1 to Y, we get the following lemma:

Lemma 1. *Let $x_i \in X$ and $y_j \in Y$. There is a shortest path Q in J from x_i to y_j such that either Q does not cross P, or every prefix of Q crosses P from the left side of P to its right side at most once more than it crosses P from its right side to its left side.*

Proof. Assume that every shortest path from x_i to y_j in J crosses P, and let Q be such a path. The first time that Q emanates from P it emanates from its right side, since otherwise there is a shortest path from x_1 to y_j ending with a suffix of Q to the left of P. We may assume that if Q meets P at a vertex w, and then again at a vertex w' such that w' follows w also in P, then the subpath of Q between w and w' is the same subpath as in P. From this assumption we get that in two consecutive times that Q crosses P, it does so from different directions, since otherwise Q must cross itself. From this the lemma follows.   □

Let M' be the partial matrix of M where M'_{ij} is non-blank if there is a shortest path in J from x_i to y_j that does not cross P, and let M'' be the partial matrix of M where M''_{ij} is non-blank if every shortest path from x_i to y_j crosses P.

The partial matrix M' has the Monge property, we get this by cutting open J along P and using the claim from Sect. 2.3 (see x_i, x_k in Fig. 3). Using Lemma 1, a similar argument (by taking two copies of J open at P and "gluing" the right side of one of them to the left side of the other) shows that M'' also has the Monge property (see x'_i, x'_k in Fig. 3).

The entire first row of M is in M', the non-blank entries of row $i > 1$ in M' are from $\ell(x_i)$ to $y_{|Y|} = \ell(x_1)$, and the rest of the row is in M''. The partial matrix M' is a falling staircase matrix and M'' is an inverse falling staircase matrix, since $L(x_{i+1})$, the leftmost path from x_{i+1} to Y, cannot cross the path $L(x_i)$ from its right side to its left. Let N' and N'' be the corresponding staircase matrices partial to N (with same blank entries as M' and M'' respectively), both of them have the Monge property.
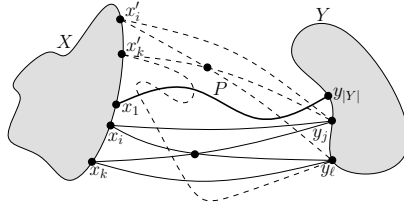
Fig. 3. If a shortest path from X to Y does not cross the path P (*solid lines*), or always cross P (*dashed* lines), then M has the Monge property (cf. Fig. 1)

In order use the insights above, we take parts (i) and (ii) of the data structure of the previous section together with the following two new parts, where X, Y and J are as defined above (part (iii) is not necessary):

(iv) For each two pieces B and B' we store $d_J(x, y)$ for every $x \in X$, $y \in Y$.
(v) For each two pieces B and B', and every $x \in X$, we store $\ell(x)$.

Notice that $d_J(x, y)$ and $\ell(x)$ depend on the specific pieces B and B'. There are $O(n/r)$ pieces, each piece has $O(\sqrt{r})$ boundary vertices, so the total space required for the two new parts is $O(n^2/r)$. This does not increase the total space complexity of the data structure which remains $O(n^2/r + n\sqrt{r})$.

5.1   The Preprocessing Algorithm

We have an r-decomposition of G obtained by a recursive decomposition, where we decomposed each piece into two pieces and stopped the recursion at pieces of size $O(r)$, from which we took only the pieces that correspond to leaves of the recursion tree. Now we will take all the pieces of the entire recursive decomposition which defined the r-decomposition. We build a dense distance graph for G based on this recursive decomposition using the algorithm of Fakcharoenphol and Rao [8] with the improvement of Klein [13] in $O(n \log^2 n)$ time and $O(n \log n)$ space.

For two fixed pieces B and B' we use a subgraph of the dense distance graph to compute the distances from vertices of X to vertices of Y in J. We should choose carefully a set S of pieces of the recursive decomposition to use, we must obey three rules – we cannot take any piece of the $O(\log n)$ pieces that contain either B or B', we should cover all the paths from X to Y, and the total number of boundary vertices of pieces in S should not be too large.

We start with the entire graph G, the root of the recursive decomposition, as the single piece in S. As long as there is a piece C in S containing either B or B' in it (an ancestor of B or B' in the recursion tree), we replace C with both of its children in the recursion tree. When we get to B or B', we remove them from S. At the end of this process, S contains $O(\log n)$ pieces with $O(\sqrt{n})$ boundary vertices. Every vertex of X or Y is a boundary vertex of some piece in S (otherwise such a vertex will be an internal vertex of some member of S which is an ancestor of B or B'), and the paths in J among these vertices are covered

by S (since S only misses internal vertices of B and B'), as required. Denote the subgraph of the dense distance graph that includes exactly the pieces of S by D.

For $x \in X$, we compute part (iv), by computing $d_J(x, y)$ for every $y \in Y$ using the Dijkstra implementation of [8] on D in $O(\sqrt{n} \log^2 n)$ time.

We use the distances of vertices of D from x to find $\ell(x)$ for part (v) as well. We add to D a vertex u' inside B and connect it to x, and a vertex v' inside B' and connect every vertex of Y to it, as described before. Since the edge from u' to x has length 0, for every vertex z in D, $d_J(u', z) = d_J(x, z)$, so we can determine for each edge of D whether it is tight in constant time. Even though D is not planar, we can define a cyclic order on the edges incident to each vertex using the embedding of G. The order of edges incident to a specific vertex of D is defined by the order of the shortest paths in G that they represent. We can compute this order in a total time of $O(n \log^3 n)$ when we construct the dense distance graph of G, we skip the details of this procedure. Then, we ignore all non-tight edges in D and find $\ell(x)$ by finding the leftmost path from u' to v'. This takes time proportional to the number of vertices in D, which is $O(\sqrt{n})$.

We perform the computation of parts (iv) and (v) for every B and B', and $x \in X$. For every x this computation takes $O(\sqrt{n} \log^2 n)$ time, and we repeat it $O(n^2 / r^{3/2})$ times. The total time required for constructing the data structure is $O((n^{5/2}/r^{3/2}) \log^2 n + n\sqrt{r})$ (note that the term $n \log^3 n$ is dominated by this bound). We note that it is also possible to perform the computation in $O((n^3/r^2) \log n)$ time using the algorithm of Klein [13] for every pair of pieces, however this does not improve the time bound for our range of r.

5.2   The Query Algorithm

Let u, v be a query pair. We use the data structure of this section to find $d_G(u, v)$ in $O(\sqrt{r})$ time. Again, let B, B' be the pieces that contain u, v respectively.

If $B = B'$ then we can answer the query in $O(\sqrt{r})$ time in same way as in the previous data structure, which uses the query algorithm of [4]. Either the shortest path from u to v is inside B or the shortest path contains some vertex $b \in \partial B$. In other words, $d_G(u, v) = \min\{d_B(u, v), \min_{b \in \partial B}\{d_G(u, b) + d_G(b, v)\}\}$. We retrieve the distance $d_B(u, v)$ from part (ii) of the data structure in $O(\sqrt{r})$ time. For each $b \in \partial B$ we retrieve $d_G(u, b) + d_G(b, v)$ from part (i) in $O(1)$ time. Since $|\partial B| = O(\sqrt{r})$, it takes $O(\sqrt{r})$ time to go over all vertices of ∂B and find $b \in \partial B$ that minimizes $d_G(u, b) + d_G(b, v)$. Then, $d_G(u, v)$ is the minimum between $d_B(u, v)$ and $d_G(u, b) + d_G(b, v)$.

For the case where u and v are in different pieces, let X and Y be as before. Fix x_1 to be an arbitrary vertex of X and let $y_{|Y|} = \ell(x_1)$. Let the matrices N, N', N'' be as before. We compute an entry $N_{ij} = d_G(u, x_i) + d_J(x_i, y_j) + d_G(y_j, v)$ in $O(1)$ time using parts (i) and (iv) of the data structure. We determine whether an entry of N is in N' or in N'' in $O(1)$ time using part (v). Therefore, we can use the SMAWK algorithm [1] as in [2] to find the minimum value in N in $O(\sqrt{r})$ time. This value is the requested distance.

We conclude that for a planar graph with n vertices and any $r \in (0, n)$, we can construct in $O((n^{5/2}/r^{3/2}) \log^2 n + n\sqrt{r})$ time a data structure of size

$O(n^2/r + n\sqrt{r})$ that computes the distance between any two vertices in $O(\sqrt{r})$ time. As in Sect. 4, by setting $r = n^2/S$ we get:

Theorem 3. *For a planar graph with n vertices and $S \in [n^{4/3}, n^2]$, we can construct in $O((S^{3/2}/\sqrt{n}) \log^2 n)$ time a data structure of size $O(S)$ that computes the distance between any two vertices in $O(n/\sqrt{S})$ time.*

References

1. Aggarwal, A., Klawe, M.M., Moran, S., Shor, P., Wilber, R.: Geometric applications of a matrix-searching algorithms. Algorithmica 2, 195–208 (1987)
2. Aggarwal, A., Klawe, M.: Applications of generalized matrix searching to geometric algorithms. Discrete Appl. Math. 27, 3–23 (1990)
3. Arikati, S.R., Chen, D.Z., Chew, L.P., Das, G., Smid, M.H., Zaroliagis, C.D.: Planar spanners and approximate shortest path queries among obstacles in the plane. In: Díaz, J., Serna, M.J. (eds.) ESA 1996. LNCS, vol. 1136, pp. 514–528. Springer, Heidelberg (1996)
4. Cabello, S.: Many distances in planar graphs. Algorithmica (to appear)
5. Chen, D.Z., Xu, J.: Shortest path queries in planar graphs. In: STOC 2000, pp. 469–478. ACM, New York (2000)
6. Djidjev, H.N.: Efficient algorithms for shortest path queries in planar digraphs. In: d'Amore, F., Franciosa, P.G., Marchetti-Spaccamela, A. (eds.) WG 1996. LNCS, vol. 1197, pp. 151–165. Springer, Heidelberg (1997)
7. Djidjev, H.N., Venkatesan, S.M.: Planarization of graphs embedded on surfaces. In: Nagl, M. (ed.) WG 1995. LNCS, vol. 1017, pp. 62–72. Springer, Heidelberg (1995)
8. Fakcharoenphol, J., Rao, S.: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72, 868–889 (2006)
9. Feuerstein, E., Marchetti-Spaccamela, A.: Dynamic algorithms for shortest paths in planar graphs. Theor. Comput. Sci. 116, 359–371 (1993)
10. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput. 16, 1004–1022 (1987)
11. Henzinger, M.R., Klein, P., Rao, S., Subramanian, S.: Faster shortest-path algorithms for planar graphs. J. Comput. Syst. Sci. 55, 3–23 (1997)
12. Hutchinson, J.P., Miller, G.L.: Deleting vertices to make graphs of positive genus planar. In: Discrete Algorithms and Complexity Theory, pp. 81–98. Academic Press, Boston (1986)
13. Klein, P.N.: Multiple-source shortest paths in planar graphs. In: SODA 2005, pp. 145–155. SIAM, Philadelphia (2005)
14. Klein, P.N., Mozes, S., Weimann, O.: Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n \log^2 n)$-time algorithm. ACM Trans. Algorithms 6, 1–18 (2010)
15. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM J. on Appl. Math. 36, 177–189 (1979)
16. Miller, G.L.: Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci. 32, 265–279 (1986)
17. Mozes, S., Sommer, C.: Exact Distance Oracles for Planar Graphs, arXiv:1011.5549v2 (2010)
18. Mozes, S., Wulff-Nilsen, C.: Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In: de Berg, M., Meyer, U. (eds.) ESA 2010. LNCS, vol. 6347, pp. 206–217. Springer, Heidelberg (2010)

Piercing Quasi-Rectangles: On a Problem of Danzer and Rogers

János Pach[1,*] and Gábor Tardos[2,**]

[1] EPFL, Lausanne and Rényi Institute, Budapest
pach@cims.nyu.edu
[2] Department of Computer Science, Simon Fraser University,
Burnaby and Rényi Institute, Budapest
tardos@cs.sfu.edu

Abstract. It is an old problem of Danzer and Rogers to decide whether it is possible arrange $O(\frac{1}{\varepsilon})$ points in the unit square so that every rectangle of area ε contains at least one of them. We show that the answer to this question is in the negative if we slightly relax the notion of rectangles, as follows. Let δ be a fixed small positive number. A *quasi-rectangle* is a region swept out by a continuously moving segment s, with no rotation, so that throughout the motion the angle between the trajectory of the center of s and its normal vector remains at most δ. We show that the smallest number of points needed to pierce all quasi-rectangles of area ε is $\Theta\left(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}\right)$.

Faster Algorithms for Minimum-Link Paths with Restricted Orientations

Valentin Polishchuk and Mikko Sysikaski

Helsinki Institute for Information Technology,
Department of Computer Science, University of Helsinki
`firstname.lastname@cs.helsinki.fi`

Abstract. We give an $O(n^2 \log^2 n)$-time algorithm for computing a minimum-link rectilinear path in an n-vertex rectilinear domain in three dimensions; the fastest previously known algorithm of [Wagner, Drysdale and Stein, 2009] has running time $O(n^{2.5} \log n)$. We also present an algorithm to find a minimum-link C-oriented path in a C-oriented domain in the plane; our algorithm is simpler and more time-space efficient than the earlier one of [Adegeest, Overmars and Snoeyink, 1994].

1 Introduction

Computing minimum-link rectilinear paths amidst rectilinear obstacles in the plane is one of the oldest and best studied problems in computational geometry [2, 3, 12, 13, 14, 15, 16, 19, 22, 25, 26, 27]. For a domain with n vertices, an optimal, $O(n \log n)$-time linear-space algorithm was presented at WADS 1991 by Das and Narasimhan [2]. A similar algorithm was claimed in the lesser known work of Sato, Sakanaka and Ohtsuki [22].

1.1 3D Rectilinear Paths

In three dimensions, the problem is somewhat farther from being solved. De Berg et al. [4] give an $O(n^d \log n)$-time algorithm to find shortest path in combined metric (a linear combination of length and number of links) amidst axis-parallel boxes in $\mathbb{R}^d$. An $O(n^3)$-time algorithm for minimum-link rectilinear path in a 3D rectilinear domain is implied by results of Mikami and Tabuchi [17]. Fitch, Butler and Rus [6] presented an algorithm for the 3D case with good practical performance; in many cases, their algorithm runs in $O(n^2 \log n)$ time. Still, the worst-case running time of the algorithm in [6] is cubic.

The fastest current solution for the minimum-link rectilinear path problem in $\mathbb{R}^3$ is due to Wagner, Drysdale and Stein [23]. Their algorithm, based on searching binary space partition (BSP) of the domain, runs in $O(n^{2.5} \log n)$ time and $O(n^{2.5})$ space.

1.2 C-oriented Paths in the Plane

Another generalization of the 2D rectilinear setting is the C-oriented version [1, 8, 9, 10, 18, 21, 24] in which orientations of path edges and the domain

sides come from a fixed set C of directions. (Abusing the notation, we use C to denote also the cardinality of the set C.) Minimum-link C-oriented paths in the plane were studied by Adegeest, Overmars and Snoyeink [1]. Two algorithms are presented in [1]: one running in $O(C^2 n \log n)$ time and space, the other—in $O(C^2 n \log^2 n)$ time and $O(C^2 n)$ space.

1.3   Contributions

We revisit the two generalizations of the 2D minimum-link rectilinear path problem, giving more efficient algorithms for both versions. Specifically,

- In Section 2 we give an $O(n^2 \log^2 n)$-time $O(n^2)$-space algorithm for the 3D minimum-link rectilinear path problem.
- In Section 3 we present an $O(C^2 n \log n)$-time $O(Cn)$-space algorithm to find a minimum-link C-oriented path in a C-oriented domain in the plane.

Similarly to the earlier works, our algorithms actually build shortest path *maps* – data structures to answer efficiently link distance queries to a fixed source (in the case of rectilinear paths in 3D, constructing the map with our algorithm requires $O(n^2 \log^2 n)$ space – slightly more than just finding one shortest path).

2   Rectilinear Paths Amidst Rectilinear Obstacles in $\mathbb{R}^3$

Let P be an n-vertex rectilinear domain in 3D, and let $s, t \in P$ be given points; the goal is to find a minimum-link rectilinear s-t path in P. We follow the "staged illumination" paradigm [7, Sections 26.4, 27.3], dominant in the minimum-link literature: on first step illuminate (and label with link distance 1) the set reachable with a single link from s, on second step illuminate and label with 2 what is reachable with two links, and so on, until t is lit.

At any step k, the illumination is done by sweeping a plane in each of the six directions $\pm x, \pm y, \pm z$. We will describe the sweep of a horizontal plane in $+z$ (vertical) direction; the other sweeps are analogous. The goal of the sweep is two-fold: (1) to discover (and label with k) the volume illuminated when shining light upwards from the volume lit at step $k - 1$, and (2) to generate events for (all six) sweeps on step $k + 1$.

As in [23], we assume that obstacle faces are decomposed into rectangles; we call them *obstacle rectangles*. The sweep is guided by a decomposition of the free space into axis-aligned boxes (cells), with each cell storing links to the neighbor boxes and the obstacle rectangles touching the box.

The status of the sweep plane is the subset of the plane that gets illuminated if light is shone upwards from the points that were illuminated at step $k - 1$; the status is maintained as a set of rectangles in a 2D segment tree. With each rectangle R we store a variable z_R which is the smallest z-coordinate of the origin of an upward-shining light ray that hits R; in other words, if one moves down from a point $r \in R$, then z_R is the height below which there is no point that was illuminated at step $k - 1$ and can see r.

The event queue contains events of three types:

- **CellEvent** occurs when the sweep plane reaches the upper boundary of a cell. Processing the event involves the following operations: (1) each upper neighbor of the cell is inserted, as a **CellEvent**, into the event queue; (2) each obstacle rectangle (if any) touching the cell from above is inserted, as an **ObstacleEvent**, into the event queue; (3) the cell is inserted as a **CellEvent** into the event queues for the sweeps in the other directions at the next step, $k+1$; (4) if the cell is not labeled (i.e., has not been illuminated before), the cell is labeled with k.
- **ObstacleEvent** occurs when the sweep plane reaches an obstacle rectangle R_o; let z_o be the time (height) of the event. For each rectangle R from the status, intersected by R_o, let $[x_R^{\min}, x_R^{\max}] \times [y_R^{\min}, y_R^{\max}]$ be the intersection $R \cap R_o$. We generate an **AddRectangleEvent** into the event queue for the $+x$ sweep at step $k+1$: the generated rectangle is $[y_R^{\min}, y_R^{\max}] \times [z_R, z_o]$ and the time of the event is $x_R^{\max}$. Analogous rectangles are inserted to other sweeps. R_o is removed from the sweep plane status.
- **AddRectangleEvent**, on which a rectangle is added to the segment tree.

This sweep-label-generate strategy is a common one. Clearly, the running time of the algorithm employing the strategy, depends heavily on the number of cells in the decomposition (all cells must be labeled, after all): using the $O(n^{1.5})$-size BSP [5, 11, 20] instead of the $O(n^3)$ "full grid" allowed [23] to bring the time complexity from cubic down to $O(n^{2.5} \log n)$. Following this direction, one might improve the running time by using a yet smaller-size decomposition; we, however were not able to do this. Instead, we resort to a seemingly worse, *quadratic*-size decomposition. We compensate the increase in the number of cells by a better (albeit still quadratic) bound on the number of neighbor relations between the cells; the number enters the algorithm's running time via the need to perform Operation (1) in **CellEvent**s. While [23] spend $O(n^{2.5})$ time determining the neighbor relations, we get them for free. *Any* quadratic-size decomposition with quadratic number of cell-to-cell pointers works for us; one simple decomposition is obtained by sweeping a horizontal plane in $+z$ direction stopping at every horizontal obstacle rectangle, decomposing the cross-section of P with the stopped plane into rectangles by extending maximal free horizontal segments through vertices of the cross-section, and pulling the rectangles up in $+z$ direction until the height of the next stop of the sweep plane.

The above description is very generic, and many technical details have to be filled in, both by [23] and by us (below). We remark that also in these details, our algorithm is different from [23] (our analysis is different from [23] as well). This is not surprising: shooting for a smaller running time while working with a larger-complexity decomposition is challenging, and prevents directly reusing ideas from [23].[1]

---

[1] For completeness, let us mention that we slightly differ from [23] already in the generic description of the events: [23] inserts **AddRectangleEvent**s into the other sweeps when processing **CellEvent**s; because we have too many cells, we cannot do it, and instead insert **AddRectangleEvent**s when processing **ObstacleEvent**s.

In particular, a standard way to avoid reilluminating the same cell at too many steps is to declare labeled cells as obstacles. We cannot afford this because updating the sweep plane status at one ObstacleEvent may take as much as $O(n \log n)$ time — and we might declare all our $O(n^2)$ cells as obstacles. (Note that handling CLEARRECT events—part of the sweep plane status updates—is the bottleneck in the $O(n^{2.5} \log n)$ algorithm of [23].) So we approach limiting the number of status updates from the other end: instead of trying to decrease directly the number of rectangles *deleted* from the status, we restrict the number of rectangles that are ever *added*. See Section 2.1 for the details.

To bound the number of AddRectangleEvents processed by the algorithm, we would like to have "nice" rectangles; unfortunately, the rectangles generated as described above are somewhat arbitrary. One nice set of rectangles is as follows: take the cross-section of P by the horizontal plane supporting an obstacle rectangle, and decompose the cross-section (which is a 2D rectilinear domain) into $O(n)$ rectangles by extending maximal free-space segments parallel to x trough vertices of the domain (this is the standard trapezoidal decomposition of a 2D rectilinear domain; we use it e.g., in Section 3). To enforce that the AddRectangleEvents deal only with such nice rectangles, at every ObstacleEvent we filter out the added rectangles locally, keeping for each sweep direction only those rectangles that are not "dominated" by larger rectangles going in the same direction and are not "annihilated" by rectangles going in the opposite direction. Section 2.2 presents the details.

2.1 Avoiding Reillumination

We only describe how we add xz- and yz-rectangles, generated during the $+z$ sweep; the other sweeps are identical. Call such rectangles *displays*.

The purpose of displays is to emit light into *dark* volumes; thus it makes sense to keep only those displays that have a potential to shine into previously unilluminated space. Any cell σ with label $k-3$ or less gets *fully* illuminated by step $k-1$. Hence after step k, everything visible from σ will be illuminated. Therefore any display R' that touches only cells with labels $k-3$ or less can be safely discarded (i.e., not added as an AddRectangleEvent): nothing new gets illuminated by shining light from R'.

To determine whether a given display R' intersects any "new" cell (cell with label $k-2, k-1$ or k) we maintain an auxiliary 2D segment tree storing projection of the new cells onto the xy plane; the tree is updated on every CellEvent and ObstacleEvent, and also cleared of old cells when k increases. (Note that directly testing R' for intersection with every new cell would be too inefficient because R' may in principle intersect $\Omega(n)$ cells). The query for R' reduces then to the simple test of whether the projection of R' on the horizontal plane intersects rectangles stored in the auxiliary tree.

2.2 Filtering

Again, we only describe what we do during the $+z$ sweep when generating displays (which are the potential AddRectangleEvents for the sweeps in $\pm x$ and $\pm y$

directions). Recall that on an ObstacleEvent we remove an obstacle rectangle R_o from the sweep plane status 2D segment tree, i.e., we clear all subtrees rooted at canonical nodes of R_o; to clear a subtree, we visit all canonical nodes of the rectangles in the subtree, and at every canonical node R generate 4 displays as specified in the description of the ObstacleEvent.

For our analysis of the number of AddRectangleEvents (Lemma 3) it will be important that each rectangle in the decomposition of a cross-section of P by maximal free-space horizontal segments is charged to exactly one display. To enforce this, we do the following operations after generating the displays, before moving to the next event in the $+z$ sweep:

– Find all pairs of displays with the same positions but opposite directions (Fig. 1(a), top). If the z-ranges of the displays are the same, remove both of them. Otherwise remove the one with smaller z-range, i.e., the one with higher z_R.
– Merge aligned displays with the same z-range into bigger displays (Fig. 1(a), bottom).

Figure 1 illustrates the events that are generated initially and the remaining events after the filtering.

(a) Elementary filtering operations

(b) Original rectangles of the segment tree (red) and the obstacle face to be removed (grey)

(c) AddRectangleEvents generated for each canonical node to be cleared

(d) AddRectangleEvents remaining after filtering
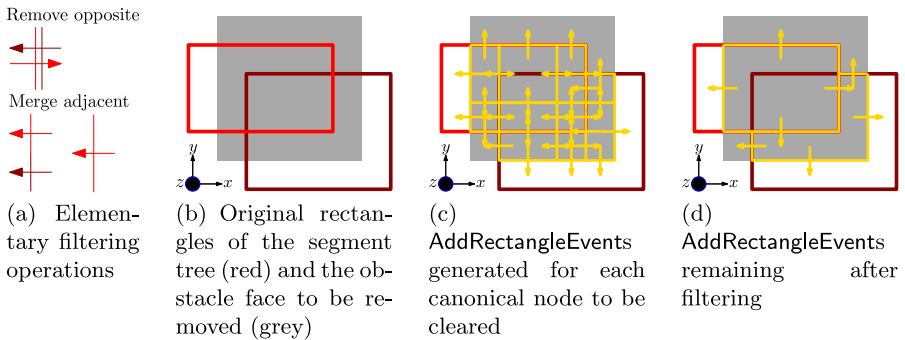
Fig. 1. Filtering the displays during an ObstacleEvent. We generate displays in all 4 directions for every canonical node cleared from the segment tree (Fig. 1(c)). After the filtering, the displays are maximal in x- and y-directions (Fig. 1(d)).

2.3 Analysis

Let σ be a cell of the decomposition, and let R_o be an obstacle rectangle.

Lemma 1. CellEvents *containing σ are generated on at most 9 steps of the illumination.*

Proof. Assume that σ is first discovered on step k. Then no later than on step $k+2$ any point inside σ is illuminated, and on step $k+3$ we discover the set S of all points visible from σ. After step $k+6$ we have discovered every point that might share a common AddRectangleEvent with a point in S so after step $k+8$ we can no longer generate AddRectangleEvents that would reach σ.   □

Lemma 2. *At most* 9 ObstacleEvent*s containing* R_o *are generated over the course of the illumination.*

Proof. Similarly to Lemma 1, after discovering R_o on step k we discover every point that can see R_o no later than on step $k + 3$, so again no sweep can reach R_o after step $k + 8$.   □

Lemma 3. *The number of* AddRectangleEvent*s is* $O(n^2)$.

Proof. The boundaries of illuminated area are always aligned with obstacle faces. Hence any AddRectangleEvent belongs to the supporting plane of some obstacle face. We will consider only the AddRectangleEvents processed in the $+z$ sweep; the other directions are identical.

Let R_o be some horizontal obstacle rectangle, and let H_o be the supporting plane of R_o. We will bound the number of AddRectangleEvents happening in H_o. We assume that the plane H_o is unique to R_o. That is, even if another obstacle rectangle R'_o has the supporting plane $H'_o = H_o$ (because R_o and R'_o belong to the same obstacle face), we will treat the AddRectangleEvents for H_o and H'_o separately. This way we may only overcount the number of AddRectangleEvents.

Let P_o be the cross-section of P by H_o. Let D_x be the trapezoidal (in fact, rectangular) decomposition of P_o obtained by extending maximal free-space segments parallel to x through vertices of P_o; define D_y similarly. Consider finding a minimum-link path in P_o from some arbitrary set of maximal (either in x or in y or both) rectangles lying in P_o. The staged illumination from the rectangles proceeds in P_o by lighting up rectangles from $D_x \cup D_y$ (this is the classical process central to the study of 2D minimum-link paths).

Now look at our staged illumination of the 3D domain P, and consider the "cross-section of the illumination" by H_o; i.e., look at when and how the light from the 3D illumination intersects P_o. The crucial observation is the following: after the 3D illumination has reached H_o, the cross-section of the illumination is exactly the 2D staged illumination in P_o. In particular, at most 2 AddRectangleEvents will happen in each rectangle from $D_x \cup D_y$.

Indeed, the 3D illumination arriving at H_o can be viewed as "piercing" P_o with rectangular "pillars of light"; the cross-section of the pillars by H_o is a set of rectangles – not necessarily maximal (yet). However, already on the next step, when the light is shone in x- and y-directions from the rectangles, the lit area of P_o *is* a set of maximal rectangles, and the illumination from the set proceeds along $D_x \cup D_y$.

Thus, the number of AddRectangleEvents happening at H_o is the sum of 2 parts: (1) the total number of rectangles added immediately after the piercing, and (2) the complexity of D_x, D_y. The events in (1) are defined by those rectangles from the xz- and yz- sweep planes segment trees, that are intersected by H_o: the number of the events is the number of the rectangles in the trees that are intersected by H_o. But the number of rectangles in a segment tree intersected by any line is $O(n)$; thus the number of type-(1) events is $O(n)$. The complexity of D_x, D_y is also linear; hence the total number of AddRectangleEvents on H_o is linear.

To finish the proof, recall that H_o is the supporting plane of an obstacle rectangle; the number of such planes is $O(n)$.   □

Theorem 1. *Minimum-link path can be computed in time* $O(n^2 \log^2 n)$ *using* $O(n^2)$ *space.*

Proof. The time in the algorithm is spent in creating the decomposition, maintaining event queues and handling events, of which the event handling is the dominating component. Each CellEvent and AddRectangleEvent is handled in $O(\log^2 n)$ time using standard segment tree operations, and the number of such events is $O(n^2)$ by Lemmas 1 and 3. Clearing a rectangle in an ObstacleEvent can be performed in $O(n \log n + \rho)$ time, where ρ is the number of the tree nodes to remove (and to generate AddRectangleEvents for); the $O(\rho)$-part can be charged to the earlier events that created the nodes. Together with Lemma 2 this gives an $O(n^2 \log n)$ time bound for handling the ObstacleEvents. Thus in total handling the events takes $O(n^2 \log^2 n)$ time.

The size of the decomposition as well as the number of events and the size of a segment tree are all $O(n^2)$ so the space used is $O(n^2)$.   □

To build the shortest path map we use a persistent version of the segment tree and store with each cell the snapshots of the sweepline status from the times when a sweep reached the cell.

3   *C*-oriented Paths in the Plane

In this section P is a C-oriented n-vertex polygonal domain in the plane, and we want to build the C-oriented link distance map from a given point $s \in P$.

3.1   Overview

As in [1], we build C trapezoidations of P; the trapezoids in the trapezoidation $c \in C$ are obtained by extending maximal free c-oriented segments through vertices of P. We label the trapezoids with link distance from s; the labeling proceeds in *n steps*, with label-k trapezoids receiving their label at step k. Any label-k trapezoid must be intersected by a label-$(k-1)$ trapezoid of a different orientation; hence, step k boils down to detecting all unlabeled trapezoids intersected by label-$(k-1)$ trapezoids.

Some trapezoids may get labeled only partially during a step k; in the final link distance map, such trapezoids are split into subtrapezoids. The partial labeling and splitting are due to the possibility that two different-orientation trapezoids do not "straddle" each other; instead they both may be "flush" with an obstacle edge whose orientation is different from the orientations of both trapezoids. Such flushness can be read off easily from lists of incident trapezoids stored with every edge of P.

After the partially labeled trapezoids are processed, we are left with discovering unlabeled trapezoids "fully straddled" by trapezoids labeled at the previous

step. We clip the latter trapezoids into parallelograms, with the new sides parallel to the sweepline, and finish the step with $C(C-1)$ sweeps—one per pair of orientations.

The main difference of our algorithm from that of [1] is the separate treatment of flush and straddling trapezoids. It allows us to use only elementary data structures and improve the time-space bounds to $O(C^2 n \log n)$ and $O(Cn)$.

3.2   Definitions and Notation

Any trapezoid T will have two opposite edges belonging to the boundary of P; these edges are *sides* of T. The other two edges are T's *bases*; the bases are parallel segments whose orientation belongs to C. For an orientation $c \in C$, a c-segment is a segment with orientation c. A *c-trapezoid* has c-segments as bases.

A *c-path* is a path (starting from s) whose last link is a c-segment. A point $p \in P$ is at *c-distance* k from s if p can be reached by a k-link c-path (but not faster). The c-distance equivalence decomposition of P (the *c-map*) is the partition of P into c-trapezoids such that the c-distance to any point within a cell is the same. If the c-distance to points in a c-trapezoid of the c-map is k, then the c-trapezoid has *label* k. Using the illumination analogy we also say that the trapezoid is *lit* at step k; unlit trapezoids are *dark*. We denote the set of c-trapezoids lit at step k by S_c^k.

In our algorithm, finding S_c^k is completely identical to (and independent from) finding $S_{c^*}^k$ for any $c^* \in C \setminus c$; in what follows we focus on finding S_c^k. Where it creates no confusion, we omit the subscript c and the prefix c-. E.g., "path to trapezoid in S^{k-1}" means "c-path to c-trapezoid in S_c^{k-1}", etc. We let C^* denote $C \setminus c$, and use c^* for a generic orientation from C^*. Assume w.l.o.g. that c is horizontal.

Denote by D^k the "at-most-k-links map", i.e., the trapezoidation whose trapezoids are of $k+1$ types—dark trapezoids, and trapezoids lit at steps $1 \ldots k$; in particular, D^n is the c-map. Let T' be a dark trapezoid from D^{k-1}. The crucial (albeit obvious) observation about minlink paths is that there exists a k-link path to a point $p \in T'$ if and only if there exists a $(k-1)$-link c^*-path π^* to some point $q \in T'$ that has the same y-coordinate as p. Restated in our terms, this means that a dark trapezoid $T' \in D^{k-1}$ gets fully or partially lit at step k if and only if it is intersected by some trapezoid $T^* \in S_{c^*}^{k-1}$ lit at step $k-1$. We distinguish between two types of trapezoids intersection (Fig. 2):

Definition 1. T', T^* are flush *if a side of T' overlaps with a side of T^*; we say that T' is (fully or partially) flush-lit by T^*. T', T^* straddle each other if both bases of T' intersect both bases of T^* (in particular, if a side of T^* overlaps with a base of T' or vice versa, then T', T^* are counted as straddling, not as flush); we say that T' is (fully) straddle-lit by T^*.*

Note that flushness and straddling are the only possible ways for two trapezoids from $D_c^{k-1}, D_{c^*}^{k-1}$ to intersect.

With Definition 1, step k of the algorithm can be completed as follows: Find dark c-trapezoids flush with trapezoids from $S_{c^*}^{k-1}$, and (fully or partially) light
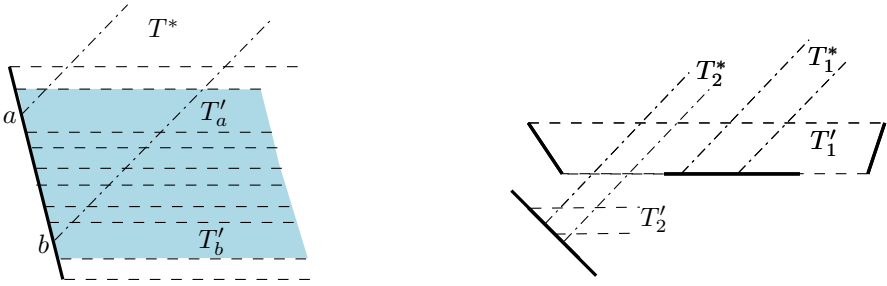
Fig. 2. Intersection types. Left: T^* is flush with the light blue trapezoids. T'_b will be split in D^k unless more of it is lit by another trapezoid. T_a will be the pot for the parallelogram cut out of T^* by the c-segment through a. Right: T_1^*, T_2^* straddle T'_1. The parallelogram cut out of T_1^* is planted into T'_1; the pot for the parallelogram cut out of T_2^* is a trapezoid T'_2 flush with T_2^*.

them. After this has been done for all $c^* \in C^*$, i.e., after all flush trapezoids are processed, any dark trapezoid will either fully remain dark in D^k or will be fully straddle-lit (i.e., there will be no more partial lighting and splitting). To straddle-light c-trapezoids, we clip each c^*-trapezoid $T^* \in S_{c^*}^{k-1}$ to a (c, c^*)-parallelogram P^*, using c-segments going through vertices of T^*. We then do a sweep in the direction perpendicular to c. The clipping and sweeping is repeated for each $c^* \in C^*$, i.e., overall, to straddle-light the c-trapezoids in S_c^k, we perform $C - 1$ sweeps—one per $c^* \in C^*$.

We proceed to a detailed description of the flush- and straddle-lighting.

3.3 Flush-Lighting

Sides of trapezoids belong to edges of P; we say that an edge e *supports* a trapezoid if one of its sides belongs to e. We maintain the ordered list $L_c(e)$ of c-trapezoids supported by e. The flush-lighting is done as follows: For every c^*-trapezoid $T^* \in S_{c^*}^{k-1}$ and each edge e that supports T^*, locate the vertices a, b of T^* (lying on e) in the list $L_c(e)$. All (dark) trapezoids lying between a and b are labeled k. One of the trapezoids T'_a, T'_b containing a, b in the interior of the side is marked to be split, at the end of flush-lighting, by horizontal cut through a or b—unless more of the trapezoid is flush-lit by another trapezoid. Refer to Fig. 2, left.

3.4 Straddle-Lighting

Clip each c^*-trapezoid $T^* \in S_{c^*}^{k-1}$ to the parallelogram P^* using horizontal lines through vertices of T^*; denote the set of the obtained parallelograms by $S^{\succ\circ}$. Any c-trapezoid straddled by T^* is also straddled by P^*, and thus straddle-lighting with c^*-trapezoids is equivalent to finding dark trapezoids intersected by parallelograms from $S^{\succ\circ}$. This can be accomplished with a sweep, called

(c, c^*)-*sweep*, which discovers the trapezoids from S_c^k in the order of increasing y-coordinate of the lower bases, by sweeping upwards a horizontal line. The sweepline status will be the intersection of the sweepline with the (interiors of) parallelograms from $S_{c^*}^{k-1}$; the status thus is a set of disjoint (open) intervals. The status changes at *parallelogram events* when the sweepline reaches horizontal sides of parallelograms. Because the intervals in the status are disjoint, we can keep them in any ordered structure, e.g., a balanced binary search tree indexed by left endpoints of the intervals. Clearly, the tree handles any of the following three operations in $O(\log n)$ time: (1) adding an interval, (2) removing part of an interval that hits an obstacle edge, and (3) checking whether any of the intervals overlaps with a given query interval. In the last operation, the query interval is a trapezoid lower base; we need it for the trapezoid events, described next.

The main events in the sweep are *trapezoid events* that occur when the sweepline reaches a lower base of a trapezoid (some of the trapezoid events happen simultaneously with parallelogram events; in this case parallelogram events take priority). Suppose that a trapezoid T is the event. We check whether the lower base of T is intersected by the intervals in the sweepline status. If yes, we insert T's upper neighbors into the event queue. In addition, if T is unlabeled, we label it with k.

To initialize the sweep, we "plant" each parallelogram into a "pot" trapezoid; the pots are initially inserted into the event queue. Say that a parallelogram $P^* \in S_{c^*}^{k-1}$ is *planted* into a trapezoid T if the lower side of P^* belongs to T; say also that T is the *pot* of P^* (Fig. 2). Each parallelogram is planted into exactly one pot (even though a pot may have many parallelograms planted side-by-side into it). Now, some c^*-trapezoids from $S_{c^*}^{k-1}$ (such as, e.g., trapezoid T_1^* from Fig. 2, right) have lower bases supported by c-edges of P—the pots for such parallelograms are read off directly from trapezoidation D_c and the lists $L_{c^*}(e)$. The rest of the trapezoids from $S_{c^*}^{k-1}$ (such as, e.g., trapezoid T_2^* from Fig. 2, right, or T^* from Fig. 2, left) are flush with trapezoids from D_c^{k-1}. The pot T' for the parallelogram P^* cut out from such a trapezoid T^* can be determined from the list $L_c(e)$, where e is the edge supporting T^* and T': all that is needed is to locate in which trapezoid from the list the vertex of T^* lands.

We emphasize that clipping by the c-segments is done only to find c-trapezoids straddle-lit by c^*-trapezoids; after the (c, c^*)-sweep completes, the c^*-trapezoids are "unclipped" back to what they were (and in general, during an (a, b)-sweep, b-trapezoids lit at the previous step are only temporarily clipped into (a, b)-parallelograms using a-segments through the vertices).

3.5   Analysis

Flush-lighting takes overall $O(C^2 n \log n)$ time: For every trapezoid T^* that flush-lights c-trapezoids through an edge e, it takes $O(\log n)$ time to locate the vertices a, b of T^* (lying on e) in the list $L_c(e)$. Overall there are $O(Cn)$ trapezoids T^*, and for each we have to locate the vertices a, b in the $C - 1$ lists $L_c(e)$; thus the locating takes overall $O(C^2 n \log n)$ time. After the vertices a, b have been located, it takes $O(n_e)$ time to label each (dark) trapezoid T' supported by e,

where n_e is the number of the trapezoids that are flush with T^*. Again, overall there are $O(Cn)$ trapezoids T', and each can be flush-lit from at most $C - 1$ directions; thus the total time spent in the labeling (not counting the time spent in locating the vertices a, b) is $O(C^2 n)$.

As for straddle-lighting, any trapezoid has $O(1)$ neighbors (assume no two edges of P are supported by the same line); thus, processing an event during any of the sweeps involves a constant number of the priority queue and/or interval tree operations, i.e., $O(\log n)$ time per event. To bound the number of events, observe that any trapezoid inserted in the event queue at step k is either itself intersected by a parallelogram form $S^{\succ^\circ_\circ}$, or has a lower neighbor intersected by a parallelogram from $S^{\succ^\circ_\circ}$; thus any trapezoid enters the event queue on at most 7 consecutive steps. At any step k, a c-trapezoid may appear in the event queue during each of the $C - 1$ (c, c^*)-sweeps. Thus, as there are $O(Cn)$ trapezoids, we have $O(C^2 n)$ events, and the total running time of $O(C^2 n \log n)$.

As for the space, the dominating factor is storing the C trapezoidations.

References

1. Adegeest, J., Overmars, M.H., Snoeyink, J.: Minimum-link c-oriented paths: Single-source queries. IJCGA 4(1), 39–51 (1994)
2. Das, G., Narasimhan, G.: Geometric searching and link distance. In: Dehne, F., Sack, J.-R., Santoro, N. (eds.) WADS 1991. LNCS, vol. 519. Springer, Heidelberg (1991)
3. de Berg, M.: On rectilinear link distance. CGTA 1, 13–34 (1991)
4. de Berg, M., van Kreveld, M.J., Nilsson, B.J., Overmars, M.H.: Shortest path queries in rectilinear worlds. IJCGA 2(3), 287–309 (1992)
5. Dumitrescu, A., Mitchell, J.S.B., Sharir, M.: Binary space partitions for axis-parallel segments, rectangles, and hyperrectangles. DCG 31(2), 207–227 (2004)
6. Fitch, R., Butler, Z., Rus, D.: 3d rectilinear motion planning with minimum bend paths. In: International Conference on Intelligent Robots and Systems (2001)
7. Goodman, J.E., O'Rourke, J.: Handbook of disc. and comp. geom. CRC Press series on discrete mathematics and its applications. Chapman & Hall/CRC (2004)
8. Güting, R.: Conquering Contours: Efficient Algorithms for Computational Geometry. PhD thesis, Fachbereich Informatik. Universität Dortmund (1983)
9. Güting, R.H., Ottmann, T.: New algorithms for special cases of the hidden line elimination problem. Comp. Vis., Graph., Image Proc. 40(2), 188–204 (1987)
10. Hershberger, J., Snoeyink, J.: Computing minimum length paths of a given homotopy class. CGTA 4, 63–97 (1994)
11. Hershberger, J., Suri, S.: BSP for 3d subdivisions. In: SODA 2003 (2003)
12. Imai, H., Asano, T.: Efficient algorithms for geometric graph search problems. SIAM J. Comput. 15(2), 478–494 (1986)
13. Imai, H., Asano, T.: Dynamic orthogonal segment intersection search. J. Algorithms 8(1), 1–18 (1987)

14. Lee, D.T., Yang, C.D., Wong, C.K.: Rectilinear paths among rectilinear obstacles. Discrete Appl. Math. 70, 185–215 (1996)
15. Lingas, A., Maheshwari, A., Sack, J.-R.: Optimal parallel algorithms for rectilinear link-distance problems. Algorithmica 14(3), 261–289 (1995)
16. Maheshwari, A., Sack, J.-R., Djidjev, D.: Link distance problems. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Comp. Geom., pp. 519–558. Elsevier, Amsterdam (2000)
17. Mikami, K., Tabuchi, K.: A computer program for optimal routing of printed circuit conductors. In: Int. Federation Inf. Proc. Congress, pp. 1475–1478 (1968)
18. Neyer, G.: Line simplification with restricted orientations. In: Dehne, F., Gupta, A., Sack, J.-R., Tamassia, R. (eds.) WADS 1999. LNCS, vol. 1663, pp. 13–24. Springer, Heidelberg (1999)
19. Ohtsuki, T.: Gridless routers - new wire routing algorithm based on computational geometry. In: Internat. Conf. on Circuits and Systems, China (1985)
20. Paterson, M., Yao, F.F.: Optimal binary space partitions for orthogonal objects. J. Algorithms 13(1), 99–113 (1992)
21. Rawlins, G.J.E., Wood, D.: Optimal computation of finitely oriented convex hulls. Inf. Comput. 72(2), 150–166 (1987)
22. Sato, M., Sakanaka, J., Ohtsuki, T.: A fast line-search method based on a tile plane. In: Proc. IEE ISCAS, pp. 588–591 (1987)
23. Wagner, D.P., Drysdale, R.L.S., Stein, C.: An $O(n^{2.5} \log n)$ algorithm for the rectilinear minimum link-distance problem in three dimensions. CGTA 42(5), 376–387 (2009)
24. Widmayer, P., Wu, Y.-F., Wong, C.K.: On some distance problems in fixed orientations. SIAM J. Comput. 16(4), 728–746 (1987)
25. Yang, C.D., Lee, D.T., Wong, C.K.: On bends and lengths of rectilinear paths: a graph theoretic approach. IJCGA 2(1), 61–74 (1992)
26. Yang, C.D., Lee, D.T., Wong, C.K.: On bends and distances of paths among obstacles in 2-layer interconnection model. IEEE Tran. Comp. 43(6), 711–724 (1994)
27. Yang, C.D., Lee, D.T., Wong, C.K.: Rectilinear paths problems among rectilinear obstacles revisited. SIAM J. Comput. 24, 457–472 (1995)

Streaming Algorithms for 2-Coloring Uniform Hypergraphs

Jaikumar Radhakrishnan and Saswata Shannigrahi

Tata Institute of Fundamental Research, Mumbai, India
{jaikumar,saswata}@tifr.res.in

Abstract. We consider the problem of two-coloring n-uniform hypergraphs. It is known that any such hypergraph with at most $\frac{1}{10}\sqrt{\frac{n}{\ln n}}2^n$ hyperedges can be two-colored [7]. In fact, there is an efficient (requiring polynomial time in the size of the input) randomized algorithm that produces such a coloring. As stated [7], this algorithm requires random access to the hyperedge set of the input hypergraph. In this paper, we show that a variant of this algorithm can be implemented in the streaming model (with just one pass over the input), using space $O(|V|B)$, where V is the vertex set of the hypergraph and each vertex is represented by B bits. (Note that the number of hyperedges in the hypergraph can be superpolynomial in $|V|$, and it is not feasible to store the entire hypergraph in memory.)

We also consider the question of the minimum number of hyperedges in non-two-colorable n-uniform hypergraphs. Erdős showed that there exist non-2-colorable n-uniform hypegraphs with $O(n^2 2^n)$ hyperedges and $\Theta(n^2)$ vertices. We show that the choice $\Theta(n^2)$ for the number of vertices in Erdös's construction is crucial: any hypergraph with at most $\frac{2n^2}{t}$ vertices and $2^n \exp(\frac{t}{8})$ hyperedges is 2-colorable. (We present a simple randomized streaming algorithm to construct the two-coloring.) Thus, for example, if the number of vertices is at most $n^{1.5}$, then any non-2-colorable hypergraph must have at least $2^n \exp(\frac{\sqrt{n}}{8}) \gg n^2 2^n$ hyperedges. We observe that the exponential dependence on t in our result is optimal up to constant factors.

Keywords: Property B, hypergraph coloring, streaming algorithm, randomized algorithm.

1 Introduction

Two colorability of uniform hypergraphs, also called Property B, is a well-studied problem in hypergraph theory. A hypergraph is called two colorable if all its vertices can be colored by either red or blue colors such that each hyperedge contains vertices of either colors. Erdős [4] first showed by a simple probabilistic argument that any n-uniform hypergraph with fewer than 2^{n-1} hyperedges is 2-colorable. Beck [3] used an algorithm of recoloring the vertices and improved this result to show that any n-uniform hypergraph with at most $n^{1/3-o(1)}2^{n-1}$ hyperedges is 2-colorable. Radhakrishnan and Srinivasan [7] improved on Beck's recoloring

algorithm and obtained the currently best known result on this problem. They proved that any n-uniform hypergraph with at most $\frac{1}{10}\sqrt{\frac{n}{\ln n}}2^n$ hyperedges can be 2-colored. They also provided a randomized polynomial time algorithm that 2-colors any such hypergraph with high probability. An event happens with high probability if for given any constant $\delta > 0$, the probability of its happening is at least $1 - \delta$. In Section 2, we describe this algorithm which we refer to as *delayed recoloring* algorithm throughout the rest of this paper.

The delayed recoloring algorithm assumes that all vertices and hyperedges of the hypergraph can be stored in the random-access memory (RAM). A processor can access RAM much faster than an external memory, and therefore it is ideal if we can store the entire hypergraph in RAM. Unfortunately, the number of hyperedges can be much larger than what the RAM can afford to store. For example, the number of hyperedges can be as high as $\Omega(2^n)$ when the size of the vertex set is just $O(n)$. This gives rise to the following question. If the RAM has just about enough space to store the vertices, can we still obtain the same coloring that the delayed recoloring algorithm obtains for any n-uniform hypergraph having at most $\frac{1}{10}\sqrt{\frac{n}{\ln n}}2^n$ hyperedges?

Before answering this question, let us first decide how we are going to store a hypergraph in the external memory. Note that each hyperedge is a collection of n vertices. If each vertex is represented by a B-bit long string, each hyperedge can be stored as a sequence of nB bits. We store the hyperedges one after another in the external memory, and store a terminal symbol when there are no more hyperedges. The vertices can be explicitly stored before the hyperedges, but we prefer that they are extracted while reading the hyperedges. This way, a coloring algorithm can start working as soon as the first hyperedge is stored in the external memory.

The algorithms that deal with limited RAM space are known as streaming algorithms in the literature. This algorithmic paradigm has of late become important, especially in the context of large data sets. It is motivated by the fact that the RAM of a computer is limited, and the size of the data set is often much larger than the size of the RAM. The data set can arrive as a continuous data stream or be stored in an external memory, in which case it is sequentially accessible over one or a small number of passes. However, only a minor fraction of the data along with some local variables can be stored in the RAM at any instant.

The challenge for any streaming algorithm is to minimize the following three parameters: the number of passes over the data, maximum RAM requirement at any instant and maximum processing time for any data item. On receiving the complete data set, the algorithm should decide as fast as possible either to start another pass or output an (approximately) accurate answer with high probability. A number of important algorithms have been developed in streaming model of computation, e.g., estimating frequency moments [1] and heavy hitters [8] of a data stream. These algorithms find applications in the context of network routing, where large amount of data flows through any router but each of them have limited memory to process the data. In Section 3, we develop the following streaming algorithm for hypergraph coloring.

Result 1. *We provide a randomized one-pass streaming algorithm to 2-color with high probability any n-uniform hypergraph that has at most $\frac{1}{10}\sqrt{\frac{n}{\ln n}}2^n$ hyperedges. This algorithm requires $O(|V|B)$ RAM space at any instant and $O(nB|V|)$ processing time after reading each hyperedge. On reading the terminal symbol, the algorithm spends $O(nB|V|)$ time and then outputs a valid 2-coloring or declares failure.*

Note that if the number of edges is much smaller than 2^n, then a random 2-coloring will with high probability be a proper coloring. In that case, we have a trivial streaming algorithm by choosing the coloring first, and verifying that it does properly color all the edges in just one pass.

We next consider the question of the minimum number of hyperedges in a non-2-colorable n-uniform hypergraph. Erdős [5] showed that there is an n-uniform hypergraph with $\Theta(n^2 2^n)$ hyperedges that is not 2-colorable. This construction uses $\Theta(n^2)$ vertices. Erdős and Lovász [6] conjectured that any n-uniform hypergraph with fewer than $n2^n$ hyperedges may be 2-colorable. We study if constructing a counter-example to this conjecture is possible with $o(n^2)$ vertices. Our result in Section 4 shows that with significantly fewer than n^2 vertices, we cannot construct a non-two-colorable hypergraph with fewer than $n2^n$ hyperedges.

Result 2. *For any n-uniform hypergraph with $|V| \leq \frac{n^2}{4\ln 2n}$ and $|E| < n2^n$, we provide a randomized one-pass streaming algorithm that outputs a 2-coloring with high probability. This algorithm requires $O(\frac{n^2}{\ln n})$ RAM space at any instant and $O(n)$ processing time after reading each hyperedge. On reading the terminal symbol, the algorithm spends $O(1)$ time and then outputs a valid 2-coloring or declares failure.*

Let us elaborate on the last sentences in each of the results above. When the algorithms terminate, we expect them to produce correct outputs with probability at least $1 - \delta$ for any given constant $\delta > 0$. Moreover, we insist that the algorithms produce one-sided errors, i.e., they either produce correct colorings or do not output any colorings. The motivation for insisting this is the following. Both of our algorithms execute $O(\log \frac{1}{\delta})$ processes in parallel, where each process has a small but constant probability of success. The probability that at least one of these processes is successful then becomes at least $1 - \delta$. To find out the one that is successful, we must examine the correctness of the output of a process after its termination. In each algorithm, we output a coloring if and only if we find a successful process.

Result 1 is obtained by modifying the delayed recoloring algorithm to a streaming version, and then showing that this modification results in the same coloring as before. Result 2 is obtained by a simple application of the union bound in probabilistic method.

1.1 Open Questions

We point out two interesting questions that should be settled using existing techniques.

- Can we show that no *deterministic* algorithm can match the performance of our randomized algorithms? In particular, can we show that there are two-colorable hypergraphs with (say) n^2 vertices and (say) 2^n edges such that every deterministic one-pass streaming algorithm that is guaranteed to two-color them requires superpolynomial (in n) RAM space?
- Our result on hypergraphs with $O(\frac{n^2}{t})$ vertices does not improve on the bound provided via the delayed recoloring algorithm when it t is small, say, $O(\log n)$. We believe, it should be possible to combine our argument and the delayed recoloring algorithm to show that if the number of vertices is $o(n^2)$ then we can two-color hypegraphs with strictly more than $\frac{1}{10}\sqrt{\frac{n}{\ln n}}2^n$ hyperedges. We do not have such a result.

2   The Delayed Recoloring Algorithm

We will present the delayed recoloring coloring algorithm in this section. The analysis of the algorithm can be found in Section 2 of [7].

We start by recalling our assumption that the vertices of the hypergraphs are not explicitly stored in the external memory. With this assumption, the delayed recoloring algorithm proceeds as follows. The I/O processor first reads and stores all the hyperedges in RAM, and then extract the vertices from them. Next, one of the $|V|!$ permutations of the vertices is selected uniformly at random. The algorithm then starts with a initial random coloring $\chi_0(V)$ of the vertices, and attempts to flip the color of each vertex once in the order defined by the permutation. A randomly generated boolean vector $b(V)$ is used to indicate whether the color of a vertex is allowed to be flipped ($b(v) = 1$) or not ($b(v) = 0$). If allowed, the color of a vertex is flipped if and only if it belongs to an initially monochromatic hyperedge that continues to be monochromatic till this vertex in considered. Let us now describe the algorithm in detail.

I/O (read). Read all the hyperedges and store them in RAM.

Step 1. Extract the vertices from these hyperedges.

Step 2. Select one of the $|V|!$ permutations of the vertices uniformly at random. Denote this permutation by $P(V) = [v_1, v_2, \ldots v_{|V|}]$.

Step 3. Color each vertex $v \in V$ red or blue independently with probability $\frac{1}{2}$. Denote this coloring by $\chi_0(V)$.

Step 4. Set $b(v) = 1$ with probability p (to be specified below) and $b(v) = 0$ with probability $1 - p$, independently for each vertex $v \in V$. Denote this vector by $b(V)$.

Step 5. Recolor the vertices in $|V|$ steps as follows.

   Step 5(1).If v_1 is contained in a hyperedge that is monochromatic in $\chi_0(V)$, then flip the color of v_1 if and only if $b(v_1) = 1$. Denote this coloring by $\chi_1(V)$.

   $\ldots$

Step $5(i)$. If v_i is contained in a hyperedge that is monochromatic in $\chi_0(V)$ and continues to be monochromatic in $\chi_{i-1}(V)$, then flip the color of v_i if and only if $b(v_i) = 1$. Denote this coloring by $\chi_i(V)$.

I/O (write). Write the coloring $\chi_{|V|}(V)$ in external memory if and only if it is a valid 2-coloring of the hypergraph.

Let $\chi(V) = \chi_{|V|}(V)$. It has been shown in [7] that if $|E| \leq \frac{1}{10}\sqrt{\frac{n}{\ln n}}2^n$, then
1. Pr[there exists a monochromatic hyperedge in $\chi_0(V)$ that remains monochromatic in $\chi(V)$] $\leq \frac{2}{10}\sqrt{\frac{n}{\ln n}}(1-p)^n$.
2. Pr[there exists a non-monochromatic hyperedge in $\chi_0(V)$ which became blue in $\chi(V)$] $\leq$ Pr[there exists a non-monochromatic hyperedge in $\chi_0(V)$ whose red vertices became blue in $\chi(V)$] $\leq \frac{2np}{100 \ln n}$.
3. Pr[there exists a non-monochromatic hyperedge in $\chi_0(V)$ which became red in $\chi(V)$] $\leq$ Pr[there exists a non-monochromatic hyperedge in $\chi_0(V)$ whose blue vertices became red in $\chi(V)$] $\leq \frac{2np}{100 \ln n}$.
Note that the above three events are the only bad events which can make the 2-coloring $\chi(V)$ of a hypergraph invalid. The probability that any of these bad events happens is at most $\frac{2}{10}\sqrt{\frac{n}{\ln n}}(1-p)^n + \frac{4np}{100 \ln n}$. If $p = \frac{\ln n}{2n}$, this probability is at most $\frac{2}{10}\sqrt{\frac{1}{\ln n}} + \frac{2}{100}$. For any $n \geq 2$, this quantity is less than $(\frac{2}{10}\sqrt{\frac{1}{\ln 2}} + \frac{2}{100}) < \frac{1}{2}$. This means that the above algorithm produces a valid 2-coloring with probability at least $\frac{1}{2}$.

To improve this success probability to at least $1 - \delta$ for any given constant $\delta > 0$, we need to execute steps $2 - 5$ of the algorithm $O(\log \frac{1}{\delta})$ times in parallel. It can be seen that the probability that at least one of these parallel processes succeeds in producing a valid coloring is at least $1 - \delta$. It takes $O(n|E|B|V|)$ time to check whether a coloring is valid. We choose any one of the successful processes (if one exists) and write the coloring produced by it in the external memory.

Let us verify that the delayed recoloring algorithm can be executed in $O(nB|V||E|)$ time. Step 1 requires $O(nB|V||E|)$ time to identify all distinct vertices and store them in a sequence Q. Step 2 can be implemented in $O(B|V|^2)$ time as follows.

Initialize $P(V)$ to the permutation in which the vertices appear in Q. For each $1 \leq i \leq |V|$, do the following three operations. First, generate a random number j from $[1, i]$. Then, place the i-th vertex of Q at j-th position of permutation $P(V)$. Vertices that used to appear on or after j-th position in $P(V)$ are now shifted one place each to produce a modified $P(V)$.

Steps $3 - 4$ can be implemented by creating bit-vectors for the set of vertices, and therefore require $O(|V|)$ time each. The k-th position in these bit-vectors corresponds to the k-th position in the permutation $P(V)$. Finally, step $5(i)$ requires $O(nB|E|)$ time for any i, which implies that the total running time for step 5 is $O(nB|V||E|)$.

Note that the I/O(read) step is the reason we require high RAM space requirement in this implementation of the delayed recoloring algorithm. However, this algorithm can be implemented by just storing the vertices and the hyperedges

that are monochromatic after the first random coloring, and therefore requires $O(|V|B + nB\sqrt{\frac{n}{\ln n}})$ RAM space at any instant with high probability. This implementation, however, does not verify that the coloring is proper, and might output invalid colorings. The algorithm in the next section is guaranteed never to do this.

3   An Efficient Streaming Algorithm

In this section, we present a modified algorithm that uses $O(|V|B)$ RAM space to store the vertices at any instant, but the total processing time remains asymptotically the same.

Assume that the hyperedges are stored in the external memory in the order $h_1, h_2, \ldots, h_{|E|}$. The I/O processor reads the hyperedges from left to right and stores only the current hyperedge in RAM. New vertices, if any, are extracted form a hyperedge once it is read and stored in the RAM, along with previously extracted vertices. The set of vertices after hyperedge h_i is read is denoted by V_i. Once hyperedge h_i is read, the algorithm first assigns initial colors χ_0 to vertices belonging to $V_i \setminus V_{i-1}$, and then attempts to recolor the vertices belonging to h_i in the order defined by a uniformly random permutation $P(V_i)$ of the vertices extracted so far. As before, a randomly generated boolean vector $b(V_i)$ is used to indicate whether the color of a vertex is allowed to be flipped ($b(u) = 1$) or not ($b(u) = 0$). A vertex is recolored at most once. Let us now describe the modified algorithm in detail.

Initialize $V_0 = \emptyset$. For $1 \leq i \leq |E|$, do each of the following steps.

I/O (read). Read the hyperedge h_i and store it in RAM. Delete h_{i-1} from RAM.

Step 1(i). Extract and add new vertices of h_i to V_{i-1}, the set of vertices currently stored in RAM. Denote this modified set of vertices by V_i. Note that $V_{|E|} = V$.

Step 2(i). Sequentially insert each vertex of $V_i \setminus V_{i-1}$ uniformly at random into the existing random permutation $P(V_{i-1})$ of the vertices belonging to V_{i-1}. If $|V_i| = j$, denote this permutation by $P(V_i) = [v_1, v_2, \ldots, v_j]$.

Step 3(i). For each vertex $u \in V_i \setminus V_{i-1}$, set $\chi_0(u)$ to red or blue independently with probability $\frac{1}{2}$. Keep $\chi_0(u)$ unchanged for each vertex $u \in V_{i-1}$.

Step 4(i). Set $b(u) = 1$ with probability p' (to be chosen later) and $b(u) = 0$ with probability $1 - p'$, independently for each vertex $u \in V_i \setminus V_{i-1}$. Keep $b(u)$ unchanged for each vertex $u \in V_{i-1}$.

Step 5(i). If h_i is monochromatic in $\chi_0(V_i)$, find the first (in permutation $P(V_i)$) vertex $u \in h_i$ with $b(u) = 1$. Flip the color of u if and only if it has not been flipped before. Keep the colors of the vertices belonging to $V_i \setminus \{u\}$ unchanged. Denote the new coloring of the vertices by $\chi_i'(V_i)$.

I/O (write). Write the coloring $\chi'_{|E|}(V)$ in external memory if and only if it is a valid 2-coloring of the hypergraph.

Let $\chi'(V) = \chi'_{|E|}(V)$. Recall that the original delayed recoloring algorithm produces a coloring $\chi(V)$ at its termination. We show below that $\chi'(V)$ is the same as $\chi(V)$ for any given $\chi_0(V)$, $P(V)$ and $b(V)$.

Lemma 1. *For each $v \in V$, $\chi'(v) = \chi(v)$ for any given χ_0, P and b.*

Proof. First, let us assume to the contrary that there exists a vertex u with $\chi_0(u) = \chi'(u) \neq \chi(u)$. Let us also assume that h is a hyperedge that necessitated u's recoloring in the original algorithm. In other words, h was one of the monochromatic hyperedges in $\chi_0(V)$ that remained monochromatic till u was considered in one of the sub-steps of step 5. This must have happened due to the fact that the vertices of h that appeared before u in $P(V)$ had their corresponding b bits set to 0. Therefore, when h is considered in the modified algorithm, u must be its first vertex (in permutation $P(V)$) with $b(u) = 1$. The color of u is flipped in this step of the algorithm, if not already flipped in a previous step. Thus, $\chi(u) \neq \chi'(u)$. This contradicts the assumption above.

On the other hand, let us assume to the contrary that there exists a vertex u' with $\chi_0(u') = \chi(u') \neq \chi'(u')$. Let h' be the hyperedge that necessitated the recoloring of u' in the current algorithm. This means that all vertices (of h') before u' (in permutation $P(V)$) had their b bits set at 0. This implies that u' must have been recolored in $\chi(V)$, because h' would have remained monochromatic till u'-th recoloring sub-step of step 5. Thus, $\chi(u') \neq \chi'(u')$. Again, this contradicts the above assumption.   □

This lemma implies upper bounds on the probabilities of the following three bad events:

1. Pr[there exists a monochromatic hyperedge in $\chi_0(V)$ that remains monochromatic in $\chi'(V)$] $\leq \frac{2}{10}\sqrt{\frac{n}{\ln n}}(1 - p')^n$.

2. Pr[there exists a non-monochromatic hyperedge in $\chi_0(V)$ whose red vertices became blue in $\chi'(V)$] $\leq \frac{2np'}{100 \ln n}$.

3. Pr[there exists a non-monochromatic hyperedge in $\chi_0(V)$ whose blue vertices became red in $\chi'(V)$] $\leq \frac{2np'}{100 \ln n}$.

Note that if none of the above events takes place, the algorithm produces a proper 2-coloring $\chi'(V)$. It can be easily seen that the RAM requirement at any instant is only $O(|V|B)$. Note, however, that the above analysis does not have the desirable property that it outputs only valid colorings. A straight-forward check does not seem possible using $O(|V|B)$ RAM space. We show below that with carefully storing some vertices of a few hyperedges, we can in fact achieve this using only $O(nB)$ RAM space.

Checking whether a coloring is proper using $O(nB)$ RAM space

In order to check whether the first event takes place for a hyperedge h, it is sufficient to look into the corresponding χ_0 and b bits as soon as the hyperedge

is read. The first event takes place for h if and only if it is monochromatic in χ_0 and all its b bits are set at 0. However, it is not immediately possible to determine after reading h whether the second (similarly, the third) event takes place because of it. Therefore, we mark h as a *potentially blue* hyperedge if all its red vertices have their corresponding b bits set at 1. In h is such a hyperedge, we store this subset R_h of red vertices in memory. At the end of $|E|$-th recoloring step, we can check whether all red vertices of h flipped their colors or not. Note that hyperedges those are not potentially blue cannot cause the second event. Similarly, we can mark a hyperedge as *potentially red* and store its subset B_h of blue vertices. For any given h, let us now calculate the expected number of vertices in R_h.

$$E[|R_h|] = \sum_{i=1}^{n} i \cdot \binom{n}{i} \cdot 2^{-n}(p')^i = np'2^{-n} \sum_{i=1}^{n} \binom{n-1}{i-1}(p')^{i-1} \leq np'2^{-n}e^{p'n}.$$

Therefore, the expected total number of vertices in R_h's of all potentially blue hyperedges is at most $\frac{1}{10} \cdot \sqrt{\frac{n}{\ln n}} \cdot np' \cdot e^{p'n}$. By Markov's inequality,

Pr[total number of vertices in R_h's of all potentially blue hyperedges $\geq \frac{100}{10} \cdot \sqrt{\frac{n}{\ln n}} \cdot np' \cdot e^{p'n}] \leq \frac{1}{100}$.

Similarly, we can bound by $\frac{1}{100}$ the probability of the bad event that the total number of vertices in B_h's of all potentially red hyperedges is more than $\frac{100}{10} \cdot \sqrt{\frac{n}{\ln n}} \cdot np' \cdot e^{p'n}$. As a result, the probability that any one of the bad events takes place is at most $\frac{2}{100} + \frac{2}{10}\sqrt{\frac{n}{\ln n}}(1-p')^n + \frac{4np'}{100\ln n}$. If $p' = \frac{\ln n}{2n}$, the probability of success is still at least $\frac{1}{2}$ as in Section 2. However, the total number of vertices in R_h's of all potentially blue hyperedges is at most $10\sqrt{\frac{n}{\ln n}} \cdot \frac{\ln n}{2} \cdot \sqrt{n} = O(n\sqrt{\ln n})$. To get the claimed bound on RAM space, we need to choose p' more carefully. In particular, we slightly reduce the value of p'.

If $p' = \frac{\ln n}{2n} - \frac{\ln\ln n}{2n}$, the probability that one of the bad events happens is at most $\frac{2}{100} + \frac{2}{10} + \frac{2}{100}$. This means the the success probability is at least $\frac{76}{100}$. With this value of p', the total number of vertices in R_h's of all potentially blue hyperedges is at most $10\sqrt{\frac{n}{\ln n}} \cdot \frac{\ln n}{2} \cdot \sqrt{\frac{n}{\ln n}} = O(n)$. Each vertex takes B bits to store, and so total RAM space required to store all R_h's and R_b's is at most $O(nB)$. This implies that the total RAM space required for the algorithm is $O(|V|B + nB) = O(|V|B)$.

For any given constant $\delta > 0$, we can repeat steps $1(i)$ to $5(i)$ of the algorithm $O(\log\frac{1}{\delta})$ times in parallel to improve the probability of success to at least $1 - \delta$. The RAM requirement for this algorithm then becomes $O(|V|B\log\frac{1}{\delta})$.

It can be easily checked that the processing time for steps $1(i)$ to $5(i)$ is $O(nB|V|)$ for each $1 \leq i \leq |E|$. This implies a total of $O(nB|V||E|)$ processing time for the entire algorithm. On reading the terminal symbol, the algorithm requires to look at the colors of $O(n)$ vertices belonging to R_h's and R_b's to determine the validity of the coloring, and $O(nB|V|)$ processing time is required for this purpose. This completes the proof of Result 1.

4 Coloring Uniform Hypergraphs with $O(\frac{n^2}{\ln n})$ Vertices

In this section, we first show that any hypergraph with at most $\frac{n^2}{4 \ln 2n}$ vertices and fewer than $n2^n$ hyperedges can be two colored. Erdős [5] constructed a random n-uniform hypergraph with $\Theta(n^2)$ vertices and $n^2 2^n$ hyperedges that is not 2-colorable [2]). Erdős and Lovász [6] conjectured that any n-uniform hypergraph with fewer than $n2^n$ hyperedges may be 2-colorable. The following lemma proves that constructing any counterexample to the conjecture of Erdős and Lovász [6] requires more than $\frac{n^2}{4 \ln 2n}$ vertices.

Lemma 2. *Any n-uniform hypergraph with fewer than $n2^n$ hyperedges and at most $\frac{n^2}{4 \ln 2n}$ vertices can be 2-colored.*

Proof. Let us assume that the hypergraph has $2Cn$ vertices (C is a function of n). We partition the vertex set randomly into two parts and color each vertex in one of them by red and the other by blue. Let us calculate the probability that a hyperedge e is monochromatic in this coloring $\chi(V)$. Let p be the probability that e is monochromatic in $\chi(V)$.

$$p = \frac{\binom{2Cn-n}{Cn}}{\binom{2Cn}{Cn}} = \frac{Cn \cdot (Cn-1) \cdot (Cn-2) \cdots (Cn-n+1)}{2Cn \cdot (2Cn-1) \cdot (2Cn-2) \cdots (2Cn-n+1)}$$

$$\leq 2^{-n} \cdot (1 - \frac{1}{4C-1})^{\frac{n}{2}} \leq 2^{-n} \cdot e^{\frac{-n}{8C-2}}.$$

If $C \leq \frac{n}{8 \ln 2n}$, it follows that p is strictly less than $\frac{1}{2n2^n}$. Therefore, the probability that at least one edge is monochromatic in the coloring $\chi(V)$ is at most $|E| \cdot \frac{1}{2n2^n} < \frac{1}{2}$. This implies that there is a proper 2-coloring of the hypergraph. □

Note that the above proof suggests the following result. If $C = \frac{n}{t}$ for a parameter t, then there is a 2-coloring of a hypergraph with at most $2^n \exp(\frac{t}{8})$ hyperedges. This hypergraph has $2Cn = \frac{2n^2}{t}$ vertices. We show below by an argument similar to the one used by Erdős [5] that if $t = o(n)$, there exists a hypergraph with $\frac{2n^2}{t}$ vertices and $O(\frac{n^2}{t} \cdot 2^n \exp(\frac{t}{2}))$ hyperedges that is not 2-colorable.

Consider a 2-coloring of the vertex set, whose size is $\frac{2n^2}{t}$. Pick a random hyperedge of size n. The probability that this hyperedge is monochromatic is at least

$$p = \frac{\binom{\frac{n^2}{t}}{n}}{\binom{\frac{2n^2}{t}}{n}} \geq \left(\frac{\frac{n^2}{t} - n}{\frac{2n^2}{t} - n} \right)^n \approx 2^{-n} \left(1 - \frac{t}{2n} \right)^n \approx 2^{-n} \exp(\frac{-t}{2}).$$

Let $S_1, S_2, \ldots, S_r$ be uniformly and independently chosen hyperedges. The probability that none of these hyperedges is monochromatic is at most $\left(1 - 2^{-n} \exp(\frac{-t}{2})\right)^r 2^{\frac{2n^2}{t}}$. If $\left(1 - 2^{-n} \exp(\frac{-t}{2})\right)^r 2^{\frac{2n^2}{t}} < 1$, there exists a hypergraph with r hyperedges that is not 2-colorable. Since $(1-x)^r \leq \exp(-xr)$, this inequality is satisfied when $r \geq \frac{2n^2}{t} \cdot 2^n \exp(\frac{t}{2}) \cdot \ln 2$.

Lemma 2 can be extended for k-coloring of n-uniform hypergraphs with vertex set size $O(\frac{n^2}{\ln n})$ but having fewer than nk^n hyperedges.

Lemma 3. *Any n-uniform hypergraph with fewer than nk^n hyperedges drawn from a set of at most $\frac{(k-1)n^2}{4\ln 2n}$ vertices can be k-colored.*

Proof. As before, let us assume that the hypergraph has kCn vertices. We partition the vertex set randomly into k parts and color them by k different colors. Let us calculate the probability that a hyperedge e is monochromatic in this coloring $\chi(V)$. Let p be the probability that e is monochromatic in $\chi(V)$.

$$p = \frac{\binom{kCn-n}{Cn-n} \cdot \binom{(k-1)\cdot Cn}{Cn} \cdots \binom{Cn}{Cn}}{\binom{kCn}{Cn} \cdot \binom{(k-1)\cdot Cn}{Cn} \cdots \binom{Cn}{Cn}} \le k^{-n} \cdot (1 - \frac{k-1}{2kC-1})^{\frac{n}{2}} \le k^{-n} \cdot e^{\frac{-n(k-1)}{4Ck-2}}.$$

If $C \le \frac{(k-1)n}{4k\ln 2n}$, it follows that p is strictly less than $\frac{1}{2nk^n}$. Therefore, the probability that at least one hyperedge is monochromatic in the coloring χ is at most $|E| \cdot \frac{1}{2nk^n} < \frac{1}{2}$. This implies that there is a proper k-coloring of the hypergraph. □

The above two lemmas also show that there exists an *equitable k-coloring* of a hypergraph with fewer than nk^n hyperedges if it has $O(\frac{n^2}{\ln n})$ vertices. In fact, both these proofs can easily be transformed into randomized streaming algorithms to find such colorings. At the beginning, we store the colors of the vertices (half red and remaining half blue) in a bitvector of size $O(\frac{n^2}{\ln n})$. With the arrival of each hyperedge, we just need to check whether it is monochromatic by checking the corresponding bits. If there are fewer than nk^n hyperedges, the probability of success (no hyperedge is monochromatic) is at least $\frac{1}{2}$. By repeating this algorithm $\log \frac{1}{\delta}$ times in parallel, we can improve the probability of success to at least $1 - \delta$. The memory space requirement for this algorithm is $O(\frac{n^2}{\ln n} \log \frac{1}{\delta})$.

In the following, we derandomize this algorithm to k-color any hypergraph with at most $\frac{(k-1)n^2}{4\ln 2n}$ vertices and fewer than nk^n hyperedges. We derandomize using conditional expectations, in a way it is used to derandomize the algorithm to find a large cut in a graph [10]. We first provide the algorithm for $k = 2$. Let us assume that $|V|$ is even and the vertices are $v_1, v_2, \ldots, v_{|V|}$. The order of vertices in which they are colored will be denoted by $u_1, u_2, \ldots, u_{|V|}$.

Step 1: Color v_1 by red, and call this vertex u_1.

Step 2: For $j = 2$ to n, calculate the expected number of monochromatic hyperedges conditioned on coloring v_j by red. Select the vertex that gives the lowest expectation and color it by red. Call this vertex u_2.

Step $i \le \frac{|V|}{2}$: After the $(i-1)$-th step, the colors of $u_1, u_2, \ldots, u_{i-1}$ are red. For $j = i$ to n, calculate the expected number of monochromatic hyperedges conditioned on coloring v_i by red. Select the i-th red vertex as the one that gives the lowest expectation and call it u_i.

Step $\frac{|V|}{2} + 1$: Color all the remaining vertices by blue.

It can be easily seen that such an algorithm produces a deterministic 2-coloring in time $O(|V|^2|E|)$. The space requirement, however, becomes $O(|E|B)$.

A similar deterministic algorithm exists for k-coloring as well. We first find the vertices with color1, followed by color2, ..., colork.

5   Streaming and the Lovász Local Lemma

Radhakrishnan and Srinivasan [7] considered the local version of the problem of 2-coloring. In this section, we mention a streaming algorithm for this version. In particular, we observe from the parallel version of the algorithm of Moser and Tardos [11] that for any given constant $\epsilon > 0$, there exists an $O(\log|V|)$-pass streaming algorithm to 2-color any n-uniform hypergraph none of whose hyperedges intersects more than $\frac{(1-\epsilon)2^{n-1}}{e} - 1$ other hyperedges. This algorithm requires $O(|V|B)$ memory space. For such hypergraphs, it is an interesting open problem to find an $O(1)$-pass streaming algorithm that uses asymptotically just enough RAM space to store the vertices.

Moser and Tardos [11] recently proposed a parallel algorithm for Lovász Local Lemma, which we describe below. Let X be a finite set of events determined by a finite set P of mutually independent random variables, such that each event of X is determined by a subset of the variables in P. Let G_X denote the dependency graph of the events, i.e., two events A and B are connected by an edge if and only if they share common variables. For any event $A \in X$, we denote by $N(A)$ the events which are neighbors of A in G_X. An assignment or evaluation of the variables *violates* an event A if it makes A happen. With these notations, we explain their algorithm:

Step 1. Evaluate each variable in P independently at random.

Step 2. If there exists at least one violated event in X, construct a maximal independent set M of the sub-graph (of G_X) induced by the violated events in X. Independently perform random re-evaluation of each variable that belongs to one of the events of M.

Step 3. If there are no violated events, output the current evaluation of the variables. Otherwise, go to Step 2.

If a local condition is assumed to hold for each of the events, the following theorem bounds the expected number of times Step 2 of the algorithm is executed.

Theorem 1. *[11] If $\epsilon > 0$ and there exists real number assignments $x : X \to (0,1)$ such that*

$$\forall A \in X : \Pr[A] \leq (1-\epsilon)x(A) \prod_{B \in N(A)} (1 - x(B)), \tag{1}$$

then the algorithm executes step 2 an expected $O(\frac{1}{\epsilon} \log \sum_{A \in X} \frac{x(A)}{1-x(A)})$ number of times before it finds an evaluation of P violating no event in X.

For each hyperedge h of a hypergraph H, let X_h denote the event that h is monochromatic in a 2-coloring of H and let $X = \{X_h : h \in H\}$. Therefore,

$\Pr[X_h] = 2^{1-n}$. If each hyperedge of H intersects at most $\frac{(1-\epsilon)2^{n-1}}{e} - 1$ other hyperedges, we can assign $x(H_h) = \frac{e}{(1-\epsilon)2^{n-1}}$ to satisfy Equation 1 for all $X_h \in X$. (We use $(1 - \frac{1}{r+1})^r \geq e^{-1}$ for any $r \geq 1$.)

In our streaming algorithm, we start with a uniformly random coloring of the vertices. Thereafter, Step 2 of the algorithm can be executed once in each pass as follows. Whenever an hyperedge arrives, we mark and store its vertices in memory if and only if it is monochromatic in the current coloring and does not intersect with any of the previously marked hyperedges. Since at most $\frac{|V|}{n}$ disjoint hyperedges can be marked in one pass, only $O(|V|B)$ memory space is required to store all their vertices. At the end of each pass, we randomly re-evaluate the colors of each of the marked vertices and return to the beginning of the secondary memory. The algorithm stops if and only if there are no monochromatic hyperedges in the current coloring. By Theorem 1, this algorithm terminates after $O(\log \frac{|E|}{2^{n-1}}) = O(\log |V|)$ expected number of passes.

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. In: Proceedings of the ACM Symposium on Theory of Computing (STOC), pp. 20–29 (1996)
2. Alon, N., Spencer, J.H.: The Probabilistic Method. Wiley-Interscience Series. John Wiley and Sons, Inc., New York (1992)
3. Beck, J.: On 3-chromatic hypergraphs. Discrete Mathematics 24, 127–137 (1978)
4. Erdős, P.: On a combinatorial problem. Nordisk Mat. Tidsskr 11, 5–10 (1963)
5. Erdős, P.: On a combinatorial problem, II. Acta Mathematica of the Academy of Sciences 15, 445–447 (1964)
6. Erdős, P., Lovász, L.: Problems and results on 3-chromatic hypergraphs and some related questions. Colloq. Math. Soc. Janos Bolyai 10, 609–627
7. Radhakrishnan, J., Srinivasan, A.: Improved bounds and algorithms for hypergraph 2-coloring. Random Structures Algorithms 16(1), 4–32 (2000); (also in FOCS 1998)
8. Karp, R.M., Papadimitriou, C.H., Shenker, S.: A simple algorithm for finding frequent elements in streams and bags. ACM Transactions on Database Systems 28, 51–55 (2003)
9. Kostochka, A.: Coloring uniform hypergraphs with few colors. Random Structures Algorithms 24(1), 1–10 (2004)
10. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
11. Moser, R.A., Tardos, G.: A constructive proof of the general Lovász local lemma. Journal of the ACM 57(2), 1–15 (2010)
12. Yuster, R.: Equitable coloring of k-uniform hypergraphs. SIAM Journal on Discrete Mathematics 16(4), 524–532 (2003)

Density-Constrained Graph Clustering[*]

Robert Görke, Andrea Schumm, and Dorothea Wagner

Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany

Abstract. Clusterings of graphs are often constructed and evaluated with the aid of a quality measure. Numerous such measures exist, some of which adapt an established measure for graph cuts to clusterings. In this work we pursue the problem of finding clusterings which simultaneously feature guaranteed intra- and good intercluster quality. To this end we systematically assemble a range of cut-based bicriteria measures and, after showing $\mathcal{NP}$-hardness for some, focus on the classic heuristic of constrained greedy agglomeration. We identify key behavioral traits of a measure, (dis-)prove them for each one proposed and show how these translate to algorithmic efficiency.

1 Introduction

The guiding intuition in the field of graph clustering is "intracluster density vs. intercluster sparsity". Mathematical formalizations thereof abound, most of which, however, incorporate both aspects into a single criterion, which then serves as a quality measure for graph clusterings. Balance between the two aspects is a fine line and treating them separately allows to adjust their tradeoff as to fit given desiderata. While the recent literature on graph clustering (we recommend [10,7] for an overview) has mainly been focusing on large data sets and on single criteria such as Modularity [12], Kannan et al. [11] propose to minimize the cut between, subject to a guaranteed conductance within the clusters and show that this approach avoids the drawbacks of many simpler measures. This stepping stone in bicriterial graph clustering inspired Flake et al. [6], who give an algorithm with provable, but interdependent bounds on both intra- and a variant intercluster expansion. Brandes et al. [3] were the first to use a notion of intercluster conductance to experimentally evaluate clustering algorithms.

   Together with sparsity, expansion and conductance are well-known and indisputable measures for quantifying the clarity of a cut, and each one suggests adaptions to measuring clusterings with respect to both aspects. However, only very few have so far been coined and used. We systematically assemble such measures and set our main focus on scrutinizing their behavior in the light of the question which combinations enable efficient greedy agglomeration, putting aside other algorithmic approaches [7]. This classic hierarchical technique used for clustering [7,2,5] starts with singletons and iteratively merges clusters, usually driven by some objective function, until a stopping criterion is met. We

---

show that algorithmic efficiency and behavior strongly depends on three traits of a measure, roughly described as the range of feasible merges in terms of connectedness, the robustness of comparisons between merges and the monotonicity of a measure. For each established measure we prove or disprove these traits, leading both to qualitative insights and assertions on time and space complexity. We further motivate the use of a greedy heuristic by showing $\mathcal{NP}$-hardness for some of our problem statements and give a brief discussion how the constraints and objectives we use can be cast into integer linear programs. Systematic experiments evaluating how well these measures conform to human intuition and how well the proposed algorithms discover existing clusterings are beyond the scope of this work. Many proofs and details are deferred to the full version [9].

Notation and Preliminaries. Let $G = (V, E)$ be an undirected, unweighted, and simple graph[1]. We set $|V| =: n, |E| =: m$ and $\mathcal{C} = \{C_1, \ldots, C_k\}$ to be a partition of V. We call $\mathcal{C}$ a *clustering* of G and sets C_i *clusters*. Note that we restrict ourselves to disjoint clusters in this work. The cluster containing vertex v is $\mathcal{C}(v)$. A clustering is *trivial* if either $k = 1$ (*all-clustering*), or each cluster contains only one element (*singletons*). $\binom{\mathcal{C}}{2}$ denotes the set of all unordered pairs of clusters. We identify cluster C with the set of nodes it constitutes and with its vertex-induced subgraph of G. Then $E(\mathcal{C}) := \bigcup_{C \in \mathcal{C}} E(C)$ are called *intracluster* edges and $E \setminus E(\mathcal{C})$ *intercluster* edges. For two subsets A and B of V, $m_{A,B} := |\{\{u, v\} \in E \mid u \in A, v \in B\}|$ is the number of edges between A and B, $n_A := |A|$ is the number of vertices in A, $m_A := |E(A)|$ is its number of intracluster edges and $x_A := m_{A,V\setminus A}$ the number of intercluster edges incident to A. Further, the *volume* v_A of A is defined as $v_A := \sum_{v \in A} \deg(v)$. For $A \neq B \in \mathcal{C}$, we call $\{A, B\}$ a *merge* and abbreviate $AB := A \cup B$. Then, $\mathcal{C}_{\{A,B\}} := \mathcal{C} \setminus \{A, B\} \cup \{AB\}$ is the result of this merge. A *clustering measure* is a function that maps clusterings to real numbers, thereby assessing the quality of a clustering. We define high quality to correspond to high (low) values of intracluster (intercluster) measures.

2   Quality Measures for Clusterings

The bicriteria measures we construct and use in this work build upon conductance, expansion and density. The *conductance* of a cut (S, T) measures the bottleneck between S and T, defined as $\frac{m_{S,T}}{\min\{v_S, v_T\}}$; *expansion* substitutes volume by cardinality: $\frac{m_{S,T}}{\min\{n_S, n_T\}}$. The *density* (or *sparsity*) of a cut is $\frac{m_{S,T}}{n_S n_T}$, which equals the *uniform minimum-ratio cut*; the *density of a graph* is $\frac{m}{0.5n(n-1)}$, and the *conductance of a graph* is $\min_{S \subseteq V} \frac{m_{S,V\setminus S}}{\min\{v_S, v_{V\setminus S}\}}$ (expansion is analogous). *Intracluster measures* quantify how well the vertices within a cluster are interconnected. For *intercluster measures*, we distinguish two ways of measuring cuts: between pairs of clusters (*pairwise*), or cutting off a cluster (*isolating*). Hereby, isolated measures assess how well a cluster is separated from the remainder of the graph and pairwise measures how well the clusters are separated from one

---

[1] A *simple* graph in this work is both loopless and has no parallel edges.

another. To have these quantities express the quality of an entire clustering, we can either construct a worst-case measure (*minimum/maximum*) or an *average* measure. Density works analogously, however, it also lends itself to the natural idea of adding up all values before normalization (*global*). A simple alternative is globally counting intercluster edges. For convenience we list the measures thus constructed in Tables 1 and 2 and henceforth use their abbreviations.

The very evaluation of both the conductance and the expansion of a graph is $\mathcal{NP}$-hard ([1] and [8] respectively).While there are many ways to deal with this, it generally discourages the use of these functions as intracluster measures.[2] Density, by contrast, is well suited, yielding gid, mid and aid; we refer to a cluster C's contribution as $\mathsf{id}(C) = \frac{2m_C}{n_C(n_C-1)}$. Intercluster cuts, in turn, are efficiently computable. Thus, in accordance with the above classification, we define all twelve resulting intercluster measures (Tables 1 and 2), plus the two measures with a global nature: gxd and nxe. Adhering to our abbreviations, we denote individual clusters' contributions by $\mathsf{ixd}(C) := \frac{x_C}{n_C n_{V\setminus C}}$, $\mathsf{ixc}(C) := \frac{x_C}{\min\{v_C,v_{V\setminus C}\}}$ and $\mathsf{pxd}(\{C,D\}) := \frac{m_{C,D}}{n_C n_D}$, furthermore we call the number of intracluster edges $\mathsf{nie} := n - \mathsf{nxe}$. We generally define the intracluster density of a singleton to be 1, and, analogously, the intercluster quality of the all-clustering to be maximal. Any other choice is counterintuitive on trivial examples such as a clique or a clique plus an isolated vertex.

Table 1. Density and counting

intracluster density		
global	gid	$\frac{\sum_{C\in\mathcal{C}} m_C}{\sum_{C\in\mathcal{C}} \binom{n_C}{2}}$
minimum	mid	$\min_{C\in\mathcal{C}} \frac{m_C}{\binom{n_C}{2}}$
average	aid	$\frac{1}{|\mathcal{C}|}\sum_{C\in\mathcal{C}} \frac{m_C}{\binom{n_C}{2}}$

intercluster density		
global	gxd	$\frac{\sum_{A\neq B\in\mathcal{C}} m_{A,B}}{\sum_{A\neq B\in\mathcal{C}} n_A n_B}$
max. pw.	mpxd	$\max_{A\neq B\in\mathcal{C}} \frac{m_{A,B}}{n_A n_B}$
max. is.	mixd	$\max_{C\in\mathcal{C}} \frac{x_C}{n_C n_{V\setminus C}}$
av. pw.	apxd	$\binom{|\mathcal{C}|}{2}^{-1}\sum_{\{A,B\}\in\binom{\mathcal{C}}{2}} \frac{m_{A,B}}{n_A n_B}$
av. is.	aixd	$\frac{1}{|\mathcal{C}|}\sum_{C\in\mathcal{C}} \frac{x_C}{n_C n_{V\setminus C}}$

intercluster edges		
global	nxe	$\sum_{\{A,B\}\in\binom{\mathcal{C}}{2}} m_{A,B}$

Table 2. Intracluster measures based on conductance and expansion

		intercluster conductance			intercluster expansion
maximum pairwise	mpxc	$\max_{A\neq B\in\mathcal{C}} \frac{m_{A,B}}{\min\{v_A,v_B\}}$	mpxe	$\max_{A\neq B\in\mathcal{C}} \frac{m_{A,B}}{\min\{n_A,n_B\}}$
maximum isolated	mixc	$\max_{C\in\mathcal{C}} \frac{m_{C,V\setminus C}}{\min\{v_C,v_{C\setminus S}\}}$	mixe	$\max_{C\in\mathcal{C}} \frac{m_{C,V\setminus C}}{\min\{n_C,n_{C\setminus S}\}}$
average pairwise	apxc	$\frac{1}{\binom{|\mathcal{C}|}{2}}\sum_{\{A,B\}\in\binom{\mathcal{C}}{2}} \frac{m_{A,B}}{\min\{v_A,v_B\}}$	apxe	$\frac{1}{\binom{|\mathcal{C}|}{2}}\sum_{\{A,B\}\in\binom{\mathcal{C}}{2}} \frac{m_{A,B}}{\min\{n_A,n_B\}}$
average isolated	aixc	$\frac{1}{|\mathcal{C}|}\sum_{C\in\mathcal{C}} \frac{m_{C,V\setminus C}}{\min\{v_C,v_{V\setminus C}\}}$	aixe	$\frac{1}{|\mathcal{C}|}\sum_{C\in\mathcal{C}} \frac{m_{C,V\setminus C}}{\min\{n_C,n_{V\setminus C}\}}$

Qualitative Observations. While all proposed intra- and intercluster measures are based on the same intuition, there are fundamental differences in the

---

[2] Note that a bottom-up approach cannot use the approximation results used in [11].

way they assess particular clusterings. One important point is whether balanced clusterings, i.e., homogeneous cluster sizes, are rewarded or penalized. As an example aid has a tendency to favor unbalanced clusterings, as singletons degeneratively yield optimum values and it is easy to compensate the existence of a large cluster with poor intracluster density with an appropriate number of singletons. In contrast to that, mid rewards balanced clusterings, as clusters that are larger than the average are more likely to have low intracluster density and thus to be the qualitative bottleneck. Gid ranges somewhere between these extremes. Using the number of intercluster edges to measure intercluster quality clearly favors unbalanced clusterings, as cutting off small portions of the vertex set from the remainder of the graph usually cuts far fewer edges than partitioning the graphs in two blocks of roughly equal size. To some extent this effect can be compensated by combining nxe with an appropriate intracluster measure.

In the context of intercluster measures, another interesting aspect is how vertices that are only loosely connected to the remainder of the graph are handled. For example, singletons with degree one have a low intercluster density of $1/(|V| - 1)$ but maximum intercluster conductance of one. Thus, algorithms minimizing intercluster conductance are prone to put "outsiders" in the clusters of their neighbors, while algorithms minimizing intercluster density will tend to consider these vertices as singletons. Both views can be motivated, depending on the desiderata: If a vertex is linked to just a single vertex of a larger group, it can be hardly considered as an integral part of this group and should thus be treated separately. On the other hand, this vertex has no links to other groups and thus, from its point of view, it has a strong affiliation to its neighbor's group.

2.1   Problem Statement

In the following we narrow down the myriad formalizations for combining intra- and intercluster quality and state the problem we focus on. Not only do these two aspects capture different properties of a clustering, they even tend to oppose each other: Fine clusterings easily achieve high intracluster density but poor intercluster quality, while the converse is true for coarse clusterings. In the light of a bottom-up strategy, intercluster density aspires a coarse clustering and starts out poorly, which suggests using it as the driving objective function. By contrast, intracluster density starts out with optimum quality, which, on the one hand, discourages using it as the driving force, but, on the other hand, suggests it as a suitable constraint. We thus formalize our problem statement as follows, an exemplary instance and its solution are given in Fig. 1.

*Problem (*DENSITY-CONSTRAINED CLUSTERING*).* Given a graph $G = (V, E)$, among all clusterings with an intracluster density of no less than α, find a clustering $\mathcal{C}$ with optimum intercluster quality.

2.2   Complexity

An exhaustive study of hardness results for all combinations of intra- and intercluster measures is beyond the scope of this work. We exemplarily show
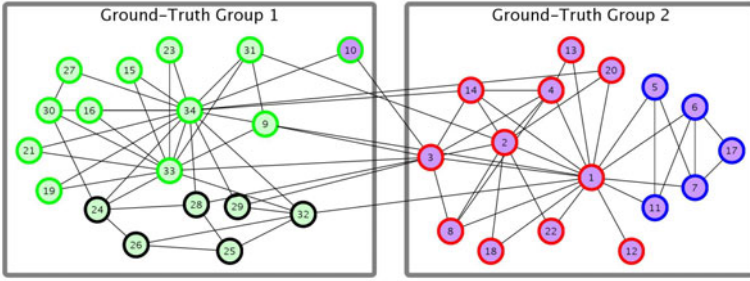
Fig. 1. Zachary's karate club [13] represents a social network and is traditionally used for a test of feasibility in the graph clustering literature. Groups represent the split of the network in reality, fill colors depict the optimal solution to our problem statement using nxe constrained by mid with $\alpha = 0.25$. Reviewing an optimal solution (see full version [9]) helps judging this measure's usefulness independently of an algorithmic approach. For comparison, border colors indicate a Modularity-optimal clustering [12]. By contrast, aid yields the all-clustering with the exception of one singleton vertex (12), pointing out its undesirable tendency to allow degeneratively imbalanced clusterings.

$\mathcal{NP}$-hardness for Density-Constrained Clustering combining mid, aid or gid with nxe, but conjecture $\mathcal{NP}$-hardness for all remaining combinations. We use that, if we set $\alpha = 1$, the decision variant is equivalent to the following problem.

*Problem (*Min-Cut Clique Partition*).* Given a graph $G = (V, E)$, is there a partition $\mathcal{C}$ of V into cliques such that nie$(\mathcal{C})$ is at least k?

This problem is similar to both the classic problem Partition into Cliques [8], which instead minimizes the *number* of cliques and the edge-maximizing variant of the $\mathcal{K}_r$-Packing Problem [4], which differs in that it only allows cliques with bounded size. To the best of our knowledge, Min-Cut Clique Partition has not yet been investigated. We reduce from Exact Cover by 3-Sets [8].

*Problem (*Exact Cover by 3-Sets, X3C*).* Given set $\mathcal{X}$ with $|\mathcal{X}| = 3q$ and collection $\mathcal{S}$ of 3-element subsets of $\mathcal{X}$. Does $\mathcal{S}$ contain an exact cover for $\mathcal{X}$, i.e., a subcollection $\mathcal{S}' \subseteq \mathcal{S}$ such that every $x \in \mathcal{X}$ occurs in exactly one $S \in \mathcal{S}'$?

We transform an instance $I = (\mathcal{X}, \mathcal{S})$ of X3C with $|\mathcal{X}| =: n$ into a graph $G(I)$ as follows. For each $x \in \mathcal{X}$ we add a vertex v_x, and interconnect the resulting set $V_{\mathcal{X}}$ into an n-clique. Then, we map each set $S \in \mathcal{S}$ to an n-clique $K_n(S)$. For $x \in S$, we link v_x with each vertex in $K_n(S)$. A sketch of this polynomial reduction is given in Fig. 2. For a proof of the following lemma see the full version [9].
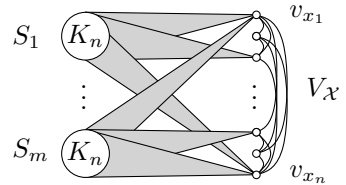
Fig. 2. Sketch of reduction

Lemma 1. *Let $I = (\mathcal{X}, \mathcal{S})$ be an instance of* X3C. *Then, I is solvable iff there exists a partition $\mathcal{C}$ of $G(I)$ into cliques such that* ixc$(\mathcal{C})$ *is at least* $|\mathcal{S}| \cdot \binom{n}{2} + n^2 + n$.

Corollary 2. MIN-CUT CLIQUE PARTITIONING *is $\mathcal{NP}$-hard.*

Algorithm 1. GENERIC GREEDY AGGLOMERATION

> **Input** : graph $G = (V, E)$, function objective, (constraint-) predicate allowed
> **Output**: clustering $\mathcal{C}$ of G
> $C \leftarrow$ singletons
> $\mathcal{A} \leftarrow \{\{A, B\} \in \binom{\mathcal{C}}{2} \mid$ allowed$(\mathcal{C}_{A,B})$ and objective$(\mathcal{C}_{A,B}) \leq$ objective$(\mathcal{C})\}$
> **while** $|\mathcal{C}| > 1$ *and* $\mathcal{A} \neq \emptyset$ **do**
> > $M \leftarrow \arg \min_{M \in \mathcal{A}} \{$objective$(\mathcal{C}_M)\}$
> > $\mathcal{C} \leftarrow \mathcal{C}_M$
> > $\mathcal{A} \leftarrow \{\{A, B\} \in \binom{\mathcal{C}}{2} \mid$ allowed$(\mathcal{C}_{A,B})$ and objective$(\mathcal{C}_{A,B}) \leq$ objective$(\mathcal{C})\}$
> **return** $\mathcal{C}$

---

3   Generic Greedy Agglomeration

The general structure of a greedy merge algorithm based on an objective function is given in Alg. 1. The idea is to choose from a constrained set of allowed merges the one that improves (w.l.o.g. minimizes) the objective function the most. The Modularity-based approach [5] fits into this concept if we set the objective function to be negative Modularity and use no constraint. Recalling our problem statement, our objective is to minimize intercluster density, subject to the restriction that no merge decreases intracluster density below a given threshold $\alpha \leq 1$. If allowed$(\mathcal{C}_{A,B})$ only depends on A and B, in each step of Alg. 1 at most $2n-3$ elements are deleted from and at most $n-2$ new elements are inserted into $\mathcal{A}$. Together with the condition that elements can be compared in constant time and that $\mathcal{A}$ can be maintained in a heap, using benefits as keys, the time complexity of Alg. 1 is in $O(n^2 \log n)$. Before we detail this observation (Sect. 3.2), we first determine whether intercluster measures as objectives efficiently drive greedy agglomeration, in that they iteratively suggest eligible merges.

3.1   Merge Behavior

An objective function f is said to have *unbounded merge behavior* if for any clustering $\mathcal{C}$ with at least two clusters, there exist clusters $A, B \in \mathcal{C}$, such that merge $\{A, B\}$ does not increase f. We elucidate the merge behavior of each proposed intercluster measure, either by proving its unboundedness or by giving an example instance which poses a local minimum. We defer the proofs of all affirmative observations to the full version [9], and summarize them as follows.

Proposition 3. *The intercluster measures* nxe, gxd, mixc, mixe, aixc, aixe, mixd *and* mpxd *exhibit unbounded merge behavior.*

Roughly speaking, the ingredient common to all proofs on maximum measures is the fact that, by investigating the adjacencies of the worst cluster B, we can always identify some worst contributor to B's value as an eligible partner

(a) mpxc, mpxe   (b) apxd, apxc, apxe   (c) aixd (edges are summarized)
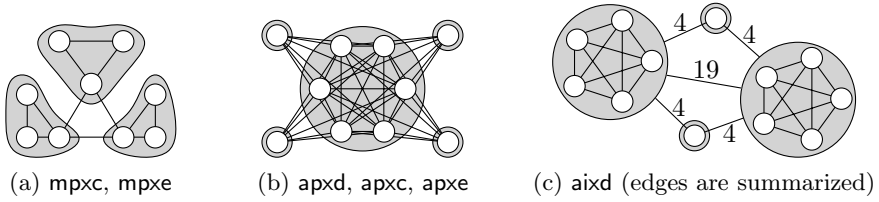
Fig. 3. These instances illustrate bounded merge behavior. Given clustering $\mathcal{C}$ (grey), no further merge is non-increasing for the measures pointed out.

for a merge. Likewise, aixc (aixe) allows us to find one or more clusters with a detrimental contribution, and then identify those adjacent clusters which are mainly liable for this as candidates for an improving merge. All the proofs are constructive in that they point out how to find a non-increasing merge.

Bounded Merge Behavior. From our set of fourteen objective functions, the remaining six do not have unbounded merge behavior, but can instead get stuck in local minima, such that no further merge is non-increasing. Thus, even without a constraint, the all-clustering cannot be reached. In Fig. 3 we give specific instances which are local minima of mpxc and mpxe (a), apxd, apxc and apxe (b), and of aixd (c). The common intuition for average measures is that a merge must not reduce the number of beneficial clusters (or pairs thereof) too dearly. Mpxc and mpxe are prone to local minima near balanced clusterings. Roughly speaking, this is due to the case distinction in the denominator of their base measures ruining arguments analogous to those usable for mpxd [9].

3.2 Impact of Clustering Measures on Running Times

We already gave conditions under which Alg. 1 can be implemented with a time complexity in $O(n^2 \log n)$. Here, we first review the constraints' impact, and then examine how the stated conditions can be relaxed without losing efficiency.

Intra-Density. Using constraints potentially impedes quick agglomeration, as it does not suffice to determine the merge that improves the objective function the most. The good news is that if we use mid as a constraint, the feasibility of a merge only depends on the density of the merged cluster, which is clearly independent of the remainder of the clustering, and thus need only be checked once, incurring no penalty in running time. However, for gid and aid, this does not hold, as the status of a merge can change from allowed to disallowed and back again. In Fig. 4, starting from the gray clustering, merging the path to the left is not allowed if the constraint $\mathsf{gid}(\mathcal{C}) \geq 0.7$ is used. If singletons A and B are merged, this is allowed again, as the number of intracluster pairs increases. A similar example for aid can be found in [9].
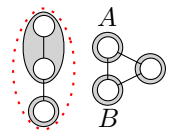
Fig. 4.

As a heuristic approach, if it is possible to store all merges in a binary search tree, sorted by their benefit to the objective function, this tree can be traversed

until we find a feasible merge. This might be more efficient than just searching through all possible merges, as the number of disallowed merges we encounter is limited to the number of more beneficial ones. However, in the worst case, it yields no improvement. We therefore focus our analysis on using mid as the constraint, supported by its good behavior in preliminary experiments, and leave a more efficient treatment of gid and aid open.

Locality. Independent of issues concerning the handling of constraints, in the following we resolve which properties an objective function f has to fulfill such that a set of feasible merges can efficiently be maintained in a heap. Intuitively, f should allow us to decide, without knowledge about the remainder of the clustering, which of two given merges is more beneficial to it. If the benefit exclusively depends on the participating clusters, as for nxe alone, this is immediate and decisions never change. For maximum isolated measures, by contrast, a merge can be non-improving at some point of the algorithm and then become improving again. The intuition is to require the existence of a relation $\leq_f$ that almost behaves like ordering the set of merges by their benefit for the objective function but allows for clever tie-breaking. To serve as a comparator in a priority queue, $\leq_f$ should closely resemble a total quasiorder on the set of all possible merges.[3] Informally, we call an objective function *local*, if it allows for such a relation. More formally, we get the following definition. Let us denote the set of all possible merges, i.e., the set of all unordered pairs of subsets of V, as $\mathcal{M}$.

Definition 4. *An objective function f on clusterings is* local, *if there exists a relation $\leq_f$ on $\mathcal{M} \times \mathcal{M}$ such that for any clustering $\mathcal{C}$ and for all M_1, M_2, M_3 in $\mathcal{M}$ with $M_1 \cup M_2 \cup M_3 \subseteq \mathcal{C}$, the following holds:*

$$M_1 \leq_f M_2 \text{ or } M_2 \leq_f M_1 \qquad\qquad \approx \textit{totality}$$
$$M_1 \leq_f M_2 \wedge M_2 \leq_f M_3 \implies M_1 \leq_f M_3 \qquad \approx \textit{transitivity}$$
$$M_1 \leq_f M_2 \qquad\qquad \implies f(\mathcal{C}_{M_1}) \leq f(\mathcal{C}_{M_2}) \quad \textit{consistency with } f$$

Based on the above considerations, we will now state sufficient conditions for both non-locality (Lemma 5) and locality (Lemmas 6,8), thereby resolving locality for all our objective functions. We exemplarily state the short proof of Lemma 8, but defer all other proofs to the full version [9].

Lemma 5. *Let f be a clustering measure. If there exists a graph with two clusterings $\mathcal{C}$ and $\mathcal{D}$ both containing two merges M_1 and M_2 such that $f(\mathcal{C}_{M_1}) < f(\mathcal{C}_{M_2})$ and $f(\mathcal{D}_{M_1}) > f(\mathcal{D}_{M_2})$, then f is not local.*

For mpxd, apxd, mpxc, mpxe and gxd, Figure 5 shows examples where the preconditions of Lemma 5 hold, implying that these measures are not local. The rough idea behind the examples in Figs. 5a-5c is that a low pairwise intercluster quality between two clusters can be improved by merging one of the partners with a third cluster. Figure 5d exploits that merging large clusters becomes

---

[3] We do not need a proper total quasiorder as we never have to compare pairs of merges that cannot coexist in a clustering, e.g., because the clusters considered intersect.
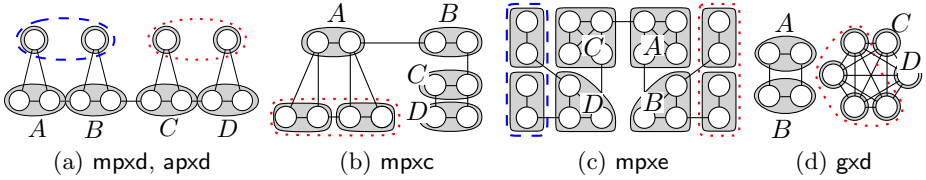
Fig. 5. Locality counterexamples: The base clustering consists of the gray clusters. In (a) and (c), if the blue, dashed merge is performed, merge $\{C, D\}$ is better, if the red, dotted merge is performed, $\{A, B\}$ is better. In (b) and (d), in the base clustering, $\{C, D\}$ is better than $\{A, B\}$, if the red, dotted merge is performed, the opposite holds.

more attractive if the global intercluster density is low. An instance proving the nonlocality of apxe and apxc is given in the full version [9].

Lemma 6. *Let f be an objective function such that $f(\mathcal{C})$ can be expressed as $f(\mathcal{C}) = \max\limits_{C \in \mathcal{C}} f'(C)$, with $f'(C)$ solely depending on C. Then, f is local.*

Proof sketch. Let $\mathcal{C}$ be a clustering and $\mathcal{D} = \{D_1, \ldots, D_k\} \subseteq \mathcal{C}$. Then we define $L(\mathcal{D}) := (f'(D_{i_1}) \geq \ldots \geq f'(D_{i_k}))$ to be the non-increasing sequence of function values of all $D_i \in \mathcal{D}$. Let $\leq_\ell$ be the lexicographical order on sequences of real numbers. The proof relies on the observation that for arbitrary merges $M_1 = \{A, B\}$ and $M_2 = \{C, D\} \subseteq \mathcal{C}$, $L(\mathcal{C}_{M_1}) \leq_\ell L(\mathcal{C}_{M_2})$ implies $f(\mathcal{C}_{M_1}) \leq f(\mathcal{C}_{M_2})$. We show that $L(\mathcal{C}_{M_1}) \leq_\ell L(\mathcal{C}_{M_2})$ is equivalent to $L(\{f'(A \cup B), f'(C), f'(D)\}) \leq_\ell L(\{f'(C \cup D), f'(A), f'(B)\})$ and that the latter relation satisfies locality.

Corollary 7. Mixd, mixc *and* mixe *are local.*

Lemma 8. *Let f be an objective function such that $f(\mathcal{C})$ can be expressed as $f(\mathcal{C}) = \frac{1}{|\mathcal{C}|} \sum_{C \in \mathcal{C}} f'(C)$, with $f'(C)$ solely depending on C. Then, f is local.*

Proof. Let $\mathcal{C}$ be an arbitrary clustering containing four clusters A, B, C and D. Then, $f(\mathcal{C}_{A,B}) \leq f(\mathcal{C}_{C,D})$ implies that

$$\underbrace{f'(A \cup B) - f'(C) - f'(D)}_{:=k_{A,B}} \leq \underbrace{f'(C \cup D) - f'(C) - f'(D)}_{:=k_{C,D}}$$

As for each cluster, f' is independent of the remainder of $\mathcal{C}$, this inequation shows that $\leq_f := \{(\{A, B\}, \{C, D\}) \mid k_{A,B} \leq k_{C,D}\}$ is a quasiorder on $\mathcal{M} \times \mathcal{M}$ such that $\{A, B\} \leq_f \{C, D\}$ implies $f(\mathcal{C}_{A,B}) \leq f(\mathcal{C}_{C,D})$. Thus, f is local. $\qquad\square$

Corollary 9. Aixd, aixc *and* aixe *are local.*

For nxe, it is easy to see that choosing $\leq_f$ such that $\{A, B\} \leq_f \{C, D\}$ is equivalent to $m_{A,B} \geq m_{C,D}$ satisfies the definition of locality. We have now proven or disproven the locality of all intercluster measures (for a summary see the full version [9]). Note that all proofs of locality are constructive in that they induce comparators which can be used to efficiently maintain the set of

all possible merges considered by Alg. 1 in a priority queue. For the maximum functions, triples of real numbers can be used as keys, compared as described in the proof of Lemma 6. Using average functions, it suffices to store the values $k_{A,B}$ defined in the proof of Lemma 8. All keys as well as the density of a new cluster can be computed and compared in constant time if, for any two clusters A and B, the values v_A, x_A, n_A and $m_{A,B}$ are maintained. Summarizing, we obtain the following corollary.

Corollary 10. *Algorithm 1 combining* mid *with* mixd*,* mixc*,* mixe*,* aixd*,* aixc*,* aixe *or* nxe *can be implemented with a running time in* $O(n^2 \log n)$*.*

Disconnected Merges. Whenever no single edge links two clusters, intuitively, merging them should not be beneficial to the clustering, or at least, such a merge should not be the best option. In the light of our bicriterial approach, an objective function which does encourage such a *disconnected merge* is naturally opposed by the separate mechanism of a constraint on the intracluster density. Superficially, this resolves the issue for non-degenerate instances; however, a more accurate assertion is algorithmically relevant: If we can rule out disconnected merges, it suffices to maintain only the set of *connected* merges in the heap (see Alg. 1), of which there are at most m (instead of $\Omega(n^2)$). This implies linear space and—given locality—$O(md \log n)$ time complexity, where d denotes the height of the dendrogram.[4] For sparse graphs ($m \in O(n)$), this bound can be an improvement, since d usually approaches $\log n$ (but never exceeds n). This analysis has initially been observed for Modularity [5], which enforces connected merges. It uses that, for each level of the dendrogram, only $O(m)$ heap entries are updated. In the following we resolve the question whether our objective functions enforce connected merges.

We say an objective function f *enforces connected merges* if for any pair of clusters $C \neq D \in \mathcal{C}$ with $m_{C,D} = 0$ and $f(\mathcal{C}) - f(\mathcal{C}_{\{C,D\}}) > 0$ (i.e., an improving, disconnected merge), there exist clusters $A \neq B \in \mathcal{C}$ such that $m_{A,B} > 0$ and $f(\mathcal{C}_{\{C,D\}}) > f(\mathcal{C}_{\{A,B\}})$ (i.e., a better, connected merge). In fact, only nxe and gxd enforce connected merges in general. Both measures never even benefit at all from disconnected merges: The former does not change, and the latter even deteriorates. The circumstances under which all other measures potentially encourage disconnected merges are intuitively illustrated in Fig. 6. If most clusters are reasonable, merging two clusters with a particularly ill contribution to the measure can be the best option (Fig. 6a). For all pairwise measures, this is immediate, and it is also not hard to see for all average measures, as, roughly speaking, the number of bad contributors decreases. While the above arguments fail for mixc, mixe and mixd, a disconnected merge of a bad cluster with a very good one can be the best option for them (see, e.g., Fig. 6b and [9]). Note that it is always possible to artificially restrict the set of allowed merges to connected ones, yielding a modified greedy algorithm. Evaluating the practical impact of such a restriction shall be subject to an experimental study; in our preliminary

---

[4] A dendrogram is a binary forest with singletons as leaves, and inner vertices representing the merge operations of an agglomerative process.

(a) apxc, apxe, apxd, aixc, aixe, aixd, mpxc,
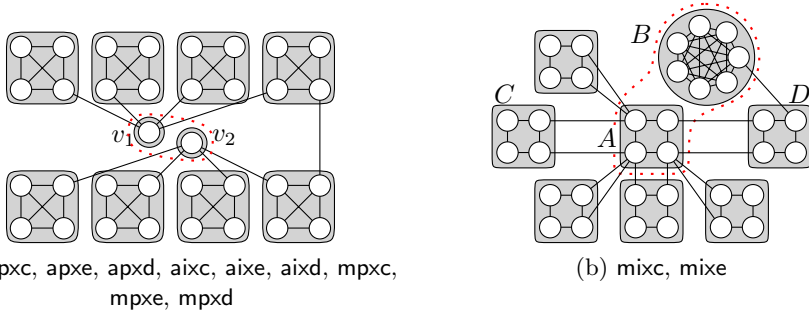mpxe, mpxd

(b) mixc, mixe

Fig. 6. Given the gray clusterings, disconnected merges (red, dotted) yield the highest improvement for the objective functions pointed out. Thus, all objective functions, except nxe and gxd, potentially favor disconnected merges (for mixd, see [9]).

experiments, we observed none. We summarize our positive observations in the following corollary, which can be extended to also apply to any local objective function if we restrict ourselves to connected merges.

Corollary 11. *Algorithm 1 combining* mid *with* nxe *can be implemented with a running time in $O(md \log n)$ and linear space complexity.*

4   Concluding Remarks

Established measures for graph cuts lend themselves well for precisely expressing desiderata on graph clusterings. Despite the scarce attention this approach has received from the graph clustering literature so far, existing studies did indicate its appropriateness. With a focus on finding graph clusterings that feature guaranteed intra- and high intercluster quality, we revived this ansatz and systematically formalized bicriteria quality measures based on expansion, conductance and sparsity. The classification of these measures with respect to their behavior in the context of greedy agglomeration yields conditions that render this widespread heuristic efficient, namely the locality and the connectedness of a measure, which we observed to coincide with common intuition about what is a good cut-based clustering measure. On top of that, we showed that a guaranteed density inside each cluster is especially suited to constrain agglomeration and that most definitions of intercluster quality do not suffer from local minima. We complemented our findings by exemplarily showing $\mathcal{NP}$-hardness for some variants of our problem statement. An experimental evaluation of density-constrained graph clustering and the adaption to local greedy optimization and to weighted graphs shall be subject to future work. We illustrate the outcome of greedy agglomeration combining guaranteed intracluster density with high average isolated intercluster conductance in Fig. 7.
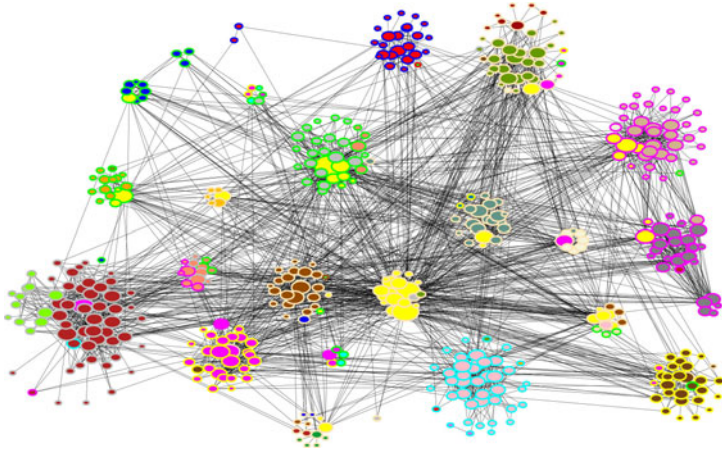
Fig. 7. This graph is a three-month snapshot of the email traffic at KIT's CS department, groups represent chairs, which serve as a ground truth (vertices are scaled by degree, $n = 472, m = 2845$). We ran Alg.1 using mid with $\alpha = 0.25$ and aixc to arrive at the color-clustering. Border colors indicate a Modularity-based clustering [5].

References

1. Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A.: Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties, 2nd edn. Springer, Heidelberg (2002)
2. Berkhin, P.: A Survey of Clustering Data Mining Techniques. In: Grouping Multi-dimensional Data: Recent Advances in Clustering, pp. 25–71. Springer, Heidelberg (2006)
3. Brandes, U., Gaertler, M., Wagner, D.: Engineering Graph Clustering: Models and Experimental Evaluation. ACM J. of Exp. Algorithmics 12(1.1), 1–26 (2007)
4. Chataigner, F., Manic, G., Wakabayashi, Y., Yuster, R.: Approximation algorithms and hardness results for the clique packing problem. Electronic Notes in Discrete Mathematics 29, 397–401 (2007)
5. Clauset, A., Newman, M.E.J., Moore, C.: Finding community structure in very large networks. Physical Review E 70(066111) (2004)
6. Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K.: Graph Clustering and Minimum Cut Trees. Internet Mathematics 1(4), 385–408 (2004)
7. Fortunato, S.: Community detection in graphs. Phys. Rep. 486(3-5), 75–174 (2009)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness. W. H. Freeman and Company, New York (1979)
9. Görke, R., Schumm, A., Wagner, D.: Density-Constrained Graph Clustering. Technical report, ITI Wagner, Department of Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe Reports in Informatics 2011-2017 (2011)
10. Jain, A.K., Dubes, R.C.: Algorithms for Clustering Data. Prentice Hall, Englewood Cliffs (1988)
11. Kannan, R., Vempala, S., Vetta, A.: On Clusterings - Good, Bad and Spectral. In: Proc. of FOCS 2000, pp. 367–378 (2000)
12. Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Physical Review E 69(026113) (2004)
13. Zachary, W.W.: An Information Flow Model for Conflict and Fission in Small Groups. Journal of Anthropological Research 33, 452–473 (1977)

The MST of Symmetric Disk Graphs (in Arbitrary Metric Spaces) is Light

Shay Solomon[⋆]

Department of Computer Science, Ben-Gurion University of the Negev,
POB 653, Beer-Sheva 84105, Israel
shayso@cs.bgu.ac.il

Abstract. Consider an n-point metric space $M = (V, \delta)$, and a transmission range assignment $r : V \rightarrow \mathbb{R}^+$ that maps each point $v \in V$ to the disk of radius $r(v)$ around it. The *symmetric disk graph* (henceforth, SDG) that corresponds to M and r is the undirected graph over V whose edge set includes an edge (u, v) if both $r(u)$ and $r(v)$ are no smaller than $\delta(u, v)$. SDGs are often used to model wireless communication networks.

Abu-Affash, Aschner, Carmi and Katz (SWAT 2010, [1]) showed that for any n-point *2-dimensional Euclidean* space M, the weight of the MST of every connected SDG for M is $O(\log n) \cdot w(MST(M))$, and that this bound is tight. However, the upper bound proof of [1] relies heavily on basic geometric properties of constant-dimensional Euclidean spaces, and does not extend to Euclidean spaces of super-constant dimension. A natural question that arises is whether this surprising upper bound of [1] can be generalized for wider families of metric spaces, such as high-dimensional Euclidean spaces.

In this paper we generalize the upper bound of Abu-Affash et al. [1] for Euclidean spaces of any dimension. Furthermore, our upper bound extends to *arbitrary metric spaces* and, in particular, it applies to any of the normed spaces ℓ_p. Specifically, we demonstrate that for *any* n-point metric space M, the weight of the MST of every connected SDG for M is $O(\log n) \cdot w(MST(M))$.

1 Introduction

1.1 The MST of Symmetric Disk Graphs

Consider a network that is represented as an (undirected) weighted graph $G = (V, E, w)$, and assume that we want to compute a spanning tree for G of *small weight*, i.e., of weight $w(G)$ that is close to the weight $w(MST(G))$ of the minimum spanning tree (MST) $MST(G)$ of G. (The weight of a graph $\mathcal{G}$, denoted $w(\mathcal{G})$, is defined as the sum of all edge weights in it.) However, due to some physical constraints (e.g., network faults) we are only given a connected spanning subgraph G' of G, rather than G itself. In this situation it is natural to use

---

the MST $MST(G')$ of the given subgraph G'. The *weight-coefficient* of G' with respect to G is defined as the ratio between $w(MST(G'))$ and $w(MST(G))$. If the weight-coefficient of G' is small enough, we can use $MST(G')$ as a spanning tree for G of small weight.

The problem of computing spanning trees of small weight (especially the MST) is a fundamental one in Computer Science [19,17,7,26,13,10], and the above scenario arises naturally in many practical contexts (see, e.g., [30,12,35,23,24,25,11]). In particular, this scenario is motivated by wireless network design.

In this paper we focus on the *symmetric disk graph model* in wireless communication networks, which has been subject to considerable research. (See [16,14,15,31,5,33,1,22], and the references therein.) Let $M = (V, \delta)$ be an n-point metric space that is represented as a complete weighted graph $G(M) = (V, \binom{V}{2}, w)$ in which the weight $w(e)$ of each edge $e = (u, v)$ is equal to $\delta(u, v)$. Also, let $r : V \to \mathbb{R}^+$ be a transmission range assignment that maps each point $v \in V$ to the disk of radius $r(v)$ around it. The *symmetric disk graph* (henceforth, SDG) that corresponds to M and r, denoted $SDG(M, r)$,[1] is the undirected spanning subgraph of $G(M)$ whose edge set includes an edge $e = (u, v)$ if both $r(u)$ and $r(v)$ are no smaller than $w(e)$. Under the symmetric disk graph model we cannot use all the edges of $G(M)$, but rather only those that are present in $SDG(M, r)$. Clearly, if $r(v) \geq diam(M)$,[2] for each point $v \in V$, then $SDG(M, r)$ is simply the complete graph $G(M)$. However, the transmission ranges are usually significantly shorter than $diam(M)$, and many edges that belong to $G(M)$ may not be present in $SDG(M, r)$. Therefore, it is generally impossible to use the MST of M under the symmetric disk graph model, simply because some of the edges of $MST(M)$ are not present in $SDG(M, r)$ and thus cannot be accessed. Instead, assuming the weight-coefficient of $SDG(M, r)$ with respect to M is small enough, we can use $MST(SDG(M, r))$ as a spanning tree for M of small weight.

Abu-Affash et al. [1] showed that for any n-point *2-dimensional Euclidean* space M, the weight of the MST of every connected SDG for M is $O(\log n) \cdot w(MST(M))$. In other words, they proved that for any n-point 2-dimensional Euclidean space, the weight-coefficient of every connected SDG is $O(\log n)$. In addition, Abu-Affash et al. [1] provided a matching lower bound of $\Omega(\log n)$ on the weight-coefficient of connected SDGs that applies to a basic 1-dimensional Euclidean space. Notably, the upper bound proof of [1] relies heavily on basic geometric properties of constant-dimensional Euclidean spaces, and does not extend to Euclidean spaces of super-constant dimension. A natural question that arises is whether this surprising upper bound of [1] on the weight-coefficient of

---

[1] The definition of symmetric disk graph can be generalized in the obvious way for any (undirected) weighted graph. Specifically, the *symmetric disk graph $SDG(G, r)$* that corresponds to a weighted graph $G = (V, E, w)$ and a transmission range assignment $r : V \to \mathbb{R}^+$ is the undirected spanning subgraph of G whose edge set includes an edge $e = (u, v) \in E$ if both $r(u)$ and $r(v)$ are no smaller than $w(e)$.

[2] The *diameter* of a metric space M, denoted $diam(M)$, is defined as the largest pairwise distance in M.

connected SDGs can be generalized for wider families of metric spaces, such as high-dimensional Euclidean spaces.

In this paper we generalize the upper bound of Abu-Affash et al. [1] for Euclidean spaces of any dimension. Furthermore, our upper bound extends to *arbitrary metric spaces* and, in particular, it applies to any of the normed spaces ℓ_p. Specifically, we demonstrate that for *any* n-point metric space M, every connected SDG has weight-coefficient $O(\log n)$. In fact, our upper bound is even more general, applying to disconnected SDGs as well. That is, we show that the weight of the minimum spanning forest[3] (MSF) of every (possibly disconnected) SDG for M is $O(\log n) \cdot w(MST(M))$.

1.2 The Range Assignment Problem

Given a network $G = (V, E, w)$, a *(transmission) range assignment* for G is an assignment of transmission ranges to each of the vertices of G. A range assignment is called *complete* if the induced (directed) communication graph is strongly connected. In the *range assignment problem* the objective is to find a complete range assignment for which the total power consumption (henceforth, cost) is minimized. The power consumed by a vertex $v \in V$ is $r(v)^\alpha$, where $r(v) > 0$ is the range assigned to v and $\alpha \geq 1$ is some constant. Thus the cost of the range assignment is given by $\sum_{v \in V} r(v)^\alpha$. The range assignment problem was first studied by Kirousis et al. [18], who proved that the problem is NP-hard in 3-dimensional Euclidean spaces, assuming $\alpha = 2$, and also presented a 2-approximation algorithm. Subsequently, Clementi et al. [9] proved that the problem remains NP-hard in 2-dimensional Euclidean spaces.

We believe that it is more realistic to study the range assignment problem under the symmetric disk graph model. Specifically, the potential transmission range of a vertex v is bounded by some maximum range $r'(v)$, and any two vertices u, v can directly communicate with each other if and only if v lies within the range assigned to u and vice versa. Blough et al. [3] showed that this version of the range assignment problem is also NP-hard in 2-dimensional and 3-dimensional Euclidean spaces. Also, Calinescu et al. [4] devised a $(1+\frac{1}{2}\ln 3+\epsilon)$-approximation scheme and a more practical $(\frac{15}{8})$-approximation algorithm. Abu-Affash et al. [1] showed that, assuming $\alpha = 1$, the cost of an optimal range assignment with bounds on the ranges is greater by at most a logarithmic factor than the cost of an optimal range assignment without such bounds. This result of Abu-Affash et al. [1] is a simple corollary of their upper bound on the weight-coefficient of SDGs for 2-dimensional Euclidean spaces. Consequently, this result of [1] for the range assignment problem holds only in 2-dimensional Euclidean spaces. By applying our generalized upper bound on the weight-coefficient of SDGs, we extend this result of Abu-Affash et al. [1] to arbitrary metric spaces.

---

[3] The *minimum spanning forest* of a (possibly disconnected) weighted graph G is the union of the MSTs for its connected components. In other words, it is the maximal cycle-free spanning subgraph of G of minimum weight.

1.3   Proof Overview

As was mentioned above, the upper bound proof of [1] is very specific, and relies heavily on basic geometric properties of constant-dimensional Euclidean spaces. Hence, it does not apply to Euclidean spaces of super-constant dimension, let alone to arbitrary metric spaces. Our upper bound proof is based on completely different principles. In particular, it is independent of the geometry of the metric space and applies to every complete graph whose weight function satisfies the triangle inequality. In fact, at the heart of our proof is a lemma that applies to an even wider family of graphs, namely, the family of all traceable[4] weighted graphs. Specifically, let S and H be an SDG and a minimum-weight Hamiltonian path of some traceable weighted n-vertex graph G, respectively, and let F be the MSF of S. Our lemma states that there is a set $\tilde{E}$ of edges in F of weight at most $w(H)$, such that the graph $F \setminus \tilde{E}$ obtained by removing all edges of $\tilde{E}$ from F contains at least $\frac{1}{5} \cdot n$ isolated vertices. The proof of this lemma is based on a delicate combinatorial argument that does not assume either that the graph G is complete or that its weight function satisfies the triangle inequality. We believe that this lemma is of independent interest. (See Lemma 1 in Sect. 2.) By employing this lemma inductively, we are able to show that the weight of F is bounded above by $\log_{\frac{5}{4}} n \cdot w(H)$, which, by the triangle inequality, yields an upper bound of $2 \cdot \log_{\frac{5}{4}} n$ on the weight-coefficient of S with respect to G. Interestingly, our upper bound of $2 \cdot \log_{\frac{5}{4}} n$ on the weight-coefficient of SDGs for arbitrary metric spaces improves the corresponding upper bound of Abu-Affash et al. [1] (namely, $90 \cdot \log_{\frac{5}{4}} n + 1$), which holds only in 2-dimensional Euclidean spaces, by a multiplicative factor of 45.

1.4   Related Work on Disk Graphs

The symmetric disk graph model is a generalization of the extremely well-studied *unit disk graph model* (see, e.g., [8,23,20,25,21]). The *unit disk graph* of a metric space M, denoted $UDG(M)$, is the symmetric disk graph corresponding to M and the range assignment $r \equiv 1$ that maps each point to the unit disk around it. (It is usually assumed that M is a 2-dimensional Euclidean space.) Observe that in the case when $UDG(M)$ is connected, all edges of $MST(M)$ belong to $UDG(M)$, and so $MST(UDG(M)) = MST(M)$. Hence the weight-coefficient of connected unit disk graphs for arbitrary metric spaces is equal to 1. In the general case, it is easy to see that all edges of $MSF(UDG(M))$ belong to $MST(M)$, and so the weight-coefficient of (possibly disconnected) unit disk graphs for arbitrary metric spaces is at most 1.

Another model that has received much attention in the literature is the *asymmetric disk graph model* (see, e.g., [20,32,28,29,1]). The *asymmetric disk graph* corresponding to a metric space $M = (V, \delta)$ and a range assignment $r : V \to \mathbb{R}^+$ is the directed graph over V, where there is an arc (u, v) of weight $\delta(u, v)$ from u to v if $r(u) \geq \delta(u, v)$. On the negative side, Abu-Affash et al. [1] provided a lower bound of $\Omega(n)$ on the weight-coefficient of strongly connected asymmetric

---

[4] A graph is called *traceable* if it contains a Hamiltonian path.

disk graphs that applies to an n-point 2-dimensional Euclidean space. However, asymmetric communication models are generally considered to be impractical, because in such models many communication primitives become unacceptably complicated [27,34]. In particular, the asymmetric disk graph model is often viewed as less realistic than the symmetric disk graph model, where, as was mentioned above, we obtain a logarithmic upper bound on the weight-coefficient for arbitrary metric spaces.

1.5   Structure of the Paper

In Sect. 2 we obtain a logarithmic upper bound on the weight-coefficient of SDGs for arbitrary metric spaces. An application of this upper bound to the range assignment problem is given in Sect. 3.

1.6   Preliminaries

Given a (possibly weighted) graph G, its vertex set (respectively, edge set) is denoted by $V(G)$ (resp., $E(G)$). For an edge set $E' \subseteq E(G)$, we denote by $G \setminus E'$ the graph obtained by removing all edges of E' from G. Also, for an edge set E'' over the vertex set $V(G)$, we denote by $G \cup E''$ the graph obtained by adding all edges of E'' to G. The weight of an edge e in G is denoted by $w(e)$. For an edge set $E \subseteq E(G)$, its weight $w(E)$ is defined as the sum of all edge weights in it, i.e., $w(E) = \sum_{e \in E} w(e)$. The weight of G is defined as the weight of its edge set $E(G)$, namely, $w(G) = w(E(G))$. Finally, for a positive integer n, we denote the set $\{1, 2, \ldots, n\}$ by $[n]$.

2   The MST of SDGs is Light

In this section we prove that the weight-coefficient of SDGs for arbitrary n-point metric spaces is $O(\log n)$.

We will use the following well-known fact in the sequel.

Fact 1. *Let G be a weighted graph in which ell edge weights are distinct. Then G has a unique MSF, and the edge of maximum weight in every cycle of G does not belong to the MSF of G.*

In what follows we assume for simplicity that all the distances in any metric space are distinct. This assumption does not lose generality, since any ties can be broken using, e.g., lexicographic rules. Given this assumption, Fact 1 implies that there is a unique MST for any metric space, and a unique MSF for every SDG of any metric space.

The following lemma is central in our upper bound proof.

Lemma 1. *Let $M = (V, \delta)$ be an n-point metric space and let $r : V \to \mathbb{R}^+$ be a range assignment. Also, let $F = (V, E_F)$ be the MSF of the symmetric disk graph $S = SDG(M, r)$ and let $H = (V, E_H)$ be a minimum-weight Hamiltonian path of M. Then there is an edge set $\tilde{E} \subseteq E_F$ of weight at most $w(H)$, such that the graph $F \setminus \tilde{E}$ contains at least $\frac{1}{5} \cdot n$ isolated vertices.*

Remark: This statement remains valid if instead of the metric space M we take an arbitrary traceable weighted graph.

Proof. First, we construct a bijection $f : E \to \tilde{E}$, where $E \subseteq E_H$ and $\tilde{E} \subseteq E_F$, that satisfies that $w(f(e)) \leq w(e)$, for each edge $e \in E$. This would imply that $w(\tilde{E}) \leq w(E) \leq w(H)$. We then show that the graph $F \setminus \tilde{E}$ contains at least $\frac{1}{5} \cdot n$ isolated vertices, which concludes the proof of the lemma.

The edge set E (respectively, $\tilde{E}$) is defined as the union of three disjoint edge sets to be specified later, denoted E'_1, E'_2 and E''_3 (resp., $\tilde{E}_1, \tilde{E}_2$ and $\tilde{E}_3$); thus $E = E'_1 \cup E'_2 \cup E''_3$ and $\tilde{E} = \tilde{E}_1 \cup \tilde{E}_2 \cup \tilde{E}_3$. We will construct three bijections $f_1 : E'_1 \to \tilde{E}_1$, $f_2 : E'_2 \to \tilde{E}_2$ and $f_3 : E''_3 \to \tilde{E}_3$. The bijection f will be obtained as the extension of these functions to the domain E, that is, for an edge $e \in E$,

$$
f(e) = \begin{cases} f_1(e), & \text{if } e \in E'_1; \\ f_2(e), & \text{if } e \in E'_2; \\ f_3(e), & \text{if } e \in E''_3. \end{cases}
$$

In other words, the function f_1 (respectively, f_2; resp., f_3) defines the restriction of the function f to the domain E'_1 (resp., E'_2; resp., E''_3).

Denote by E' the set of all edges in H that belong to the SDG S, i.e., $E' = E_H \cap E(S)$, and let $E'' = E_H \setminus E'$ be the complementary edge set of E' in E_H. We define E'_1 as the set of all edges in E' that belong to the MSF F, i.e., $E'_1 = E' \cap E_F$, and $E'_2 = E' \setminus E'_1$ as the complementary edge set of E'_1 in E'. Note that (1) $E' \subseteq E(S)$, (2) $E'' \cap E(S) = \emptyset$, (3) $E'_1 \subseteq E_F$, and (4) $E'_2 \cap E_F = \emptyset$. Also, observe that, by definition, the edge set E contains the entire edge set $E' = E'_1 \cup E'_2$ and only a subset E''_3 of E''; thus $E = E'_1 \cup E'_2 \cup E''_3 \subseteq E' \cup E'' = E_H$.

The function f_1 is defined as the identity map, namely, for each edge $e \in E'_1$, we define $f_1(e) = e$. Also, define $\tilde{E}_1 = E'_1$. Observe that $\tilde{E}_1 \subseteq E_F$, and f_1 is a bijection from E'_1 to $\tilde{E}_1$.

We proceed with constructing the function f_2.

Write $k = |E'_2|$, and let $e'_1, e'_2, \ldots, e'_k$ denote the edges of E'_2 by increasing order of weight. Next, we compute $k = |E'_2|$ spanning forests $F_1, F_2, \ldots, F_k$ of S, where each forest F_i contains a unique edge $\tilde{e}_i$ in $E_F \setminus E_H$ that satisfies that $w(\tilde{e}_i) \leq w(e'_i)$; thus we can define $f_2(e'_i) = \tilde{e}_i$. The first forest F_1 is simply a copy of F. The rest of the forests $F_2, F_3, \ldots, F_k$ are computed iteratively as follows. For each index $i = 1, 2, \ldots, k$, the graph $F_i \cup \{e'_i\}$ obtained from F_i by adding to it the edge e'_i contains a unique cycle C_i. Since H is cycle-free, at least one edge of C_i does not belong to H; take $\tilde{e}_i$ to be an arbitrary such edge and define $f_2(e'_i) = \tilde{e}_i$. Finally, denote by $F_{i+1} = F_i \cup \{e'_i\} \setminus \{f_2(e'_i)\}$ the graph obtained from F_i by adding to it the edge e'_i and removing the edge $f_2(e'_i)$, for each $i \in [k-1]$. Define $\tilde{E}_2 = \{f_2(e'_i) \mid i \in [k]\}$. Observe that f_2 is a bijection from E'_2 to $\tilde{E}_2$.

Claim. (1) $\tilde{E}_2 \subseteq E_F \setminus \tilde{E}_1$. (2) For each index $i \in [k]$, $w(f_2(e'_i)) \leq w(e'_i)$.

Proof. Fix an arbitrary index $i \in [k]$, and define $E'_{(i)} = \{e'_1, \ldots, e'_i\}$.

Note that the cycle C_i that is identified during the ith iteration of the above process is a subgraph of S. Moreover, $E(C_i) \subseteq E_F \cup E'_2$. Since $E'_2 \subseteq E_H$ and

$f_2(e_i')$ is an edge of C_i that does not belong to H, it follows that $f_2(e_i') \in E_F \setminus E_H$. This argument holds for any index $i \in [k]$, and so $\tilde{E}_2 = \{f_2(e_i') \mid i \in [k]\} \subseteq E_F \setminus E_H \subseteq E_F \setminus \tilde{E}_1$. (The last inequality follows from the fact that $\tilde{E}_1 = E_1' \subseteq E_H$.)

To prove the second assertion of the claim, notice that each edge of C_i that do not belong to F must belong to $E_{(i)}'$, i.e., $E(C_i) \setminus E_F \subseteq E_{(i)}'$. Fact 1 implies that the edge of maximum weight in C_i, denoted e_i^*, does not belong to F, hence $e_i^* \in E_{(i)}'$. Since e_i' is the edge of maximum weight in $E_{(i)}'$, it follows that $w(e_i^*) \leq w(e_i')$. Also, as $f_2(e_i')$ belongs to C_i, we have by definition $w(f_2(e_i')) \leq w(e_i^*)$. Consequently, $w(f_2(e_i')) \leq w(e_i^*) \leq w(e_i')$, and we are done.   □

Next, we construct the function f_3.

Denote by $H'' = H \setminus (E_1' \cup E_2')$ and $F'' = F \setminus (\tilde{E}_1 \cup \tilde{E}_2)$ the graphs obtained from H and F by removing all edges of $E' = E_1' \cup E_2'$ and $\tilde{E}_1 \cup \tilde{E}_2$, respectively. By definition, $E(H'') = E''$. For an edge $e = (u, v)$, denote by $min(e)$ the endpoint of e with smaller radius, i.e., $min(e) = u$ if $r(u) < r(v)$, and $min(e) = v$ otherwise. The construction of the function f_3 is done in parallel to the computation of its domain E_3''; recall that E_3'' is the set of all edges in E'' that belong to E.

We start with initializing $E_3'' = \emptyset$. Then we examine the edges of E'' one after another in an arbitrary order. For each edge $e'' \in E''$, we check whether the vertex $min(e'')$ is isolated in F'' or not. If $min(e'')$ is isolated in F'', we leave H'', F'' and E_3'' intact. Otherwise, at least one edge is incident to $min(e'')$ in F''. Let $\tilde{e}$ be an arbitrary such edge, and define $f_3(e'') = \tilde{e}$. We remove the edge e'' from the graph H'' and add it to the edge set E_3'', and also remove the edge $f_3(e'')$ from the graph F''. This process is repeated iteratively until all edges of E'' have been examined. Define $\tilde{E}_3 = \{f_3(e'') \mid e'' \in E_3''\}$. At the end of this process, it holds that $H'' = H \setminus E = H \setminus (E_1' \cup E_2' \cup E_3'')$ and $F'' = F \setminus \tilde{E} = F \setminus (\tilde{E}_1 \cup \tilde{E}_2 \cup \tilde{E}_3)$. Observe that $\tilde{E}_3 \subseteq E_F \setminus (\tilde{E}_1 \cup \tilde{E}_2)$, and f_3 is a bijection from E_3'' to $\tilde{E}_3$.

Claim. For each edge $e'' \in E_3''$, $w(f_3(e'')) \leq w(e'')$.

Proof. Consider an arbitrary edge $e'' \in E_3''$ and the graph F'' just before the edge e'' was examined. Since no edge of E_3'' belongs to the SDG S, we have by definition $r(min(e'')) < w(e'')$. Also, since the graph F'' is a subgraph of S, the weight of every edge that is incident to $min(e'')$ in F'', including $f_3(e'') = \tilde{e}$, is no greater than $r(min(e''))$. Hence, $w(f_3(e'')) \leq r(min(e'')) < w(e'')$.   □

We showed that the functions $f_1 : E_1' \to \tilde{E}_1$, $f_2 : E_2' \to \tilde{E}_2$ and $f_3 : E_3'' \to \tilde{E}_3$ are bijective, and that for each edge $e \in E_1'$ (respectively, $e \in E_2'$; resp., $e \in E_3''$), it holds that $w(f_1(e)) \leq w(e)$ (resp., $w(f_2(e)) \leq w(e)$; resp., $w(f_3(e)) \leq w(e)$). Furthermore, the domains E_1', E_2', E_3'' (respectively, images $\tilde{E}_1, \tilde{E}_2, \tilde{E}_3$) of these functions are pairwise disjoint subsets of E_H (resp., E_F). Hence, the extension f of these functions to the domain E is a bijection from $E = E_1' \cup E_2' \cup E_3'' \subseteq E_H$ to $\tilde{E} = \tilde{E}_1 \cup \tilde{E}_2 \cup \tilde{E}_3 \subseteq E_F$, such that $w(f(e)) \leq w(e)$, for each edge $e \in E$. It follows that

$$w(\tilde{E}) = \sum_{e \in \tilde{E}} w(e) = \sum_{e \in E} w(f(e)) \leq \sum_{e \in E} w(e) = w(E) \leq w(H).$$

To complete the proof of Lemma 1, we show that the graph $F'' = F \setminus \tilde{E}$ contains at least $\frac{1}{5} \cdot n$ isolated vertices. Denote by m_H (respectively, m_F) the number $|E(H'')|$ (resp., $|E(F'')|$) of edges in the graph H'' (resp., F'').

Suppose first that $m_F < \frac{2}{5} \cdot n$. Observe that in any n-vertex graph with m edges there are at least $n - 2m$ isolated vertices. Thus, the number of isolated vertices in F'' is bounded below by $n - 2m_F > n - \frac{4}{5} \cdot n = \frac{1}{5} \cdot n$, as required.

We henceforth assume that $m_F \geq \frac{2}{5} \cdot n$.

Since $H'' = H \setminus E$ and $E \subseteq E_H$, it holds that $|E(H'')| = |E_H| - |E|$. Similarly, we get that $|E(F'')| = |E_F| - |\tilde{E}|$. Also, observe that $|E_H| = n - 1 \geq |E_F|$ and $|E| = |\tilde{E}|$. Therefore,

$$m_H = |E(H'')| = |E_H| - |E| \geq |E_F| - |\tilde{E}| = |E(F'')| = m_F. \quad (1)$$

Let $\mathcal{M}''$ be a maximal independent edge set (i.e., a maximal set of pairwise non-adjacent edges) in H''. Since H'' is a subgraph of the Hamiltonian path H, we conclude that at least half of the edges of H'' must belong to $\mathcal{M}''$. Consequently,

$$|\mathcal{M}''| \geq \frac{1}{2} \cdot |E(H'')| = \frac{1}{2} \cdot m_H \geq \frac{1}{2} \cdot m_F \geq \frac{1}{5} \cdot n.$$

(The second inequality follows from (1) whereas the third inequality follows from the above assumption.) By definition, for any pair e, e' of edges in $\mathcal{M}''$, $min(e) \neq min(e')$, hence the size of the vertex set $\mathcal{I}'' = \{min(e'') \mid e'' \in \mathcal{M}''\}$ satisfies $|\mathcal{I}''| = |\mathcal{M}''| \geq \frac{1}{5} \cdot n$. By construction, for each edge e'' in H'', the vertex $min(e'')$ is isolated in F''. In particular, all the vertices of $\mathcal{I}''$ are isolated in F''. Thus, the number of isolated vertices in F'' is bounded below by $|\mathcal{I}''| \geq \frac{1}{5} \cdot n$.

Lemma 1 follows.   □

Next, we employ Lemma 1 inductively to upper bound the weight of SDGs in terms of the weight of a minimum-weight Hamiltonian path of the metric space. The desired upper bound of $O(\log n)$ on the weight-coefficient of SDGs for arbitrary n-point metric spaces would immediately follow.

Lemma 2. *Let $M = (V, \delta)$ be an n-point metric space and let $r : V \to \mathbb{R}^+$ be a range assignment. Also, let $F = (V, E_F)$ be the MSF of the symmetric disk graph $S = SDG(M, r)$ and let $H = (V, E_H)$ be a minimum-weight Hamiltonian path of M. Then $w(F) \leq \log_{\frac{5}{4}} n \cdot w(H)$.*

Proof. The proof is by induction on the number n of points in V.

Basis: $n \leq 4$. The case $n = 1$ is trivial. Suppose next that $2 \leq n \leq 4$. In this case $\log_{\frac{5}{4}} n \geq \log_{\frac{5}{4}} 2 > 3$. Also, the MSF F of S contains at most 3 edges. By the triangle inequality, the weight of each edge of F is bounded above by the weight $w(H)$ of the Hamiltonian path H. Hence, $w(F) \leq 3 \cdot w(H) < \log_{\frac{5}{4}} n \cdot w(H)$.

Induction step: We assume that the statement holds for all smaller values of n, $n \geq 5$, and prove it for n. By Lemma 1, there is an edge set $\tilde{E} \subseteq E_F$ of weight at most $w(H)$, such that the set I of isolated vertices in the graph $F \setminus \tilde{E}$ satisfies $|I| \geq \frac{1}{5} \cdot n$. Consider the complementary edge set $\hat{E} = E_F \setminus \tilde{E}$ of edges in F. Observe that no edge of $\hat{E}$ is incident to a vertex of I. Let $\hat{M}$ be the sub-metric

of M induced by the point set of $\hat{V} = V \setminus I$, and let $\hat{r}$ be the restriction of the range assignment r to $\hat{V}$. Also, let $\hat{S} = SDG(\hat{M}, \hat{r})$ be the SDG corresponding to $\hat{M}$ and $\hat{r}$, and let $\hat{F} = (\hat{V}, E_{\hat{F}})$ be the MSF of $\hat{S}$. Notice that the induced subgraph of S over the vertex set $\hat{V}$ is equal to $\hat{S}$, implying that all edges of $\hat{E}$ belong to $\hat{S}$. Thus, since $\hat{F}$ is a spanning forest of $\hat{S}$, replacing the edge set $\hat{E}$ of F by the edge set $E_{\hat{F}}$ does not affect the connectivity of the graph, i.e., the graph $\bar{F} = F \setminus \hat{E} \cup E_{\hat{F}}$ that is obtained from F by removing the edge set $\hat{E}$ and adding the edge set $E_{\hat{F}}$ has exactly the same connected components as F. Consequently, by breaking all cycles in the graph $\bar{F}$, we get a spanning forest of S. The weight of this spanning forest is bounded above by the weight $w(\bar{F}) = w\left(F \setminus \hat{E} \cup E_{\hat{F}}\right)$ of $\bar{F}$, and is bounded below by the weight $w(F)$ of the MSF F of S. Hence $w(F) \leq w\left(F \setminus \hat{E} \cup E_{\hat{F}}\right)$, which implies that $w(\hat{E}) \leq w(E_{\hat{F}}) = w(\hat{F})$. Write $\hat{n} = |\hat{V}|$, and let $\hat{H} = (\hat{V}, E_{\hat{H}})$ be a minimum-weight Hamiltonian path of $\hat{M}$. Since $|I| \geq \frac{1}{5} \cdot n$, we have

$$\hat{n} = |\hat{V}| = |V \setminus I| \leq \frac{4}{5} \cdot n \leq n - 1.$$

(The last inequality holds for $n \geq 5$.) By the induction hypothesis for $\hat{n}$, $w(\hat{F}) \leq \log_{\frac{5}{4}} \hat{n} \cdot w(\hat{H})$. Also, the triangle inequality implies that $w(\hat{H}) \leq w(H)$. Hence,

$$\begin{aligned} w(\hat{E}) &\leq w(\hat{F}) \leq \log_{\frac{5}{4}} \hat{n} \cdot w(\hat{H}) \leq \log_{\frac{5}{4}} \left(\frac{4}{5} \cdot n\right) \cdot w(H) \\ &= \log_{\frac{5}{4}} n \cdot w(H) - w(H). \end{aligned}$$

We conclude that

$$\begin{aligned} w(F) &= w(E_F) = w(\tilde{E}) + w(E_F \setminus \tilde{E}) = w(\tilde{E}) + w(\hat{E}) \\ &\leq w(H) + \log_{\frac{5}{4}} n \cdot w(H) - w(H) = \log_{\frac{5}{4}} n \cdot w(H). \qquad \square \end{aligned}$$

By the triangle inequality, the weight of a minimum-weight Hamiltonian path of any metric space is at most twice greater than the weight of the MST of that metric. We derive the main result of this paper as a corollary of Lemma 2.

Theorem 1. *For any n-point metric space $M = (V, \delta)$ and any range assignment $r : V \to \mathbb{R}^+$, $w(MSF(SDG(M, r))) = O(\log n) \cdot w(MST(M))$.*

3   The Range Assignment Problem

In this section we demonstrate that for any metric space, the cost of an optimal range assignment with bounds on the ranges is greater by at most a logarithmic factor than the cost of an optimal range assignment without such bounds. This result follows as a simple corollary of the upper bound given in Theorem 1.

Let $M = (V, \delta)$ be an n-point metric space, and assume that the n points of V, denoted by $v_1, v_2, \ldots, v_n$, represent transceivers. Also, let $r' : V \to \mathbb{R}^+$ be

a *bounding range assignment* for V, i.e., a function that provides a maximum transmission range for each of the points of V, such that the SDG $SDG(M, r')$ corresponding to M and r' is connected. In the *bounded range assignment problem* the objective is to compute a range assignment $r : V \rightarrow \mathbb{R}^+$, such that (i) for each point $v_i \in V$, $r(v_i) \leq r'(v_i)$, (ii) the induced SDG (using the ranges $r(v_1), r(v_2), \ldots, r(v_n)$), namely $SDG(M, r)$, is connected, and (iii) $\sum_{i=1}^{n} r(v_i)$ is minimized. The sum $\sum_{i=1}^{n} r(v_i)$ is called the *cost* of the range assignment r. In the *unbounded range assignment problem* the maximum transmission range for each of the points of V is unbounded; that is, the unbounded range assignment problem is a special case of the bounded range assignment problem, where the bounding range assignment r' satisfies $r'(v_i) = diam(M)$, for each point $v_i \in V$.

Fix an arbitrary bounding range assignment $r' : V \rightarrow \mathbb{R}^+$. Denote by $OPT(M, r')$ the cost of an optimal solution for the bounded range assignment problem corresponding to M and r'. Also, denote by $OPT(M)$ the cost of an optimal solution for the unbounded range assignment problem corresponding to M. Notice that $OPT(M) \leq OPT(M, r')$. Next, we show that $OPT(M, r') = O(\log n) \cdot OPT(M)$.

Let $SDG(M, r')$ be the SDG corresponding to M and r', and let T be the MST of $SDG(M, r')$. We define r to be the range assignment that assigns $r(v_i)$ with the weight of the heaviest edge incident to v_i in T, for each point $v_i \in V$. By construction, $r(v_i) \leq r'(v_i)$, for each point $v_i \in V$. Also, notice that the SDG corresponding to M and r, namely $SDG(M, r)$, contains T and in thus connected. Hence, the range assignment r provides a feasible solution for the bounded range assignment problem corresponding to M and r', yielding $OPT(M, r') \leq \sum_{i=1}^{n} r(v_i)$. By a double counting argument, we get that $\sum_{i=1}^{n} r(v_i) \leq 2 \cdot w(T)$. Also, by Theorem 1, $w(T) = w(MST(SDG(M, r'))) = O(\log n) \cdot w(MST(M))$. Finally, it is easy to verify that $w(MST(M)) \leq OPT(M)$. Altogether,

$$OPT(M, r') \leq \sum_{i=1}^{n} r(v_i) \leq 2 \cdot w(T) = 2 \cdot w(MST(SDG(M, r')))$$
$$= O(\log n) \cdot w(MST(M)) = O(\log n) \cdot OPT(M).$$

Theorem 2. *For any n-point metric space $M = (V, \delta)$ and any bounding range assignment $r' : V \rightarrow \mathbb{R}^+$, $OPT(M, r') = O(\log n) \cdot OPT(M)$.*

Acknowledgments

References

1. Abu-Affash, A.K., Aschner, R., Carmi, P., Katz, M.J.: The MST of Symmetric Disk Graphs Is Light. In: Kaplan, H. (ed.) SWAT 2010. LNCS, vol. 6139, pp. 236–247. Springer, Heidelberg (2010)

2. Althöfer, I., Das, G., Dobkin, D.P., Joseph, D., Soares, J.: On sparse spanners of weighted graphs. Discrete & Computational Geometry 9, 81–100 (1993)
3. Blough, D.M., Leoncini, M., Resta, G., Santi, P.: On the symmetric range assignment problem in wireless ad hoc networks. In: Proc. of the IFIP 17th World Computer Congress TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS), pp. 71–82 (2002)
4. Calinescu, G., Mandoiu, I.I., Zelikovsky, A.: Symmetric connectivity with minimum power consumption in radio networks. In: Proc. of the IFIP 17th World Computer Congress TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science (TCS), pp. 119–130 (2002)
5. Caragiannis, I., Fishkin, A.V., Kaklamanis, C., Papaioannou, E.: A tight bound for online colouring of disk graphs. Theor. Comput. Sci. 384(2-3), 152–160 (2007)
6. Chandra, B., Das, G., Narasimhan, G., Soares, J.: New sparseness results on graph spanners. Int. J. Comput. Geometry Appl. 5, 125–144 (1995)
7. Chazelle, B.: A minimum spanning tree algorithm with inverse-Ackermann type complexity. J. ACM 47(6), 1028–1047 (2000)
8. Clark, B.N., Colbourn, C.J., Johnson, D.S.: Unit disk graphs. Discrete Mathematics 86(1-3), 165–177 (1990)
9. Clementi, A.E.F., Penna, P., Silvestri, R.: Hardness results for the power range assignment problem in packet radio networks. In: Hochbaum, D.S., Jansen, K., Rolim, J.D.P., Sinclair, A. (eds.) RANDOM 1999 and APPROX 1999. LNCS, vol. 1671, pp. 197–208. Springer, Heidelberg (1999)
10. Czumaj, A., Sohler, C.: Estimating the Weight of Metric Minimum Spanning Trees in Sublinear Time. SIAM J. Comput. 39(3), 904–922 (2009)
11. Damian, M., Pandit, S., Pemmaraju, S.V.: Distributed Spanner Construction in Doubling Metric Spaces. In: Shvartsman, A.A. (ed.) OPODIS 2006. LNCS, vol. 4305, pp. 157–171. Springer, Heidelberg (2006)
12. Das, S.K., Ferragina, P.: An o(n) Work EREW Parallel Algorithm for Updating MST. In: van Leeuwen, J. (ed.) ESA 1994. LNCS, vol. 855, pp. 331–342. Springer, Heidelberg (1994)
13. Elkin, M.: An Unconditional Lower Bound on the Time-Approximation Trade-off for the Distributed Minimum Spanning Tree Problem. SIAM J. Comput. 36(2), 433–456 (2006)
14. Erlebach, T., Jansen, K., Seidel, E.: Polynomial-time approximation schemes for geometric graphs. In: Proc. of 12th SODA, pp. 671–679 (2001)
15. Fiala, J., Fishkin, A.V., Fomin, F.V.: On distance constrained labeling of disk graphs. Theor. Comput. Sci. 326(1-3), 261–292 (2004)
16. Hliněný, P., Kratochvíl, J.: Representing graphs by disks and balls. Discrete Mathematics 229(1-3), 101–124 (2001)
17. Karger, D.R., Klein, P.N., Tarjan, R.E.: A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. J. ACM 42(2), 321–328 (1995)
18. Kirousis, L., Kranakis, E., Krizanc, D., Pelc, A.: Power consumption in packet radio networks. Theoretical Computer Science 243(1-2), 289–305 (2000)
19. Khuller, S., Raghavachari, B., Young, N.E.: Low degree spanning trees of small weight. In: Proc. of 26th STOC, pp. 412–421 (1994)
20. Kumar, V.S.A., Marathe, M.V., Parthasarathy, S., Srinivasan, A.: End-to-end packet-scheduling in wireless ad-hoc networks. In: Proc. of 15th SODA, pp. 1021–1030 (2004)
21. van Leeuwen, E.J.: Approximation Algorithms for Unit Disk Graphs. In: Kratsch, D. (ed.) WG 2005. LNCS, vol. 3787, pp. 351–361. Springer, Heidelberg (2005)

22. van Leeuwen, E.J., van Leeuwen, J.: On the Representation of Disk Graphs. Technical report UU-CS-2006-037, Utrecht University (2006)
23. Li, X.-Y.: Approximate MST for UDG Locally. In: Warnow, T., Zhu, B. (eds.) COCOON 2003. LNCS, vol. 2697, pp. 364–373. Springer, Heidelberg (2003)
24. Li, X.-Y., Wang, Y., Wan, P.-J., Frieder, O.: Localized Low Weight Graph and Its Applications in Wireless Ad Hoc Networks. In: Proc. of 23rd INFOCOM (2004)
25. Li, X.-Y., Wang, Y., Song, W.-Z.: Applications of k-Local MST for Topology Control and Broadcasting in Wireless Ad Hoc Networks. IEEE Trans. Parallel Distrib. Syst. 15(12), 1057–1069 (2004)
26. Pettie, P., Ramachandran, V.: An optimal minimum spanning tree algorithm. J. ACM 49(1), 16–34 (2002)
27. Prakash, R.: Unidirectional links prove costly in wireless ad hoc networks. In: Proc. of 3rd DIAL-M, pp. 15–22 (1999)
28. Peleg, D., Roditty, L.: Localized spanner construction for ad hoc networks with variable transmission range. ACM Trans. on Sensor Net. 7(3), Article 25 (2010)
29. Peleg, D., Roditty, L.: Relaxed Spanners for Directed Disk Graphs. In: Proc. of 27th STACS, pp. 609–620 (2010)
30. Salowe, J.S.: Construction of Multidimensional Spanner Graphs, with Applications to Minimum Spanning Trees. In: Proc. of 7th SoCG, pp. 256–261 (1991)
31. Thai, M.T., Du, D.-Z.: Connected Dominating Sets in Disk Graphs with Bidirectional Links. IEEE Communications Letters 10(3), 138–140 (2006)
32. Thai, M.T., Tiwari, R., Du, D.-Z.: On Construction of Virtual Backbone in Wireless Ad Hoc Networks with Unidirectional Links. IEEE Trans. Mob. Comput. 7(9), 1098–1109 (2008)
33. Thai, M.T., Wang, F., Liu, D., Zhu, S., Du, D.-Z.: Connected Dominating Sets in Wireless Networks with Different Transmission Ranges. IEEE Trans. Mob. Comput. 6(7), 721–730 (2007)
34. Wattenhofer, R.: Algorithms for ad hoc and sensor networks. Computer Communications 28(13), 1498–1504 (2005)
35. Zhou, H., Shenoy, N.V., Nicholls, W.: Efficient minimum spanning tree construction without Delaunay triangulation. Inf. Process. Lett. 81(5), 271–276 (2002)

Theory vs. Practice in the Design and Analysis of Algorithms

Robert E. Tarjan

Department of Computer Science, Princeton University and HP Labs

Abstract. In this talk I'll explore gaps between the theoretical study of algorithms and the use of algorithms in practice. Examples will be drawn from my own experiences in industry and academia, and will include data structures and network algorithms. Based on these examples I'll try to draw conclusions to help guide the work of theoreticians and experimentalists, in an effort to make this work more relevant to the needs of practitioners.

A Fully Polynomial Approximation Scheme for a Knapsack Problem with a Minimum Filling Constraint⋆
(Extended Abstract)

Zhou Xu⋆⋆ and Xiaofan Lai

Dept. of Logistics and Maritime Studies,
Faculty of Business, The Hong Kong Polytechnic University
{lgtzx,10901006r}@polyu.edu.hk

Abstract. We study a variant of the knapsack problem, where a minimum filling constraint is imposed such that the total weight of selected items cannot be less than a given threshold. We consider the case when the ratio of the threshold to the capacity equals a given constant α with $0 \leq \alpha < 1$. For any such constant α, since finding an optimal solution is NP-hard, we develop the first FPTAS for the problem, which has a time complexity polynomial in $1/(1 - \alpha)$.

Keywords: approximation algorithm, FPTAS, knapsack problem, minimum filling constraint.

1 Introduction

The knapsack problem is a well-studied combinatorial optimization problem. Given a capacity c, and given a set $J = \{1, 2, ..., n\}$ of n items, where each item has a weight w_j and a profit p_j, where $0 < w_j \leq c$ and $p_j \geq 0$, the problem is to select a subset of J such that the total profit of selected items is maximized and the total weight does not exceed the capacity c. The knapsack problem is NP-hard, and has a fully polynomial approximation scheme (FPTAS) which can deliver a feasible solution with a total profit not less than $(1 - \epsilon)$ times the total profit of an optimal solution [9], and with a time complexity polynomial in n and $1/\epsilon$, for any $0 < \epsilon \leq 1$.

We study a variant of the knapsack problem, where a minimum filling constraint is imposed, such that the total weight of selected items cannot be less than a given threshold d where $0 \leq d \leq c$. We call this the knapsack problem with a minimum filling constraint (KPMFC). Using binary decision variables x_j for $j \in J$, the problem can be formulated as an integer programming model.

$$\max \sum_{j \in J} p_j x_j$$

$$\text{s.t. } d \le \sum_{j \in J} w_j x_j \le c,$$

$$x_j \in \{0, 1\}, \text{ for } j \in J.$$

If the problem has a feasible solution, let $\{x_j^* : j \in J\}$ denote an optimal solution.

Problem KPMFC has several applications in business. For instance when a shipping company operates a ship with a capacity, and makes decisions on acceptance or rejection of cargos for sailing, the minimum filling constraint is often imposed, to avoid the total weight of cargos accepted being too light to cause an unbalancing problem of the ship. Another typical application occurs when a buyer with a budget limit (denoted by c) selects items to purchase from a seller. The seller often offers rewards, such as free delivery or coupons, to the buyer if the total value of the purchased items exceeds a threshold, or in other words, satisfies a minimum filling constraint with w_j representing the unit price of item j. This type of reward schemes is a common practice in buying services from logistics service providers, and buying products from online retailers in e-Commerce [5]. With p_j denoting her perceived value of item j, the buyer aims to maximize the total perceived values of items purchased and of rewards granted.

Finding an optimal solution to problem KPMFC is NP-hard, since it contains the knapsack problem as a special case with $d = 0$. Moreover, finding a feasible solution to problem KPMFC is still NP-hard, since it contains the partition problem [3], which is a well-known NP-complete problem, as a special case with $d = c = \sum_{j=1}^{n} w_j / 2$. This implies that problem KPMFC is more challenging than the knapsack problem, because the latter problem not only always has a feasible solution with $x_j = 0$ for all $j \in J$, but also has an FPTAS.

One may notice that the transformation mentioned above, which reduces the partition problem to problem KPMFC, assigns the same value to the threshold d and the capacity c. However, in many situations, including those for applications mentioned before, the value of d is strictly less than c. For example, any buyer who orders books from Powell's Books will be provided free shipping if the order has a total value above 50 USD, which is often strictly less than the buyer's budget limit.

Therefore, it is natural to study problem KPMFC under the assumption that the ratio of d to c equals a constant $\alpha \in [0, 1)$ that does not belong to the problem instance and is fixed from outside. By a reduction from the knapsack problem, one can see that even under this assumption, finding an optimal solution to problem KPMFC is NP-hard for any constant $\alpha \in [0, 1)$. This leads to the following research question: does problem KPMFC have an FPTAS with a time complexity polynomial in $1/(1 - \alpha)$? In this paper, we give a positive answer to this question by developing such an FPTAS.

1.1 Related Works

While the knapsack problem has received wide attention in the literature, to our knowledge only two papers considered the minimum filling constraint in problems related to KPMFC. Bettinelli et al. [1] developed a branch-and-price

algorithm for a bin packing problem with the minimum filling constraint, where for each bin used, the total weight of items packed in it cannot be less than a threshold. Cappanera and Trubian [2] developed a local-search-based heuristic for a variant of the multidimensional knapsack problem, where each item j has an m-dimensional weight, and for each i where $1 \leq i \leq m$, the total weight in the i-th dimension of select items needs to satisfy a capacity constraint and a minimum filling constraint. Since both problems contain problem KPMFC as a special case, finding a feasible solution to them is NP-hard.

The knapsack problem is one of the earliest combinatorial optimization problems discovered to admit an FPTAS. Its first FPTAS was proposed by Ibarra and Kim [4], and later on improved by Lawler [10], and Kellerer and Pferschy [6, 7]. All these schemes are converted from a dynamic programming by profits, which is a pseudo-polynomial time algorithm to reach every possible total profit value with a subset of items having a smallest total weight. Clearly, the largest total profit value, which can be reached by a subset of items having a total weight not greater than the capacity c, is an optimal solution to the knapsack problem. To convert this dynamic programming to an FPTAS, it requires to scale profit values p_j to $\widehat{p}_j = \lfloor p_j/\delta_p \rfloor$ where δ_p is an appropriate scaling parameter to ensure that the resulting algorithm has an approximation ratio of $(1 - \epsilon)$ and a time complexity polynomial in n and $1/\epsilon$. For this purpose, δ_p is often determined by a lower bound L on the optimal solution, where the ratio of the optimal solution to L must be polynomial in n and $1/\epsilon$. For the knapsack problem, such a lower bound can be easily obtained by setting $L = \max\{p_j : j \in J\}$ because $w_j \leq c$ for all $j \in J$ and the total profit of an optimal solution does not exceed $n \max\{p_j : j \in J\}$. With advanced refinements of the scaling technique and other careful tuning of the algorithm, the FPTAS proposed by Kellerer and Pferschy [7] has achieved the current best time complexity of $O(n \log(1/\epsilon) + 1/\epsilon^3 \log^2(1/\epsilon))$, assuming that n is not in $O(1/\epsilon \log(1/\epsilon))$. Besides, with similar techniques and some algorithmic extensions, a number of FPTAS have been developed for various variants of the knapsack problem [9].

Unfortunately, the approach based on dynamic programming by profits is not applicable for problem KPMFC, mainly due to the following two facts: (1) the subset of items, which has the smallest total weight among those with the same total profit as the optimal solution, may not satisfy the minimal filling constraint; (2) the traditional way to compute the lower bound on the optimal solution is not valid for problem KPMFC, because selecting only the largest profitable item may not satisfy the minimal filling constraint. Therefore, to develop an FPTAS for problem KPMFC, one needs to resolve these challenges raised by the minimal filling constraint.

It is worthy to be mentioned that if the threshold d is a constant that does not belong to the problem instance, problem KPMFC would become a lot easier, and one can obtain an FPTAS for this special case by simply modifying the approach based on dynamic programming by profits. Moreover, Kellerer and Strusevich [8] recently proposed an FPTAS for a symmetric quadratic knapsack problem, which aims to minimize the total of costs associated with linear terms x_j and

$(1 - x_j)$, and costs associated with quadratic terms $x_j x_i$ and $(1 - x_j)(1 - x_i)$. Their FPTAS is converted from a dynamic programming by costs and weights, which is different from the traditional approach, and has enlightened us on the development of an FPTAS in this paper for problem KPMFC.

1.2   Our Results, Techniques, and Paper Outline

Consider problem KPMFC with the ratio of d to c equal to a constant $\alpha \in [0, 1)$. We develop the first FPTAS for this problem which returns "infeasible" if the problem has no feasible solution, and otherwise, delivers a feasible solution $\{x_j : j \in J\}$ with $\sum_{j \in J} p_j x_j \leq (1 - \epsilon) \sum_{j \in J} p_j x_j^*$ for any $\epsilon > 0$, where the time complexity is $O(n \log(n) + [n + K(n)]/[\epsilon(1 - \alpha)^3])$, and $K(n)$ is the time complexity of an existing FPTAS for the knapsack problem.

The main idea behind our algorithm is as follows. We first separate items by weights, where items with $w_j \geq (1 - \alpha)c$ are considered "big", and those with $w_j < (1 - \alpha)c$ are considered "small". We use S and B to denote the set of small items and the set of big items of J. Accordingly, at most $1/(1 - \alpha)$ big items can be selected in any feasible solution. Consider any subset of at most $1/(1 - \alpha)$ big items with a total weight value denoted by W_B. It can be seen that if problem KPMFC on J has a feasible solution $\{x_j : j \in J\}$ with $\sum_{j \in B} w_j x_j = W_B$, then the values of x_j for small items $j \in S$ form a feasible solution to problem KPMFC with $c - W_B$ as the capacity and $d - W_B$ as the threshold. We prove in Section 2 that this problem KPMFC on small items is equivalent to a knapsack problem with $c - W_B$ as the capacity, and therefore, the existing FPTAS for the latter is applicable to the former. This suggests the following way to develop an approximation algorithm for problem KPMFC on J. Firstly, we search possible values of W_B with corresponding values of x_j for big items $j \in B$ that lead to $\sum_{j \in B} w_j x_j = W_B$. Secondly, for each W_B, we apply the existing FPTAS on the knapsack problem on small items in S with $c - W_B$ as the capacity and $d - W_B$ as the threshold, and then, combine the values of x_j obtained for $j \in S$ with the values of x_j for $j \in B$ to form a feasible solution to problem KPMFC on J. Lastly, among all feasible solutions obtained, we return the one that has the largest total profit.

To ensure a time complexity polynomial in $1/(1 - \alpha)$, we cannot enumerate all possible values of W_B for big items. Therefore, we develop a partial search procedure in Section 3 to explore part of them. To further ensure an approximation ratio of $(1 - \epsilon)$ for any $0 < \epsilon \leq 1$, for each W_B explored, the partial search procedure reaches all possible values of total scaled profit of a subset of big items with a total weight equal to W_B. This is different from the traditional approach for the knapsack problem based on dynamic programming by profits.

Like those existing FPTAS for the knapsack problem, to choose an appropriate value of the scaling parameter for profits, the partial search procedure needs a lower bound L on the total profit of big items in an optimal solution. As we have explained earlier, the traditional methods to compute L are not applicable for problem KPMFC. Therefore, we revise the partial search procedure to develop an algorithm in Section 4 to compute L.

Finally, we present the implementation details of the FPTAS for problem KPMFC, and prove the correctness and time complexity in Section 5. One possible direction of the future work is to apply the results presented in this paper in developing approximation algorithms for other variants of the knapsack problem with the minimum filling constraint.

2   Separations of Items

Consider the set of big items $B \subseteq J$. Note that each item in B has a weight greater than or equal to $(1-\alpha)c$. Thus, for each $\{x_j : j \in J\}$ with $\sum_{j \in J} w_j x_j \leq c$, we have

$$\sum_{j \in B} x_j \leq c/[(1-\alpha)c] = 1/(1-\alpha). \tag{1}$$

Consider the set of small items $S \subseteq J$. Note that each item in S has a weight less than $(1-\alpha)c$. We can prove the following statement, which implies that problem KPMFC on S is equivalent to a knapsack problem.

Lemma 1. *Consider the small item set S. Given any d' and c' with $c' - d' = (1-\alpha)c$, if $\sum_{j \in S} w_j \geq d'$, then each $\{x'_j : j \in S\}$ with $x'_j \in \{0, 1\}$ for $j \in S$ and $\sum_{j \in S} w_j x'_j \leq c'$ can be transformed in $O(|S|)$ time to $\{x_j : j \in S\}$ such that $x_j \geq x'_j$ and $x_j \in \{0, 1\}$ for $j \in S$, and that $d' \leq \sum_{j \in S} w_j x_j \leq c'$.*

Proof. The proof is omitted due to space limitations.   □

Lemma 1 suggests the following way to develop an approximation algorithm for problem KPMFC. We can first search possible pairs of the total profit value and the total weight value of a subset of big items in B. For each of such pairs denoted by (P_B, W_B), let x_j for $j \in B$ denote the corresponding values of decision variables for big items that lead to $\sum_{j \in B} p_j x_j = P_B$ and $\sum_{j \in B} w_j x_j = W_B$. We then apply an existing FPTAS on the knapsack problem on small items in S with $c - W_B$ as the capacity, the solution of which can be transformed, according to Lemma 1, to values of x_j for small items $j \in S$ to satisfy $d - W_B \leq \sum_{j \in S} w_j x_j \leq c - W_B$. By combining $\{x_j : j \in B\}$ and $\{x_j : j \in S\}$, we obtain a feasible solution to problem KPMFC. Among all feasible solutions obtained, we can return the one that has the largest total profit.

3   Partial Search Procedure

To follow the way suggested by Lemma 1, we need to search possible pairs of the total profit value and the total weight value of a subset of big items in B. However, to keep a polynomial time complexity, we can search only part of them by the following partial search procedure on items with scaled profits.

Consider the big item set B. For the ease of presentation, assume that items in J are relabeled such that $B = \{1, 2, ..., n_B\}$. Consider three parameters, L,

U, and N, which will be determined in Section 4, where $N > 0$ and $U \geq L \geq 0$. For each $j \in B$, let $\widehat{p}_j = \lfloor p_j/\delta_p \rfloor$ denote the scaled value of p_j, where the scaling parameter $\delta_p = \epsilon L/N$, and define $\widehat{p}_j = 0$ when $\delta_p = 0$.

Our partial search procedure, as shown in Algorithm 1, scans decision variables in the order of their indices, and assigns each variable either the value of 0 or 1. Suppose that values have been assigned to $x_1, x_2, ..., x_j$, where $0 \leq j \leq n_B$. The procedure deals with partial solutions associated with states of the form (j, P_j, W_j), where j is the number of assigned variables, $P_j = \sum_{i=1}^{j} \widehat{p}_j x_j$ is the current total scaled profit, and $W_j = \sum_{i=1}^{j} w_j x_j$ is the current total weight. Starting with the initial state $(0, P_0, W_0) = (0, 0, 0)$, the procedure in Algorithm 1 creates new states of the form $(j+1, P_{j+1}, W_{j+1})$ from each stored state of the form (j, P_j, W_j), for each iteration j where $0 \leq j \leq n_B - 1$, by using the following relation:

$$P_{j+1} = P_j + \widehat{p}_{j+1} x_{j+1}, \text{ and } W_{j+1} = W_j + w_{j+1} x_{j+1}. \tag{2}$$

Then, among all states $(j+1, P_{j+1}, W_{j+1})$ with the same value of P_{j+1} and with W_{j+1} belonging to the same sub-intervals of range δ_w (defined in Step 1), where $\delta_w = (c-d)/2$, the procedure only stores at most two of them in Step 2(b) for further search, so as to save the running time. Finally, all the stored states of the form (j, P_j, W_j) for $0 \leq j \leq n_B$ are returned in Step 3.

Algorithm 1 (Procedure PS on (L, U, N))

1. Relabel items in J such that the big item set $B = \{1, 2, ..., n_B\}$, where n_B denotes the number of big items in B. Set $\delta_p = \epsilon L/N$, and $\delta_w = (c-d)/2 = (1-\alpha)c/2$. Split the interval $[0, (\lceil c/\delta_w \rceil + 1)\delta_w]$ into $\lceil c/\delta_w \rceil + 1$ subintervals I_r of range δ_w, where

$$I_r = [(r-1)\delta_w, r\delta_w), \text{ for } 1 \leq r \leq \lceil c/\delta_w \rceil + 1.$$

2. Store the initial state $(0, P_0, W_0) = (0, 0, 0)$. For all j from 0 to $n_B - 1$ do:
   (a) For each stored state of the form (j, P_j, W_j), and for $x_{j+1} = 0$ and 1, move (j, P_j, W_j) to a state of the form $(j+1, P_{j+1}, W_{j+1})$ using the relation (2), where P_{j+1} is less than or equal to U/δ_p, and W_{j+1} is less than or equal to $c + \delta_w$.
   (b) For each selection of the states obtained in Step 2(a) with the same value of P_{j+1} and with W_{j+1} belonging to the same subinterval I_r for $1 \leq r \leq \lceil c/\delta_w \rceil + 1$, determine the value $W_{j+1}^{\max}$ as the largest of W_{j+1} that belongs to I_r, and determine the value $W_{j+1}^{\min}$ as the smallest of W_{j+1} that belongs to I_r. If these values exist, then out of all states $(j+1, P_{j+1}, W_{j+1})$ with the same value of P_{j+1} and with W_{j+1} belonging to the same subinterval I_r for $W_{j+1}^{\min} \leq W_{j+1} \leq W_{j+1}^{\max}$, store only the states $(j+1, P_{j+1}, W_{j+1}^{\max})$ and $(j+1, P_{j+1}, W_{j+1}^{\min})$.
3. Return all states of the form (j, P_j, W_j) for $0 \leq j \leq n_B$ stored in Step 2. □

Consider all states of the form (j, P_j, W_j) for $0 \leq j \leq n_B$, explored in Step 2 of Procedure PS. There are at most $\lceil c/\delta_w \rceil + 1$ different intervals I_r that values of

W_j belong to. If $L > 0$, there are at most $\lfloor U/\delta_p \rfloor + 1$ different values of P_j, and otherwise, $L = 0$, which implies $\delta_p = 0$ and that P_j always equals 0. Thus, the time complexity of Procedure PS is $O(N(U/L)n_B/[\epsilon(1 - \alpha)])$ if $L > 0$, and is $O(n_B/(1 - \alpha))$ otherwise.

From each stored state of the form (n_B, P_{n_B}, W_{n_B}) returned by Procedure PS, we can perform a backtracking to find the corresponding values of decision variables x_j for $1 \le j \le n_B$. Thus, the following statement holds.

Lemma 2. *Suppose $B = \{1, 2, ..., n_B\}$. For each stored state (n_B, P_{n_B}, W_{n_B}) returned by Procedure PS on (L, U, N), the corresponding values of decision variables x_j for $1 \le j \le n_B$, obtained by backtracking from (n_B, P_{n_B}, W_{n_B}), satisfy*

$$\sum_{j=1}^{n_B} p_j x_j \ge \delta_p \sum_{j=1}^{n_B} \widehat{p}_j x_j = \delta_p P_{n_B}, \ and \ \sum_{j=1}^{n_B} w_j x_j = W_{n_B}.$$

Proof. Suppose that the backtracking from (n_B, P_{n_B}, W_{n_B}) finds a chain of states $(0, P_0, W_0), (1, P_1, W_1), ..., (n_B, P_{n_B}, W_{n_B})$. It is easy to show $\sum_{i=1}^{j} w_i x_i = W_j$ and $\sum_{i=1}^{j} \widehat{p}_i x_i = P_j$ by an induction for $j = 0, 1, ..., n_B$. Moreover, $\sum_{j=1}^{n_B} p_j x_j \ge \delta_p \sum_{j=1}^{n_B} \widehat{p}_j x_j$ because $p_j \ge \delta_p \widehat{p}_j$. $\square$

The following statements study the behavior of Step 2 of Procedure PS.

Lemma 3. *Suppose $B = \{1, 2, ..., n_B\}$. Assume that problem KPMFC has a feasible solution. Consider the optimal solution $\{x_j^* : j \in J\}$ in the optimal solution. If $\sum_{j=1}^{n_B} p_j x_j^* \le U$, then*

1. *for each j with $0 \le j \le n_B$, Step 2 of Procedure PS on (L, U, N) stores two states (j, P_j', W_j') and (j, P_j'', W_j'') such that*

$$\sum_{i=1}^{j} p_i x_i^* \ge \delta_p \max\{P_j', P_j''\} \ge \delta_p \min\{P_j', P_j''\} \ge \sum_{i=1}^{j} p_i x_i^* - \delta_p (\sum_{i=1}^{j} x_i^*), \quad (3)$$

$$\sum_{i=1}^{j} w_i x_i^* - \delta_w \le W_j' \le \sum_{i=1}^{j} w_i x_i^* \le W_j'' \le \sum_{i=1}^{j} w_i x_i^* + \delta_w; \quad (4)$$

2. *if $\sum_{j=1}^{n_B} x_j^* \le N$, $L \le \sum_{j=1}^{n_B} p_j x_j^*$, and $d' \le \sum_{j=1}^{n_B} w_j x_j^* \le c'$, where c' and d' are any non-negative numbers with $d' + (1 - \alpha)c \le c' \le c$, then Step 2 of Procedure PS on (L, U, N) stores a state $(n_B, \widetilde{P}_{n_B}, \widetilde{W}_{n_B})$, such that*

$$d' \le \widetilde{W}_{n_B} \le c', \quad (5)$$

$$\delta_p \widetilde{P}_{n_B} \ge (1 - \epsilon) \sum_{j=1}^{n_B} p_j x_j^*. \quad (6)$$

Proof. The proof is omitted due to space limitations. $\square$

4   Computing L and U

Our FPTAS for problem KPMFC is based on Procedure PS, by exploiting the properties shown in Lemma 2 and Lemma 3. Thus, we need to determine values of parameters, N, L and U, for Procedure PS to satisfy the conditions specified in Lemma 3. Since (1) implies that setting $N = 1/(1-\alpha)$ satisfies $\sum_{j \in B} x_j^* \leq N$, we devote the remainder of this section to the computation of L and U such that $L \leq \sum_{j \in B} w_j x_j^* \leq U$ and $U/L \leq 1/(1-\alpha)$.

Consider the big item set B. If problem KPMFC has a feasible solution, we define $\pi(B)$ as the largest value of $\max\{p_j x_j : j \in B\}$ over all feasible solutions $\{x_j : j \in J\}$ to problem KPMFC, and define $\pi(B) = 0$ when B is empty. If problem KPMFC has no feasible solution, we define $\pi(B) = -\infty$. The following statement holds for $\pi(B)$, suggesting some feasible values for L and U.

Lemma 4. *If $\pi(B) \geq 0$, setting $L = \pi(B)$ and $U = \pi(B)/(1-\alpha)$ satisfies that $L \leq \sum_{j \in B} p_j x_j^* \leq U$ and $U/L = 1/(1-\alpha)$.*

Proof. If $\pi(B) \geq 0$, then by definition, there exists a feasible solution $\{x_j : j \in J\}$ and an item $i \in B$ with $p_i x_i = \pi(B)$, which implies that $\sum_{j \in B} p_j x_j^* \geq p_i x_i = L$. Moreover, for each $i' \in B$ with $x_{i'}^* = 1$, we know that $p_{i'} \leq \pi(B)$. Thus, $\sum_{j \in B} p_j x_j^* \leq \pi(B) \sum_{j \in B} x_j^*$. This, together with (1), implies that $\sum_{j \in B} p_j x_j^* \leq \pi(B)/(1-\alpha) = U$, which completes the proof. $\qquad\square$

The value of $\pi(B)$ can be computed as follows. Suppose items in J are relabeled such that $B = \{n - n_B + 1, n - n_B + 2, ..., n\}$ and $p_j \leq p_{j+1}$ for $n - n_B + 1 \leq j \leq n - 1$. Define a state of the form (j, γ_j, W_j), where $1 \leq j \leq n$, $\gamma_j \in \{0, 1\}$, and $W_j \geq 0$, to indicate that there exist values of x_i for $1 \leq i \leq j$ with $x_j = \gamma_j$ and $\sum_{i=1}^{j} w_i x_i = W_j$. Algorithm 2 follows an approach similarly to Procedure PS to search states (j, γ_j, W_j), from which the value of $\pi(B)$ can be determined.

Algorithm 2 (Computing $\pi(B)$)

1. Relabel items in J and sort items in B, so that $B = \{n - n_B + 1, n - n_B + 2, ..., n\}$ and $p_j \leq p_{j+1}$ for $n - n_B + 1 \leq j \leq n - 1$. Similarly to Step 1 of Procedure PS, set $\delta_w = (c - d)/2 = (1 - \alpha)c/2$, and split the interval $[0, (\lceil c/\delta_w \rceil + 1)\delta_w]$ into subintervals I_r for $1 \leq r \leq \lceil c/\delta_w \rceil + 1$ of range δ_w.
2. Store the initial state $(0, \gamma_0, W_0) = (0, 0, 0)$. For all j from 0 to $n - 1$ do:
   (a) Similarly to Step 2(a) of Procedure PS, for each stored state of the form (j, γ_j, W_j), and for $x_{j+1} = 0$ and 1, move (j, γ_j, W_j) to a state of the form $(j+1, \gamma_{j+1}, W_{j+1})$, by setting $W_{j+1} = W_j + w_{j+1}x_{j+1}$ and $\gamma_{j+1} = x_{j+1}$.
   (b) Similarly to Step 2(b) of Procedure PS, for each selection of the states obtained in Step 2(a) with the same value of γ_{j+1} and with W_{j+1} belonging to the same subinterval I_r for $1 \leq r \leq \lceil c/\delta_w \rceil + 1$, determine the value $W_{j+1}^{\max}$ as the largest of W_{j+1} that belongs to I_r, and determine the value $W_{j+1}^{\min}$ as the smallest of W_{j+1} that belongs to I_r. If these values exist, then out of all states $(j+1, \gamma_{j+1}, W_{j+1})$ with the same value of γ_{j+1} and with W_{j+1} belonging to the same subinterval I_r for $W_{j+1}^{\min} \leq W_{j+1} \leq W_{j+1}^{\max}$, store only the states $(j+1, \gamma_{j+1}, W_{j+1}^{\max})$ and $(j+1, \gamma_{j+1}, W_{j+1}^{\min})$.

3. Among all states of the form (j, γ_j, W_j) stored in Step 2, if no state satisfies $d \leq W_j \leq c$, then return $-\infty$. Otherwise, if B is empty, then return 0, and otherwise, among all states with $d \leq W_j \leq c$ and $j \in B$, identify the state with the largest $p_j \gamma_j$, and then return $p_j \gamma_j$ of this state.   □

Accordingly, the following statement holds for Algorithm 2.

Theorem 1. *Algorithm 2 returns $\pi(B)$ in $O(n \log(n) + n/(1 - \alpha))$ time.*

Proof. Step 1 of Algorithm 2 takes $O(n \log(n))$ time, and Step 2 takes $O(n \lceil c/\delta_w \rceil)$ time. Since $\delta_w = (1 - \alpha)c/2$, the time complexity of Algorithm 2 is $O(n \log(n) + n/(1 - \alpha))$.

We are next going to prove that Algorithm 2 returns the value of $\pi(B)$. If there exists a state (j, γ_j, W_j) with $d \leq W_j \leq c$ stored in Step 2 of Algorithm 2, then by following an argument similarly to the proof of Lemma 2, we can obtain that the corresponding values of x_i for $1 \leq i \leq j$, obtained by backtracking from (j, γ_j, W_j), satisfy $d \leq \sum_{i=1}^{j} w_i x_i = W_j \leq c$ and $x_j = \gamma_j$. Therefore, combining $\{x_i : 1 \leq i \leq j\}$ and $x_i = 0$ for $j+1 \leq i \leq n$ forms a feasible solution to problem KPMFC with $x_j = \gamma_j$.

Thus, if problem KPMFC has no feasible solution, no state (j, γ_j, W_j) with $d \leq W_j \leq c$ can be stored in Step 2 of Algorithm 2. Thus, according to Step 3, Algorithm 2 returns $-\infty$, which is equal to $\pi(B)$. Moreover, if problem KPMFC has a feasible solution and B is empty, according to Step 3, Algorithm 2 returns 0, which is also equal to $\pi(B)$.

Finally, consider the case when problem KPMFC has a feasible solution and B is not empty. Note that after Step 1 of Algorithm 2, $B = \{n - n_B + 1, n - n_B + 2, ..., n\}$ and $p_j \leq p_{j+1}$ for $n - n_B + 1 \leq j \leq n - 1$. Thus, no state (j, γ_j, W_j) with $j \in B$, $d \leq W_j \leq c$, and $p_j \gamma_j > \pi(B)$ can be stored in Step 2 of Algorithm 2, because otherwise, as we have shown earlier, there must exist a feasible solution $\{x_j : j \in J\}$ to problem KPMFC with $x_j = \gamma_j$ for $j \in B$, implying that $p_j x_j > \pi(B)$, which contradicts the definition of $\pi(B)$.

Therefore, to prove that Algorithm 2 returns the value of $\pi(B)$, we only need to show that it stores in Step 2 a state $(y, \widetilde{\gamma}_y, \widetilde{W}_y)$ with $y \in B$, $d \leq \widetilde{W}_y \leq c$, and $p_y \widetilde{\gamma}_y = \pi(B)$. To prove this, define $y \in B$ as the item for which there exists a feasible solution $\{x_j : j \in J\}$ to problem KPMFC with $p_y x_y = \pi(B)$, breaking ties by selecting larger y. Thus, for all $y + 1 \leq i \leq n$, since $p_i \geq p_y$ and $i \in B$, by the definition of $\pi(B)$ and y, we have $x_i = 0$, which implies

$$d \leq \sum_{i=1}^{y} w_i x_i = \sum_{i \in J} w_i x_i \leq c. \tag{7}$$

By following an induction similarly to the proof of statement 1 of Lemma 3, we can show that for $j = 0, 1, ..., n$, Step 2 of Algorithm 2 stores two states (j, γ_j', W_j') and (j, γ_j'', W_j'') such that (4) holds for W_j' and W_j'', and that

$$\gamma_j' = \gamma_j'' = x_j.$$

From this, by following an argument similarly to the proof of statement 2 of Lemma 3, and noting that $d \leq \sum_{i=1}^{y} w_i x_i \leq c$ by (7), we can obtain that

Algorithm 2 stores a state $(y, \widetilde{\gamma}_y, \widetilde{W}_y)$ with $d \leq \widetilde{W}_y \leq c$ and $\widetilde{\gamma}_y = x_y$, which together with $p_y x_y = \pi(B)$, implies $p_y \widetilde{\gamma}_y = p_y x_y = \pi(B)$. Therefore, the value returned by Algorithm 2 must be equal to $\pi(B)$.   □

5   FPTAS

By Lemma 4 and Theorem 1, we can determine L and U such that $L \leq \sum_{j \in B} w_j x_j^* \leq U$ and $U/L = 1/(1 - \alpha)$. This, together with $N = 1/(1 - \alpha)$, allows us to apply Procedure PS on big items in B to obtain stored states of the form (n_B, P_{n_B}, W_{n_B}) which hold properties specified in Lemma 2 and Lemma 3. With such states, we can follow the idea suggested by Lemma 1 to develop an approximation algorithm for problem KPMFC, which is shown in Algorithm 3.

Algorithm 3 (FPTAS)

1. Apply Algorithm 2 to compute $\pi(B)$. If $\pi(B)$ equals $-\infty$, then return "infeasible".
2. Apply Procedure PS on (L, U, N), where $L = \pi(B)$, $U = \pi(B)/(1 - \alpha)$, and $N = 1/(1 - \alpha)$.
3. For each stored state of the form (n_B, P_{n_B}, W_{n_B}), returned by Procedure PS in Step 2, do the followings:
   (a) Backtrack from the state (n_B, P_{n_B}, W_{n_B}) to obtain values of x_j for $j \in B$.
   (b) If $\sum_{j \in S} w_j \geq d - W_{n_B}$ then
      i. Apply an existing FPTAS on a knapsack problem on small items in S with $c - W_{n_B}$ as the capacity. Let $\{x_j' : j \in S\}$ denote the solution returned.
      ii. Transform $\{x_j' : j \in S\}$ to $\{x_j : j \in S\}$, according to Lemma 1, such that $x_j \geq x_j'$, $x_j \in \{0, 1\}$, and $d - W_{n_B} \leq \sum_{j \in S} w_j x_j \leq c - W_{n_B}$;
      iii. Combine values of x_j for $j \in B$ obtained in Step 3(a), and values of x_j for $j \in S$ obtained in Step 3(b).ii, to obtain $\{x_j : j \in J\}$.
4. Among all $\{x_j : j \in J\}$ obtained in Step 3(b).iii, return the one that has the largest value of $\sum_{j \in J} p_j x_j$.   □

The following statement studies the behavior of Step 2 of Algorithm 3.

Lemma 5. *Assume that problem KPMFC has a feasible solution. Consider the optimal solution $\{x_j^* : j \in J\}$. Then, Procedure PS in Step 2 of Algorithm 3 returns at least one stored state $(n_B, \widetilde{P}_{n_B}, \widetilde{W}_{n_B})$ such that*

$$d \leq \widetilde{W}_{n_B} + \sum_{j \in S} w_j x_j^* \leq c, \text{ and } \delta_p \widetilde{P}_{n_B} \geq (1 - \epsilon) \sum_{j \in B} p_j x_j^*.$$

Proof. According to Lemma 4, setting $L = \pi(B)$ and $U = \pi(B)/(1 - \alpha)$ satisfies that $L \leq \sum_{j \in B} p_j x_j^* \leq U$. Since $\sum_{j \in B} x_j^* \leq 1/(1 - \alpha)$ by (1), due to statement 2 of Lemma 3, we obtain that applying Procedure PS on (L, U, N), where $L = \pi(B)$, $U = \pi(B)/(1 - \alpha)$, and $N = 1/(1 - \alpha)$, returns at least one stored state $(n_B, \widetilde{P}_{n_B}, \widetilde{W}_{n_B})$ with $d - \sum_{j \in S} w_j x_j^* \leq \widetilde{W}_{n_B} \leq c - \sum_{j \in S} w_j x_j^*$ and $\delta_p \widetilde{P}_{n_B} \geq (1 - \epsilon) \sum_{j \in B} p_j x_j^*$, which completes the proof.   □

Finally, we can establish the following theorem for Algorithm 3.

Theorem 2. *Algorithm 3 is an FPTAS for problem KPMFC, and has a time complexity of $O(n \log(n) + [n + K(n)]/[\epsilon(1 - \alpha)^3])$, where $K(n)$ is the time complexity of an existing FPTAS for the knapsack problem.*

Proof. If problem KPMFC has no feasible solution, by Lemma 4, we obtain that $\pi(B)$ returned by Step 1 of Algorithm 3 equals $-\infty$. Thus, Algorithm 3 returns "infeasible".

Otherwise, problem KPMFC has a feasible solution. For each (n_B, P_{n_B}, W_{n_B}) returned by Procedure PS in Step 2, if $\sum_{j \in S} w_j \geq d - W_{n_B}$, then according to Lemma 2 and Lemma 1, values of x_j for $1 \leq j \leq n$ stored in Step 3(b).iii form a feasible solution to problem KPMFC.

Moreover, according to Lemma 5 and Lemma 2, there exists a stored state $(n_B, \widetilde{P}_{n_B}, \widetilde{W}_{n_B})$ with the corresponding values of x_j for $j \in B$ satisfying

$$\sum_{j \in B} p_j x_j \geq \delta_p \widetilde{P}_{n_B} \geq (1 - \epsilon) \sum_{j \in B} p_j x_j^*, \tag{8}$$

and $d - \sum_{j \in S} w_j x_j^* \leq \sum_{j \in B} w_j x_j = \widetilde{W}_{n_B} \leq c - \sum_{j \in S} w_j x_j^*$, which implies

$$d - \widetilde{W}_{n_B} \leq \sum_{j \in S} w_j x_j^* \leq c - \widetilde{W}_{n_B}. \tag{9}$$

Thus, $\{x_j^* : j \in S\}$ is a feasible solution to a knapsack problem on small items in S with $c - \widetilde{W}_{n_B}$ as the capacity. By (9) and $\sum_{j \in S} w_j x_j^* \leq \sum_{j \in S} w_j$, we have $d - \widetilde{W}_{n_B} \leq \sum_{j \in S} w_j$, which implies that Step 3(b).i of Algorithm 3 has applied an existing FPTAS on this knapsack problem on S. Consider the solution $\{x_j' : j \in S\}$ returned in Step 3(b).i for this knapsack problem on S, which is transformed in Step 3(b).ii to $\{x_j : j \in S\}$ such that $x_j \geq x_j'$, $x_j \in \{0, 1\}$, and $d - \widetilde{W}_{n_B} \leq \sum_{j \in S} w_j x_j \leq c - \widetilde{W}_{n_B}$. We have

$$d \leq \sum_{j \in S} w_j x_j + \sum_{j \in B} w_j x_j \leq c, \tag{10}$$

$$\sum_{j \in S} p_j x_j \geq \sum_{j \in S} p_j x_j' \geq (1 - \epsilon) \sum_{j \in S} p_j x_j^*. \tag{11}$$

By (10), $\{x_j : j \in J\}$ is a feasible solution to problem KPMFC on J. Thus, Step 4 of Algorithm 3 must return a feasible solution with the total profit greater than or equal to $\sum_{j \in J} p_j x_j$, which is greater than or equal to $(1 - \epsilon) \sum_{j \in J} p_j x_j^*$ due to (8) and (11).

Finally, due to Theorem 1, Step 1 of Algorithm 3 takes $O(n \log(n) + n/(1 - \alpha))$ time. Since $U/L = 1/(1 - \alpha)$ when $L > 0$ and $N = 1/(1 - \alpha)$, Procedure PS in Step 2 of Algorithm 3 takes $O(n/[\epsilon(1 - \alpha)^3])$ time. Note that the number of stored states of the form (n_B, P_{n_B}, W_{n_B}) obtained in Step 2 is in $O(N(U/L)/[\epsilon(1 - \alpha)])$ if $L > 0$, and in $O(1/(1 - \alpha))$ otherwise. The time complexity of Step 3 is $O(K(n)/[\epsilon(1 - \alpha)^3])$. Thus, Algorithm 3 has a time complexity of $O(n \log(n) + [n + K(n)]/[\epsilon(1 - \alpha)^3])$, and is an FPTAS. □

Note that $K(n)$ can be $O(n \log(1/\epsilon) + \log^2(1/\epsilon)/\epsilon^3)$ according to Kellerer and Pferschy [7]. Thus, by Theorem 2, the FPTAS developed in this paper for problem KPMFC can achieve a time complexity of $O(n \log(n) + n \log(1/\epsilon)/[\epsilon(1 - \alpha)^3] + \log^2(1/\epsilon)/[\epsilon^4(1 - \alpha)^3])$.

References

[1] Bettinelli, A., Ceselli, A., Righini, G.: A branch-and-price algorithm for the variable size bin packing problem with minimum filling constraint. Annals of Operations Research (2010) (forthcoming)

[2] Cappanera, P., Trubian, M.: A local-search-based heuristic for the demand-constrained multidimensional knapsack problem. INFORMS Journal on Computing 17(1), 82–98 (2005)

[3] Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. WH Freeman & Co., New York (1979)

[4] Ibarra, O.H., Kim, C.E.: Fast approximation algorithms for the knapsack and sum of subset problems. Journal of the ACM (JACM) 22(4), 463–468 (1975)

[5] Kameshwaran, S., Benyoucef, L.: Optimal buying from online retailers offering total value discounts. In: Proceedings of 10th International Conference on Electronic Commerce, Innsbruck, Austria (2008)

[6] Kellerer, H., Pferschy, U.: A new fully polynomial time approximation scheme for the knapsack problem. Journal of Combinatorial Optimization 3(1), 59–71 (1999)

[7] Kellerer, H., Pferschy, U.: Improved dynamic programming in connection with an fptas for the knapsack problem. Journal of Combinatorial Optimization 8(1), 5–11 (2004)

[8] Kellerer, H., Strusevich, V.A.: Fully polynomial approximation schemes for a symmetric quadratic knapsack problem and its scheduling applications. Algorithmica 57(4), 769–795 (2010)

[9] Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack problems. Springer, Heidelberg (2004)

[10] Lawler, E.L.: Fast approximation algorithms for knapsack problems. Mathematics of Operations Research 4(4), 339–356 (1979)

Author Index