Algorithms for Planar Graphs and Graphs in Metric Spaces

by

Christian Wulff-Nilsen
Department of Computer Science, University of Copenhagen, Denmark

Ph.D. Thesis

The Ph.D. School of Science, Faculty of Science, University of Copenhagen, Denmark

March 2010

Abstract

Algorithms for network problems play an increasingly important role in modern society. The graph structure of a network is an abstract and very useful representation that allows classical graph algorithms, such as Dijkstra and Bellman-Ford, to be applied. Real-life networks often have additional structural properties that can be exploited. For instance, a road network or a wire layout on a microchip is typically (near-)planar and distances in the network are often defined w.r.t. the Euclidean or the rectilinear metric. Specialized algorithms that take advantage of such properties are often orders of magnitude faster than the corresponding algorithms for general graphs.

The first and main part of this thesis focuses on the development of efficient planar graph algorithms. The most important contributions include a faster single-source shortest path algorithm, a distance oracle with subquadratic preprocessing time, an $O(n \log n)$ time algorithm for the replacement paths problem, and a min *st*-cut oracle with near-linear preprocessing time. We also give improved time bounds for computing various graph invariants such as diameter and girth.

In the second part, we consider stretch factor problems for geometric graphs and graphs embedded in metric spaces. Roughly speaking, the stretch factor is a real value expressing how well a (geo-)metric graph approximates the underlying complete graph w.r.t. distances. We give improved algorithms for computing the stretch factor of a given graph and for augmenting a graph with new edges while minimizing stretch factor.

The third and final part of the thesis deals with the Steiner tree problem in the plane equipped with a weighted fixed orientation metric. Here, we give an improved theoretical analysis of the strength of pruning techniques applied by many Steiner tree algorithms. We also present an algorithm that computes a so called Steiner hull, a structure that may help in the computation of a Steiner minimal tree.

Preface

This thesis is submitted in partial fulfilment of the requirements for the Ph.D. degree at the Department of Computer Science, University of Copenhagen (DIKU). The thesis has been prepared during the period May 2007 to March 2010 and the work has been supervised by Martin Zachariasen.

The research project consists of three main areas: planar graph problems, stretch factor problems for graphs in (geo-)metric spaces, and the Steiner tree problem for weighted fixed orientation metrics in the plane. The thesis consists of a short introduction to the subjects and 13 research papers. These papers are listed among other references in the introduction (page 20) and separately on page 26. A short Danish summary is given on page 25. It is a direct translation of the English abstract.

I would like to thank (in alphabetical order) Glencora Borradaile, Prosenjit Bose, Stefan Langerman, Jun Luo, Pat Morin, Shay Mozes, David Pisinger, Michiel Smid, Pawel Winter, and Martin Zachariasen.

Copenhagen, March 2010

Christian Wulff-Nilsen

Contents

1 Introduction

Algorithms for network problems play an increasingly important role in modern society. They find numerous applications in transportation, communication networks, production and inventory planning, facility location and allocation, and VLSI design, to name just a few areas.

Since real-world network problems are often of considerable size, the development of efficient algorithms for these problems has been an active and important area of research for many years.

A network can be represented as a graph which may be either weighted or unweighted and directed or undirected, depending on the application. Since we are dealing with networks which often represent road maps, wires on a microchip, etc., the underlying graph is often planar, meaning that it can be embedded in the plane such that no two edges cross, and/or geometric, meaning that it is embedded in Euclidean space, where the weight of an edge is equal to the Euclidean distance between its endpoints.

In this thesis, we focus on algorithms for the following classes of graphs: planar graphs, graphs in arbitrary metric spaces, and graphs in geometric and fixed orientation metric spaces. For planar graphs, we present new and faster algorithms for shortest paths, for min cuts, and for computing various graph invariants. For graphs in (geo-)metric spaces, we are mainly interested in computing their so called detour and/or stretch factor and we give improved algorithms for computing these quantities. We also look at the Steiner tree problem for fixed orientation metrics.

In Sections 2, 3, and 4, we give an overview of the problems we consider and we state our new results and present some of the main ideas involved. We conclude in Section 5 with a summary as well as ideas for future research. The papers containing our research results can be found in the appendix.

2 Planar Graphs

Real-world networks such as road maps and electrical circuits tend to be planar or "nearly" planar. From an algorithmic point of view, it is therefore natural to consider network problems for planar graphs. Often, enforcing planarity allows a network problem to be solved efficiently.

2.1 Toolbox

Before presenting our results, let us give a short overview of some of the main tools applied in efficient planar graph algorithms. We make use of these tools to obtain our results.

Separator theorem The first and perhaps most important is the celebrated separator theorem of Lipton and Tarjan [22]. It gives a linear-time algorithm for splitting an n-vertex plane graph G into two subgraphs of approximately the same size such that the number of vertices shared by the two graphs is small, namely $O(\sqrt{n})$. We call such vertices *boundary vertices*.

A standard technique for solving a planar graph problem, one which we use extensively in this thesis, is to apply the separator theorem together with divide-and-conquer: the separator theorem splits G in two, the problem is recursively solved for the subgraphs, and the two solutions are combined into a solution for G.

r-division Another well-known variant, which we also use in some of our papers, is not to solve the problem recursively but instead to apply the separator theorem recursively until the subgraps have a certain size. Typically, some preprocessing is then made for each subgraph which can be used to speed up an algorithm for the problem for G. The result of this repeated application of the separator theorem is a so-called *r-division* of G, where r is some parameter in $(0, n)$. Introduced by Frederickson [12], an r-division consists of $O(n/r)$ subgraphs each of size $O(r)$ and each containing $O(\sqrt{r})$ boundary vertices. Distinct subgraphs only share boundary vertices.

Cycle separator theorem A stronger version of Lipton and Tarjan's result is Miller's cycle separator theorem [25]. Here, it can be assumed that all boundary vertices are on a Jordan curve which does not cross any edges of G. This property has proven very useful and we make extensive use of Miller's theorem in several of our papers.

Dense distance graph Fakcharoenphol and Rao [10] defined what they called the *dense distance graph* of a plane graph G. It is obtained by first applying the cycle separator theorem of Miller recursively to G. The resulting recursive subdivision can be represented as a tree, where the root corresponds to G and each leaf corresponds to a single edge. The various subgraphs obtained are called *pieces*. A piece containing r vertices has $O(\sqrt{r})$ boundary vertices and the dense distance graph is obtained by adding, for each piece P, an edge between every pair of boundary vertices; the weight of the edge is set to the weight of the shortest path in P between its endpoints. Fakcharoenphol and Rao show how to compute the dense distance graph in $O(n \log^3 n)$ time and that, given this graph, distance queries in G can be answered in $\tilde{O}(\sqrt{n})$ time [1]. We use the dense distance graph in a different way to obtain our min st-cut oracle.

Multiple-source shortest paths The final tool that deserves attention is the multiple-source shortest path algorithm of Klein [19]. With $O(n \log n)$ preprocessing, this algorithm can answer in $O(\log n)$ time shortest path distance queries between vertices u and v, where u is any vertex of G and v is a vertex of some fixed face of G. This algorithm has proven extremely useful together with Miller's cycle theorem and has lead to faster algorithms for several fundamental planar graph problems. We also use this algorithm as well as the ideas behind it in some of our papers.

2.2 Shortest paths

From an algorithmic point of view, computing shortest paths is one of the most fundamental graph problems and has received a lot of attention. The classical Dijkstra's algorithm can be implemented to solve the single-source shortest path (SSSP) problem for digraphs with non-negative edge weights in $O(m + n \log n)$ time, where m is the number of edges and n is the number of vertices of the graph. The Bellman-Ford algorithm can also deal with negative weight edges and solves the SSSP problem in $O(mn)$ time, assuming that no cycles of negative weight are present.

For planar n-vertex digraphs, an optimal $O(n)$ time algorithm for the SSSP was presented in [16]. However, like Dijkstra's algorithm, this linear time algorithm assumes that all edge weights are non-negative.

For planar n-vertex digraphs with arbitrary real edge weights (and no negative cycles), Lipton, Rose, and Tarjan [21] gave an algorithm that runs in $O(n^{3/2})$ time. Henzinger, Klein, Rao, and Subramanian [16] obtained a (not strongly) polynomial bound of $\tilde{O}(n^{4/3})$. A major breakthrough was due

[1]The $\tilde{O}$-notation is defined like O-notation but log-factors are ignored.

to Fakcharoenphol and Rao [10] who gave a near-linear time algorithm. More precisely, their algorithm runs in $O(n \log^3 n)$ time and requires $O(n \log n)$ space. It makes use of the recursive subdivision and the dense distance graph mentioned above. Later, Klein, Mozes, and Weimann [20] improved time to $O(n \log^2 n)$ and space to $O(n)$.

One of the main contributions in this thesis is an improvement of the algorithm in [20]. Our algorithm runs in $O(n \log^2 n / \log \log n)$ time and also requires linear space [26].

The new idea in our algorithm is relatively simple. Klein, Mozes, and Weimann [20] apply the cycle separator theorem of Miller together with divide-and-conquer, solving the problem in $O(n \log^2 n)$ time by using $O(n \log n)$ time at each of the $O(\log n)$ recursion levels. We show that instead of splitting the graph in two at each level, we obtain an n/p-division, thereby splitting the graph into $O(p)$ subgraphs, for a suitable choice of p. The problem is recursively solved for each such graph and the solutions are combined into a solution for the entire graph. The recursion depth is now only $O(\log n / \log p)$ and we show that the solutions can be merged in $O(n \log n + np\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function. It then follows easily that the total running time of our algorithm is

$$O\left(\frac{\log n}{\log p}(n \log n + np\alpha(n))\right).$$

By setting $n \log n = np\alpha(n)$, i.e., $p = \log n / \alpha(n)$, we obtain the desired $O(n \log^2 n / \log \log n)$ time bound.

2.3 Replacement paths

Communication networks are in general not static but may change due to link failures. In such cases, alternative lines of communication need to be established and it may be of interest to determine the "quality" of such lines.

This motivates the *replacement paths problem (RPP)*: given two vertices s and t in a graph G with non-negative edge lengths and given a shortest path P (the line of communication) in G from s to t, compute, for each edge e on P, the length of a shortest path in G from s to t that avoids e.

Another variant of the RPP, which we do not consider here, requires the actual replacement path for each choice of e to be reported.

For undirected graphs, the RPP can be solved in time $O(m + n \log n)$ [24] and $O(m\alpha(m, n))$ [27], respectively (the latter bound assumes a stronger model of computation). The directed case can be solved in $O(mn+n^2 \log\log n)$ time [14].

For planar n-vertex digraphs, an $O(n \log^3 n)$ time recursive algorithm was given in [8]. Klein, Mozes, and Weimann [20] showed how to avoid recursion and improved running time to $O(n \log^2 n)$. Space requirement is linear.

In this thesis, we show how to shave off another log-factor, thereby obtaining an $O(n \log n)$ time algorithm [39]. Space requirement is linear. Whereas the algorithms in [8] and [20] use Klein's multiple-source shortest path algorithm as a black-box, we instead take a more direct approach and adapt his algorithm to the replacement paths problem. We show that the replacement paths can be computed by maintaining a dynamic shortest path tree (as well as its dual) which changes as the choice of the edge e on P changes (using the above notation). During the course of the algorithm, the length of the replacement path that avoids e can be extracted from the corresponding dynamic shortest path tree. Using top trees [2], each elementary tree operation takes $O(\log n)$ time. By proving that the total number of changes to the dynamic trees is linear, we obtain our $O(n \log n)$ time algorithm.

2.4 Distance oracle

Above, we considered shortest path problems where the source and/or target vertex is fixed. Now, suppose we need to answer distance queries where we know neither vertex in advance. The problem is to build a data structure that can answer such queries efficiently, preferably in constant time.

Thorup [28] gave an oracle for *approximate* distance queries in planar digraphs with near-linear preprocessing time. An *exact* distance oracle for n-vertex planar digraphs can be constructed using $\Theta(n^2)$ time and space. It is obtained by applying the quadratic time all-pairs-shortest path algorithm of Frederickson [12] and storing the distances between all pairs of vertices.

It was open whether this quadratic bound could be improved. A main result of our thesis is that this is indeed the case: there is an oracle for exact distance queries requiring subquadratic time and space for preprocessing. More precisely, we show that a time and space bound of $O(n^2 \log\log n/\log n)$ is achievable for unweighted and undirected planar graphs and $O(n^2(\log\log n)^4/\log n)$ is achievable for weighted planar digraphs [41, 42][2]. The $O(n^2 \log\log n/\log n)$

[2]The $O(n^2 \log\log n/\log n)$ bound for unweighted and undirected planar graphs is not stated in [41] but it follows by applying the same ideas as in [42].

time algorithm generalizes to the larger class of subgraph-closed $\sqrt{n}$-separable graphs for which an r-division can be found efficiently. This includes important subclasses such as graphs of bounded genus.

Of course, this is only a slight improvement of the time and space bounds of Frederickson's algorithm [12] and has no practical applications but it is interesting from a theoretical point of view since it shows that the lower bound is not quadratic.

2.5 Graph invariants

A *graph invariant* is a property of a graph that depends only on the abstract structure of the graph. Three important graph invariants that we focus on are:

- diameter (maximum distance between any pair of vertices),
- Wiener index (sum of all-pairs shortest path distances).
- girth (length of the shortest cycle in the graph),

2.5.1 Diameter and Wiener Index

For planar graphs, it was open whether subquadratic time algorithms exist for computing the diameter (Problem 6.2 in [7]) and Wiener index[6]. We solve both of these open problems by showing how the ideas in our distance oracle can be used to obtain algorithms with the same running time for computing the diameter and Wiener index [41, 42].

2.5.2 Girth

Eppstein [9] showed that the girth of a planar graph can be computed in linear time but only when the girth is bounded by a constant. Recently, it was shown how to find the girth in $O(n \log n)$ time without this assumption [30]. However, these results assume that the graph is undirected and unweighted. For planar unweighted digraphs, Weimann and Yuster [30] gave an $O(n^{3/2})$ time algorithm and they asked whether a faster algorithm exists.

We answer this in the affirmative by exhibiting an $O(n \log^3 n)$ time algorithm [37]. Our algorithm applies to planar digraphs with arbitrary real edge weights. It computes the girth and can be extended to output the corresponding cycle in time proportional to its size (unless a negative weight cycle exists in which case our algorithm reports its existence).

Our girth algorithm is relatively straightforward: apply the cycle separator theorem of Miller and recursively compute the girth of the two subgraphs. Then use an efficient Dijkstra variant due to Fakcharoenphol and Rao [10] to compute the smallest weight of a cycle crossing the separator. The minimum of these three values is the girth of the entire graph.

2.6 Min st-cut oracle

Another important graph problem is that of finding min cuts. Given two vertices s and t in an undirected graph G with non-negative edge weights, a min st-cut in G is a set of edges of minimum weight whose removal leaves s and t in distinct connected components. The *all-pairs min cut problem* (APMCP) is the problem of finding a minimal collection of cuts such that for each pair of vertices s and t, the collection contains a min st-cut. Gomory and Hu [13] showed how all these cuts can be compactly represented in a tree, where:

- the nodes of the tree correspond one-to-one with the vertices of G,
- the minimum edge weight on the unique simple s-to-t path in the tree is the weight of a min st-cut in G, and
- removing this edge from the tree partitions the nodes into two sets S and T which is a partition of the vertices of G corresponding to a min st-cut.

We call such a tree a *Gomory-Hu tree* of G. Gomory and Hu also showed that this tree can be obtained using just $n - 1$ calls to a min cut algorithm, where n is the number of vertices of G. This results in an $O(n^2 \log n)$ time algorithm for planar graphs using the fastest known min cut algorithm for such graphs [4].

For planar graphs, it has been shown that the following problem is dual equivalent to the APMCP [15] (meaning that one problem can be transformed into the other in linear time): given a weighted and undirected graph G, find a basis of minimum weight for the cycle space of G. This problem, called the *minimum cycle basis problem* (MCBP), is of independent interest since it has applications in such diverse areas as electrical circuit theory, algorithm analysis, chemical and biological pathways, periodic scheduling, and graph drawing.

2.6.1 An $\tilde{O}(n^{3/2})$ time algorithm

The MCBP (and hence the APMCP) can be solved in $O(n^2)$ time for planar n-vertex graphs [3]. We show that this is optimal by presenting a family of planar graphs of arbitrarily large size n such that any cycle basis of the graph has total size $\Theta(n^2)$ [38]. We then show how to break the quadratic time barrier by computing a cycle basis *implicitly*, using $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space. From this result, we obtain a Gomory-Hu tree within the same time and space bounds. We show how to derive from it an oracle for min st-cut queries. More precisely, with $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space for preprocessing, min st-cut weight queries can be answered in constant time per query. The previous best time/space bound for such an oracle was quadratic.

The oracle is obtained by performing the following steps. First, the APMCP is implicitly solved for the input graph G by implicitly solving the MCBP for the dual graph (here, we make use of the above dual equivalence). The algorithm applies the cycle separator theorem of Miller and recursively finds a minimum cycle basis of the two subgraphs defined by the separator cycle. Additional cycles that cross the separator are then computed and we end up with a superset of the minimum cycle basis. A greedy algorithm is then applied to extract the basis from this superset. This algorithm considers cycles in order of non-decreasing weight and adds a cycle if it separates a pair of faces not separated by previously added cycles.

2.6.2 An $\tilde{O}(n)$ time algorithm

Since a Gomory-Hu tree has a linear size representation, it is natural to ask if a linear or near-linear time Gomory-Hu tree algorithm exists. A main contribution of this thesis is a proof that such an algorithm indeed exists [5]. To obtain this result, we depart from the greedy approach above. Instead, we rely on a property inherent in the Gomory-Hu tree construction which allows us to consider cycles in any order. The addition of the first cycle splits the graph in two and we can consider the two subgraphs separately.

The problem with this approach is that we may get bad splits of the graph. In order to obtain efficient running time, we make use of the recursive subdivision of Fakcharoenphol and Rao to guide our algorithm. We start by separating faces incident to pieces at the bottom-level of the subdivision and then move up the tree. The faces that are hard to separate are at the higher levels of the tree. We show that when we reach these levels, not too many pairs of faces are left to separate.

This approach allows us to obtain in $O(n \log^6 n)$ time and $O(n \log n)$ space a tree defining the nesting of cycles in the basis for the dual graph. From this, we obtain a Gomory-Hu tree of the primal in additional linear time. We then make use of an algorithm due to Kaplan and Shafrir [18] to answer path weight queries in a tree in constant time with $O(n \log n)$ preprocessing time. Applying this algorithm to the Gomory-Hu tree, we obtain our min *st*-cut oracle. Total preprocessing time is $O(n \log^6 n)$ and space requirement is $O(n \log n)$.

3 Stretch Factor and Maximum Detour

Given a connected and undirected graph G with non-negative edge weights, a spanning subgraph S of G is called a *t-spanner* of G if for all pairs of vertices u and v in G, the shortest path distance $d_S(u, v)$ between u and v in S is at most t times longer than the shortest path distance $d_G(u, v)$ between them in G. We say that S has *stretch* t. Observe that $t \geq 1$.

In the following, we focus mainly on geometric graphs. Here, the underlying graph is the complete geometric graph on a given set P of points in the plane. A sparse spanner for P with small stretch can be viewed as a compact representation of all the pairwise distances for points in this set. Finding such a spanner that also keeps other cost measures low, like weight, degree, and diameter, is important in many areas including VLSI design, distributed computing, and robotics.

3.1 Stretch factor

We consider a sort of dual to the spanner problem. Here, we are given a connected geometric graph G and we need to find the smallest $t \geq 1$ such that G is a t-spanner of the underlying complete graph. We call this smallest value the *stretch factor* of G. An equivalent definition of stretch factor is the maximum, over all pairs of distinct input points p and q, of the ratio between the distance between p and q in G and the Euclidean distance $\|pq\|$.

Agarwal et al. [1] showed how to compute the stretch factor of paths, trees, and cycles in near-linear time. Cabello and Knauer [6] gave an $O(n \log^{k+1} n)$ expected time algorithm for geometric graphs with treewidth k.

An open problem is whether the stretch factor of a plane geometric graph can be computed in subquadratic time [1]. In our distance oracle paper [42], we show how to obtain an $O(n^2(\log\log n)^c/\log n)$ time algorithm, where c is a constant, thereby solving this open problem.

3.2 Maximum detour

We achieve a better time bound for the related maximum detour problem. Given a connected straight-line embedded plane geometric graph G, the *maximum detour* of G is defined as the stretch factor of G except that we consider all pairs of *points* of G, i.e., vertices as well as interior points of edges of G. This problem can be solved in $O(n^2)$ time but it was open whether a subquadratic time algorithm existed. We solve this open problem by exhibiting an algorithm with $O(n^{3/2}\log^3 n)$ running time [35].

In order to achieve this bound, we apply the separator theorem of Lipton and Tarjan [22] to separate our graph in two. The maximum detour of the two subgraphs is recursively computed. The shortest paths between the remaining pairs of points that we need to consider must cross the separator. Furthermore, a property of maximum detour allows us to restrict our attention to pairs of points belonging to the same face. We obtain colourings of the points on each face, defined by the separator vertices that shortest paths must go through. From these colourings, we can reduce the problem to a number of maximum detour problems for trees and efficiently solve them with the algorithm in [1].

A more careful analysis reveals that the running time of our algorithm is $O(nk\log^3 n)$, where k is the separator size. Since graphs of bounded treewidth have bounded size separators, we thus obtain an $O(n\log^3 n)$ time algorithm if the input graph has bounded treewidth.

3.3 Best shortcuts

Most algorithms construct networks from scratch, but frequently one is interested in extending an already given network with a number of edges such that the stretch factor of the resulting network is minimized.

Farshi et al. [11] considered the following problem: given a graph $G = (V, E)$ with m edges and n vertices embedded in a metric space, find a vertex pair $(u, v) \in V^2$ (called a shortcut) such that the stretch factor of $G \cup \{(u, v)\}$ is minimized. They gave an $O(n^4)$ time and $O(n^2)$ space algorithm for this problem together with various approximation algorithms.

We show how to improve running time to $O(n^3 \log n)$ while maintaining quadratic space requirement [34]. In fact, our algorithm not only computes the best shortcut but the stretch factor of every edge obtained from G by a single edge-augmentation.

The main idea of the algorithm is the following. Instead of explicitly considering all pairs of endpoints of the edge to be added to G, we fix only one endpoint and parameterize the position of the other. We show that the information required to obtain the stretch factors of edge-augmented graphs for all parameter choices can be represented as a set of upper envelopes of piecewise linear functions. Calculating the stretch factor for all parameter values corresponding to the feasible positions of the other edge endpoint can then be efficiently obtained as upper envelope function values. As a result, we shave off almost a linear factor in running time compared to the algorithm in [11].

We conjecture that our algorithm is near-optimal for the following reason. Computing all-pairs shortest path distances in the input graph seems to be necessary to solve the problem. The fastest all-pairs shortest path algorithm for general graphs runs in time cubic in n (ignoring log-factors) and it is conjectured that this cannot be significantly improved.

The best known lower bound for the best shortcut problem is only $\Omega(n^2)$ so a gap of more than a linear factor remains.

An open problem asked in [11] is whether there exists a linear-space algorithm with $o(n^4)$ running time. The algorithm described above runs in $O(n^3 \log n)$ time but requires $O(n^2)$ space. We show how to obtain a trade-off between time and space in this algorithm to obtain a linear-space algorithm with $O((n^4 \log n)/\sqrt{m})$ time [23]. Since we may assume that G consists of at most two connected components (otherwise, no single edge can connect the graph and the problem is trivial), $m = \Omega(n)$ and we solve this open problem.

3.4 Weighted fixed orientation metrics

The algorithm in [1] for computing the stretch factor of paths, trees, and cycles is somewhat involved and only runs in low *expected* running time.

Complicated parametric search techniques are relied on to also obtain low worst-case running time but at the cost of some extra log-factors in running time.

We consider the same problem but for *weighted fixed orientation metrics* [31] in the plane. Each such metric is defined by a set of weighted fixed orientations and the distance between two points is the length of a shortest path between them consisting of line segments with orientations from this set; each segment is weighted by the weight of its orientation.

These metrics have received attention due to their application in VLSI design, where wires on a chip are typically restricted to having a small number of orientations. It seems natural to consider the stretch factor problem in this setting since low-stretch networks are well-connected and may be desirable in VLSI design.

We give an $O(\sigma n \log^2 n)$ time algorithm to find the stretch factor of an n-vertex path, where σ is the number of fixed orientations [36]. For the L_1-metric (a special type of fixed orientation metric), we generalize our algorithm to d dimensions, where the running time is $O(n \log^d n)$. At the cost of an extra log-factor in running time, we can find the stretch factor of trees and cycles as well. Our algorithms do not rely on any advanced data structures or techniques as in the Euclidean metric and should be relatively simple to implement.

4 Steiner Trees in Fixed Orientation Metrics

The *Euclidean Steiner tree problem* is a classical geometric problem dating back to Fermat in the 17th century [17]. It asks for a minimum length tree spanning a given set of points in the Euclidean plane. Such a tree is called a Steiner minimal tree. This problem differs from the MST problem in that new points may be added to shorten the tree. To distinguish between the two types of points, given points are referred to as *terminals* and new points are called *Steiner points*. The Euclidean Steiner tree problem is known to be NP-hard.

As mentioned earlier, (weighted) fixed orientation metrics have received attention in later years due to their use in VLSI design. A good candidate network of wires interconnecting a set of pins on a chip is one of minimal length. For this reason, the Steiner tree problem for fixed orientation metrics is well-studied from an algorithmic point of view.

4.1 Bounding the number of full Steiner trees

Unfortunately, the Steiner tree problem for fixed orientation metrics is also NP-hard (in fact NP-complete). Yet, exact algorithms exist that can solve even very large instances. Perhaps the most powerful algorithm is GeoSteiner [29] that solves many instances consisting of several thousand terminals in a reasonable amount of time for the Euclidean and fixed orientation metrics.

The GeoSteiner algorithm uses a two-phase approach to solve the problem. In the first phase, a set of so called full components is generated. Full components are Steiner minimal trees for terminal subsets with the requirement that all leaves are terminals and all interior vertices are Steiner points. Every Steiner minimal tree can be partitioned into full components and they can therefore be regarded as building blocks to form such a tree.

The first phase generates a set of full components guaranteed to contain a subset defining the full components of a Steiner minimal tree. Various pruning techniques are applied to keep the size of this set small. In a second phase, full components from this set are concatenated to form a Steiner minimal tree.

Experimental results suggest that for n terminals randomly distributed with uniform distribution in a unit square, the number of full components generated in the first phase is only $O(n)$. This can help explain why GeoSteiner is so powerful. A matching theoretical bound has not been shown, however. What has been shown is that, for any $K > 2$, the expected number of full components generated which span exactly K terminals in the L_1-plane is $O(n(\log\log n)^{K-2})$ [43].

Our contribution is an improvement of this bound to $O(n\pi^K)$ which reduces to $O(n)$ for fixed K [33]. We also give bounds for full components in higher dimensions.

A linear bound on the total number of full components generated is yet to be found. We managed to reduce this problem to one of proving that the value of a certain integral is less than one. Unfortunately, the domain of integration is extremely complex and of high dimension and we estimated that on a modern pc, it would take thousands of years to compute the exact value of this integral. Experiments suggest that the value is indeed less than one and we hope that one day, finding a computer-assisted proof is feasible.

4.2 Steiner hull

A way to speed up the computation of a Steiner minimal tree in the plane is to first compute a subset of the plane guaranteed to contain such a tree. Such a set is called a *Steiner hull*. If a small Steiner hull can be obtained, it will give some structural properties of a solution which can then be used to speed up the computation.

Winter [32] considered a certain type of Steiner hull in the Euclidean plane. It is formed by starting with the convex hull of the set of terminals and then repeatedly cutting off triangles from this set satisfying certain conditions. He proved that all triangles are part of a Delaunay triangulation and obtained an $O(n \log n)$ time algorithm to find the Steiner hull.

We consider a similar type of Steiner hull in the *uniform orientation metrics*, a subclass of the class of fixed orientation metrics, where orientations are uniformly distributed [40]. As in the Euclidean plane, we show that this Steiner hull can be obtained by starting with (a superset of) the convex hull of the set of terminals and then repeatedly cutting triangles from this set. We cannot use the Delaunay triangulation to identify the triangles however. Instead, we use a Euclidean MST to separate the problem. We prove that the total time to solve the subproblems is $O(\sigma n \log n)$, where σ is the number of uniform orientations.

Experimental results, which are not included in this thesis, suggest that for the octilinear metric in particular (where horizontal, vertical, and diagonal orientations are allowed), rather tight Steiner hulls are obtained for many smaller point sets and we hope that these can be used to obtain faster Steiner tree algorithms.

5 Concluding Remarks

We presented improved algorithms for a number of planar graph problems. In particular, we gave better time bounds for computing shortest paths, replacement paths, and various graph invariants, and we gave distance and min cut oracles with better preprocessing time and space bounds. For geometric graphs, we presented faster algorithms for computing the stretch factor and maximum detour, and for graphs embedded in metric spaces, we showed how to efficiently augment them with new edges while minimizing stretch factor. Finally, we considered the Steiner tree problem in the plane equipped with a weighted fixed orientation metric. We gave an algorithm to compute a Steiner hull and we gave a stronger theoretical analysis of the strength of pruning techniques applied by Steiner tree algorithms such as GeoSteiner.

We see several directions for future research. For planar graphs, current focus is on the min st-cut problem. The best known bound is $O(n \log n)$. We believe that $O(n \log \log n)$ time is achievable. For the replacement paths problem, is $\Theta(n \log n)$ the true complexity? Since single-source shortest path distances in planar graphs with non-negative edge weights can be computed in linear time, it would not surprise us if the replacement paths problem also admits an $O(n)$ time algorithm. For the distance oracle, Wiener index, and diameter problems, is there a "truly" subquadratic time algorithm, i.e., an algorithm with $O(n^c)$ running time for constant $c < 2$? We believe so, at least for the diameter problem.

For the stretch factor problems, it would be interesting to extend the edge-augmenting algorithm to handle not just one but, say, a constant number of edges. However, it seems difficult to apply our ideas to this more general problem. For the maximum detour resp. stretch factor problem for geometric plane graphs, we conjecture that a near-linear resp. truly subquadratic time algorithm exists.

For the Steiner tree problem, we would like to prove a linear bound on the number of full components generated by GeoSteiner. Due to lack of computing power, our current approach of a computer-assisted proof involving the computation of a very complicated integral is infeasible. We hope that a better analysis can lead to a simpler integral that may be solvable now or in the near future.

References

[1] P. K. Agarwal, R. Klein, C. Knauer, S. Langerman, P. Morin, M. Sharir, and M. Soss. Computing the Detour and Spanning Ratio of Paths, Trees and Cycles in 2D and 3D. Discrete and Computational Geometry, 39 (1): 17–37 (2008).

[2] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully-dynamic trees with top trees. ACM Transactions on Algorithms (TALG), Volume 1, Issue 2 (October 2005).

[3] E. Amaldi, C. Iuliano, T. Jurkiewicz, K. Mehlhorn, and R. Rizzi. Breaking the $O(m^2n)$ Barrier for Minimum Cycle Bases. A. Fiat and P. Sanders (Eds.): ESA 2009, LNCS 5757, pp. 301–312, 2009.

[4] G. Borradaile and P. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. Journal of the ACM, 56(2):1–30, 2009.

[5] G. Borradaile and C. Wulff-Nilsen. Min *st*-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. Manuscript, 2010 (**Paper [A]**).

[6] S. Cabello and C. Knauer. Algorithms for graphs of bounded treewidth via orthogonal range searching. Computational Geometry, Volume 42, Issue 9, November 2009, Pages 815–824.

[7] F. R. K. Chung. Diameters of Graphs: Old Problems and New Results. Congressus Numerantium, 60:295–317, 1987.

[8] Y. Emek, D. Peleg, and L. Roditty. A Near-Linear Time Algorithm for Computing Replacement Paths in Planar Directed Graphs. In SODA'08: Proceedings of the Nineteenth Annual ACM-SIAM symposium on Discrete Algorithms, pages 428–435, Philadelphia, PA, USA, 2008, Society for Industrial and Applied Mathematics.

[9] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. Journal of Graph Algorithms and Applications, 3(3):1–27, 1999.

[10] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. Journal of Computer and System Sciences, Volume 72, Issue 5, August 2006, Pages 868–889, Special Issue on FOCS 2001.

[11] M. Farshi, P. Giannopoulos, and J. Gudmundsson. Finding the Best Shortcut in a Geometric Network. 21st Ann. ACM Symp. Comput. Geom. (2005), pp. 327–335.

[12] G. N. Frederickson Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput., 16 (1987), pp. 1004–1022.

[13] R. Gomory and T. C. Hu. Multi-terminal network flows. J. SIAM, 9 (1961), pp. 551–570.

[14] Z. Gotthilf and M. Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. Information Processing Letters, Vol. 109, 7/2009, Pages 352–355.

[15] D. Hartvigsen and R. Mardon. The All-Pairs Min Cut Problem and the Minimum Cycle Basis Problem on Planar Graphs. SIAM J. Discrete Math. Volume 7, Issue 3, pp. 403–418 (May 1994).

[16] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. Journal of Computer and System Sciences, 55(1):3–23, 1997.

[17] F. K. Hwang, D. S. Richards, and P. Winter. The Steiner Tree Problem. North-Holland, 1992.

[18] H Kaplan and N. Shafrir. Path Minima in Incremental Unrooted Trees. Proceedings of the 16th annual European Symposium on Algorithms, Lecture Notes in Computer Science, Vol. 5193, 2008, pp. 565–576.

[19] P. N. Klein. Multiple-source shortest paths in planar graphs. Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 146–155.

[20] P. N. Klein, S. Mozes, and O. Weimann. Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n \log^2 n)$-Time Algorithm. Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms, p. 236–245, 2009.

[21] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. SIAM Journal on Numerical Analysis, 16:346–358, 1979.

[22] R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. STAN-CS-77-627, October 1977.

[23] J. Luo and C. Wulff-Nilsen. Computing Best and Worst Shortcuts of Graphs Embedded in Metric Spaces. In S.-H. Hong, H. Nagamochi, and T. Fukunaga (Eds.): ISAAC 2008, LNCS 5369, pp. 764–775, 2008 (**Paper [B]**).

[24] K. Malik, A. K. Mittal, and S. K. Gupta. The k most vital arcs in the shortest path problem. Operations Research Letters, 8(4):223–227, 1989.

[25] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.

[26] S. Mozes and C. Wulff-Nilsen. Shortest Paths in Planar Graphs with Real Lengths in $O(n \log^2 n / \log \log n)$ Time. Manuscript, 2010 (**Paper [C]**).

[27] E. Nardelli, G. Proietti, and P. Widmayer. A faster computation of the most vital edge of a shortest path. Information Processing Letters, 79 (2): 81–85, 2001.

[28] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. FOCS (2001), pp. 242–251.

[29] D. M. Warme, P. Winter, and M. Zachariasen. GeoSteiner 3.1. Department of Computer Science, University of Copenhagen, `http://www.diku.dk/geosteiner/`, 2001.

[30] O. Weimann and R. Yuster. Computing the Girth of a Planar Graph in $O(n \log n)$ time. In Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009).

[31] P. Widmayer, Y. F. Wu, and C. K. Wong. On Some Distance Problems in Fixed Orientations. SIAM J. Comput. Volume 16, Issue 4, pp. 728–746 (1987).

[32] P. Winter. Optimal Steiner hull algorithm. Computational Geometry 23 (2002) 163–169.

[33] C. Wulff-Nilsen. Bounding the Expected Number of Rectilinear Full Steiner Trees. To appear in Networks (**Paper [D]**).

[34] C. Wulff-Nilsen. Computing the dilation of edge-augmented graphs in metric spaces. Computational Geometry, Volume 43, Issue 2, February 2010, Pages 68–72, Special Issue on the 24th European Workshop on Computational Geometry (**Paper [E]**).

[35] C. Wulff-Nilsen. Computing the Maximum Detour of a Plane Graph in Subquadratic Time. In S.-H. Hong, H. Nagamochi, and T. Fukunaga (Eds.): ISAAC 2008, LNCS 5369, pp. 740–751, 2008 (**Paper [F]**).

[36] C. Wulff-Nilsen. Computing the Stretch Factor of Paths, Trees, and Cycles in Weighted Fixed Orientation Metrics. Proc. 20th Canadian Conference on Computational Geometry (CCCG 2008), Montreal, Canada, 2008, p. 59–62 (**Paper [G]**).

[37] C. Wulff-Nilsen. Girth of a Planar Digraph with Real Edge Weights in $O(n \log^3 n)$ Time. arXiv:0908.0697v1 [cs.DM], August 2009 (**Paper [H]**).

[38] C. Wulff-Nilsen. Minimum Cycle Basis and All-Pairs Min Cut of a Planar Graph in Subquadratic Time. Manuscript, 2010 (**Paper [I]**).

[39] C. Wulff-Nilsen. Solving the Replacement Paths Problem for Planar Directed Graphs in $O(n \log n)$ Time. In proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, p. 756–765, 2010 (**Paper [J]**).

[40] C. Wulff-Nilsen. Steiner hull algorithm for the uniform orientation metrics. Computational Geometry - Theory and Applications, Vol. 40/1, May 2008, pp. 1–13 (**Paper [K]**).

[41] C. Wulff-Nilsen. Wiener Index and Diameter of a Planar Graph in Subquadratic Time. Proc. 25th European Workshop on Computational Geometry, Brussels, 2009, p. 25–28 (selected for special issue of Computational Geometry) (**Paper [L]**).

[42] C. Wulff-Nilsen. Wiener Index, Diameter, and Stretch Factor of a Weighted Planar Graph in Subquadratic Time. Manuscript, 2010 (**Paper [M]**).

[43] M. Zachariasen. Rectilinear Full Steiner Tree Generation. Networks, 33: 125-143, 1999.

Summary (in Danish)

Algoritmer for netværksproblemer spiller en stadig større rolle i det moderne samfund. Et netværks grafstruktur er en abstrakt og meget nyttig repræsentation, som gør det muligt at anvende klassiske graf-algoritmer såsom Dijkstra og Bellman-Ford. Netværk fra virkelighedens verden har ofte andre strukturelle egenskaber, som kan udnyttes. For eksempel er et vejnet eller wire-layoutet på en mikrochip typisk (næsten) planart og afstande i netværket er ofte defineret ved den Euklidiske eller den rektilineære metrik. Specialiserede algoritmer, der udnytter sådanne egenskaber, er ofte væsentligt hurtigere end de tilsvarende algoritmer for generelle grafer.

I den første og primære del af denne afhandling fokuseres på udviklingen af effektive algoritmer for planare grafer. De vigtigste bidrag er en hurtigere algoritme for korteste-vej-problemet, et afstands-orakel med subkvadratisk præprocesseringstid, en $O(n \log n)$-tids-algoritme for replacement paths-problemet og et min st-cut-orakel med næsten lineær præprocesseringstid. Vi opnår desuden bedre køretider til bestemmelse af forskellige graf-invarianter såsom diameter og girth.

I den anden del betragtes stretch factor-problemer for geometriske grafer samt grafer indlejret i metriske rum. Stretch factor er groft sagt en reel værdi, der angiver, hvor godt en (geo-)metrisk graf tilnærmer den underliggende komplette graf mht. afstande. Vi præsenterer forbedrede algoritmer til bestemmelse af stretch factor af en given graf og til at udvide en graf med nye kanter, således at stretch factor minimeres.

I den tredje og sidste del af afhandlingen behandles Steiner-træ-problemet i planen udstyret med en vægtet fixed orientation-metrik. Vi giver en forbedret teoretisk analyse af styrken ved pruning-teknikker anvendt af flere Steiner-træ-algoritmer. Vi præsenterer desuden en algoritme, der bestemmer et såkaldt Steiner hull, en struktur, der kan gøre bestemmelsen af et minimalt Steiner-træ nemmere.

Research Papers

[**A**] G. Borradaile and C. Wulff-Nilsen. Min *st*-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. Manuscript, 2010.

[**B**] J. Luo and C. Wulff-Nilsen. Computing Best and Worst Shortcuts of Graphs Embedded in Metric Spaces. In S.-H. Hong, H. Nagamochi, and T. Fukunaga (Eds.): ISAAC 2008, LNCS 5369, pp. 764–775, 2008.

[**C**] S. Mozes and C. Wulff-Nilsen. Shortest Paths in Planar Graphs with Real Lengths in $O(n \log^2 n / \log \log n)$ Time. Manuscript, 2010.

[**D**] C. Wulff-Nilsen. Bounding the Expected Number of Rectilinear Full Steiner Trees. To appear in Networks.

[**E**] C. Wulff-Nilsen. Computing the dilation of edge-augmented graphs in metric spaces. Computational Geometry, Volume 43, Issue 2, February 2010, Pages 68–72, Special Issue on the 24th European Workshop on Computational Geometry.

[**F**] C. Wulff-Nilsen. Computing the Maximum Detour of a Plane Graph in Subquadratic Time. In S.-H. Hong, H. Nagamochi, and T. Fukunaga (Eds.): ISAAC 2008, LNCS 5369, pp. 740–751, 2008.

[**G**] C. Wulff-Nilsen. Computing the Stretch Factor of Paths, Trees, and Cycles in Weighted Fixed Orientation Metrics. Proc. 20th Canadian Conference on Computational Geometry (CCCG 2008), Montreal, Canada, 2008, p. 59–62.

[**H**] C. Wulff-Nilsen. Girth of a Planar Digraph with Real Edge Weights in $O(n \log^3 n)$ Time. arXiv:0908.0697v1 [cs.DM], August 2009.

[**I**] C. Wulff-Nilsen. Minimum Cycle Basis and All-Pairs Min Cut of a Planar Graph in Subquadratic Time. Manuscript, 2010.

[**J**] C. Wulff-Nilsen. Solving the Replacement Paths Problem for Planar Directed Graphs in $O(n \log n)$ Time. In proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, p. 756–765, 2010.

[**K**] C. Wulff-Nilsen. Steiner hull algorithm for the uniform orientation metrics. Computational Geometry - Theory and Applications, Vol. 40/1, May 2008, pp. 1–13.

[**L**] C. Wulff-Nilsen. Wiener Index and Diameter of a Planar Graph in Subquadratic Time. Proc. 25th European Workshop on Computational Geometry, Brussels, 2009, p. 25–28 (selected for special issue of Computational Geometry).

[**M**] C. Wulff-Nilsen. Wiener Index, Diameter, and Stretch Factor of a Weighted Planar Graph in Subquadratic Time. Manuscript, 2010.

Min st-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time

Glencora Borradaile[*] Christian Wulff-Nilsen [†]

March 13, 2010

Abstract

For an undirected n-vertex planar graph G with non-negative edge-weights, we consider the following type of query: given two vertices s and t in G, what is the weight of a min st-cut in G? We show how to answer such queries in constant time with $O(n \log^4 n)$ preprocessing time and $O(n \log n)$ space. We use a Gomory-Hu tree to represent all the pairwise min cuts implicitly. Previously, no subquadratic time algorithm was known for this problem. Since all-pairs min cut and the minimum cycle basis are dual problems in planar graphs, we also obtain an implicit representation of a minimum cycle basis in $O(n \log^4 n)$ time and $O(n \log n)$ space and an explicit representation with additional $O(C)$ time and space where C is the size of the basis.

These results require that shortest paths be unique. We deterministically remove this assumption with an additional $\log^2 n$ factor in the running time.

1 Introduction

A minimum cycle basis is a minimum-cost representation of all the cycles of a graph and the all-pairs min cut problem asks to find all the minimum cuts in a graph. In planar graphs the problems are intimately related (in fact, equivalent [7]) via planar duality. We give the first sub-quadratic algorithm for these problems, running in $O(n \log^4 n)$ time. In the following, we consider undirected, graphs with non-negative edge weights.

All-pairs min cut The *all-pairs min cut problem* is to find the minimum st-cut for every pair s, t of vertices in a graph G. Gomory and Hu [6] showed that these minimum cuts can be represented by an edge-weighted tree such that:

- the nodes of the tree correspond one-to-one with the vertices of G,
- for any distinct vertices s and t, the minimum edge weight on the unique simple s-to-t path in the tree has weight equal to the min st-cut weight in G, and
- removing the corresponding minimum weight edge from the tree partitions the nodes into two sets S and T that is a partition of the vertices in G corresponding to a min st-cut.

[*]School of Electrical Engineering and Computer Science, Oregon State University, `glencora@eecs.orst.edu`, `www.glencora.org`

[†]Department of Computer Science, University of Copenhagen, `koolooz@diku.dk`, `http://www.diku.dk/hjemmesider/ansatte/koolooz/`

We call such a tree a *Gomory-Hu* tree or GH tree. Gomory and Hu also showed how to find such a tree with $n-1$ calls to a minimum cut algorithm. To date, this is the best known method for general graphs and results in an $O(n^2 \log n)$-time algorithm for planar graphs using the best-known algorithm for min *st*-cuts in planar graphs[8, 9, 4]. There is an algorithm for unweighted graphs that beats the $n-1$ times minimum cut time bound [3].

Minimum cycle basis A cycle basis of a graph is a set of independent cycles. Viewing a cycle as an incidence vector in $\{0,1\}^E$, a set of cycles is independent if their vectors are independent over $GF(2)$. The weight of a set of cycles is the sum of the weights of the cycles. The *minimum-cycle basis (MCB) problem* is to find a cycle basis of minimum weight. This problem dates to the electrical circuit theory of Kirchhoff [13] in 1847 and has been used in the analysis of algorithms by Knuth [15]. For a complete survey, see [10]. The best known algorithm in general graphs takes time $O(m^\omega)$ where ω is the exponent for matrix multiplication [2].

An embedded planar graph is a mapping of the vertices to distinct points and edges to non-crossing curves in the plane. A face of the embedded planar graph is a maximal open connected set of points that are not in the image of any embedded edge or vertex. Exactly one face is unbounded and it is called the infinite face. We identify a face with the embedded vertices and edges on its boundary.

For a simple cycle C in a planar embedded graph G, let $int(C)$ resp. $ext(C)$ denote the open bounded resp. unbounded subset of the plane defined by C. We refer to the closure of these sets as $\overline{int}(C)$ and $\overline{ext}(C)$, respectively. We say that a pair of faces of G are *separated* by C in G and that C *separates* this pair if one face is contained in $\overline{int}(C)$ and the other face is contained in $\overline{ext}(C)$. A set of simple cycles of G is called *nested* if, for any two distinct cycles C and C' in that set, either $int(C) \subset int(C')$, $int(C') \subset int(C)$, or $int(C) \subset ext(C')$.

Hartvigsen and Mardon [7] prove that if G is planar, then there is a minimum cycle basis whose cycles are simple and nested in the drawing in the embedding. As such, one can represent a minimum cycle basis of a planar embedded graph as an edge-weighted tree such that:

- the nodes of the tree correspond one-to-one with the faces of the planar embedded graph, and
- each edge in the tree corresponds to a cycle in the basis, namely the cycle that separates the faces in the components resulting from removing said edge from the tree.

Hartvigsen and Mardon also gave an $O(n^2 \log n)$-time algorithm for the problem that was later improved to $O(n^2)$ by Amaldi et. al. [2].

1.1 Planar duality

In planar graphs, the MCB and GH problems are related via planar duality.

Corresponding to every connected planar embedded graph G (the *primal*) there is another connected planar embedded graph (the *dual*) denoted G^*. The faces of G are the vertices of G^* and vice versa. The edges of G correspond one-to-one with those of G^*. Cycles and cuts are equivalent through duality:

> In a connected planar graph, a set of edges forms a cycle in the primal iff it forms a cut in the dual. [18]

Equivalence between minimum cycle bases and GH trees Just as cuts and cycles are intimately related via planar duality, so are the all-pairs min cut and minimum cycle basis problems. In fact, Hartvigsen and Mardon showed that they are equivalent in the following sense:

Theorem 1. *For a planar embedded graph G, a tree T represents a minimum cycle basis of G if and only if T is a Gomory-Hu tree for G^* (after mapping the node-face relationship to a node-vertex relationship via planar duality). (Corollary 2.2 [7])*

Herein, we focus on the frame of reference of the minimum cycle basis. Our algorithm works by finding a minimum (weight) cycle that separates two faces f and g. By duality, this cycle is a min fg-cut in G^*. We recurse on as-yet unseparated faces, gradually building the tree T that represents the minimum cycle basis and is the also GH-tree (for the dual graph). This alone will not achieve a sub-quadratic running time. In order to beat quadratic time, we guide the recursion with planar separators and use precomputed distances to efficiently find the minimum separating cycles.

1.2 Planar separators

A *decomposition* of a graph G is a set of subgraphs $P_1, \ldots, P_k$ such that the union of vertex sets of these subgraphs is the vertex set of G and such that every edge of G is contained in a unique subgraph. We call $P_1, \ldots, P_k$ the *pieces* of the decompostion. The *boundary vertices* of a piece P_i is the set of vertices u in that piece such that there exists an edge (u, v) in G with $v \notin P_i$

By recursive application of Miller's Cycle Separator Theorem [16] to a planar embedded graph G, we obtain a recursive subdivision where at each level, a piece with r vertices and s boundary vertices is divided into two subpieces each of which has at most $2r/3$ vertices and at most $2s/3+c\sqrt{r}$ boundary vertices, for some constant c. The recursion stops when only one edge remains in a piece. We define the $O(\log n)$ *levels* of the recursive decomposition in the natural way: level 0 consists of one piece (G) and level i-pieces are obtained by applying the Cycle Separator Theorem to each level $i-1$-piece. We represent the recursive subdivision as a binary tree, called the *subdivision tree (of G)*, with level i-pieces corresponding to vertices at level i in tree. Parent/child and ancestor/descendant relationships between pieces correspond to their relationships in the subdivision tree.

Fakcharoenphol and Rao [5] show how to find a recursive subdivision such that for each piece, its boundary vertices belong to a constant number of faces. We shall make the assumption that all boundary vertices are on the external face of the piece. This will simplify the description of the algorithm. Using results from [5], our results can easily be extended to the general case.

Dense distance graphs For a piece P, the *internal dense distance graph of P* or *int*DDG(P) is the complete graph on the set of boundary vertices of P, where the weight of each edge (u, v) is equal to the shortest path distance between u and v in P. The union of internal dense distance graphs of all pieces in the recursive subdivision of G is the *internal dense distance graph (of G)*, or simply *int*DDG. Fakcharoenphol and Rao showed how to compute *int*DDG in $O(n \log^3 n)$ time [5]; Klein improved this to $O(n \log^2 n)$ [14].

The *external dense distance graph of P* or *ext*DDG(P) is the complete graph on the set of boundary vertices of P where the weight of an edge (u, v) is the shortest path distance between u and v in $G \setminus E(P)$. The *external dense distance graph of G* or *ext*DDG is the union of all external dense distance graphs of the pieces in the recursive subdivision of G.

Theorem 2. *The external dense distance graph of G can be computed in $O(n \log^3 n)$ time.*

Proof. Fakcharoenphol and Rao compute *int*DDG bottom-up by applying a variant of Dijkstra to obtain *int*DDG(P) for a piece P from the internal dense distance graphs of its two children. A similar algorithm can compute *ext*DDG(P) from the external dense distance graph of the parent of P and the internal dense distance graph of the sibling of P. Hence, we can obtain *ext*DDG with a top-down algorithm after having found *int*DDG. Running time matches the time to find *int*DDG. □

1.3 Overview of the algorithm

We gradually build up the tree representing the minimum cycle basis. Initially the tree is a star centered at a root r and each leaf corresponding to a face in the graph (including the infinite face). We update the tree to reflect the cycles that we add iteratively to the basis. When the first cycle C is found, we create a new node x_C for the tree, make C a child of the root and make all the faces that C encloses children of C. Each non-face node in the tree corresponding to a cycle C defines a *region* R, defined as the subgraph of G contained in the closed subset of the plane defined by the interior of C and the exterior of the children (if any) of C. We say that R is *bounded* by C, that C is a *bounding cycle* of R, and that R *contains* the child regions and/or child faces defined by the tree. The tree of regions is called the *region tree.* We observe the following:

A pair of faces not yet separated by a basis cycle belong to the same region.

The root r will remain a special region that represents the entire plane. We only add cycles to the basis that nest with the cycles found so far. Whenever the basis is updated, the region tree is updated accordingly. This is illustrated in Figure 1. We show how to efficiently update the region tree in Section 5.

In the final tree, all faces have been separated: each face is the only face-child of a region. We call such a region tree a *complete region tree.* Mapping each face to its parent, creating a tree with one node for each face in the graph, will create the tree representing the minimum cycle basis.

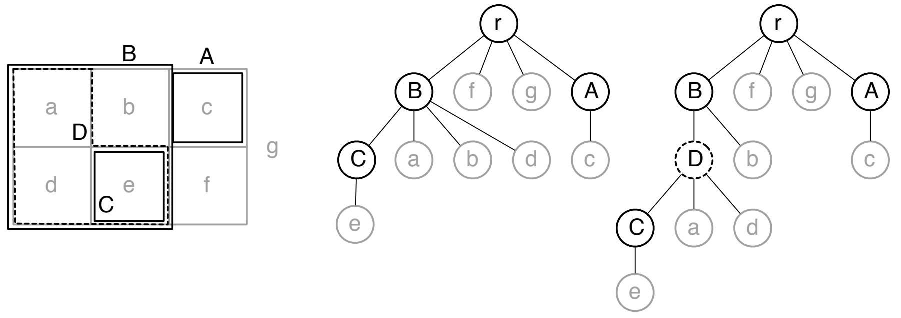


Figure 1: A graph with faces a through g; four nesting cycles A through D (left). A region tree for cycles A,B and C (center). A region tree for cycles A through D (right).

Our algorithm is guided by the recursive subdivision of G. Starting at the deepest level of the recursive subdivision, we separate all pairs of faces of G that have an edge in a common piece of the subdivision. Each (nesting) separating cycle is added to the region tree.

Suppose we are at level i and that our algorithm has been applied to all levels deeper than i: every pair of faces that have an edge in a common subpiece have been separated. Consider some level i-piece P and suppose two faces f_1 and f_2 of G both sharing edges with P have not yet been separated. Let P_1 and P_2 be the two child pieces of P in the recursive subdivision. Since all pairs of faces sharing edges with P_1 and all pairs of elementary faces sharing edges with P_2 have already been separated, w.l.o.g. f_1 shares edges with P_1 and f_2 shares edges with P_2.

Since f_1 and f_2 have not been separated, they must belong to a common region R in the region tree. There cannot be any other pair of faces f_1' and f_2' in R which share edges with P. For otherwise, assume w.l.o.g. that f_1' and f_2' share edges with P_1 and P_2, respectively. Since all pairs of faces sharing edges with a common child of P have already been separated and since f_1, f_2, f_1', and f_2' all belong to R, $f_1' = f_1$ and $f_2' = f_2$, a contradiction.

Let the *region subpieces* of P be the subgraphs defined by the non-empty intersections between P and regions defined by the region tree. We say that a region R is *associated* with a region subpiece P_R and that P_R is associated with R if $P_R = P \cap R$ is not empty. The boundary nodes ∂P_R of P_R are the boundary nodes that are inherited from P. The above shows that in each region subpiece P_R of P, at most one pair of faces need to be separated. These constructs are illustrated in Figure 2. In Section 4, we show how to separate such a pair in time $O(|P_R| \log^3 |P_R|)$ and show that this amounts to $O(|P| \log^3 |P|)$ time over all region subpieces of P. This will imply that our algorithm spends $O(n \log^3 n)$ time at level i in the recursive decomposition. Summing over all levels, this gives a total running time of $O(n \log^4 n)$.

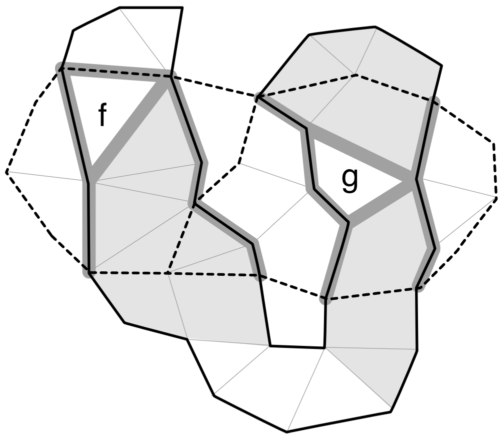


Figure 2: The dotted edges are the edges belonging to the boundaries of a piece P and P's children P_1 and P_2. f is a face of P_1 and g is a face of P_2. The solid black edges are the bounding cycle of a region R. The three shaded regions are three child regions of R. The thick grey edges are the edges of region subpiece P_R. (The remaining edges of the graph are the thin, grey edges.) The intersection of a minimum separating cycle for f and g with P uses only edges of P_R.

1.4 Results

The bulk of the paper will be devoted to presenting the details of the *region tree algorithm* outlined in the previous section. In Section 3, we will show how to find the region subpieces that contain unseparated faces. In Section 4, we will show how to find the minimum separating cycle for each of those pairs of unseparated faces. In Section 5, we will show how to update the region tree given that separating cycle. Together these three sections will prove the main result of our paper:

Theorem 3. *A complete region tree of a planar undirected n-vertex graph with non-negative edge weights can be found in $O(n \log^4 n)$ time and $O(n \log n)$ space.*

The region tree algorithm has several uses as previously indicated. For example, if we wish to compute the Gomory-Hu tree of G (assuming G is connected), first find the complete region tree T of G^* (this assumes that G^* is simple which can be achieved by adding new vertices on edges without increasing the asymptotic complexity of the problem). Since T is complete, every node x of T corresponding to a face of G^* is the only face-child of a node u in T. Contract, in T, the edge ux and identify the resulting node with face x. The resulting tree is exactly the tree that represents the minimum cycle basis of G^*. By Theorem 1, mapping the faces of G^* represented in this tree to their corresponding vertices in G builds the Gomory-Hu tree for G. This gives:

Theorem 4. *A Gomory-Hu tree of an n-vertex connected undirected planar graph with non-negative edge weights can be computed in $O(n \log^4 n)$ time and $O(n \log n)$ space.*

Given the Gomory-Hu tree, one finds a minimum st-cut by finding the minimum weight edge on the s-to-t path in the tree. With $O(n \log n)$ preprocessing time, on can answer such queries in $O(1)$ time using a tree-product data structure [12], giving:

Theorem 5. *With $O(n \log^4 n)$ time and $O(n \log n)$ space for preprocessing, the weight of a min st-cut between for any two given vertices s and t of an n-vertex planar, undirected graph with non-negative edge weights can be reported in constant time.*

In Section 6, we will show how to explicitly find the cycles given the complete region tree, giving the following results:

Theorem 6. *Without an increase in preprocessing time or space, the min st-cut oracle of Theorem 5 can be extended to report cuts in time proportional to their size.*

Theorem 7. *The minimum cycle basis of an undirected planar graph with non-negative edge weights can be computed in $O(n \log^4 n + C)$ time and $O(n \log n + C)$ space, where C is the total size of the cycles in the basis.*

Theorem 8. *The minimum cycle basis of an undirected and unweighted planar graph can be computed in $O(n \log^4 n)$ time and $O(n \log n)$ space.*

Proof. The faces of a planar embedded graph excluding the infinite face define a (not necessarily minimum) cycle basis with $O(n)$ edges. The result now follows from Theorem 7. □

2 Preliminaries

To simplify the presentation of the algorithm and the analysis, we make a few structural assumptions on the input graph. These assumptions are not truly restrictive.

2.1 Simplifying structural assumptions

Simple faces We assume that each face in the graph is simple: each vertex appears only once on the (boundary of the) face. We can achieve this by triangulating the graph with infinite-weight edges. This will simplify the faces. For each face f of the original graph, identify a face f' of the triangulated graph that is enclosed by f in the inherited embedding. The min $f'g'$-cut will not use any infinite-weight edge: the cut contains the same set of edges as in the min fg-cut in the original graph. Likewise, the set of cycles in a minimum cycle basis in the original graph are mapped to the set of finite-weight cycles in a minimum-cycle basis of the triangulated graph.

Degree three We assume that each vertex has degree 3. This can be achieved by triangulating the faces of the dual graph with ϵ weight edges, where ϵ is much smaller than the smallest weight edge. We use ϵ weight rather than 0 weight to aid in the next structural assumption. As in the above triangulation, we can map between minimum cycle bases and min cuts in the original graph and the degree-three graph. Triangulating the dual will increase the size of the faces of the primal, but the faces will remain simple: assuming that faces are simple and that vertices have degree three are compatible assumptions. Each vertex v in the original graph is mapped to a tree T_v of ϵ-weight edges in the degree-three graph.

Unique shortest paths We assume that between any pair of vertices there is a unique shortest path. To make this assumption a reality one can add a tiny, random fraction $\ll \epsilon$ to the weight of each edge and make the probability of having non-unique shortest paths arbitrarily small. Since this fraction is much smaller than the ϵ used above, we guarantee that any shortest path that enters and leaves a tree T_v of ϵ-weight edges does so exactly once. Theorem 11 in Section 7 gives a more robust, deterministic way to avoid this assumption that requires a $\log^2 n$-increase in the running time of our algorithm.

Every region is bounded by a cycle The root of the region tree described in Section 1.3 was a special region corresponding to the entire graph. We can treat this region as any other region by bounding the graph by a zero-weight cycle, and so treat every region as having a bounding cycle.

2.2 Isometric cycles

In a planar embedded graph, a simple cycle C is said to *cross* another simple cycle C' if $\{C, C'\}$ is not nested.

A cycle C in a graph is said to be *isometric* if for any two vertices $u, v \in C$, there is a shortest path in the graph between u and v which is contained in C. A set of cycles is said to be isometric if all cycles in the set are isometric. The following two results will prove useful.

Lemma 1. *Assume that shortest paths in a graph G are unique. The intersection between an isometric cycle and a shortest path in G is a (possibly empty) shortest path. The intersection between two distinct isometric cycles C and C' in G is a (possibly empty) shortest path; in particular, if G is a planar embedded graph, C and C' do not cross.*

Proof. Let C be an isometric cycle and let P be a shortest path intersecting C. Let u resp. v be the first resp. last vertex of intersection between C and P for some orientation of P. Since C is isometric, there is a shortest path P' in C between u and v. Since shortest paths are unique, P' is the subpath of P between u and v. Hence, the intersection between C and P is shortest path P'. This shows the first part of the lemma.

Let $C' \neq C$ be an isometric cycle. Then $C \cap C'$ is a set of vertex-disjoint paths. We claim that there can be at most one such path. To see this, let u and v be distinct vertices on paths P_u and

P_v. Let P_1 and P_2 be the two edge-disjoint paths in C between u and v. Define P'_1 and P'_2 similarly for C'. Since C and C' are isometric, we may assume that, say, P_1 and P'_1 are shortest paths. By uniqueness of shortest paths, $P_1 = P'_1$ so $P_u = P_v$, implying that $C \cap C'$ is a path. If we pick u and v as the first and last vertex on this path, it follows that $C \cap C'$ is a shortest path. □

Lemma 2. *Any minimum cycle basis of a graph is isometric. (Proposition 4.4 [7])*

It follows from these two lemmas that the minimum cycle basis that our algorithm constructs is isometric and nested.

2.3 Top trees

We will represent the region tree using the top tree data structure [1]. This will allow us to find the lowest common ancestor $lca(x, y)$ of two vertices x and y and determine whether one vertex is a descendant of another in logarithmic time and also to update the region tree efficiently as we add cycles to the basis (the latter is described in Section 5). We will also make use of the operation $\texttt{jump}(x, y, d)$, which for two vertices x and y in a top tree finds in logarithmic time the vertex of distance d (in terms of edges) from x on the simple path between x and y.

If the top tree represents a weighted tree, it can report the weight of a simple path in logarithmic time, given the endpoints of this path.

3 Finding region subpieces

In Section 1.3, we defined the region subpieces of a piece as the intersection between a region and a piece (Figure 2). In this section, we show how to identify the region subpieces and the edges that are in them. We start by identifying the set of regions $\mathcal{R}_P$ whose corresponding region subpieces of piece P each contain a pair of unseparated faces (Section 3.1). For each region $R \in \mathcal{R}_P$ we initialize the corresponding region subpiece P_R as an empty graph. For each edge e of P we determine to what region subpieces e belongs using least-common-ancestor and ancestor-descendent queries in the region tree (Section 3.2). In Section 3.3 we show how to do all this in $O(r \log^2 n)$ time where r is the size of P.

3.1 Identifying region subpieces

Since each edge is on the boundary of two faces, we start by marking all the faces of G that share edges with P in $O(r)$ time. Since a pair of unseparated faces in P are siblings in the region tree, we can easily determine the regions that contain unseparated faces. So, in $O(r)$ time we can identify $\mathcal{R}_P$, the set of regions with unseparated faces in P. We will need the following bound on the size of $\mathcal{R}_P$ in our analysis, illustrated in Figure 3.

Lemma 3. $|\mathcal{R}_P| = O(\sqrt{r})$.

Proof. Let S be the separator cycle applied to P to obtain its children P_1 and P_2. We may assume that P_1 is contained in $\overline{int}(S)$ and that P_2 is contained in $\overline{ext}(S)$. Let $\mathcal{F}_1$ be the set of faces containing edges of P_1 and not edges of P_2 and let $\mathcal{F}_2$ be the set of faces containing edges of P_2 and not edges of P_1.

Any region $R \in \mathcal{R}_P$ must contain at least one face of each $\mathcal{F}_1$ and $\mathcal{F}_2$. So, if C is the cycle bounding R, either $\overline{int}(C)$ contains S or C crosses S.

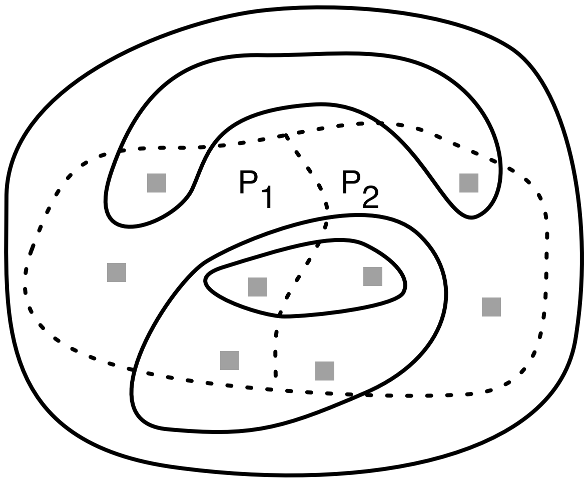


Figure 3: A piece P is given by the boundaries (dotted) by its two child pieces P_1 and P_2. $\mathcal{R}_P$ is a nesting set (solid cycles), each containing a pair of unseparated faces (grey). Since these faces must be separated by the child pieces, each bounding cycle (except for one outer cycle) in $\mathcal{R}_P$ must cross the dotted lines, resulting in a bound on $|\mathcal{R}_P|$.

If $\overline{int}(C)$ contains S then we claim that no other cycle bounding a region in $\mathcal{R}_P$ has this property. To see this, suppose for the sake of contradiction that there is another such cycle C' bounding a region $R' \in \mathcal{R}_P$. Since the cycles nest, either $\overline{int}(C) \subset \overline{int}(C')$ or $\overline{int}(C') \subset \overline{int}(C)$. Assume w.l.o.g. the former. Then all faces of $\mathcal{F}_1$ are contained in the interior of a face of R'. But this contradicts the assumption that R' contains at least one face from $\mathcal{F}_1$.

We may therfore restrict our attention to regions $R \subseteq \mathcal{R}_P$ whose bounding cycle C crosses S. Let $\mathcal{C}$ be the set of all cycles bounding regions of $\mathcal{R}_P$ and let $\mathcal{T}$ be the corresponding region tree. We will show that every region $R \in \mathcal{R}_P$ that has at most one child region must contain at least one face of $\mathcal{F}_1 \cap \mathcal{F}_2$. Since each face $\mathcal{F}_1 \cap \mathcal{F}_2$ is adjacent to at least one vertex of S and since each vertex can be adjacent to at most three faces (since G has degree 3), $|\mathcal{F}_1 \cap \mathcal{F}_2| = O(|S|) = O(\sqrt{r})$. The number of regions with more than one child region is bounded by the number of regions with no child regions, implying that $|\mathcal{R}_P| = O(\sqrt{r})$.

Let R be a region with no child region in $\mathcal{T}$ and let C be the cycle bounding R. Since the children of R are exactly the faces in $\overline{int}(C)$ and since C crosses S, R contains at least one face of $\mathcal{F}_1 \cap \mathcal{F}_2$. No other region can contain the child faces of R. The number of regions containing no other region is therefore $O(\sqrt{r})$.

Now, let us give the same bound on the number of regions of $\mathcal{T}$ with exactly one child region. Let R be the region associated with such a node and let R' be its unique child region. Let C and C' be corresponding bounding cycles. Then $\overline{int}(C') \subset \overline{int}(C)$. We may assume that R does not contain any faces of $\mathcal{F}_1 \cap \mathcal{F}_2$ since the number of such faces is $O(\sqrt{r})$, as shown above. Then C and C' must cross S in exactly the same set of boundary vertices and in the same order.

Let $P_1, \ldots, P_k$ be the minimal subpaths of C between vertices of S (P_i does not cross S). Define $P'_1, \ldots, P'_k$ similarly for C' such that P_i and P'_i start and end in the same boundary vertex, $i = 1, \ldots, k$. By Lemma 1, $P_i = P'_i$ for all i except $i = j$ for some j. Let C_j be the cycle $P_j \cup P'_j$. Then the set of faces of G in R all belong to $\overline{int}(C_j)$. Since either $\overline{int}(C_j) \subset \overline{int}(S)$ or $\overline{int}(C_j) \subset \overline{ext}(S)$ and since R contains no faces of $\mathcal{F}_1 \cap \mathcal{F}_2$, all faces of G in R are contained in either $\mathcal{F}_1$ or $\mathcal{F}_2$. This is a contradiction, since $R \in \mathcal{R}_P$. □

3.2 Identifying edges of region subpieces

Region subpieces are composed of two types of edges: *internal edges* and *boundary edges*. Let R be a region and let C be the bounding cycle of R. An edge e is an internal edge of a region subpiece R if the faces on either side of e are enclosed by C. An edge e is a boundary edge of R if e is an edge of C. Every edge is an internal edge for exactly one region subpiece: Lemma 4 shows how we can identify this region. We can also determine if an edge is a boundary edge for some region (Lemma 5). However, an edge can be a boundary edge for several region subpieces. We show how to deal with this potential problem efficiently in Section 3.3.

Lemma 4. *Let e be an edge of G and let f_1 and f_2 be the faces adjacent to e. Then e is an internal edge of a region R iff R is the lowest common ancestor of f_1 and f_2 in the region tree.*

Proof. There must exist some region R satisfying the lemma. Let C be its bounding cycle. Then both f_1 and f_2 are contained in $\overline{int}(C)$ and it follows that R must be a common ancestor of f_1 and f_2. If R' is another common ancestor then either R is an ancestor of R' or R' is an ancestor of R. If R is an ancestor of R' then R' is contained in a face of R so e cannot belong to R. But this contradicts the choice of R. Hence, $R = lca(f_1, f_2)$. □

Iterating over each edge e of P, we can find the region R for which e is an internal edge. If $R \in \mathcal{R}_P$, we add e to the corresponding region subpiece P_R. The total time to add internal edges to their corresponding region subpieces is $O(r \log n)$.

Lemma 5. *Let e be an edge of G and let f_1 and f_2 be the faces adjacent to e. Let R' be the lowest common ancestor of f_1 and f_2 in the region tree. Then e is a boundary edge of a region R iff R is a descendant of R' and one of f_1, f_2 is a descendant of R.*

Proof. Assume first that $e \in C$, where C is the cycle bounding R. Then exactly one of the faces f_1 and f_2 is in $\overline{int}(C)$ and the other is in $\overline{ext}(C)$. Assume w.l.o.g. that $f_1 \in \overline{int}(C)$ and $f_2 \in \overline{ext}(C)$. Then f_1 is a descendant of R and since f_2 is not, R must be a descendant of R'.

Now assume that R is a descendant of R' and that, say, f_1 is a descendant of R. Then f_2 is not a descendant of R since otherwise, R' could not be an ancestor of R. This implies that $e \in C$. □

Let R be a region in $\mathcal{R}_P$ and let C be the bounding cycle of R. Let P_1 and P_2 be the children of P and let B be the union of boundary vertices of P_1 and P_2. We have seen (in the proof of Lemma 3) that the intersection between C and P are subpaths in P between pairs of vertices of B. We shall refer to these as *cycle paths* (of C). Consider the following algorithm to find cycle paths.

Cycle path identification algorithm Pick a boundary vertex $u \in B$. For every edge e adjacent to u (there are at most three such edges), check to see if e is a boundary edge of R. If there is no such edge, then there is no cycle path through u. If there is, walk along C starting with e until another vertex in B encountered. Add all visited edges to the boundary subpiece. Repeat this process for every vertex in B.

Using Lemma 5 and the top tree, this process takes $O((\sqrt{r} + |C \cap P_R|) \log n)$ time since a constant number of tree queries is required for every edge of C in P_R and for every vertex in B.

If the cycles are edge-disjoint over all regions $R \in \mathcal{R}_P$, then the cycle paths will also be edge-disjoint. By the above discussion, the time to find all the region subpieces is $O((|\mathcal{R}_P|\sqrt{r} + |P|)\log n) = O(r \log n)$. However, the cycles are not necessarily edge disjoint. We overcome this complication in the next section.

3.3 Efficiently identifying boundaries of region subpieces

Since cycles will share edges, the total length of cycle paths over all cycles can be as large as $O(r^{3/2})$. However, we can maintain the efficiency of the cycle path identification algorithm by using a compact representation of each cycle path. The compact representation consists of edges of P and *cycle edges* that represent paths in P shared by multiple cycle paths.

View each edge of G as two oppositely directed darts and view the cycle bounding a region as a clockwise cycle of darts. The following is a corollary of Lemma 1.

Corollary 1. *If two isometric cycles C and C' of G share a dart, then either $\overline{int}(C) \subseteq \overline{int}(C')$ or $\overline{int}(C') \subseteq \overline{int}(C)$.*

Let $\mathcal{F}$ be the forest representing the ancestor/descendant relationship between the bounding cycles of regions in $\mathcal{R}_P$. By Lemma 3, there are $O(\sqrt{r})$ bounding cycles and since we can make descendent queries in the region tree in $O(\log n)$ time per query, we can find $\mathcal{F}$ in $O(r \log n)$ time. Let d be the maximum depth of a node in $\mathcal{F}$ (roots have depth 0). For $i = 0, \ldots, d$, let $\mathcal{C}_i$ be the set of cycles corresponding to nodes at depth i in $\mathcal{F}$. As a result of Lemma 1:

Corollary 2. *For any $i \in \{0, \ldots, d\}$, every pair of cycles in $\mathcal{C}_i$ are pairwise dart-disjoint.*

Bottom-up algorithm

We find cycle paths for cycles in $\mathcal{C}_d$, then $\mathcal{C}_{d-1}$, and so on. The cycles in $\mathcal{C}_d$ are dart disjoint, so any edge appears in at most two cycles of $\mathcal{C}_d$. We find the corresponding cycle paths using the cycle path identification algorithm in near-linear time. While Corollory 2 ensures that the cycles in $\mathcal{C}_d$ are mutually dart-disjoint, they can share darts with cycles in C_{d-1}. In order to efficiently walk along subpaths of cycle paths Q that we have already discovered, we use a binary search tree (BST) to represent Q. We can also process the BST such that, given two nodes in Q, the weight of the corresponding subpath of Q can be determined in logarithmic time.

To find the cycle paths of a cycle $C \in \mathcal{C}_{d-1}$ that bounds a region R, we emulate the cycle path identification algorithm: start walking along a cycle path Q of C, starting from a vertex of B, and stop if you reach an edge $e = uv$ that has already been visited *(linear search)*. In this case, e must be an edge of a cycle path Q' of a cycle $C' \in \mathcal{C}_d$. By Lemma 1, the intersection of Q and Q' is a single subpath and so we can use the BST to find the last vertex w common to Q and Q' *(binary search)*. We add to P_R an edge uw of weight equal to the weight of the u-to-w subpath of Q to compactly represent this subpath. If $w \in B$, we stop our walk along Q. Otherwise we continue walking (and adding edges to the corresponding region subpiece) in a linear fashion, alternating between linear and binary searches until a boundary vertex is reached. See Figure 4.

We have shown how to obtain region subpieces for cycles in $\mathcal{C}_d$ and in $\mathcal{C}_{d-1}$. In order to repeat the above idea to find cycle paths for cycles in $\mathcal{C}_{d-2}$, we need to build BSTs for cycle paths of cycles in $\mathcal{C}_{d-1}$. Let Q be one such cycle path. Q can be decomposed into subpaths $Q_1Q'_1 \cdots Q_kQ'_k$, where $Q_1, \ldots, Q_k$ are paths obtained with linear searches and $Q'_1, \ldots, Q'_k$ are paths obtained with binary

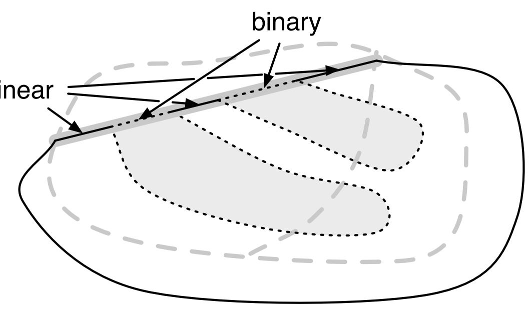


Figure 4: Finding a cycle path (highlighted straight line) for a cycle $C \in \mathcal{C}_{d-1}$ between boundary nodes of P_1 and P_2 (grey dashed lines) is found by alternating linear (solid) and binary (dotted) searches. Binary searches corrrespond to cycle paths of region subpieces (shaded) bounded by cycles in $\mathcal{C}_d$.

searches (possibly Q_1 and/or Q'_k are empty). To obtain a binary search tree $\mathcal{T}$ for Q, we start with $\mathcal{T}$ the BST for Q_1. We extract a BST for Q'_1 from the BST we used to find Q'_1 and merge it into $\mathcal{T}$. We continue merging with BSTs representing the remaining subpaths.

Once BSTs have been obtained for cycle paths arising from $\mathcal{C}_{d-1}$, we repeat the process for cycles in $\mathcal{C}_{d-2}, \ldots, \mathcal{C}_0$.

Running time We now show that the bottom-up algorithm runs in $O(r \log^2 n)$ time over all region subpieces. We have already described how to identify boundary vertices that are starting points of cycle paths in $O(r \log n)$ time. It only remains to bound the time required for linear and binary searches and BST construction.

A subpath identified by a linear search consists only of edges that have not yet been discovered. Since each step of a linear search takes $O(\log n)$ time, the total time for linear searches is $O(r \log n)$.

The number of cycle paths corresponding to a cycle C is bounded by the number of boundary vertices, $O(\sqrt{r})$. We consider three types of cycles paths. Those where

1. all edges are shared by a single child of C in $\mathcal{F}$,

2. no edges are shared by a child, and

3. some but not all edges are shared by a single child.

Cycle paths of the first type are identified in a single binary search for a total of $O(r)$ such searches over all cycles in $C \in \mathcal{F}$. Cycle paths of the second type do not require binary search. For a cycle path Q in the third group, Q can only share one subpath with each child (in $\mathcal{F}$) cycle by Lemma 1; hence, there can be at most two binary searches per child. Summing over all such cycles, the total number of binary searches is $O(\sqrt{r})$.

In total there are $O(r)$ binary searches. Each BST has $O(r)$ nodes. In traversing the binary search tree, an edge is checked for membership in a given cycle path using Lemma 5 in $O(\log n)$ time. Each binary search therefore takes $O(\log r \log n) = O(\log^2 n)$ time. The total time spent performing binary searches is $O(r \log^2 n)$.

It remains to bound the BST construction time. We merge BSTs T_1 and T_2 in $O(\min\{|T_1|, |T_2|\} \log(|T_1|+|T_2|\})) = O(\min\{|T_1|, |T_2|\} \log n)$ time by inserting elements from the smaller tree into the larger.

When forming a BST for a cycle path of a cycle C, it may be necessary to delete parts of cycle paths of children of C. By Lemma 1, these parts intersect $int(C)$ and will not be needed for the remainder of the algorithm. The total number of deletions is $O(r)$ and they take $O(r \log r)$ time to execute. So, ignoring deletions, notice that paths represented by BSTs are pairwise dart disjoint (due to Corollary 2). Applying the following lemma with $k = \log n$ and $W = r$ then gives Theorem 9.

Lemma 6. *Consider a set of objects, where each object o is assigned a positive integer weight $w(o)$. Let $merge(o, o')$ be an operation that replaces two distinct objects o and o' by a new object whose weight is at most $w(o) + w(o')$. Assume that the time to execute $merge(o, o')$ is bounded by $O(\min\{w(o), w(o')\}k)$ for some value k. Then repeating the merge-operation on pairs of objects in any order until at most one object remains takes $O(kW \log W)$ time where W is the total weight of the original objects.*

Proof. We only need to consider the hard case where in beginning, all objects have weight 1 and at termination, exactly one object of weight W remains. Furthermore, we may assume that the weight of $merge(o, o')$ is exactly $w(o) + w(o')$ for two objects o and o'.

Consider running the algorithm backwards: starting with one object of weight W, repeatedly apply an operation *split* that splits an object of weight at least two into two new objects of positive integer weights such that the sum of weights of the two equals the weight of the original object. Assume that *split* runs in time proportional to the smaller weight of the two new objects times k. If we can give a bound of $O(kW \log W)$ for this algorithm, we also get a bound on the algorithm stated in the theorem.

The running time for the new algorithm satisfies:

$$T(w) \leq k \max_{1 \leq w' \leq \lfloor w/2 \rfloor} \{T(w') + T(w - w') + cw'\}$$

for integer $w > 1$ and constant $c > 0$. It is easy to see that the right-hand side is maximized when $w' = \lfloor w/2 \rfloor$. This gives $T(W) = O(kW \log W)$, as desired. □

Theorem 9. *The region subpieces of a piece of size r can be identified in $O(r \log^2 n)$ time.*

4 Separating a pair of faces

In this section we show how to find the minimum fg-separating cycle for the unique pair of faces f, g in a region subpiece P_R. We assume that the region subpiece is given to us by the work in Section 3. We emulate the algorithm due to Reif [17] to find the minimum separating cycle.

We will use an operation of cutting open planar embedded graph G along a path X: duplicate every edge of X and every internal vertex of X and create a new, simple face whose boundary is composed of edges of X and their duplicates. The resulting graph is denoted G_X.

Paths P and Q cross if there is a quadruple of faces adjacent to P and Q that cover the set product $\{\text{left of } P, \text{right of } P\} \times \{\text{left of } Q, \text{right of } Q\}$.

4.1 Reif's minimum separating cycle algorithm

Let X be the shortest path between any vertex on the boundary of f and any vertex on the boundary of g. Reif uses the fact that, since X is a shortest path, there is a minimum gf-separating cycle, C, that crosses X only once:

Theorem 10. *Let X be the shortest g-to-f path. For each vertex $x \in X$, let C_x be the minimum weight cycle that crosses X exactly once and does so in x. Then a minimum fg-separating cycle is a cycle C_x of minimum weight. Further, C_x is the shortest path between duplicates of x in G_X. (Proposition 3 [17], originally due to Itai and Shiloach [11].)*

Reif computes, for every vertex $x \in X$, the minimum separating cycle C_x. Finding C_x amounts to finding a shortest x-to-x' path in G_X, where x' is the duplicate of x. The running time of the algorithm is bounded by divide and conquer: start with x, the midpoint of X in terms of the number of vertices, and recurse on the subgraphs obtained by cutting along C_x. The algorithm takes $O(n \log n)$ time using the $O(n)$ algorithm for shortest paths in planar graphs.

Our goal is to emulate Reif's algorithm in time $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$. In order to attain this running time, we must deal with the following peculiarities:

- X is not contained entirely in P_R. For a vertex $x \in X$ that is outside P_R, we will compute C_x by composing distances in *ext*DDG and *int*DDG between restricted pairs of boundary vertices of P_R. We call such cycles external cycles. We show how to find these cycles in Section 4.4.
- For vertices $x \in X$ that are inside P_R, C_x may not be contained by P_R. We find C_x by first modifying *ext*DDG to disallow paths from crossing X. We call such cycles internal cycles. We show how to find these cycles in Section 4.3.
- *ext*DDG corresponds to distance in G, not G_X. We compute modified dense distance graphs to account for this (Section 4.2).

We can compute X with Dijkstra using $O((|\partial P_R|^2 + |P_R|) \log |P_R|)$ time. In the next two sections we show how to find all the internal cycles and external cycles in time $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$ time. The minimum length cycle over all internal and external cycles is the minimum fg-separating cycle in G.

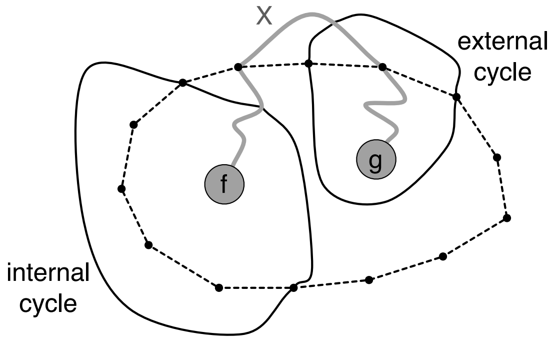


Figure 5: An external and internal cycle separating faces f and g in a region subpiece, whose boundary vertices are given by the bold vertices.

4.2 Modifying the external dense distance graph

We use *ext*DDG to create *ext*DDG$_X$, the dense distance graph that corresponds to distances between boundary vertices of P_R when the graph is cut open along X. Let B be the set of boundary vertices of P_R. Cutting G open along X duplicates vertices of B that are in X, creating B'. $ext\mathrm{DDG}_X$ can be represented as a table of distances between every pair of vertices of B': $ext\mathrm{DDG}_X(x,y) = \infty$ if x is a copy of y or if x and y are separated in G_X outside P_R and $ext\mathrm{DDG}_X(x,y) = ext\mathrm{DDG}(x,y)$ otherwise.

We describe how to determine if x and y are separated in G_X outside P_R. The portions of X that appear outside P_R form a parenthesis of (a subset of) the boundary vertices, illustrated in Figure 6. By travelling along X we can mark the start and endpoints of these parentheses. By walking along the boundary of the subpiece starting at a vertex a, we can easily determine the vertices b that are cut off by a part of the parenthesis and set the corresponding distance in $ext\mathrm{DDG}_X$ to ∞. For vertex a, this will take $O(|\partial P_R|)$ time. Repeating for every boundary vertex of P_R takes a total of $O(|\partial P_R|^2)$ time.

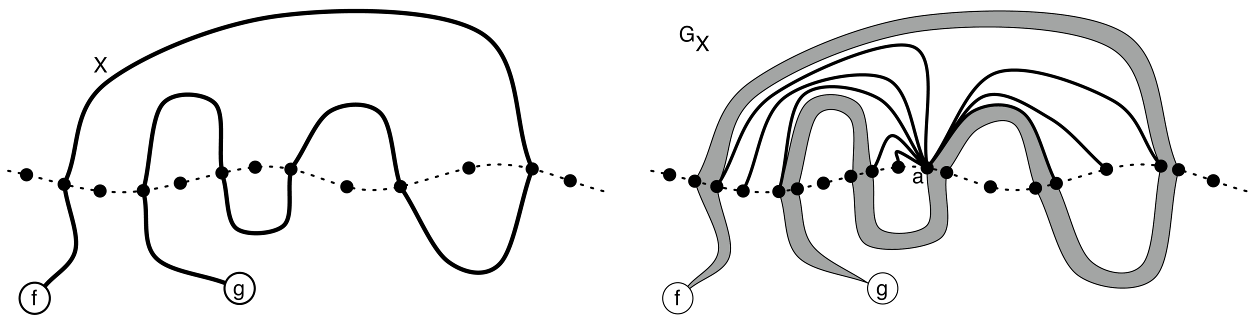


Figure 6: Modifying the external dense distance graph. (Left) X is given by the solid line and the boundary of the subpiece is given by the dotted line. The parts of X outside the subpiece form a parenthesis. (Right) In G_X, the only finite distances from a in $ext\mathrm{DDG}_X$ correspond to the thick lines. The shaded area represents the new face created by cutting along X.

4.3 Finding internal cycles

Let $D = V(X \cap P_R)$. Let D be ordered according to the order of the vertices along X. For each vertex $x \in D$, we compute the shortest x-to-x' paths in G_X where x' is the copy of x in G_X. We do this using Dijsktra's algorithm on the cut-open graph induced by the vertices in P_R ($G_X[P_R]$) and the modified dense distance graph: $ext\mathrm{DDG}_X$. Each cycle can then be found in $O((|P_R|+|\partial P_R|^2)\log|P_R|)$ time. Let x_m be the midpoint vertex of D according to the order inherited from X. C_{x_m} splits P_R and $ext\mathrm{DDG}_X$ into two parts (not necessarily balanced). Recursively finding the cycles through the midpoint in each graph part[1] results in $\log|D|$ levels for a total of $O((|P_R|+|\partial P_R|^2)\log|P_R|\log|D|) = O((|P_R|+|\partial P_R|^2)\log^2|P_R|)$ time to find all the internal cycles.

4.4 Finding external cycles

We show that every external cycle is composed of a single edge ab in the *unmodified* extDDG and a shortest path π_{ab} between boundary vertices of P_R in G that does not cross X.

Using the modified Dijkstra algorithm of Fakcharoenphol and Rao (Section 3.2.2 and 3.2.2 [5]), we can compute all such cycles in $O(|\partial P_R|^2\log^2|P_R|)$ time if $ext\mathrm{DDG}_X$ and $int\mathrm{DDG}_X$ are given.

[1]In order to properly bound the running time, one must avoid repeating long paths in the subproblems: in a subproblem resulting from divide and conquer, we remove degree-two vertices by merging the adjacent edges.

$int\mathrm{DDG}_X$ can be found in $O((|\partial P_R|^2 + |P_R|)\log^3 |P_R|)$ time by cutting open X and using the algorithm of Fakcharoenphol and Rao. Note that we need to compute $int\mathrm{DDG}_X$ from scratch because X has been cut open and because P_R is no longer a subgraph of G due to the compact representation presented in Section 3.3.

In order to compute all the external cycles, one enumerates over all pairs a, b of vertices that are split *exactly once* by the parenthesis given by X, summing $d_B(a, b)$ and $ext\mathrm{DDG}(a, b)$. Since there are $O(|\partial P_R|)$ boundary vertices, there are $O(|\partial P_R|^2)$ such cycles. The minimum-weight such cycle is the minimum separating cycle that crosses X outside P_R.

It remains to prove the required structure of the external cycles.

Lemma 7. *The shortest external cycle is composed of a single edge ab in the* unmodified extDDG *and a shortest path π_{ab} between boundary vertices of P_R in G that does not cross X.*

Proof. Let C be the shortest external cycle that separates faces f and g. By Theorem 10, C is a cycle that crosses X exactly once, say at vertex x. Further, C is a shortest path P between duplicates of x in the graph G_X. Since C must separate f and g, C must enter P_R. Starting at x and walking along C in either direction from X, let a and b be the first boundary vertices that C reaches. Let π_{ab} be the a-to-b subpath of C that does not cross X. Since C is a shortest path in G_X, π_{ab} is the shortest path between boundary vertices as given in the theorem.

Let π_x be the a-to-b subpath of C that does cross X. By definition of a and b, π_x contains no vertices of P_R except a and b. Further, π_x must be the shortest such path, as otherwise, C would not be the shortest fg-separating cycle. Therefore, π_x must correspond to the edge ab in extDDG. $\square$

5 Adding a separating cycle to the region tree

Above, we showed how to find a compact representation of a minimum cycle cycle C separating a pair of faces in a region R. This cycle should be added to the basis we are constructing and in this section, we show how to update the region tree $\mathcal{T}$ accordingly. As in the previous section, let P_R be the region subpiece $P \cap R$ of piece P.

When C is added to the partial basis, R is split into two regions, R_1 and R_2. Equivalently, in $\mathcal{T}$, R will be replaced by two nodes R_1 and R_2. The children $\mathcal{F}$ of R will be partitioned into children $\mathcal{F}_1$ of R_1 and $\mathcal{F}_2$ of R_2. Define R_1 to be the region defined by the children of R that are contained to the left of C. Likewise define R_2 to be the region for the children to the right of C. We describe an algorithm that finds $\mathcal{F}_1$ and detects whether $\mathcal{F}_1$ is contained by $\overline{int}(C)$ or $\overline{ext}(C)$. Finding $\mathcal{F}_2$ is symmetric. The algorithms take $O(|\mathcal{F}_i|\log^3 n + (|P_R| + |\partial P_R|^2)\log n)$ time and so we can identify the smaller side of the partition in $O(\min\{|\mathcal{F}_1|, |\mathcal{F}_2|\}\log^3 n + (|P_R| + |\partial P_R|^2)\log n)$ time. This will imply that the total time for all region tree updates during the course of the algorithm is $O(n \log^4 n)$ (see proof of Lemma 6).

Updating the region tree Given the smaller side of the partition, we use cut-and-link operations to update $\mathcal{T}$ in $O(\min\{|\mathcal{F}_1|, |\mathcal{F}_2|\}\log n)$ additional time. See Figure 1 for an illustration. Assume, w.l.o.g., that $\mathcal{F}_1$ is the smaller set. If $\mathcal{F}_1$ is contained by $\overline{int}(C)$ then update $\mathcal{T}$ by: cutting the edges between R and each element in $\mathcal{F}_1$, linking each element in $\mathcal{F}_1$ to R_1, making R the parent of R_1, identifying R with R_2. If $\mathcal{F}_1$ is contained by $\overline{ext}(C)$ then update $\mathcal{T}$ by: cutting the edges between R and each element in $\mathcal{F}_1$, linking each element in $\mathcal{F}_1$ to a new node u, making u the parent of R; identifying R with R_1 and u with R_2.

5.1 Partitioning the faces

R is represented compactly: vertices in $G[R]$ of degree 2 are removed by merging the adjacent edges creating *super edges*. Each super edge is associated with the first and last edge on the corresponding path. In addition to partitioning the faces, we must find the compact representation for the two new regions, R_1 and R_2.

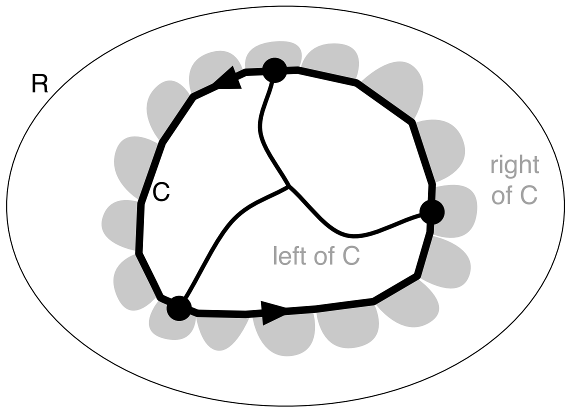


Figure 7: C (bold cycle) is given with a counterclockwise orientation. The children of R (boundary given by thin cycle) adjacent and to the right of C are grey. The edges to the left of C (and not on C) will never reach a boundary edge of R: therefore the left of C forms $\overline{int}(C)$. Vertices of L are given by dark circles.

The algorithm for finding $\mathcal{F}_1$ starts with an empty set and consists of three steps:

Left root vertices Identify the set L of vertices v on C having an edge emanating to the left of C; also identify, for each $v \in L$, the two edges on C incident to v (in G, not the compact representation).

Search Start a search in R from each vertex of L avoiding edges on C or emanating to the right of C; for each super edge $\hat{e}$ of R visited, find the first (or last) edge e on the path represented by $\hat{e}$.

Add For each pair of faces f_1, f_2 adjacent to e, find the two children of R in $\mathcal{T}$ having f_1 and f_2 as descendants, respectively, and add these nodes to $\mathcal{F}_1$.

This algorithm correctly builds $\mathcal{F}_1$: The algorithm visits all super edges $\hat{e}$ that are strictly inside R and on the left side of C. Let A_1 and A_2 be the children of R that are added corresponding to edge e. A_i is a region or a face of G: let C_i be the bounding cycle. Since f_i is a descendent of A_i, f_i is contained by $\overline{int}(C_i)$. Since e is in C_1 and C_2: so must $\hat{e}$. A_1 and A_2 are therefore the child regions of R on either side of $\hat{e}$.

The algorithm can easily determine if $\mathcal{F}_1$ is contained by $\overline{int}(C)$ or $\overline{ext}(C)$: let f_1 and f_2 be the adjacent faces of a searched edge e. Lemma 5 tells us if e is in the bounding cycle of R. If it is, then $\mathcal{F}_1$ must be contained by $\overline{ext}(C)$: otherwise, the search could never reach an edge on the cycle bounding R (since edges of C are not visited) and $\mathcal{F}_1$ must be contained by $\overline{int}(C)$.

Analysis

The above-described algorithm can be implemented in $O(|\mathcal{F}_1|\log^3 n + (|P_R| + |\partial P_R|^2)\log n)$ time. Finding the left-root vertices is the trickiest part; while $|L| = O(|\mathcal{F}_1|)$, $|L|$ could be much smaller than the number of vertices in C, even in the compact representation. We give details in Section 5.2. Assuming that left-root vertices can be found quickly, we analyze the remaining steps.

Search step The total number of super edges that must be searched is $O(|\mathcal{F}_1|)$ since each super edge is incident to two elements of $\mathcal{F}_1$. The search can be done by DFS or BFS in linear time, starting with vertices of L. Given a super edge $\hat{e}$ found by this search, we find the first or last edge e (of G) on the path the super edge represents; this takes $O(1)$ time since e is associated with $\hat{e}$.

In order for this to work, we need to identify, for each $v \in L$, the set of super edges incident to v which are not on C and which do not emanate to the right of C. Since G has degree three, so has the compact representation of R since all faces and isometric cycles in G are simple. It follows that no super edge incident to v emanates to the right of C. To identify the super edge incident to v which is not on C, we can apply Lemma 5 to the first edge on each of the three super edge paths.

Add step The faces f_1 and f_2 incident to an edge e can be found in constant time using the graph representation of G. Applying Lemma 5 to e (to check if $\mathcal{F}_1$ is contained by $\overline{int}(C)$ or $\overline{ext}(C)$) takes $O(\log n)$ time. Finding the children of R that are ancestors of f_1 and f_2 also takes $O(\log n)$ time using the operations $\texttt{jump}(R, f_1, 1)$ and $\texttt{jump}(R, f_2, 1)$ in the top tree for $\mathcal{T}$. The total time spent adding is $O(|\mathcal{F}_1|\log n)$.

5.2 Finding left-root vertices

We show how to find the set L of left-root vertices along C in $O(|\mathcal{F}_1|\log^3 n + |C|\log n)$ time where $|C|$ is the number of super edges in the compact representation of C. As a result of Section 4, C has $O(|P_R| + |\partial P_R|^2)$ super edges and of three different types corresponding to: edges in *ext*DDG and *int*DDG(P_R), and edges and cycle paths in P_R. We will show how to use binary search to prune certain super edges of C that do not contain vertices of L. Assume for now that each super edge is on the boundary of a child region (as opposed to a child face) of R that is to the left of C. We will show how to relax this assumption in Section 5.2.3.

The following lemma is the key to using binary search along C:

Lemma 8. *Let P be the shortest u_1-to-u_2 path in G that is also a subpath of C. For $i = 1, 2$, let e_i be the edge on P incident to u_i and let r_i be the child-region of R that is left of C and is bounded by e_i. Then $r_1 = r_2$ if and only if no interior vertex of P belongs to L.*

Proof. The "if" part is trivial.

By our assumption, r_1 and r_2 are regions, not faces. Their bounding cycles must therefore be isometric. If $r_1 = r_2$ then Lemma 1 implies that P is a subpath of the boundary of r_1: no interior vertex of P could belong to L in this case. This shows the "only if" part. □

5.2.1 Shortest path covering

In order to use Lemma 8, we cover the left-root vertices of C with two shortest paths P and Q. Let r be a vertex that is the endpoint of a super edge of C. Since C is isometric, there is a unique edge e such that C is the union of e and two shortest paths P' and Q' between r and the endpoints of e. Note that e could be in the interior of a super edge of C. The paths P and Q that we use to cover L are prefixes of P' and Q'.

To find e, we first find $\hat{e}$, the super edge that contains e. Since P and Q are shortest paths and shortest paths are unique, the weight of each path is at most half the weight of the cycle. To find $\hat{e}$, simply walk along the super edges of C and stopping when more than half the weight is traversed: $\hat{e}$ is the last super edge on this walk.

Given $\hat{e}$, we continue this walk according to the type of super edge that $\hat{e}$ is. If $\hat{e}$ corresponds to a cycle path, then, by definition, all the interior vertices of $\hat{e}$ have degree two in R and so cannot contain a left-root vertex; there is no need to continue the walk. P and Q are simply the paths along C from r to $\hat{e}$'s endpoints. This takes $O(|C|)$ time.

If $\hat{e}$ is an edge of $int\text{DDG}(P_R)$ or $ext\text{DDG}$, we continue the walk. We describe the process for $int\text{DDG}(P_R)$ as $ext\text{DDG}$ is similar: we continue the walk started above through the subdivision tree of P_R that is used to find $int\text{DDG}(P_R)$. $\hat{e}$ is given by a path of edges in the internal dense distance graph of P_R's children in the subdivision tree. We may assume that we have a top tree representation of the shortest path tree containing this path and so we can find the child super edge $\hat{e}_c$ that contains e using binary search taking $O(\log^2 n)$ time. Recursing through the subdivision tree finds a cycle path or edge that contains e for a total of $O(\log^3 n)$ time.

When we are done, P and Q are paths of super edges from $ext\text{DDG}$ or $int\text{DDG}(P_R)$. P and Q each have $O(|C| + \log n)$ super edges and they are found in $O(|C| + \log^3 n)$ time.

5.2.2 Building L

Using Lemma 8, we will decompose P into maximal subpaths $P_1, \ldots, P_k$ such that no interior vertex of a subpath belongs to L. Each subpath P_i will be associated with the child region of R to the left of C that is bounded (partly) by P_i. We repeat this process for Q and find L in $O(k)$ time by testing the endpoints of the subpaths.

Let $\hat{e}$ be one of the $O(|C|+\log n)$ super edges of P. $\hat{e}$ is either an edge of $ext\text{DDG}$ or $int\text{DDG}(P_R)$. Suppose $\hat{e}$ is in $int\text{DDG}(P_R)$. We can apply Lemma 8 to the first and last edges on the path that $\hat{e}$ represents, and stop if there are no vertices of L in the interior of the path. Otherwise, with the top tree representation of the shortest path tree containing the shortest path representing $\hat{e}$, we find the midpoint of this path and recurse. If $\hat{e}$ is in $ext\text{DDG}$, the process is similar. Adjacent subpaths may still need to be merged after the above process, but this can be done in time proportional to their number.

How long does it take to build L? Let $\hat{e}$ be a super edge representing subpath $P_{\hat{e}}$ of P and let m be the number of interior vertices of $P_{\hat{e}}$ belonging to L. Then there are m leaves in the recursion tree for the search applied to $\hat{e}$. We claim that the height of the recursion tree is $O(\log^2 n)$. Let S be some root-to-leaf path in the recursion tree. If $\hat{e}$ is in $int\text{DDG}(P_R)$, S is split into $O(\log n)$ subpaths, one for each level of the subdivision tree; in each level, the corresponding subpath is halved $O(\log n)$ times before reaching a single edge. If $\hat{e}$ is in $ext\text{DDG}$, the search may go rootwards in the subdivision tree but once we traverse down, we are in $int\text{DDG}$ and will thus not go up again. The depth of the recursion tree is still $O(\log^2 n)$.

At each node in the recursion tree, we apply two top tree operations to check the condition in Lemma 8 and one top tree operation to find the midpoint of a path for a total of $O(\log n)$ time. The total time spent finding the m vertices of L in Q is $O(m \log^3 n)$ time. If $m = 0$, we still need

$O(\log n)$ time to check the condition in Lemma 8. Summing over all super edges of P, the time required to identify L is $O(|C|\log n + |L|\log^3 n) = O(|C|\log n + |\mathcal{F}_1|\log^3 n)$, as desired.

5.2.3 Handling faces

We have assumed that every child of R incident to and left of C is a region, not a face. Lemma 8 is only true for this case: boundaries of faces need not be isometric, and so the intersection between a face and shortest path may have multiple components. However, notice that after the triangulation of the primal followed by the triangulation of the dual, every face f of G is bounded by a simple cycle of the form $e_1P_1e_2P_2e_3P_3$ where e_1, e_2, and e_3 are edges and P_1, P_2, and P_3 are tiny-weight shortest paths (see Section 2.1). Call the six endpoints of edges e_1, e_2, e_3 the *corners of f*. We associate each edge of f with the path containing it among the six paths e_1, e_2, e_3, P_1, P_2, and P_3.

We present a stronger version of Lemma 9 which implies the correctness of the left-root vertex finding algorithm even when children of R are faces, not regions:

Lemma 9. *Let P be the shortest u_1-to-u_2 path in G that is also a subpath of C. For $i = 1, 2$, let e_i be the edge on P incident to u_i and let r_i be the child of R to the left of C and containing e_i. Then*

1. *if neither r_1 nor r_2 are faces, then $r_1 = r_2$ if and only if no interior vertex of P belongs to L,*
2. *if exactly one of r_1, r_2 is a face then some interior vertex of P belongs to L,*
3. *if both r_1 and r_2 are faces and $r_1 \neq r_2$ then some interior vertex of P belongs to L,*
4. *if both r_1 and r_2 are faces, $r_1 = r_2$, and e_1 and e_2 are associated with different subpaths of r_1 then some interior vertex of P is a corner of r_1 or belongs to L,*
5. *if both r_1 and r_2 are faces, $r_1 = r_2$, and e_1 and e_2 are associated with the same subpath of r_1 then no interior vertex of P belongs to L.*

Proof. Part 1 is Lemma 8 and parts 2 and 3 are trivial. For part 4, we may assume that P is fully contained in the boundary of r_1 since otherwise, some interior vertex of P belongs to L. Since e_1 and e_2 are associated with different subpaths of r_1, it follows that some interior vertex of P is a corner of r_1. For part 5, we may assume that $e_1 \neq e_2$. Then e_1 and e_2 are on the same (tiny weight) shortest path in r_1 so P must be contained in the boundary of r_1. It follows that no interior vertex of P belongs to L. □

Using Lemma 9 instead of Lemma 8, our L-finding algorithm will also identify corners of faces incident to P. Since each face has only 6 corners but contributes at least two vertices to L, this will not increase the asymptotic running time.

5.3 Obtaining new regions

While we have found the required partition of the children of R and updated the region tree accordingly, it remains to find compact representations of the new regions R_1 and R_2. Recall that we only explicitly find one side of the partition, w.l.o.g., $\mathcal{F}_1$.

To find R_1, start an initially empty graph. In the *search* step, we explicitly find all the super edges of R_1 that are not on the boundary of C. Remove these edges from R and add them to

R_1. The remaining super edges are simply subpaths of C between consecutive vertices of L. These edges can be added to R_1 *without* removing them from R.

The super edges left in R are exactly those in R_2. However, there may be remaining degree-two vertices that should be removed by merging adjacent super edges. All such vertices, by construction, must be in L, and so can be removed quickly.

That super edges are associated with the first and last edges on their respective paths is easy to maintain given the above construction. The entire time required to build the new compact representation is $O(|\mathcal{F}_1|)$.

6 Reporting min cuts

By Theorem 5, we can report the weight of any min st-cut in constant time. We extend this to report the cuts themselves in time proportional to their size. Size refers to the number of edges in the cut. We will show how to report a minimum separating cycle for a given pair of faces in G^* in time proportional to the number of edges in the cycle. By duality of the min cuts and min separating cycles, this will prove Theorem 6.

In this section, we do not assume that the graph has degree 3. The edges added to achieve that may increase the number of edges in a cycle. However, we can still compute $\mathcal{T}$, the region tree of G^*, with the degree-3 assumption. We will rely only on the relationship between faces in G^*, which did not change in the construction for the degree-3 assumption. Since cycles in the min cycle basis are boundaries of regions represented by $\mathcal{T}$, this tree also reflects the ancestor/descendant relationships between cycles in the min cycle basis.

Consider a cycle C. By Lemma 5 the set of darts in C that are not in an ancestor of C form a path (possibly equal to C). It follows that the set of darts in C that are not in *any* strict ancestor of C also form a path, denoted $P(C)$. Using the next lemma, we can succinctly represent any cycle using these paths. See Figure 8 for an illustration.

Lemma 10. *Let $C = C_0, C_1, C_2, \ldots$ be the ancestral path to the root of $\mathcal{T}$ for C. C can be written as the concatenation of the path $P(C)$, prefixes of $P(C_1), P(C_2), \ldots, P(C_{k-1})$, a subpath of $P(C_k)$ and suffixes of $P(C_{k-1}), \ldots, P(C_2), P(C_1)$, in that order.*

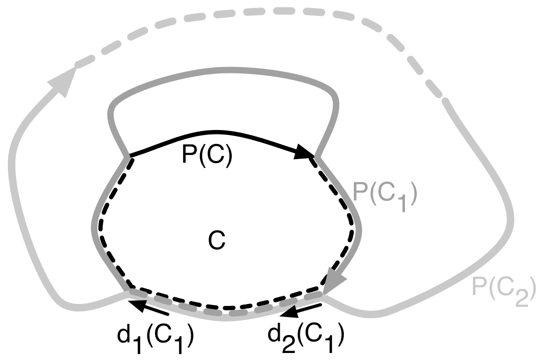


Figure 8: A succinct representation of a cycle in the min cycle basis.

Proof. Let C_k be the root-most cycle that shares a dart d with C: d is in $P(C_k)$. By Lemma 5, the intersection of C with C_k is a single path: it must be a subpath P' of $P(C_k)$. Let $Q = C \setminus P'$. By the definition of $P(C_{k-1})$, the start of Q must be the start of $P(C_{k-1})$ and the end of Q must be the end of $P(C_{k-1})$. The remainder of the proof follows with a simple induction. □

Let $d_1(C)$ be the last dart of C before $P(C)$ and let $d_2(C)$ be the first dart of C after $P(C)$. Suppose additionally that we know, for every dart d, the cycle $C(d)$ for which $d \in P(C)$.

6.1 Finding the min separating cycle

If we are given $d_1(C)$, $d_2(C)$, and $P(C)$ for every cycle (node) in $\mathcal{T}$ and $C(d)$ for every dart d, we can find the minimum fg-separating cycle C in $O(|C|)$ time by the following procedure. First we can find the node in $\mathcal{T}$ corresponding to C in $O(1)$ time using the oracle (Theorem 5). To find C, walk along C, starting with $P(C)$, until you reach the end. Let $C_1 = C(d_2(C))$ and walk along $P(C_1)$ starting with $d_2(C)$. Suppose we are at dart d along C. Let $C_i = C(d)$. Walk along $P(C_i)$ until either you reach its end or you hit dart $d_1(C_{i-1})$. In the first case, continue the process with $d_2(C_i)$. In the second case, continue the process with the first dart of $P(C_{i-1})$. By Lemma 10, this process will eventually reach the start of $P(C)$.

6.2 Preprocessing step

It remains to show how to precompute $d_1(C)$, $d_2(C)$, $P(C)$ for every cycle in $\mathcal{T}$ and $C(d)$ for every dart. We find these using the top tree representation of $\mathcal{T}$ with $O(n \log n)$ preprocessing time.

Let f_ℓ and f_r be the faces to the left and right of a dart d. Then it follows easily from Lemma 5 and the clockwise orientation $C(d)$ that $C(d)$ is the bounding cycle of region $\texttt{jump}(lca(f_\ell, f_r), f_r, 1)$ and can be found in $O(\log n)$ time.

We can easily construct $P(C)$ from the set of darts with $C(d) = C$. The ordering can be found just using the endpoints of these darts so that we can walk along $P(C)$ as required in the previous section.

To find $d_1(C)$ and $d_2(C)$, we work from leaf to root in $\mathcal{T}$ as in the bottom-up algorithm of Section 3.3. We will show how to find $d_2(C)$. Finding $d_1(C)$ is symmetric. For cycle C we can easily find the last dart d_ℓ of $P(C)$. Consider the darts d_o leaving the endpoint of $P(C)$ in counterclockwise order, in the embedding, starting with the reverse of d_ℓ, we test if d_o is on the boundary of C using Lemma 5 in $O(\log n)$ time. As we test darts we remove them from further consideration as they will be in the interior of all ancestor cycles. In total, this takes $O(n \log n)$ time.

7 Lex-shortest paths

Let $w : E \to \mathbb{R}$ be the weight function on the edges of G. So far, we have assumed uniqueness of shortest paths in G between any two vertices w.r.t. w. We now show how to avoid this assumption. We assume in the following that the vertices of G are given indices from 1 to n. For $V' \subseteq V$, define $I(V')$ as the smallest index of vertices in V'.

As shown in [7], there is another weight function w' on the edges of G such that for any pair of vertices in G, there is a unique shortest path between them w.r.t. w' and this path is also a shortest path w.r.t. w. Furthermore, for two paths P and P' between the same pair of vertices in G, $w'(P) < w'(P)$ exactly when one of the following three conditions is satisfied:

1. P is strictly shorter than P' w.r.t. w,
2. P and P' have the same weight w.r.t. w and P contains fewer edges than P',
3. P and P' have the same weight w.r.t. w and the same number of edges and $I(V(P)\setminus V(P')) < I(V(P') \setminus V(P))$.

A shortest path w.r.t. w' is called a *lex-shortest path* and a shortest path tree w.r.t. w' is called a *lex-shortest path tree.*

The properties of w' allow us to apply this function instead of w in our algorithm. In the following, we show how to do this efficiently.

We first use a small trick from [7]: for function w, a sufficiently small $\varepsilon > 0$ is added to the weight of every edge. This allows us to disregard the second condition above. When comparing weights of paths, we may treat ε symbolically so we do not need to worry about precision issues. The tricky part is efficiently testing the third condition.

We need to make modifications to every part of our algorithm in which the weights of two shortest paths are compared. All such comparisons occur when we (1) apply the Dijkstra variant of Fakcharoenphol and Rao [5] and (2) find a shortest path covering of an isometric cycle C in Section 5.2.1.

Note that in making the graph degree three (Section 2.1), we used an ϵ weight on the introduced edges. To combine this with lex-shortest paths, we make $\varepsilon \gg \epsilon$ so that shortest paths will still be chosen to have the fewest edges (in the original graph) and run along the trees introduced to make vertices degree three. Using both ϵ and ε in the symbolic comparisons does not increase the asymptotic run time.

7.1 Dijkstra

Let us first adapt Dijkstra to compute lex-shortest paths. The type of shortest path weight comparisons in [5] are of the form $D(u) + d(u, v) < D(u') + d(u', v)$, where u, v, u', and v' are vertices, $D(u)$ and $D(u')$ are the distances from the root of the partially built tree to u and u', respectively, and $d(u, v)$ and $d(u', v)$ are the weights of edges (u, v) and (u', v). Here, an edge can belong to G, be a cycle edge (see Section 3.3), or an edge of an external or internal dense distance graph.

For simplicity, assume first that all edges considered by Dijkstra belong to G. Then we can test whether $D(u)+d(u, v) < D(u')+d(u', v)$ as follows. Let T be the partially built shortest path tree rooted at a vertex r and let Q and Q' be the lex-shortest paths from r to u and u', respectively. If the first two lex-shortest conditions are inconclusive, we need to check if $I(V(Q) \setminus V(Q')) < I(V(Q') \setminus V(Q))$.

Letting $a = lca(u, u',)$ in T, $V(Q)\setminus V(Q')$ is the set of vertices on $Q[a, u]$, excluding a. Similarly, $V(Q')\setminus V(Q)$ is the set of vertices on $Q'[a, u']$, excluding a. It follows from this that by representing T as a top tree, we can find the smallest index in the two sets in logarithmic time. Using top trees, we can also deal with cycle edges in the same way by keeping, for each such edge, the smallest index of its interior vertices. These indices can be found during the construction of region subpieces in Section 3.1 without an increase in running time.

7.2 Internal dense distances

We also need to handle edges from internal and external dense distance graphs. Let us first consider the problem of computing lex-shortest path trees in *int*DDG. As before, we compute shortest path trees for pieces bottom-up. Let P be a piece with children P_1 and P_2 and assume that we have computed lex-shortest path trees in both of them. Assume also that every edge in $\mathit{int}\mathrm{DDG}(P_1) \cup \mathit{int}\mathrm{DDG}(P_2)$ is associated with the smallest index of interior vertices on the path in G that the edge represents. This information can be computed bottom-up during the construction of *int*DDG without increasing running time.

Let T be a partially built shortest path tree in P and with the above defintions, consider the problem of testing whether $D(u)+d(u,v) < D(u')+d(u',v)$. Let (a, u_a) and (a, u'_a) be the first edges on $Q[a, u]$ and $Q'[a, u']$, respectively. Define Q_G and Q'_G as the paths in G represented by Q and Q', respectively. Let $i_1 = I(V(Q_G[u_a, u]))$, $i'_1 = I(V(Q'_G[u'_a, u']))$, $i_2 = I(V(Q_G[a, u_a]) \setminus V(Q'_G[a, u'_a]))$, and $i'_2 = I(V(Q'_G[a, u'_a]) \setminus V(Q_G[a, u_a]))$. By definition of a, $Q_G[u_a, u]$ and $Q'_G[u'_a, u']$ are vertex-disjoint. Hence, we need to find the four indices and check if $\min\{i_1, i_2\} < \min\{i'_1, i'_2\}$.

Each edge of T belongs to $\mathit{int}\mathrm{DDG}(P_1) \cup \mathit{int}\mathrm{DDG}(P_2)$ and is thus associated with the smallest index of interior vertices on the path in G represented by that the edge. Top tree operations on T as above then allow us to find i_1 and i'_1 in logarithmic time.

To find i_2 and i'_2, we consider two cases: (a, u_a) and (a, u'_a) belong to the internal dense distance graph for the same child of P or they belong to different graphs. In the first case, assume that, say, $(a, u_a), (a, u'_a) \in \mathit{int}\mathrm{DDG}(P_1)$. Then we can decompose these two edges into shortest paths in the same shortest path tree in $\mathit{int}\mathrm{DDG}(P_1)$ and we can recursively find i_2 and i'_2. In the second case, assume that, say, $(a, u_a) \in \mathit{int}\mathrm{DDG}(P_1)$ and $(a, u'_a) \in \mathit{int}\mathrm{DDG}(P_2)$. Since the lex-shortest paths representing these edges in $\mathit{int}\mathrm{DDG}(P_1)$ and $\mathit{int}\mathrm{DDG}(P_2)$ are edge-disjoint and since T is a partially built lex-shortest path tree in P, $Q_G[a, u_a]$ and $Q'_G[a, u'_a]$ share no vertices except a. Thus, i_2 is the smallest index of vertices in $V(Q_G[a, u_a]) \setminus \{a\}$ and we can obtain this index in constant time from the index of u_a and the index associated with edge (a, u_a) which is the smallest index of interior vertices on $Q_G[a, u_a]$. Similarly, we can find i'_2 in constant time.

Since the subdivision tree has $O(\log n)$ height, the recursion depth of the above algorithm is $O(\log n)$, implying that we can determine whether $D(u) + d(u, v) < D(u') + d(u', v)$ in $O(\log^2 n)$ time. Hence, lex-shortest path trees in *int*DDG can be computed in a total of $O(n \log^5 n)$ time.

7.3 External dense distances

Computing lex-shortest path trees in *ext*DDG within the same time bound is very similar so we only highlight the differences. Having computed lex-shortest path trees in *int*DDG bottom-up, we compute lex-shortest path trees in *ext*DDG top-down. For a piece P, we obtain lex-shortest path trees from lex-shortest path trees in its sibling and parent pieces. We can then use an algorithm similar to the one above to find lex-shortest path trees in P. At each recursive step, we either go up one level in *ext*DDG or go to *int*DDG. It follows that the recursive depth is still $O(\log n)$ so lex-shortest path trees in *ext*DDG can be found in $O(n \log^5 n)$ time.

7.4 Distances in region subpieces

We also apply Dijkstra in Section 4 for a region subpiece P_R. First, we computed shortest path X between two faces of the region subpiece in $O((|\partial P_R|^2 + |P_R|) \log |P_R|)$ time using Dijkstra. With an algorithm similar to the one above, we can instead compute a lex-shortest path between the two faces in $O((|\partial P_R|^2 + |P_R|) \log |P_R| \log^2 n)$ time. Next, we cut open the region subpiece along this path and computed a shortest path in G_X between each pair of duplicate vertices on the two copies of X. Using Reif's algorithm, we obtained a time bound of $O((|\partial P_R|^2 + |P_R|) \log^3 |P_R|)$. We need to find similar shortest paths but with respect to weight function w'.

For each pair of duplicate vertices corresponding to a vertex $x \in X$, we assign the index of x to both of them. Then the lex-shortest path between the two vertices in G_X w.r.t. w must be the shortest path w.r.t. w'. Hence, using the above Dijkstra algorithm, the total time spent on computing shortest paths w.r.t. w' for P_R is $O((|P_R| + |\partial P_R|^2) \log^3(|P_R|) \log^2 n)$.

7.5 Shortest path coverings

In Section 5.2.1, we gave an algorithm to find the unique edge $e = (u, v)$ on isometric cycle C such that the two shortest paths from a fixed vertex r on C to u and to v cover all vertices of C and all edges except e. We showed how to do this in $O(|C| + \log^3 n)$ time, where $|C|$ is the size of the compact representation of C obtained in Section 4. We need to modify the algorithm to deal with lex-shortest paths.

Recall that to find e, a linear search of the super edges of C from r was first applied to find the super edge $\hat{e}$ of C such that the shortest path in G representing $\hat{e}$ contains e. As above, we may assume that every super edge of C is associated with the smallest index of interior vertices on the path it represents. Hence, by keeping track of the smallest interior vertex index for super edges visited so far in the linear search as well as the smallest interior vertex index for edges yet to be visited, we can find $\hat{e}$ in $O(|C|)$ time w.r.t. lex-shortest paths.

Having found $\hat{e}$, we need to apply binary search on a path representing $\hat{e}$ in a lex-shortest path tree. We do this by first finding the midpoint of this path as in Section 5.2.1. If the two halves have the same weight and the same number of edges, we can use a top tree operation on each half to determine which half has the smallest index. It follows that all binary searches to find e take $O(\log^3 n)$ time. The total time to find e is thus $O(|C| + \log^3 n)$, which matches the time in Section 5.2.1.

As a result, we have:

Theorem 11. *The Theorems 3 through 7 hold without the shortest-path uniqueness assumption with only an additional $O(\log^2 n)$ factor in the preprocessing time.*

8 Concluding remarks

We gave an oracle for min st-cut weight queries in a planar undirected graph with non-negative edge weights. Construction time is $O(n \log^4 n)$ and space requirement is $O(n \log n)$, where n is the size of the graph. The actual cut can be reported in time proportional to its size. We obtained this oracle from a Gomory-Hu tree algorithm with the same time and space bounds. Previously, no subquadratic time algorithm was known. We also showed how to compute an implicit representation of a minimum cycle basis in $O(n \log^4 n)$ time and $O(n \log n)$ space and an explicit representation with additional $O(C)$ time and space where C is the size of the basis.

In order to obtain our results, we needed shortest paths to be unique. We showed how to deterministically remove this assumption at the cost of an extra $\log^2 n$-factor in running time.

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Maintaining information in fully dynamic trees with top trees. *ACM Transactions on Algorithms*, 1(2):243–264, 2005.

[2] E. Amaldi, C. Iuliano, T. Jurkiewicz, K. Mehlhorn, and R. Rizzi. Breaking the $o(m^2 n)$ barrier for minimum cycle bases. In A. Fiat and P. Sanders, editors, *Proceedings of the 17th European Symposium on Algorithms*, number 5757 in Lecture Notes in Computer Science, pages 301–312, 2009.

[3] A. Bhalgat, R. Hariharan, D. Panigrahi, and K. Telikepalli. An $\tilde{O}(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 605–614, 2007.

[4] G. Borradaile and P. Klein. An $O(n \log n)$ algorithm for maximum st-flow in a directed planar graph. *Journal of the ACM*, 56(2):1–30, 2009.

[5] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *J. Comput. Syst. Sci.*, 72(5):868–889, 2006.

[6] R. Gomory and T. Hu. Multi-terminal network flows. *Journal of SIAM*, 9(4):551–570, 1961.

[7] D. Hartvigsen and R. Mardon. The all-pairs min cut problem and the minimum cycle basis problem on planar graphs. *SIAM Journal on Discrete Mathematics*, 7(3):403–418, 1994.

[8] R. Hassin and D. B. Johnson. An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar networks. *SIAM Journal on Computing*, 14:612–624, 1985.

[9] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[10] J. Horton. A polynomial time algorithm to find the shortest cycle basis of a graph. *SIAM Journal on Computing*, 16:358–366, 1987.

[11] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM Journal on Computing*, 8:135–150, 1979.

[12] H. Kaplan and N. Shafrir. Path minima in incremental unrooted trees. In *Proceedings of the 16th European Symposium on Algorithms*, number 5193 in Lecture Notes in Computer Science, pages 565–576, 2008.

[13] G. Kirchhoff. Ueber die auflösung der gleichungen, auf welche man bei der untersuchung der linearen vertheilung galvanischer ströme geführt wird. *Poggendorf Ann. Physik*, 72:497–508, 1847. English transl. in Trans. Inst. Radio Engrs. CT-5 (1958), pp. 4-7.

[14] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 146–155, 2005.

[15] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[16] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32(3):265–279, 1986.

[17] J. Reif. Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM Journal on Computing*, 12:71–81, 1983.

[18] H. Whitney. Planar graphs. *Fundamenta mathematicae*, 21:73–84, 1933.

B

Computing Best and Worst Shortcuts of Graphs Embedded in Metric Spaces

Jun Luo[1,⋆] and Christian Wulff-Nilsen[2]

[1] Department of Information and Computing Sciences, Universiteit Utrecht, Centrumgebouw Noord, office A210, Padualaan 14, De Uithof, 3584CH Utrecht, The Netherlands
ljroger@cs.uu.nl
http://www.cs.uu.nl/staff/ljroger.html

[2] Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen O, Denmark
koolooz@diku.dk
http://www.diku.dk/hjemmesider/ansatte/koolooz/

Abstract. Given a graph embedded in a metric space, its dilation is the maximum over all distinct pairs of vertices of the ratio between their distance in the graph and the metric distance between them. Given such a graph G with n vertices and m edges and consisting of at most two connected components, we consider the problem of augmenting G with an edge such that the resulting graph has minimum dilation. We show that we can find such an edge in $O((n^4 \log n)/\sqrt{m})$ time using linear space which solves an open problem of whether a linear-space algorithm with $o(n^4)$ running time exists. We show that $O(n^2 \log n)$ time is achievable if G is a simple path or the union of two vertex-disjoint simple paths. Finally, we show how to find an edge that maximizes the dilation of the resulting graph in $O(n^3)$ time with $O(n^2)$ space and in $O(n^3 \log n)$ time with linear space.

1 Introduction

Given a set of cities, represented by points in the plane, consider the problem of finding a road network that interconnects these cities. We seek a network with low cost in which no large detours are required to get from any city to any other city, that is, the road distance between the two cities should be at most a constant factor larger than the Euclidean distance between them. Typical cost measures of the network include size, length, diameter, and maximum degree.

Spanners are networks in which the largest detour is small, and the problem of finding a low-cost spanner for a given point set has received a lot of attention in recent years [3,5,6].

Typically however, one is not interested in constructing a network from scratch but rather to modify a given network. Consider the following problem. Suppose

⋆ This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.065.503.

we are given a network and we would like to extend it with new edge connections to reduce the largest detour. A cost is involved in adding a connection and we can only afford to add, say, a constant $k \geq 1$ number of connections. The problem is to pick these connections such that the largest detour in the resulting network is minimized.

When $k = 1$, an optimal edge to add is called a best shortcut. Farshi et al. [4] considered the problem of finding a best shortcut in a graph embedded in an arbitrary metric space and showed how to solve it in $O(n^4)$ time with $O(n^2)$ space and in $O(n^3 m + n^4 \log n)$ time with linear space (linear in the size of the input graph), where n is the number of vertices and m is the number of edges of the given network. Various approximation algorithms with faster running times were also presented. The authors posed the following open problem: is there an (exact) algorithm with running time $o(n^4)$ using linear space?

An algorithm with $O(n^3 \log n)$ running time was presented in [7]. It has $O(n^2)$ space requirement however, leaving the problem in [4] open.

In this paper, we present a linear-space algorithm with $O((n^4 \log n)/\sqrt{m})$ running time. Since it may be assumed that the input graph consists of at most two connected components (otherwise, the problem is trivial), $m = \Omega(n)$. Hence, we solve the open problem in [4].

For more special types of graphs, we give faster algorithms. We show how to obtain $O(n^2 \log n)$ running time when the graph is a simple path or the union of two vertex-disjoint simple paths.

Finally, we consider the problem of finding a worst shortcut, i.e. an edge that *maximizes* the largest detour of the resulting graph. This relates to a problem in [1] where edge-deletions were considered. We show how to solve this for general graphs in $O(n^3)$ time with $O(n^2)$ space and in $O(n^3 \log n)$ time with linear space.

The organization of the paper is as follows. In Section 2, we give some definitions and introduce some notation. In Section 3, we present our algorithm for computing a best shortcut in a graph embedded in a metric space together with the algorithms for special types of graphs. In Section 4, we present the algorithm for finding a worst shortcut in a graph. Finally, we make some concluding remarks in Section 5.

2 Definitions and Notation

Given a non-empty set M, a *metric* on M is a function $d : M \times M \rightarrow \mathbb{R}$ such that for all $x, y, z \in M$,

$$\begin{aligned} &d(x, y) \geq 0 \\ &d(x, y) = 0 \Leftrightarrow x = y \\ &d(x, y) = d(y, x) \\ &d(x, y) \leq d(x, z) + d(z, y). \end{aligned}$$

The pair (M, d) is called a *metric space*.

Let $G = (V, E)$ be a graph embedded in metric space (V, d). We regard G as a weighted graph where each edge $e \in E$ is assigned cost $d(e)$. For any two vertices $u, v \in V$, we define $d_G(u, v)$ as the length of a shortest path between u and v in G. If no such path exists, we define $d_G(u, v) = \infty$.

If $u \neq v$, the *dilation* (or detour) between u and v is defined as $d_G(u, v)/d(u, v)$ and is denoted by $\delta_G(u, v)$. The *dilation* δ_G of G is defined as

$$\delta_G = \max_{u,v \in V, u \neq v} \delta_G(u, v).$$

We define a *shortcut* (in G) as a vertex pair $(u, v) \in V \times V$ and call it a *best shortcut* resp. *worst shortcut* if it minimizes resp. maximizes $\delta_{G \cup \{(u,v)\}}$.

3 Finding a Best Shortcut

In the following, let $G = (V, E)$ be a graph embedded in metric space (V, d). In this section, we present our algorithm for finding a best shortcut in G. Due to space constraints, we only consider the case where G is connected. The disconnected case may be handled in a way similar to that in [7].

3.1 Staircase Functions

We start by introducing so called staircase functions as in [7]. Let $u, v, w_1, w_2 \in V$ be given with $u \neq v$. Define G' as the graph obtained by augmenting G with shortcut $e = (w_1, w_2)$.

A shortest path from u to v in G' either avoids e, visits e in direction $w_1 \to w_2$, or visits e in direction $w_2 \to w_1$. Hence,

$$\begin{aligned} \delta_{G'}(u, v) = \min\{&d_G(u, w_1) + d(w_1, w_2) + d_G(w_2, v), \\ &d_G(u, w_2) + d(w_2, w_1) + d_G(w_1, v), \\ &d_G(u, v)\}/d(u, v). \end{aligned}$$

Let us assume that $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$. Then no shortest path from u in G' visits e in direction $w_1 \to w_2$. Thus, letting

$$\begin{aligned} x &= d_G(u, w_2) + d(w_2, w_1) \\ a &= 1/d(u, v) \\ b &= d_G(w_1, v)/d(u, v) \\ c &= \delta_G(u, v), \end{aligned}$$

we have

$$\delta_{G'}(u, v) = \min\{ax + b, c\} = \begin{cases} ax + b & \text{for } x \leq (c - b)/a \\ c & \text{for } x \geq (c - b)/a \end{cases}.$$

If we keep u, v, and w_1 fixed, we see that a, b, and c are constants and that the dilation between u and v in G' may be expressed as a piecewise linear function $\delta_{(u,v,w_1)}(x)$ of $x = d_G(u, w_2) + d(w_2, w_1)$.

We define *staircase function* $s_{(u,w_1)} : [0, \infty) \to [0, \infty)$ by

$$s_{(u,w_1)}(x) = \max\{\delta_{(u,v,w_1)}(x) | v \in V \setminus \{u\}\}.$$

Note that this function is piecewise linear and non-decreasing.

3.2 The Algorithm

In this section, we present our $O((n^4 \log n)/\sqrt{m})$ time algorithm with linear space for finding a best shortcut in G. In fact, we will show a more general result. Letting M be a positive function of n and m with $M = O(n^2)$, we present an algorithm with $O(M+n)$ space requirement (in addition to the space for storing G and a representation of the metric space) that finds a best shortcut in G in time $O((mn^4 + n^5 \log n)/(M\sqrt{M}) + (n^4 \log n)/\sqrt{M})$.

In the following, let $v_1, \dots, v_n$ be an arbitrary ordering of the vertices of G. Pseudo-code of the algorithm is shown in Figure 1. To simplify the code and the following analysis, we have made the following assumptions: $\sqrt{M}$ is an integer that divides n and M/n is an integer that divides $\sqrt{M}$. It should be clear how to modify the algorithm to handle the case where these assumptions are not satisfied without affecting the asymptotic time and space bounds.

Let us give a high-level overview of the algorithm before proving its correctness and bounding its time and space requirement. The main ideas are similar to those in [7]: build a table T with n^2 entries, one entry for each vertex pair, and fill

```
1. set $\delta_{\min} := \infty$ and let $u_{\min}, v_{\min}$ be uninitialized vertex variables
2. for $i := 1$ to $n - \sqrt{M}$ step $\sqrt{M}$
3.   for $j := 1$ to $n - \sqrt{M}$ step $\sqrt{M}$
4.     initialize a table $T[i, \dots, i + \sqrt{M} - 1][j, \dots, j + \sqrt{M} - 1]$
5.     with all entries set to zero
6.     for $r := i$ to $i + \sqrt{M} - M/n$ step $M/n$
7.       for $s := 1$ to $n - M/n$ step $M/n$
8.         for $a := r$ to $r + M/n - 1$
9.           compute and store SSSP distances in $G$ with source $v_a$
10.        for $b := s$ to $s + M/n - 1$
11.          compute and store SSSP distances in $G$ with source $v_b$
12.        for $a := r$ to $r + M/n - 1$
13.          for $b := s$ to $s + M/n - 1$
14.            let $(w_1, u) = (v_a, v_b)$
15.            compute staircase function $s_{(u,w_1)}$
16.            for $c := j$ to $j + \sqrt{M} - 1$
17.              let $w_2 = v_c$
18.              if $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$
19.                set $T[a][c] := \max\{T[a][c], s_{(u,w_1)}(d_G(u, w_2) + d(w_2, w_1))\}$
20.        free memory used to store SSSP distances
21.    repeat lines 4 to 19 with the values of $i$ and $j$ swapped and
22.    store values in another table $T'[j, \dots, j + \sqrt{M} - 1][i, \dots, i + \sqrt{M} - 1]$
23.    for $r := i$ to $i + \sqrt{M} - 1$
24.      for $s := j$ to $j + \sqrt{M} - 1$
25.        if $r \neq s$ and $\max\{T[r][s], T'[s][r]\} < \delta_{\min}$
26.          set $\delta_{\min} := \max\{T[r][s], T'[s][r]\}$
27.          set $(u_{\min}, v_{\min}) := (v_r, v_s)$
28. return $(u_{\min}, v_{\min})$ as a best shortcut
```

Fig. 1. Pseudo-code for algorithm to find a best shortcut in G

in entries such that, at termination, $\max\{T[u][v], T[v][u]\}$ equals $\delta_{G\cup\{(u,v)\}}$ for each pair of distinct vertices (u, v).

However, now we only have $O(M)$ space available so we subdivide T into n^2/M sub-tables each of width and height $\sqrt{M}$, see Figure 2.

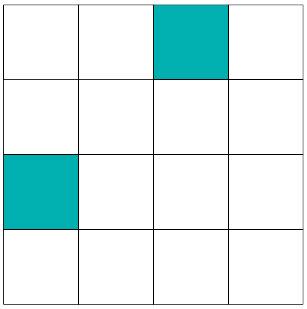


Fig. 2. T is decomposed into n^2/M sub-tables. Here, $n/\sqrt{M} = 4$. If the entries of T in the two marked sub-tables are computed then $\max\{T[u][v], T[v][u]\} = \delta_{G\cup\{(u,v)\}}$ can be obtained for all M vertex pairs (u, v) belonging to those two subtables.

We only keep two sub-tables in memory at a time. More precisely, for $i = 1, \ldots, n/\sqrt{M}$ and $j = 1, \ldots, n/\sqrt{M}$, we fill in entries of the sub-table in row i and column j and entries of the sub-table in row j and column i (this is the reason why lines 4 to 19 in the pseudo-code are repeated with indices i and j swapped) and from these entries we obtain $\max\{T[u][v], T[v][u]\}$ for vertex pairs in those two sub-tables.

In the following, we show the correctness of the algorithm and then bound its running time and space requirement.

Correctness. In order to prove the correctness of the algorithm it is enough to show that each value $\max\{T[r][s], T'[s][r]\}$ computed in line 25 equals $\delta_{G\cup\{(v_r,v_s)\}}$ for $r \neq s$ since (r, s) covers all distinct pairs in $\{1, \ldots, n\}^2$ throughout the course of the algorithm.

So consider any i-iteration and any j-iteration of the algorithm. At the end of each iteration of the for-loop in lines 7 to 20, we have (with $w_1 = w_a$ and $w_2 = w_c$),

$$T[a][c] = \max\{s_{(u,w_1)}(d_G(u, w_2) + d(w_2, w_1))|u \in \{v_s, \ldots, v_{s+M/n-1}\},\\ d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)\}.$$

or $T[a][c] = 0$ if there is no $u \in \{v_s, \ldots, v_{s+M/n-1}\}$ such that $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$, for $a = r, \ldots, r + M/n - 1$ and $c = j, \ldots, j + \sqrt{M} - 1$, $a \neq c$. Hence, at the end of each iteration of the for-loop in lines 6 to 20,

$$\begin{aligned} T[a][c] &= \max\{s_{(u,w_1)}(d_G(u, w_2) + d(w_2, w_1))|u \in V,\\ &\qquad d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)\}\\ &=\max\{\delta_{G\cup\{(w_1,w_2)\}}(u, v)|u, v\in V, u\neq v, d_G(u, w_2)<d_G(u, w_1)+d(w_1, w_2)\}. \end{aligned}$$

for $a = i, \ldots, i + \sqrt{M} - 1$ and $c = j, \ldots, j + \sqrt{M} - 1$, $a \neq c$ (this is well-defined since $u = w_2$ satisfies $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$.

Similarly, after lines 21 to 22,

$$T'[c][a] = \max\{\delta_{G\cup\{(w_1,w_2)\}}(u, v) | u, v \in V, u \neq v, d_G(u, w_1) < d_G(u, w_2) + d(w_2, w_1)\}.$$

for $a = i, \ldots, i + \sqrt{M} - 1$ and $c = j, \ldots, j + \sqrt{M} - 1$, $a \neq c$.

For any $u \in V$, any $w_1 \in \{v_i, \ldots, v_{i+\sqrt{M}-1}\}$, and any $w_2 \in \{v_j, \ldots, v_{j+\sqrt{M}-1}\}$ with $w_1 \neq w_2$, either $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$ or $d_G(u, w_1) < d_G(u, w_2) + d(w_2, w_1)$ for otherwise we get the contradiction

$$\begin{aligned} d_G(u, w_2) + d(w_2, w_1) &\leq d_G(u, w_1) \\ &\leq d_G(u, w_2) - d(w_1, w_2) \\ &< d_G(u, w_2) + d(w_2, w_1). \end{aligned}$$

Hence, after lines 6 to 22,

$$\max\{T[a][c], T'[c, a]\} = \max_{u,v\in V, u\neq v} \delta_{G\cup\{(w_1,w_2)\}}(u, v) = \delta_{G\cup\{(w_1,w_2)\}}.$$

for $a = i, \ldots, i + \sqrt{M} - 1$ and $c = j, \ldots, j + \sqrt{M} - 1$, $a \neq c$. This shows the correctness of the algorithm.

Space Requirement. As for space requirement, we observe that each table takes up $\sqrt{M}^2 = M$ space. In lines 8 to 11, we store shortest path distances from $O(M/n)$ vertices to all vertices in G. This takes up a total of $O(M)$ space. Staircase function $s_{(u,w_1)}$ can be represented using $O(n)$ space since it consists of $O(n)$ line segments. Hence, the algorithm uses $O(M + n)$ space.

Running Time. What remains is to bound the running time of the algorithm. The total time spent in lines 4 to 5 is $O(n^2)$. The total number of iterations of the for-loop in lines 7 to 19 is $(n/\sqrt{M})^2(\sqrt{M}/(M/n))(n/(M/n)) = n^5/(M^2\sqrt{M})$.

If we use Dijkstra's SSSP algorithm in lines 9 and 11, the total time spent in lines 8 to 11 is

$$O((n^5/(M^2\sqrt{M}))(M/n)(m + n \log n)) = O((n^4/(M\sqrt{M}))(m + n \log n)).$$

Computing staircase function $s_{(u,w_1)}$ in line 15 can be done in $O(n \log n)$ time since it is the upper envelope of $O(n)$ line segments (or halflines) and each line segment can be found in constant time using the precomputed shortest path distances. Since the total number of iterations of the for-loop in lines 13 to 19 is $(n^5/(M^2\sqrt{M}))(M/n)^2 = n^3/\sqrt{M}$, it follows that the time spent in line 15 throughout the course of the algorithm is $O((n^4 \log n)/\sqrt{M})$.

The number of iterations of the for-loop in lines 16 to 19 is $(n^3/\sqrt{M})\sqrt{M} = n^3$. The test in line 18 can be performed in constant time using the precomputed shortest path distances. Computing value $s_{(u,w_1)}(d_G(u, w_2) + d(w_2, w_1)$ can be done in $O(\log n)$ time with binary search since $s_{(u,w_1)}$ is a non-decreasing piecewise linear function. Hence, the time spent in lines 16 to 19 throughout the course of the algorithm is $O(n^3 \log n)$.

Adding up, it follows that the total time spent in lines 4 to 20 is

$$O((n^4/(M\sqrt{M}))(m + n \log n) + (n^4 \log n)/\sqrt{M} + n^3 \log n)$$

which is

$$O((mn^4 + n^5 \log n)/(M\sqrt{M}) + (n^4 \log n)/\sqrt{M})$$

since $M = O(n^2)$. A similar argument shows that the total time spent in lines 21 to 22 is

$$O((mn^4 + n^5 \log n)/(M\sqrt{M}) + (n^4 \log n)/\sqrt{M}).$$

Finally, the total time spent in lines 23 to 27 is $O(n^2)$.

We have now shown the following.

Theorem 1. *A best shortcut in G can be found in $O((mn^4+n^5 \log n)/(M\sqrt{M})+(n^4 \log n)/\sqrt{M})$ time using $O(M + n)$ space.*

Note the tradeoff between time and space. Setting $M = m + n$ solves the open problem in [4].

Corollary 1. *A best shortcut in G can be found in $O((n^4 \log n)/\sqrt{m})$ time using linear space.*

Also note that by setting $M = n^2$, it follows that a best shortcut can be found in $O(n^3 \log n)$ time using $O(n^2)$ space which is the result in [7].

Next, we consider special types of graphs and show that we can achieve faster running times for these.

3.3 Best Shortcut of Two Vertex-Disjoint Simple Paths

In this section, we assume that G is the union of two simple vertex disjoint simple paths L_1 and L_2. We will show that a best shortcut in G can be found in $O(n^2 \log n)$ time.

Clearly, we may restrict our attention to finding shortcuts that connect L_1 and L_2. In the following, let (w_1, w_2) denote a shortcut and assume w.l.o.g. that $w_1 \in L_1$ and $w_2 \in L_2$. Let G' denote the graph $G \cup \{(w_1, w_2)\}$. Denote the endpoints of L_1 by v_1 and v_3 and the endpoints of L_2 by v_2 and v_4. We do not yet know how to pick w_1 and w_2 so let us regard them as variables and introduce real parameters x_1, x_2, x_3, and x_4, defined by

$$\begin{aligned}
x_1 &= d_G(v_1, w_1) + d(w_1, w_2) + d_G(w_2, v_2),\\
x_2 &= d_G(v_3, w_1) + d(w_1, w_2) + d_G(w_2, v_4),\\
x_3 &= d_G(v_3, w_1) + d(w_1, w_2) + d_G(w_2, v_2),\\
x_4 &= d_G(v_1, w_1) + d(w_1, w_2) + d_G(w_2, v_4),
\end{aligned}$$

see Figure 3.

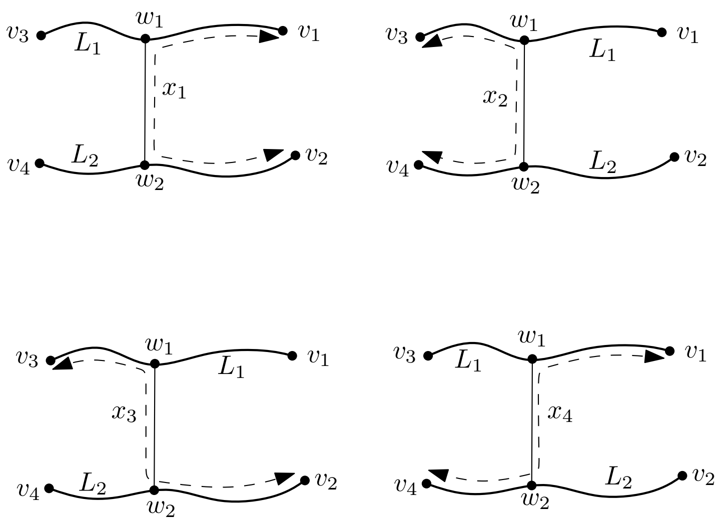


Fig. 3. Definition of four parameters for two vertex-disjoint simple paths

Now, let us express the dilation between vertices $u \in L_1$ and $v \in L_2$ by these parameters. First define

$$\delta^1_{G'}(u,v) = \frac{x_1 - d_G(u,v_1) - d_G(v,v_2)}{d(u,v)},$$
$$\delta^2_{G'}(u,v) = \frac{x_2 - d_G(u,v_3) - d_G(v,v_4)}{d(u,v)},$$
$$\delta^3_{G'}(u,v) = \frac{x_3 - d_G(u,v_3) - d_G(v,v_2)}{d(u,v)},$$
$$\delta^4_{G'}(u,v) = \frac{x_4 - d_G(u,v_1) - d_G(v,v_4)}{d(u,v)}.$$

The following lemma shows how to express $\delta_{G'}(u,v)$ as a function of x_1, x_2, x_3, and x_4.

Lemma 1. $\delta_{G'}(u,v) = \max\{\delta^1_{G'}(u,v), \delta^2_{G'}(u,v), \delta^3_{G'}(u,v), \delta^4_{G'}(u,v)\}$.

Proof. Define $L_{w_1v_i}$ as the subpath of L_1 from w_1 to v_i, $i = 1,3$. Similarly, define $L_{w_2v_i}$ as the subpath of L_2 from w_2 to v_i, $i = 2,4$. Then $\delta_{G'}(u,v) \geq \delta^1_{G'}(u,v)$, $\delta_{G'}(u,v) \geq \delta^2_{G'}(u,v)$, $\delta_{G'}(u,v) \geq \delta^3_{G'}(u,v)$, and $\delta_{G'}(u,v) \geq \delta^4_{G'}(u,v)$.

Furthermore,

$$\delta_{G'}(u,v) = \delta^1_{G'}(u,v) \Leftrightarrow u \in L_{w_1v_1} \text{ and } v \in L_{w_2v_2},$$
$$\delta_{G'}(u,v) = \delta^2_{G'}(u,v) \Leftrightarrow u \in L_{w_1v_3} \text{ and } v \in L_{w_2v_4},$$

$$\delta_{G'}(u,v) = \delta^3_{G'}(u,v) \Leftrightarrow u \in L_{w_1v_3} \text{ and } v \in L_{w_2v_2},$$
$$\delta_{G'}(u,v) = \delta^4_{G'}(u,v) \Leftrightarrow u \in L_{w_1v_1} \text{ and } v \in L_{w_2v_4}.$$

Since $u \in L_1 = L_{w_1v_i} \cup L_{w_1v_3}$ and $v \in L_2 = L_{w_2v_2} \cup L_{w_2v_4}$, we have $\delta_{G'}(u,v) = \max\{\delta^1_{G'}(u,v), \delta^2_{G'}(u,v), \delta^3_{G'}(u,v), \delta^4_{G'}(u,v)\}$. □

From Lemma 1, it follows that

$$\delta_{G'} = \max\{\delta_{L_1}, \delta_{L_2}, \max_{i=1,2,3,4}\{\max_{u\in L_1, v\in L_2} \delta^i_{G'}(u,v)\}\}.$$

We now give an algorithm that finds a vertex pair (w_1, w_2) that maximizes the value $\max_{u\in L_1,v\in L_2} \delta^1_{G'}(u,v)$. By symmetry, it is enough to show that this problem can be solved in $O(n^2 \log n)$ time in order to prove our claim.

For each pair of distinct vertices u and v, $\delta^1_{G'}(u,v)$ is a linear and non-decreasing function of x_1. Thus, $\max_{u\in L_1,v\in L_2} \delta^1_{G'}(u,v)$ is a piecewise linear, non-decreasing function of x_1 consisting of $O(n^2)$ line segments (and one halfline) and it can be found in $O(n^2 \log n)$ time.

Having found this upper envelope function, we determine an x_1-value for each pair of vertices w_1 and w_2 and compute the corresponding value of the upper envelope function. Using binary search, this takes $O(\log n)$ time for each vertex pair. The pair (w_1, w_2) with the largest value is the pair maximizing $\max_{u\in L_1,v\in L_2} \delta^1_{G'}(u,v)$.

We have now obtained the following result.

Theorem 2. *If G is the union of two vertex-disjoint simple paths, a best shortcut in G can be found in $O(n^2 \log n)$ time.*

3.4 Best Shortcut of a Simple Path

We now show how to find a best shortcut in G when G is a simple path L.

Let v_1 and v_2 be the end vertices of L. By symmetry, we may restrict our attention to shortcuts (w_1, w_2) where $d_G(w_1, v_1) < d_G(w_2, v_1)$. And when finding a pair (u, v) achieving the dilation of $\delta_{G'}$, $G' = G \cup \{(w_1, w_2)\}$, we only need to consider pairs where $d_G(u, v_1) < d_G(u, v_2)$.

We will present an algorithm for the above problem with $O(n^2 \log n)$ running time. The basic idea is the same as in Section 3.3. We introduce real parameters x_1, x_2, x_3, and x_4, defined by

$$\begin{aligned}
x_1 &= d(w_1, w_2) + d_G(w_1, w_2),\\
x_2 &= d_G(v_1, w_1) + d(w_1, w_2) + d_G(w_2, v_2),\\
x_3 &= d_G(v_1, w_1) + d(w_1, w_2) + d_G(w_2, v_1),\\
x_4 &= d_G(v_2, w_2) + d(w_1, w_2) + d_G(w_1, v_2),
\end{aligned}$$

see Figure 4. And we define

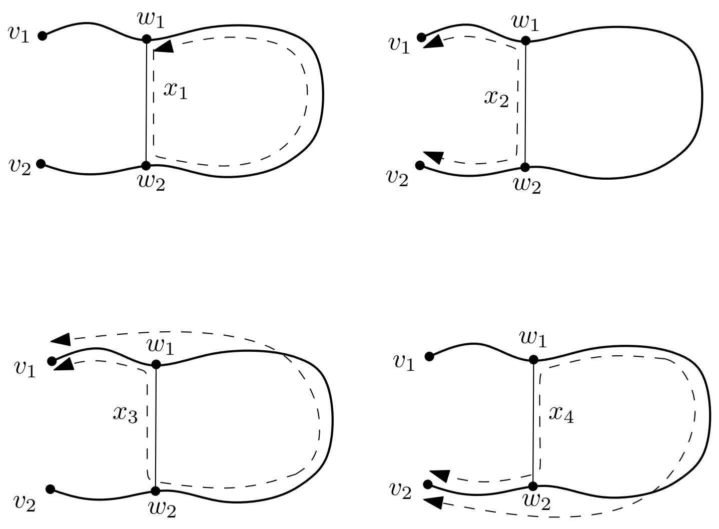


Fig. 4. Definition of four parameters for simple path

$$\delta^1_{G'}(u,v) = \min\{\frac{x_1 - d_G(u,v)}{d(u,v)}, \delta_G(u,v)\},$$

$$\delta^2_{G'}(u,v) = \min\{\frac{x_2 - d_G(u,v_1) - d_G(v,v_2)}{d(u,v)}, \delta_G(u,v)\},$$

$$\delta^3_{G'}(u,v) = \min\{\frac{x_3 - d_G(u,v_1) - d_G(v,v_1)}{d(u,v)}, \delta_G(u,v)\},$$

$$\delta^4_{G'}(u,v) = \min\{\frac{x_4 - d_G(u,v_2) - d_G(v,v_2)}{d(u,v)}, \delta_G(u,v)\}.$$

We have the following result.

Lemma 2. $\delta_{G'}(u,v) = \max\{\delta^1_{G'}(u,v), \delta^2_{G'}(u,v), \delta^3_{G'}(u,v), \delta^4_{G'}(u,v)\}$

Proof. For each pair of vertices $u', v' \in L$, define $L_{u'v'}$ as the subpath of L from u' to v'. As in the proof of Lemma 1, we have $\delta_{G'}(u,v) \geq \delta^1_{G'}(u,v)$, $\delta_{G'}(u,v) \geq \delta^2_{G'}(u,v)$, $\delta_{G'}(u,v) \geq \delta^3_{G'}(u,v)$, and $\delta_{G'}(u,v) \geq \delta^4_{G'}(u,v)$.

Furthermore,

$$\begin{aligned} u,v \in L_{w_1w_2} &\Rightarrow \delta_{G'}(u,v) = \delta^1_{G'}, \\ u \in L_{v_1w_1}, v \in L_{w_2v_2} &\Rightarrow \delta_{G'}(u,v) = \delta^2_{G'}, \\ u \in L_{v_1w_1}, v \in L_{v_1w_2} &\Rightarrow \delta_{G'}(u,v) = \delta^3_{G'}, \\ u \in L_{w_1v_2}, v \in L_{w_2v_2} &\Rightarrow \delta_{G'}(u,v) = \delta^4_{G'}. \end{aligned}$$

Since either $u,v \in L_{w_1w_2}$, or $u \in L_{v_1w_1}, v \in L_{w_2v_2}$, or $u \in L_{v_1w_1}, v \in L_{v_1w_2}$, or $u \in L_{w_1v_2}, v \in L_{w_2v_2}$, the lemma follows. □

We can now use an algorithm similar to that in Section 3.3 to find a best shortcut in G in $O(n^2 \log n)$ time. The only difference is that we get upper envelopes not just of halflines but also of line segments as in Section 3. This gives us the following result.

Theorem 3. *If G is a simple path, a best shortcut in G can be found in $O(n^2 \log n)$ time.*

4 Finding a Worst Shortcut

We now describe an algorithm for a different problem, that of finding a worst shortcut in G. We assume that G is connected. Furthermore, we only allow vertex pairs (w_1, w_2) such that $w_1 \neq w_2$ and such that (w_1, w_2) is not already an edge in G since any other vertex pair would be a trivial worst shortcut.

We will need the following observation. Suppose that (w_1, w_2) is a worst shortcut and let u and v be a pair of vertices such that $\delta_{G'} = \delta_{G'}(u, v)$. Then if $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$, we may assume that w_2 is a vertex in $\{w_2' \in V | d_G(u, w_2') < d_G(u, w_1) + d(w_1, w_2')\}$ that maximizes $d_G(u, w_2') + d(w_2', w_1)$.

The following algorithm finds a worst shortcut in G. A main loop iterates over all vertices w_1. In the following, consider one of these iterations.

For each u, a vertex w_2 is picked (if any) such that $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$, $w_1 \neq w_2$, $(w_1, w_2) \notin E$, and such that $d_G(u, w_2) + d(w_2, w_1)$ is maximized and the maximum of $\delta_{G'}(u, v)$ over all $v \neq u$ is computed. Over all u, this gives at most n dilation values and we keep the largest of them together with a w_2 giving this dilation.

This is done for all w_1, again giving at most n dilation values and we pick the largest of them together with the (w_1, w_2)-pair giving this dilation.

From the observation above and the observation from Section 3.2 that either $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$ or $d_G(u, w_1) < d_G(u, w_2) + d(w_2, w_1)$ for all u and $w_1 \neq w_2$, it follows that the algorithm picks a worst shortcut in G.

If we precompute APSP distances using, say, the algorithm in [2], we obtain an $O(n^3)$ time algorithm with $O(n^2)$ space requirement. We can also obtain linear space requirement but then in each iteration of the main loop, we have to compute SSSP distances with source w_1 and with source u for each u. With Dijkstra's algorithm, this gives $O(n^3 \log n)$ running time.

Theorem 4. *A worst shortcut in G can be found in $O(n^3)$ time with $O(n^2)$ space requirement and in $O(n^3 \log n)$ time with linear space requirement.*

5 Concluding Remarks

In this paper, we considered the problem of finding a best shortcut in a graph G with at most two connected components embedded in a metric space. We presented an algorithm with $O((n^4 \log n)/\sqrt{m})$ running time and linear space requirement, where n is the number of vertices and m is the number of edges in G. This solves an open problem of whether an $o(n^4)$ time algorithm with linear space requirement exists for this problem. We showed that if G is a simple path or the union of two simple vertex-disjoint paths, a best shortcut in G can be found in $O(n^2 \log n)$ time. Finally, we considered the problem of finding a worst shortcut in G. We gave an $O(n^3)$ time algorithm with $O(n^2)$ space requirement and an $O(n^3 \log n)$ time algorithm with linear space requirement for this problem.

It would be interesting to consider the more general case where a fixed number of edges are inserted and to consider edge-removals. Another direction for future research is to consider other special types of graphs like cycles and trees. Finally, not much is known about lower bounds on running time. It is quite easy to show that to find a best shortcut it is sometimes necessary to look at all entries of the $n \times n$-matrix defining the metric. Can this $\Omega(n^2)$ bound be improved?

References

1. Ahn, H.-K., Farshi, M., Knauer, C., Smid, M., Wang, Y.: Dilation-optimal edge deletion in polygonal cycles. In: Tokuyama, T. (ed.) ISAAC 2007. LNCS, vol. 4835, pp. 88–99. Springer, Heidelberg (2007)
2. Chan, T.M.: More algorithms for all-pairs shortest paths in weighted graphs. In: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, pp. 590–598 (2007)
3. Eppstein, D.: Spanning trees and spanners. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 425–461. Elsevier Science Publishers, Amsterdam (2000)
4. Farshi, M., Giannopoulos, P., Gudmundsson, J.: Finding the Best Shortcut in a Geometric Network. In: 21st Ann. ACM Symp. Comput. Geom., pp. 327–335 (2005)
5. Narasimhan, G., Smid, M.: Geometric Spanner Networks. Cambridge University Press, Cambridge (2007)
6. Smid, M.: Closest point problems in computational geometry. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 877–935. Elsevier Science Publishers, Amsterdam (2000)
7. Wulff-Nilsen, C.: Computing the Dilation of Edge-Augmented Graphs in Metric Spaces. In: 24th European Workshop on Computational Geometry, Nancy, pp. 123–126 (2008)

Shortest Paths in Planar Graphs with Real Lengths in $O(n \log^2 n / \log \log n)$ Time

Shay Mozes[1] and Christian Wulff-Nilsen[2]

[1] Department of Computer Science, Brown University, Providence, RI 02912, USA
shay@cs.brown.edu
[2] Department of Computer Science, University of Copenhagen, DK-2100, Copenhagen, Denmark. koolooz@diku.dk

Abstract. Given an n-vertex planar directed graph with real edge lengths and with no negative cycles, we show how to compute single-source shortest path distances in the graph in $O(n \log^2 n / \log \log n)$ time with $O(n)$ space. This improves on a recent $O(n \log^2 n)$ time bound by Klein et al.

1 Introduction

Computing shortest paths in graphs is one of the most fundamental problems in combinatorial optimization with a rich history. Classical shortest path algorithms are the Bellman-Ford algorithm and Dijkstra's algorithm which both find distances from a given source vertex to all other vertices in the graph. The Bellman-Ford algorithm works for general graphs and has running time $O(mn)$ where m resp. n is the number of edges resp. vertices of the graph. Dijkstra's algorithm runs in $O(m + n \log n)$ time when implemented with Fibonacci heaps but it only works for graphs with non-negative edge lengths.

We are interested in the single-source shortest path (SSSP) problem for planar directed graphs. There is an optimal $O(n)$ time algorithm for this problem when all edge lengths are non-negative [3]. For planar graphs with arbitrary real edge lengths and with no negative cycles, Lipton, Rose, and Tarjan [7] gave an $O(n^{3/2})$ time algorithm. Henzinger, Klein, Rao, and Subramanian [3] obtained a (not strongly) polynomial bound of $\tilde{O}(n^{4/3})$. Later, Fakcharoenphol and Rao [2] showed how to solve the problem in $O(n \log^3 n)$ time and $O(n \log n)$ space. Recently, Klein, Mozes, and Weimann [6] presented a linear space $O(n \log^2 n)$ time recursive algorithm.

In this paper, we improve on this by exhibiting a linear space algorithm with $O(n \log^2 n / \log \log n)$ running time. The speed-up comes from a reduction of the recursion depth of the algorithm in [6] from order $\log n$ to order $\log n / \log \log n$. Each recursive step now becomes more involved and to deal with this, we show a new technique for using a certain property, called the Monge property, in graphs that do not necessarily posses that property. Both [2] and [6] showed how to partition a set of distances that are not Monge, into subsets, each of which is Monge. Exploiting this property, the distances within each subset can be processed efficiently. Here we extend that technique by exhibiting sets of Monge distances whose union is a *superset* of the distances we are actually interested in. We believe this technique may be useful in solving other problems as well, not necessarily in the context of the Monge property.

From observations in [6], our algorithm can be used to solve bipartite planar perfect matching, feasible flow, and feasible circulation in planar graphs in $O(n \log^2 n / \log \log n)$ time. From observations in [1], our algorithm generalizes to bounded genus graphs.

The organization of the paper is as follows. In Section 2, we give some definitions and review some basic results, most of them related to planar graphs. Our algorithm is very similar to that of Klein et al. so in Section 3, we give an overview of some of their ideas. We then show how to improve the running time in Section 4. Finally, we make some concluding remarks in Section 5.

2 Preliminaries

In the following, $G = (V, E)$ denotes an n-vertex planar directed graph with real edge lengths and with no negative cycles. For vertices $u, v \in V$, let $d_G(u, v) \in \mathbb{R} \cup \{\infty\}$ denote the length of a shortest path in G from u to v. We extend this notation to subgraphs of G. We will assume that G is triangulated such that there is a path of finite length between each ordered pair of vertices of G. The new edges added have sufficiently large lengths so that finite shortest path distances in G will not be affected.

Given a graph H, let V_H and E_H denote its vertex set and edge set, respectively. For an edge $e \in E_H$, let $l(e)$ denote the length of e (we omit H in the definition but this should not cause any confusion). Let $P = u_1, \ldots, u_m$ be a path in H, where $|P| = m$. For $1 \leq i \leq j \leq m$, $P[u_i, u_j]$ denotes the subpath $u_i, \ldots, u_j$. If $P' = u_m, \ldots, u_{m'}$ is another path, we define $PP' = u_1, \ldots, u_{m-1}, u_m, u_{m+1}, \ldots, u_{m'}$. Path P' is said to *intersect* P if $V_P \cap V_{P'} \neq \emptyset$.

Define a *region* R to be the subgraph of G induced by a subset of V. In G, the vertices of V_R adjacent to vertices in $V \setminus V_R$ are called *boundary vertices* (of R) and the set of boundary vertices of R is called the *boundary* of R. Vertices of V_R that are not boundary vertices of R are called *interior vertices* (of R).

The cycle separator theorem of Miller [8] states that, given an m-vertex plane graph, there is a Jordan curve C intersecting $O(\sqrt{m})$ vertices and no edges such that between $m/3$ and $2m/3$ vertices are enclosed by C. Furthermore, this Jordan curve can be found in linear time.

Let $r \in (0, n)$ be a parameter. Fakcharoenphol and Rao [2] showed how to recursively apply the cycle separator theorem so that in $O(n \log n)$ time, (a plane embedding of) G is divided into $O(n/r)$ regions with the following properties:

1. Each region contains at most r vertices and $O(\sqrt{r})$ boundary vertices,
2. No two regions share interior vertices,
3. Each region has a boundary consisting of $O(1)$ faces, defined by simple cycles.

We refer to such a division as an *r-division* of G. The bounded faces of a region are its *holes*. We will assume that for each region R in an r-division, R is contained in the bounded region defined by one of the cycles C in the boundary of

R. This can always be achieved by adding a new cycle if needed. We refer to C as the *external face* of R.

For a graph H, a *price function* is a function $p : V_H \to \mathbb{R}$. The *reduced cost function* induced by p is the function $w_p : E_H \to \mathbb{R}$, defined by

$$w_p(u, v) = p(u) + l(u, v) - p(v).$$

We say that p is a *feasible* price function for H if for all $e \in E_H$, $w_p(e) \geq 0$.

It is well known that reduced cost functions preserve shortest paths, meaning that we can find shortest paths in H by finding shortest paths in H with edge lengths defined by the reduced cost function w_p. Furthermore, given p and the distance in H w.r.t. w_p from a $u \in V_H$ to a $v \in V_H$, we can extract the original distance in H from u to v in constant time [6].

Observe that if p is feasible, Dijkstra's algorithm can be applied to find shortest path distances since then $w_p(e) \geq 0$ for all $e \in E_H$. The distances$d_H(s, u)$ from any $s \in V_H$ are an example of a feasible price function $u \mapsto d_H(s, u)$. This assumes that $d_H(s, u) < \infty$ for all $u \in V_H$, as we have assumed above.

A matrix $M = (M_{ij})$ is *totally monotone* if for every i, i', j, j' such that $i < i'$, $j < j'$, $M_{ij} \leq M_{ij'}$ implies $M_{i'j} \leq M_{i'j'}$. Totally monotone matrices were introduced by Aggarwal et al. [9], who gave an algorithm, nicknamed SMAWK, that, given a totally monotone $n \times m$ matrix M, finds all column minima of M in just $O(n + m)$ time. A matrix $M = (M_{ij})$ is *convex Monge* if for every i, i', j, j' such that $i < i'$, $j < j'$, we have $M_{ij} + M_{i'j'} \geq M_{ij'} + M_{i'j}$. It is immediate that if M is convex Monge then it is totally monotone. Thus SMAWK can be used to find the column minima of a convex Monge matrix. The algorithm in [6] uses a generalization of SMAWK to so called *falling staircase matrices*, due to Klawe and Kleitman [4]. Klawe and Kleitman's algorithm finds all column minima in $O(m\alpha(n) + n)$ time, where $\alpha(n)$ is the inverse Ackerman function.

3 The Algorithm of Klein et al.

In this section, we give an overview of the algorithm of [6]. Let s be a vertex of G. To find SSSP distances in G with source s, the algorithm finds a cycle separator C with $O(\sqrt{n})$ boundary vertices that separates G into two subgraphs, G_0 and G_1. Let r be any of these boundary vertices. The algorithm consists of five stages:

Recursion: SSSP distances from r are computed recursively in G_0 and G_1.

Intra-part boundary distances: Distances in G_i between every pair of boundary vertices of G_i are computed in $O(n \log n)$ time using the algorithm of [5] for $i = 0, 1$.

Single-source inter-part boundary distances: A variant of Bellman-Ford is used to compute SSSP distances in G from r to all boundary vertices on C. The algorithm consists of $O(\sqrt{n})$ iterations. Each iteration runs in $O(\sqrt{n}\alpha(n))$ time using an algorithm of Klawe and Kleitman [4]. This stage takes $O(n\alpha(n))$ time.

Single-source inter-part distances: Distances in the previous stage are used to modify G such that all edge lengths are non-negative without changing the shortest paths. Dijkstra's algorithm is then used in the modified graph to obtain SSSP distances in G with source r. Total running time for this stage is $O(n \log n)$.

Rerooting single-source distances: The computed distances from r in G form a feasible price function for G. Dijkstra's algorithm is applied to obtain SSSP distances in G with source s in $O(n \log n)$ time.

The last four stages of the algorithm in [6] run in a total of $O(n \log n)$ time. Since there are $O(\log n)$ recursion levels, the total running time is $O(n \log^2 n)$. We next describe how to improve this time bound.

4 An Improved Algorithm

The main idea is to reduce the number of recursion levels by applying the cycle separator theorem of Miller not once but several times at level of the recursion. More precisely, for a suitable p, we obtain an n/p-division of G in $O(n \log n)$ time. For each region R_i in this n/p-division, we pick an arbitrary boundary vertex r_i and recursively compute SSSP distances in R_i with source r_i. This is similar to the first stage of the algorithm in [6], except that we recurse on $O(p)$ regions instead of just two.

We will show how all these recursively computed distances can be used to compute SSSP distances in G with source s in $O(n \log n + np\alpha(n))$ additional time. This bound is no better than the $O(n \log n)$ bound of the original algorithm but does result in fewer recursion levels. Since the size of regions is reduced by a factor of p for each recursion level, the depth of the recursion is only $O(\log n / \log p)$. Furthermore, by recursively applying the separator theorem of Miller as done by Fakcharoenphol and Rao [2], the subgraphs at the kth recursion level defines an r-division of G where $r = n/p^k$. This r-division consists of $O(n/r)$ regions each containing at most r vertices, implying that the total time spent at the kth recursion level is $O(n/r(r \log r + rp\alpha(r))) = O(n \log n + np\alpha(n))$. Summing over all $O(\log n / \log p)$ levels, it follows that the total running time of our algorithm is

$$O\left(\frac{\log n}{\log p}(n \log n + np\alpha(n))\right).$$

To minimize this expression, we set $n \log n = np\alpha(n)$, so $p = \log n/\alpha(n)$. This gives the desired $O(n \log^2 n / \log \log n)$ running time.

It remains to show how to compute SSSP distances in G with source s in $O(n \log n + np\alpha(n)) = O(n \log n)$ time, excluding the time for recursive calls. Assume that we are given an n/p-division of G and that for each region R, we are given SSSP distances in R with some boundary vertex of R as source. Note that the number of regions is $O(p)$ and each region contains at most n/p vertices and $O(\sqrt{n/p})$ boundary vertices.

The main technical difficulty arises from the existence of holes. We will first describe a generalization of [6] using multiple regions instead of just two, but assuming that no region has holes. In this case, as is the case in [6], all of the boundary vertices in a region are cyclically ordered on its external face. In section 4.4 we show how to handle the existence of holes.

Without holes, the remaining four steps of the algorithm are very similar to those in the algorithm of Klein et al. We give an overview here and go into greater detail in the subsections below. Each step takes $O(n \log n)$ time.

Intra-region boundary distances: For each region R, distances in R between each pair of boundary vertices of R are computed.

Single-source inter-region boundary distances: Distances in G from an arbitrary boundary vertex r of an arbitrary region to all boundary vertices of all regions are computed.

Single-source inter-region distances: Using the distances obtained in the previous stage to obtain a modified graph, distances in G from r to all vertices of G are computed using Dijkstra's algorithm on the modified graph.

Rerooting single-source distances: Identical to the final stage of the original algorithm.

4.1 Intra-region Boundary Distances

Let R be a region. Since R has no holes, we can apply the multiple-source shortest path algorithm of [5] to R since we have a feasible price function from the recursively computed distances in R. Total time for this is $O(|V_R| \log |V_R|)$ time which is $O(n \log n)$ over all regions.

4.2 Single-source Inter-region Boundary Distances

Let r be some boundary vertex of some region. We need to find distances in G from r to all boundary vertices of all regions. To do this, we use a variant of Bellman-Ford similar to that in stage three of the original algorithm.

Let $\mathcal{R}$ be the set of $O(p)$ regions, let $B \subseteq V$ be the set of boundary vertices over all regions, and let $b = |B| = O(p\sqrt{n/p}) = O(\sqrt{np})$. Note that a vertex in B may belong to several regions.

Pseudocode of the algorithm is shown in Figure 1. Notice the similarity with the algorithm in [6] but also an important difference: in [6], each table entry $e_j[v]$ is updated only once. Here, it may be updated several times in iteration j since more than one region may have v as a boundary vertex. For $j \geq 1$, the final value of $e_j[v]$ will be

$$e_j[v] = \min_{w \in B_v} \{e_{j-1}[w] + d_R(w, v)\}, \tag{1}$$

where B_v is the set of boundary vertices of regions having v as boundary vertex.

To show the correctness of the algorithm, we need the following two lemmas.

```
1. initialize vector $e_j[v]$ for $j = 0, \ldots, b$ and $v \in B$
2. $e_j[v] := \infty$ for all $v \in B$ and $j = 0, \ldots, b$
3. $e_0[r] := 0$
4. for $j = 1, \ldots, b$
5.     for each region $R \in \mathcal{R}$
6.         let $C$ be the cycle defining the boundary of $R$
7.         $e_j[v] := \min\{e_j[v], \min_{w \in V_C}\{e_{j-1}[w] + d_R(w, v)\}\}$ for all $v \in V_C$
8. $D[v] := e_b[v]$ for all $v \in B$
```

Fig. 1. Pseudocode for single-source inter-region boundary distances algorithm.

Lemma 1. *Let P be a simple r-to-v shortest path in G where $v \in B$. Then P can be decomposed into at most b subpaths $P = P_1P_2P_3\ldots$, where the endpoints of each subpath P_i are boundary vertices and P_i is a shortest path in some region of $\mathcal{R}$.*

Lemma 2. *After iteration j of the algorithm in Figure 1, $e_j[v]$ is the length of a shortest path in G from r to v that can be decomposed into at most j subpaths $P = P_1P_2P_3\ldots P_j$, where the endpoints of each subpath P_i are boundary vertices and P_i is a shortest path in a region of $\mathcal{R}$.*

Both lemmas are straightforward generalizations of the corresponding lemmas in [6]. They imply that after b iterations, $D[v]$ holds the distance in G from r to v for all $v \in B$. This shows the correctness of our algorithm.

Line 7 can be executed in $O(|V_C|\alpha(|V_C|))$ time using the technique of [6] using the distances $d_R(w, v)$ which have been precomputed in the previous stage for all $v, w \in V_C$. It is important to note that the techniques of [6] only apply since we have assumed that all boundary vertices of R are cyclically ordered on its external face. Thus, each iteration of lines 4–7 takes $O(b\alpha(n))$ time, giving a total running time for this stage of $O(b^2\alpha(n)) = O(np\alpha(n))$. Recalling that $p = \log n/\alpha(n)$, this bound is $O(n \log n)$, as desired.

4.3 Single-source Inter-region Distances

In this step we need to compute, for each region R, the distances in G from r to each vertex of R. We apply a nearly identical construction to the one used in the corresponding step of [6].

Let R be a region. Let R' be the graph obtained from R by adding a new vertex r' and an edge from r' to each boundary vertex of R whose length is set to the distance in G from r to the boundary vertex. Note that $d_G(r, v) = d_{R'}(r', v)$ for all $v \in V_R$, so it suffices to find distances in R' from r' to each vertex of V_R.

Let r_R be the boundary vertex of R for which distances in R from r_R to all vertices of R have been recursively computed. Define a price function ϕ for R' as follows. Let B_R be the set of boundary vertices of R and let $D = \max\{d_R(r_R, b)-$

$d_G(r,b)|b \in B_R\}$. Then for all $v \in V_{R'}$,

$$\phi(v) = \begin{cases} d_R(r_R, v) & \text{if } v \neq r' \\ D & \text{if } v = r'. \end{cases}$$

Lemma 3. *Function ϕ defined above is a feasible price function for R'.*

Proof. Let $e = (u,v)$ be an edge of R'. By construction, no edges enter r' so $v \neq r'$. If $u \neq r'$ then $\phi(u) + l(e) - \phi(v) = d_R(r_R, u) + l(u,v) - d_R(r_R, v) \geq 0$ by the triangle inequality so assume that $u = r'$. Then $v \in B_R$ so $\phi(u)+l(e)-\phi(v) = D + d_G(r,v) - d_R(r_R, v) \geq 0$ by definition of D. This shows the lemma. □

Price function ϕ can be computed in time linear in the size of R and Lemma 3 implies that Dijkstra's algorithm can be applied to compute distances in R' from r' to all vertices of V_R in $O(|V_R| \log |V_R|)$ time. Over all regions, this is $O(n \log n)$, as requested.

We omit the description of the last stage where single-source distances are rerooted to source s since it is identical to the last stage of the original algorithm. We have shown that all stages run in $O(n \log n)$ time and it follows that the total running time of our algorithm is $O(n \log^2 n / \log \log n)$. It remains to deal with holes in regions.

4.4 Dealing with Holes

In Sections 4.1 and 4.2, we made the assumption that no region has holes. In this section we remove this restriction. This is the main technical contribution of this paper. As mentioned in Section 2, each region of $\mathcal{R}$ has at most a constant number, h, of holes.

Intra-region boundary distances: In Section 4.1 we used the fact that all boundary vertices of each region are on the external face, to apply the multiple-source shortest path algorithm of [5]. Consider a region R with h holes. If we apply [5] to R we get distances from boundary vertices on the external face of R to all boundary vertices of R. This does not compute distances from boundary vertices belonging to the holes of R. Consider one of the holes of R. We can apply the algorithm of [5] with this hole considered as the external face to get the distances from the boundary vertices of this hole to all boundary vertices of R. Repeating this for all holes, we get distances in R between all pairs of boundary vertices of R in time $O(|V_R| \log |V_R| + h|V_R| \log |V_R|) = O(|V_R| \log |V_R|)$ time. Thus, the time bound in Section 4.1 still holds when regions have holes.

Single-source inter-region boundary distances: It remains to show how to compute single-source inter-region boundary distances when regions have holes. Let C be the external face of region R. Let H_R be the directed graph having the boundary vertices of R as vertices and having an edge (u,v) of length $d_R(u,v)$ between each pair of vertices u and v.

As usual in this context, we say that we relax an edge if it is being considered by the algorithm as the next edge in the shortest path. Line 7 in Figure 1 relaxes all edges in H_R having both endpoints on C. We need to relax all edges of H_R. In the following, when we say that we relax edges of R, we really refer to the edges of H_R.

To relax the edges of R, we consider each pair of cycles (C_1, C_2), where C_1 and C_2 are C or a hole, and we relax all edges starting in C_1 and ending in C_2. This will cover all edges we need to relax.

Since the number of choices of (C_1, C_2) is $O(h^2) = O(1)$, it suffices to show that in a single iteration, the time to relax all edges starting in C_1 and ending in C_2 is $O((|V_{C_1}|+|V_{C_2}|)\alpha(|V_{C_1}|+|V_{C_2}|))$, with $O(|V_R| \log |V_R|)$ preprocessing time. We may assume that $C_1 \neq C_2$, since otherwise we can relax edges as described in Section 4.2.

Before going into the details, let us give an intuitive and informal overview of our approach. We transform R in such a way that C_1 is the external face of R and C_2 is a hole of R. Let P be a simple path from some vertex $r_1 \in V_{C_1}$ to some vertex $r_2 \in V_{C_2}$. Let R_P be the graph obtained by "cutting along P" (see Figure 2). Note that every shortest path in R_P corresponds to a shortest path in R that does not cross P. We will show that we can relaxing all edges in H_R from C_1 to C_2 with respect to distances in R_P can be done efficiently. Unfortunately, relaxing edges w.r.t. R_P will not suffice since shortest paths in R that do cross P are not represented in R_P. To overcome this obstacle we will identify two particular paths P_r and P_ℓ such that for any $u \in C_1, v \in C_2$ there exists a shortest path in R that does not cross both P_r and P_ℓ. Then, relaxing all edges between boundary vertices once in R_{P_r} and once in R_{P_ℓ} suffices to compute shortest path distances in R. More specifically, let T be a shortest path tree in R from r_1 to all vertices of C_2. The rightmost and leftmost paths in T satisfy the above property (see Figure 3).

We now proceed with the formal description. In the following, we define graphs, obtained from R, that are needed in our algorithm. It is assumed that these graphs are constructed in a preprocessing step. Later, we bound the time to construct them.

We transform R in such a way that C_1 is the external face of R and C_2 is a hole of R. We may assume that there is a shortest path in R between every ordered pair of vertices, say, by adding a pair of oppositely directed edges between each consecutive pair of vertices of C_i in some simple walk of C_i, $i = 1, 2$ (if an edge already exists, a new edge is not added). The lengths of the new edges are chosen sufficiently large so that shortest paths in R and their lengths do not change. Where appropriate, we will regard R as some fixed planar embedding of that region.

We say that an edge $e = (u, v)$ with exactly one endpoint on path P *emanates right (left) of* P if (a) e is directed away from P, and (b) e is to the right (left) of P in the direction of P (see e.g., [5] for a more precise definition). If e is directed towards P, then we say that e *enters* P *from the right (left)* if (v, u) emanates right (left) of P. We extend these definitions to paths and say, e.g., that a path Q emanates right of path P if there is an edge of Q that emanates right of P.

For a simple path P from a vertex $r_1 \in V_{C_1}$ to a vertex $r_2 \in V_{C_2}$, take a copy R_P of R and remove P and all edges incident to P in R_P. let $\overleftarrow{E}$ resp. $\overrightarrow{E}$ be the set of edges that either emanate left resp. right of P or enter P from the left resp. right. Add two copies, $\overleftarrow{P}$ and $\overrightarrow{P}$, of P to R_P. Connect path $\overleftarrow{P}$ resp. $\overrightarrow{P}$ to the rest of R_P by attaching the edges of $\overleftarrow{E}$ resp. $\overrightarrow{E}$ to the path, see Figure 2. If $(u, v) \in E_R$, where $(v, u) \in E_P$, we add (u, v) to both $\overleftarrow{P}$ and $\overrightarrow{P}$ in R_P.

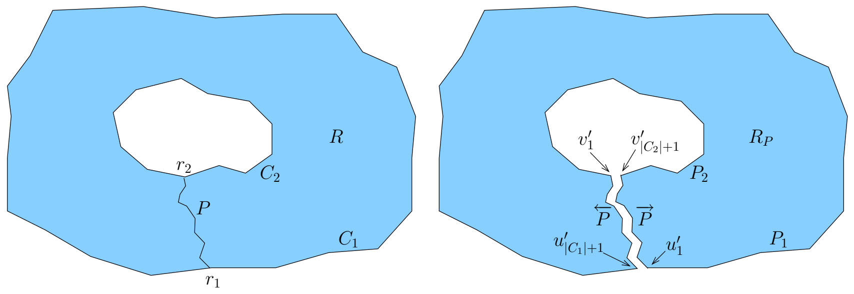


Fig. 2. Region R_P is obtained from R essentially by cutting open at P the "ring" bounded by C_1 and C_2.

A simple, say counter-clockwise, walk $u_1, u_2, \ldots, u_{|C_1|}, u_{|C_1|+1}$ of C_1 in R where $u_1 = u_{|C_1|+1} = r_1$ corresponds to a simple path $P_1 = u'_1, \ldots, u'_{|C_1|+1}$ in R_P. In the following, we identify u_i with u'_i for $i = 2, \ldots, |C_1|$. The vertex r_1 in R corresponds to two vertices in R_P, namely u'_1 and $u'_{|C_1|+1}$. We will identify both of these vertices with r_1. Similarly, a simple, say clockwise, walk of C_2 in R from r_2 to r_2 corresponds to a simple path $P_2 = v'_1, \ldots, v'_{|C_2|+1}$ in R_P. We make a similar identification between vertices of C_2 and P_2.

In the following, when we say that we relax all edges in R_P starting in vertices of C_1 and ending in vertices of C_2, we really refer to relaxing edges in H_R with respect to the distances between the corresponding vertices of P_1 and P_2 in R_P. More precisely, suppose we are in iteration j. Then relaxing all edges entering a vertex $v \in V_{C_2}$ in R_P means updating

$$e_j[v] := \min_{u \in V_{C_1}} \{e_{j-1}[v], e_{j-1}[u] + d_{R_P}(u', v')\}.$$

It is implicit in this notation that if $u = r_1$, we relax w.r.t. both u'_1 and $u'_{|C_1|+1}$ and if $v = r_2$, we relax w.r.t. both v'_1 and $v'_{|C_2|+1}$.

The fact that in R_P P_1 and P_2 both belong to the external face implies (see Lemma 4.3 in [6] or the appendix):

Lemma 4. *Relaxing all edges from V_{C_1} to V_{C_2} in R_P can be done in $O(|V_{C_1}| + |V_{C_2}|)$ time in any iteration of Bellman-Ford.* $\square$

As we have mentioned, relaxing edges between boundary vertices in R_P does not suffice since shortest paths in R that cross P are not represented in R_P. Let T be a shortest path tree in R from r_1 to all vertices of C_2. A *rightmost (leftmost)* path P in T is a path such that no other path Q in T emanates right (left) of P. Let P_r and P_ℓ be the rightmost and leftmost root-to-leaf simple paths in T, respectively; see Figure 3(a). Let $v_r \in C_2$ and $v_\ell \in C_2$ denote the leaves of P_r and P_ℓ, respectively.

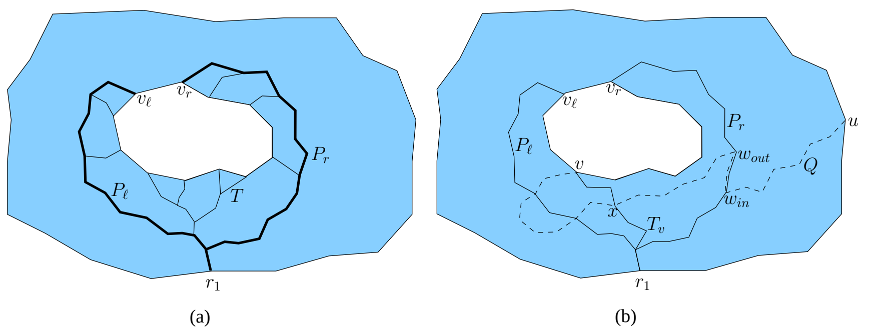


Fig. 3. (a): The rightmost root-to-leaf simple path P_r and the leftmost root-to-leaf simple path P_ℓ in T. (b): In the proof of Lemma 5, if Q first crosses P_r from right to left and then crosses P_ℓ from right to left then there is a u-to-v shortest path in R that does not cross P_ℓ.

In order to state the desired property of P_r and P_ℓ we now define what we mean when we say that path $Q = q_1, q_2, q_3, \dots$ *crosses* path P. Let out_0 be the smallest index such that q_{out_0} does not belong to P. We recursively define in_i to be smallest index greater than out_{i-1} such that q_{in_i} belongs to P, and out_i to be smallest index greater than in_i such that q_{out_i} does not belong to P. We say that Q crosses P from the right (left) with entry vertex v_{in} and exit vertex v_{out} if (a) $v_{in} = q_{in_i}$ and $v_{out} = q_{out_i - 1}$ for some $i > 0$ and (b) $q_{in_i - 1} q_{in_i}$ enters P from the right (left) and (c) $q_{out_i - 1} q_{out}$ emanates left (right) of P.

Lemma 5. *For any $u \in V_{C_1}$ and any $v \in V_{C_2}$, there is a simple shortest path in R from u to v which does not cross both P_r and P_ℓ.*

Proof. Let Q be a simple u-to-v shortest path in R which is minimal with respect to the total number of time it crosses P_r and P_ℓ. If Q does not cross P_r or P_ℓ, we are done, so assume it crosses both. Also assume that Q crosses P_r first. The case where Q crosses P_ℓ first is symmetric. Let w_{in} and w_{out} be the entry and exit vertices of the first crossing, see Figure 3(b). There are two cases:

– Q first crosses P_r from left to right. In this case Q must cross P_ℓ at the same vertices. In fact, it must be that all root-to-leaf paths in T coincide until w_{out} and that Q crosses them all. In particular, Q crosses the root-to-v

path in T, which we denote by T_v. Since T_v does not cross P_r, the path $Q[u, w_{out}]T_v[w_{out}, v]$ is a shortest u-to-v path in R that does not cross P_r.

- Q first crosses P_r from right to left. Consider the path $S = Q[u, w_{out}]P_r[w_{out}, v_r]$. We claim that Q does not cross S. To see this, assume the contrary and let w' denote the exit point corresponding to the crossing. Since Q is simple, $w' \notin Q[u, w_{out}]$. So $w' \in P_r[w_{out}, v_r]$, but then $Q[u, w_{out}]P_r[w_{out}, w']Q[w', v]$ is a shortest path from u to v in R that crosses P_r and P_ℓ fewer times than Q. But this contradicts the minimality of Q.
  Since Q first crosses P_r from right to left and never crosses S, its first crossing with P_ℓ must be right-to-left as well, see Figure 3(b). This implies that Q enters all root-to-leaf paths in T before (not strictly before) it enters P_ℓ. In particular, Q enters T_v. Let x be the entry vertex. Then $Q[u, x]T_v[x, v]$ is a u-to-v shortest path in R that does not cross P_ℓ. $\square$

The algorithm: We can now describe our Bellman-Ford algorithm to relax all edges from vertices of C_1 to vertices of C_2. Pseudocode is shown in Figure 4.

Assume that R_{P_l} and R_{P_r} and distances between pairs of boundary vertices in these graphs have been precomputed. In each iteration j, we relax edges from vertices of V_{C_1} to all $v \in V_{C_2}$ in R_{P_ℓ} and in R_{P_r} (lines 9 and 10). Lemma 5 implies that this corresponds to relaxing all edges in R from vertices of V_{C_1} to vertices of V_{C_2}. By the results in Section 4.2, this suffices to show the correctness of the algorithm.

Lemma 4 shows that lines 9, 10 can each be implemented to run in $O(|V_{C_1}| + |V_{C_2}|)$ time. Thus, each iteration of lines 6–10 takes $O((|V_{C_1}| + |V_{C_2}|)\alpha(|V_{C_1}| + |V_{C_2}|))$ time, as desired.

```
1.  initialize vector $e_j[v]$ for $j = 0, \ldots, b$ and $v \in B$
2.  $e_j[v] := \infty$ for all $v \in B$ and $j = 0, \ldots, b$
3.  $e_0[r] := 0$
4.  for $j = 1, \ldots, b$
5.    for each region $R \in \mathcal{R}$
6.      for each pair of cycles, $C_1$ and $C_2$, defining the boundary of $R$
7.        if $C_1 = C_2$, relax edges from $C_1$ to $C_2$ as in Section 4.2
8.        else (assume $C_1$ is external and that $d_{R_{P_r}}$ and $d_{R_{P_\ell}}$ have been precomputed)
9.          $e_j[v] := \min\{e_j[v], \min_{w \in V_{C_1}}\{e_{j-1}[w] + d_{R_{P_r}}(w', v')\}\}$ for all $v \in V_{C_2}$
10.         $e_j[v] := \min\{e_j[v], \min_{w \in V_{C_1}}\{e_{j-1}[w] + d_{R_{P_\ell}}(w', v')\}\}$ for all $v \in V_{C_2}$
11. $D[v] := e_b[v]$ for all $v \in B$
```

Fig. 4. Pseudocode for the Bellman-Ford variant that handles regions with holes.

It remains to show that R_{P_r} and R_{P_ℓ} and distances between boundary vertices in these graphs can be precomputed in $O(|V_R| \log |V_R|)$ time Shortest path tree T in R with source r_1 can be found in $O(|V_R| \log |V_R|)$ time with Dijkstra using the feasible price function ϕ obtained from the recursively computed distances in R. Given T, we can find its rightmost path in $O(|V_R|)$ time by starting at the root r_1. When entering a vertex v using the edge uv, leave that vertex on the edge that comes after vu in counterclockwise order. Computing R_{P_r} given P_r also takes $O(|V_R|)$ time. We can next apply Klein's algorithm [5] to compute distances between all pairs of boundary vertices in R_{P_r} in $O(|V_R|\log|V_R|)$ time (here, we use the non-negative edge lengths in R defined by the reduced cost function induced by ϕ). We similarly compute P_ℓ and pairwise distances between boundary vertices in R_{P_ℓ}. We can finally state our result.

Theorem 1. *Given a planar directed graph G with real edge lengths and no negative cycles and given a source vertex s, we can find SSSP distances in G with source s in $O(n\log^2 n/\log\log n)$ time and linear space.*

5 Concluding Remarks

We gave a linear space algorithm for single-source shortest path distances in a planar directed graph with arbitrary real edge lengths and no negative cycles. The running time is $O(n\log^2 n/\log\log n)$, which improves on the previous bound by a factor of $\log\log n$. As corollaries, bipartite planar perfect matching, feasible flow, and feasible circulation in planar graphs can be solved in $O(n\log^2 n/\log\log n)$ time. The true complexity of the problem remains unsettled as there is a gap between our upper bound and the linear lower bound. Is $O(n\log n)$ time achievable?

References

1. E. W. Chambers, J. Erickson, and A. Nayyeri. Homology flows, cohomology cuts. Proc. 42nd Ann. ACM Symp. Theory Comput., 273–282, 2009.
2. J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. Available from the authors' webpages. Preliminary version in FOCS'01.
3. M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. Journal of Computer and System Sciences, 55(1):3–23, 1997.
4. M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. SIAM Journal On Discrete Math, 3(1):81–97, 1990.
5. P. N. Klein. Multiple-source shortest paths in planar graphs. Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 146–155.
6. P. N. Klein, S. Mozes, and O. Weimann. Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n\log^2 n)$-Time Algorithm. Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms, p. 236–245, 2009.
7. R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. SIAM Journal on Numerical Analysis, 16:346–358, 1979.
8. G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.
9. A. Aggarwal, M. Klawe, S. Moran, P. W. Shor, and R. Wilber. Geometric applications of a matrix searching algorithm. SCG '86: Proceedings of the second annual symposium on Computational geometry, 285–292, 1986.

Appendix

The proofs of Lemmas 2 and 4 are very similar to those in [6]. The (easy) proof of the space complexity in Theorem 1 was omitted due to lack of space.

Proof of Lemma 2

We need to show that after iteration j of the algorithm in Figure 1, $e_j[v]$ is the length of a shortest path in G from r to v that can be decomposed into at most j subpaths $P = P_1P_2P_3 \ldots P_j$, where the endpoints of each subpath P_i are boundary vertices and P_i is a shortest path in a region of $\mathcal{R}$.

The proof is by induction on $j \geq 0$ (and very similar to the proof of Lemma 4.2 in [6]). When $j = 0$, $e_j[r] = 0$ and $e_j[v] = \infty$ for all $v \in B \setminus \{r\}$ after line 3 and the base case holds.

Suppose $j > 0$ and that the lemma holds for $j - 1$. Consider a shortest path P in G from r to a $v \in B$ that can be decomposed into subpaths $P_1P_2P_3 \ldots P_j$, where the endpoints of each subpath P_i are boundary vertices and P_i is a shortest path in a region of $\mathcal{R}$. We need to show that after iteration j, $e_j[v]$ is the length of P.

Subpath $P' = P_1P_2 \ldots, P_{j-1}$ is a shortest path in G from r to a $w \in B$ which can be decomposed into at most $j - 1$ subpaths as above. Furthermore, there is a region $R \in \mathcal{R}$ such that v and w are boundary vertices of R and P_j is a shortest path in R from w to v.

At some point in iteration j, we reach line 7 with C being the cycle defining the boundary of R and $v, w \in V_C$. By the induction hypothesis, $e_{j-1}[w]$ is the length of P'. Since $e_j[v]$ is set to a value of at most $e_{j-1}[w] + d_R(w, v)$, $e_j[v]$ is at most the length of P.

Let us show the other inequality. For any $w \in B$, $e_{j-1}[w]$ is clearly the length of some path in G from r to w that can be decomposed into at most $j - 1$ subpaths, where each subpath is a shortest path in a region between two boundary vertices of that region. Hence, when $e_j[v]$ is updated in line 7, its value is the length of some path in G from r to v that can be decomposed into at most j such subpaths. This shows that $e_j[v]$ is at least the length of P, completing the proof.

Proof of Lemma 4

We need to show that relaxing all edges from V_{C_1} to V_{C_2} in R_P can be done in $O(|V_{C_1}| + |V_{C_2}|)$ time in any iteration of Bellman-Ford.

We only sketched a proof in the main paper. Let paths P_1 and P_2 and $|P_1| \times |P_2|$ matrix A be defined as in the proof sketch. We need to show that the column-minima of A can be found in $O(|V_{C_1}|+|V_{C_2}|)$ time. As shown in the main paper, this amounts to showing that for $1 \leq k < k' \leq |P_1|$ and $1 \leq l < l' \leq f(i)$, we have $A_{kl} + A_{k'l'} \geq A_{kl'} + A_{k'l}$.

Since the cycle $P_1 \overleftarrow{P} P_2 \overrightarrow{P}$ is the external face of R_P and since R_P is planar, any pair of paths in R_P from u'_k to v'_l and from $u'_{k'}$ to $v'_{l'}$ must intersect in some

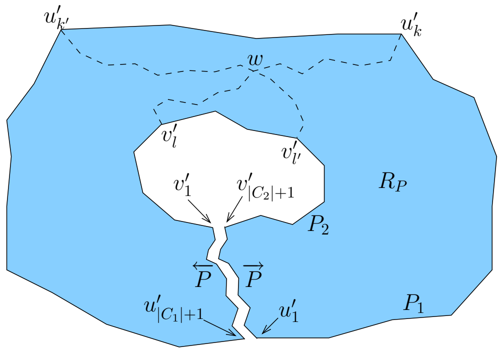


Fig. 5. The situation in the proof of Lemma 4. Any pair of paths in R_P from u'_k to v'_l and from $u'_{k'}$ to $v'_{l'}$ must intersect in some $w \in V_{R_P}$.

$w \in V_{R_P}$, see Figure 5. Let $b_k = e_{j-1}[u_k]$ and $b'_k = e_{j-1}[u_{k'}]$ (recall that we identified u_k with u'_k and $u_{k'}$ with $u'_{k'}$). Then

$$\begin{aligned}
A_{kl} + A_{k'l'} &= (b_k + d_{R_P}(u'_k, w) + d_{R_P}(w, v'_l)) + (b_{k'} + d_{R_P}(u'_{k'}, w) + d_{R_P}(w, v'_{l'})) \\
&= (b_k + d_{R_P}(u'_k, w) + d_{R_P}(w, v'_{l'})) + (b_{k'} + d_{R_P}(u'_{k'}, w) + d_{R_P}(w, v'_l)) \\
&\geq (b_k + d_{R_P}(u'_k, v'_{l'})) + (b_{k'} + d_{R_P}(u'_{k'}, v'_l)) \\
&= A_{kl'} + A_{k'l},
\end{aligned}$$

as requested.

Proof of Theorem 1

Proof. We analyzed the running time in the main paper. To bound the space, first note that finding an n/p-division of G using the algorithm of [2] requires $O(n)$ space. Klein's algorithm [5] and Dijkstra also has linear space requirement. The recursively computed distances take up a total of $O(p\frac{n}{p}) = O(n)$ space. In the intra-region boundary distances stage, the total memory used for storing distances is $O(p(\sqrt{n/p})^2) = O(n)$. In the single-source inter-region boundary distances stage, we need to bound the space for our Bellman-Ford variant. The size of each table is $O(b) = O(n)$. Since we only need to keep tables from the current and previous iteration in memory, Bellman-Ford uses $O(n)$ space. It is easy to see that the last two stages use $O(n)$ space. Hence the entire algorithm has linear space requirement.

Bounding the Expected Number of Rectilinear Full Steiner Trees

Christian Wulff-Nilsen
Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

Given a finite set Z of n points, called terminals, in $\mathbb{R}^d$, the Rectilinear Steiner Tree Problem asks for a tree of minimal L_1-length spanning Z. An optimal solution has a unique decomposition into full Steiner trees (FSTs). By using geometric properties and combinatorial arguments, we bound the expected number of FSTs satisfying simple necessary conditions for being part of an optimal solution. More specifically, we show that the expected number of FSTs spanning exactly K terminals and satisfying the empty lune property, a weak version of the bottleneck property, and the so-called empty hyperbox property is $O(n(\log\log n)^{2(d-1)})$ for $K = 3$ and $O(n(\log\log n)^{d-1}\log^{K-2} n)$ for $K > 3$, assuming terminals are randomly distributed in a hypercube with a uniform distribution. In the plane, we improve an earlier bound by showing that the expected number of FSTs with the Hwang form spanning exactly K terminals and satisfying the empty lune property and the so-called disjoint lunes property is $O(n\pi^K)$. © 2009 Wiley Periodicals, Inc. NETWORKS, Vol. 00(00), 000–000 2009

Keywords: rectilinear Steiner minimal tree; full component; expected bound; Hwang form; empty lune property; bottleneck property

1. INTRODUCTION

Let Z be a set of $n < \infty$ points, called *terminals*, in $\mathbb{R}^d$, $d \geq 2$. The *Rectilinear Steiner Tree Problem (RSTP)* asks for a tree of minimal L_1-length spanning Z. The RSTP is known to be NP-complete [3].

The RSTP is different from the minimum spanning tree problem since new points, called *Steiner points*, may be added to shorten the tree. An optimal solution to the RSTP is called a *rectilinear Steiner minimal tree (RSMT)*.

One of the main applications of the RSTP in the plane is in the area of VLSI design. Here, an important objective is to minimize the total length of wire interconnecting a set of pins. Typically, wires are restricted to having horizontal and vertical orientations only, making an RSMT of the set of pins a good candidate net.

At present, GeoSteiner [7] is probably the fastest exact algorithm for the RSTP in the plane. It uses a two-phase approach to construct an RSMT. In the first phase, a set $\mathcal{F}$ of so called full Steiner tree (FST) components is generated such that some subset of $\mathcal{F}$ is guaranteed to constitute the full components of an RSMT. In the second phase, full components of $\mathcal{F}$ are concatenated to form an RSMT; see [6].

In the first phase, pruning techniques are applied to minimize $|\mathcal{F}|$. An FST is pruned if it does not have certain properties. As FSTs of an RSMT can be assumed to have these properties, surviving FSTs can always be concatenated in the second phase to form an RSMT.

Although simple, the pruning techniques of GeoSteiner appear to be very powerful. Experimental results suggest that the expected number of surviving FSTs grows only linearly in n when terminals are randomly distributed in a square with a uniform distribution.

No theoretical linear bound has been found however. The best known bound on the expected number of surviving FSTs spanning a fixed number $K \geq 4$ of terminals in the plane is $O(n(\log\log n)^{K-2})$ [8]. Also, it has been shown that the expected number of FSTs spanning exactly two terminals is $O(n)$ (this bound also holds in higher dimensions) and that the expected number of FSTs spanning $\Omega(n)$ terminals is $O(1)$ [5]. All these bounds are obtained by assuming the above distribution of terminals. For surviving FSTs spanning exactly three terminals, an $O(n)$ (worst-case) bound is known for any distribution of terminals [2]. To the author's knowledge, no non-trivial bounds are known in dimensions higher than two.

In this article, we improve the bounds of [8] by showing that the expected number of surviving FSTs spanning a fixed number $K \geq 4$ of terminals in the plane is $O(n\pi^K)$, assuming a random distribution of terminals. This bound is achieved by requiring FSTs to have three properties, namely, the Hwang form, the empty lune property, and the so-called disjoint lunes property.

Furthermore, we bound the expected number of FSTs spanning exactly $K \geq 3$ terminals in dimension $d > 2$ when FSTs are required to have three simple necessary properties and when terminals are randomly distributed in a hypercube with a uniform distribution on each axis. We obtain a bound of $O(n(\log\log n)^{2(d-1)})$ for $K = 3$ and $O(n(\log\log n)^{d-1}\log^{K-2} n)$ for $K > 3$.

Received March 2008; accepted March 2009
Correspondence to: C. Wulff-Nilsen; e-mail: koolooz@diku.dk
DOI 10.1002/net.20342
Published online in Wiley InterScience (www.interscience.wiley.com).

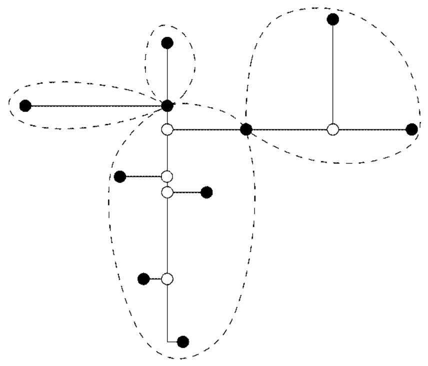


FIG. 1. An RSMT of a set of ten terminals. Black nodes are terminals and white nodes are Steiner points. The RSMT consists of four full components.

The article is organized as follows. In Section 2, we give various definitions and introduce some notation. In Section 3, we present the empty lune property and the disjoint lunes property and, using these properties, we prove our bound in the plane. In Section 4, we consider three properties which hold for FSTs of RSMTs in arbitrary dimensions, namely, the empty lune property, the bottleneck property, and the empty hyperbox property. These properties are then used to prove our bounds in higher dimensions. Finally, we make some concluding remarks in Section 5.

2. DEFINITIONS AND NOTATION

If $A \subseteq \mathbb{R}^d$ is a subset of $\mathbb{R}^d$, we denote the interior of A by A°. If $\boldsymbol{x} \in \mathbb{R}^d$ is a point in $\mathbb{R}^d$, we write its ith component as $\boldsymbol{x}[i]$. We define $\pi_i : \mathbb{R}^d \to \mathbb{R}$ to be the ith projection map, $(\boldsymbol{x}[1], \dots, \boldsymbol{x}[d]) \mapsto \boldsymbol{x}[i]$.

Given two points $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^d$, we write the L_1- and L_2-distance between them as $L_1(\boldsymbol{p}, \boldsymbol{q})$ and $L_2(\boldsymbol{p}, \boldsymbol{q})$, respectively. The L_1- resp. L_2-length of a vector $\boldsymbol{v}$ is written as $\|\boldsymbol{v}\|_1$ resp. $\|\boldsymbol{v}\|_2$.

In the following, we will assume that we are given a set Z of n points, called *terminals*, in $\mathbb{R}^d$, $d \geq 2$, and that these terminals are randomly distributed in an axis-aligned hypercube, say in $\mathcal{U} = [0, 1]^d$, with a uniform distribution on each axis.

We define a *Steiner tree* as a tree spanning a subset of Z and possibly containing Steiner points. As we are interested in minimal length networks, we may assume that no Steiner point has degree less than three. In fact, we will require Steiner points to have degree exactly three. This gives no loss in generality because we allow degenerate trees, i.e., trees with zero-length edges.

A *full Steiner tree (FST)* is a Steiner tree in which all terminals are leaves and all edges ending at terminals have length strictly greater than zero. Any RSMT of Z has a unique decomposition into FSTs; see Figure 1. We refer to an FST spanning exactly $K \geq 2$ terminals as a *K-FST*.

We call a property of an FST *necessary* if all FSTs of any RSMT have this property.

Given points $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^d$, the *lune* $L(\boldsymbol{p}, \boldsymbol{q})$ of $\boldsymbol{p}$ and $\boldsymbol{q}$ is defined as the set of points with distance at most $L_1(\boldsymbol{p}, \boldsymbol{q})$ to both $\boldsymbol{p}$ and $\boldsymbol{q}$, i.e., $L(\boldsymbol{p}, \boldsymbol{q}) = \{\boldsymbol{r} \in \mathbb{R}^d \mid \max\{L_1(\boldsymbol{p}, \boldsymbol{r}), L_1(\boldsymbol{q}, \boldsymbol{r})\} \leq L_1(\boldsymbol{p}, \boldsymbol{q})\}$. We define the lune of an edge of a Steiner tree to be the lune of the endpoints of the edge and we define the lunes of a Steiner tree to be the lunes of its edges.

Let $\boldsymbol{e} = (\boldsymbol{u}, \boldsymbol{v})$ be an edge of some Steiner tree embedded in $\mathbb{R}^d$. Where appropriate, we will regard $\boldsymbol{e}$ as representing any shortest (w.r.t. the L_1-metric) path between $\boldsymbol{u}$ and $\boldsymbol{v}$. It is easy to see that the union of all these shortest paths is the smallest axis-aligned hypercube containing $\boldsymbol{u}$ and $\boldsymbol{v}$. If $\boldsymbol{p} \in \mathbb{R}^d$ is a point then we define the distance $L_1(\boldsymbol{e}, \boldsymbol{p}) = L_1(\boldsymbol{p}, \boldsymbol{e})$ between $\boldsymbol{e}$ and $\boldsymbol{p}$ to be the distance between $\boldsymbol{p}$ and this hypercube.

We say that a subet S of $\mathbb{R}^d$ is *empty* if the interior S° of S contains no terminals of Z.

3. FSTS IN TWO DIMENSIONS

We start by considering FSTs in the plane. In this section, we prove that the expected number of K-FSTs having the Hwang form and satisfying the empty lune property and the disjoint lunes property is $O(n\pi^K)$ for $K \geq 4$.

3.1. The Hwang Form

Hwang [4] showed that FSTs of an RSMT in the plane can be assumed to have a very restricted form, the *Hwang form*. An FST with this form consists of a *backbone* defined by two terminals, a *root* z_1 and a *tip* z_k; see Figure 2. The backbone consists of a long and a short leg. Terminal z_1 is incident to the long leg and z_k is incident to the short leg. Alternating line segments attach terminals to the long leg. There are two main types of FSTs with the Hwang form: a *type one FST* has no terminals attached to the short leg (except the tip) and a *type two FST* has exactly one terminal attached to the short leg (in addition to the tip).

In Ref. [8], the algorithm used by GeoSteiner to generate FSTs is presented. It grows FSTs along their backbones as follows. First, a root and a direction of the long leg is selected. Then a line is swept in the direction of the long leg

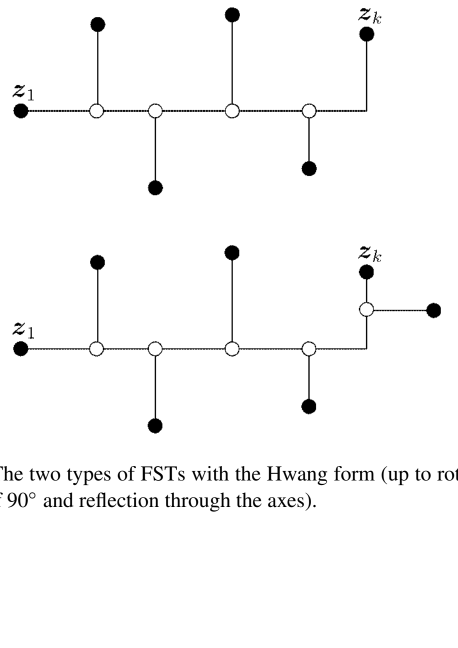


FIG. 2. The two types of FSTs with the Hwang form (up to rotation by a multiple of 90° and reflection through the axes).

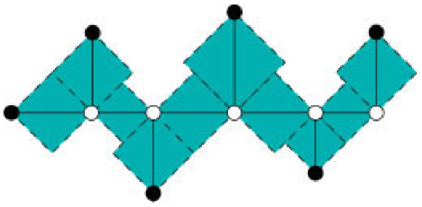


FIG. 3. The lunes of the edges of a partially grown FST. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

and terminals intersected by this line are recursively attached to the backbone. At each step, various pruning techniques are applied to the partially generated FST. If pruned, the partial FST is not grown any further. If a partial FST defines an FST, it is stored in a set of candidate FSTs.

This idea of growing FSTs will prove useful in our analysis in the next section.

3.2. *Obtaining the Bound*

To obtain our bound, we need two simple necessary properties of FSTs in two dimensions, the *empty lune property* and the *disjoint lunes property*. The first property requires lunes of an FST to be empty and the second property requires lunes of an FST to have disjoint interiors. An example of the lunes of a partially grown FST is shown in Figure 3.

To show that the empty lune property is a necessary property of FSTs, suppose for the sake of contradiction that a terminal z belongs to the interior of some lune $L(\boldsymbol{u}, \boldsymbol{v})$ of an RSMT T. The removal of $(\boldsymbol{u}, \boldsymbol{v})$ splits T into two components, one containing $\boldsymbol{u}$ and one containing $\boldsymbol{v}$. Assume w.l.o.g. that z and $\boldsymbol{u}$ belong to the same component. Then adding edge $(\boldsymbol{v}, z)$ reconnects T and shortens its L_1-length, a contradiction.

The following lemma shows that the disjoint lunes property is also a necessary property of FSTs.

Lemma 1. *The lunes of two distinct edges belonging to the same FST of an RSMT in the plane have disjoint interiors.*

Proof. If one of the edges is a backbone edge or if the two edges are not on the same side of the backbone then the lemma is clearly satisfied.

Now, consider two edges $(\boldsymbol{s}, z)$ and $(\boldsymbol{s}', z')$, where $\boldsymbol{s}$ and $\boldsymbol{s}'$ are Steiner points on the backbone and z and z' are terminals on the same side of the backbone; see Figure 4. Assume w.l.o.g. that the backbone is horizontal, that $L_1(\boldsymbol{s}, z) \geq L_1(\boldsymbol{s}', z')$, and that $(\boldsymbol{s}, z)$ is to the left of $(\boldsymbol{s}', z')$. Let $z\boldsymbol{p}$ be the horizontal line segment to the left of z having length $L_1(\boldsymbol{s}, z) = L_2(\boldsymbol{s}, z)$.

Suppose for the sake of contradiction that $L(\boldsymbol{s}, z)$ and $L(\boldsymbol{s}', z')$ share interior points. Then z' must belong to triangle $\Delta \boldsymbol{s}z\boldsymbol{p}$ and not to line segment $\boldsymbol{sp}$. This implies that $L_1(z, z') < L_1(z, \boldsymbol{s})$. Removing edge $(\boldsymbol{s}, z)$ and adding edge (z, z') therefore shortens the RSMT, a contradiction. ■

Let $K \geq 4$ be given. To prove our bound for K, suppose we have grown a partial FST F with terminals $z_1, \ldots, z_{k-1}$, where z_{k-1} is the last terminal added. Let z_k be the next terminal to be added.

Lemma 2. *With the above definitions, the expected number of candidates for z_k is at most π if z_k is neither the tip nor a terminal attached to the short leg of the FST.*

Proof. Recall that $\mathcal{U}$ is the square in which the n terminals are distributed and let $\mathcal{L}$ be the union of lunes of edges in the partially grown FST F. Then the remaining $n - k + 1$ terminals are randomly distributed in $\mathcal{U} \setminus \mathcal{L}$ with a uniform distribution. Let A be the area of $\mathcal{U} \setminus \mathcal{L}$.

Let $z_k = (x_k, y_k)$ be a candidate terminal. For convenience, assume that the Steiner point $\boldsymbol{s}_{k-1}$ attached to z_{k-1} is located at the origin and that z_k belongs to the first quadrant. Letting $\boldsymbol{s}_k$ be the new Steiner point attached to $\boldsymbol{s}_{k-1}$ and z_k, we have $\boldsymbol{s}_k = (x_k, 0)$ (here we use the assumption that z_k is not a tip and not a terminal attached to the short leg of the FST).

As z_k is a candidate terminal, lunes $L(\boldsymbol{s}_{k-1}, \boldsymbol{s}_k)$ and $L(\boldsymbol{s}_k, z_k)$ contain no terminals in their interiors.

The area of $L(\boldsymbol{s}_{k-1}, \boldsymbol{s}_k) \cup L(\boldsymbol{s}_k, z_k)$ is $\frac{1}{2}(x_k^2 + y_k^2) = \frac{1}{2}r^2$, where r is the Euclidean distance from z_k to the origin. As $\boldsymbol{s}_{k-1}, \boldsymbol{s}_k, z_k \in \mathcal{U}$, at least half of $L(\boldsymbol{s}_{k-1}, \boldsymbol{s}_k) \cup L(\boldsymbol{s}_k, z_k)$ is contained in $\mathcal{U}$. Thus, by Lemma 1, at least half of $L(\boldsymbol{s}_{k-1}, \boldsymbol{s}_k) \cup L(\boldsymbol{s}_k, z_k)$ is contained in $\mathcal{U} \setminus \mathcal{L}$.

By the above, if a terminal in the first quadrant has Euclidean distance r to the origin, the probability that it is a z_k-candidate is no more than

$$\left(1 - \frac{r^2}{4A}\right)^{n-k},$$

and we see that $1 - r^2/(4A) > 0$, giving the upper bound $2\sqrt{A}$ on r.

Given $r, h > 0$, the region

$$C(r, h) = \{\boldsymbol{p} \in \mathbb{R}_+^2 | r \leq \|\boldsymbol{p}\|_2 \leq r + h\}$$

has area

$$\frac{\pi}{4}((r+h)^2 - r^2) = \frac{\pi}{2}rh + \frac{\pi}{4}h^2.$$

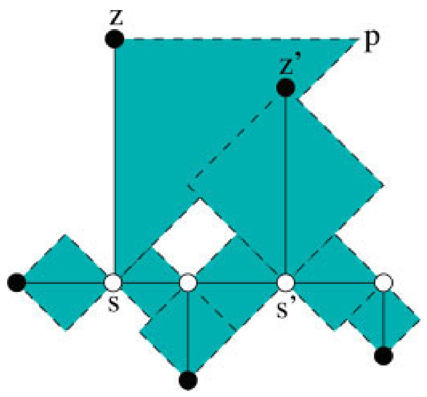


FIG. 4. The lunes of an FST of an RSMT have disjoint interiors. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

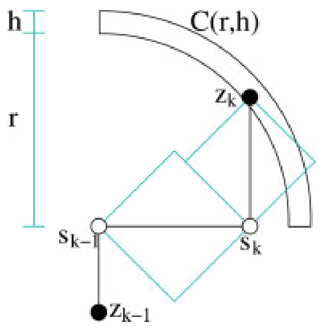


FIG. 5. The situation in the proof of Lemma 2. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

A terminal in $C(r,h)$ has Euclidean distance at least r to the origin; see Figure 5. By the above, the probability that it is a z_k-candidate is no more than $(1 - r^2/(4A))^{n-k}$.

As the expected number of terminals in $C(r,h)$ is at most

$$(n-k+1)\left(\frac{\pi}{2}rh + \frac{\pi}{4}h^2\right)/A,$$

the expected number of candidate terminals in $C(r,h)$ is bounded by

$$\left(1 - \frac{r^2}{4A}\right)^{n-k}(n-k+1)\left(\frac{\pi}{2}rh + \frac{\pi}{4}h^2\right)/A.$$

Integrating, we obtain a bound E on the expected number of z_k-candidates,

$$\begin{aligned} E &\le \int_{r=0}^{2\sqrt{A}} \left(1 - \frac{r^2}{4A}\right)^{n-k}(n-k+1)\frac{\pi}{2A}rdr \\ &= \frac{\pi(n-k+1)}{2A}\int_{r=0}^{2\sqrt{A}}\left(1 - \frac{r^2}{4A}\right)^{n-k} rdr \\ &= \frac{\pi(n-k+1)}{2A}\left[-2A\frac{\left(1-\frac{r^2}{4A}\right)^{n-k+1}}{n-k+1}\right]_{r=0}^{2\sqrt{A}} \\ &= \pi. \end{aligned}$$

■

Using Lemma 2, we are now ready for the main result of this section.

Theorem 1. *Given n terminals randomly distributed in a square with a uniform distribution, the expected number of K-FSTs with the Hwang form satisfying the empty lune property and the disjoint lunes property is* $O(n\pi^K)$ *for any* $K \ge 4$.

Proof. Let us fix a root z_1, a direction of the backbone, and the quadrant of z_1 containing the first backbone terminal z_2. As there are n possible choices of z_1, four directions of the backbone, and two possible quadrants of z_1 containing z_2 for each of these four directions, the theorem will follow if we can show that the expected number of K-FSTs with these fixed choices is $O(\pi^K)$.

By symmetry, assume that the backbone direction is to the right and denote the terminals from left to right in the K-FST by $z_1, \ldots, z_K$.

First, let us count the expected number of type one K-FSTs satisfying the above. By Lemma 2, the total expected number of choices for $z_2, \ldots, z_{K-1}$ is at most π^{K-2}. By regarding the point at which the long and the short legs meet as a degree two Steiner point, it follows from the proof of Lemma 2 that the expected number of choices for z_K is at most π when the other $K-1$ terminals are fixed. Hence, the expected number of type one K-FSTs with the above fixed choices is $O(\pi^{K-1})$.

Now, let us count the expected number of type two K-FSTs satisfying the above. Lemma 2 implies that the total expected number of choices for $z_2, \ldots, z_{K-1}$ is at most π^{K-2}. Hence, assuming $z_1, \ldots, z_{K-2}$ are fixed, it remains to show that the expected number of choices of pair (z_{K-1}, z_K) is bounded by some constant independent of K. Note that, as terminals are ordered from left to right, z_{K-1} is the tip and z_K is attached to the short leg.

By symmetry, we may assume that the tip z_{K-1} is above the long leg; see Figure 6. Let $\boldsymbol{s}_{K-2}$ and $\boldsymbol{s}_{K-1}$ be the Steiner points attached to z_{K-2} and z_{K-1}, respectively, and let $\boldsymbol{s}$ be the point at which the long and the short leg meet. We will regard $\boldsymbol{s}$ as a degree two Steiner point.

Recall that $\mathcal{U}$ is the square in which the n terminals are distributed and let $\mathcal{L}$ be the union of lunes in the partially grown FST defined by terminals $z_1, \ldots, z_{K-2}$. Then the remaining $n-K+2$ terminals are randomly distributed in $\mathcal{U} \setminus \mathcal{L}$ with a uniform distribution. Let A be the area of $\mathcal{U} \setminus \mathcal{L}$.

Consider a random choice of z_{K-1} and z_K such that the resulting K-FST has the Hwang form. Define $\hat{L}(z_{K-1}, \boldsymbol{s}_{K-1})$ and $\check{L}(z_{K-1}, \boldsymbol{s}_{K-1})$ as the upper and lower half of $L(z_{K-1}, \boldsymbol{s}_{K-1})$, respectively. Observe that as Steiner points and terminals belong to $\mathcal{U}$ and as we assume that our FST satisfies the disjoint lunes property, at least half of each of the lunes $L(\boldsymbol{s}_{K-2}, \boldsymbol{s})$, $L(\boldsymbol{s}, \boldsymbol{s}_{K-1})$, $L(z_{K-1}, \boldsymbol{s}_{K-1})$, and $L(z_K, \boldsymbol{s}_{K-1})$ are contained in $\mathcal{U} \setminus \mathcal{L}$. Furthermore, at least half of $\check{L}(z_{K-1}, \boldsymbol{s}_{K-1})$ and at least half of $\hat{L}(z_{K-1}, \boldsymbol{s}_{K-1})$ are contained in $\mathcal{U} \setminus \mathcal{L}$.

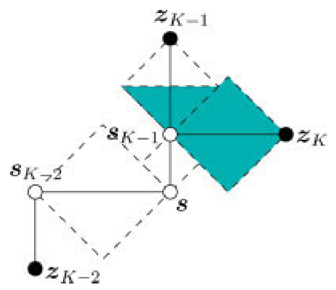


FIG. 6. The situation in Theorem 1 for type two FSTs. Lunes of edges are shown. The coloured area shows the set $\check{L}(z_{K-1}, s_{K-1}) \cup L(z_K, s_{K-1})$. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

The area of $\hat{L}(z_{K-1}, s_{K-1}) \cup L(s, s_{K-1})$ is at least 1/8 of the area of $L(s, z_{K-1})$. Thus, letting r_1 denote the Euclidean distance between s_{K-2} and z_{K-1}, the probability that $S_1 = L(s_{K-2}, s) \cup L(s, s_{K-1}) \cup \hat{L}(z_{K-1}, s_{K-1})$ contains no terminals in its interior is less than

$$\left(1 - \frac{r_1^2}{16A}\right)^{n-K}.$$

Similarly, letting r_2 denote the Euclidean distance between z_{K-1} and z_K, the probability that $S_2 = \check{L}(z_{K-1}, s_{K-1}) \cup L(z_K, s_{K-1})$ contains no terminals in its interior is less than

$$\left(1 - \frac{r_2^2}{16A}\right)^{n-K}.$$

As $S_1 \cap S_2 = \emptyset$, the probability that lunes $L(s_{K-2}, s)$, $L(s, s_{K-1})$, $L(z_{K-1}, s_{K-1})$, and $L(z_K, s_{K-1})$ contain no terminals in their interiors is less than

$$\left(1 - \frac{r_1^2}{16A}\right)^{n-K}\left(1 - \frac{r_2^2}{16A}\right)^{n-K}.$$

Integrating w.r.t. r_1 and r_2, it follows easily from the proof of Lemma 2 that the expected number of choices of (z_{K-1}, z_K) is less than $(4\pi)^2$. ∎

4. HIGHER DIMENSIONS

We now turn our attention to $d > 2$ dimensions. In this section, we prove that the expected number of K-FSTs having the empty lune property, a weak version of the bottleneck property, and the empty hyperbox property is $O(n(\log\log n)^{2(d-1)})$ for $K = 3$ and $O(n(\log\log n)^{d-1}\log^{K-2} n)$ for $K > 3$.

We start by presenting the three properties and show that they are necessary properties of FSTs in arbitrary dimensions. The empty lune property and the (weak) bottleneck property are well known in two dimensions and are immediately generalized to arbitrary dimensions. The empty hyperbox property is new and we shall prove that it is in fact a necessary property.

4.1. Empty Lune Property

Recall from Section 3.2 that a Steiner tree has the empty lune property if each of its lunes is empty. The proof that this property is a necessary property of FSTs is immediately generalized to arbitrary dimensions.

4.2. Bottleneck Property

Consider two terminals $z_1, z_2 \in Z$ and let P be a simple path between z_1 and z_2 in the complete graph C of Z in the space $(\mathbb{R}^d, L_1)$. Let l_P denote the length of a longest edge on P and let $\mathcal{P}(z_1, z_2)$ be the set of all simple paths in C between z_1 and z_2. Then the *bottleneck Steiner distance* $l_b(z_1, z_2)$ between z_1 and z_2 is defined as

$$l_b(z_1, z_2) = \min\{l_P | P \in \mathcal{P}(z_1, z_2)\}. \qquad (1)$$

We say that a Steiner tree T satisfies the *bottleneck property* if for all pairs of terminals z_1 and z_2 in T, all edges on the path in T between z_1 and z_2 have length at most $l_b(z_1, z_2)$.

It has been shown that in the plane, any RSMT of Z satisfies the bottleneck property. In particular, any FST of an RSMT satisfies this property, implying that it is a necessary property of FSTs. Furthermore,

$$l_b(z_1, z_2) = l_{\mathrm{MST}}(z_1, z_2)$$

for all pairs of terminals $z_1, z_2 \in Z$, where $l_{\mathrm{MST}}(z_1, z_2)$ is the length of a longest edge on the path between z_1 and z_2 in an MST of Z in $(\mathbb{R}^d, L_1)$. These two results are easily generalized to arbitrary dimensions.

We say that a Steiner tree satisfies the *weak bottleneck property* if all its edges have length at most the length of a longest MST edge. Note that a Steiner tree having the bottleneck property also has the weak bottleneck property. Hence, the weak bottleneck property is a necessary property of FSTs.

4.3. Empty Hyperbox Property

Given points $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^d$, we define $B(\boldsymbol{p}, \boldsymbol{q})$ to be the smallest axis-parallel hyperbox containing $\boldsymbol{p}$ and $\boldsymbol{q}$.

If T is a Steiner tree, we say that it satisfies the *empty hyperbox property* if for all edges $(\boldsymbol{u}, \boldsymbol{w})$ and $(\boldsymbol{v}, \boldsymbol{w})$ of T incident to the same Steiner point $\boldsymbol{w}$, the hyperbox $B(\boldsymbol{u}, \boldsymbol{v})$ is empty; see Figure 7.

The following lemma shows that the empty hyperbox property is a necessary property of FSTs. It can be viewed as a generalization of the empty corner rectangle property in the plane presented in [8].

Lemma 3. *Any RSMT of Z satisfies the empty hyperbox property.*

Proof. Let $(\boldsymbol{u}, \boldsymbol{w})$ and $(\boldsymbol{v}, \boldsymbol{w})$ be edges of an RSMT of Z incident to the same Steiner point $\boldsymbol{w}$. Suppose for the sake of contradiction that z is a terminal in $B(\boldsymbol{u}, \boldsymbol{v})^\circ$. Steiner point $\boldsymbol{w}$ must belong to $B(\boldsymbol{u}, \boldsymbol{v})$ for otherwise, edges $(\boldsymbol{u}, \boldsymbol{w})$ and $(\boldsymbol{v}, \boldsymbol{w})$ would overlap, contradicting the optimality of the RSMT of Z containing the edges. It follows that

$$L_1(\boldsymbol{w}, \boldsymbol{u}) + L_1(\boldsymbol{w}, \boldsymbol{v}) = L_1(\boldsymbol{u}, \boldsymbol{v}).$$

Either $L_1(z, \boldsymbol{u}) \le L_1(\boldsymbol{w}, \boldsymbol{u})$ or $L_1(z, \boldsymbol{v}) \le L_1(\boldsymbol{w}, \boldsymbol{v})$ for otherwise, we would have

$$\begin{aligned} &L_1(\boldsymbol{u}, \boldsymbol{v}) \\ &\quad = L_1(z, \boldsymbol{u}) + L_1(z, \boldsymbol{v}) > L_1(\boldsymbol{w}, \boldsymbol{u}) + L_1(\boldsymbol{w}, \boldsymbol{v}) = L_1(\boldsymbol{u}, \boldsymbol{v}). \end{aligned}$$

By symmetry, we may assume that $L_1(z, \boldsymbol{u}) \le L_1(\boldsymbol{w}, \boldsymbol{u})$; see Figure 8. Consider removing edge $(\boldsymbol{w}, \boldsymbol{u})$. This splits the

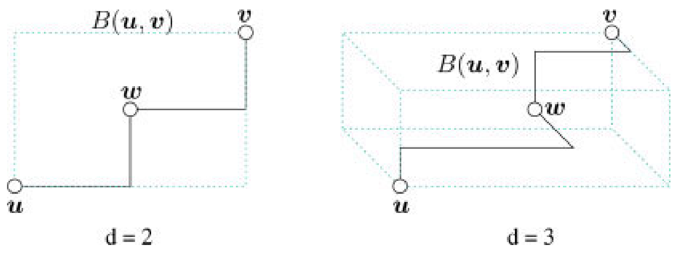


FIG. 7. Hyperbox $B(\boldsymbol{u}, \boldsymbol{v})$. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

RSMT into two components, $C_{\boldsymbol{u}}$ containing $\boldsymbol{u}$ and $C_{\boldsymbol{w}}$ containing $\boldsymbol{w}$. We consider two cases, $z \in C_{\boldsymbol{u}}$ and $z \in C_{\boldsymbol{w}}$, and we will show that we reach a contradiction in both cases.

Assume first that $z \in C_{\boldsymbol{u}}$. On each axis c, $\boldsymbol{u}[c]$ is the point in $\pi_c(B(\boldsymbol{u},\boldsymbol{v}))$ farthest from the projection of edge $(\boldsymbol{w},\boldsymbol{v})$ on axis c, as shown in Figure 8. Since $z \in B(\boldsymbol{u},\boldsymbol{v})^\circ$,

$$L_1(z,(\boldsymbol{w},\boldsymbol{v})) < L_1(\boldsymbol{u},(\boldsymbol{w},\boldsymbol{v})),$$

so connecting $\boldsymbol{z}$ to edge $(\boldsymbol{w},\boldsymbol{v})$ reconnects the tree and decreases the length of the tree, a contradiction. Note that we have disregarded the degeneracy where $\boldsymbol{w} = \boldsymbol{u}$, as in this case, the empty lune property is violated.

Now, assume that $z \in C_{\boldsymbol{w}}$. As $L_1(z,\boldsymbol{u}) \le L_1(\boldsymbol{w},\boldsymbol{u})$, reconnecting the tree by adding edge $(z,\boldsymbol{u})$ decreases the length of the tree unless $L_1(z,\boldsymbol{u}) = L_1(\boldsymbol{w},\boldsymbol{u})$.

So assume $L_1(z,\boldsymbol{u}) = L_1(\boldsymbol{w},\boldsymbol{u})$. In the RSMT, z has at least one incident edge. As this edge may be embedded such that it consists of axis-parallel line segments, there is a point $\boldsymbol{p} \in B(\boldsymbol{u},\boldsymbol{v})^\circ$ on this edge such that $L_1(\boldsymbol{p},\boldsymbol{u}) \neq L_1(\boldsymbol{w},\boldsymbol{u})$. Repeating the above for $\boldsymbol{p}$ instead of z, we again obtain a shorter tree. ∎

4.4. *Half FSTs*

In our analysis, we will regard the topology of an FST as a terminal connected by an edge to the root of a binary tree $\mathcal{T}$. All leaves in $\mathcal{T}$ are terminals and all interior vertices are Steiner points.

We refer to binary trees like $\mathcal{T}$ as *half FSTs (HFSTs)*. An HFST spanning exactly K terminals is called a *K-HFST*. A 1-HFST is a terminal and for $K > 1$, a K-HFST is obtained by *merging* a K_1-HFST and a K_2-HFST, where $K_1 + K_2 = K$, i.e., by connecting the roots of the two HFSTs to a common Steiner point.

Consider an FST T with topology $\mathcal{T}$. An HFST contained in $\mathcal{T}$ induces a subtree in T. We also refer to this subtree as an HFST and as a K-HFST if it spans exactly K terminals.

We say that an (H)FST is *interesting* if it satisfies the empty lune property, the weak bottleneck property, and the empty hyperbox property. As the three properties are necessary, only interesting (H)FSTs can be part of an RSMT of Z.

Note that all 1-HFSTs are trivially interesting. We say that two interesting HFSTs are *mergable* if the HFST obtained by merging them is interesting.

In the following, we will bound the expected number of interesting (H)FSTs.

4.5. *Bounding the Longest MST Edge*

To make use of the weak bottleneck property, we need a bound on the expected length of a longest MST edge. The main ideas in the proofs of this section are similar to those in [8].

We will need the following well-known result which we state as a lemma.

Lemma 4. *For any point $\boldsymbol{p} \in \mathbb{R}^d$, $\|\boldsymbol{p}\|_1 \le \sqrt{d}\|\boldsymbol{p}\|_2$.*

The following lemma bounds the probability of long MST edges.

Lemma 5. *Suppose $n > 4$, let D be a constant and let $C = D\sqrt[d]{\log n/n}$. The probability that there exists an MST edge (z_i, z_j) in the L_1-metric such that $L_1(z_i,z_j) > C$ is bounded by $n^{2-D^d/(2(2d)^d)}$.*

Proof. Let $\boldsymbol{e} = (z_i, z_j)$ be a pair of distinct terminals. We will bound the probability that $\boldsymbol{e}$ is a long MST edge.

Suppose $\boldsymbol{e}$ is an MST edge and let $\boldsymbol{m} = \frac{1}{2}(z_i + z_j)$ be the midpoint of $\boldsymbol{e}$. Define

$$\overline{H} = \left\{\boldsymbol{p} \in \mathbb{R}^d | L_1(\boldsymbol{p},\boldsymbol{m}) \le \frac{1}{2}L_1(z_i,z_j)\right\},$$

$$B = \left\{\boldsymbol{p} \in \mathbb{R}^d | L_2(\boldsymbol{p},\boldsymbol{m}) \le \frac{1}{2\sqrt{d}}L_1(z_i,z_j)\right\}.$$

See Figure 9 for an illustration when $d = 2$.

If $\boldsymbol{p} \in B$ then by Lemma 4,

$$L_1(\boldsymbol{p},\boldsymbol{m}) \le \sqrt{d}L_2(\boldsymbol{p},\boldsymbol{m}) \le \frac{1}{2}L_1(z_i,z_j),$$

showing that $B \subseteq \overline{H}$.

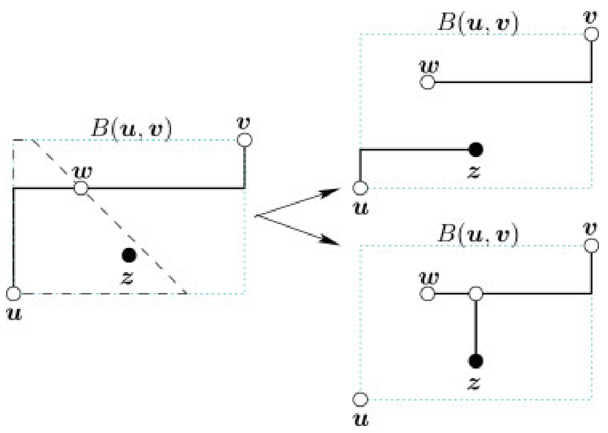


FIG. 8. When edge $(\boldsymbol{w},\boldsymbol{u})$ is removed, the RSMT may be linked in one of the two ways shown. In both cases, the length of the tree is reduced. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

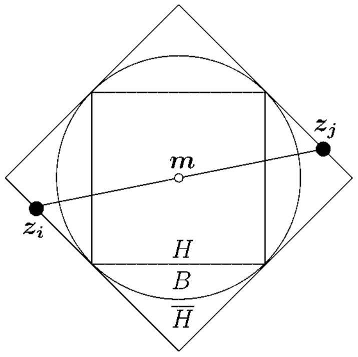


FIG. 9. The three sets $\overline{H}$, B, and H considered in the proof of Lemma 5, here shown for $d = 2$.

Let H be the hypercube with center $\boldsymbol{m}$ and sidelength $\frac{1}{d}L_1(z_i, z_j)$, i.e.,

$$H = \left[\boldsymbol{m}[1] - \frac{1}{2d}L_1(z_i,z_j), \boldsymbol{m}[1] + \frac{1}{2d}L_1(z_i,z_j)\right] \times \cdots \times \left[\boldsymbol{m}[d] - \frac{1}{2d}L_1(z_i,z_j), \boldsymbol{m}[d] + \frac{1}{2d}L_1(z_i,z_j)\right]. \quad (2)$$

If $\boldsymbol{p} \in H$ then

$$L_2(\boldsymbol{p},\boldsymbol{m}) \le \sqrt{d\left(\frac{1}{2d}L_1(z_i,z_j)\right)^2} = \frac{1}{2\sqrt{d}}L_1(z_i,z_j),$$

so $\boldsymbol{p} \in B$. Thus, we have the inclusions $H \subseteq B \subseteq \overline{H}$. Note that as $L_1(z_i, z_j) > 0$ and $d > 1$,

$$\frac{1}{2}L_1(z_i,z_j) > \frac{1}{2\sqrt{d}}L_1(z_i,z_j),$$

showing that $z_i, z_j \notin H$.

Consider some axis c, let $\mathcal{U}_c = \pi_c(\mathcal{U})$ be the projection of the unit hypercube on axis c, and let $H_c = \pi_c(H)$. Suppose $H_c \nsubseteq \mathcal{U}_c$. We claim that then $\mathcal{U}_c \nsubseteq H_c$ holds.

To see this, assume otherwise. Then H has larger sidelength than $\mathcal{U}$. As we saw above, $z_i, z_j \notin H$. Hence, there is an axis c' such that $z_i[c'] \notin H_{c'} = \pi_{c'}(H)$. As $L_1(\boldsymbol{m}[c'], z_i[c']) = L_1(\boldsymbol{m}[c'], z_j[c'])$, we also have $z_j[c'] \notin H_{c'}$.

By assumption, $z_i, z_j \in \mathcal{U}$. Hence $H_{c'} \subset \mathcal{U}_{c'} = \pi_{c'}(\mathcal{U})$. This gives us

$$|\mathcal{U}_c| \le |H_c| = |H_{c'}| < |\mathcal{U}_{c'}| = |\mathcal{U}_c|,$$

a contradiction.

As $\boldsymbol{m} \in H \cap \mathcal{U}$, the above shows that at least half of H_c is contained in $\mathcal{U}_c$ if $H_c \nsubseteq \mathcal{U}_c$ and this clearly also holds if $H_c \subseteq \mathcal{U}_c$. It follows that at least $\frac{1}{2^d}$ of H is contained in $\mathcal{U}$.

The volume of H is $(\frac{1}{d}L_1(z_i, z_j))^d$ so the volume of $H \cap \mathcal{U}$ is at least

$$\left(\frac{1}{2d}L_1(z_i,z_j)\right)^d.$$

If $\boldsymbol{e}$ is an MST edge then by the empty lune property, $L(z_i, z_j)$ contains no terminals in its interior. Let $\boldsymbol{p} \in H$. Then

$$L_1(\boldsymbol{p}, z_i) \le d \cdot \frac{1}{d}L_1(z_i,z_j) = L_1(z_i,z_j)$$

and similarly, $L_1(\boldsymbol{p}, z_j) \le L_1(z_i, z_j)$. Thus, $H \subseteq L(z_i, z_i)$ and it follows that if $\boldsymbol{e}$ is an MST edge then H° contains no terminals. From this and from the above, we get

$$\begin{aligned}
&Pr(\boldsymbol{e} \text{ is an MST edge and } L_1(z_i,z_j) > C)\\
&\le Pr(H^\circ \text{ empty}|L_1(z_i,z_j) > C)\\
&\le \left(1 - \left(\frac{1}{2d}C\right)^d\right)^{n-2}\\
&= \left(1 - \frac{(n-2)\left(\frac{1}{2d}C\right)^d}{n-2}\right)^{n-2}\\
&\le e^{-(n-2)(1/(2d)C)^d}\\
&= e^{-(n-2)(1/(2d)D\sqrt[d]{\log n/n})^d}\\
&= n^{-(1/(2d)D)^d(n-2)/n}\\
&\le n^{-D^d/(2(2d)^d)},
\end{aligned}$$

where the last inequality follows from the assumption $n > 4$.

Finally, we obtain the desired bound on the probability of long MST edges:

$$Pr(\exists \text{ MST edge } (z_i,z_j) \text{ such that } L_1(z_i,z_j) > C) \le \binom{n}{2} n^{-D^d/(2(2d)^d)} \le n^{2-D^d/(2(2d)^d)}.$$

■

We conclude this section with the following corollary which bounds the expected length of a longest MST edge (and hence also the expected length of a longest RSMT edge).

Corollary 1. *With high probability, the length of a longest edge in MST(Z) under the L_1-metric is $O(\sqrt[d]{\log n/n})$. The expected length of a longest MST edge is $O(\sqrt[d]{\log n/n})$.*

Proof. Choose D such that $2 - D^d/(2(2d)^d) < -1/d$ (any value greater than $2d\sqrt[d]{4 + 2/d}$ will do). By Lemma 5, the probability that there exists an MST edge longer than $D\sqrt[d]{\log n/n}$ is bounded by $n^{2-D^d/(2(2d)^d)}$. By the choice of D, this probability approaches 0 as n approaches ∞, showing the first claim.

Note that, as all terminals are in $\mathcal{U}$, an MST edge can never be longer than d. Thus, the expected length of a longest MST edge is at most

$$\begin{aligned}
&(1 - n^{2-D^d/(2(2d)^d)})D\sqrt[d]{\log n/n} + n^{2-D^d/(2(2d)^d)}d\\
&\le D\sqrt[d]{\log n/n} + d\sqrt[d]{1/n}\\
&\le (D + d)\sqrt[d]{\log n/n}.
\end{aligned}$$

■

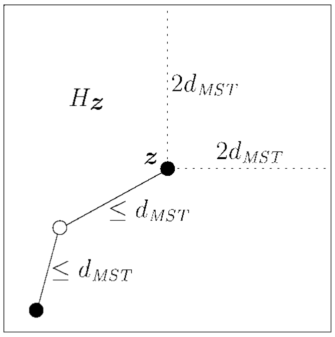


FIG. 10. All terminals mergable with z are confined to hypercube H_z.

4.6. *Bounding the Number of 2-HFSTs*

We will bound the expected number of interesting K-HFSTs, $K \geq 2$, as this will make it easy to bound the expected number of interesting FSTs. In this section, we focus on the case where $K = 2$.

Consider any interesting 2-HFST. As it satisfies the weak bottleneck property, the L_1-distance between the two terminals spanned by the HFST is at most two times the length d_{MST} of a longest edge in an MST of Z. Thus, if $z \in Z$ is a given terminal then all terminals mergable with z are confined to the hypercube H_z centered at z and having sidelength $4d_{MST}$ (in fact, they are confined to an even smaller set, namely, the L_1-ball centered at z and having radius $2d_{\text{MST}}$). See Figure 10.

By Corollary 1, the expected value of d_{MST} is at most $k\sqrt[d]{\log n/n}$ for some constant k. As terminals are uniformly distributed, the expected number of terminals in hypercube H_z is bounded by the volume of H_z times n (as $\mathcal{U}$ has volume 1), i.e.,

$$n(k\sqrt[d]{\log n/n})^d = O(\log n).$$

It follows that the expected number of interesting 2-HFSTs is $O(n \log n)$.

4.7. *Maximal Terminals*

In the following, we will obtain an even tighter bound on the expected number of interesting 2-HFSTs by considering empty hyperbox pruning as well.

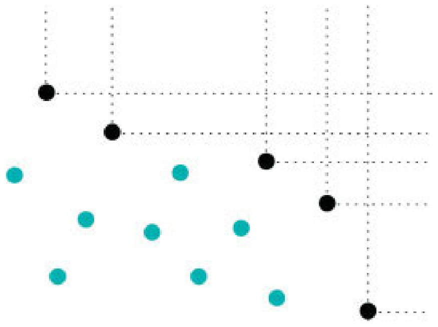


FIG. 11. A set of terminals in the plane of which five (marked black) are maximal in the set of terminals. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

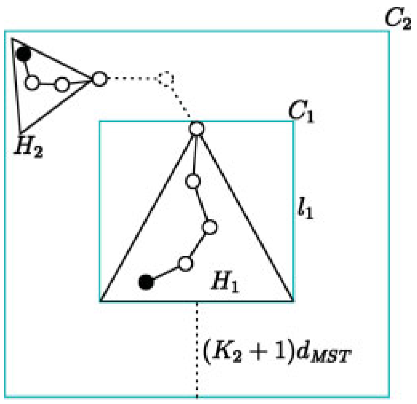


FIG. 12. If K_2-HFST H_2 is mergable with K_1-HFST H_1 then H_2 is fully contained in hypercube C_2. [Color figure can be viewed in the online issue, which is available at www.interscience.wiley.com.]

To do this, we need the following definitions. Let $z, z' \in Z$ be two given terminals. Then z is said to be *dominated* by z' if $z[c] < z'[c]$ for all $c = 1, \ldots, d$. If $Z' \subseteq Z$, a terminal is said to be *maximal* in Z' if it belongs to Z' and is not dominated by any terminal in Z'; see Figure 11.

Maximal terminals and empty hyperboxes are related in the following way. Let z be a terminal, let $Z_{\text{dom}} \subset Z$ be the set of terminals dominated by z, and let z' be any terminal in Z_{dom}.

Lemma 6. *With the above definitions, z' is maximal in Z_{dom} iff $B(z, z')$ is empty.*

Proof. Suppose the hyperbox $B(z, z')$ spanned by z and z' is not empty and let $z'' \neq z, z'$ be a terminal in $B(z, z')^\circ$. As $z' \in Z_{\text{dom}}$, we have $z'[c] < z[c]$ for all axes c. Hence, $z'' \in Z_{\text{dom}}$ and $z''[c] > z'[c]$ on all axes c, showing that z' is not maximal in Z_{dom}.

Now, suppose z' is not maximal in Z_{dom}. Then there is a terminal z'' in Z_{dom} such that $z[c] > z''[c] > z'[c]$ on all axes c. It follows that $z'' \in B(z, z')^\circ$. ■

By making suitable transformations, we see from Lemma 6 that the problem of finding empty hyperboxes is equivalent to finding maximal terminals. From this it follows that an upper bound on the expected number of maximal terminals is also an upper bound on the expected number of terminals that span empty hyperboxes with z.

Buchta [1] showed that, under our assumption that terminals are randomly distributed in $\mathcal{U}$ with a uniform distribution, the expected number of maximal terminals in Z is

$$\frac{1}{(d-1)!}\log^{d-1} n + \frac{\gamma}{(d-2)!}\log^{d-2} n + O(\log^{d-3} n) = O(\log^{d-1} n),$$

where $\gamma = 0.577\ldots$ is Euler's constant.

To bound the expected number of interesting 2-HFSTs, let C_z be the hypercube with center at some terminal z and having sidelength $4d_{\text{MST}}$. Recall that the expected number of terminals in C_z is $O(\log n)$ and note that these terminals are uniformly distributed in C_z.

Thus, by combining the weak bottleneck property and the empty hyperbox property, the above shows that the expected number of 2-HFSTs containing z is $O((\log\log n)^{d-1})$. The total expected number of interesting 2-HFSTs is therefore $O(n(\log\log n)^{d-1})$.

4.8. Bounding the Number of K-HFSTs

To bound the expected number of interesting K-FSTs for $K \geq 3$, we will start by bounding the expected number of interesting K-HFSTs for $K \geq 2$. We already have a bound for $K = 2$. The corollary to the following lemma bounds the expected number of interesting K-HFSTs for all other values of K as well.

Lemma 7. *Let $K_1, K_2 \geq 1$ be given, let $K = K_1 + K_2$, and assume that $3 \leq K \leq n$. The expected number of interesting K-HFSTs obtained by merging interesting K_1-HFSTs with interesting K_2-HFSTs is $O(m_{K_1} m_{K_2} \log n/n)$, where m_{K_1} resp. m_{K_2} is the expected number of interesting K_1-HFSTs resp. K_2-HFSTs.*

Proof. Let H_1 be any interesting K_1-HFST. We will prove the lemma by showing that the expected number of interesting K_2-HFSTs mergable with H_1 is $O(m_{K_2} \log n/n)$.

Let $C_1 \subseteq \mathcal{U}$ be the smallest axis-aligned hypercube containing H_1; see Figure 12. As H_1 contains $O(K_1)$ edges, the expected sidelength of C_1 is

$$O(K_1 d_{MST}) = O(\sqrt[d]{\log n/n}).$$

Letting l_1 denote the sidelength of C_1, define C_2 to be the hypercube having the same center as C_1 and having sidelength

$$l_2 = l_1 + 2(K_2 + 1)d_{\text{MST}}.$$

Let H_2 be an interesting K_2-HFST mergable with H_1. Any simple path in H_2 ending at the root of H_2 contains at most $K_2 - 1$ edges. The distance from the root of H_1 to the root of H_2 is at most $2d_{\text{MST}}$. Hence, H_2 is fully contained in C_2.

If we translate C_2 inside $\mathcal{U}$, the expected number of interesting K_2-HFSTs fully contained in C_2 is unchanged. As the volume of C_2 is $O(\log n/n)$, the expected number of interesting K_2-HFSTs in C_2 is $O(m_{K_2} \log n/n)$ if C_2 is contained in $\mathcal{U}$. Clearly, this also holds when C_2 is not contained in $\mathcal{U}$. ▪

Corollary 2. *For any $K \geq 2$, the expected number of interesting K-HFSTs is $O(n(\log\log n)^{d-1} \log^{K-2} n)$.*

Proof. The proof is by induction on K. If $K = 2$, the corollary follows from the above. Now assume that $K > 2$ and that the corollary holds for all values less than K.

To show the induction step, we will apply Lemma 7. Pick any $K_1, K_2 \geq 1$ such that $K_1 + K_2 = K$. Assume w.l.o.g. that $K_1 \leq K_2$.

If $K_1 > 1$, the induction hypothesis implies that the expected number of interesting K-HFSTs obtained by merging interesting K_1-HFSTs with interesting K_2-HFSTs is

$$O(n(\log\log n)^{2(d-1)} \log^{K_1+K_2-3} n) = O(n(\log\log n)^{2(d-1)} \log^{K-3} n).$$

If $K_1 = 1$, this expected number is

$$O(n(\log\log n)^{d-1} \log^{K_2-1} n) = O(n(\log\log n)^{d-1} \log^{K-2} n),$$

as the number of interesting 1-HFSTs is n. It follows that the total expected number of of interesting K-HFSTs is

$$O(n(\log\log n)^{d-1} \log^{K-2} n),$$

as requested. ▪

4.9. Bounding the Number of FSTs

The above results allow us to prove the main result of this section.

Theorem 2. *For any dimension $d \geq 2$, the expected number of interesting 3-FSTs is $O(n(\log\log n)^{2(d-1)})$, and if $K > 3$, the expected number of interesting K-FSTs is $O(n(\log\log n)^{d-1} \log^{K-2} n)$.*

Proof. The bound on the expected number of interesting 3-FSTs follows from the ideas at the end of Section 4.7. The bound for $K > 3$ follows from Corollary 2 and from the observation that any K-FST can be regarded as a K-HFST by adding a degree two Steiner point to an edge of the K-FST and rooting the tree at this Steiner point. ▪

5. CONCLUDING REMARKS

We bounded the expected number of FSTs satisfying the empty lune property, a weak version of the bottleneck property, and the empty hyperbox property in the space $(\mathbb{R}^d, L_1)$, $d \geq 2$, when terminals are randomly distributed in a hypercube with a uniform distribution. We showed that the expected number of such FSTs spanning exactly $K \geq 3$ terminals is $O(n(\log\log n)^{2(d-1)})$ for $K = 3$ and $O(n(\log\log n)^{d-1} \log^{K-2} n)$ for $K > 3$.

In the plane, we showed that the expected number of K-FSTs with the Hwang form satisfying the empty lune property and the disjoint lunes property is $O(n\pi^K)$ for any $K \geq 4$. This improves the bound of [8].

In two dimensions, experimental results suggest that the expected number of FSTs with the Hwang form satisfying the empty lune property and the disjoint lunes property grows exponentially in n. This makes it unlikely that our bound of Section 3 can be improved significantly without considering other properties of FSTs. It appears (again from experimental results) that making use of the bottleneck property in addition to the empty lune property and the Hwang form is sufficient to obtain a linear expected bound.

Experiments in dimensions higher than two suggest that the empty lune property, the bottleneck property, and the empty hyperbox property are not sufficient to obtain even a polynomial bound on the expected number of FSTs.

REFERENCES

[1] C. Buchta, On the average number of maxima in a set of vectors, Info Process Lett 33 (1989), 63–65.

[2] U. Fößmeier, M. Kaufmann, and A. Zelikovsky, Faster approximation algorithms for the rectilinear Steiner tree problem, Discrete Comput Geom 18 (1997), 93–109.

[3] M.R. Garey and D.S. Johnson, The rectilinear Steiner tree problem is NP-complete, SIAM J Appl Math 32 (1977), 826–834.

[4] F.K. Hwang, On Steiner minimal trees with rectilinear distance, SIAM J Appl Math 30 (1976), 104–114.

[5] J.S. Salowe and D.M. Warme, Thirty-five point rectilinear Steiner minimal trees in a day, Networks 25 (1995), 69–87.

[6] D.M. Warme, Spanning trees in hypergraphs with applications to Steiner trees, Ph.D. Thesis, Computer Science Dept., The University of Virginia, 1998.

[7] D.M. Warme, P. Winter, and M. Zachariasen, GeoSteiner 3.1, Department of Computer Science, University of Copenhagen. Available at: http://diku.dk/hjemmesider/ansatte/martinz/geosteiner/, 2001. Accessed on July 22, 2009.

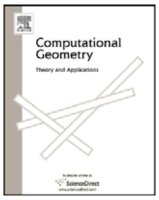
[8] M. Zachariasen, Rectilinear full Steiner tree generation, Networks 33 (1999), 125–143.

Computational Geometry 43 (2010) 68–72

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo



Computing the dilation of edge-augmented graphs in metric spaces

Christian Wulff-Nilsen

Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

A R T I C L E   I N F O

Article history:
Received 14 April 2008
Received in revised form 10 March 2009
Accepted 10 March 2009
Available online 23 June 2009
Communicated by S. Lazard

Keywords:
Dilation
Stretch factor
Best shortcut
Edge augmentation
Metric space

A B S T R A C T

Let $G = (V, E)$ be an undirected graph with n vertices embedded in a metric space. We consider the problem of adding a shortcut edge in G that minimizes the dilation of the resulting graph. The fastest algorithm to date for this problem has $O(n^4)$ running time and uses $O(n^2)$ space. We show how to improve the running time to $O(n^3 \log n)$ while maintaining quadratic space requirement. In fact, our algorithm not only determines the best shortcut but computes the dilation of $G \cup \{(u, v)\}$ for every pair of distinct vertices u and v.

1. Introduction

In areas such as VLSI design, telecommunication, and distributed systems, a problem often arising is that of interconnecting a set of sites in a network of small cost. There are many different ways of measuring the cost of a network, such as its size, total length, minimum and maximum degree, diameter, and dilation (also known as stretch factor).

Spanners are sparse or economic representations of networks, making them important geometric structures in the areas mentioned above. They have received a great deal of attention in recent years, see e.g. [7,2,8].

A *t-spanner* is a graph embedded in a metric space such that, for any pair of vertices in this graph, the graph distance between them is at most t times their metric distance. The smallest t such that a graph is a t-spanner is called the *dilation* of the graph.

Most algorithms construct networks from scratch, but frequently one is interested in extending an already given network with a number of edges such that the dilation of the resulting network is minimized.

Farshi et al. [3] considered the following problem: given a graph $G = (V, E)$ with n vertices embedded in a metric space, find a vertex pair $(u, v) \in V \times V$ (called a shortcut) such that the dilation of $G \cup \{(u, v)\}$ is minimized. They gave a trivial $O(n^4)$ time and $O(n^2)$ space algorithm for this problem together with various approximation algorithms.

In this paper, we present an $O(n^3 \log n)$ time and $O(n^2)$ space algorithm for the above problem. This algorithm not only computes the best shortcut but returns a table T with a row and a column for every vertex in G such that for any pair of distinct vertices u and v, $T(u, v)$ is the dilation of $G \cup \{(u, v)\}$.

The organization of the paper is as follows. In Section 2, we give various basic definitions and assumptions. In Section 3, we present one of the key ideas of the paper which gives a more efficient way of obtaining the dilation of edge-augmented graphs. We present our algorithm and prove its correctness in Sections 4 and 5, we show that the above time and space bounds hold. In Section 6, we consider the case where the given graph is disconnected. Finally, we make some concluding remarks and pose open problems in Section 7.

E-mail address: koolooz@diku.dk.

0925-7721/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.comgeo.2009.03.008

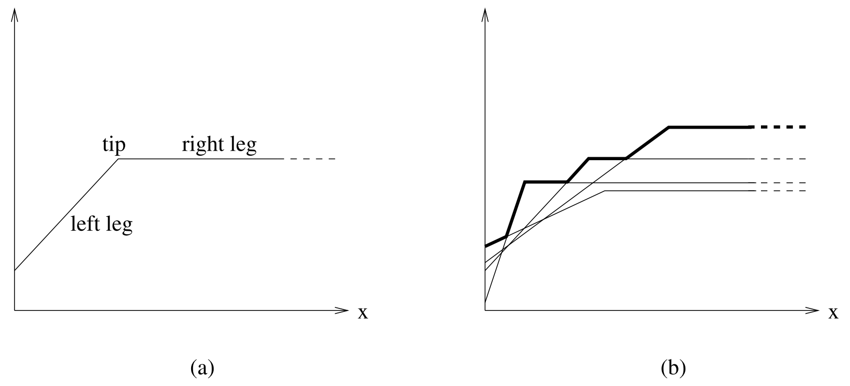


Fig. 1. (a) A staircase step and (b) the upper envelope (thick line segments) of a set of four staircase steps.

2. Basic definitions and assumptions

Let $G = (V, E)$ be an undirected graph embedded in metric space (V, d) and assume that G is connected.

Given two vertices $u, v \in V$, a *shortest path* between u and v is a path in G between u and v for which the sum of lengths of the edges of the path is minimal. We denote by $d_G(u, v)$ the length of such a path.

We define the *dilation* $\delta_G(u, v)$ of a pair of distinct vertices $u, v \in V$ as $d_G(u, v)/d(u, v)$. The dilation of G is defined as

$$\delta_G = \max_{u,v \in V, u \neq v} \delta_G(u, v).$$

In the following, G denotes an undirected, connected graph (V, E) embedded in metric space (V, d) and we define $n = |V|$.

3. Upper envelope of functions

In this section, we consider the upper envelope of certain functions which will help us to compute the dilation of graphs obtained from G by the addition of a shortcut (a single edge).

Let u, v, and w_1 be three fixed vertices of G such that $u \neq v$ and let w_2 be a fourth vertex of G (not fixed).

Let $G' = G \cup \{e\}$ be the graph obtained by adding shortcut $e = (w_1, w_2)$ to G. Suppose that $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$. Then no shortest path in G' from u traverses e in the direction $w_1 \to w_2$. Let $x = d_G(u, w_2) + d(w_2, w_1)$. Since we have not fixed w_2, we can regard x as a non-negative variable depending on w_2.

Let us analyze how $\delta_{G'}(u, v)$ changes as a function of x. For $x = 0$, if $x + d_G(w_1, v) > d_G(u, v)$ then a shortest path between u and v in G' cannot traverse e in direction $w_2 \to w_1$. By the above, a shortest path between u and v in G is also a shortest path between u and v in G'. This will also hold when x increases so $\delta_{G'}(u, v)$ is just a constant function of x.

Now, suppose $x + d_G(w_1, v) \leqslant d_G(u, v)$ for $x = 0$. Then there will be a shortest path from u to v in G' with a subpath traversing e in direction $w_2 \to w_1$ (by the inequality $d_G(u, w_2) < d_G(u, w_1) + d(w_1, w_2)$ above). As x increases it gets more and more expensive to use this subpath and thus $\delta_{G'}(u, v)$ increases; the increase in $\delta_{G'}(u, v)$ is a linear function of the increase in x. Eventually, $x + d_G(w_1, v) \geqslant d_G(u, v)$ and it will be cheaper to use a shortest path from u to v in the original graph G so $\delta_{G'}(u, v)$ will no longer increase but stay constant (as a function of x).

To be more precise, define constants $a = 1/d(u, v)$, $b = d_G(w_1, v)/d(u, v)$, and $c = \delta_G(u, v)$. Then $x \geqslant (c - b)/a$ is equivalent to the inequality $x + d_G(w_1, v) \geqslant d_G(u, v)$ above, giving

$$\delta_{G'}(u, v) = \min\{c, ax + b\} = \begin{cases} c & \text{if } x \geqslant \frac{c-b}{a}, \\ ax + b & \text{if } x \leqslant \frac{c-b}{a}. \end{cases}$$

Hence, the dilation between u and v in G' may be expressed as a piecewise linear function $\delta(x)$ of the length $x \geqslant 0$ of a shortest path among those paths in G' from u to w_1 having e as their last edge.

We refer to the graph of $\delta(x)$ as a *staircase step*, see Fig. 1(a). Assuming $(c - b)/a > 0$, the part of the graph on interval $[0, (c - b)/a]$ is a line segment with slope a, which we refer to as the *left leg* of $\delta(x)$.

If $(c - b)/a \leqslant 0$, we define the left leg of $\delta(x)$ to be the degenerate line segment with slope a starting and ending in the point $(0, \delta(0))$.

The part of the graph on interval $[\max\{0, (c - b)/a\}, \infty)$ is a horizontal halfline, called the *right leg* of $\delta(x)$.

We define the *slope* of $\delta(x)$ to be the slope a of its left leg.

The left and right leg of $\delta(x)$ meet in a single point. We refer to this point as the *tip* of $\delta(x)$.

Now, suppose we fix only u and w_1. For each $v \in V \setminus \{u\}$, we obtain a staircase step expressing the dilation between u and v in G'. We define $s_{(u,w_1)}$ to be the *staircase function* representing the upper envelope of the union of all these staircase steps as a function of $x \geqslant 0$, see Fig. 1(b). Note that this function is piecewise linear and non-decreasing.

4. The algorithm and its correctness

In this section, we present our algorithm and prove its correctness.

Initially, $d_G(u,v)$ is computed and stored for each $(u,v) \in V \times V$, and a table T with an entry for each ordered pair of vertices of G is initialized.

Let (w_1,w_2) be a pair of distinct vertices of V. At termination, entry $T(w_1,w_2)$ will hold the maximum dilation in $G \cup \{(w_1,w_2)\}$ over a certain set of vertex pairs. Similarly, entry $T(w_2,w_1)$ will hold the maximum dilation in $G \cup \{(w_1,w_2)\}$ over another set of vertex pairs. As we shall see, the union of these two sets cover all pairs of distinct vertices, implying that

$$\max\{T(w_1,w_2), T(w_2,w_1)\} = \delta_{G\cup\{(w_1,w_2)\}}. \tag{1}$$

A subsequent step may update T in $\Theta(n^2)$ time such that $T(w_1,w_2) = \delta_{G\cup\{(w_1,w_2)\}}$ for all $w_1 \neq w_2$. A best shortcut is then a pair (w_1,w_2) minimizing $T(w_1,w_2)$.

The algorithm consists of a loop which iterates over all vertices of G. Let w_1 be the vertex in the current iteration. First, staircase functions $s_{(u,w_1)}$ are computed for each $u \in V$. Then for each vertex $w_2 \neq w_1$, entry (w_1,w_2) in T is set to

$$T(w_1,w_2) = \max\{s_{(u,w_1)}(x) \mid u \in V,\ d_G(u,w_2) < d_G(u,w_1) + d(w_1,w_2)\},$$

where $x = d_G(u,w_2) + d(w_2,w_1)$. This is well-defined since $u = w_2$ satisfies $d_G(u,w_2) < d_G(u,w_1) + d(w_1,w_2)$. Note that x and the inequality in the definition of $T(w_1,w_2)$ are the same as in the previous section.

The following theorem shows the correctness of our algorithm.

Theorem 1. *When the above algorithm terminates,* (1) *holds for each pair* (w_1,w_2) *of distinct vertices.*

Proof. Let (w_1,w_2) be any pair of distinct vertices of G and let $G' = G \cup \{(w_1,w_2)\}$. For any $u \in V$ for which $d_G(u,w_2) < d_G(u,w_1) + d(w_1,w_2)$ holds,

$$s_{(u,w_1)}\big(d_G(u,w_2) + d(w_2,w_1)\big) = \max_{v \in V\setminus\{u\}} \delta_{G'}(u,v).$$

Similarly, for any $u \in V$ for which $d_G(u,w_1) < d_G(u,w_2) + d(w_2,w_1)$ holds,

$$s_{(u,w_2)}\big(d_G(u,w_1) + d(w_1,w_2)\big) = \max_{v \in V\setminus\{u\}} \delta_{G'}(u,v).$$

Furthermore, for any $u \in V$, either $d_G(u,w_2) < d_G(u,w_1) + d(w_1,w_2)$ or $d_G(u,w_1) < d_G(u,w_2) + d(w_2,w_1)$ for otherwise we get the contradiction

$$\begin{aligned} d_G(u,w_2) + d(w_2,w_1) &\leqslant d_G(u,w_1) \\ &\leqslant d_G(u,w_2) - d(w_1,w_2) \\ &< d_G(u,w_2) + d(w_2,w_1), \end{aligned}$$

where the strict inequality follows from the assumption that $w_1 \neq w_2$. Thus, at termination,

$$\max\{T(w_1,w_2), T(w_2,w_1)\} = \max_{u,v \in V,\, u \neq v} \delta_{G'}(u,v) = \delta_{G'},$$

as requested. □

5. Running time and space requirement

In this section, we show that the algorithm of the previous section has $O(n^3 \log n)$ running time and $O(n^2)$ space requirement. We will need the following lemma.

Lemma 1. *Given vertices u and w_1, the graph of staircase function $s_{(u,w_1)}$ consists of $O(n)$ line segments and one halfline and can be computed in $O(n \log n)$ time when $d_G(w_1,v)$ and $d_G(u,v)$ are precomputed for all $v \in V$. Furthermore, when this graph is given, $s_{(u,w_1)}(x)$ can be computed in $O(\log n)$ time for any $x \geqslant 0$.*

Proof. Note that when $d_G(w_1,v)$ and $d_G(u,v)$ are precomputed for all vertices v, each staircase step may be computed in constant time.

We represent the graph of $s_{(u,w_1)}$ as a polygonal chain P. To construct P, we start by computing each staircase step and the tip with maximum x-coordinate, say $x_{\max}$. The upper envelope of P to the right of $x_{\max}$ is the upper envelope of $O(n)$ horizontal halflines and may be computed in $O(n)$ time. The upper envelope of P on interval $[0, x_{\max}]$ is the upper envelope of $O(n)$ line segments. We use the algorithm of Hershberger [5] to compute this upper envelope in $O(n \log n)$ time. It follows that P may be constructed in $O(n \log n)$ time.

Clearly, P consists of line segments and exactly one halfline. We need to show that the number of line segments is $O(n)$.

Consider constructing P by iteratively adding staircase steps in non-decreasing order of slope. Let P_i be the upper envelope of the first i staircase steps.

Upper envelope P_1 consists of exactly one line segment (and one halfline). For $i > 1$, the left leg of the ith staircase step s_i intersects P_{i-1} at most once due to the order of staircase steps. Since the right leg is horizontal, it cannot intersect P_{i-1} more than once. Hence, P_i has at most two more line segments than P_{i-1}.

One fine point: a degeneracy may occur if the left leg of s_i overlaps with a line segment of P_{i-1}. It is easy to see that in this case, P_i cannot contain more line segments than P_{i-1}, again due to the order of the staircase steps.

Since there are $O(n)$ staircase steps, the above shows that P consists of $O(n)$ line segments.

Since $s_{(u,w_1)}$ is a non-decreasing function of x, we may apply a binary search in P to compute $s_{(u,w_1)}(x)$ for any $x \geqslant 0$. Since P consists of $O(n)$ line segments, this takes $O(\log n)$ time. $\square$

We are now ready for the main result of this section.

Theorem 2. *The algorithm described in Section 4 has $O(n^3 \log n)$ running time and $O(n^2)$ space requirement.*

Proof. To prove the time bound, first observe that computing all-pairs shortest paths takes $O(n^3)$ time with the Floyd–Warshall algorithm [4] (faster algorithms exist, see e.g. [1], but they will not improve the asymptotic running time of our algorithm).

Furthermore, the graph of each staircase function is computed exactly once throughout the course of the algorithm. Hence, by Lemma 1, the total time spent on computing these functions is $O(n^3 \log n)$. Once the staircase functions have been computed, computing an entry of T takes $O(n \log n)$ time by Lemma 1. Since T has n^2 entries, computing T takes $O(n^3 \log n)$ time. When all entries in T have been computed, finding the best shortcut takes $O(n^2)$ time. Hence, the total running time of the algorithm is $O(n^3 \log n)$.

Space requirement is bounded by that of the Floyd–Warshall algorithm and the space for storing the staircase functions, the shortest path lengths, and the table T. The Floyd–Warshall algorithm requires $\Theta(n^2)$ space. Clearly, T and the shortest path lengths can be stored using a total of $\Theta(n^2)$ space. In each iteration of the algorithm, we only store n staircase functions. By Lemma 1, they take up a total of $O(n^2)$ space. $\square$

6. Disconnected graph

Recall our assumption that G is connected. In this section, we show that some simple modifications of our algorithm allow us to handle the case where G is disconnected without affecting the worst-case running time and space requirement of the algorithm.

Note that if G consists of more than two connected components, G has infinite dilation and no single edge can be added to G to reduce the dilation, making the problem we consider trivial. Since there are efficient algorithms for determining the connected components of a graph, we may therefore restrict our attention to the case where G consists of exactly two connected components and assume that these two components have been computed.

So let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be the subgraphs defining the two connected components of G. For all shortcuts $(w_1, w_2) \in V_1 \times V_1 \cup V_2 \times V_2$, we set $T(w_1, w_2) = \infty$ since they leave the graph disconnected and hence leave the dilation of the graph unchanged.

As for the other entries in T, consider a pair of vertices (w_1, w_2) in $V_1 \times V_2$ and let $G' = G \cup \{(w_1, w_2)\}$. Let $u \neq v$ be two vertices of V. If $u, v \in V_1$ or $u, v \in V_2$ then clearly $\delta_{G'}(u, v) = \delta_G(u, v)$.

Now assume that $v \in V_1$ and $u \in V_2$. Then

$$\delta_{G'}(u, v) = ax + b,$$

where $x = d_G(u, w_2) + d(w_2, w_1)$, $a = 1/d(u, v)$, and $b = d_G(w_1, v)/d(u, v)$. Comparing this with the results of Section 3, we see that we in effect obtain staircase steps with no right leg. We let $s_{(u,w_1)}$ denote the staircase function representing the upper envelope of the staircase steps obtained from each $v \in V_1$ as a function of x.

To determine all entries $T(w_1, w_2)$ of T where $(w_1, w_2) \in V_1 \times V_2$, we make the following small changes to the algorithm of Section 4. The loop only iterates over vertices $w_1 \in V_1$. Furthermore, we only compute staircase functions $s_{(u,w_1)}$ for $u \in V_2$ and we set

$$T(w_1, w_2) = \max\bigl\{\delta_{G_1}, \delta_{G_2}, \max\bigl\{s_{(u,w_1)}(x) \bigm| u \in V_2\bigr\}\bigr\}$$

for each $w_2 \in V_2$ with x defined as above.

By the above it follows that, at termination, the modified algorithm satisfies

$$\max\bigl\{T(w_1, w_2), T(w_2, w_1)\bigr\} = \delta_{G \cup \{(w_1, w_2)\}}$$

for all distinct pairs of vertices w_1 and w_2 in G.

Computing the graph of staircase function $s_{(u,w_1)}$ is done in $O(n \log n)$ time using the algorithm of Hershberger [5] (as in the proof of Lemma 1, we pick a maximum x-value in order to consider line segments instead of halflines. Pick, say, the largest x-value ever needed by the algorithm). The graph of $s_{(u,w_1)}$ has complexity $O(n)$ and when it is given, $s_{(u,w_1)}(x)$ can be computed in $O(\log n)$ time for any $x \geqslant 0$; the proof of these claims is similar to the proof of Lemma 1.

From the above and from the results of Section 5, it follows easily that all entries of T can be computed in $O(n^3 \log n)$ time using $O(n^2)$ space when G is disconnected.

7. Concluding remarks

We presented an $O(n^3 \log n)$ time and $O(n^2)$ space algorithm for the problem of computing the best shortcut of a graph $G = (V, E)$ with n vertices embedded in a metric space. This improves upon a previous bound of $O(n^4)$ time and $O(n^2)$ space [3]. Our algorithm in fact solves a harder problem, namely that of computing the dilation of $G \cup \{(u, v)\}$ for each pair of distinct vertices u and v.

Based on ideas of this paper, the open problem stated in [3] of whether there exists a linear space algorithm with $o(n^4)$ running time for finding the best shortcut is solved in [6]. We pose the following problems. Is our algorithm optimal in terms of running time? Is it possible to extend our results to the more general case of adding a constant number of edges to G?

Acknowledgements

I thank Pawel Winter, Martin Zachariasen, and the anonymous referees for their comments and remarks.

References

[1] T.M. Chan, More algorithms for all-pairs shortest paths in weighted graphs, in: Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing, 2007, pp. 590–598.
[2] D. Eppstein, Spanning trees and spanners, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier Science Publishers, Amsterdam, 2000, pp. 425–461.
[3] M. Farshi, P. Giannopoulos, J. Gudmundsson, Improving the stretch factor of a geometric network by edge augmentation, SIAM J. Comput. 38 (1) (2008) 226–240.
[4] R.W. Floyd, Algorithm 97 (Shortest Path), Comm. ACM 5 (6) (1962) 345.
[5] J. Hershberger, Finding the upper envelope of n line segments in $O(n \log n)$ time, Inform. Process. Lett. 33 (4) (1989) 169–174.
[6] J. Luo, C. Wulff-Nilsen, Computing best and worst shortcuts of graphs embedded in metric spaces, in: S.-H. Hong, H. Nagamochi, T. Fukunaga (Eds.), Proceedings of the 19th International Symposium on Algorithms and Computation, in: Lecture Notes in Computer Sience, vol. 5369, Springer-Verlag, Berlin, 2008, pp. 764–775.
[7] G. Narasimhan, M. Smid, Geometric Spanner Networks, Cambridge University Press, 2007.
[8] M. Smid, Closest point problems in computational geometry, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier Science Publishers, Amsterdam, 2000, pp. 877–935.

Computing the Maximum Detour of a Plane Graph in Subquadratic Time

Christian Wulff-Nilsen

Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen O, Denmark
koolooz@diku.dk
http://www.diku.dk/hjemmesider/ansatte/koolooz/

Abstract. Let G be a plane graph where each edge is a line segment. We consider the problem of computing the maximum detour of G, defined as the maximum over all pairs of distinct points p and q of G of the ratio between the distance between p and q in G and the distance $|pq|$. The fastest known algorithm for this problem has $\Theta(n^2)$ running time where n is the number of vertices. We show how to obtain $O(n^{3/2}\log^3 n)$ expected running time. We also show that if G has bounded treewidth, its maximum detour can be computed in $O(n\log^3 n)$ expected time.

1 Introduction

Given a geometric graph G, its stretch factor (or dilation) is the maximum over all pairs of distinct vertices u and v of the ratio between the distance between u and v in G and the Euclidean distance $|uv|$ between u and v.

A spanner is a network with small stretch factor. Spanners that keep other cost measures low, such as size, weight, degree, and diameter, are important structures in areas such as VLSI design, distributed computing, and robotics. For more on spanners, see [5,9,10].

An interesting dual problem is that of computing the stretch factor of a given geometric graph. In this paper, we consider a related problem, namely that of computing the maximum detour of a plane graph where edges are line segments. Maximum detour is defined like stretch factor except that the maximum is taken over all pairs of distinct *points* of the graph, i.e., interior points of edges as well as the vertices.

If the graph is planar then its stretch factor can be computed in $\Theta(n^2)$ time, where n is the number of vertices, by applying the APSP algorithm in [6]. This bound also holds for the problem of computing the maximum detour of a plane graph [1]. It is an open problem whether subquadratic time algorithms exist.

For more special types of graphs such as paths, trees, cycles, and graphs having bounded treewidth, faster algorithms are known for the stretch factor and maximum detour problem [1,2].

In this paper, we show how to compute the maximum detour of a plane graph with n vertices in $O(n^{3/2}\log^3 n)$ expected time, thereby solving the open problem of whether a subquadratic time algorithm exists for this problem. We also show that if the graph has bounded treewidth, its maximum detour can be computed in $O(n \log^3 n)$ expected time.

The organization of the paper is as follows. In Section 2, we give various definitions and introduce some notation. In Section 3, we make use of the separator theorem by Lipton and Tarjan which enables us to apply the divide-and-conquer paradigm to the input graph. We define colourings of points of a face of the graph in Section 4, show some properties of these colourings and how to efficiently compute them. In Section 5, we show how the colourings give an efficient way of computing the maximum detour between points on a face of the graph and this in turn gives an efficient algorithm for computing the maximum detour of the entire graph. Finally, we make some concluding remarks in Section 6.

2 Definitions and Notation

Let $G = (V, E)$ be a plane graph where each edge is a line segment and let P_G be the set of points of G (vertices as well as interior points of edges). Given two points $p, q \in P_G$, we define $d_G(p, q)$ as the length of a shortest path in G between p and q, where the length of a path is measured as the sum of the Euclidean lengths of the (parts of) edges on this path. If there is no such path, we define $d_G(p, q) = \infty$. If $p \neq q$ then the *detour* $\delta_G(p, q)$ between p and q (in G) is defined as the ratio $d_G(p, q)/|pq|$. The *maximum detour* δ_G of G is the maximum of this ratio over all pairs of distinct points of P_G.

Where appropriate, we will regard a plane graph as the set of points belonging to the graph. So for instance, if G and H are plane graphs then $G \cap H$ is the set of points belonging to both G and H. If well-defined, we will regard the resulting point set as a graph.

For a graph G, we let $|G|$ denote its size, i.e. the number of vertices plus the number of edges in G. Given two subsets P_1 and P_2 of the set P_G of points of G, we define

$$\delta_G(P_1, P_2) = \max_{p \in P_1, q \in P_2, p \neq q} \delta_G(p, q).$$

If p is a point of G and $P \subseteq P_G$, we write $\delta_G(p, P) = \delta_G(P, p)$ instead of $\delta_G(\{p\}, P)$ and we write $\delta_G(P)$ as a shorthand for $\delta_G(P, P)$. We extend these definitions to subgraphs, edges, and vertices by regarding them as sets of points.

Given paths $P = p_1 \to p_2 \to \ldots \to p_r$ and $Q = q_1 \to q_2 \to \ldots \to q_s$, where $p_r = q_1$, we let PQ denote the combined path $p_1 \to p_2 \to \ldots \to p_{r-1} \to q_1 \to q_2 \to \ldots \to q_s$. For a vertex v, we let $v \to P$ denote the path $v \to p_1 \to p_2 \to \ldots \to p_r$.

3 Separating the Problem

In all the following, let $G = (V, E)$ be an n-vertex plane graph in which edges are line segments. We seek to compute δ_G in $O(n^{3/2} \log^3 n)$ expected time. We will assume that G is connected since otherwise, the problem is trivial.

To compute δ_G, we apply the divide-and-conquer paradigm. Our strategy is in some ways similar to that in [2], the main difference being that in the merge step we use face colourings (Section 4) whereas range searching is used in [2].

In this section, we separate G into two smaller graphs, G_A and G_B, of roughly the same size. We recursively compute $\delta_G(G_A)$ and $\delta_G(G_B)$ and in Section 5, we describe an algorithm for efficiently obtaining the maximum detour $\delta_G(G_A, G_B)$.

To separate our problem, we use the separator theorem of Lipton and Tarjan [8]. This gives us, in $O(n)$ time, a partition of V into three sets, A, B, and P, such that the following three properties hold

1. no edge joins a vertex in A with a vertex in B,
2. neither A nor B contains more than $n/2$ vertices, and
3. P contains no more than $\frac{2\sqrt{2}}{1-\sqrt{2/3}}\sqrt{n}$ vertices.

We refer to vertices of P as *portals*. We must have $P \neq \emptyset$ since otherwise, one of the sets A and B would be empty by property 1, implying that $|A| = n$ or $|B| = n$, contradicting property 2. In the following, let $k \geq 1$ denote the number of portals and let $p_1, \ldots, p_k$ denote the portals.

Having found this partition, we compute and store shortest path lengths from each portal to each vertex of V. Using Dijkstra's SSSP algorithm with, say, binary heaps, this can be done in $O(n \log n)$ time for each portal, giving a total running time of $O(n^{3/2} \log n)$.

Let G_A be the subgraph of G induced by $A \cup P$ and let G_B be the subgraph of G induced by $B \cup P$. We construct G_A and G_B and recursively compute $\delta_G(G_A)$ and $\delta_G(G_B)$. Clearly,

$$\delta_G = \max\{\delta_G(G_A), \delta_G(G_B), \delta_G(G_A, G_B)\}.$$

In the following, we deal with the problem of computing $\delta_G(G_A, G_B)$.

We will need the following lemma which is a generalization of a result in [4] (we omit the proof since it is virtually identical to that in [4]).

Lemma 1. *Maximum detour δ_G is achieved by a pair of co-visible points.*

This result allows us to consider only detours between pairs of points of the same face of G (here we include the external face). In all the following, let f be a face of G, let $f_A = f \cap G_A$, and let $f_B = f \cap G_B$. We will show how to compute $\delta_G(f_A, f_B)$ in $O(|f| k \log^2 n)$ expected time. From this it will follow that $\delta_G(G_A, G_B)$ can be found in $O(n^{3/2} \log^2 n)$ expected time.

We will assume that f is an internal face of G. The external face is dealt with in a similar way. We also assume that f_A and f_B do not share any edges since any edge e shared by them must have portals as both endpoints and the detours from points in e to all points in G will be considered in the two recursive calls that compute $\delta_G(G_A)$ and $\delta_G(G_B)$, respectively. Hence, we may disregard e when computing $\delta_G(f_A, f_B)$.

We assume that f is simple. The case where f is non-simple is handled in a similar way.

4 Colouring Points

In this section, we define colourings of points of f. As we shall see in Section 5, these colourings will prove helpful when computing $\delta_G(f_A, f_B)$. More specifically, they will speed up shortest path computations between pairs of points in f by indicating, for each point pair, which portal is on some shortest path between those two points.

Face f is defined by a simple cycle $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{n_f} \rightarrow v_1$ such that the interior of f is to the left as we walk in the direction specified by the vertices.

For a point $p \in f$, we let $d_f(p)$ denote the Euclidean length of the path from v_1 to p that visits vertices in the order specified above. We define an order $<^f_{v_1}$ of the set of points of f as follows. For two points p and q of f, $p <^f_{v_1} q$ if and only if $d_f(p) < d_f(q)$. Order $p >^f_{v_1} q$ is defined in a similar way. In the following, we assume that the sum of lengths of all edges in f and $d_f(v_i)$ for all $v_i \in f$ are precomputed.

By starting the walk in any other vertex v_i of f, we can similarly define orders $<^f_{v_i}$ and $>^f_{v_i}$. It is easy to see that determining whether e.g. $p <^f_{v_i} q$ holds for any points $p, q \in f$ can be done in constant time using the above precomputed values. Where appropriate, we will regard the points of f (or f_A or f_B) occuring between two points w.r.t. order $<^f_{v_i}$ as an interval of points.

In the following, let a be a vertex in f_A and let $a' \in f \cap V$ be its successor w.r.t. $<^f_a$. We now consider associating with a a colouring of points in f_B using colours $c_1, \ldots, c_k$. A point $p \in f_B$ is given colour c_i if portal p_i is on a shortest path in G from a to p. In case of ties, pick the colour such that the corresponding portal has minimum distance to a in G. In case of further ties, pick the colour with the smaller index. We let $c_a(p)$ denote the colour assigned to p.

We will show that colours occur in intervals as we walk around f_B with each colour assigned to at most one interval. Furthermore, we will show that the order of these intervals is induced by an order of the portals which we define next.

Let u_0 and u_1 be distinct vertices of G connected by an edge and consider a portal p_i. Choose edge $(u_1, u_2) \in E$ such that u_2 is on a shortest path from u_1 to p_i and such that $u_0 \rightarrow u_1 \rightarrow u_2$ makes the sharpest possible left turn at u_1 (if a left turn is not possible we regard the least possible right turn as a sharpest possible left turn and we regard a turn of angle π as a left turn of angle π).

Repeat this procedure by picking, for $j = 3, \ldots, r$, an edge (u_{j-1}, u_j) such that u_j is on a shortest path from u_{j-1} to p_i and such that $u_{j-2} \rightarrow u_{j-1} \rightarrow u_j$ makes the sharpest possible left turn at u_{j-1}; here, r is the smallest index such that $u_j = p_i$. The resulting path $u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_r$ is uniquely defined and is a shortest path from u_1 to p_i. We denote it by $\overleftarrow{P_i}(u_0, u_1)$. We define $\overleftarrow{P_i}'(u_0, u_1) = u_0 \rightarrow \overleftarrow{P_i}(u_0, u_1)$. In the following, we will write $\overleftarrow{P_i}$ resp. $\overleftarrow{P'_i}$ as a shorthand for $\overleftarrow{P_i}(a', a)$ resp. $\overleftarrow{P'_i}(a', a)$, where a and a' are defined as above.

For two distinct portals p_i and p_j we write $p_i \prec^f_a p_j$ if $p_i \in \overleftarrow{P_j}$ or if $\overleftarrow{P'_i}$ makes a sharper left turn than $\overleftarrow{P'_j}$ at some shared interior vertex. As an example, in Figure 1, $p_i \prec^f_a p_j$ since $\overleftarrow{P'_i}$ makes a sharper left turn than $\overleftarrow{P'_j}$ at vertex v.

Lemma 2. *If paths $\overleftarrow{P_i}$ and $\overleftarrow{P_j}$ split at a vertex v they cannot meet after v.*

Lemma 3. *The relation $\prec_a^f$ above is a strict total order of the portals.*

We now show the relation between the order $\prec_a^f$ of portals and the order $<_a^f$ of vertices in f_B.

Theorem 1. *Let p and q be two distinct points of f_B and assume that $c_a(p) = c_i$ and $c_a(q) = c_j$, $i \neq j$. Then $p_i \prec_a^f p_j$ if and only if $p <_a^f q$.*

Proof. By symmetry, it is enough to show that if $p_i \prec_a^f p_j$ then $p <_a^f q$.

So assume that $p_i \prec_a^f p_j$. Since $c_a(q) = c_j$, we have $p_i \notin \overleftarrow{P_j}$. It follows that there is a vertex v at which $\overleftarrow{P_i}$ and $\overleftarrow{P_j}$ split and $\overleftarrow{P_i'}$ makes a sharper left turn at v than $\overleftarrow{P_j}'$. By Lemma 2, the two paths do not meet again after v. In particular, $\overleftarrow{P_i}$ does not cross $\overleftarrow{P_j}$, see Figure 1.

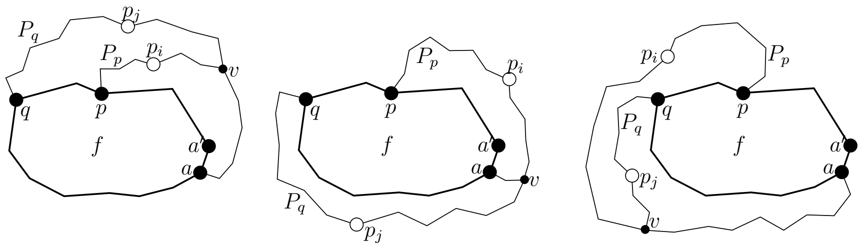


Fig. 1. The possible situations in the proof of Theorem 1 when $p_i \prec_a^f p_j$

Let P_p be a shortest path from p_i to p. Then P_p cannot intersect $\overleftarrow{P_j}$ since then p_i and p_j would both be on a shortest path from a to q with $d_G(a, p_i) < d_G(a, p_j)$, contradicting the assumption that $c_a(q) = c_j$.

Let P_q be a shortest path from p_j to q. Then $\overleftarrow{P_i}$ cannot intersect P_q since otherwise, p_i and p_j would both be on a shortest path from a to p with $d_G(a, p_j) < d_G(a, p_i)$, contradicting the assumption that $c_a(p) = c_i$.

Furthermore, P_q cannot intersect P_p. For assume it did. Then there would be a shortest path from a to p through p_j and a shortest path from a to q through p_i. If $d_G(a, p_i) < d_G(a, p_j)$ then $c_a(q) \neq c_j$ and if $d_G(a, p_j) < d_G(a, p_i)$ then $c_a(p) \neq c_i$. Hence, $d_G(a, p_i) = d_G(a, p_j)$. But then $i < j$ would imply $c_a(q) \neq c_j$ and $i > j$ would imply $c_a(p) \neq c_i$, contradicting the colours assigned to p and q.

It follows from the above that paths $\overleftarrow{P_i}P_p$ and $\overleftarrow{P_j}P_q$ do not intersect except in the vertices they share until reaching v, see Figure 1. This implies that $p <_a^f q$, showing the theorem. □

Corollary 1. *Interval f_B can be split up into $O(k)$ sub-intervals such that points in the same sub-interval are assigned the same colour w.r.t. vertex $a \in f_A$.*

If the sub-intervals of Corollary 1 are picked such that they have maximal size, we refer to them as *colour intervals of a*.

We now show how the colour intervals of a can be computed efficiently.

Lemma 4. *The order of colour intervals of a can be computed in $O(k \log^2 n)$ time assuming $O(kn \log n)$ time for preprocessing. The preprocessing step is independent of a and f.*

Proof. We will prove the lemma by presenting a data structure that, given distinct portals p_i and p_j, determines whether $p_i \prec_a^f p_j$ and does so for any a and f. We will show how to construct this data structure in $O(kn \log n)$ time such that it can determine whether $p_i \prec_a^f p_j$ in $O(\log n)$ time. This will allow us to sort the portals according to $\prec_a^f$ in time $O(k \log^2 n)$ time using a sorting algorithm like merge or heap sort since such an algorithm performs $O(k \log k)$ comparisons. This result together with Theorem 1 will show the lemma.

We first show how to construct the data structure. In the following, let E' be the set of directed edges obtained by regarding each edge of E as two oppositely directed edges.

With each edge $e = (u, v) \in E'$ and each portal p_i, we associate pointers $\pi_j(e, i)$, $j = 0, \ldots, j_e$. Pointer $\pi_j(e, i)$ points to the edge e_j of $\overleftarrow{P_i}(e)$ such that the number of edges between e and e_j in $\overleftarrow{P_i}(e)$ is $2^j - 1$, see Figure 2. Here, j_e is the largest j such that e_j exists. Note that $j_e = O(\log n)$ so the total number of pointers over all edges e and all portals p_i is $O(kn \log n)$.

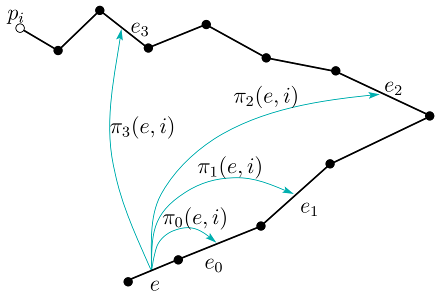


Fig. 2. Example with π-pointers from edge e to edges $e_0 = \pi_0(e, i)$, $e_1 = \pi_1(e, i)$, $e_2 = \pi_2(e, i)$, and $e_3 = \pi_3(e, i)$ on $\overleftarrow{P_i}(e)$. Here, $j_e = 3$.

It is easy to show that all π-pointers can be computed in $O(kn \log n)$ time. Now, given these pointers, for any edge e and any portals p_i and p_j, binary search allows us to determine, in $O(\log n)$ time, the vertex at which $\overleftarrow{P_i}(e)$ and $\overleftarrow{P_j}(e)$ split. If they do not split, binary search will also detect this and determine whether $p_i \in \overleftarrow{P_j}(e)$ or $p_j \in \overleftarrow{P_i}(e)$. From this it follows that we can determine whether $p_i \prec_a^f p_j$ in $O(\log n)$ time, given these pointers. □

Theorem 2. *Endpoints of colour intervals of a can be computed in $O(k \log^2 n)$ time assuming $O(kn \log n)$ preprocessing time. The preprocessing step is independent of a and f.*

Proof. We start by computing the order of colour intervals of a. By Lemma 4, this can be done in $O(k \log^2 n)$ time with $O(kn \log n)$ preprocessing time. Let π be the permutation of $\{1, \ldots, k\}$ such that $p_{\pi(1)} \prec_a^f \cdots \prec_a^f p_{\pi(k)}$.

We then compute the colour of the first point $b_{\min}$ and the last point $b_{\max}$ of f_B w.r.t. the order $<_a^f$. This can be done in time proportional to the number k of colours since SSSP lengths for each portal have been precomputed.

If $c_a(b_{\min}) = c_a(b_{\max})$ then by Theorem 1, all points between $b_{\min}$ and $b_{\max}$ have this colour. In this case, the algorithm associates this colour with the sub-interval between the two vertices and returns (the sub-interval is not stored explicitly, only its end vertices $b_{\min}$ and $b_{\max}$).

Otherwise, a vertex $b \in f_B$ is picked, such that the number of edges in f_B before resp. after b w.r.t. the order $<_a^f$ is (approximately) the same, and its colour $c_a(b)$ is computed. Let i be the index such that $c_{\pi(i)} = c_a(b)$. The algorithm calls itself recursively on vertices between $b_{\min}$ and b with colours $c_{\pi(1)}, \ldots, c_{\pi(i)}$. And it calls itself recursively on vertices between b and $b_{\max}$ with colours $c_{\pi(i)}, \ldots, c_{\pi(k)}$.

The recursion stops when $b_{\min}$ and $b_{\max}$ are the endpoints of a single edge e of f_B. If $c_a(b_{\min}) = c_a(b_{\max})$, the algorithm associates this colour with e and returns. Otherwise, we need the following simple observation: there is a point p on e such that all points on $b_{\min}p$ have colour $c_a(b_{\min})$ and all points on $pb_{\max}$ have colour $c_a(b_{\max})$. Furthermore, p can be computed in $O(1)$ time, given these two colours. The algorithm associates the two colours with their respective segments of e and returns.

When the algorithm terminates, each colour c_i is associated with $O(\log n)$ sub-intervals and their union defines the colour interval of a with colour c_i. Finding the colour intervals of a from these $O(k \log n)$ sub-intervals takes $O(k \log n)$ time. What remains is to show that the algorithm above has $O(k \log^2 n)$ running time.

Let $T(m, k)$ be a function expressing the time for the above algorithm where m is the number of edges of f_B and k is the number of colours. If we assume that vertices of f_B are stored in an array then the point b that splits points of f_B into two equal halves can be found in $O(1)$ time. Thus, there is a constant $c' > 0$ such that the algorithm uses at most $c'k$ time steps excluding time spent in recursive calls. There is also a constant c'' such that $T(m, k) \le c''k$ when $m \le 2$. Let $c = \max\{c', c''\}$. Then (ignoring floors and ceilings)

$$T(m, k) \le ck + T(m/2, k_1) + T(m/2, k_2).$$

where $m > 2$ and $k_1, k_2 \in \{1, \ldots, k\}$, $k_1 + k_2 = k + 1$.

Let $\tilde{T}(m, k)$ be the running time $T(m, k)$ minus a value of $c \log n$ charged to each split vertex b encountered in the current and in recursive calls. Then it can be shown by induction on $m \ge 2$ that $\tilde{T}(m, k) < ck \log m$, implying that $T(m, k) \le ck \log m + xc \log n$, where x is the total number of split vertices. This number is proportional to the number of sub-intervals returned by the algorithm which is $O(k \log m)$. □

Note that a colour interval I need not be closed, i.e. one or both of the endpoints need not belong to I. We conclude this section with the following simple result which will prove useful when we compute detours between points that may be interior points of edges.

Lemma 5. *Let p a point of edge $e = (u, v)$ of G and let q be a point in G. Suppose that p_i is on a shortest path from u to q and that p_j is on a shortest path from v to q. Then either p_i or p_j is on a shortest path from p to q.*

Proof. A shortest path from p to q goes through either u or v. □

5 The Detour of Points in a Face

In this section, we show how to compute $\delta_G(f_A, f_B)$ in $O(|f|k \log^2 n)$ expected time.

We start by computing, for each edge $e = (u, v) \in f_A$, $O(k)$ colour intervals of u and of v using $O(k \log^2 n)$ time (with $O(kn \log n)$ preprocessing) and take the union of the endpoints of these colour intervals. This gives $O(k)$ smaller sub-intervals which we associate with e. The total running time for this over all edges is $O(|f|k \log^2 n)$.

Now, let P be one of the sub-intervals associated with edge $e = (u, v)$. Then there are $i, j \in \{1, \ldots, k\}$ such that $c_u(p) = c_i$ and $c_v(p) = c_j$ for all $p \in P$. Hence, for any point $q \in P$, p_i is on a shortest path from u to q and p_j is on a shortest path from v to q. Lemma 5 implies that for any point $p \in e$ and any point $q \in P$, either p_i or p_j is on a shortest path from p to q.

We refer to P as a *type* 1*-interval* (of e) if $c_i \neq c_j$ and a *type* 2*-interval* (of e) if $c_i = c_j$.

For any edge e of f_A and any type i-interval P of e, $i = 1, 2$, we may assume that P is a closed interval having endpoints in vertices of f_B. For otherwise, we could compute the maximum detour between e and the first resp. last edge e' of P (all other edges of P have endpoints in vertices of f_B). Computing $\delta_G(e, e')$ is a constant-size problem (when SSSP lengths for each portal have been precomputed) since we know that for each point $p \in e$ and each point $p' \in e$, there is a shortest path from p to p' through either of two portals p_i and p_j. Thus, it takes $O(1)$ time to compute $\delta_G(e, e')$ (see also [1]). Over all e and P, this amounts to $O(|f|k)$ time.

The value $\delta_G(f_A, f_B)$ is computed in two phases. In phase i, the maximum detour between points in edges of f_A and points in associated type i-intervals is computed, $i = 1, 2$.

5.1 Phase 1

We will show that phase 1 takes $O(|f|k)$ time when shortest path lengths from portals and colour intervals have been computed.

The algorithm for this phase is straightforward. For each edge e of f_A it considers all edges e' of each type 1-interval of e and computes $\delta_G(e, e')$ in constant time.

To show that phase 1 takes $O(f|k|)$ time, we need to show that the number of edge pairs (e, e') is $O(f|k|)$. To this end, we introduce a so called *dual colouring* of vertices of f_A for each vertex of f_B. Let b be a vertex of f_B. Then vertex $a \in f_A$ is given *dual colour* $c_b(a)$, defined as the colour $c_a(b)$.

Assigning dual colours to all vertices of f_A partitions this set into maximal sub-intervals with vertices in each sub-interval having the same dual colour. We call these sub-intervals the *dual colour intervals of* b. These dual colour intervals will help us bound the number of edge pairs (e, e'). First, we need two lemmas.

Lemma 6. *Let b be a vertex of f_B and let a, a_1, a_2 be vertices of f_A such that $a_1 <_b^f a <_b^f a_2$, $c_b(a_1) = c_b(a_2) = c_i$, and $c_b(a) = c_j$, $i \neq j$. Then for any vertex a' of f_A, $c_b(a') \neq c_j$ if either $a' <_b^f a_1$ or $a' >_b^f a_2$.*

Lemma 7. *The number of dual colour intervals of a vertex $b \in f_B$ is $O(k)$.*

Proof. Let $N(k)$ be the maximum number of dual colour intervals of b when the number of distinct colours in these intervals is exactly k. We will show that $N(k) \leq 2k - 1$. The proof is by induction on $k \geq 1$. If $k = 1$ then there is only one dual colour interval and we have $N(k) = 1 = 2k - 1$.

Now, suppose that $k > 1$ and that $N(k') \leq 2k' - 1$ for all k' less than k. Let c_i be the colour of the first dual colour interval w.r.t. the order $<_b^f$. By Lemma 6, there is a finite number r of dual colour intervals with colour c_i (in fact at most k). Let $I_1, \ldots, I_s$ be the intervals between each consecutive pair of these dual colour intervals and let $k_1, \ldots, k_s$ be the number of colours in each of them. Note that $s \geq r - 1$. Also note that by the choice of c_i and by Lemma 6, two points in two different I-intervals cannot have the same colour since for at least one of the two intervals, the colour of the dual colour interval preceding and succeeding it is c_i. From this and from the fact that the I-intervals do not contain colour c_i, $\sum_{j=1}^{s} k_j = k - 1$. Applying the induction hypothesis, this gives

$$N(k) \leq r + \sum_{j=1}^{s} N(k_j) \leq r + \sum_{j=1}^{s} 2k_j - 1 = r - s + 2(k-1) \leq 1 + 2(k-1) = 2k - 1.$$

□

We are now ready to bound the running time for the phase 1 algorithm.

Theorem 3. *Phase* 1 *runs in* $O(|f|k)$ *time.*

Proof. Consider edges $e = (u, v) \in f_A$ and $e' = (u', v') \in f_B$ such that e' is an edge of a type 1-interval of e. Let $c_i = c_u(u') = c_u(v')$ and let $c_j = c_v(u') = c_v(v')$. Since u' is a vertex of f_B, dual colours $c_{u'}(u) = c_i$ and $c_{u'}(v) = c_j$ are well-defined and since $i \neq j$ by assumption, these two dual colours are distinct, implying that two dual colour intervals of b meet at e.

It follows by the above and by Lemma 7 that for each e', $O(k)$ edges e are picked. Thus, the total number of edge pairs considered by the algorithm is $O(|f|k)$. By our earlier discussion, this suffices to show the theorem. □

5.2 Phase 2

We now consider the problem of computing the maximum detour between points of edges of f_A and points of type 2-intervals.

For any pair (e, P), where e is an edge of f_A and P is a type 2-interval of e, there is a portal p_i such that for any point $p \in e$ and any point $q \in P$, p_i is on a shortest path from p to q. In the following, we consider all such pairs for a fixed p_i. We will show that computing the maximum detour δ_i over all these pairs can be done in $O(|f| \log^2 |f|)$ expected time. From this, it will follow that phase 2 takes $O(|f| k \log^2 |f|)$ expected time.

Before showing how to compute δ_i, we need the idea of a canonical decomposition of f_B, defined next.

Canonical Decomposition. Define $b_1, \ldots, b_m$ as the interval of vertices of f_B ordered according to $<_a^f$ for some arbitrary vertex $a \in f_A$. Consider splitting this interval at vertex $b_j = b_{\lceil m/2 \rceil}$ and repeat this process recursively on the two sub-intervals, stopping when an interval containing only two vertices is reached. This gives us $O(m) = O(|f|)$ intervals of total size $O(|f| \log |f|)$ which we refer to as *canonical intervals*. The subgraphs of f_B induced by these canonical intervals are referred to as *canonical subgraphs*. The set of these subgraphs, which we denote by $\mathcal{C}$, can be found in $O(|f| \log |f|)$ time.

Every sub-interval of $b_1, \ldots, b_m$ can be decomposed into $O(\log |f|)$ canonical intervals in $O(\log |f|)$ time. This is easily seen by applying a greedy algorithm that picks canonical intervals as large as possible. We refer to such a decomposition as a *canonical decomposition*.

Let e be an edge of f_A. By the assumption earlier that type 2-intervals end in vertices and by Theorem 1, the union of all type 2-intervals of e of colour c_i is exactly the set of points of f_B between two vertices of $b_1, \ldots, b_m$. We compute a canonical decomposition of the sub-interval between these two vertices and add a pointer to e from each canonical subgraph corresponding to canonical intervals in this decomposition. This is done for all e in f_A.

When finished we have a total of $O(|f| \log |f|)$ pointers from canonical subgraphs in $\mathcal{C}$ to edges of f_A. Note that some canonical subgraphs may contain pointers to several edges. The total time spent on constructing $\mathcal{C}$ and on finding pointers is $O(|f| \log |f|)$.

Observe that δ_i is the maximum of $\delta_G(e, C)$ over all pairs consisting of an edge $e \in f_A$ and a canonical subgraph $C \in \mathcal{C}$ with a pointer to e. We now show how to compute this maximum in $O(|f| \log^2 |f|)$ expected time.

Sweep-plane Algorithm. To efficiently compute the maximum over pairs of edges and canonical subgraphs, we will use the idea of lifting and lowering points followed by a sweep-plane algorithm as described in [7]. In order to do this, we consider the following decision problem below: given $\delta \in \mathbb{R}$, is $\delta_i \geq \delta$? If we can answer this quickly we can compute δ_i in low expected time using a randomized algorithm by Chan [3] as described in [7].

For each canonical subgraph $C \in \mathcal{C}$, we lift each point $p \in C$ to height $d_G(p_i, p)$. And for each edge $e \in f_A$, we lower each point $p \in e$ to height $-d_G(p, p_i)$. Since we have precomputed SSSP lengths for portal p_i, this lifting/lowering can be done in $O(|f| \log |f|)$ time since the total size of all canonical subgraphs and edges is $O(|f| \log |f|)$.

Let e be a vertex in f_A and let C be a canonical subgraph with a pointer to e. Then it is clear that the height difference between a point $p \in e$ and a point $q \in C$ equals $d_G(p, q)$.

For each lifted and lowered point p, we associate a cone extending downwards from p and spanning an angle of $\alpha = 2\arctan(1/\delta)$. Then as shown in [7], $\delta_i \geq \delta$ if and only if a cone of a lowered point of some edge is contained in a cone of a lifted point of some canonical subgraph with a pointer to that edge.

Now, we sweep a plane over the cones. The sweep-plane is parallel to the x-axis and forms an angle of $(\pi - \alpha)/2$ with the xy-plane. During the sweep, we maintain, for each canonical subgraph C, the intersection between the sweep-plane and the upper envelope of lifted points of C together with lowered points in edges that C points to. If it is detected that a cone of a lowered point is contained in a cone of a lifted point, the algorithm reports that $\delta_i \geq \delta$ and if no such event occurs, the algorithm reports that $\delta_i < \delta$.

It follows from the results of [7] that maintaining intersections between the sweep-plane and upper envelopes takes a total of $O(|f|\log^2|f|)$ time since the number of sweep-plane event points is $O(|f|\log|f|)$ and each event point takes $O(\log|f|)$ time to handle. However, this is under the assumption that no cone of a lifted resp. lowered point is contained in the interior of another cone of a lifted resp. lowered point (see [7] for details). It can be shown that this assumption is satisfied if we only consider values $\delta \geq \max\{\delta_G(G_A), \delta_G(G_B)\}$.

So by recursively computing $\delta_G(G_B)$ and $\delta_G(G_A)$ before computing $\delta_G(G_A, G_B)$ it follows that the above decision problem can be solved in $O(|f|\log^2|f|)$ time and Chan's algorithm gives us the following result.

Theorem 4. *Phase* 2 *runs in* $O(|f|k\log^2|f|)$ *expected time.*

We have shown that $\delta_G(G_A, G_B)$ can be computed in $O(n^{3/2}\log^2 n)$ expected time. Due to space constraints, we leave the details for recursively computing $\delta_G(G_A)$ and $\delta_G(G_B)$. But it can be shown that the time spent in a level of the recursion tree is $O(n^{3/2}\log^2 n)$ and we get the main result of our paper.

Theorem 5. *The maximum detour of a plane graph with* n *vertices can be computed in* $O(n^{3/2}\log^3 n)$ *expected time.*

The second main result of this paper shows that if the graph has bounded *treewidth* (see definition in [2]), faster running time can be obtained.

Theorem 6. *The maximum detour of a plane graph with* n *vertices and bounded treewidth can be computed in* $O(n\log^3 n)$ *expected time.*

The proof of Theorem 6 follows easily from our results above and from [2].

6 Concluding Remarks

In this paper, we showed how to compute the maximum detour of a plane graph in $O(n^{3/2}\log^3 n)$ expected time. This is an improvement over the best known algorithm with $\Theta(n^2)$ running time. We also showed that if the graph has bounded treewidth, its maximum detour can be computed in $O(n \log^3 n)$ expected time.

We believe that by using parametric search as described in [1], we can obtain an algorithm computing the maximum detour of a plane graph in $O(n^{3/2} \operatorname{polylog} n)$ *worst-case* time and in $O(n \operatorname{polylog} n)$ *worst-case* time when the graph has bounded treewidth.

It would be interesting to try to beat the quadratic time bound also for the problem of computing the stretch factor of a plane geometric graph. This problem appears harder since pairs of vertices achieving the maximum detour need not be co-visible.

References

1. Agarwal, P.K., Klein, R., Knauer, C., Langerman, S., Morin, P., Sharir, M., Soss, M.: Computing the Detour and Spanning Ratio of Paths, Trees and Cycles in 2D and 3D. Discrete and Computational Geometry 39(1), 17–37 (2008)
2. Caballo, S., Knauer, C.: Algorithms for Graphs With Bounded Treewidth Via Orthogonal Range Searching, Berlin (manuscript, 2007)
3. Chan, T.M.: Geometric applications of a randomized optimization technique. Discrete Comput. Geom. 22(4), 547–567 (1999)
4. Ebbers-Baumann, A., Klein, R., Langetepe, E., Lingas, A.: A Fast Algorithm for Approximating the Detour of a Polygonal Chain. Comput. Geom. Theory Appl. 27, 123–134 (2004)
5. Eppstein, D.: Spanning trees and spanners. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 425–461. Elsevier Science Publishers, Amsterdam (2000)
6. Frederickson, G.N.: Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput. 16, 1004–1022 (1987)
7. Langerman, S., Morin, P., Soss, M.: Computing the Maximum Detour and Spanning Ratio of Planar Paths, Trees and Cycles. In: Alt, H., Ferreira, A. (eds.) STACS 2002. LNCS, vol. 2285, pp. 250–261. Springer, Heidelberg (2002)
8. Lipton, R.J., Tarjan, R.E.: A Separator Theorem for Planar Graphs. STAN-CS-77-627 (October 1977)
9. Narasimhan, G., Smid, M.: Geometric Spanner Networks. Cambridge University Press, Cambridge (2007)
10. Smid, M.: Closest point problems in computational geometry. In: Sack, J.-R., Urrutia, J. (eds.) Handbook of Computational Geometry, pp. 877–935. Elsevier Science Publishers, Amsterdam (2000)

Appendix

Proof of Lemma 2

Suppose the lemma does not hold. Let w be a vertex where $\overleftarrow{P_i}$ and $\overleftarrow{P_j}$ meet after v. Assume w.l.o.g. that $\overleftarrow{P_i}$ makes a sharper left turn at v than $\overleftarrow{P_j}$. Replacing the subpath of $\overleftarrow{P_j}$ from v to w by the subpath of $\overleftarrow{P_i}$ from v to w gives a shortest path from a to p_j but this contradicts the assumption that $\overleftarrow{P'_j}$ makes the sharpest possible left turn at v.

Proof of Lemma 3

Let p_{i_1}, p_{i_2}, and p_{i_3} be three distinct portals. Clearly, $p_{i_1} \prec_a^f p_{i_2}$ or $p_{i_2} \prec_a^f p_{i_1}$, showing that the relation $\prec_a^f$ is total.

We need to show that $p_{i_1} \prec_a^f p_{i_2}$ and $p_{i_2} \prec_a^f p_{i_1}$ cannot both hold and we need to show transitivity: if $p_{i_1} \prec_a^f p_{i_2}$ and $p_{i_2} \prec_a^f p_{i_3}$ then $p_{i_1} \prec_a^f p_{i_3}$.

To show the first part, suppose for the sake of contradiction that $p_{i_1} \prec_a^f p_{i_2}$ and $p_{i_2} \prec_a^f p_{i_1}$. Then we cannot have $p_{i_1} \in \overleftarrow{P_{i_2}}$ and we cannot have $p_{i_2} \in \overleftarrow{P_{i_1}}$. Consider the first vertex v at which paths $\overleftarrow{P_{i_1}}$ and $\overleftarrow{P_{i_2}}$ split. Then they cannot meet after v by Lemma 2. But this implies that one of the two paths $\overleftarrow{P_{i'_1}}$ and $\overleftarrow{P_{i'_2}}$ cannot make a sharper left turn than the other path at any shared interior vertex, a contradiction.

To show transitivity, suppose $p_{i_1} \prec_a^f p_{i_2}$ and $p_{i_2} \prec_a^f p_{i_3}$. We need to show that either $p_{i_1} \in \overleftarrow{P}_{i_3}$ or that $\overleftarrow{P_{i_1}}'$ makes a sharper left turn than $\overleftarrow{P}'_{i_3}$ at some interior vertex of $\overleftarrow{P}'_{i_3}$.

Assume first that $p_{i_1} \in \overleftarrow{P_{i_2}}$. If $p_{i_2} \in \overleftarrow{P_{i_3}}$ then $p_{i_1} \in \overleftarrow{P_{i_3}}$ (Figure 3(i)). If $p_{i_2} \notin \overleftarrow{P_{i_3}}$ then $\overleftarrow{P_{i_2}}'$ makes a sharper left turn than $\overleftarrow{P_{i_3}}'$ at some interior vertex v of $\overleftarrow{P_{i_3}}'$. If $v \in \overleftarrow{P_{i_1}} \setminus \{p_{i_1}\}$ (Figure 3(ii)) then $\overleftarrow{P_{i_1}}'$ makes a sharper left turn than $\overleftarrow{P_{i_3}}'$ at v. And if $v \notin \overleftarrow{P_{i_1}} \setminus \{p_{i_1}\}$ (Figure 3(iii)) then $p_{i_1} \in \overleftarrow{P_{i_3}}$. In all cases, $p_{i_1} \prec_a^f p_{i_3}$.

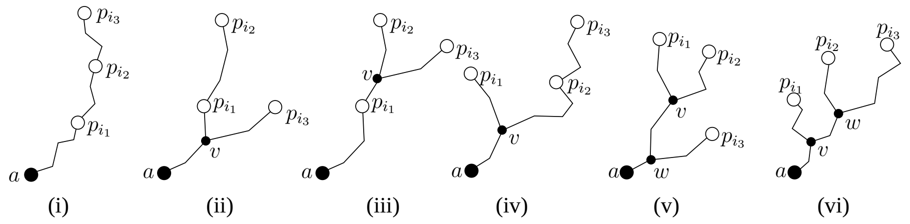


Fig. 3. The six cases considered in Lemma 3.

Now, assume that $p_{i_1} \notin \overleftarrow{P_{i_2}}$. Then $\overleftarrow{P_{i_1}}'$ makes a sharper left turn than $\overleftarrow{P_{i_2}}'$ at some interior vertex v of $\overleftarrow{P_{i_2}}'$. If $p_{i_2} \in \overleftarrow{P_{i_3}}$(Figure 3(iv)) then $\overleftarrow{P_{i_1}}'$ makes a sharper left turn than $\overleftarrow{P_{i_3}}'$ at v. If $p_{i_2} \notin \overleftarrow{P_{i_3}}$ then let w be a vertex such that $\overleftarrow{P_{i_2}}'$ makes a sharper left turn than $\overleftarrow{P_{i_3}}'$ at w. If $w \in \overleftarrow{P_{i_1}}$ (Figure 3(v)) then $\overleftarrow{P_{i_1}}'$ makes a sharper left turn than $\overleftarrow{P_{i_3}}'$ at w. And if $w \notin \overleftarrow{P_{i_1}}$ (Figure 3(vi)) then $\overleftarrow{P_{i_1}}'$ makes a sharper left turn than $\overleftarrow{P_{i_3}}'$ at v. So again, $p_{i_1} \prec_a^f p_{i_3}$.

Part of the Proof of Lemma 4

In this section, we show how to prove that all π-pointers can be computed in $O(kn \log n)$ time. More specifically, given a portal p_i, we show how to compute pointers $\pi_j(e, i)$, $e \in E$, $j = 1, \ldots, j_e$, in a total of $O(n \log n)$ time.

For each edge (u, v) of E', we colour it white if a shortest path from u to p_i goes through v, that is, if $d_G(u, p_i) = |uv| + d_G(v, p_i)$. Otherwise, we colour it black.

Now, for each vertex v of G, we consider the edges of E' starting or ending in v in counter-clockwise order. For each edge (u, v) ending in v, we add a pointer to the previous (in the counter-clockwise order) white edge (v, w) that starts in v. Note that (v, w) is the edge satisfying that $u \to v \to w$ makes the sharpest possible left turn such that (v, w) is on a shortest path in G from v to p_i.

Next, we consider the edges of E' in some order $e_1, \ldots, e_{|E'|}$. For edge $e_1 = (u_1, v_1)$, we compute $\overleftarrow{P_i}(e_1)$ by starting in v_1 and then computing subsequent vertices using the pointers just added. We colour e_1 red and as we visit the edges of $\overleftarrow{P_i}(e_1)$, we colour them red as well.

We then compute pointers $\pi_j(e, i)$ for each edge $e \in \overleftarrow{P_i'}(e_1)$ and for each $j = 0, \ldots, j_e$. Below, we show how to do this efficiently.

For edge e_j, $j > 1$, we do exactly the same as for e_1 except that we stop when reaching a red edge. All visited edges including e_j are coloured red and π-pointers are computed for each of these edges.

The above algorithm is correct since it computes π-pointers for all edges. Its running time, excluding the time to compute π-pointers, is $O(n \log n)$. To see this, note that sorting edges counter-clockwise for each vertex takes a total of $O(n \log n)$ time. Colouring edges black and white takes $O(n)$ time. Finally, visiting (parts of) paths $\overleftarrow{P_i}(e_j)$ takes time proportional to the number of edges coloured red which is $O(n)$.

What remains is to show how to compute π-pointers in $O(n \log n)$ time. We observe that $\pi_j(e, i)$, $j = 0, \ldots, j_e$, can be computed in a total of $O(\log n)$ time if π-pointers of all edges of $\overleftarrow{P_i}(e)$ have been computed. This follows easily from repeated applications of the identity $\pi_j(e, i) = \pi_{j-1}(\pi_{j-1}(e, i), i)$, $j > 0$ (see Figure 2). Hence, by computing π-pointers in the opposite order of the order in which edges are coloured red, we obtain all π-pointers in $O(n \log n)$ time.

Proof of Lemma 6

Let a' be a vertex of f_A such that either $a' <_b^f a_1$ or $a' >_b^f a_2$. Portal p_i is on some shortest path P_1 from a_1 to b. Pick P_1 such that the subpath P_1' from a_1 to p_i makes sharpest possible left turns as described in the previous section. In a similar way, we define P_2 and P_2' for a_2. Note that any path from a to b must intersect either P_1' or P_2', see Figure 4.

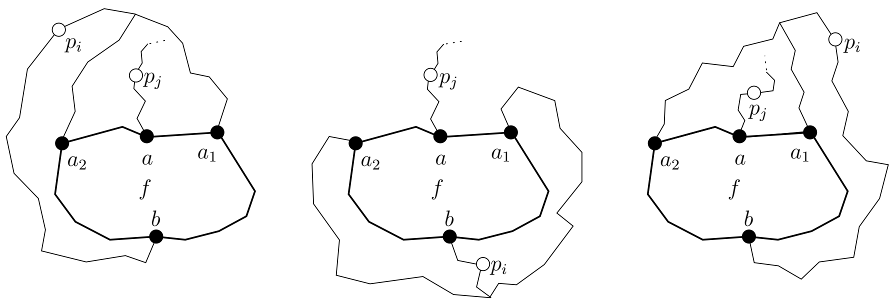


Fig. 4. The situations considered in the proof of Lemma 6.

Portal p_j is on a shortest path P from a to b. Pick P such that the subpath P' from a to p_j makes sharpest possible left turns. Path P' cannot cross P_1 for otherwise $c_a(b) \neq c_j$, and $p_j \notin P_1'$ for otherwise $c_{a_1}(b) \neq c_i$. Similarly, P' cannot cross P_2 for otherwise $c_{a_2}(b) \neq c_i$, and $p_j \notin P_2'$ for otherwise $c_{a_2}(b) \neq c_i$. Furthermore, p_j cannot belong to the subpath shared by P_1 and P_2 from p_i to b since then $c_a(b) \neq c_j$.

It follows that any shortest path from a' to p_j crosses either P_1' or P_2'. This implies that $c_b(a') = c_{a'}(b) \neq c_j$, as requested.

More Details on the Sweep-Plane Algorithm

We left out some details in the description of the sweep-plane algorithm which we now consider. We will show that the assumption that no cone of a lifted (lowered) point is contained in another cone of a lifted (lowered) point is satisfied when $\delta \geq \max\{\delta_G(G_A), \delta_G(G_B)\}$

For suppose that $\delta \geq \delta_G(G_B)$ and consider two points p and p' belonging to the same canonical subgraph.

Let h be the height of p and let h' be the height of p'. Assume w.l.o.g. that $h \geq h'$. Then

$$h - h' = d_G(p, p_i) - d_G(p', p_i).$$

We know that $d_G(p, p') \leq \delta_G(G_B)|pp'| \leq \delta|pp'|$. This gives

$$\frac{h - h'}{|pp'|} = \frac{d_G(p, p_i) - d_G(p', p_i)}{|pp'|} \leq \frac{d_G(p, p')}{|pp'|} \leq \delta,$$

showing that the cone associated with p' cannot belong to the interior of the cone associated with p. A similar argument shows that with $\delta \geq \delta_G(G_A)$, no cone of a lowered point is contained in the interior of another cone of a lowered point.

Non-simple Faces

Suppose f is non-simple. The walk of f defined in the beginning of Section 4 still applies if we inflate the edges and allowing an edge to be visited twice, see Figure 5(a). Then running through the arguments of the preceding sections, it can be seen that all results still hold (as an example, compare Figure 5(b) with Figure 1).

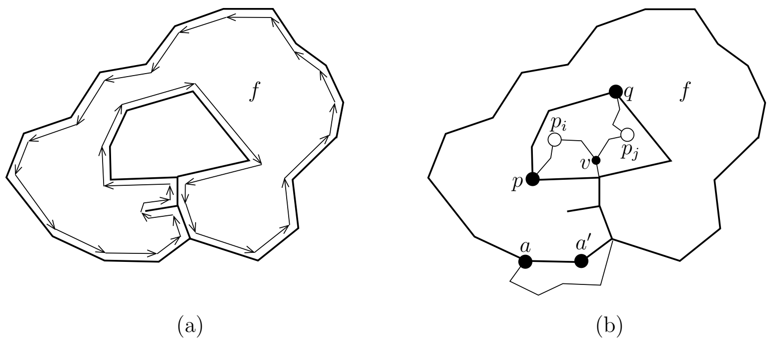


Fig. 5. (a): The walk of f when f is non-simple. (b): Theorem 1 holds also when f is non-simple.

Dealing with Recursive Calls

In the paper, we showed how to compute $\delta_G(G_A, G_B)$ in $O(n^{3/2} \log^2 n)$ time. In this section, we present an algorithm that recursively computes $\delta_G(G_A)$ (computing $\delta_G(G_B)$ is dealt with in a similar way) and we will analyze its running time. From this analysis it will follow that δ_G can be computed in $O(n^{3/2} \log^3 n)$ expected time.

Let $n_A = |A|$ be the number of vertices of A. We start by applying the separator theorem to G_A which partitions A into three sets, A_1, A_2, and C_A such that no edge joins a vertex in A_1 with a vertex in A_2, neither A_1 nor A_2 contains more than $n_A/2$ vertices, and C_A contains no more than $\frac{2\sqrt{2}}{1-\sqrt{2/3}}\sqrt{n_A}$ vertices. Let G_{A_1} be the subgraph of G_A induced by $A_1 \cup C_A$ and let G_{A_2} be the subgraph of G_A induced by $A_2 \cup C_A$. We have

$$\delta_G(G_A) = \max\{\delta_G(G_{A_1}), \delta_G(G_{A_2}), \delta_G(G_{A_1}, G_{A_2})\}.$$

In the following, we will present an algorithm for computing $\delta_G(G_{A_1}, G_{A_2})$. From this it will easily follow how to handle subgraphs in any recursion level.

Let P be the set of portals for G and let $P_A = P \cup C_A$. Given a vertex $a_1 \in A_1$ and a vertex $a_2 \in A_2$, any path in G from a_1 to a_2 must contain at least one vertex of P_A. Thus, by defining P_A as the set of portals for G_A, most of the results we presented in order to compute $\delta_G(G_A, G_B)$ now also apply to the problem of computing $\delta_G(G_{A_1}, G_{A_2})$. However, three problems need to be dealt with.

The first problem is that G_A may be disconnected and it causes problems when defining cyclic orderings of faces. We deal with this as follows. Let f be a face of G such that $f \cap A \neq \emptyset$. Then as the cyclic ordering of the vertices of $f \cap A$, we use that which is induced by the cyclic ordering of vertices of f. Essentially, this defines faces of G_A where "holes" are allowed.

The second problem is that of computing SSSP distances from each portal of P_A to all vertices in G_A. The running time for this should depend on the size of G_A, not the size of G. To obtain this, note that we only need to compute SSSP distances from portals of C_A since we have computed distances from each portal in P to all vertices of G and in particular to vertices of G_A.

So consider a portal $p_i \in C_A$. We compute SSSP distances from p_i to vertices in G_A using Dijkstra's algorithm but with a different initialization. We are already given distances from p_i to portals in P so we add these portals to the priority queue and associate a shortest path distance from p_i to each of them. We also add p_i to the priority queue and set the shortest distance from p_i to itself equal to zero. The rest of the algorithm runs like normal Dijkstra on G_A. This way, we find SSSP distances in G from p_i to all vertices in G_A in $O(n_A \log n_A)$ time. Since $|C_A| = O(\sqrt{n_A})$, computing SSSP distances for all portals of C_A takes a total of $O(n_A^{3/2} \log n_A)$ time.

The third and final problem we need to deal with is determining the order of colour intervals of $f \cap G_{A_2}$ for each vertex of $f \cap G_{A_1}$ where f is a face. Lemma 4 states that the order of colour intervals of a vertex of f_A can be computed in $O(k \log^2 n)$ time with $O(kn \log n)$ time for preprocessing. Looking at the proof of the lemma, we see that the preprocessing step only considers portals of G, not the set of portals P_A for G_A. It thus needs to be recomputed for G_A. But a priori, this requires that we look at the entire graph G since shortest paths from vertices in G_A to portals in P_A need not be fully contained in G_A. This will make our recursive algorithm too slow.

We modify the preprocessing step for G_A as follows. We compute π-pointers for each vertex of G_A and each portal of P_A as in the proof of Lemma 4 except that when finding a path to a portal $p_i \in P_A$ we stop if we reach a portal of P (or as before, if we reach a red edge or p_i).

This modification gives $O(k_A n_A \log n_A)$ preprocessing time where $k_A = |P_A|$. We now show how this preprocessing can be used to efficiently determine whether $p_i \prec_a^f p_j$ for two distinct portals $p_i, p_j \in P_A$ where a is a vertex of G_{A_1} belonging to a face f of G.

Let a' be the successor of a in f w.r.t. $<^f_a$. To determine whether $p_i \prec^f_a p_j$, we apply binary searches as before in paths $\overleftarrow{P_i}(a', a)$ and $\overleftarrow{P_j}(a', a)$. If a split is detected, or if it is detected that $p_i \in \overleftarrow{P_j}(a', a)$ or $p_j \in \overleftarrow{P_i}(a', a)$ then we can correctly decide if $p_i \prec^f_a p_j$.

The only problem that may arise is if both binary searches end in an edge e from which there are no π-pointers and it still has not been determined whether $p_i \prec^f_a p_j$. But in this case, one of the endpoints of e must be a portal of P_A and this portal must belong to both $\overleftarrow{P_i}(a', a)$ and $\overleftarrow{P_j}(a', a)$ implying that no colour intervals of a will get colour c_i or c_j. Hence, it is irrelevant whether $p_i \prec^f_a p_j$ and we may simply delete the two portals in the algorithm that sorts portals. When the sorting algorithm terminates, the order of the remaining portals will give the order of colour intervals of a.

With the above three modifications, we get an algorithm that computes $\delta_G(G_{A_1}, G_{A_2})$ in $O(k_A n_A \log^2 n_A)$ expected time. We compute $\delta_G(G_{A_1})$ and $\delta_G(G_{A_2})$ recursively where the portals for G_{A_1} are defined as those that belong to $P_A \cap G_{A_1}$ together with those obtained when applying the separator theorem to G_{A_1} (and similarly for G_{A_2}).

More generally, consider a subgraph G' in some node of the recursion tree. If the size of G' is less than some constant, a brute-force algorithm that computes APSP-distances for vertices in G' is applied to find $\delta_G(G')$ in constant time.

Otherwise, let G_{c_1} and G_{c_2} be the subgraphs in the two child nodes obtained by applying the separator theorem to G'. Let P' be the set of portals of G'. Then for $i = 1, 2$, the set of portals of G_{c_i} is defined as the union of $P' \cap G_{c_i}$ and the set of portals obtained by applying the separator theorem to G_{c_i}.

Let k_p denote the number of portals of G' that are also portals of the subgraph of G belonging to the parent node of the node containing G', let k' be the number of additional portals of G', and let n' be the number of vertices of G'. Then similar arguments as above show that $\delta_G(G_{c_1}, G_{c_2})$ can be computed in $O((k_p + k')n' \log^2 n')$ expected time. Since $k' = O(\sqrt{n'})$, this can be rewritten as $O(k_p n' \log^2 n' + n'\sqrt{n'} \log^2 n')$.

Clearly, the sum of $O(n'\sqrt{n'} \log^2 n')$ over all non-leaf nodes of the recursion tree is $O(n^{3/2} \log^2 n)$. Let us bound the sum of $O(k_p n' \log^2 n')$ over all these nodes. Let $T(n', k_p)$ denote the total time spent in the subtree of the recursion tree rooted at a node containing a graph with n' vertices and sharing k_p portals with the subgraph in the parent node. Then

$$T(n', k_p) \le T(n'/2, k_1 + c\sqrt{n'}) + T(n'/2, k_2 + c\sqrt{n'}) + O(k_p n' \log^2 n'),$$

where $c = \frac{2\sqrt{2}}{1-\sqrt{2/3}}$ and $k_1 + k_2 \le k_p$.

It follows that the sum of all k_p in the ith level of the recursion tree is $O(\sum_{j=0}^{i} 2^j \sqrt{n/2^j})$ so the total time spent in this level is

$$O(\sum_{j=0}^{i} 2^j \sqrt{n/2^j} n/2^i \log^2 n) = O(n^{3/2} \log^2 n \sum_{j=0}^{i} 2^{j/2-i}).$$

Since $\sum_{j=0}^{i} 2^{j/2-i} \leq \sum_{j=0}^{i} 2^{j-i} < \sum_{j=0}^{\infty} 2^{-j} = 2$ and since there are $O(\log n)$ recursion levels, we have now obtained the result in Theorem 5.

Proof of Theorem 6

The treewidth of a graph is, in a sense, a measure of the complexity of graph. The following definition and lemma are taken from [2].

A *tree decomposition* of a graph $G = (V, E)$ is a pair (X, T), where $X = \{X_i \subseteq V | i \in I\}$ is a collection of subsets of V (called *bags*), and a tree $T = (I, F)$ with a node set I such that

1. $V = \cup_{i \in I} X_i$
2. For every edge $(u, v) \in E$ there is some bag $X_i \in X$ such that $u, v \in X_i$
3. For all $u \in V$, the nodes $\{i \in I | u \in X_i\}$ form a connected subtree of T

The *width* of a tree decomposition $(\{X_i | i \in I\}, T)$ is $\max_{i \in I} |X_i| - 1$. The *treewidth* of G is the minimum width over all tree decompositions of G.

Lemma 8. *Let $w \geq 1$ be a constant. Given a graph $G = (V, E)$ with $n > w + 1$ vertices and treewidth at most w, we can find in linear time a partition of V into three subsets A, B, and P such that*

1. *no edge joins a vertex in A with a vertex in B,*
2. *A and B each have between $\frac{n}{w+1} - w$ and $\frac{nw}{w+1}$ vertices,*
3. *P contains no more than w vertices, and*
4. *adding edges between the vertices of P does not change the treewidth.*

If we apply Lemma 8 to G instead of the separator theorem by Lipton and Tarjan we get the subgraph G_A of G induced by $A \cup P$ and the subgraph G_B of G induced by $B \cup P$. Since the number of portals is at most w, we may compute $\delta_G(G_A, G_B)$ in $O(wn \log^2 n) = O(n \log^2 n)$ time.

To recursively compute $\delta_G(G_A)$, we define graph G' as the graph obtained by removing all edges of G not belonging to G_A and adding an edge between each pair of portals of G. The cost of each such edge is set to the distance in G between the corresponding pair of portals. Note that the number of edges added is constant.

When we regard G' as a point set, we disregard edges added between portals. Observe that for each pair of points p and q in G', $d_{G'}(p, q) = d_G(p, q)$.

Since G' has treewidth at most w by Lemma 8, an inductive argument shows that $\delta_G(G_A) = \delta_{G'}(G_A)$ can be computed in $O(n \log^3 n)$ expected time. A similar argument shows that $\delta_G(G_B)$ can be computed in $O(n \log^3 n)$ expected time. This proves Theorem 6.

Computing the Stretch Factor of Paths, Trees, and Cycles in Weighted Fixed Orientation Metrics

Christian Wulff-Nilsen*

Abstract

Let G be a connected graph with n vertices embedded in a metric space with metric δ. The stretch factor of G is the maximum over all pairs of distinct vertices $u, v \in G$ of the ratio $\delta_G(u, v)/\delta(u, v)$, where $\delta_G(u, v)$ is the metric distance in G between u and v. We consider the plane equipped with a weighted fixed orientation metric, i.e. a metric that measures the distance between a pair of points as the length of a shortest path between them using only a given set of $\sigma \geq 2$ weighted fixed orientations. We show how to compute the stretch factor of G in $O(\sigma n \log^2 n)$ time when G is a path and in $O(\sigma n \log^3 n)$ time when G is a tree or a cycle. For the L_1-metric, we generalize the algorithms to d-dimensional space and show that the stretch factor can be computed in $O(n \log^d n)$ time when G is a path and in $O(n \log^{d+1} n)$ time when G is a tree or a cycle. All algorithms have $O(n)$ space requirement. Time and space bounds are worst-case bounds.

1 Introduction

Designing modern microchips is a complicated process involving several steps. One of these steps, the so called routing step, deals with the problem of finding a layout of wires on the chip interconnecting a given set of pins.

Many factors need to be taken into consideration when finding such a layout. An important measure is the total wire length. Minimizing this will help reduce heat generation, space on the chip, and signal delay. For this reason, Steiner minimal trees play an important role in VLSI design.

Due to manufacturing limitations, wires are typically restricted to having a finite set of fixed orientations. This has lead to an interest in the so called fixed orientation metrics, initially considered by Widmayer et al. [21], where the distance between a pair of points is the length of a shortest path between them using a fixed set of orientations.

Routing is typically performed in several layers with only one orientation allowed in each layer. Some layers may be more easily congested than others and thus, some orientations of wires may be less desirable than others [22]. For this reason, fixed orientatation metrics with a weight associated with each orientation have been considered.

Steiner minimal trees in the plane equipped with the rectilinear metric and more recently in general (weighted) fixed orientation metrics have received a great deal of attention [11, 13, 14, 7, 6, 18, 4, 5, 19]. A disadvantage of using Steiner minimal trees is that wire distance between some pairs of pins may be very large compared to the shortest possible distance. As a consequence, signal delay will be high between such pairs.

To obtain networks with small detours between any pair of points, spanners have been considered. For $t \geq 1$, a t-spanner for a set of points is a network interconnecting the points such that the distance in the network between any pair of the given points is at most t times longer than the shortest possible distance between them. The smallest t for which the network is a t-spanner is called the stretch factor of the network. Computing networks with small stretch factors is an active area of research. For more on spanners, see e.g. [17, 8, 20].

An interesting dual problem is the following: given a network interconnecting a set of n points in the plane, what is the stretch factor of this network?

Surprisingly, the fastest known algorithm for computing the stretch factor of a Euclidean network is a naive one that computes all-pairs shortest paths. If the network is planar, all-pairs shortest paths can be computed in $O(n^2)$ time [9], giving a quadratic time algorithm for computing the stretch factor of the network.

For simpler types of graphs, faster algorithms exist. For instance, it has been shown that the stretch factor of a path in the Euclidean plane can be found in $O(n \log n)$ *expected* time and that the stretch factor of trees and cycles can be found in $O(n \log^2 n)$ *expected* time [1, 15]. Using parametric search gives (rather complicated) $O(n\text{polylog } n)$ worst-case time algorithms for these types of networks.

To our knowledge, the problem of efficiently computing the stretch factor of networks in weighted fixed orientation metrics has not received any attention. Since these metrics may be used to approximate other metrics and due to their applications in VLSI design mentioned above, we believe this problem to be an important one.

In this paper, we give an $O(\sigma n \log^2 n)$ worst-case

*Department of Computer Science, University of Copenhagen, koolooz@diku.dk

time algorithm for computing the stretch factor of an n-vertex path embedded in the plane with a weighted fixed orientation metric defined by $\sigma \geq 2$ vectors. For the L_1-metric, we generalize the algorithm to $d \geq 3$ dimensions. Here, the running time is $O(n \log^d n)$. At the cost of an extra $\log n$-factor in running time, we show how to compute the stretch factor of trees and cycles. All algorithms have $O(n)$ space requirement. Compared to the complicated worst-case time algorithms for the Euclidean metric, our algorithms are relatively simple and should be easy to implement.

The organization of the paper is as follows. In Section 2, we make basic definitions and observations and introduce some notation. In Section 3, we consider the problem of computing the stretch factor of paths in the plane equipped with the L_1-metric. We give a new way of expressing the stretch factor which enables us to develop an efficient algorithm for this problem. Using simple linear transformations, we generalize the algorithm to arbitrary weighted fixed orientation metrics in Section 4 and in Section 5, we generalize it to higher dimensions. Using ideas of [15], we show how to efficiently compute the stretch factor of trees in Section 6. In Section 7, we present an algorithm that computes the stretch factor of cycles. In Section 8, we show how to modify the algorithms to compute a pair of vertices achieving the stretch factor of the path, tree, or cycle. Finally, we make some concluding remarks in Section 9.

2 Basic Definitions and Observations

Let G be a graph embedded in a metric space with metric d. For two vertices u and v in G, we define $d^G(u,v)$ as the d-length of a shortest path P between u and v in G, i.e.

$$d^G(u,v) = \sum_{(x,y)\in E_P} d(x,y),$$

where E_P is the set of edges of P. If u and v belong to distinct connected components of G then we define $d^G(u,v) = \infty$.

For distinct vertices u and v in G, the *detour* $\delta_G(u,v)$ between u and v in G is defined as $d^G(u,v)/d(u,v)$. The *stretch factor* δ_G of G is the maximum detour over all pairs of distinct vertices of G, i.e.

$$\delta_G = \max_{u,v\in G, u\neq v} \delta_G(u,v).$$

Given two subgraphs G_1 and G_2 of G, we define

$$\delta_G(G_1,G_2) = \max_{u\in G_1, v\in G_2, u\neq v} \delta_G(u,v).$$

Points and vectors in $\mathbb{R}^d$, $d \geq 2$, will be written in boldface. Unless otherwise stated, subsets of $\mathbb{R}^d$ that we consider are assumed to be closed.

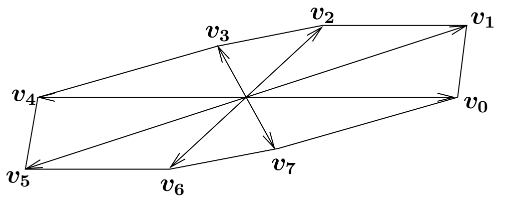


Figure 1: A unit circle in a weighted fixed orientation metric with $\sigma = 4$.

In all the following, let $\mathcal{V}$ denote a finite set of $\sigma \geq 2$ vectors $\boldsymbol{v_0}, \ldots, \boldsymbol{v_{\sigma-1}}$ all belonging to the upper half-plane. The *weighted fixed orientation metric* $d_\mathcal{V}$ is defined as follows. Letting $\boldsymbol{v_{\sigma+i}} = -\boldsymbol{v_i}$ for $i = 0, \ldots, \sigma-1$, the unit circle in $d_\mathcal{V}$ is the boundary of the convex hull of $\boldsymbol{v_0}, \ldots, \boldsymbol{v_{2\sigma-1}}$, see Figure 1.

We assume that all vectors, regarded as points, are on this boundary since any vectors that are not can be discarded without changing the metric.

Furthermore, we assume that vectors are ordered counter-clockwise starting with $\boldsymbol{v_0}$ and that $\boldsymbol{v}_0$, extends horizontally to the right. The latter can always be achieved by rotating the plane.

Drawing lines through the 2σ vectors partitions the plane into 2σ wedge-shaped regions. For $i = 0, \ldots, 2\sigma-1$, the region W_i defined by vectors $\boldsymbol{v_i}$ and $\boldsymbol{v_{(i+1) \bmod 2\sigma}}$ is called the *ith $\mathcal{V}$-cone.* For a point $\boldsymbol{p}$, we refer to $\boldsymbol{p}+W_i$ as the *ith $\mathcal{V}$-cone of $\boldsymbol{p}$.*

Let $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^2$. It can be shown that if $\boldsymbol{q}$ belongs to the ith $\mathcal{V}$-cone of $\boldsymbol{p}$ then a shortest path from $\boldsymbol{p}$ to $\boldsymbol{q}$ in the metric $d_\mathcal{V}$ consists of line segments each of which is parallel to either $\boldsymbol{v_i}$ or $\boldsymbol{v_{(i+1) \bmod 2\sigma}}$.

For $\boldsymbol{p} \in \mathbb{R}^2$, $r \geq 0$, we define $B_\mathcal{V}(\boldsymbol{p}, r)$ as the closed disc in $d_\mathcal{V}$ with center $\boldsymbol{p}$ and radius r, that is,

$$B_\mathcal{V}(\boldsymbol{p}, r) = \{\boldsymbol{q} \in \mathbb{R}^2 | d_\mathcal{V}(\boldsymbol{p},\boldsymbol{q}) \leq r\}.$$

The L_1-metric in the plane is a special type of fixed orientation metric, defined by vectors $(1,0)$ and $(0,1)$. More generally, in $\mathbb{R}^d$, the L_1-distance between two points $\boldsymbol{p} = (p_1, \ldots, p_d)$ and $\boldsymbol{q} = (q_1, \ldots, q_d)$ is

$$L_1(\boldsymbol{p},\boldsymbol{q}) = \sum_{c=1}^{d} |q_c - p_c|.$$

For $\boldsymbol{p} \in \mathbb{R}^d$ and $r \geq 0$, we let $B_1(\boldsymbol{p}, r)$ denote the closed L_1-ball in $\mathbb{R}^d$ with center $\boldsymbol{p}$ and radius r, i.e.

$$B_1(\boldsymbol{p}, r) = \{\boldsymbol{q} \in \mathbb{R}^d | L_1(\boldsymbol{p},\boldsymbol{q}) \leq r\}.$$

3 Stretch Factor of Paths in the L_1-Plane

In this section, we show how to compute the stretch factor of an n-vertex path embedded in metric space $(\mathbb{R}^2, L_1)$ in $O(n \log^2 n)$ time and $O(n)$ space.

In the following, let $P = \boldsymbol{p_1} \to \boldsymbol{p_2} \to \cdots \to \boldsymbol{p_n}$ be an n-vertex path in the plane. We will make the simplifying assumption that all vertices of P are distinct. For if they were not, we would have $\delta_P = \infty$ and checking whether all vertices are distinct can be done in $O(n \log n)$ time.

In all the following, let $N = \{1, \ldots, n\}$. For $i \in N$ and $\delta > 0$, let $B_i(\delta)$ denote the L_1-disc $B_1(\boldsymbol{p_i}, r_i(\delta))$, where radius $r_i(\delta) = L_1^P(\boldsymbol{p_i}, \boldsymbol{p_n})/\delta$. The following lemma relates these discs to the stretch factor of P.

Lemma 1 *With the above definitions, the stretch factor of P is $\delta_P = \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall i, j \in N, i \neq j\}$.*

Proof. Let $\delta > 0$. For any $i, j \in N$,

$$\frac{L_1^P(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} = \frac{|L_1^P(\boldsymbol{p_i}, \boldsymbol{p_n}) - L_1^P(\boldsymbol{p_j}, \boldsymbol{p_n})|}{\delta} = |r_i(\delta) - r_j(\delta)|.$$

Hence,

$$\begin{aligned}\delta_P < \delta &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > |r_i(\delta) - r_j(\delta)| \\ &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > r_i(\delta) - r_j(\delta) \\ &\Leftrightarrow \forall i, j \in N, i \neq j : L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta) > r_i(\delta) \\ &\Leftrightarrow \forall i, j \in N, i \neq j : B_j(\delta) \not\subseteq B_i(\delta).\end{aligned}$$

□

The idea of our algorithm is to see how much the size of the above defined L_1-discs can be increased before at least one of them includes another L_1-disc. By Lemma 1, this will then give us the stretch factor of path P.

For each $i \in N$ and for $w = 1, 2, 3, 4$, define $P_w(i)$ as the set of vertices of $P \setminus \{\boldsymbol{p_i}\}$ belonging to the wth quadrant of $\boldsymbol{p_i}$. Lemma 1 gives

$$\delta_P = \max_{w=1,2,3,4} \max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\}.$$

Hence, δ_P is the maximum of four δ-values, one for each value of w. In the following, let us therefore restrict our attention to $w = 1$ and on computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\} \qquad (1)$$

(the other quadrants are handled in a similar way).

The following lemma gives a useful way of determining whether $B_j(\delta)$ is contained in $B_i(\delta)$ when $\boldsymbol{p_j}$ belongs to the first quadrant of $\boldsymbol{p_i}$.

Lemma 2 *Let $\boldsymbol{p_i}$ be a given vertex and let $\boldsymbol{p_j} \in P_1(i)$. For $\delta > 0$, define $\boldsymbol{r_i}(\delta)$ and $\boldsymbol{r_j}(\delta)$ as the rightmost points in $B_i(\delta)$ and $B_j(\delta)$, respectively (see Figure 2). Then*

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow \boldsymbol{r_i}(\delta) \cdot (1,1) \geq \boldsymbol{r_j}(\delta) \cdot (1,1).$$

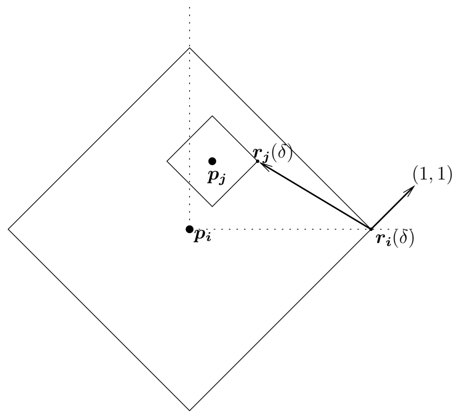


Figure 2: The situation in Lemma 2.

Proof. The point $\boldsymbol{r_j}(\delta)$ is to the right of $\boldsymbol{p_j}$ and belongs to the first quadrant of $\boldsymbol{p_i}$, implying that $L_1(\boldsymbol{p_i}, \boldsymbol{r_j}(\delta)) = L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta)$. Since $B_j(\delta) \subseteq B_i(\delta)$ if and only if $L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) + r_j(\delta) \leq r_i(\delta)$, we have

$$\begin{aligned}B_j(\delta) \subseteq B_i(\delta) &\Leftrightarrow L_1(\boldsymbol{p_i}, \boldsymbol{r_j}(\delta)) \leq r_i(\delta) \\ &\Leftrightarrow \boldsymbol{r_j}(\delta) \in B_i(\delta) \\ &\Leftrightarrow (\boldsymbol{r_j}(\delta) - \boldsymbol{r_i}(\delta)) \cdot (1,1) \leq 0,\end{aligned}$$

since vector $(1,1)$ is normal to the part of the boundary of $B_i(\delta)$ in the first quadrant of $\boldsymbol{p_i}$. □

Recall that, for any $i \in N$, $r_i(\delta) = L_1^P(\boldsymbol{p_i}, \boldsymbol{p_n})/\delta$. Hence, the dot product $\boldsymbol{r_i}(\delta) \cdot (1,1)$ of Lemma 2 is an affine function of $1/\delta$, i.e. on the form $a(1/\delta) + b$, where a and b are constants. Denote this function by f_i.

Associate with each $\boldsymbol{p_i}$ a *lower envelope function* l_i of $1/\delta$, defined by

$$l_i(1/\delta) = \min\{f_j(1/\delta) | \boldsymbol{p_j} \in P_1(i)\}.$$

Recall that our goal is to compute (1). The following lemma relates this value to the intersection between f_i and l_i.

Lemma 3 *There is at most one intersection point between f_i and l_i on interval $]0, \infty[$. If $1/\delta'$ is such a point then*

$$\delta' = \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\}.$$

If there is no intersection point then for any $\delta > 0$ and any $\boldsymbol{p_j} \in P_1(i)$, $B_j(\delta) \not\subseteq B_i(\delta)$.

Proof. Figure 3 illustrates the lemma.

For any point $\boldsymbol{p}$ in the first quadrant of $\boldsymbol{p_i}$, the value $\boldsymbol{p} \cdot (1,1)$ is minimized when $\boldsymbol{p} = \boldsymbol{p_i}$. It follows that $f_i(1/\delta) < l_i(1/\delta)$ for all sufficiently small $1/\delta$. Hence, since the graph of l_i on $]0, \infty[$ is a chain of line segments (and one halfline) whose slopes decrease as we move

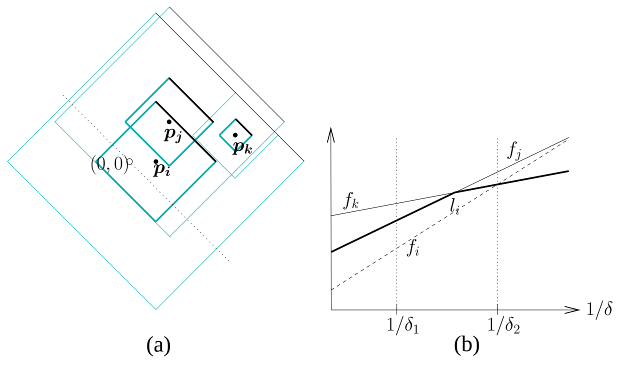


Figure 3: Illustration of Lemma 3. (a): L_1-discs $B_i(\delta)$, $B_j(\delta)$, and $B_k(\delta)$ for two values of δ: δ_1 (bold boundaries) and δ_2. (b): The corresponding functions f_i, f_j, and f_k. Lower envelope l_i is shown in bold. The distances in (a) between the dotted line and the black parts of L_1-discs correspond to values of functions f_i, f_j, and f_k at $1/\delta_1$ and $1/\delta_2$ in (b). Note that $B_k(\delta_2) \subseteq B_i(\delta_2)$. For all $1/\delta < 1/\delta_2$, $B_j(\delta) \nsubseteq B_i(\delta)$ and $B_k(\delta) \nsubseteq B_i(\delta)$.

from left to right, there is at most one intersection point $1/\delta'$ between f_i and l_i on interval $]0, \infty[$.

If intersection point $1/\delta'$ exists, the above shows that, on interval $]0, \infty[$, $f_i(1/\delta) < l_i(1/\delta')$ if $1/\delta < 1/\delta'$ and $f_i(1/\delta) > l_i(1/\delta')$ if $1/\delta > 1/\delta'$. And if no intersection point exists then f_i is below l_i on interval $]0, \infty[$.

Lemma 2 shows that $B_j(\delta) \subseteq B_i(\delta)$ if and only if $f_i(1/\delta) \geq f_j(1/\delta)$. Hence, $B_j(\delta) \nsubseteq B_i(\delta)$ for all $\boldsymbol{p_j}$ in the first quadrant $P_1(i)$ of $\boldsymbol{p_i}$ if and only if $f_i(1/\delta) < l_i(1/\delta)$. This shows the lemma. □

For each $i \in N$, let $\delta_i = 1/x_i$, where x_i is the intersection point between f_i and l_i. If no such point exists, set $\delta_i = 0$. Lemma 3 shows that (1) equals $\max_{i \in N} \delta_i$.

What remains therefore is the problem of computing the intersection (if any) between f_i and l_i for all i.

A naive algorithm for this problem computes, for each $i \in N$, lower envelope l_i in $O(n \log n)$ time (this is possible by Lemma 4 of Section 3.2) and then the intersection between f_i and l_i in $O(\log n)$ time. The total running time is $O(n^2 \log n)$.

A slightly faster algorithm computes, for each i, the intersection between f_i and f_j for each $\boldsymbol{p_j} \in P_1(i)$. The leftmost of these is then the intersection between f_i and l_i. This gives a total running time of $O(n^2)$.

Note that, for any $j \in N$, the lower envelope of f_j is f_j itself. Hence, the two algorithms above apply two extremes of the following strategy: for each $i \in N$, compute the leftmost of the intersections between f_i and lower envelopes associated with subsets of points in $P_1(i)$. The first algorithm considers, for each $i \in N$, only one subset (namely $P_1(i)$) whereas the second algorithm considers $|P_1(i)|$ subsets (each containing one element of $P_1(i)$).

In the next section, we present a faster algorithm which applies a strategy somewhere in between these two extremes.

3.1 The Algorithm

To simplify the description of the algorithm, we will leave out some of the details and return to them in Section 3.2 and Section 3.3, where we show how to obtain $O(n \log^2 n)$ running time and $O(n)$ space requirement.

The algorithm stores vertices of P in a balanced binary search tree $\mathcal{T}$ of height $\Theta(\log n)$ which is similar to a 1-dimensional range tree. Let V be the set of vertices of P. If V contains exactly one vertex, the root r of $\mathcal{T}$ is a leaf containing this vertex. Otherwise, r contains the median m of x-coordinates of vertices of V (in case of ties, order the vertices on the y-axis) and the subtree rooted at the left resp. right child of r is defined recursively for the set of vertices of V with x-coordinates less or equal to resp. greater than m.

Each node v of $\mathcal{T}$ corresponds to a subset S_v of vertices of P, namely those vertices stored at the leaves of the subtree of $\mathcal{T}$ rooted at v. We refer to these S_v-subsets as *canonical subsets*.

Note that each vertex $\boldsymbol{p_i}$ of P belongs to $\Theta(\log n)$ canonical subsets, namely those corresponding to vertices visited on the path from the root of $\mathcal{T}$ to the leaf containing $\boldsymbol{p_i}$.

In addition to a median, we associate with each node of $\mathcal{T}$ a lower envelope of line segments. This lower envelope is initially empty and will be dynamically updated during the course of the algorithm.

After having constructed $\mathcal{T}$, the algorithm makes a pass over the vertices of P in order of descending y-coordinate. In case of ties, vertices are visited from right to left.

The following invariant will be maintained throughout the course of the algorithm: *for each vertex v of $\mathcal{T}$, the lower envelope associated with v is the lower envelope of f_i-functions of vertices in S_v visited so far.*

When a vertex $\boldsymbol{p_i}$ of P is visited, the invariant is maintained by adding f_i to the $\Theta(\log n)$ lower envelopes associated with vertices on the path from the root of $\mathcal{T}$ to the leaf containing $\boldsymbol{p_i}$.

When the algorithm visits a vertex $\boldsymbol{p_i}$, it needs to find the intersection between f_i and l_i. As we saw earlier, explicitly computing l_i is too time-consuming.

Instead, we make use of our invariant which ensures that lower envelopes of visited vertices of all canonical subsets are given. The vertices in the first quadrant of $\boldsymbol{p_i}$ have all been visited and the set $P_1(i)$ of these vertices is therefore the union of visited vertices of the canonical subsets to the right of $\boldsymbol{p_i}$. So the intersection between f_i and l_i is the leftmost of the intersections between f_i and the lower envelopes associated with these canonical subsets, see Figure 4.

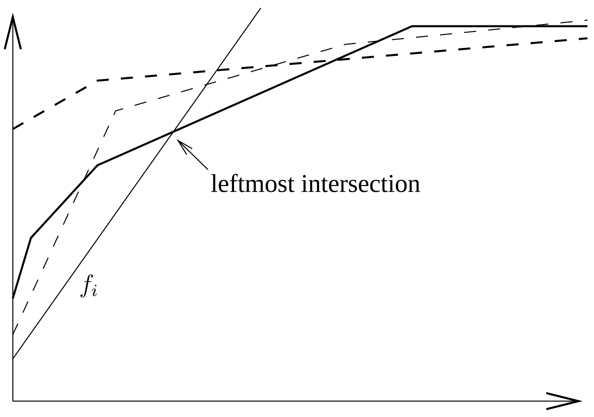


Figure 4: The intersection between f_i and l_i is the leftmost of the intersections between f_i and lower envelopes associated with canonical subsets to the right of $\boldsymbol{p_i}$.

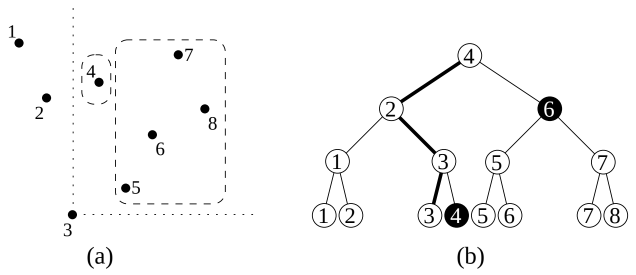


Figure 5: (a) Eight points shown with x-coordinates $1, \ldots, 8$. The set of points with x-coordinate greater than 3 is the union of two canonical subsets. (b) The two canonical subsets are found by picking right children on the path (shown in bold) from the root of the range tree to the leaf with median 3. Picked children are coloured black.

Since canonical subsets may overlap, not all canonical subsets to the right of $\boldsymbol{p_i}$ are needed. The idea is to pick a small number in order to minimize running time.

The algorithm picks canonical subsets (or more precisely, nodes of $\mathcal{T}$ corresponding to canonical subsets) as follows. Let v_i be the leaf of $\mathcal{T}$ associated with $\boldsymbol{p_i}$. For each vertex v on the path from the root r of $\mathcal{T}$ to v_i, the canonical subset associated with the right child of v is picked unless this child itself is on the path from r to v_i, see Figure 5.

It is easy to see that the visited vertices in the union of the picked canonical subsets are exactly the vertices of $P_1(i)$. Since the height of $\mathcal{T}$ is $\Theta(\log n)$, the number of picked canonical subsets is $O(\log n)$.

One fine point: if there are vertices of P above $\boldsymbol{p_i}$ and with the same x-coordinate as $\boldsymbol{p_i}$, they may not all belong to the picked canonical subsets even though they belong to $P_1(i)$. We may ignore these however, since they will be picked when second quadrants are handled.

Intersections between f_i and each of the lower envelopes of the picked canonical subsets are then computed and the leftmost of these is picked as the intersection between f_i and l_i.

From these intersections, the value (1) is obtained. This is repeated for the other three quadrants, giving the stretch factor of P.

3.2 Running Time

In the description of the algorithm above, we left out some details. We now focus on them in order to analyze the running time of the algorithm.

It is easy to see that tree $\mathcal{T}$ can be constructed top-down in $O(n \log n)$ time. In the y-descending pass over the vertices of P, maintaining our invariant requires adding each f_i-function to $O(\log n)$ lower envelopes. Finding these lower envelopes takes $O(\log n)$ time by a traversal from the root to a leaf of $\mathcal{T}$ using the medians at vertices to guide the search. The following lemma shows that each insertion of a f_i-function into a lower envelopes takes $O(\log n)$ amortized time.

Lemma 4 *Let $l_1, \ldots, l_k$ be k lines in the plane with positive slope and let L be the lower envelope of these lines on interval $]0, \infty[$. Constructing L incrementally can be done in $O(\log k)$ amortized time per line. Furthermore, L consists of at most $k-1$ line segments and exactly one halfline.*

Proof. We may assume that lines are added to L in the order $l_1, \ldots, l_k$. For $i = 1, \ldots, k$, let L_i be the lower envelope of $l_1, \ldots, l_i$.

For $i > 1$, suppose that L_{i-1} has been computed and that the set Q_{i-1} of points on the graph of L_{i-1} where line segments meet are ordered from left to right in a red-black tree. Then computing the at most two intersections between l_i and L_{i-1} can be done in $O(\log i)$ time using two binary searches in the tree. When intersections have been found (if any), L_i is obtained from L_{i-1} in $O(q_{i-1} \log q_{i-1})$ time, where q_{i-1} is the number of points of Q_{i-1} that need to be removed to obtain L_i, i.e. the number of points of Q_{i-1} above l_i.

Since a point is removed at most once and each new line increases the number of points defining the lower envelope by at most two, it follows that the total time spent on constructing the lower envelopes in the order $L_1, \ldots, L_k$ is $O(k \log k)$.

Since slopes of the line segments (and the halfline) defining the graph of L decrease from left to right, each of the k lines contribute with at most one line segment to the graph of L. Exactly one of the lines contribute with a halfline to the graph of L. Thus, the graph of L consists of at most $k-1$ line segments and exactly one halfline. $\square$

Next, we need to analyze the time it takes to compute the intersection between f_i and l_i for each i. This involves picking $O(\log n)$ canonical subsets and computing the intersection between f_i and the lower envelopes associated with these subsets.

Clearly, the time it takes to find the canonical subsets is bounded by the height of $\mathcal{T}$ which is $\Theta(\log n)$. Since each lower envelope l is a monotonically increasing function and its graph consists of line segments (and one halfline), computing the intersection between f_i and l can be done in $O(\log n)$ time by using a data structure like a red-black tree to represent the ordered list of points of l where line segments meet.

It follows that our algorithm has $O(n \log^2 n)$ running time.

3.3 Improving Space Requirement

By Lemma 4, space requirement of our algorithm is $\Theta(n \log n)$ since this is the amount of space required to store all lower envelopes. We now show how to improve space requirement to linear without affecting running time.

We modify the algorithm so that, instead of making only one y-descending pass over the vertices of P, it makes $h(\mathcal{T})$ passes (for each of the four quadrants), where $h(\mathcal{T})$ is the height of $\mathcal{T}$.

In the kth pass, only lower envelopes at level k-nodes of $\mathcal{T}$ are updated; all other nodes of $\mathcal{T}$ contain empty lower envelopes. And only intersections between f_i-functions and lower envelopes at level k of $\mathcal{T}$ are computed.

The modified algorithm is correct since it computes exactly the same intersections as the old algorithm.

In each y-descending pass, the time spent on a vertex $\boldsymbol{p_i}$ of P is bounded by the time to add f_i to a lower envelope, and the time to compute the intersection between f_i and a lower envelope (if any). Hence, the total time spent in each pass is bounded by $O(n \log n)$. Since there are $h(\mathcal{T}) = \Theta(\log n)$ passes, the total running time is $O(n \log^2 n)$.

The modified algorithm has $O(n)$ space requirement since $\mathcal{T}$ has $O(n)$ nodes and since storing lower envelopes at one level of $\mathcal{T}$ requires $O(n)$ space by Lemma 4. This gives the first main result of the paper.

Theorem 5 *The stretch factor of an n-vertex path in $(\mathbb{R}^2, L_1)$ can be computed in $O(n \log^2 n)$ time and $O(n)$ space.*

4 Weighted Fixed Orientation Metrics

Recall that $d_{\mathcal{V}}$ denotes a weighted fixed orientation metric defined by a set $\mathcal{V}$ of $\sigma \geq 2$ vectors in the plane.

In this section, we generalize the algorithm of the preceding section to $d_{\mathcal{V}}$. The idea is simple: we apply a certain linear transformation to the vertices of P so that ith $\mathcal{V}$-cones are mapped to first quadrants and then use the algorithm of Section 3. This is done for all $\mathcal{V}$-cones, giving an algorithm with $O(\sigma n \log^2 n)$ time and $O(n)$ space requirement.

First, we observe that Lemma 1 also applies to the weighted fixed orientation metrics: simply define $B_i(\delta)$ as $B_{\mathcal{V}}(\boldsymbol{p_i}, r_i(\delta))$, where $r_i(\delta) = d_{\mathcal{V}}^P(\boldsymbol{p_i}, \boldsymbol{p_n})/\delta$, and replace L_1 by $d_{\mathcal{V}}$ in the proof. This gives us

$$\delta_P = \max_{1 \leq w \leq 2\sigma} \max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\},$$

where $P_w(i)$ is the set of vertices of $P \setminus \{\boldsymbol{p_i}\}$ belonging to the wth $\mathcal{V}$-cone of $\boldsymbol{p_i}$. We restrict our attention to computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\} \qquad (2)$$

(the other $\mathcal{V}$-cones are handled in a similar way).

Let $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ be the first and second vector of $\mathcal{V}$ respectively. It is easy to find the linear transformation T of the plane that maps $\boldsymbol{v_0}$ to $(1,0)$ and $\boldsymbol{v_1}$ to $(0,1)$. This allows us to generalize Lemma 2.

Lemma 6 *Let $\boldsymbol{p_i}$ be a given vertex and let $\boldsymbol{p_j} \in P_1(i)$. For $\delta > 0$, let $\boldsymbol{r_i}(\delta)$ and $\boldsymbol{r_j}(\delta)$ be the rightmost point in $B_i(\delta)$ and $B_j(\delta)$ respectively. Then*

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow T(\boldsymbol{r_i}(\delta)) \cdot (1,1) \geq T(\boldsymbol{r_j}(\delta)) \cdot (1,1),$$

where T is defined as above.

Proof. Let $\boldsymbol{v_0}$ and $\boldsymbol{v_1}$ be defined as above. Applying the ideas of the proof of Lemma 2, it follows easily that $B_j(\delta) \subseteq B_i(\delta)$ if and only if path $\boldsymbol{r_j}(\delta) \to \boldsymbol{r_i}(\delta) \to r_i(\delta)\boldsymbol{v_1}$ does not make a right turn at $\boldsymbol{r_i}(\delta)$.

Since T maps the triangle with corners $(0,0)$, $\boldsymbol{v_0}$, and $\boldsymbol{v_1}$ to the triangle with corners $(0,0)$, $(1,0)$, and $(0,1)$, linearity of T implies that $B_j(\delta) \subseteq B_i(\delta)$ if and only if path $T(\boldsymbol{r_j}(\delta)) \to T(\boldsymbol{r_i}(\delta)) \to T(r_i(\delta)\boldsymbol{v_1})$ does not make a right turn at $T(\boldsymbol{r_i}(\delta))$.

Since the vector from $T(\boldsymbol{r_i}(\delta))$ to $T(r_i(\delta)\boldsymbol{v_1})$ and vector $(-1,1)$ have the same orientation,

$$\begin{aligned} B_j(\delta) \subseteq B_i(\delta) &\Leftrightarrow (T(\boldsymbol{r_j}(\delta)) - T(\boldsymbol{r_i}(\delta))) \cdot \widehat{(-1,1)} \geq 0 \\ &\Leftrightarrow (T(\boldsymbol{r_j}(\delta)) - T(\boldsymbol{r_i}(\delta))) \cdot (1,1) \leq 0. \end{aligned}$$

□

By defining affine functions f_i by $f_i(1/\delta) = T(\boldsymbol{r_i}(\delta)) \cdot (1,1)$ and l_i by

$$l_i(1/\delta) = \min\{f_j(1/\delta) | \boldsymbol{p_j} \in P_1(i)\}$$

for $i \in N$, Lemma 6 and the results of Section 3 show that the value (2) may be computed in $O(n \log^2 n)$ time using $O(n)$ space. Since there are 2σ $\mathcal{V}$-cones to consider, we thus obtain the following generalization of Theorem 5.

Theorem 7 *Let $\mathcal{V}$ be a set of $\sigma \geq 2$ vectors defining a weighted fixed orientation metric $d_{\mathcal{V}}$ on $\mathbb{R}^2$. Then the stretch factor of an n-vertex path in $(\mathbb{R}^2, d_{\mathcal{V}})$ can be computed in $O(\sigma n \log^2 n)$ time and $O(n)$ space.*

5 Higher Dimensions

In this section, we generalize the algorithm of Section 3 to higher dimensions. In metric space $(\mathbb{R}^d, L_1)$, $d \geq 3$, we will show how to compute the stretch factor of an n-vertex path in $O(n \log^d n)$ time using $O(n)$ space.

In the following, assume that path $P = \boldsymbol{p_1} \rightarrow \boldsymbol{p_2} \rightarrow \cdots \rightarrow \boldsymbol{p_n}$ is embedded in $(\mathbb{R}^d, L_1)$, where $d \geq 3$. For $i \in N$, let $B_i(\delta)$ and $r_i(\delta)$ be defined as in Section 3.

First, we observe that Lemma 1 also holds in d dimensions. This gives us

$$\delta_P = \max_{1 \leq w \leq 2^d} \max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_w(i)\},$$

where $P_w(i)$ is the set of vertices of $P \backslash \{\boldsymbol{p_i}\}$ belonging to the wth orthant[1] of $\boldsymbol{p_i}$ for some ordering of the orthants. We assume that

$$O_1(i) = \{\boldsymbol{p} \in \mathbb{R}^d | \boldsymbol{p}[c] \geq \boldsymbol{p_i}[c] \forall c\}$$

is the first orthant of $\boldsymbol{p_i}$ and we restrict our attention to computing

$$\max_{i \in N} \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall \boldsymbol{p_j} \in P_1(i)\} \qquad (3)$$

(the other orthants are handled in a similar way).

The following lemma generalizes Lemma 2 to higher dimensions.

Lemma 8 *Let $\boldsymbol{p_i}$ be a given vertex and let $\boldsymbol{p_j} \in P_1(i)$. For $\delta > 0$, let*

$$\boldsymbol{r_i}(\delta) = (\boldsymbol{p_i}[1] + r_i(\delta), \boldsymbol{p_i}[2], \boldsymbol{p_i}[3], \ldots, \boldsymbol{p_i}[d])$$
$$\boldsymbol{r_j}(\delta) = (\boldsymbol{p_j}[1] + r_j(\delta), \boldsymbol{p_j}[2], \boldsymbol{p_j}[3], \ldots, \boldsymbol{p_j}[d])$$

and let $\boldsymbol{e}$ be the d-dimensional vector with d ones. Then

$$B_j(\delta) \subseteq B_i(\delta) \Leftrightarrow \boldsymbol{r_i}(\delta) \cdot \boldsymbol{e} \geq \boldsymbol{r_j}(\delta) \cdot \boldsymbol{e}.$$

Proof. Similar to the proof of Lemma 2, we have

$$\begin{aligned} B_j(\delta) \subseteq B_i(\delta) &\Leftrightarrow L_1(\boldsymbol{p_i}, \boldsymbol{r_j}(\delta)) \leq r_i(\delta) \\ &\Leftrightarrow \boldsymbol{r_j}(\delta) \in B_i(\delta). \end{aligned}$$

Let $B_i^1(\delta)$ be the part of the boundary of $B_i(\delta)$ belonging to $O_1(i)$. Then $B_i^1(\delta)$ contains the points $\boldsymbol{p_i} + r_i(\delta)\boldsymbol{e_j}$, $j = 1, \ldots, d$, where $\boldsymbol{e_j}$ is the jth unit vector. Since $\boldsymbol{r_i}(\delta) \in B_i(\delta)$, the hyperplane H containing $B_i^1(\delta)$ is defined by $H = \{\boldsymbol{p} \in \mathbb{R}^d | (\boldsymbol{p} - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} = 0\}$.

[1]An orthant is the higher dimensional equivalent of a quadrant. A d-dimensional coordinate system has 2^d orthants.

Since $\boldsymbol{p_i}$ is an interior point of $B_i(\delta)$ and since $(\boldsymbol{p_i} - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} < 0$, it follows that, for any point $\boldsymbol{p} \in P_1(i)$, $(\boldsymbol{p} - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} \leq 0$ if and only if $\boldsymbol{p} \in B_i(\delta)$. Hence,

$$\begin{aligned} B_j(\delta) \subseteq B_i(\delta) &\Leftrightarrow \boldsymbol{r_j}(\delta) \in B_i(\delta) \\ &\Leftrightarrow (\boldsymbol{r_j}(\delta) - \boldsymbol{r_i}(\delta)) \cdot \boldsymbol{e} \leq 0. \end{aligned}$$

□

For each $i \in N$, we define affine function f_i by $f_i(1/\delta) = \boldsymbol{r_i}(\delta) \cdot \boldsymbol{e}$, where $\boldsymbol{r_i}$ and $\boldsymbol{e}$ are defined as in Lemma 8 and we define lower envelope function l_i as in Section 3.

Since Lemma 3 holds with the f_i- and l_i-functions defined as above, it follows that the problem we are facing is to compute the intersection between f_i and l_i (if any) for all i. We deal with this problem in the next subsection where we generalize the algorithm of Section 3.1 to d dimensions.

5.1 The Algorithm

In Section 3.1, we used a 1-dimensional range tree. To compute the stretch factor of path P in d dimensions, we now consider a $(d-1)$-dimensional range tree.

First, a binary search tree $\mathcal{T}$ is constructed on the first coordinate of the vertices of P as described in Section 3.1. Each node v of $\mathcal{T}$ is associated with a $(d-2)$-dimensional range tree for the vertices in canonical subset S_v restricted to their last $d-1$ coordinates. This construction is repeated recursively for the $(d-2)$-dimensional range tree. The recursion stops when we reach a 1-dimensional range tree for coordinate $d-1$.

We associate lower envelopes only with nodes of the 1-dimensional range trees. Note that range trees are not defined for coordinate d. In this way, coordinates $d-1$ and d correspond to coordinates x and y in Section 3.1 respectively.

The algorithm then visits vertices of P in order of descending d-coordinate. In case of ties, vertices are visited from right to left on axis $d-1$.

Let $\boldsymbol{p_i}$ be the vertex currently being visited. The algorithm needs to update all lower envelopes corresponding to canonical subsets in 1-dimensional range trees that contain $\boldsymbol{p_i}$.

This is done as follows. All nodes of the $(d-1)$-dimensional range tree whose canonical subsets contain $\boldsymbol{p_i}$ are visited. Each of their associated $(d-2)$-dimensional range trees are visited recursively. When the nodes of a 1-dimensional range tree are visited, the function f_i corresponding to $\boldsymbol{p_i}$ is inserted into their associated lower envelopes.

For vertex $\boldsymbol{p_i}$, the algorithm also needs to compute intersections between f_i and lower envelopes corresponding to canonical subsets which are to the right of $\boldsymbol{p_i}$ on axes 1 to $d-1$.

This is done as follows. Let r be the root and let v_i be the leaf containing $\boldsymbol{p_i}$ in the $(d-1)$-dimensional range tree. For each node v on the path from r to v_i the $(d-2)$-dimensional range tree associated with the right child of v is visited recursively unless this child itself is on the path from r to v_i. When reaching a 1-dimensional range tree, intersections between f_i and lower envelopes are found as described in Section 3.1.

The correctness of the above algorithm follows by generalizing the arguments of Section 3.1.

5.2 Running Time

We will now show that the algorithm described above has $O(n \log^d n)$ running time. In our analysis, we assume that dimension $d \geq 3$ is a constant.

As shown in [3], a $(d-1)$-dimensional range tree can be constructed in $O(n \log^{d-2} n)$ time. For each vertex $\boldsymbol{p_i}$ of P, finding the lower envelopes in which f_i is to be inserted takes $O(\log^{d-1} n)$ time. To see this, note that the algorithm recurses on $O(\log n)$ $(d-2)$-dimensional range trees associated with nodes of the $(d-1)$-dimensional range tree. For each of these range trees, the algorithm recurses on $O(\log n)$ $(d-3)$-dimensional range trees and so on. Thus, the total time to visit lower envelopes in which f_i is to be inserted is $O(\log^{d-1} n)$.

Since it takes $O(\log n)$ time to insert f_i into a lower envelope, the total time spent on inserting f_i-functions into lower envelopes is $O(n \log^d n)$.

A similar argument shows that it takes $O(n \log^d n)$ time to compute intersections between f_i-functions and lower envelopes. Hence, the total running time of the algorithm is $O(n \log^d n)$.

5.3 Improving Space Requirement

The above algorithm does not have linear space requirement. For instance, constructing the $(d-1)$-dimensional range tree using the algorithm of [3] requires $O(n \log^{d-2} n)$ space. By generalizing the idea of Section 3.3, we will modify our algorithm so that space requirement is improved to $O(n)$ without affecting running time.

The algorithm above makes one pass over the vertices of P in order of descending d-coordinate. We modify it so that it makes h_{d-1}^{d-1} passes, where h_{d-1} is the height of the $(d-1)$-dimensional range tree.

We enumerate the passes using a $(d-1)$-dimensional vector $\boldsymbol{C}$. Each entry of $\boldsymbol{C}$ is a number between 1 and h_{d-1}. Note that $h_{d-1} = \Theta(\log n)$ and that h_{d-1} is an upper bound on the height of all other range trees (since they all correspond to smaller sets of vertices of P).

Vector $\boldsymbol{C}$ can attain $\Theta(\log^{d-1} n)$ values and each of these values determine which parts of the range trees we are interested in in the current pass. More specifically, $\boldsymbol{C}[i] = k$ indicates that we are interested only in level k of all $(d-i)$-dimensional range trees, $i = 1, \ldots, d-1$. If some $(d-i)$-dimensional range tree has height less than $\boldsymbol{C}[i]$, it means that we are not interested in any levels of that tree in the current pass.

Consider a given pass of the vertices of P. We construct the $(d-1)$-dimensional range tree as before except that we only associate $(d-2)$-dimensional range trees with nodes at depth $\boldsymbol{C}[1]$. For each of these $(d-2)$-dimensional range trees, we only associate $(d-3)$-dimensional range trees with nodes at depth $\boldsymbol{C}[2]$ and so on. We refer to these range trees as *restricted range trees*.

Since canonical subsets associated with nodes at the same level of a range tree are disjoint, it follows easily that restricted range trees can be constructed in $O(n \log n)$ time (since d is assumed to be a constant). Thus, the total time spent on constructing restricted range trees over all passes is $O(n \log^d n)$.

In each pass, we only update those lower envelopes allowed by $\boldsymbol{C}$. The time spent on this for a given vertex $\boldsymbol{p_i}$ of P and a given pass is $O(\log n)$ since there is at most one lower envelope in which f_i is to be inserted in the current pass and it takes $O(\log n)$ time to find it and update it.

Similarly, computing the intersection between lower envelopes and a given vertex of P in a given pass takes $O(\log n)$ time. Thus, the total time spent in each pass is $O(n \log n)$. Since there are $\Theta(\log^{d-1} n)$ passes, the total running time of the modified algorithm is $O(n \log^d n)$, that is, the same as the original algorithm.

As for space requirement, we observe that the space required to represent restricted range trees is $O(n)$ and the space used for storing lower envelopes in a given pass is $O(n)$ since for a given vertex $\boldsymbol{p_i}$ of P, f_i is stored in at most one lower envelope in the current pass. This gives us the following generalization of Theorem 5.

Theorem 9 *The stretch factor of an n-vertex path in $(\mathbb{R}^d, L_1)$ can be computed in $O(n \log^d n)$ time and $O(n)$ space.*

6 Stretch Factor of Trees

In this section, we generalize our algorithms for paths to trees. We will show that the stretch factor of a tree with n vertices can be computed in $O(\sigma n \log^3 n)$ time for the weighted fixed orientation metrics and in $O(n \log^{d+1} n)$ time in the space $(\mathbb{R}^d, L_1)$ using $O(n)$ space.

Let us focus on space $(\mathbb{R}^d, L_1)$. The weighted fixed orientation metrics are handled in a similar way.

Let T be a tree with n vertices $\boldsymbol{p_1}, \ldots, \boldsymbol{p_n}$ embedded in $(\mathbb{R}^d, L_1)$. We use the idea of [15] to compute the stretch factor of T. First, T is partitioned into two subtrees T_1 and T_2 sharing a single vertex $\boldsymbol{p}$ such that each subtree contains between $n/4$ and $3n/4$ vertices. As shown in [15], this can be done in $O(n)$ time. Then the stretch factors of T_1 and T_2 are found recursively.

What remains is to determine value $\delta_T(T_1, T_2)$ if it is larger than either δ_{T_1} or δ_{T_2}.

Let I_1 and I_2 be indices of vertices in T_1 and T_2 respectively and let $I = I_1 \cup I_2$ be the indices of vertices in T. Let $m = \max_{i \in I_2} L_1(\boldsymbol{p_i}, \boldsymbol{p})$ and let $\delta > 0$. For each $i \in I$, let $B_i(\delta)$ be the L_1-ball $B_1(\boldsymbol{p_i}, r_i(\delta))$, where

$$r_i(\delta) = \begin{cases} (m + L_1^T(\boldsymbol{p_i}, \boldsymbol{p}))/\delta & \text{if } i \in I_1, \\ (m - L_1^T(\boldsymbol{p_i}, \boldsymbol{p}))/\delta & \text{if } i \in I_2. \end{cases}$$

Note that $r_i(\delta) \geq m$ for all $i \in I_1$ and $0 \leq r_i(\delta) \leq m$ for all $i \in I_2$.

Lemma 10 *With the above definitions, $\delta_T(T_1, T_2) = \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall (i,j) \in I_1 \times I_2, i \neq j\}$. Furthermore, for $c = 1, 2$, $\delta_T \geq \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall i, j \in I_c, i \neq j\}$.*

Proof. Let $\delta > 0$. For any $(i, j) \in I_1 \times I_2$, $i \neq j$,

$$\frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} = \frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p}) + L_1^T(\boldsymbol{p_j}, \boldsymbol{p})}{\delta} = r_i(\delta) - r_j(\delta).$$

Hence,

$$\begin{aligned} &\delta_T(T_1, T_2) < \delta \Leftrightarrow \forall (i,j) \in I_1 \times I_2, i \neq j : \\ &\qquad L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) > \frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} = r_i(\delta) - r_j(\delta) \\ &\qquad \Leftrightarrow \forall (i,j) \in I_1 \times I_2, i \neq j : B_j(\delta) \nsubseteq B_i(\delta). \end{aligned}$$

This shows the first part of the lemma.

To show the second part, let $\delta > 0$ and $i, j \in I_c$, $i \neq j$, be given. Since $r_i(\delta) - r_j(\delta) = (|L_1^T(\boldsymbol{p_i}, \boldsymbol{p}) - L_1^T(\boldsymbol{p_j}, \boldsymbol{p})|)/\delta$,

$$\begin{aligned} B_j(\delta) \subseteq B_i(\delta) \Rightarrow L_1(\boldsymbol{p_i}, \boldsymbol{p_j}) &\leq \frac{|L_1^T(\boldsymbol{p_i}, \boldsymbol{p}) - L_1^T(\boldsymbol{p_j}, \boldsymbol{p})|}{\delta} \\ &\leq \frac{L_1^T(\boldsymbol{p_i}, \boldsymbol{p_j})}{\delta} \\ \Rightarrow \delta \leq \delta_T. \end{aligned}$$

It follows that

$$\begin{aligned} \delta_T &\geq \sup\{\delta > 0 | B_j(\delta) \subseteq B_i(\delta) \text{ for some } i, j \in I_c, i \neq j\} \\ &= \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall i, j \in I_c, i \neq j\}. \end{aligned}$$

□

Corollary 11 *With the above definitions, $\delta_T \geq \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall i, j \in I, i \neq j\}$ with equality if $\delta_T = \delta_T(T_1, T_2)$.*

Proof. Lemma 10 implies that

$$\begin{aligned} \delta_T &\geq \delta_T(T_1, T_2) \\ &= \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall (i,j) \in I_1 \times I_2, i \neq j\} \end{aligned}$$

and that, for $c = 1, 2$,

$$\delta_T \geq \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall i, j \in I_c, i \neq j\}.$$

□

Corollary 11 shows that if we pick the maximum of $\delta_T(T_1)$, $\delta_T(T_2)$, and the value

$$\inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall i, j \in I, i \neq j\}, \qquad (4)$$

then we obtain the stretch factor of T. Computing (4) is done exactly as for paths in $O(n \log^d n)$ time and $O(n)$ space.

It follows from the above that our algorithm has $O(\log n)$ recursion levels and uses $O(n \log^d n)$ time per level. This gives the following result.

Theorem 12 *The stretch factor of a tree with n vertices embedded in $(\mathbb{R}^d, L_1)$ can be computed in $O(n \log^{d+1} n)$ time and $O(n)$ space.*

The above arguments also apply to the weighted fixed orientation metrics, giving the following theorem.

Theorem 13 *Let $\mathcal{V}$ be a set of $\sigma \geq 2$ vectors defining a weighted fixed orientation metric $d_{\mathcal{V}}$ on $\mathbb{R}^2$. Then the stretch factor of a tree with n vertices in $(\mathbb{R}^2, d_{\mathcal{V}})$ can be computed in $O(\sigma n \log^3 n)$ time and $O(n)$ space.*

7 Stretch Factor of Cycles

We now show how to compute the stretch factor of an n-vertex cycle in $O(\sigma n \log^3 n)$ time for the weighted fixed orientation metrics and in $O(n \log^{d+1} n)$ time in the space $(\mathbb{R}^d, L_1)$ using $O(n)$ space.

We will restrict our attention to the space $(\mathbb{R}^2, L_1)$ since the weighted fixed orientation metrics are handled in a similar way and since generalizing the results of this section to higher dimensions follows from ideas similar to those of Section 5. So let C be a an n-vertex cycle $\boldsymbol{p_1} \to \boldsymbol{p_2} \to \cdots \to \boldsymbol{p_n} \to \boldsymbol{p_1}$ embedded in $(\mathbb{R}^2, L_1)$.

The problem of computing the stretch factor of C is harder than for paths since there are now two possible paths between each pair of distinct vertices.

To handle this, it will prove useful to replace C by a $2n$-vertex path $P = \boldsymbol{q_1} \to \cdots \to \boldsymbol{q_{2n}}$, where $\boldsymbol{q_i} = \boldsymbol{q_{n+i}} = \boldsymbol{p_i}$ for $i = 1, \ldots, n$.

For $i = 1, \ldots, 2n$ and for $\delta > 0$, let $B_i(\delta)$ denote the L_1-disc $B_1(\boldsymbol{q_i}, r_i(\delta))$, where $r_i(\delta) = L_1^P(\boldsymbol{q_i}, \boldsymbol{q_{2n}})/\delta$. Let m_C denote half the length of C.

With these definitions, we obtain the following result which is similar to Lemma 1.

Lemma 14 *With the above definitions, the stretch factor of C is $\delta_C = \inf\{\delta > 0 | B_j(\delta) \nsubseteq B_i(\delta) \forall 1 \leq i, j \leq 2n, i \neq j, L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}$.*

Proof. The proof follows by applying the ideas of the proof of Lemma 1 and from the observation that

$$\delta_C = \max\{\frac{L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j})}{L_1(\boldsymbol{q_i}, \boldsymbol{q_j})} | 1 \leq i, j \leq 2n, i \neq j, L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}.$$

□

For $1 \leq i \leq 2n$ and for $w = 1, 2, 3, 4$, define $P_w(i)$ as the set of vertices of $P \setminus \{\boldsymbol{p_i}\}$ belonging to the wth quadrant of $\boldsymbol{q_i}$. Lemma 14 gives

$$\delta_C = \max_{w=1,2,3,4} \max_{1 \leq i \leq 2n} \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{q_j} \in P_w(i), L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}.$$

Hence, restricting our attention to first quadrants, the problem of computing

$$\max_{1 \leq i \leq 2n} \inf\{\delta > 0 | B_j(\delta) \not\subseteq B_i(\delta) \forall \boldsymbol{q_j} \in P_1(i), L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\} \qquad (5)$$

is essentially the same as the problem of computing (1) of Section 3 except for one thing: only pairs of indices i, j, where $L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C$, are allowed.

Note that Lemma 2 also applies in this section. Let us therefore associate f_i- and l_i-functions to each vertex $\boldsymbol{q_i}$ of P. We define f_i as in Section 3 and define l_i by

$$l_i(1/\delta) = \min\{f_j(1/\delta) | \boldsymbol{q_j} \in P_1(i), L_1^P(\boldsymbol{q_i}, \boldsymbol{q_j}) \leq m_C\}.$$

For $1 \leq i \leq 2n$, let $\delta_i = 1/x_i$, where x_i is the intersection point between f_i and l_i. If no such point exists, set $\delta_i = 0$. Then it follows easily from the results of Section 3 that (5) equals $\max_{1 \leq i \leq 2n} \delta_i$

What remains is to compute values x_i for all i. In the following, we describe an algorithm for this problem.

The algorithm stores vertices of P in a 1-dimensional range tree $\mathcal{T}$. Unlike in Section 3.1 however, vertices are not ordered in ascending x but by ascending distance to $\boldsymbol{q_1}$ in P. We will denote the leaves of $\mathcal{T}$ from left to right by $q_1, \ldots, q_{2n}$.

Let v be a node of $\mathcal{T}$ and let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ rooted at v. Associated with v is a 1-dimensional range tree for those vertices of P stored in the leaves of $\mathcal{T}_v$. This range tree is of the form described in Section 3.1 and will be updated in the same way.

Vertices of P are then considered in order of descending y (as in Section 3.1). Let $\boldsymbol{q_i}$ be the current vertex. Then range trees associated with nodes of $\mathcal{T}$ need to be updated w.r.t. $\boldsymbol{q_i}$. These nodes are the nodes on the path from the root of $\mathcal{T}$ to the leaf containing $\boldsymbol{q_i}$ since their associated range trees are exactly those that contain $\boldsymbol{q_i}$.

What remains in the processing of $\boldsymbol{q_i}$ is to compute the intersection between f_i and l_i. This involves computing intersections between f_i and lower envelopes in range trees associated with nodes of $\mathcal{T}$. However, only range trees containing vertices all of distance at most m_C to $\boldsymbol{q_i}$ in P should be considered.

Such range trees are picked as follows. First, $O(\log n)$ subtrees of $\mathcal{T}$ are picked such that they cover all leaves associated with vertices of P having distance at most m_C to $\boldsymbol{p_i}$. This is done using an algorithm similar to the range query algorithm of Section 5.1 of [3] with query range $[L_1^P(\boldsymbol{p_1}, \boldsymbol{p_i}) - m_C, L_1^P(\boldsymbol{p_1}, \boldsymbol{p_i}) + m_C]$. Then the range trees picked are those associated with roots of these subtrees.

The intersections between f_i and lower envelopes in the picked range trees are found as described in Section 3.1. The leftmost of these intersections is then the intersection x_i between f_i and l_i.

When all vertices of P have been considered in the y-descending path of vertices, the value (5) is found as $\max_{1 \leq i \leq 2n} \delta_i$, where $\delta_i = 1/x_i$.

The running time of the above algorithm is $O(n \log^3 n)$. This follows easily from the results of Section 3.2 and from the fact that the number of range trees considered for each vertex of P is $O(\log n)$.

Linear space requirement is obtained by making $O(\log^2 n)$ y-descending passes instead of one pass. In each pass, only range trees associated with nodes at a certain depth of $\mathcal{T}$ are considered and only nodes at a certain depth of each of the range trees associated with nodes of $\mathcal{T}$ are considered. We will leave out the details since they are similar to those of Section 3.3.

Generalizing the above to higher dimensions and to weighted fixed orientation metrics gives the following theorems.

Theorem 15 *The stretch factor of a cycle with n vertices embedded in $(\mathbb{R}^d, L_1)$ can be computed in $O(n \log^{d+1} n)$ time and $O(n)$ space.*

Theorem 16 *Let $\mathcal{V}$ be a set of $\sigma \geq 2$ vectors defining a weighted fixed orientation metric $d_{\mathcal{V}}$ on $\mathbb{R}^2$. Then the stretch factor of a cycle with n vertices in $(\mathbb{R}^2, d_{\mathcal{V}})$ can be computed in $O(\sigma n \log^3 n)$ time and $O(n)$ space.*

8 Finding a Vertex Pair Achieving the Stretch Factor

In the previous sections, we have considered the problem of computing the stretch factor of paths, trees, and cycles. Suppose that we are also interested in actually finding a pair of vertices for which the detour between them equals the stretch factor of the graph.

The algorithms described above are easily modified to find such a pair without affecting the time and space bounds. To achieve this, we make the following small change to the data structures defining lower envelope functions as follows. Let l be a lower envelope function considered during the course of the algorithm. Then we associate with every line segment (and halfline) of l the f_j-function defining this segment.

Now, suppose the stretch factor of the graph has been found. This value corresponds to a computed intersection between an f_i-function and a lower envelope function for some i. The above modification then allows us to find, in constant time, a vertex $\boldsymbol{p_j}$ such that the stretch factor of the graph is achieved as the detour between $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$.

In fact, we can obtain a slightly stronger result which we state in the following theorem.

Theorem 17 *Without affecting time and space bounds, all algorithms described above can be modified to compute, for every vertex $\boldsymbol{p_i}$, a vertex $\boldsymbol{p_j}$ maximizing the detour between $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$.*

9 Concluding Remarks

Given an n-vertex path P embedded in metric space $(\mathbb{R}^d, L_1)$, $d \geq 2$, we showed how to compute the stretch factor of P in $O(n \log^d n)$ worst-case time. For a general weighted fixed orientation metric in the plane, we gave an $O(\sigma n \log^2 n)$ time algorithm, where $\sigma \geq 2$ is the number of fixed orientations. We generalized our algorithms to trees and cycles at the cost of an extra $\log n$-factor in running time. All our algorithms have $O(n)$ space requirement.

An obvious question is whether our algorithms are optimal with respect to running time. In the Euclidean plane, an $\Omega(n \log n)$ lower bound is known for paths (and thus also for trees) and it is easily extended to the weighted fixed orientation metrics (for fixed σ). Thus, in the plane, there is a gap of $\log n$ for paths and $\log^2 n$ for trees between our time bounds and this lower bound.

It should be possible to modify the algorithms presented in [15] for computing the stretch factor of paths, trees, and cycles in the Euclidean plane to handle the rectilinear plane and possibly the more general weighted fixed orientation metrics. This would give an $O(n \log n)$ expected time algorithm for paths and an $O(n \log^2 n)$ expected time algorithm for trees and cycles. Is it possible to extend the ideas of this paper to handle other classes of graphs?

Finally, we believe that it is possible to handle more general fixed orientation metrics in higher dimensions using the ideas of this paper.

References

[1] P. K. Agarwal, R. Klein, C. Knauer, S. Langerman, P. Morin, M. Sharir, and M. Soss. Computing the Detour and Spanning Ratio of Paths, Trees and Cycles in 2D and 3D. *Discrete and Computational Geometry*, 39 (1): 17–37 (2008).

[2] S. Arya, G. Das, D. M. Mount, J. S. Salowe, and M. Smid. Euclidean spanners: short, thin, and lanky. *Proc. 27th ACM STOC*, 1995, pp. 489–498.

[3] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry - Algorithms and Applications (2nd ed.). *Springer-Verlag Berlin*, 1997, 2000.

[4] M. Brazil. Steiner Minimum Trees in Uniform Orientation Metrics. In *D.-Z. Du and X. Cheng, editor, Steiner Trees in Industries*, pages 1–27, Kluwer Academic Publishers, 2001.

[5] M. Brazil, D. A. Thomas, and J. F. Weng. Minimum Networks in Uniform Orientation Metrics. *SIAM Journal on Computing*, 30: 1579–1593, 2000.

[6] M. Brazil, P. Winter, and M. Zachariasen. Flexibility of Steiner Trees in Uniform Orientation Metrics. In *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC), Lecture Notes in Computer Science 3341*, pp. 196–208, 2004.

[7] M. Brazil and M. Zachariasen. Steiner Trees for Fixed Orientation Metrics. *Technical Report 06-11, DIKU, Department of Computer Science, University of Copenhagen*, 2006.

[8] D. Eppstein. Spanning trees and spanners. In *J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry*, pages 425–461, Elsevier Science Publishers, Amsterdam, 2000.

[9] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16 (1987), pp. 1004–1022.

[10] J. Gudmundsson, G. Narasimhan, and M. Smid. Fast pruning of geometric spanners. *STACS* 2005:508–520.

[11] M. Hanan. On Steiner's Problem with Rectilinear Distance. *SIAM Journal on Applied Mathematics*, 14(2):255–265, 1966.

[12] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters*, Vol. 33, no. 4, 1989, pp. 169–174.

[13] F. K. Hwang. On Steiner Minimal Trees with Rectilinear Distance. *SIAM Journal on Applied Mathematics*, 30:104–114, 1976.

[14] F. K. Hwang, D. S. Richards, and P. Winter. The Steiner Tree Problem. *Annals of Discrete Mathematics* 53, Elsevier Science Publishers, Netherlands, 1992

[15] S. Langerman, P. Morin, and M. Soss. Computing the maximum detour and spanning ratio of planar chains, trees and cycles. *Proceedings of the 19th International Symposium on Theoretical Aspects of Computer Science* (STACS '02), Lecture Notes in Computer Science, Vol. 2285, 2002, pp. 250–261.

[16] X. Y. Li and Y. Wang. Efficient construction of low weighted bounded degree planar spanner. *Int. J. Comput. Geometry Appl.* 14(1–2):69–84 (2004).

[17] G. Narasimhan and M. Smid. Geometric Spanner Networks. *Cambridge University Press*, 2007.

[18] B. K. Nielsen, P. Winter, and M. Zachariasen. An Exact Algorithm for the Uniformly-Oriented Steiner Tree Problem. In *Proceedings of the* 10*th European Symposium on Algorithms, Lecture Notes in Computer Science*, pp. 760–772, Springer, 2002.

[19] M. Sarrafzadeh and C. K. Wong. Hierarchical Steiner Tree Construction in Uniform Orientations. *IEEE Transactions on Computer-Aided Design*, 11: 1095–1103, 1992.

[20] M. Smid. Closest point problems in computational geometry. In *J.-R. Sack and J. Urrutia, editors, Handbook of Computational Geometry*, pages 877–935, Elsevier Science Publishers, Amsterdam, 2000.

[21] P. Widmayer, Y. F. Wu, and C. K. Wong. On Some Distance Problems in Fixed Orientations. *SIAM Journal on Computing*, 16(4): 728–746, 1987.

[22] M. C. Yildiz and P. H. Madden. Preferred Direction Steiner Trees. In *Proceedings of the* 11*th Great Lakes Symposium on VLSI (GLSVLSI)*, pages 56–61, 2001.

Girth of a Planar Digraph with Real Edge Weights in $O(n \log^3 n)$ Time

Christian Wulff-Nilsen [*]

Abstract

The girth of a graph is the length of its shortest cycle. We give an algorithm that computes in $O(n \log^3 n)$ time and $O(n)$ space the (weighted) girth of an n-vertex planar digraph with arbitrary real edge weights. This is an improvement of a previous time bound of $O(n^{3/2})$, a bound which was only valid for non-negative edge-weights. Our algorithm can be modified to output a shortest cycle within the same time and space bounds if such a cycle exists.

1 Introduction

The *girth* of an unweighted graph is the length of a shortest cycle in the graph or ∞ if the graph is acyclic. It is a well-studied graph characteristic and has been shown to be related to numerous other properties of graphs, including vertex degree, diameter, connectivity, maximum genus, and vertex colourings [2, 4]. For instance, it is easy to see that for a graph containing a cycle, its girth is at most twice its diameter plus one.

The problem of computing the girth of a graph has received some attention. An $O(mn)$ time algorithm is known where m and n are the number of edges and vertices, respectively [6]. Finding the length of a shortest cycle of even length can be done in $O(n^2)$ time [11].

In this paper, we focus on the problem of computing the girth of planar graphs. For this class of graphs, faster algorithms are known. A linear

[*]Department of Computer Science, University of Copenhagen, koolooz@diku.dk, http://www.diku.dk/~koolooz/

time algorithm is presented in [3] but it only applies when the graph has bounded girth. For general planar graphs, Djidjev gave an $O(n^{5/4} \log n)$ time algorithm. This can be improved to $O(n \log^2 n)$ time by applying the minimum cut algorithm of [1] to the dual graph. Recently, it was shown how to find the girth in $O(n \log n)$ time [10].

All these results for planar graphs assume that the graph is undirected. For planar digraphs, Weimann and Yuster [10] gave an $O(n^{3/2})$ time algorithm and they asked whether a faster algorithm exists.

The girth of a graph is defined for unweighted graphs but this definition immediately extends to the case where edges have real weight. Of the above algorithms for planar graphs, only the $O(n \log^2 n)$ time algorithm in [1] and the $O(n^{3/2})$ time algorithm in [10] can handle weighted graphs and only if all edge weights are non-negative. The $O(n \log^2 n)$ time algorithm only applies when the graph is undirected.

We consider the most general version of the problem for planar graphs. We show how to find the girth of an n-vertex planar digraph with arbitrary real edge weights (non-negative as well as negative) in $O(n \log^3 n)$ time and $O(n)$ space. In particular, we answer the question by Weimann and Yuster [10] of whether an $o(n^{3/2})$ time algorithm exists for computing the girth of a planar digraph. Our algorithm can output a shortest cycle within the same time and space bounds, assuming such a cycle exists.

The organization of the paper is as follows. In Section 2, we introduce some basic definitions and notation. We give our algorithm for computing the girth of a planar digraph in Section 3 and in Section 4, we show how to extend this algorithm to output a shortest cycle. Finally, we make some concluding remarks in Section 5.

2 Definitions and Notation

For a graph H, we let V_H and E_H denote its vertex and edge set, respectively. Let $G = (V, E)$ be a digraph with real edge weights defined by weight function $w : E \rightarrow \mathbb{R}$. For vertices $u, v \in V$, we let $d_G(u, v)$ denote the length of a shortest path in G from u to v w.r.t. w (we omit w in the notation but this should not cause any confusion). If no path from u to v exists, we define $d_G(u, v) = \infty$ and if there is a path from u to v containing a negative-weight cycle, $d_G(u, v) = -\infty$.

The *(weighted) girth* of G is the length of a shortest cycle in G w.r.t. w. If

G is acyclic, we define its girth to be ∞. If G contains a negative-weight cycle C, cycles of arbitrarily large negative weight can be obtained by traversing C sufficiently many times so in this case, we define the girth of G to be $-\infty$.

3 Computing the Girth

In the following, let $G = (V, E)$ be an n-vertex planar digraph with real edge weights defined by weight function $w : E \rightarrow \mathbb{R}$. In this section, we show how to compute the girth of G in $O(n \log^3 n)$ time and $O(n)$ space.

We may assume that G contains no negative-weight cycles since the algorithm in [5] can detect such cycles within our time and space bounds. If a negative-weight cycle is present, our algorithm outputs $-\infty$ as the girth of G.

We will assume that G is triangulated with pairs of oppositely directed edges. If it is not, this can be achieved by adding edges of sufficiently high weight W so that finite shortest path distances in G will not be affected. We define

$$W = 1 + 2 \sum_{e \in E'} |w(e)|,$$

where E' is the set of edges in the original graph. This way, we avoid dealing with infinite shortest path distances. And we can still detect the case where the girth is ∞ in the original graph since this holds if and only if the girth of the triangulated graph is at least $1 + \sum_{e \in E'} |w(e)|$.

We will identify G with a fixed plane embedding of the graph.

Next, we apply the linear time cycle separator theorem of Miller [9] to G. This gives a simple cycle C in G (ignoring edge orientations) containing $O(\sqrt{n})$ vertices such that at most $2n/3$ vertices of G are in the closed bounded region R_1 resp. closed unbounded region R_2 of the plane defined by C. Let G_1 resp. G_2 be the subgraph of G containing the set of vertices and edges of G in R_1 resp. R_2. If an edge of G belongs to both R_1 and R_2, i.e., to C, then we only add it to one of the two subgraphs, say G_1. This ensures that G_1 and G_2 are edge-disjoint.

We recursively compute the girth g_1 of G_1 and the girth g_2 of G_2. Let g denote the length of a shortest cycle in G that contains at least two vertices of C. Any simple cycle in G that is neither fully contained in G_1 nor in G_2 must contain at least two vertices of C. Thus, $\min\{g_1, g_2, g\}$ is the girth of G so let us consider the problem of computing g.

Let H be the complete digraph with vertex set $V_H = V_C$ and with weight function $w_H : E_H \to \mathbb{R}$ defined by $w_H(u,v) = \min\{d_{G_1}(u,v), d_{G_2}(u,v)\}$ for all distinct $u, v \in V_H$.

For any $u, v \in V_H$, a shortest path from u to v in G can be decomposed into subpaths each of which has the property that it is a shortest path in either G_1 or in G_2 with both its endpoints on C. It follows that $d_H(u,v) = d_G(u,v)$.

We apply the algorithm in [7] to compute $d_{G_1}(u,v)$ and $d_{G_2}(u,v)$ for all $u, v \in V_C$ using a total of $O(n \log^2 n)$ time over all recursion levels. Hence, we obtain H and its edge weights in this amount of time over all recursion levels.

Any shortest cycle in G containing at least two vertices of C can be decomposed into subpaths each having the property above. Hence, such a cycle has the same length as a shortest cycle in H so H has girth g.

What remains therefore is to find the length of a shortest cycle in H. We will show how to do this in $O(n \log^2 n)$ time. Since there are $O(\log n)$ recursion levels, this will imply that the girth of G can be obtained in $O(n \log^3 n)$ time.

Let $u_1, \ldots, u_m$ be the vertices of C. To compute the girth g of H, we will compute, for $i = 1, \ldots, m$, the length g_i of a shortest cycle in H containing u_i. Then it is clear that we can obtain the girth of H as $g = \min\{g_1, \ldots, g_m\}$.

We first reduce our problem to one where all edge weights are non-negative. This is done as follows. We compute single source shortest path distances with source, say, u_1, using the $O(n \log^2 n)$ time Bellman-Ford variant of [5] (in fact, it can be done in only $O(n\alpha(n))$ time with ideas from [8] but this will not improve our bound).

Then in $O(|E_H|) = O(n)$ additional time, we can obtain the *reduced cost* $w_H^+(e)$ of each edge $e = (u_i, u_j)$ of H (w.r.t. w_H):

$$w_H^+(e) = d_H(u_1, u_i) + w_H(e) - d_H(u_1, u_j).$$

By the triangle inequality, $w_H^+ \geq 0$ and it is easy to see that for any cycle in H, its length w.r.t. w_H is identical to its length w.r.t. w_H^+. This gives us the desired reduction.

For each pair of vertices u_i and u_j in H, we let $d_H^+(u_i, u_j)$ denote the shortest path distance from u_i to u_j in H w.r.t. w_H^+.

Now, let us consider the problem of computing g_i for some i. Since $g_i = \min\{d_H^+(u_i, u_j) + w_H^+(u_j, u_i) | j = 1, \ldots, m, j \neq i\}$, we can obtain this value in $O(m) = O(\sqrt{n})$ time in addition to the time for computing single source shortest path distances from u_i w.r.t. w_H^+. And since $w_H^+ \geq 0$, we can apply the Dijkstra variant of [5] to compute these distances in $O(m \log^2 n) = O(\sqrt{n} \log^2 n)$ time. Over all i, this is $O(n \log^2 n)$, as desired.

It follows from the above that we can find the girth of H in $O(n \log^2 n)$ time and we can conclude this section with the following result.

Theorem 1. *The girth of an n-vertex planar digraph with real edge weights can be computed in $O(n \log^3 n)$ time and $O(n)$ space.*

Proof. We have shown the time bound above and since the algorithms in [5, 7, 8, 9] all require $O(n)$ space, this bound also holds for our algorithm. □

4 Finding a Shortest Cycle

We now show how to extend the algorithm of the previous section to compute a shortest cycle in G within the same time and space bounds. We may assume that G contains no negative-weight cycles since otherwise, a shortest cycle does not exist.

With the definitions above, it suffices to find in $O(n \log^2 n)$ time and $O(n)$ space a shortest cycle in G containing at least two vertices of C. As already noted, such a cycle corresponds to a shortest cycle in H. The algorithm above finds vertices u_i and u_j of C such that a shortest path P_{ij} in H from u_i to u_j followed by the edge (u_j, u_i) is a shortest cycle C' in H. The edges on P_{ij} can be found by traversing the shortest path tree in H rooted at u_i. Since this shortest path tree has already been computed, we can thus find all edges of C' within the required time and space bounds.

What remains is to replace each edge e of C' by a simple shortest path P_e in G between its endpoints. We will show how to do this in $O(n \log n)$ additional time.

To obtain these paths, we need to take a closer look at the multiple-source shortest path algorithm of Klein [7] which we applied to find the weights of edges of H. For G_i, $i = 1, 2$, his algorithm maintains a dynamic tree data structure, which is initally a shortest path tree in G_i rooted at, say, u_1, then at u_2, and so on. To obtain the weights of edges of H, this data structure is repeatedly queried, first for the distance in G_i from u_1 to all other vertices of C, then from u_2, et cetera.

Now, we also need the actual paths in the shortest path trees corresponding to these distances. Once the edges of C' have been identified, we can run Klein's algorithm again. During the course of this second run of the algorithm, we can query the dynamic tree data structure to obtain the paths corresponding to edges of C'. Querying for a single vertex on a path takes $O(\log n)$ time so the total time is $O(k \log n)$, where k is the total number of vertices (with repetitions) over all paths.

For this strategy to work, k should not be too big. We will ensure this by modifying Dijkstra without increasing its time and space bounds such that C' will have the least number of edges among all shortest cycles in H. We then show that none of the paths defining edges of C' will share any edges of G, implying that $k = O(n)$ and hence that a shortest cycle in G can be found in $O(n \log^3 n + k \log n) = O(n \log^3 n)$ time and $O(n)$ space.

To output a shortest cycle in H with the minimum number of edges, we modify the Dijkstra algorithm in [5]. We will not go through all the details but assume that the reader is familiar with the paper.

During the course of that algorithm, we maintain not only shortest path distances w.r.t. w_H^+ but also w.r.t. the unit weight function. Now, whenever there is a tie between which vertex to pick next from the heap, we pick one for which the number of edges on a shortest path from the root of the partially built shortest path tree to u_i is minimized. This can be incorporated into the algorithm without increasing its time and space bounds by lexicographically ordering vertices, first according to weighted and then according to unweighted distances from the root of the shortest path tree.

It remains to show that if C' is a shortest cycle in H and it is picked such that it has the minimum number of edges then none of the shortest paths in G corresponding to edges of C' share edges.

So let P_1 and P_2 be shortest paths in G corresponding to distinct edges $e_1 = (u_i, u_j)$ and $e_2 = (u_{i'}, u_{j'})$ in H, respectively. Assume for the sake of contradiction that $e = (u, v)$ is an edge shared by P_1 and P_2. Let P be the subpath of C' starting in u_j and ending in $u_{i'}$. Without increasing the length of C', the subpath $P_1 P P_2$ of C' can be replaced by the path P' defined as the prefix of P_1 ending in v followed by the suffix of P_2 starting in v. Since G_1 and G_2 are edge-disjoint, either P_1 and P_2 both belong to G_1 or both belong to G_2. Hence, P' is a path in either G_1 or in G_2 so we can replace e_1 and e_2 by the edge $(u_i, u_{j'})$ in H. But this reduces the number of edges of C' without increasing its length, contradicting the choice of the cycle.

It follows that none of the shortest paths in G corresponding to edges of

C' share edges. By the above, this suffices to show the following.

Theorem 2. *A shortest cycle in an n-vertex planar digraph with real edge weights can be computed in $O(n \log^3 n)$ time and $O(n)$ space, assuming such a cycle exists.*

If G has girth $-\infty$, a shortest cycle does not exist. But we can still output a negative-weight cycle within our time and space bounds by applying the algorithm in [5].

5 Concluding Remarks

We showed how to compute the girth of an n-vertex planar digraph with real edge weights in $O(n \log^3 n)$ time and $O(n)$ space. This is a significant improvement over the previous best bound of $O(n^{3/2})$ which only applied to planar digraphs with non-negative edge weights. We also showed how to output a shortest cycle without an increase in time or space, assuming such a cycle exists.

In [5], it is suggested that the results of that paper can be generalized to the class of bounded genus graphs. If this is true, we believe that a generalization of our algorithm to this class is also achievable.

References

[1] P. Chalermsook, J. Fakcharoenphol, and D. Nanongkai. A deterministic near-linear time algorithm for finding minimum cuts in planar graphs. In SODA, pages 828–829, 2004.

[2] R. Diestel. Graph theory. Springer-Verlag, 2nd edition, 2000.

[3] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. Journal of Graph Algorithms and Applications, 3(3):1–27, 1999.

[4] P. Erdős. Graph theory and probability. Canadian Journal of Mathematics, 11:34–38, 1959.

[5] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. Available from the authors' webpages. Preliminary version in FOCS'01.

[6] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. SIAM J. Comput., 7(4):413–423, 1978.

[7] P. N. Klein. Multiple-source shortest paths in planar graphs. Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 146–155.

[8] P. N. Klein, S. Mozes, and O. Weimann. Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n \log^2 n)$-Time Algorithm. Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms, p. 236–245, 2009.

[9] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.

[10] O. Weimann and R. Yuster. Computing the Girth of a Planar Graph in $O(n \log n)$ time. In Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP 2009).

[11] R. Yuster and U. Zwick. Finding even cycles even faster. SIAM J. Discrete Math., 10(2):209–222, 1997.

Minimum Cycle Basis and All-Pairs Min Cut of a Planar Graph in Subquadratic Time

Christian Wulff-Nilsen [*]

March 13, 2010

Abstract

A minimum cycle basis of a weighted undirected graph G is a basis of the cycle space of G such that the total weight of the cycles in this basis is minimized. If G is a planar graph with non-negative edge weights, such a basis can be found in $O(n^2)$ time and space, where n is the size of G. We show that this is optimal if an explicit representation of the basis is required. We then present an $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space algorithm that computes a minimum cycle basis *implicitly*. From this result, we obtain an output-sensitive algorithm that explicitly computes a minimum cycle basis in $O(n^{3/2}\log n + C)$ time and $O(n^{3/2} + C)$ space, where C is the total size (number of edges and vertices) of the cycles in the basis. These bounds reduce to $O(n^{3/2}\log n)$ and $O(n^{3/2})$, respectively, when G is unweighted. We get similar results for the all-pairs min cut problem since it is dual equivalent to the minimum cycle basis problem for planar graphs. We also obtain $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space algorithms for finding, respectively, the weight vector and a Gomory-Hu tree of G. The previous best time and space bound for these two problems was quadratic. From our Gomory-Hu tree algorithm, we obtain the following result: with $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space for preprocessing, the weight of a min cut between any two given vertices of G can be reported in constant time. Previously, such an oracle required quadratic time and space for preprocessing. The oracle can also be extended to report the actual cut in time proportional to its size.

[*]Department of Computer Science, University of Copenhagen, `koolooz@diku.dk`, `http://www.diku.dk/hjemmesider/ansatte/koolooz/`

1 Introduction

A cycle basis of a graph is a set of cycles that gives a compact representation of the set of all the cycles in the graph. Such a representation is not only of theoretical interest but has also found practical use in a number of fields. One of the earliest applications is in electrical circuit theory and dates back to the work of Kirchhoff [17] in 1847. Knuth [18] used them in the analysis of algorithms. Furthermore, cycle bases play an important role in chemical and biological pathways, periodic scheduling, and graph drawing [15]. See also [4, 5, 6, 8, 20, 23, 25].

In many of the above applications, it is desirable to have a cycle basis of minimum total length or, more generally, of minimum total weight if edges of the graph are assigned weights. The minimum cycle basis problem, formally defined below, is the problem of finding such a cycle basis. For a survey of applications and the history of this problem, see [14].

Let us define cycle bases and minimum cycle bases. Let $G = (V, E)$ be an undirected graph. To each simple cycle C in G, we associate a vector x indexed on E, where $x_e = 1$ if e belongs to C and $x_e = 0$ otherwise. A set of simple cycles of G is said to be *independent* if their associated vectors are independent over $GF(2)$. The vector space over this field generated by these vectors is the *cycle space* of G and a maximal independent set of simple cycles of G is called a *cycle basis* of G. Any cycle basis of G consists of $m - n + c$ cycles, where m is the number of edges, n the number of vertices, and c is the number of connected components of G [26].

Assume that the edges of G have real weights. Then a *minimum cycle basis* (MCB) of G is a cycle basis such that the sum of weights of edges of the cycles in this basis is minimized. The MCB problem (MCBP) is the problem of finding an MCB of G.

The MCBP is NP-hard if negative edge weights are allowed [12]. The first polynomial time algorithm for graphs with non-negative edge weights was due to Horton [14]. His idea was to first compute a polynomial size set of cycles guaranteed to contain an MCB. In a subsequent step, such a basis is then extracted from this set using a greedy algorithm. Running time is $O(m^3 n)$. This was improved in a sequence of papers [7, 9, 2, 16, 21, 1] to $O(m^\omega)$, where ω is the exponent of matrix multiplication.

For planar graphs with non-negative edge weights, an $O(n^2 \log n)$ algorithm was presented in [12]. This was recently improved to $O(n^2)$ [1].

It is well-known that a cycle in a plane graph corresponds to a cut in the dual plane graph. Using this fact, Hartvigsen and Mardon [12] showed that the following problem is dual equivalent to the MCBP for planar graphs (meaning that one problem can be transformed into the other in linear time): find a minimal collection of cuts such that for any pair of vertices s and t, this collection contains a minimum s-t cut. We refer to this problem as the *all-pairs min cut problem* (APMCP). It follows from this result that the quadratic time bound in [1] also holds for the APMCP for planar graphs.

We prove that quadratic running time for the MCBP for planar graphs is optimal by presenting a family of planar graphs of arbitrarily large size for which the total length (number of edges) of all cycles in any MCB is $\Theta(n^2)$. Due to the correspondence between cycles in the primal and cuts in the dual, this lower bound also holds for the APMCP for planar graphs.

We then present an algorithm with $O(n^{3/2}\log n)$ running time and $O(n^{3/2})$ space requirement that computes an MCB of a planar graph *implicitly*. From this result, we get an output-sensitive algorithm with $O(n^{3/2}\log n + C)$ time and $O(n^{3/2} + C)$ space requirement, where C is the total size of cycles in the MCB that the algorithm returns. For unweighted planar graphs, these bounds simplify to $O(n^{3/2}\log n)$ and $O(n^{3/2})$, respectively. Since the MCBP and the APMCP are dual equivalent for planar graphs, we get similar bounds for the latter problem.

The *weight vector* of a weighted graph G is a vector containing the weights of cycles of an MCB in order of non-decreasing weight. Finding such a vector has applications in chemistry and biology [3]. From our implicit representation of an MCB, we obtain an $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space algorithm for finding the weight vector of a planar graph. The best previous bound was $O(n^2)$, obtained by applying the algorithm in [1].

A *Gomory-Hu tree*, introduced by Gomory and Hu in 1961 [10], is a compact representation of minimum weight cuts between all pairs of vertices of a graph. Formally, a Gomory-Hu tree of a weighted connected graph G is a tree T with weighted edges spanning the vertices of G such that:

1. for any pair of vertices s and t, the weight of the minimum s-t cut is the same in G and in T, and

2. for each edge e in T, the weight of e equals the weight of the cut in G, defined by the sets of vertices corresponding to the two connected components in $T \setminus \{e\}$.

Such a tree T is very useful for finding a minimum s-t cut in G since we only need to consider the cuts of G encoded by the edges on the simple path between s and t in T. Gomory-Hu trees have also been applied to solve the minimum k-cut problem [24].

For planar graphs, quadratic time and space is the best known bound for finding such a tree. The bound can easily be obtained with the algorithm in [1]. From our MCB algorithm, we obtain an algorithm that constructs a Gomory-Hu tree in only $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space.

An important corollary of the latter result is that with $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space for preprocessing, a query for the weight of a min cut (or max flow) between two given vertices of a planar undirected graph with non-negative edge weights can be answered in constant time. Previously, quadratic preprocessing time and space was required to obtain such an oracle. The actual cut can be reported in time proportional to its size.

The organization of the paper is as follows. In Section 2, we give some definitions and notation and state some basic results. We give the quadratic lower bound for an explicit representation of an MCB of a planar graph in Section 3. In Section 4, we mention the greedy algorithm which has been applied in previous papers to find an MCB. Based on it, we present a new divide-and-conquer algorithm in Section 5. We first only desccribe how to deal with cycles that have not been recursively computed. In Section 6, we give an implementation of this algorithm with the desired time and space bounds. In Section 7, we extend the algorithm to deal with the recursively computed cycles as well and we again bound time and space. The corollaries of our result are presented in Section 8. In order for our ideas to work, we need shortest paths to be unique. We show how to ensure this in Section 9. Finally, we give some concluding remarks in Section 10.

2 Definitions, Notation, and Basic Results

In the following, $G = (V, E)$ denotes an n-vertex plane, straight-line embedded, undirected graph. This embedding partitions the plane into maximal open connected sets and we refer to the closure of these sets as the *elementary faces* (of G). Exactly one of the elementary faces is unbounded and we call it the *external elementary face* (of G). All other elementary faces are called *internal*.

A Jordan curve $\mathcal{J}$ partitions the plane into an open bounded set and an open unbounded set. We denote them by $int(\mathcal{J})$ and $ext(\mathcal{J})$, respectively.

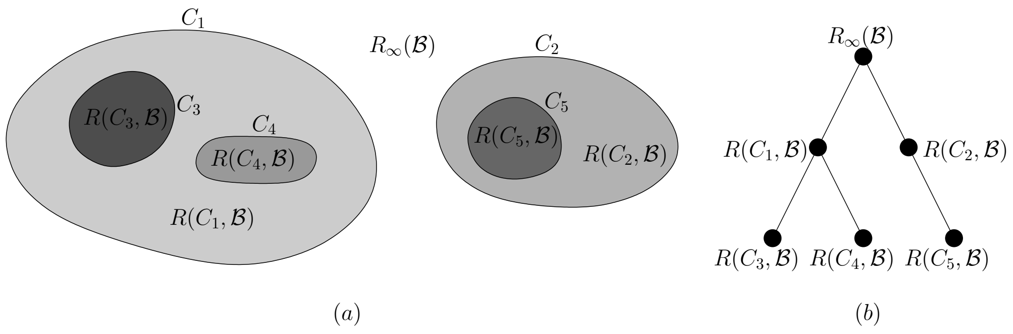


Figure 1: (a): A nested set $\mathcal{B}$ of five cycles C_1, C_2, C_3, C_4, C_5 defining five internal regions and an external region $R_\infty(\mathcal{B})$ (white). (b): The region tree $\mathcal{T}(\mathcal{B})$ of $\mathcal{B}$.

We refer to the closure of these sets as $\overline{int}(\mathcal{J})$ and $\overline{ext}(\mathcal{J})$, respectively.

We say that a pair of elementary faces of G are *separated* by a Jordan curve $\mathcal{J}$ if one face is contained in $\overline{int}(\mathcal{J})$ and the other face is contained in $\overline{ext}(\mathcal{J})$.

A set of simple cycles of G is called *nested* if, for any two distinct cycles C and C' in that set, either $int(C) \subset int(C')$, $int(C') \subset int(C)$, or $int(C) \subset ext(C')$. A simple cycle C is said to *cross* another simple cycle C' if $\{C, C'\}$ is not nested.

For cycles C and C' in a nested set $\mathcal{B}$, we say that C is a *child* of C' and C' is the *parent* of C (w.r.t. $\mathcal{B}$) if $int(C) \subset int(C')$. We also define ancestors and descendants in the obvious way. We can represent these relationships in a forest where each tree vertex corresponds to a cycle of $\mathcal{B}$.

For any cycle $C \in \mathcal{B}$, we define *internal region* $R(C, \mathcal{B})$ as the subset $\overline{int}(C) \setminus (\cup_{i=1,\ldots,k} int(C_i))$ of the plane, where $C_1, \ldots, C_k$ are the children (if any) of C, see Figure 1(a).

The *external region* $R_\infty(\mathcal{B})$ is defined as the set $\mathbb{R}^2 \setminus (\cup_{i=1,\ldots,k} int(C_i))$, where $C_1, \ldots, C_k$ are the cycles associated with roots of trees in the forest defined above. Collectively, we refer to the internal regions and the external region as *regions*.

With $C_1, \ldots, C_k$ defined as above for a region R (internal or external), we refer to the internal regions $R(C_i, \mathcal{B})$ as the *children* of R and we call R the *parent* of these regions. Again, we can define ancestors and descendants in the obvious way. Note that the external region is the ancestor of all other regions. We can thus represent the relationships in a tree where each vertex corresponds to a region. We call it the *region tree* of $\mathcal{B}$ and denote it by $\mathcal{T}(\mathcal{B})$, see Figure 1(b).

Note that for two cycles C and C' in $\mathcal{B}$, C is a child of C' if and only if $R(C, \mathcal{B})$ is a child of $R(C', \mathcal{B})$. Hence, the region tree $\mathcal{T}(\mathcal{B})$ also describes the parent/child relationships between cycles of $\mathcal{B}$.

The elementary faces of G belonging to a region R are the *elementary faces* of R. For each child C_i of R, $\overline{int}(C_i)$ is called a *non-elementary face* of R. If R is an internal region $R(C, \mathcal{B})$, the *external face* of R is the subset $\overline{ext}(C)$ of the plane and we classify it as a non-elementary face of R. Collectively, we refer to the elementary and non-elementary faces of R as its *faces*.

A cycle C in G is said to be *isometric* if for any two vertices $u, v \in C$, there is a shortest path in G between u and v which is contained in C. A set of cycles is said to be isometric if all cycles in the set are isometric.

The *dual* G^* of G is a multigraph having a vertex for each elementary face of G. For each edge e in G, there is an edge e^* in G^* between the vertices corresponding to the two faces of G adjacent to e. The weight of e^* in G^* is equal to the weight of e in G. We identify elementary faces of G with vertices of G^* and since there is a one-to-one correspondence between edges of G and edges of G^*, we identify an edge of G with the corresponding edge in G^*.

Assume in the following that G is connected. Given a vertex $u \in V$, we let $T(u)$ denote a shortest path tree in G with source u. The *dual* of $T(u)$ is the subgraph of G^* defined by the edges not in $T(u)$. It is well-known that this subgraph is a spanning tree in G^* and we denote it by $\tilde{T}(u)$.

A *Horton cycle* of G is a cycle obtained by adding a single edge e to a shortest path tree in G rooted at some vertex r. We denote this cycle by $C(r, e)$. For a subset V' of V, we let $\mathcal{H}(V')$ denote the set of Horton cycles of G obtained from shortest path trees rooted at vertices of V'.

For any graph H, we let V_H and E_H denote its vertex and edge set, respectively. If $w : E \to \mathbb{R}$ is a weight function on the edges of G, we say that a subgraph H of G has *weight* $W \in \mathbb{R}$ if $\sum_{e \in E_H} w(e) = W$.

3 A Tight Lower Bound

In this section, we show that there are planar graphs of arbitrarily large size for which the total length of cycles in any MCB is quadratic. This implies that the quadratic time algorithm in [1] is optimal.

The instance G_n containing n vertices is defined as follows. Let $v_1, \ldots, v_n$ be the vertices of G_n. For $i = 1, \ldots, n-1$, there is an edge $e_i = (v_i, v_{i+1})$ of weight 0. For $i = 1, \ldots, n-2$, there is an edge $e'_i = (v_1, v_{i+2})$ of weight 1.

Since G_n has $m = 2n-3$ edges, any MCB of G_n consists of $m-n+1 = n-2$ cycles. In such a basis, every cycle must contain at least one of the edges e'_i, $i = 1, \ldots, n-2$. Hence, the cycles in any MCB of G_n have total weight at least $n-2$.

For $i = 1, \ldots, n-2$, let C_i be the cycle containing edges $e_1, \ldots, e_{i+1}, e'_i$ in that order. It is easy to see that the set of these cycles is a cycle basis of G. Furthermore, their total weight is $n-2$ so by the above, they must constitute an MCB of G_n. In fact, it is the unique MCB of G_n since in any other cycle basis, some cycle must contain at least two weight 1 edges, implying that the total weight is at least $n-1$.

The cycles in the unique MCB of G_n clearly have quadratic total length. This gives the following result.

Theorem 1. *There are instances of planar graphs of arbitrarily large size n for which the cycles in any MCB for such an instance have total length $\Omega(n^2)$.*

In Section 5, we show how to break the quadratic time bound by computing an implicit rather than an explicit representation of an MCB.

4 The Greedy Algorithm

In the following, $G = (V, E)$ denotes an n-vertex plane, straight-line embedded, undirected graph with non-negative edge weights. We may assume that G is connected since otherwise, we can consider each connected component separately. We require that there is a unique shortest path in G between any two vertices. In Section 9, we show how to avoid this restriction.

The algorithm in Figure 2 will find an MCB of G (see [12, 19]). We call this algorithm the *generic greedy algorithm* and we call the MCB obtained this way a *greedy MCB* (GMCB) (of G). We assume that ties in the ordering in line 2 are resolved in some deterministic way so that we may refer to the cycle basis output in line 5 as *the* GMCB of G. The following two results are from [12].

Lemma 1. *The GMCB is isometric and nested and consists of Horton cycles.*

```
1. initialize $\mathcal{B} = \emptyset$
2. for each simple cycle $C$ of $G$ in order of non-decreasing weight,
3.     if there is a pair of elementary faces of $G$ separated by $C$ and not by
       any cycle in $\mathcal{B}$,
4.         add $C$ to $\mathcal{B}$
5. output $\mathcal{B}$
```

Figure 2: The generic greedy algorithm to compute the GMCB of G.

Lemma 2. *For every pair of elementary faces of a plane undirected graph H with non-negative edge weights, the GMCB of H contains a minimum-weight cycle C in H that separates those two faces. Cycle C is the first such cycle considered when applying the generic greedy algorithm to H.*

Our algorithm is essentially the generic greedy algorithm except that we consider a smaller family of cycles in line 2. The main difficulty in giving an efficient implementation of the greedy algorithm is testing the condition in line 3. Describing how to do this constitutes the main part of the paper.

5 Divide-and-Conquer Algorithm

The family of cycles that we pick in line 2 of the generic greedy algorithm is obtained with the divide-and-conquer paradigm.

To separate our problem, we apply the cycle separator theorem of Miller [22] to G. This gives in linear time a Jordan curve $\mathcal{J}$ intersecting $O(\sqrt{n})$ vertices and no edges of G such that the subgraph G_1 of G in $\overline{int}(\mathcal{J})$ and the subgraph G_2 of G in $\overline{ext}(\mathcal{J})$ each contain at most $2n/3$ vertices. We let $V_{\mathcal{J}}$ denote the set of vertices on $\mathcal{J}$ and refer to them as *boundary vertices* of G.

As in G, we assume that shortest paths in G_1 and G_2 are unique. In Section 9, we show how to avoid this assumption.

For $i = 1, 2$, let $\mathcal{B}_i$ be the GMCB of G_i. Let $\mathcal{B}'_i$ be the subset of cycles of $\mathcal{B}_i$ containing no vertices of $V_{\mathcal{J}}$.

Lemma 3. *With the above definitions, $\mathcal{B}'_1 \cup \mathcal{B}'_2 \cup \mathcal{H}(V_{\mathcal{J}})$ contains the GMCB of G.*

Proof. Let $\mathcal{B}$ be the GMCB of G and let C be a cycle of G not belonging to $\mathcal{B}'_1 \cup \mathcal{B}'_2 \cup \mathcal{H}(V_{\mathcal{J}})$. We need to show that $C \notin \mathcal{B}$.

By Lemma 1, we may assume that C is isometric. Furthermore, $C \notin \mathcal{B}_1 \cup \mathcal{B}_2$ since otherwise, C would belong to $\mathcal{B}_1 \cup \mathcal{B}_2 \setminus (\mathcal{B}'_1 \cup \mathcal{B}'_2)$ and hence to $\mathcal{H}(V_{\mathcal{J}})$ since it is isometric and since shortest paths are unique. Since $C \notin \mathcal{H}(V_{\mathcal{J}})$, C does not contain any vertices of $V_{\mathcal{J}}$ so it belongs to G_i, where $i \in \{1, 2\}$. In particular, it is considered by the generic greedy algorithm in the construction of $\mathcal{B}_i$.

Since $C \notin \mathcal{B}_i$, Lemma 2 implies that every pair of elementary faces (f_1, f_2) of G_i, where $f_1 \subseteq \overline{int}(C)$ and $f_2 \subseteq \overline{ext}(C)$, must be separated by some cycle of $\mathcal{B}_i$ having smaller weight than that of C (or a cycle having the same weight as C but considered earlier in the generic greedy algorithm). We claim that this statement also holds when replacing $\mathcal{B}_i$ by $\mathcal{B}$ and G_i by G. If we can show this, it will imply that C is not added to $\mathcal{B}$ by the generic greedy algorithm.

So let (f_1, f_2) be a pair of elementary faces of G with $f_1 \subseteq \overline{int}(C)$ and $f_2 \subseteq \overline{ext}(C)$. There is an elementary face f'_1 of G_i containing f_1 and an elementary face f'_2 of G_i containing f_2. Since C belongs to G_i, $f'_1 \subseteq \overline{int}(C)$ and $f'_2 \subseteq \overline{ext}(C)$. By the above, there is a cycle of $\mathcal{B}_i$ which is shorter than C and which separates f'_1 and f'_2. This cycle also separates f_1 and f_2 and it follows that f_1 and f_2 are separated by a cycle in $\mathcal{B}$ having weight smaller than that of C. This shows that $C \notin \mathcal{B}$, completing the proof of the lemma. $\square$

Lemma 3 suggests the following divide-and-conquer algorithm for our problem: recursively compute GMCB's $\mathcal{B}_1$ and $\mathcal{B}_2$ of G_1 and G_2 and obtain $\mathcal{B}'_1$ and $\mathcal{B}'_2$ from these sets, compute $\mathcal{H}(V_{\mathcal{J}})$, and extract from $\mathcal{B}'_1 \cup \mathcal{B}'_2 \cup \mathcal{H}(V_{\mathcal{J}})$ the GMCB of G by applying the generic greedy algorithm to this smaller set of cycles. Pseudocode of this algorithm is shown in Figure 3 (it is assumed that a brute-force algorithm is applied to find the GMCB of G when G has constant size). We call it the *recursive greedy algorithm*.

We will show how to implement the top-level of the recursion in $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space. Since each step of the recursion partitions the graph into two subgraphs of (almost) the same size [22], it will follow that these bounds hold for the entire algorithm.

Since the algorithm constructs the GMCB, Lemma 1 implies that $\mathcal{B}$ is isometric and nested at all times. Thus, $\mathcal{B}$ represents a set of regions that change during the course of the algorithm. More specifically, when the algorithm starts, $\mathcal{B} = \emptyset$ and there is only one region, namely the external region $R_\infty(\mathcal{B})$. Whenever a cycle C is added to $\mathcal{B}$ in line 5, the region R containing C is replaced by two new regions, one, R_1, contained in $\overline{int}(C)$ and one, R_2,

```
1. recursively compute GMCB's $\mathcal{B}_1$ and $\mathcal{B}_2$ of $G_1$ and $G_2$, respectively
2. initialize $\mathcal{B} = \emptyset$
3. for each cycle $C \in \mathcal{B}'_1 \cup \mathcal{B}'_2 \cup \mathcal{H}(V_\mathcal{J})$ in order of non-decreasing weight,
4.    if there is a pair of elementary faces of $G$ separated by $C$ and not by
      any cycle in $\mathcal{B}$,
5.       add $C$ to $\mathcal{B}$
6. output $\mathcal{B}$
```

Figure 3: The recursive greedy algorithm to compute the GMCB of G. For $i = 1, 2$, $\mathcal{B}'_i$ is the set of cycles of $\mathcal{B}_i$ not containing any vertices of $\mathcal{H}(V_\mathcal{J})$.

contained in $\overline{ext}(C)$. We say that C *splits* R into R_1 and R_2. We call R_1 the *internal region* and R_2 the *external region* (w.r.t. R and C). Figure 4 gives an illustration.

The following lemma relates the test in line 4 to the two regions generated by the split.

Lemma 4. *The condition in line 4 in the recursive greedy algorithm is satisfied if and only if C splits a region into two each of which contains at least one elementary face.*

Proof. Let R be the region containing C and suppose that C splits R into R_1 and R_2.

Consider two elementary faces of G separated by C. No cycle of $\mathcal{B}$ separates them if and only if the two faces belong to the same region. Hence, the condition in line 4 is satisfied if and only if C separates a pair of elementary faces both belonging to R. The latter is equivalent to the condition that there is an elementary face in R_1 and an elementary face in R_2. $\square$

5.1 Contracted and Pruned Dual Trees

Lemma 4 shows that if we can keep track of the number of elementary faces of G in regions during the course of the algorithm, then testing the condition in line 4 is easy: it holds if and only if the number of elementary faces of G in each of the two regions obtained by inserting C is at least one. In the following, we introduce so called contracted dual trees and pruned dual trees that will help us keep track of the necessary information. First, we need the following lemma.

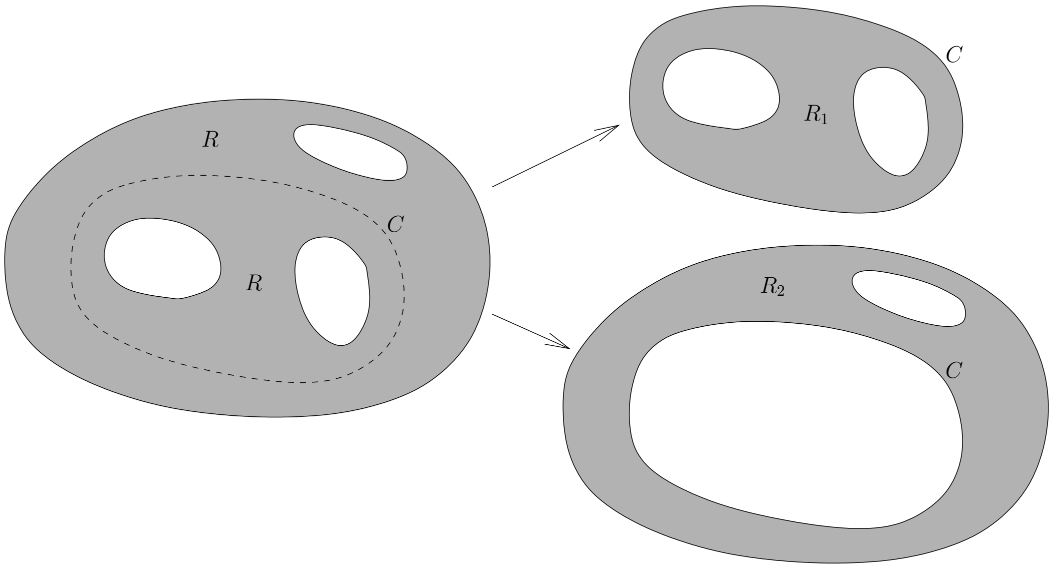


Figure 4: Adding a cycle C to $\mathcal{B}$ splits a region R into internal region R_1 and external region R_2.

Lemma 5. *Let H be a plane graph with non-negative edge weights and assume that shortest paths in H are unique. Let C be an isometric cycle in H and let P be a shortest path in H between vertices u and v. If both u and v belong to $\overline{int}(C)$ then P is contained in $\overline{int}(C)$. If both u and v belong to $\overline{ext}(C)$ then P is contained in $\overline{ext}(C)$.*

Proof. Suppose that $u, v \in \overline{int}(C)$ and assume for the sake of contradiction that P is not contained in $\overline{int}(C)$. Then there is a subpath P' of P between a vertex $u' \in C$ and a vertex $v' \in C$ such that P' is not contained in C. Since C is isometric, there is a shortest path P'' contained in C between u' and v'. But since P' is a subpath of P, P' is also a shortest path between u' and v'. Since $P' \neq P''$, this contradicts the uniqueness of shortest paths in H.

A similar proof holds when $u, v \in \overline{ext}(C)$. □

For a region R and a boundary vertex v belonging to R, the *contracted dual tree* $\tilde{T}_R(v)$ is the tree obtained from dual tree $\tilde{T}(v)$ by contracting each edge (u, u'), where u and u' are elementary faces in G both contained in the same non-elementary face of R, see Figure 5.

An important observation is that there is a one-to-one correspondence between the vertices of $\tilde{T}_R(v)$ and the faces of R. We assign the colour white

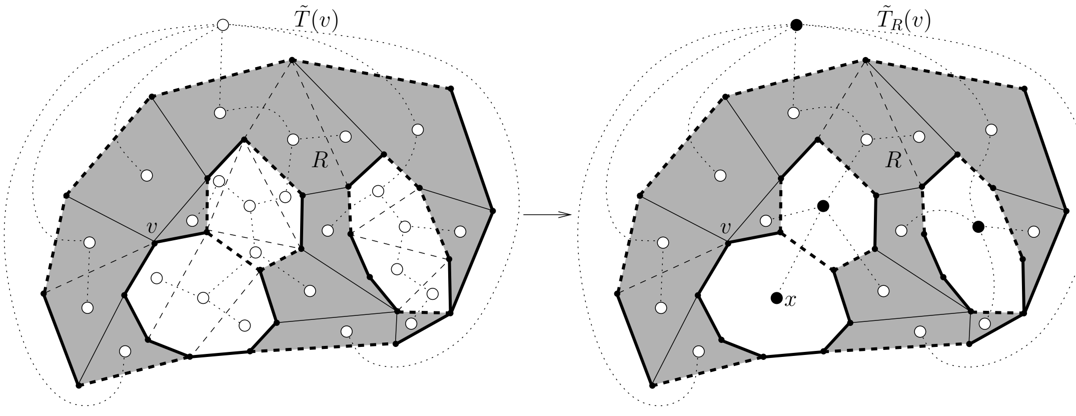


Figure 5: Contracted dual tree $\tilde{T}_R(v)$ is obtained from $\tilde{T}(v)$ by contracting edges between elementary faces belonging to the same non-elementary face (bold edges and white interior) of R. For this instance, applying the pruning procedure to obtain $\tilde{T}'_R(v)$ removes x and its adjacent edge in $\tilde{T}_R(v)$.

resp. black to those vertices of $\tilde{T}_R(v)$ corresponding to elementary resp. non-elementary faces of R, see Figure 5. We identify each edge in $\tilde{T}_R(v)$ with the corresponding edge in $\tilde{T}(v)$.

To ease the presentation of our ideas, we assume for now that only cycles from $\mathcal{H}(V_\mathcal{J})$ are encountered in line 3 of the recursive greedy algorithm. In Section 7, we show how to handle cycles from $\mathcal{B}'_1 \cup \mathcal{B}'_2$ as well.

So consider some iteration of the algorithm where a cycle $C = C(v,e) \in \mathcal{H}(V_\mathcal{J})$ has just been picked in line 3 and assume that all cycles added to $\mathcal{B}$ so far all belong to $\mathcal{H}(V_\mathcal{J})$. Cycle C should be added to $\mathcal{B}$ only if $\mathcal{B} \cup \{C\}$ is nested. We will now show how to detect whether this is the case using the contracted dual trees.

If there is a region R containing v such that $\tilde{T}_R(v)$ contains e then e (in G) belongs to R (since otherwise, e would have been contracted in $\tilde{T}_R(v)$). Since each cycle in $\mathcal{B}$ is isometric and since shortest paths are unique, Lemma 5 implies that $\mathcal{B} \cup \{C\}$ is nested. And the converse is also true: if $\mathcal{B} \cup \{C\}$ is nested then there is a region R containing C. In particular, R contains e so this edge must belong to $\tilde{T}_R(v)$.

It follows that detecting whether $\mathcal{B} \cup \{C\}$ is nested amounts to checking whether e is present in $\tilde{T}_R(v)$ for some region R.

Now, assume that $\mathcal{B} \cup \{C\}$ is nested (otherwise, we can discard C) and let us see how the contracted dual trees can help us check the condition in line 4 of the recursive greedy algorithm.

Define R to be the region containing C. Since e belongs to R, this edge belongs to the contracted dual tree $\tilde{T}_R(v)$. Let v_1 and v_2 be the end vertices of e in $\tilde{T}_R(v)$. Removing e from $\tilde{T}_R(v)$ splits this tree into two subtrees, one, $\tilde{T}_1$, attached to v_1 and one, $\tilde{T}_2$, attached to v_2. By Lemma 4, the condition in line 4 is satisfied if and only if $\tilde{T}_1$ and $\tilde{T}_2$ each contain at least one white vertex.

Unfortunately, both of these two subtrees may contain many black vertices so for performance reasons, a simple search in these trees to determine whether they contain white vertices is infeasible.

We therefore introduce *pruned (contracted) dual tree* $\tilde{T}'_R(v)$, defined as the subtree of $\tilde{T}_R(v)$ obtained by removing a black degree one vertex and repeating this procedure on the resulting tree until all degree one vertices are white, see Figure 5. We refer to this as the *pruning procedure.*

Lemma 6. *With the above definitions, $e \in \tilde{T}'_R(v)$ if and only if $\tilde{T}_1$ and $\tilde{T}_2$ both contain white vertices.*

Proof. If $\tilde{T}_1$ contains only black vertices then the pruning procedure will remove all vertices in $\tilde{T}_1$. In particular, the procedure removes v_1. Similarly, if $\tilde{T}_2$ contains only black vertices then v_2 is removed. In both cases, e is removed so $e \notin \tilde{T}'_R(v)$.

Conversely, if both $\tilde{T}_1$ and $\tilde{T}_2$ contain white vertices then the pruning procedure does not remove all vertices from $\tilde{T}_1$ and does not remove all vertices from $\tilde{T}_2$. Since no step of this procedure disconnects the resulting graph, it follows that neither v_1 nor v_2 is removed so $e \in \tilde{T}'_R(v)$. □

Lemma 6 shows that once $\tilde{T}'_R(v)$ is given, it is easy to determine whether both $\tilde{T}_1$ and $\tilde{T}_2$ contain white vertices and hence whether the condition in line 4 is satisfied: simply check whether $e \in \tilde{T}'_R(v)$.

Note that if line 4 is satisfied, $e \in \tilde{T}'_R(v)$ and hence $e \in \tilde{T}_R(v)$. By the above, this implies that $\mathcal{B} \cup \{C\}$ is nested. This shows that we only need $\tilde{T}'_R(v)$ to test the condition in line 4.

5.2 Inserting a Cycle

In the previous subsection, we introduced contracted and pruned dual trees and showed how the latter can be used to test the condition in line 4 of the recursive greedy algorithm for cycles in $\mathcal{H}(V_{\mathcal{J}})$. In this subsection, we show

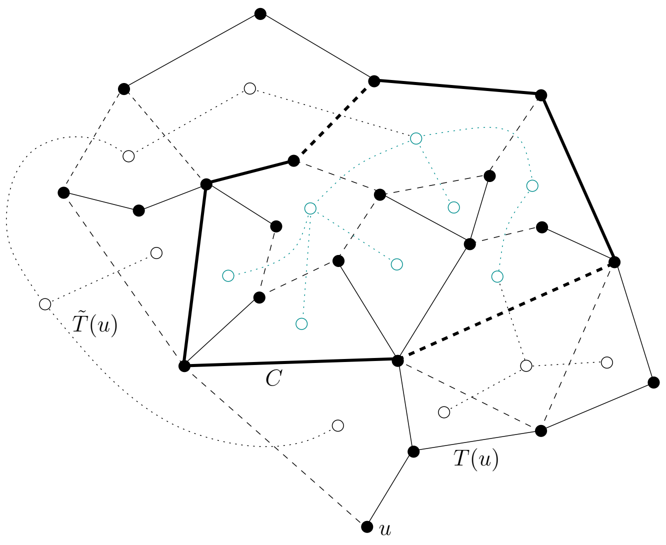


Figure 6: If $u \in \overline{ext}(C)$ then the subgraph of $\tilde{T}(u)$ in $\overline{int}(C)$ is a tree.

how to maintain regions and contracted and pruned dual trees when such cycles are added to $\mathcal{B}$ in line 5. We first need the following lemma.

Lemma 7. *Let C be an isometric cycle in G and let $u \in V$. If $u \in \overline{ext}(C)$ resp. $u \in \overline{int}(C)$ then the elementary faces of G in $\overline{int}(C)$ resp. in $\overline{ext}(C)$ are spanned by a subtree of $\tilde{T}(u)$. If $u \in \overline{ext}(C) \cap \overline{int}(C)$, i.e., $u \in C$, then these two subtrees are obtained by removing the single edge of $\tilde{T}(u)$ having one end vertex in $\overline{int}(C)$ and one end vertex in $\overline{ext}(C)$.*

Proof. Suppose that $u \in \overline{ext}(C)$, see Figure 6. Let F be the forest defined by the subgraph of shortest path tree $T(u)$ consisting of the edges belonging to $\overline{int}(C)$ and not to C. Since C is isometric and since shortest paths are unique, the edges of G belonging to $\overline{int}(C)$ and not to C or F define a connected component in the dual of G. Since all these edges belong to $\tilde{T}(u)$, it follows that the elementary faces of G in $\overline{int}(C)$ are spanned by a subtree of $\tilde{T}(u)$, as desired.

A similar argument shows that if $u \in \overline{int}(C)$ then the elementary faces of G in $\overline{ext}(C)$ are spanned by a subtree of $\tilde{T}(u)$.

Finally, assume that $u \in C$. There is at least one edge in $\tilde{T}(u)$ with one end vertex in $\overline{int}(C)$ and one end vertex in $\overline{ext}(C)$ since otherwise, $\tilde{T}(u)$ would be disconnected. There cannot be more than one such edge since that would contradict the first part of the lemma. This shows the second part. □

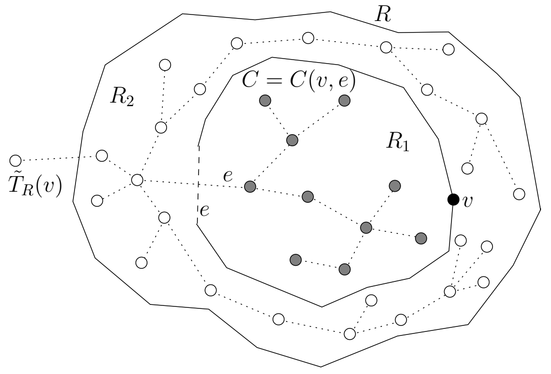


Figure 7: Faces of R belonging to R_1 resp. R_2 are identified by visiting the subtree of contracted dual tree $\tilde{T}_R(v)$ consisting of gray resp. white vertices.

Now, let us return to the problem of maintaining regions and contracted and pruned dual trees. Initially, $\mathcal{B} = \emptyset$ so the contracted and pruned dual trees are simply the dual trees $\tilde{T}(v)$ for each boundary vertex $v \in V_{\mathcal{J}}$. And there is only one region, namely the external region $R_\infty(\mathcal{B})$.

Now, suppose $C = C(v,e) \in \mathcal{H}(V_{\mathcal{J}})$ has just been inserted into $\mathcal{B}$ in line 5, see Figure 7. Let R be the region such that C splits R into internal region R_1 and external region R_2. We need to identify the faces of R belonging to R_1 and to R_2. This can be done with two searches in contracted dual tree $\tilde{T}_R(v)$. One search starts in the end vertex of e belonging to $\overline{int}(C)$ and avoids e (visiting the gray vertices in Figure 7). The other search starts in the end vertex of e belonging to $\overline{ext}(C)$ and also avoids e (visiting the white vertices in Figure 7). It follows from Lemma 7 and from the definition of contracted dual trees that the first search identifies the faces of R that should belong to R_1 and the second search identifies those that should belong to R_2.

We also need to form one new face for R_1, namely the face defined by $\overline{ext}(C)$. We denote this face by f_{R_1}. Similarly, we need to form a new face for R_2, defined by $\overline{int}(C)$, and we denote this face by f_{R_2}.

Next, we update contracted dual trees. The only ones affected are those of the form $\tilde{T}_R(u)$, where $u \in R$. There are three cases to consider: $u \in int(C)$, $u \in ext(C)$, and $u \in C$.

Case 1: Consider first a contracted dual tree $\tilde{T}_R(u)$ with $u \in int(C)$. Then $u \in R_1$ so we need to discard $\tilde{T}_R(u)$ and construct $\tilde{T}_{R_1}(u)$. We obtain the latter from the former by contracting all edges of $\tilde{T}_R(u)$ having both end vertices in $\overline{ext}(C)$ to a single vertex (this is possible by Lemma 7). We identify this new vertex with the new face f_{R_1} of R_1.

Case 2: Now, assume that $u \in ext(C)$. Then $u \in R_2$ so $\tilde{T}_R(u)$ should be replaced by $\tilde{T}_{R_2}(u)$. We do this by contracting all edges of $\tilde{T}_R(u)$ having both end vertices in $\overline{int}(C)$ to a single vertex (again, we make use of Lemma 7) and we identify this vertex with the new face f_{R_2} of R_2.

Case 3: Finally, assume that $u \in C$. Now, u belongs to both R_1 and R_2 so we need to discard $\tilde{T}_R(u)$ and construct $\tilde{T}_{R_1}(u)$ and $\tilde{T}_{R_2}(u)$. To do this, we first identify the edge e' in $\tilde{T}_R(u)$ having one end vertex u_1 in $\overline{int}(C)$ and one end vertex u_2 in $\overline{ext}(C)$. Then we construct the two trees T_1 and T_2 formed by removing e' from $\tilde{T}_R(u)$ with $u_1 \in T_1$ and $u_2 \in T_2$. We let T_1' be T_1 augmented with the edge from u_1 to f_{R_1} and let T_2' be T_2 augmented with the edge from u_2 to f_{R_2}.

It follows from Lemma 7 that T_1' is the contracted dual tree $\tilde{T}_{R_1}(u)$ for R_1 and that T_2' is the contracted dual tree $\tilde{T}_{R_2}(u)$ for R_2.

We have described how to update contracted dual trees when C is added to $\mathcal{B}$. We apply the same method to update pruned dual trees. The only difference is that the pruning procedure needs to be applied whenever a change is made to a pruned dual tree.

6 Implementation

Above, we gave an overall description of the algorithm when only cycles of $\mathcal{H}(V_{\mathcal{J}})$ are considered. We now go into more details and show how to give an efficient implementation of this algorithm. We start by describing the data structures that our algorithm makes use of. The main objects involved are regions, contracted dual trees, and pruned dual trees and we consider them in the following.

6.1 Regions

Associated with a region R is a *face list* $\mathcal{F}(R)$ which is a linked list containing the faces of R. An entry of $\mathcal{F}(R)$ corresponding to a face f is assigned the colour white resp. black if f is elementary resp. non-elementary. If it is black, it has a bidirected pointer to the child of R contained in f. This gives

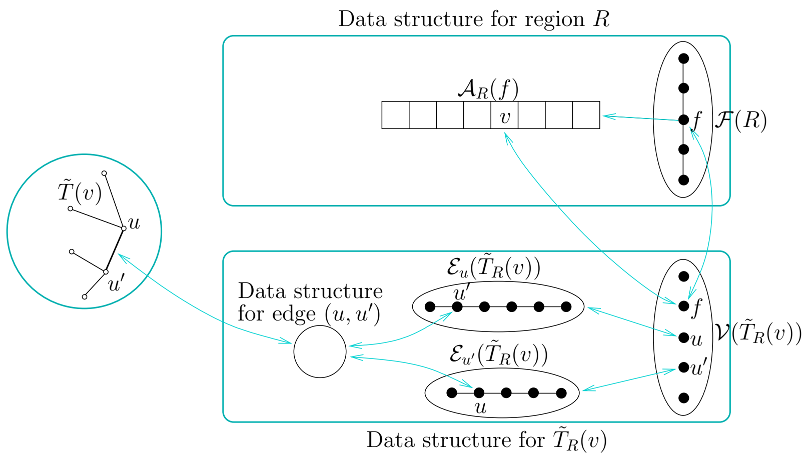

Data structure for region R
$\mathcal{A}_R(f)$
v
f
$\mathcal{F}(R)$
$\tilde{T}(v)$
u
u'
$\mathcal{E}_u(\tilde{T}_R(v))$
Data structure for edge (u, u')
u'
f
u
u'
$\mathcal{V}(\tilde{T}_R(v))$
$\mathcal{E}_{u'}(\tilde{T}_R(v))$
u
Data structure for $\tilde{T}_R(v)$

Figure 8: Illustration of data structures and some of their associated pointers.

a representation of the region tree $\mathcal{T}(\mathcal{B})$. If the entry is white, it points to the corresponding elementary face of G. The entry also points to the entire data structure for R.

Associated with the f-entry of $\mathcal{F}(R)$ is also an array $\mathcal{A}_R(f)$ with an entry for each boundary vertex in $V_{\mathcal{J}}$. The entry of $\mathcal{A}_R(f)$ for a boundary vertex v belonging to R has a bidirected pointer to vertex f in contracted dual tree $\tilde{T}_R(v)$, see Figure 8. It also has a bidirected pointer to vertex f in pruned dual tree $\tilde{T}'_R(v)$ if that vertex has not been deleted by the pruning procedure. All other entries of $\mathcal{A}_R(f)$ point to null.

6.2 Contracted and Pruned Dual Trees

Associated with a contracted dual tree $\tilde{T}_R(v)$ is a *vertex list* $\mathcal{V}(\tilde{T}_R(v))$ which is a linked list with an entry for each vertex of $\tilde{T}_R(v)$. The entry for a vertex u points to the entry of $\mathcal{F}(R)$ for the face of R corresponding to u. Associated with the u-entry of $\mathcal{V}(\tilde{T}_R(v))$ is also an *edge adjacency list* $\mathcal{E}_u(\tilde{T}_R(v))$, a linked list representing the edges adjacent to u in $\tilde{T}_R(v)$. Each list entry contains a pointer to the u-entry of vertex list $\mathcal{V}(\tilde{T}_R(v))$ (allowing us to find the head of $\mathcal{E}_u(\tilde{T}_R(v))$ in constant time) as well as a bidirected pointer to an *edge data structure*. The edge data structure thus contains two pointers, one for each of its end vertices. Furthermore, it contains a bidirected pointer to the corresponding edge in dual tree $\tilde{T}(v)$, see Figure 8.

We keep a similar data structure for pruned dual tree $\tilde{T}'_R(v)$. Both data structures need to support edge contractions, edge insertions, and edge deletions and the data structure for $\tilde{T}'_R(v)$ also needs to support the pruning procedure. We describe how to do this in the following.

6.2.1 Edge contraction

We only describe edge contractions for contracted dual trees since pruned dual trees can be dealt with in a similar way. Assume we have a set E_c of edges (or edge data structures) in $\tilde{T}_R(v)$ to be contracted to a single new vertex v_c and that these edges span a subtree of $\tilde{T}_R(v)$. We assume that we have a pointer to the entry of $\mathcal{F}(R)$ corresponding to v_c.

To contract an edge $e \in E_c$, we first remove the pointer to the edge of dual tree $\tilde{T}(v)$ corresponding to e. Traversing the two pointers associated with e, we find an entry in list $L_1 = \mathcal{E}_{u_1}(\tilde{T}_R(v))$ and an entry in list $L_2 = \mathcal{E}_{u_2}(\tilde{T}_R(v))$, where u_1 and u_2 are the end vertices of e in $\tilde{T}_R(v)$.

We remove those two entries in L_1 and L_2 and then merge the two lists to one list L since the new vertex is adjacent to edges adjacent to u_1 and u_2 except e. If L_1 is appended to the tail of L_2, we make every entry in L_1 point to the u_2-entry in vertex list $\mathcal{V}(\tilde{T}_R(v))$. Otherwise, we make every entry in L_2 point to the u_1-entry in that list. For performance reasons, we append the shorter of the two lists to the tail of the other.

We repeat the above for each edge of E_c and we end up with a single entry in $\mathcal{V}(\tilde{T}_R(v))$ representing the new vertex v_c. We make this entry point to the entry of $\mathcal{F}(R)$ corresponding to v_c and we update the pointer to the v-entry in the associated array.

How long does it take to contract edges? We will need the following lemma in our analysis (the proof can be found in the appendix).

Lemma 8. *Consider a set of objects, each assigned a positive integer weight. Let merge(o, o') be an operation that replaces two objects o and o' by a new object whose weight is the sum of the weights of o and o'. Assume that the time to execute merge(o, o') is bounded by the smaller weight of objects o and o'. Then repeating the merge-operation on pairs of objects in any order until at most one object remains takes $O(W \log W)$ time where W is the total weight of the original objects.*

Fix a $v \in V_{\mathcal{J}}$ and consider the set of contracted dual trees of the form $\tilde{T}_R(v)$ generated during the course of the algorithm. Each time a cycle from $\mathcal{H}(V_{\mathcal{J}})$ is added to $\mathcal{B}$, at most two new edges are inserted into trees of this form (case 3 in Section 5.2). Hence, there are $O(n)$ edges in total. It then follows easily from Lemma 8 and from the way we concatenate lists during edge contractions that the total time spent on edge contractions in all contracted dual trees of the form $\tilde{T}_R(v)$ is $O(n \log n)$. Since the number of choices of v is $O(\sqrt{n})$, we get a bound of $O(n^{3/2} \log n)$ time for all edge contractions performed by the algorithm.

6.2.2 Edge deletion

We also describe this only for contracted dual trees. So suppose we are to delete an edge $e = (u_1, u_2)$ from $\tilde{T}_R(v)$. We need to form two new trees, T_1 and T_2. Let T_1 be the tree containing u_1 and let T_2 be the tree containing u_2. For $i = 1, 2$, a simple search (say, depth-first) in $\tilde{T}_R(v)$ starting in u_i and avoiding e finds the vertices of T_i in time proportional to the size of this tree. By alternating between these two searches (i.e., essentially performing them in parallel), we can find the vertices of the smaller of the two trees in time proportional to the size of that tree.

Suppose that, say, T_1 is the smaller tree. Then we can form the two data structures for T_1 and T_2 in time proportional to the size of T_1: extract the entries of vertex list $\mathcal{V}(\tilde{T}_R(v))$ that should belong to T_1 and form a new vertex list containing these entries. The old data structure for $\tilde{T}_R(v)$ now becomes the new data structure for T_2 after the entries have been removed. We also need to remove the pointer between e and the corresponding edge in dual tree $\tilde{T}(v)$ and remove e from the edge adjacency lists but this can be done in constant time.

The following lemma, which is similar to Lemma 8, immediately implies that the total time for edge deletions is $O(n^{3/2} \log n)$ (the proof of the lemma is in the appendix).

Lemma 9. *Consider an object o with a positive integer weight W. Let split be an operation that splits an object of weight at least two into two new objects of positive integer weights such that the sum of weights of the two equals the weight of the original object. Assume that split runs in time proportional to the smaller weight of the two new objects. Then repeating the split-operation in any order, starting with object o, takes $O(W \log W)$ time.*

6.2.3 Edge insertion

The only situation where edge insertions are needed is in case 3 of Section 5.2. With our data structure, this can clearly be done in constant time per insertion.

6.2.4 Pruning procedure

Finally, let us describe how to implement the pruning procedure for pruned dual trees. Recall that this procedure repeatedly removes black degree one vertices until no such vertices exist.

We only need to apply the pruning procedure after an edge contraction and after an edge deletion (edge insertions are not needed in pruned dual trees since these edges will be removed by the pruning procedure). Let us only consider edge deletions since edge contractions are similar.

Consider a pruned dual tree $\tilde{T}'_R(v)$ and suppose the algorithm removes an edge $e = (u_1, u_2)$ from this tree. This forms two new trees T_1 and T_2, containing u_1 and u_2, respectively. In T_1, only u_1 can be a black degree one vertex since in $\tilde{T}'_R(v)$, no vertices had this property. Checking whether u_1 should be removed takes constant time. If it is removed, we repeat the procedure on the vertex that was adjacent to u_1. We apply the same strategy in T_2, starting in u_2.

The total time spent in the pruning procedure is proportional to the number of vertices removed. Since the number of vertices only decreases and since the initial number of vertices in all pruned dual trees is $O(n^{3/2})$, the total time spent by the pruning procedure is $O(n^{3/2})$.

6.3 The Algorithm

Having described the data structures involved and how they can support the basic operations that we need, let us show how to give an efficient implementation of our algorithm. Still, we only consider cycles from $\mathcal{H}(V_{\mathcal{J}})$ in the for-loop.

6.3.1 Initialization

First, we consider the initialization step. Applying the separator theorem of Miller gives us $\mathcal{J}$ and $V_{\mathcal{J}}$ in linear time. For each boundary vertex v, we need to compute shortest path tree $T(v)$ and shortest path distances from v in G.

This can be done in $O(n \log n)$ time with Dijkstra's algorithm for a total of $O(n^{3/2} \log n)$ time (in fact, a shortest path tree can be computed in linear time [13] but this will not improve the overall running time of our algorithm). We also need to compute dual trees $\tilde{T}(v)$ and this can easily be done in $O(n^{3/2})$ additional time. These dual trees are also the initial contracted and pruned dual trees. Since we need all three types of trees during the course of the algorithm, three copies of each dual tree are initialized.

The algorithm then recursively computes $\mathcal{B}_1$ and $\mathcal{B}_2$. It is assumed that the recursive calls also return the weights of cycles in these sets.

Our algorithm needs to extract $\mathcal{B}'_1$ and $\mathcal{B}'_2$ from these sets. This is done as follows. For every shortest path tree $T(v)$ that has been computed in recursive calls (we assume that these trees are kept in memory), we mark vertices of $T(v)$ belonging to $V_{\mathcal{J}}$. Then we mark all descendants of these vertices in $T(v)$ as well. Now, a Horton cycle $C(v, e)$ obtained by adding e to $T(v)$ contains a vertex of $V_{\mathcal{J}}$ if and only if at least one of the end vertices of e is marked. Since the total size of all recursively computed shortest path trees is bounded by the total space requirement which is $O(n^{3/2})$, it follows that $\mathcal{B}'_1$ and $\mathcal{B}'_2$ can be extracted from $\mathcal{B}_1$ and $\mathcal{B}_2$ in $O(n^{3/2})$ time.

The cycles in $\mathcal{B}'_1 \cup \mathcal{B}'_2 \cup \mathcal{H}(V_{\mathcal{J}})$ need to be sorted in order of non-decreasing weight. We are given the weights of cycles in $\mathcal{B}'_1 \cup \mathcal{B}'_2$ from the recursive calls and we can compute the weights of cycles in $\mathcal{H}(V_{\mathcal{J}})$ in a total of $O(n^{3/2})$ time using the shortest path distances computed above. Hence, sorting the cycles in $\mathcal{B}'_1 \cup \mathcal{B}'_2 \cup \mathcal{H}(V_{\mathcal{J}})$ can be done in $O(n^{3/2} \log n)$ time.

6.3.2 Testing condition in line 4

Next, we consider the for-loop of the algorithm for some cycle $C = C(v, e) \in \mathcal{H}(V_{\mathcal{J}})$. As we saw in Section 5.1, testing the condition in line 4 amounts to testing whether dual edge e in $\tilde{T}(v)$ is present in some pruned dual tree. Recall that we keep pointers between edges of dual trees and pruned dual trees. Since we remove a bidirected pointer between an edge data structure and the corresponding edge in a dual tree whenever it is contracted or deleted in a pruned dual tree, we can thus execute line 4 in constant time.

6.3.3 Inserting a cycle

Line 5 requires more work and we deal with it in the following. Suppose we are about to add the above cycle C to $\mathcal{B}$ in line 5. With the pointer associated with e, we find the corresponding edge data structure in a contracted dual tree $\tilde{T}_R(v)$. Traversing pointers from this data structure, we find the data structure for R in constant time. This region should be split into two new regions R_1 and R_2, where R_1 is the internal and R_2 the external region w.r.t. R and C. We need to identify the boundary vertices and the of faces in R that belong to R_1 and R_2, respectively.

6.3.4 Identifying boundary vertices in R_1 and R_2

We first identify the set $\delta(R)$ of boundary vertices of $V_{\mathcal{J}}$ belonging to R by traversing any one of the arrays $\mathcal{A}_R(f)$ associated with an entry of $\mathcal{F}(R)$ and picking the vertices not having null-pointers. This takes $O(\sqrt{n})$ time. Since the total number of times we add a cycle to $\mathcal{B}$ is $O(n)$, total time for this during the course of the algorithm is $O(n^{3/2})$.

We will extract three subsets from $\delta(R)$: the subset $\delta_{int}(R, C)$ of vertices belonging to $int(C)$, the subset $\delta_{ext}(R, C)$ belonging to $ext(C)$, and the subset $\delta(R, C)$ belonging to C.

If we can find these three subsets, we also obtain sets $\delta(R_1)$ and $\delta(R_2)$ of boundary vertices for R_1 and R_2, respectively, since $\delta(R_1) = \delta(R, C) \cup \delta_{int}(R, C)$ and $\delta(R_2) = \delta(R, C) \cup \delta_{ext}(R, C)$.

The following lemma bounds the time to find the three subsets. The proof is somewhat long and can be found in the appendix.

Lemma 10. *With the above definitions, we can find in $O(\sqrt{n})$ time the sets $\delta_{int}(R, C)$, $\delta_{ext}(R, C)$, and $\delta(R, C)$ with $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space for preprocessing.*

Lemma 10 implies that the total time spent on computing sets of boundary vertices over all regions generated by the algorithm is $O(n^{3/2})$ (plus $O(n^{3/2} \log n)$ time for preprocessing).

6.3.5 Identifying faces of R_1 and R_2

Having found the boundary vertices belonging to R_1 and R_2, we next focus on the problem of identifying the faces of R belonging to each of the two new regions.

As previously observed (see Figure 7), we can identify the faces of R_1 resp. R_2 with, say, a depth-first search in $\tilde{T}_R(v)$ starting in the end vertex of e belonging to $\overline{int}(C)$ resp. $\overline{ext}(C)$ and avoiding e. We use the edge adjacency lists to do this. By alternating between the two searches, we can identify the smaller set of faces in time proportional to the size of this set.

Let us assume that internal region R_1 contains this smaller set (the case where external region R_2 contains the set is similar). The search in $\tilde{T}_R(v)$ visited the entries of $\mathcal{V}(\tilde{T}_R(v))$ corresponding to faces in R_1. Since each such entry points to the corresponding entry in $\mathcal{F}(R)$, we can thus identify the faces in this face list that should belong to $\mathcal{F}(R_1)$.

We can extract these faces in time proportional to their number and thus form the face lists $\mathcal{F}(R_1)$ and $\mathcal{F}(R_2)$ in this amount of time. By reusing the arrays associated with entries of $\mathcal{F}(R)$, we do not need to form new arrays for $\mathcal{F}(R_1)$ and $\mathcal{F}(R_2)$. However, we need to set the pointers of some entries of these arrays to null. For R_1, the new null-pointers are those corresponding to boundary vertices of $\delta_{ext}(R, C)$ since these are the boundary vertices of R not belonging to R_1. And for R_2, the new null-pointers are those corresponding to boundary vertices of $\delta_{int}(R, C)$.

Since we index the arrays by boundary vertices, we can identify pointers to be set to null in constant time per pointer. Pointers that are set to null remain in this state so we can charge this part of the algorithm's time to the total number of pointers which is $O(n^{3/2})$.

We also need to associate a new face with the data structure for R_1 and for R_2 (i.e., faces f_{R_1} and f_{R_2} in Section 5.2). And we need to initialize an array for each of these two faces. This takes $O(\sqrt{n})$ time which is $O(n^{3/2})$ over all regions.

6.3.6 Contracted and pruned dual trees for R_1 and R_2

What remains is to construct contracted and pruned dual trees for R_1 and R_2. Due to symmetry, we shall only consider contracted dual trees. We have already given an overall description of how to do this in Section 5.2. As we showed,

1. for each $u \in \delta_{int}(R, C)$, we obtain $\tilde{T}_{R_1}(u)$ from $\tilde{T}_R(u)$ by contracting all edges belonging to $\overline{ext}(C)$,

2. for each $u \in \delta_{ext}(R, C)$, we obtain $\tilde{T}_{R_2}(u)$ from $\tilde{T}_R(u)$ by contracting all edges belonging to $\overline{int}(C)$, and

3. for each $u \in \delta(C)$, we obtain $\tilde{T}_{R_1}(u)$ and $\tilde{T}_{R_2}(u)$ from $\tilde{T}_R(u)$ by removing the unique edge in $\tilde{T}_R(u)$ having one end vertex in $\overline{int}(C)$ and one end

vertex in $\overline{ext}(C)$.

In Section 6.2, we described how to support edge contraction, edge deletion, and edge insertion such that the total time is $O(n^{3/2} \log n)$. The only detail missing is how to efficiently find the edges to be contracted or removed in the three cases above. We consider these cases separately in the following.

Case 1: Assume that $\delta_{int}(R, C) \neq \emptyset$ and let $u \in \delta_{int}(R, C)$. With a depth-first search in $\tilde{T}_R(v)$ as described above, we can identify all faces of R belonging to $\overline{ext}(C)$ in time proportional to the number of such faces. We can charge this time to the number of edges in $\tilde{T}_R(u)$ that are to be contracted.

For each such face f, we can mark the corresponding vertex in $\tilde{T}_R(u)$ by traversing the pointer associated with entry u of array $\mathcal{A}_R(f)$. Again, we can charge the time for this to the number of edges to be contracted.

Now, we need to contract all edges of $\tilde{T}_R(u)$ whose end vertices are both marked. In order to do this efficiently, we need to make a small modification to the contracted dual tree data structure in Section 6.2.

More precisely, we make the contracted dual trees rooted at some vertex. The choice of root is not important and may change during the course of the algorithm. What is important is that each non-root vertex now has a parent. By checking, for each marked non-root vertex whether its parent is also marked, we can identify the edges to be contracted in time proportional to the number of such edges. Of course this only works if the parent of a vertex can be obtained in constant time. Let us show how the contracted dual tree data structure can be adapted to support this.

Recall that each vertex of a contracted dual tree $\tilde{T}_R(v)$ is associated with an edge-adjacency list $\mathcal{E}_u(\tilde{T}_R(v))$ containing the edges adjacent to u in $\tilde{T}_R(v)$. We now require the edge from v to its parent (if defined) to be the located at the first entry of this list. This allows us to find parents in constant time.

How do we ensure that the parent edge is always located at the head of the list? This is not difficult after an edge insertion or deletion so let us focus on edge contractions. When an edge $e = (u_1, u_2)$ is contracted, either u_1 is the parent of u_2 or u_2 is the parent of u_1. Assume, say, the former. Then the parent of u_1 becomes the parent of the new vertex obtained by contracting e. When the two edge adjacency lists are merged, one of the two heads of the two old lists should thus be the head of the new list. This can easily be done in constant time.

Case 2: This case is similar to case 1.

Case 3: We need an efficient way of finding the unique edge e in $\tilde{T}_R(u)$ having one end vertex in $\overline{int}(C)$ and one end vertex in $\overline{ext}(C)$. We do as follows: first we mark the entries in $\mathcal{F}(R)$ corresponding to the set of faces of R belonging to $\overline{int}(C)$ or the set of faces of R belonging to $\overline{ext}(C)$. The set we choose to mark is the smaller of the two. We do this with "parallel" searches in $\tilde{T}_R(v)$ as described above, using time proportional to the number of marked faces.

We mark the corresponding vertices of $\tilde{T}_R(u)$ (using pointers from the arrays associated with entries of $\mathcal{F}(R)$). By Lemma 7, these form a subtree of $\tilde{T}_R(u)$ so we can find e by starting a search in any marked vertex of $\tilde{T}_R(u)$ and stopping once we encounter a vertex which is not marked. Then e is the last edge encountered in the search. This search also takes time proportional to the number of marked faces.

Hence, constructing the contracted and pruned dual trees for R_1 and R_2 takes time proportional to the number of marked faces. Lemma 9 then implies that the total time for this during the course of the algorithm is $O(n^{3/2}\log n)$.

Having constructed the contracted and pruned dual trees for R_1 and R_2, what remains before adding C to $\mathcal{B}$ is to add bidirected pointers between entries of the array associated with the new face in $\mathcal{F}(R_1)$ resp. $\mathcal{F}_(R_2)$ and the new vertex in the contracted/pruned dual tree for R_1 resp. R_2. Since the size of the array is $O(\sqrt{n})$, this can clearly be done in a total of $O(n^{3/2})$ time.

This concludes the description of the implementation of our algorithm. We have shown that it runs in $O(n^{3/2}\log n)$ time and requires $O(n^{3/2})$ space.

7 Recursively Computed Cycles

So far, we have assumed that only cycles from $\mathcal{H}(V_{\mathcal{J}})$ are encountered in line 3 of the recursive greedy algorithm. Now, we show how to deal with cycles from $\mathcal{B}'_1 \cup \mathcal{B}'_2$. In the following, we only consider $\mathcal{B}'_1$ since dealing with $\mathcal{B}'_2$ is symmetric.

The overall idea is the following. When a cycle $C \in \mathcal{H}(V_{\mathcal{J}})$ is added to $\mathcal{B}$, all cycles of $\mathcal{B}'_1$ that cross C are marked. If in the for-loop, a cycle $C \in \mathcal{B}'_1$ is picked, it is skipped if it is marked since the GMCB is nested by Lemma 1.

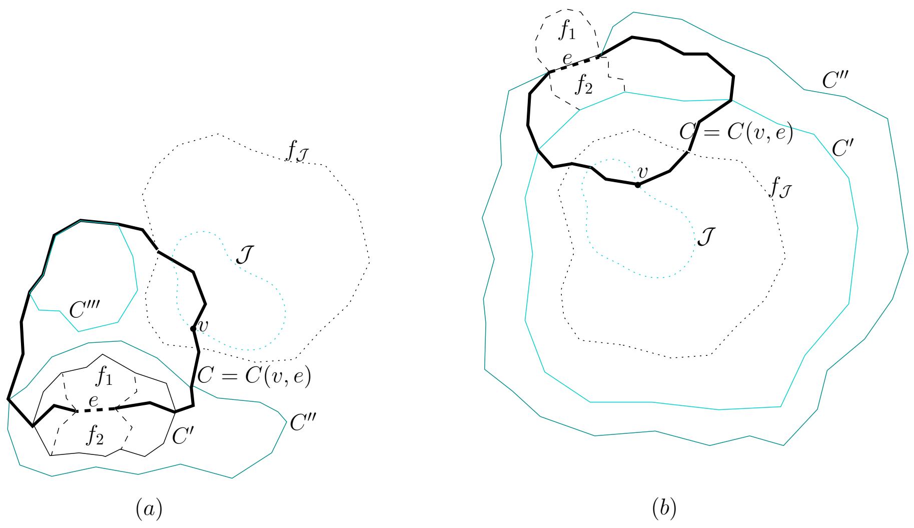


Figure 9: (a): Neither $R(C', \mathcal{B}_1)$, $R(C'', \mathcal{B}_1)$, nor $R(C''', \mathcal{B}_1)$ are ancestors of $R(f_{\mathcal{J}}, \mathcal{B}_1)$ and only $R(C', \mathcal{B}_1)$ and $R(C'', \mathcal{B}_1)$ are ancestors of both $R(f_1, \mathcal{B}_1)$ and $R(f_2, \mathcal{B}_1)$. Thus, C crosses C' and C'' and not C'''. (b): Both $R(C', \mathcal{B}_1)$ and $R(C'', \mathcal{B}_1)$ are ancestors of $R(f_{\mathcal{J}}, \mathcal{B}_1)$ and only $R(C', \mathcal{B}_1)$ is an ancestor of neither $R(f_1, \mathcal{B}_1)$ nor $R(f_2, \mathcal{B}_1)$. Thus, C crosses C' and not C''.

Otherwise, C must be fully contained in some region of the form $R(C', \mathcal{B})$, $C' \in \mathcal{B}$. Then C is added to $\mathcal{B}$ if and only if C separates a pair of elementary faces of $R(C', \mathcal{B})$.

We will assume that the recursive invocation of the algorithm in G_1 returns region tree $\mathcal{T}(\mathcal{B}_1)$ in addition to $\mathcal{B}_1$.

By applying Lemma 2, we see that every pair of elementary faces of G_1 is separated by some cycle of $\mathcal{B}_1$. Hence, each region associated with a vertex u of $\mathcal{T}(\mathcal{B}_1)$ contains exactly one elementary face of G_1 and we assume that the recursive call has associated this face with u. We let $R(f, \mathcal{B}_1)$ denote the region containing elementary face f.

We use the conditions in the following lemma to identify those cycles of $\mathcal{B}_1'$ that should be marked whenever a cycle of $\mathcal{H}(V_{\mathcal{J}})$ is added to $\mathcal{B}$.

Lemma 11. *Let $C = C(v, e) \in \mathcal{H}(V_{\mathcal{J}})$. If e does not belong to G_1 then C does not cross any cycle of $\mathcal{B}_1'$. Otherwise, let f_1 and f_2 be the elementary faces of G_1 adjacent to e and let $f_{\mathcal{J}}$ be the elementary face of G_1 containing $\mathcal{J}$. Then the set of cycles $C' \in \mathcal{B}_1'$ that C crosses are precisely those which satisfy one of the following two conditions:*

1. *$R(C', \mathcal{B}_1)$ is not an ancestor of $R(f_\mathcal{J}, \mathcal{B}_1)$ and is an ancestor of both $R(f_1, \mathcal{B}_1)$ and $R(f_2, \mathcal{B}_1)$ in $\mathcal{T}(\mathcal{B}_1)$ (Figure 9(a)),*

2. *$R(C', \mathcal{B}_1)$ is an ancestor of $R(f_\mathcal{J}, \mathcal{B}_1)$ and is an ancestor of neither $R(f_1, \mathcal{B}_1)$ nor $R(f_2, \mathcal{B}_1)$ in $\mathcal{T}(\mathcal{B}_1)$ (Figure 9(b)).*

The proof can be found in the appendix.

The next lemma will simplify the test in line 4 of the recursive greedy algorithm for $C \in \mathcal{B}_1'$. Again, the proof is in the appendix.

Lemma 12. *Suppose that in the recursive greedy algorithm, $C \in \mathcal{B}_1'$ is the cycle currently considered and assume that it does not cross any cycle of the partially constructed GMCB $\mathcal{B}$ of G. If $\mathcal{J} \subset ext(C)$ then all descendants of C in region tree $\mathcal{T}(\mathcal{B}_1)$ belong to the GMCB of G. If $\mathcal{J} \subset int(C)$ then all cycles of non-descendants of C in $\mathcal{T}(\mathcal{B}_1)$ belong to the GMCB of G.*

Now, we are ready to describe how the algorithm deals with cycles from $\mathcal{B}_1'$. Each cycle in this set is in one of three states: *active*, *passive*, or *cross* state.

Initially, all cycles in $\mathcal{B}_1'$ are active. When a cycle from $\mathcal{H}(V_\mathcal{J})$ is added to $\mathcal{B}$, Lemma 11 is applied to identify all cycles from $\mathcal{B}_1'$ that cross this cycle. These cycles have their state set to the cross state.

When the algorithm encounters a cycle $C \in \mathcal{B}_1'$ in the for-loop, C is skipped if it is in the cross state.

If C is active, it is completely contained in some region R. There are two cases to consider: $\mathcal{J} \subset ext(C)$ and $\mathcal{J} \subset int(C)$. We assume that $\mathcal{J} \subset ext(C)$ since the case $\mathcal{J} \subset int(C)$ is similar. We need to determine whether C should be added to $\mathcal{B}$. By Lemma 4, this amounts to checking whether there are two elementary faces of R which are separated by C. By Lemma 12, we know that the elementary faces of R belonging to $\overline{int}(C)$ are exactly the elementary faces of the region R' in $\overline{int}(C)$ that was generated when C was added to $\mathcal{B}_1$ during the recursive call for G_1.

Hence, we add C to $\mathcal{B}$ if and only if the number of elementary faces in R is strictly larger than the number of elementary faces in R'.

If C is added to $\mathcal{B}$, region R is split into two smaller regions. Let R_1 be the internal region and let R_2 be the external region. Since $\mathcal{J} \subset ext(C)$, Lemma 12 implies that the cycles belonging to $\overline{int}(C)$ that are added to $\mathcal{B}$ during the course of the algorithm are exactly C and its descendants in $\mathcal{T}(\mathcal{B}_1)$. We therefore do not need to maintain R_1 or any regions contained in $\overline{int}(C)$.

Instead, we make all descendants of C in $\mathcal{T}(\mathcal{B}_1)$ passive. When a passive cycle is encountered by the algorithm, there is no need to update regions or contracted or pruned dual trees and the cycle is simply added to $\mathcal{B}$.

Now, let us consider R_2. In order to obtain this region, we replace all faces of R belonging to $\overline{int}(C)$ with a single new face defined by $\overline{int}(C)$. And we contract all edges in $\overline{int}(C)$ to a single black vertex in all contracted and pruned dual trees for R.

This completes the description of the extension of our algorithm that deals with cycles from $\mathcal{B}_1' \cup \mathcal{B}_2'$.

7.1 Implementation

Let us show how to give an efficient implementation of the above algorithm for cycles from $\mathcal{B}_1' \cup \mathcal{B}_2'$. Due to symmetry, we may restrict our attention to $\mathcal{B}_1'$ in the following.

7.1.1 Identifying cross state cycles

The first problem is to identify the cycles of $\mathcal{B}_1'$ that should be in the cross state when a cycle $C = C(v, e) \in \mathcal{H}(V_{\mathcal{J}})$ is added to $\mathcal{B}$.

To solve this problem, we apply Lemma 11. Checking whether e belongs to G_1 takes constant time. If e is not an edge of G_1 then no new cycles will be in the cross state. Otherwise, we obtain in constant time the elementary faces f_1 and f_2 adjacent to e in G_1 since these are the end vertices of e in the dual of G_1.

We assume that we can compute lowest common ancestors in $\mathcal{T}(\mathcal{B}_1)$ efficiently. We can use the data structure of Harel and Tarjan [11] for this.

Let a_1 be the lowest common ancestor of $R(f_1, \mathcal{B}_1)$ and $R(f_2, \mathcal{B}_1)$ in $\mathcal{T}(\mathcal{B}_1)$, see Figure 10. Let a_2 be the lowest common ancestor of $R(f_1, \mathcal{B}_1)$ and $R(f_{\mathcal{J}}, \mathcal{B}_1)$. Let a_3 be the lowest common ancestor of $R(f_2, \mathcal{B}_1)$ and $R(f_{\mathcal{J}}, \mathcal{B}_1)$. Finally, let P be the path in $\mathcal{T}(\mathcal{B}_1)$ containing $R(f_{\mathcal{J}}, \mathcal{B}_1)$ and its ancestors.

A cycle $C' \in \mathcal{B}_1'$ satisfies the first condition in Lemma 11 if and only if it is not associated with a vertex on P and if it is associated with a_1 or an ancestor of a_1 (Figure 10(a)). And it satisfies the second condition if and only if it is associated with a vertex on P and not with a_2, a_3, or an ancestor of either of these two vertices (Figure 10(b)).

To identify cycles that satisfy the first condition, we start at a_1 and walk upwards in $\mathcal{T}(\mathcal{B}_1)$, marking cycles as we go along. The process stops when a

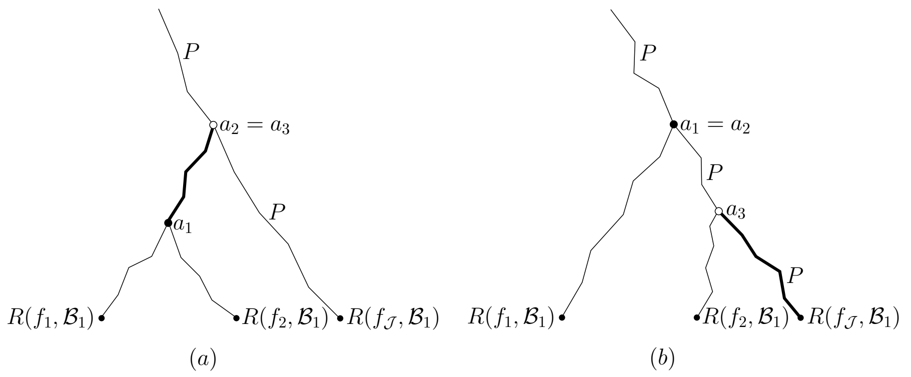


Figure 10: (a): Cycles associated with a_1 or an ancestor of a_1 and not with a vertex on P are exactly those that satisfy the first condition in Lemma 11. (b): Cycles associated with a vertex on P and not with a_2, a_3, or an ancestor of either a_2 or a_3 are exactly those satisfying the second condition in Lemma 11.

vertex on P is reached.

To identify cycles satisfying the second condition, we instead move upwards in $\mathcal{T}(\mathcal{B}_1)$ along P, starting in $R(f_{\mathcal{J}}, \mathcal{B}_1)$. We stop when the root of $\mathcal{T}(\mathcal{B}_1)$ or when a_2 or a_3 is reached.

Although this strategy works, it is slow since the same cycles may be considered several times during the algorithm. To remedy this, we first observe that when identifying cycles associated with vertices from a_1 to P, we may stop if we encounter a cycle that is already in the cross state since then all its ancestors which are not on P must also be in this state.

Next, we observe that when identifying cycles associated with vertices on P, we always consider them from bottom to top. Hence, by keeping track of the bottommost b vertex on P whose associated cycle is not in the cross state, we can start the next traversal of P from b. If the cycle associated with a_2 or with a_3 is already in the cross state, we need not consider any vertices. Otherwise, we walk upwards in P from b, changing the state of cycles to the cross state and stop if a_2 or a_3 is reached.

It follows that we can identify cycles satisfying one of the two conditions and change their state in time proportional to the number of cycles whose state changes as a result of this. Hence, the total time for this is bounded by the size of $\mathcal{T}(\mathcal{B}_1)$ which is linear.

7.1.2 Testing condition in line 4

In the following, let C be an active or passive cycle in $\mathcal{B}'_1$ just encountered by our algorithm. We will assume that $\mathcal{J} \subset ext(C)$. The case $\mathcal{J} \subset int(C)$ is similar.

We first need to determine whether C should be added to $\mathcal{B}$. This is trivial if C is passive since passive cycles should always be added. And as noted in Section 7, no pruned dual trees need to be updated after the insertion of a passive cycle.

So assume that C is active. Let R be the region containing C and let R' be the region in $\overline{int}(C)$ that was generated when C was added to $\mathcal{B}_1$ during the construction of the GMCB of G_1. As we showed above, determining whether C should be added to $\mathcal{B}$ amounts to checking whether the number of elementary faces in R is strictly larger than the number of elementary faces in R'.

We can easily extend our region data structure to keep track of the number of elementary faces in each region without increasing the time and space bounds of our algorithm. By recording this information for R' during the recursive call for G_1, it follows that we can determine in constant time whether R contains more elementary faces than R'.

Of course, this only works if we can quickly identify R and R'. Identifying R' is simple since this region is associated with the vertex v_C of region tree $\mathcal{T}(\mathcal{B}_1)$ associated with C.

To identify R, let R_{v_C} be the region associated with v_C in $\mathcal{T}(\mathcal{B}_1)$. Since $\mathcal{B}_1$ is the GMCB of G_1, each pair of elementary faces of G_1 is separated by some cycle of $\mathcal{B}_1$. It follows that R_{v_C} contains exactly one elementary face f_{v_C} of G_1. We may assume that this face was associated with v_C during the construction of $\mathcal{B}_1$ so that we can obtain this face in constant time from v_C.

Face f_{v_C} is also an elementary face in G and it belongs to R. Recall from Section 6.1 that there is a bidirected pointer between R and each elementary face of G belonging to R. Hence, we can obtain R from f_{v_C} in constant time

It follows from the above that we can check if C should be added to $\mathcal{B}$ in constant time.

7.1.3 Inserting a cycle

Now, suppose C should be inserted into $\mathcal{B}$. We first make cycles of $\mathcal{B}'_1$ passive according to Lemma 12. This can be done with, say, a depth-first search in

$\mathcal{T}(\mathcal{B}_1)$ starting in the vertex v_C of $\mathcal{T}(\mathcal{B}_1)$ associated with C and visiting descendants of this vertex. The search stops when a vertex associated with a passive cycle is encountered. Each search identifies the vertices of $\mathcal{T}(\mathcal{B}_1)$ that are associated with cycles whose state changes from non-passive to passive. And since we stop a search when a passive cycle is encountered, all searches take total time proportional to the size of $\mathcal{T}(\mathcal{B}_1)$ which is $O(n)$.

Next, we need to update regions and contracted and pruned dual trees. Let R_1 be the internal region and let R_2 be the external region w.r.t. R and C. As we showed in the overall description of the algorithm, the only problem that we need to consider is how to construct R_2 and its contracted and pruned dual trees. We showed that this amounts to replacing all faces of R belonging to $\overline{int}(C)$ with a single new face defined by $\overline{int}(C)$ and to contract all edges in $\overline{int}(C)$ to a single black vertex in all contracted and pruned dual trees for R.

We will show how to find the faces of R belonging to $\overline{int}(C)$ in time proportional to their number. Applying the charging schemes introduced in Section 6.3, this will suffice to prove the desired time and space bounds for the entire algorithm.

Identifying non-elementary faces: Consider an active cycle C' associated with a descendant u of the vertex v_C of $\mathcal{T}(\mathcal{B}_1)$ associated with C. If C' was previously considered in the for-loop of our algorithm, it must have been added to $\mathcal{B}$ (by Lemma 12). This implies that $\overline{int}(C')$ must be a non-elementary face of R since otherwise, C' would be passive, see Figure 11. The converse holds as well: any non-elementary face of R belonging to $\overline{int}(C)$ is realized by $\overline{int}(C')$ for such a cycle C' previously considered in the for-loop.

It follows that we can find all non-elementary faces of R belonging to $\overline{int}(C)$ by identifying the active descendants of C in $\mathcal{T}(\mathcal{B}_1)$ that have already been considered in the for-loop. Since all active cycles associated with descendants of C are to become passive, we can charge the time for finding these faces to the number of cycles whose state changes from active to passive.

Identifying elementary faces: What remains is to identify the elementary faces of R belonging to $\overline{int}(C)$. Recall that we have associated with each vertex u of $\mathcal{T}(\mathcal{B}_1)$ the unique elementary face of G_1 contained in the region associated with u. Vertex v_C and its descendants in $\mathcal{T}(\mathcal{B}_1)$ are thus associated with exactly the elementary faces of G_1 belonging to $\overline{int}(C)$. These

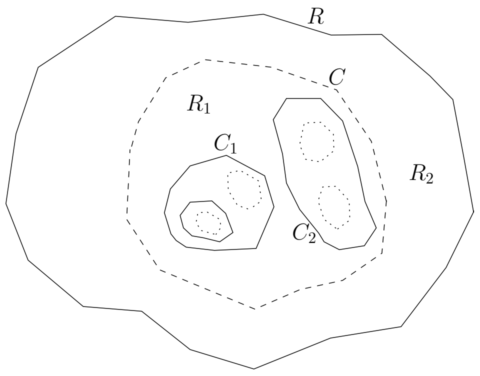


Figure 11: Solid cycles belong to the partially constructed GMCB $\mathcal{B}$ of G. All descendants of vertices of $\mathcal{T}(\mathcal{B}_1)$ associated with solid cycles are passive and thus do not define faces of R. For this instance, C_1 and C_2 define the non-elementary faces of R.

faces are also elementary faces in G.

It follows that the elementary faces of R belonging to $\overline{int}(C)$ are associated with exactly the descendants of C corresponding to active cycles not already considered by the algorithm. Using the same charging scheme as above, we can also identify these faces within the required time and space bounds.

We have shown how to implement the entire recursive greedy algorithm to run in $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space and we can conclude this section with the main result of our paper.

Theorem 2. *Given an n-vertex planar, undirected graph $G = (V, E)$ with non-negative edge weights, the following implicit representation of the GMCB $\mathcal{B}$ of G can be computed in $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space:*

1. *a set of trees $T_1, \ldots, T_k$ in G rooted at vertices $v_1, \ldots, v_k$, respectively,*

2. *a set of triples (i, e, w) representing the cycles in $\mathcal{B}$, where $i \in \{1, \ldots, k\}$, $e = (u, v) \in E \setminus E_{T_i}$, and $w \in \mathbb{R}$, where u and v are vertices in T_i. The pair (i, e) represents the cycle in $\mathcal{B}$ formed by concatenating e and the two paths in T_i from v_i to u and from v_i to v, respectively. The value of w is the weight of this cycle,*

3. *the region tree $\mathcal{T}(\mathcal{B})$ where each vertex points to the associated region,*

4. *a set of regions. Each region is associated with the unique elementary face of G contained in that region. Each internal region $R(C, \mathcal{B})$ is associated with the triple representing C.*

8 Corollaries

In this section, we present results all of which follow from Theorem 2. The first is an output-sensitive sensitive algorithm for computing an MCB.

Corollary 1. *A minimum cycle basis of an n-vertex planar, undirected graph with non-negative edge weights can be computed in $O(n^{3/2} \log n + C)$ time and $O(n^{3/2} + C)$ space, where C is the total length of all cycles in the basis.*

Proof. Follows immediately from Theorem 2. □

A stronger result holds when the graph is unweighted:

Corollary 2. *A minimum cycle basis of an n-vertex planar undirected, unweighted graph can be computed in $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space.*

Proof. Let G be an n-vertex planar, undirected, unweighted graph. The internal elementary faces of G define a cycle basis of of G of total length $O(n)$. Hence, since G is unweighted, an MCB of G has total length $O(n)$. The result now follows from Corollary 1. □

Since the all-pairs min cut problem is dual equivalent to the MCB problem for planar graphs, we also get the following two results.

Corollary 3. *All-pairs min cuts of an n-vertex planar, undirected graph with non-negative edge weights can be computed in $O(n^{3/2} \log n + C)$ time and $O(n^{3/2} + C)$ space, where C is the total length of the cuts.*

Proof. Let G be an n-vertex planar, undirected graph with non-negative edge weights. As shown in [12], if G is connected, we can solve the APMCP for G by solving the MCBP for the dual G^* of G.

We may assume that G is connected since otherwise, we can consider each connected component separately. We cannot immediately solve the MCBP for G^* since this is a multigraph. But we can avoid an edge of the form (u, u) by splitting it into two edges (u, v) and (v, u) whose sum of weights equal the weight of (u, u). And we can avoid multi-edges in a similar way. Let

G' be the resulting planar graph. It is easy to see that G' has size $O(n)$. Furthermore, an MCB $\mathcal{B}$ of G' can be transformed into an MCB of G^* in time proportional to the total size of cycles in $\mathcal{B}$. The result now follows from Corollary 1. □

Corollary 4. *All-pairs min cuts of an n-vertex planar, undirected, unweighted graph can be computed in $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space.*

Proof. This result is easily obtained by combining ideas in the proofs of Corollary 2 and Corollary 3. □

Next, we present our subquadratic time and space algorithm for finding the weight vector of a planar graph.

Corollary 5. *The weight vector of an n-vertex planar, undirected graph with non-negative edge weights can be computed in $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space.*

Proof. From Theorem 2, we obtain an implicit representation of the GMCB $\mathcal{B}$ for the input graph. We then compute the weights of all cycles in $\mathcal{B}$ using linear time and space. Sorting them takes $O(n \log n)$ time. This gives the weight vector of the input graph in a total of $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space. □

From Theorem 2, we also obtain a faster algorithm for computing a Gomory-Hu tree of a planar graph.

Corollary 6. *A Gomory-Hu tree of an n-vertex connected, planar, undirected graph with non-negative edge weights can be computed in $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space.*

Proof. The following algorithm constructs a Gomory-Hu tree for G [26]: a tree T spanning a collection of vertex sets $S_1, \ldots, S_t$ is maintained, starting with $S_1 = V$. At each step, a set S_i is picked such that $|S_i| > 1$ and any two distinct vertices $u, v \in S_i$ are chosen. Set S_i is then regarded as the root of T and each subtree of T, i.e., each tree in $T \setminus \{S_i\}$, is collapsed into a single supernode. A min u-v cut in the resulting graph is found, partitioning V into two subsets, V_1 and V_2, where $u \in V_1$ and $v \in V_2$. Tree T is then modified by splitting S_i into two vertices, S_{i_1} and S_{i_2}, where $S_{i_1} = S_i \cap V_1$ and $S_{i_2} = S_i \cap V_2$. The two vertices are connected by a new edge whose weight equals the size of the min cut found. Finally, each subtree of the old T is connected to S_{i_1} if the corresponding supernode was in the same partition as u in the cut. Otherwise, the subtree is connected to S_{i_2}.

Let us show how to implement this algorithm to obtain the desired time and space bounds. We first apply Theorem 2 to the dual G^* of G, giving an implicit representation of the GMCB of G^*. By Lemma 2, each cycle C in this basis is a minimum-weight cycle that separates some pair of faces f_1 and f_2 in G^*. Let u_1 and u_2 be the vertices of G corresponding to f_1 and f_2, respectively. By duality of the GMCBP and the APMCP [12], the edges of C are the edges of a min u_1-v_1 cut in G of weight equal to the weight of C.

Now, pick any cycle C in the GMCB of G^*. As the initial min cut in the Gomory-Hu tree algorithm, we pick the one corresponding to C. This separates the initial set $S_i = S_1 = V$ into two sets S_{i_1} and S_{i_2}, where S_{i_1} is the set of vertices of G corresponding to faces of G^* in $\overline{int}(C)$ and S_{i_2} is the set of vertices of G corresponding to faces of G^* in $\overline{ext}(C)$. Now, T consists of vertices S_{i_1} and S_{i_2} and an edge (S_{i_1}, S_{i_2}). The weight of this edge is equal to the weight of C. Since we are given the weight of C from Theorem 2, we can this obtain the weight of edge (S_{i_1}, S_{i_2}) in constant time.

Note that for each pair of vertices u and v in S_{i_1}, there is a min u-v cut defined by a cycle of $\mathcal{B}$ which is a descendant of C in $\mathcal{T}(\mathcal{B})$. And for each pair of vertices u and v in S_{i_2}, there is a min u-v cut defined by a cycle of $\mathcal{B}$ which is a non-descendant of C in $\mathcal{T}(\mathcal{B})$.

Hence, we have separated our problem in two and we can recursively compute the Gomory-Hu tree for G by splitting region tree $\mathcal{T}(\mathcal{B})$ in two at each recursive step. The recursion stops when we obtain a set S_i of size one. At this point, we obtain the elementary face f of G^* correponding to the vertex in S_i using part four of Theorem 2. The vertex of G corresponding to f in G^* is then the unique vertex in S_i.

Let us analyze the running time of this algorithm. Applying Theorem 2 takes $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space. Note that in the algorithm above, we do not need to compute the vertices in the S_i-sets until they have size one. So each step of the algorithm, where the current S_i-set has size greater than one, can be implemented to run in constant time. And we can also execute a step where $|S_i| = 1$ in constant time using the fourth part of Theorem 2 to find the vertex in S_i.

Since the GMCB of G^* contains $O(n)$ cycles, it follows that the algorithm runs in linear time and space, in addition to the time and space in Theorem 2. $\square$

Finally, we present our oracle for answering min cut queries.

Corollary 7. *Let G be an n-vertex planar, undirected graph with non-negative edge weights. With $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space for preprocessing, the weight of a min cut between any two given vertices of G can be reported in constant time. The cut itself can be reported in time proportional to its size.*

Proof. We may assume that G is connected since otherwise, we can consider each connected component separately. We first construct a Gomory-Hu tree T of G. By Corollary 6, this can be done in $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space. By definition of Gomory-Hu trees, answering the query for the weight of a min cut between two vertices u and v of G reduces to answering the query for the minimum weight of an edge on the simple path between u and v in T.

It is well-known that any tree with m vertices has a vertex v such that the tree can be split into two subtrees, each rooted at v and each containing between $m/4$ and $3m/4$ vertices. Furthermore, this separator can be found in linear time.

We find such a separator in T and recurse on the two subtrees. We stop the recursion at level $\log(\sqrt{n})$. The total time for this is $O(n\log n)$.

Let $\mathcal{S}$ be the subtrees at level $\log(\sqrt{n})$. We observe that these trees are edge-disjoint and their union is T. Furthermore, $|\mathcal{S}| = O(\sqrt{n})$ and each subtree has size $O(\sqrt{n})$. The *boundary vertices* of a subtree $S \in \mathcal{S}$ are the vertices that S shares with other subtrees in $\mathcal{S}$. Vertices of S that are not boundary vertices are called *interior vertices* of S. We let B be the set of boundary vertices over all subtrees in $\mathcal{S}$. It is easy to see that $|B| = O(\sqrt{n})$.

For each boundary vertex $b \in B$, we associate an array with an entry for each vertex of T. The entry corresponding to a vertex $v \neq b$ contains the edge of minimum weight on the simple path in T between b and v.

Since $|B| = O(\sqrt{n})$, it follows easily that we can construct all these arrays and fill in their entries in a total of $O(n^{3/2})$ time and space. This allows us to answer queries in T in constant time when one of the two vertices belongs to B.

We associate each vertex v of T not belonging to B with the unique subtree S_v in $\mathcal{S}$ containing v as an interior vertex.

Associated with v is also an array with an entry for each $S \in \mathcal{S} \setminus \{S_v\}$. This entry contains the vertex b of B belonging to S_v such that any path from v to S contains b. Note that for any other vertex v' of S_v, any path from v' to S also contains b. From this observation and from the fact that $|\mathcal{S}| = O(\sqrt{n})$, it follows that we can compute the arrays associated with interior vertices in all subtrees using a total of $O(n^{3/2})$ time and space.

Finally, we associate with v an array with an entry for each vertex v' of S_v. This entry contains the edge of minimum weight on the simple path in S_v from v to v'. Since S_v has size $O(\sqrt{n})$, the entries in this array can be computed in $O(\sqrt{n})$ time. Over all interior vertices of all subtrees of $\mathcal{S}$, this is $O(n^{3/2})$ time.

Now, let us describe how to answer a query for vertices u and v in T. In constant time, we find the subtrees $S_u, S_v \in \mathcal{S}$ such that $u \in S_u$ and $v \in S_v$. If $S_u = S_v$ or if u or v belongs to B, we can answer the query in constant time with the above precomputations.

Now, assume that $S_u \neq S_v$ and that u and v are interior vertices. We find the boundary vertex b of S_u such that any path from u to R_v contains b. Let P_1 be the simple path in S_u from u to b and let P_2 be the simple path in T from b to v. For $i = 1, 2$, the above precomputations allow us to find the least-weight edge e_i on P_i in constant time. Let e be the edge of smaller weight among e_1 and e_2. Returning the weight of e then answers the query in constant time.

To show the last part of the corollary, observe that when the weight of edge e is output by the above algorithm, the set of edges in the corresponding cut is defined by a cycle C_e in the GMCB $\mathcal{B}$ of G^*. During the construction of Gomory-Hu tree T (see Corollary 6), we can associate e with the implicit representation of C_e from Theorem 2. Hence, given e, we can output C_e in time proportional to its size. This completes the proof. $\square$

9 Obtaining Lex-Shortest Path Trees

Let $w : E \rightarrow \mathbb{R}$ be the weight function on the edges of G. In Section 4, we assumed uniqueness of shortest path in G between any two vertices w.r.t. w. We now show how to avoid this assumption. We assume in the following that the vertices of G are given indices from 1 to n.

By results in [12], there is another weight function w' on the edges of G such that for any pair of vertices in G, there is a unique shortest path between them w.r.t. w' and this path is also a shortest path w.r.t. w. Furthermore, for two paths P and P' between the same pair of vertices in G, $w'(P) < w'(P)$ exactly when one of the following three conditions is satisfied:

1. P is strictly shorter than P' w.r.t. w,
2. P and P' have the same weight w.r.t. w and P contains fewer edges than P',
3. P and P' have the same weight w.r.t. w and the same number of edges and the smallest index of vertices in $V_P \setminus V_{P'}$ is smaller than the smallest index of vertices in $V_{P'} \setminus V_P$.

A shortest path w.r.t. w' is called a *lex-shortest path* and a shortest path tree w.r.t. w' is called a *lex-shortest path tree*.

The properties of w' allow us to apply this function instead of w in our algorithm. What we need to describe is how to compute lex-shortest paths efficiently.

As shown in [12], lex-shortest paths between all pairs of vertices in G can be computed in $O(n^2 \log n)$ time. We need something faster. In the following, we show a stronger result, namely how to compute a lex-shortest path tree in $O(n \log n)$ time. Since we only need to compute shortest paths from $O(\sqrt{n})$ boundary vertices at the top-level of the recursion, this will give a total time bound of $O(n^{3/2} \log n)$.

We also need to find lex-shortest path trees in subgraphs of G when recursing and we need to compute them w.r.t. the same weight function w'. By the above, this can be achieved simply by keeping the same indices for vertices in all recursive calls.

Now, let $s \in V$ be given and let us show how to compute the lex-shortest path tree in G with source s in $O(n \log n)$ time.

We first use a small trick from [12]: for function w, a sufficiently small $\epsilon > 0$ is added to the weight of every edge. This allows us to disregard the second condition above. When comparing weights of paths, we may treat ϵ symbolically so we do not need to worry about precision issues.

We will apply Dijkstra's algorithm with a few additions which we describe in the following. We keep a queue of distance estimates w.r.t. w as in the standard implementation. Now, consider any point in the algorithm. Let d be the distance estimate function. Consider an unvisited vertex v with current distance estimate $d[v] < \infty$ and predecessor vertex p.

Suppose that at this point, the algorithm extracts a vertex p' from Q which is adjacent to v in G and suppose that $d[p'] + w(p', v) = d[v]$. The central problem is to decide whether v should keep p as its predecessor or get p' as its new predecessor. In the following, we show how to decide this in $O(\log n)$ time. This will suffice to give an $O(n \log n)$ time algorithm that computes the lex-shortest path tree in G with source s.

Let P be the path in the partially constructed lex-shortest path tree T from s to p followed by edge (p, v). Let P' be the path in T from s to p' followed by edge (p', v). Note that P and P' both have weight $d[v]$ w.r.t. w. Hence, P is shorter than P' w.r.t. w' if and only if the third condition above is satisfied. In other words, v should keep p as its predecessor if and only if this condition is satisfied.

Let q be the lowest common ancestor of p and p' in T. Paths P and P' share vertices from s to q. Then they split up and do not meet before v.

Let Q be the subpath of P from the successor of q to p. Let Q' be the subpath of P' from the successor of q to p'. Testing the third condition above is equivalent to deciding whether the smallest vertex index in V_Q is smaller than the smallest vertex index in $V_{Q'}$.

We assume that for each vertex u in T, we have pointers $p_0[u], \ldots, p_{k_u}[u]$ and values $m_0[u], \ldots, m_{k_u}[u] \in \{1, \ldots, n\}$. For $i = 0, \ldots, k_u$, $p_i[v]$ points to the ancestor a of u in T for which the number of edges from a to u is 2^i. And $m_i[u]$ is the smallest vertex index on the path in T from a to u. The value of k_u is defined as the largest i such that $p_i[u]$ is defined. Note that $k_u = O(\log n)$.

Since P and P' have the same number of edges, the same holds for Q and Q'. From this observation, it follows that we can apply binary search on the pointers defined above to find lowest common ancestor q in $O(\log n)$ time. And with these pointers and the m_i-values, we can partition Q and Q' into $O(\log n)$ intervals in $O(\log n)$ time and find the smallest index in each interval in constant time per interval. Hence, we can decide whether the smallest vertex index in V_Q is smaller than the smallest vertex index in $V_{Q'}$ in logarithmic time, which gives the desired.

The only problem that remains is how to compute pointers and m_i-values during the course of the algorithm. Whenever the partially constructed lex-shortest path tree is extended with a new vertex u, we need to compute $p_0[u], \ldots, p_{k_u}[u]$ and $m_0[u], \ldots, m_{k_u}[u] \in \{1, \ldots, n\}$. But this can easily be done in $O(\log n)$ time using the p_i-pointers and m_i-values for the ancestors of u in T.

We can now conclude this section with the following theorem. Since we did not make use of planarity in this section, we get a more general result, which we believe to be of independent interest.

Theorem 3. *A lex-shortest path tree in an undirected graph with m edges and n vertices can be computed in $O((m+n)\log n)$ time.*

Proof. Follows by combining the above with a standard implementation of Dijkstra's algorithm. □

10 Concluding Remarks

We showed that finding a minimum cycle basis of an n-vertex planar, undirected, connected graph with non-negative edge weights requires $\Omega(n^2)$ time, implying that a recent algorithm by Amaldi et al. is optimal. We then presented an algorithm with $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space requirement that computes such a basis implicitly.

From this result, we obtained an output-sensitive algorithm requiring $O(n^{3/2}\log n + C)$ time and $O(n^{3/2} + C)$ space, where C is the total length of cycles in the basis that the algorithm outputs. For unweighted graphs, we obtained $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space bounds.

Similar results were obtained for the all-pairs min cut problem for planar graphs since for planar graphs, this problem is known to be dual equivalent to the minimum cycle basis problem.

As corollaries, we obtained algorithms that compute the weight vector and a Gomory-Hu tree of a planar n-vertex graph in $O(n^{3/2}\log n)$ time and $O(n^{3/2})$ space. The previous best bound was quadratic.

From the Gomory-Hu tree algorithm, we derived an oracle for answering queries for the weight of a min cut between any two given vertices. Preprocessing time is $O(n^{3/2}\log n)$ and space is $O(n^{3/2})$. Quadratic time and space was previously the best bound for constructing such an oracle. Our algorithm can output the actual cut in time proportional to its size.

Acknowledgments

I thank Sergio Cabello for some interesting comments and for introducing me to Gomory-Hu trees.

References

[1] E. Amaldi, C. Iuliano, T. Jurkiewicz, K. Mehlhorn, and R. Rizzi. Breaking the $O(m^2n)$ Barrier for Minimum Cycle Bases. A. Fiat and P. Sanders (Eds.): ESA 2009, LNCS 5757, pp. 301–312, 2009.

[2] F. Berger, P. Gritzmann, and S. de Vries. Minimum cycle bases for network graphs. Algorithmica, 40 (1): 51–62, 2004.

[3] F. Berger, P. Gritzmann, and S. de Vries. Minimum Cycle Bases and Their Applications. Algorithmics, LNCS 5515, pp. 34–49, 2009.

[4] D. W. Cribb, R. D. Ringeisen, and D. R. Shier. On cycle bases of a graph. Congr. Numer., 32 (1981), pp. 221–229.

[5] D. Cvetkovic, I. Gutman, and N. Trinajstic. Graph theory and molecular orbitals VII: The role of resonance structures. J. Chemical Physics, 61 (1974), pp. 2700–2706.

[6] N. Deo, G. M. Prabhu, and M. S. Krishnamoorthy. Algorithms for generating fundamental cycles in a graph. ACM Trans. Math. Software, 8 (1982), pp. 26–42.

[7] J. C. De Pina. Applications of shortest path methods. PhD thesis, University of Amsterdam, The Netherlands, 1995.

[8] E. T. Dixon and S. E. Goodman. An algorithm for the longest cycle problem. Networks, 6 (1976), pp. 139–149.

[9] A. Golynski and J. D. Horton. A polynomial time algorithm to find the minimum cycle basis of a regular matroid. In SWAT 2002: Proceedings of the 8th Scandinavian Workshop on Algorithm Theory, pages 200–209, 2002.

[10] R. Gomory and T. C. Hu. Multi-terminal network flows. J. SIAM, 9 (1961), pp. 551–570.

[11] D. Harel and R. E. Tarjan. Fast Algorithms for Finding Nearest Common Ancestors. SIAM J. Comput. Volume 13, Issue 2, pp. 338–355 (1984).

[12] D. Hartvigsen and R. Mardon. The All-Pairs Min Cut Problem and the Minimum Cycle Basis Problem on Planar Graphs. SIAM J. Discrete Math. Volume 7, Issue 3, pp. 403–418 (May 1994).

[13] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster Shortest-Path Algorithms for Planar Graphs. Journal of Computer and System Sciences volume 55, issue 1, August 1997, pages 3–23.

[14] J. D. Horton. A polynomial time algorithm to find the shortest cycle basis of a graph. SIAM J. Comput., 16 (1987), pp. 358–366.

[15] T. Kavitha, C. Liebchen, K. Mehlhorn, D. Michail, R. Rizzi, T. Ueckerdt, and K. Zweig. Cycle bases in graphs: Characterization, algorithms, complexity, and applications. 78 pages, submitted for publication, March 2009.

[16] T. Kavitha, K. Mehlhorn, D. Michail, and K. E. Paluch. An $\tilde{O}(m^2 n)$ algorithm for minimum cycle basis of graphs. Algorithmica, 52 (3): 333–349, 2008. A preliminary version of this paper appeared in ICALP 2004, volume 3142, pages 846–857.

[17] G. Kirchhofff. Ueber die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme geführt wird. Poggendorf Ann. Physik 72 (1847), pp. 497–508 (English transl. in Trans. Inst. Radio Engrs., CT-5 (1958), pp. 4–7).

[18] D. E. Knuth. The Art of Computer Programming, Vol. 1. Addison-Wesley, Reading, MA, 1968.

[19] E. Lawler. Combinatorial Optimization. Holt, Rinehart and Winston, New York, 1976.

[20] P. Matei and N. Deo. On algorithms for enumerating all circuits of a graph. SIAM J. Comput., 5 (1976), pp. 90–99.

[21] K. Mehlhorn and D. Michail. Minimum cycle bases: Faster and simpler. Accepted for publication in ACM Transactions on Algorithms, 2007.

[22] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.

[23] M. Randic. Resonance energy of very large benzenoid hydrocarbons. Internat. J. Quantum Chemistry, XVII (1980), pp. 549–586.

[24] H. Saran and V. V. Vazirani. Finding k-cuts within twice the optimal. SIAM Journal on Computing, 24:101–108, 1995.

[25] N. Trinajstic. Chemical Graph Theory. CRC Press, Boca Raton, FL, Vol. 2, 1983.

[26] V. V. Vazirani. Approximation Algorithms. Springer-Verlag, 2003.

Appendix

Proof of Lemmas 8 and 9

Let us first prove Lemma 8. We only need to consider the hard case where in beginning, all objects have weight 1 and at termination, exactly one object of weight W remains.

Consider running the algorithm backwards: starting with one object of weight W, repeatedly apply an operation *split* that splits an object of weight at least two into two new objects of positive integer weights such that the sum of weights of the two equals the weight of the original object. Assume that *split* runs in time proportional to the smaller weight of the two new objects. If we can give a bound of $O(W \log W)$ for this algorithm, we also get a bound on the algorithm stated in the theorem.

The running time for the new algorithm satisfies:

$$T(w) \leq \max_{1 \leq w' \leq \lfloor w/2 \rfloor} \{T(w') + T(w - w') + cw'\}$$

for integer $w > 1$ and constant $c > 0$. It is easy to see that the right-hand side is maximized when $w' = \lfloor w/2 \rfloor$. This gives $T(W) = O(W \log W)$, as desired.

The above proof also holds for Lemma 9.

Proof of Lemma 10

We need to show that for a cycle $C = C(v, e) \in \mathcal{H}(V_{\mathcal{J}})$ belonging to a region R, sets $\delta_{int}(R, C)$, $\delta_{ext}(R, C)$, and $\delta(R, C)$ can be computed in $O(\sqrt{n})$ time with $O(n^{3/2} \log n)$ preprocessing time and $O(n^{3/2})$ space.

First, observe that since C is completely contained in R, $\delta(R, C)$ is the subset of all boundary vertices belonging to C. Hence, this subset does not depend on R. We will thus refer to it as $\delta(C)$ in the following.

Let $v_0, \ldots, v_{r-1}$ be the boundary vertices encountered in that order in a simple, say clockwise, walk of $\mathcal{J}$ and let $\mathcal{J} = \mathcal{J}_0\mathcal{J}_1 \cdots \mathcal{J}_{r-1}$ be a decomposition of $\mathcal{J}$ into smaller curves where $\mathcal{J}_i$ starts in v_i and ends in $v_{(i+1) \bmod r}$, $i = 0, \ldots, r-1$. Each curve $\mathcal{J}_i$ is completely contained in an elementary face of G and we let $f(\mathcal{J}_i)$ denote this face.

In our proof, we need the following lemma and its corollary.

Lemma 13. *Let P be a shortest path in G from a vertex u to a vertex v. Then a vertex w belongs to P if and only if $d_G(u, w) + d_G(w, v) = d_G(u, v)$.*

Proof. If w belongs to P then clearly $d_G(u, w) + d_G(w, v) = d_G(u, v)$. And the converse is also true since shortest paths in G are unique. □

Corollary 8. *Let $C = C(v, e)$ be defined as above. Let $w \in V$ and assume that single-source shortest path distances in G with sources v and w have been precomputed. Then determining whether w belongs to C can be done in constant time.*

Proof. Let u_1 and u_2 be the end vertices of e and let P_1 resp. P_2 be the shortest paths in G from v to u_1 resp. u_2. Since C is isometric, both P_1 and P_2 belong to C and the union of their vertices is exactly the vertices of C. Hence, determining whether w belongs to C is equivalent to determining whether w belongs to P_1 or to P_2. The result now follows from Lemma 13. □

We will assume that single-source shortest path distances in G with each boundary vertex as source have been precomputed. As observed earlier, this can be done in $O(n^{3/2} \log n)$ time and $O(n^{3/2})$ space. Corollary 8 then allows us to find the set $\delta(C)$ of boundary vertices belonging to C in $O(r) = O(\sqrt{n})$ time. We may assume that we have the boundary vertices on C cyclically ordered according to how they occur on $\mathcal{J}$ in a clockwise walk of that curve.

In the following, let $v_i = v$ (so $C = C(v_i, e)$). Consider two consecutive vertices v_{i_1} and v_{i_2} of $\delta(C)$ in this cyclic ordering. We assume that $i_2 \neq i$ since the case $i_2 = i$ can be handled in a similar way. There are two possible cases:

1. the boundary vertices (excluding v_{i_1} and v_{i_2}) encountered when walking from v_{i_1} to v_{i_2} along $\mathcal{J}$ all belong to $int(C)$, or
2. they all belong to $ext(C)$.

Let v_{i_3} be the predecessor boundary vertex of v_{i_2} on $\mathcal{J}$ (i.e., $i_3 = (i_2 - 1) \bmod r$), see Figure 12. Then elementary face $f(\mathcal{J}_{i_3})$ belongs to $\overline{int}(C)$ if and only if the first case above holds. This follows from the fact that $\mathcal{J}$ does not cross any edges of G.

Lemma 14 below shows how we can check whether $f(\mathcal{J}_{i_3})$ belongs to $\overline{int}(C)$. First, let u and v be the end vertices of e and let P_u and P_v be the shortest paths from v_i to u and v, respectively. Suppose w.l.o.g. that v_{i_2}

I

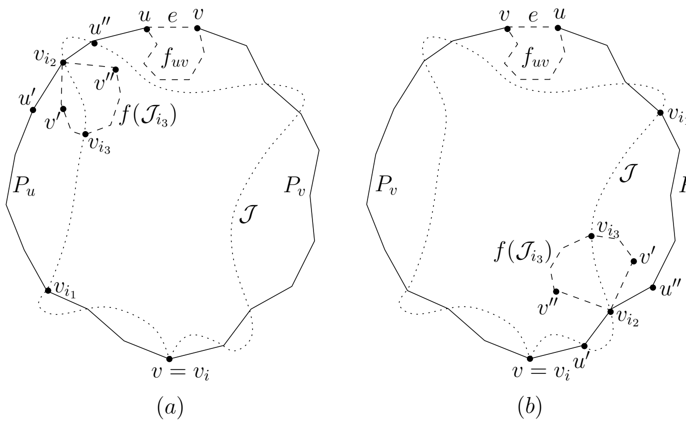


Figure 12: (a) The first condition and (b): the second condition in Lemma 14.

belongs to P_u, see Figure 12. Let u' be the predecessor of v_{i_2} on P_u. This is well-defined since $i_2 \neq i$. If $v_{i_2} \neq u$, let u'' be the successor of v_{i_2} on P_u. Otherwise, let $u'' = v$ (so u'' is the vertex $\neq u'$ adjacent to v_{i_2} on C). Let v' resp. v'' be the predecessor resp. successor of v_{i_2} in a clockwise walk of $f(\mathcal{J}_{i_3})$.

For three points p, q, r in the plane, let $W(p, q, r)$ be the wedge-shaped region with legs emanating from p and with right resp. left leg containing q resp. r.

Lemma 14. *With the above definitions, $f(\mathcal{J}_{I_3})$ belongs to $\overline{int}(C)$ if and only if one of the following conditions hold:*

1. *P_u is part of a clockwise walk of C (when directed from v_i to u) and $W(v_{i_2}, u', u'')$ contains $W(v_{i_2}, v', v'')$ (Figure 12(a)),*

2. *P_u is part of a counter-clockwise walk of C (when directed from v_i to u) and $W(v_{i_2}, u'', u')$ contains $W(v_{i_2}, v', v'')$ (Figure 12(b)).*

Proof. Assume first that P_u is part of a clockwise walk of C, see Figure 12(a). Then $\overline{int}(C)$ is to the right of the directed path $u' \to v_{i_2} \to u''$. Since G is straight-line embedded, $f(\mathcal{J}_{I_3})$ belongs to $\overline{int}(C)$ if and only if $W(v_{i_2}, u', u'')$ contains $W(v_{i_2}, v', v'')$.

Now, assume that P_u is part of a counter-clockwise walk of C, see Figure 12(b). Then $\overline{int}(C)$ is to the right of the directed path $u'' \to v_{i_2} \to u'$. Thus, $f(\mathcal{J}_{I_3})$ belongs to $\overline{int}(C)$ if and only if $W(v_{i_2}, u'', u')$ contains $W(v_{i_2}, v', v'')$. □

Lemma 14 and the above discussion show that to efficiently determine whether the boundary vertices between v_{i_1} and v_{i_2} belong to $\overline{int}(C)$ or to $\overline{ext}(C)$, we need to quickly find u', u'', v', and v'' and determine whether P_u is part of a clockwise or counter-clockwise walk of C.

By keeping a clockwise ordering of vertices of all elementary faces, we can find v' and v'' in constant time. For each shortest path tree in G rooted at a boundary vertex, we assume that each non-root vertex is associated with its parent in the tree. This allows us to find also u' in constant time.

As for u'', suppose we have precomputed, for each boundary vertex v_j and each $w \in V \setminus \{v_j\}$, the successor of v_j on the path from v_j to w in shortest path tree $T(v_j)$. Depth-first searches in each shortest path tree allow us to make these precomputations in $O(n^{3/2})$ time and space.

Now, since shortest paths are unique, the subpath of P_u from v_{i_2} to u is a path in shortest path tree $T(v_{i_2})$ and u'' is the successor of v_{i_2} on this path. With the above precomputations, we can thus find u'' in constant time.

Finally, to determine whether P_u is part of a clockwise walk of C, we do as follows. We first find the elementary faces adjacent to e in G. They can be obtained from dual tree $\tilde{T}(v_i)$ in constant time. We can also determine in constant time which of the two elementary faces belongs to $\overline{int}(C)$ since that elementary face is a child of the other in $\tilde{T}(v_i)$. Let f_{uv} be the elementary face in the interior of C. We check if the edge directed from u to v is part of a clockwise or counter-clockwise walk of f_{uv}. Again, this takes constant time. If it is part of a clockwise walk of f_{uv} then P_u is part of a clockwise walk of C (Figure 12(a)) and otherwise, P_u is part of a counter-clockwise walk of C (Figure 12(b)).

This concludes the proof of Lemma 10.

Proof of Lemma 11

Assume first that e is not an edge of G_1. Let P_1 and P_2 be the two shortest paths in G from v to the end vertices of e, respectively. Since e is not in G_1, it must belong to G_2. Hence, the intersection between C and G_1 is the union of paths Q, where Q is a subpath of either P_1 or P_2 with both its end vertices in $V_{\mathcal{J}}$. Each such path Q is a shortest path in G_1. It then follows from Lemma 5 that C does not cross any cycle of $\mathcal{B}_1'$.

Now, assume that e belongs to G_1 and let f_1, f_2, and $f_{\mathcal{J}}$ be defined as in the lemma. Let $C' \in \mathcal{B}_1'$ be given. We consider two cases: $\mathcal{J} \subset int(C')$ and $\mathcal{J} \subset ext(C')$.

Assume first that $\mathcal{J} \subset int(C')$. Then $R(C', \mathcal{B}_1)$ is an ancestor of $R(f_{\mathcal{J}}, \mathcal{B}_1)$. Since vertex v of C belongs to $V_{\mathcal{J}}$, part of C is contained in $int(C')$.

It follows that if C does not cross C' then e is contained in $\overline{int}(C')$. The converse is also true. For if e is contained in $\overline{int}(C')$ then by Lemma 5, both P_1 and P_2 are contained in $\overline{int}(C')$, implying that C does not cross C'.

Thus, C crosses C' if and only if e is not in $\overline{int}(C')$, i.e., if and only if f_1 and f_2 are both contained in $\overline{ext}(C')$. The latter is equivalent to the condition that $R(C', \mathcal{B}_1)$ is an ancestor of neither $R(f_1, \mathcal{B}_1)$ nor $R(f_2, \mathcal{B}_1)$ in $\mathcal{T}(\mathcal{B}_1)$. Hence, C crosses C' if and only if the second condition of the lemma is satisfied.

Now, assume that $\mathcal{J} \subset ext(C')$. Then $R(C', \mathcal{B}_1)$ is not an ancestor of $R(f_{\mathcal{J}}, \mathcal{B}_1)$. Again, Lemma 5 shows that C crosses C' if and only if e is not in $\overline{ext}(C')$, i.e., if and only if f_1 and f_2 are both contained in $\overline{int}(C')$. This holds if and only if $R(C', \mathcal{B}_1)$ is an ancestor of both $R(f_1, \mathcal{B}_1)$ and $R(f_2, \mathcal{B}_1)$ in $\mathcal{T}(\mathcal{B}_1)$. It follows that C crosses C' if and only if the first condition of the lemma is satisfied.

Proof of Lemma 12

Assume first that $\mathcal{J} \subset ext(C)$ and let C' be a descendant of C in $\mathcal{T}(\mathcal{B}_1)$. We need to show that C' is added to $\mathcal{B}$. Since $C' \in \mathcal{B}_1$, there is a pair of elementary faces f_1 and f_2 in G_1 which are separated by C' and not by any other cycle in $\mathcal{B}_1$. Let f_1 be contained in $\overline{int}(C')$ and let f_2 be contained in $\overline{ext}(C')$. Note that f_2 is contained in $\overline{int}(C)$ since otherwise, C would separate f_1 and f_2.

Since $\mathcal{J} \subset ext(C)$ and since no cycle of $\mathcal{B}$ crosses C, all cycles of $\mathcal{B} \setminus \mathcal{B}'_1$ belong to $\overline{ext}(C)$. Hence, no cycle of $\mathcal{H}(V_{\mathcal{J}}) \cup \mathcal{B}'_1 \cup \mathcal{B}'_2 \setminus \{C'\}$ separates f_1 and f_2. Since the set of cycles in the GMCB of G is a subset of $\mathcal{H}(V_{\mathcal{J}}) \cup \mathcal{B}'_1 \cup \mathcal{B}'_2$ by Lemma 3 and since $C' \in \mathcal{B}'_1$, it follows that C' is added to $\mathcal{B}$.

Now assume that $\mathcal{J} \subset int(C)$. Since no cycle of $\mathcal{B}$ crosses C, all cycles of $\mathcal{B} \setminus \mathcal{B}'_1$ belong to $\overline{int}(C)$. A similar argument as the above then shows that all cycles of $\mathcal{B}_1$ belonging to $\overline{ext}(C)$ must be part of the GMCB of G. These cycles are exactly the those that are not descendants of C in $\mathcal{T}(\mathcal{B}_1)$.

Solving the Replacement Paths Problem for Planar Directed Graphs in $O(n \log n)$ Time

Christian Wulff-Nilsen*

Abstract

In a graph G with non-negative edge lengths, let P be a shortest path from a vertex s to a vertex t. We consider the problem of computing, for each edge e on P, the length of a shortest path in G from s to t that avoids e. This is known as the *replacement paths problem.* We give a linear-space algorithm with $O(n \log n)$ running time for n-vertex planar directed graphs. The previous best time bound was $O(n \log^2 n)$.

1 Introduction

Computing shortest paths in graphs is a classical problem in combinatorial optimization with applications in numerous areas such as communication networks. These networks are in general not static but may change due to link failures. In such cases, alternative lines of communication need to be established and it may be of interest to determine the "quality" of such lines.

This motivates the *replacement paths problem (RPP)*: given two vertices s and t in a graph G with non-negative edge lengths and given a shortest path P (the line of communication) in G from s to t, compute, for each edge e on P, the length of a shortest path in G from s to t that avoids e (if no such path exists, the length is defined to be infinite). More motivation for the RPP is given in [3].

The RPP is a well studied problem. For undirected graphs with m edges and n vertices, algorithms are known with running time $O(m + n \log n)$ [8] and $O(m\alpha(m, n))$ [9], respectively (the latter applying to a stronger model of computation).

The directed case is harder since it has an $\Omega(m\sqrt{n})$ lower bound [5]. The trivial algorithm that removes each edge on shortest path P in turn and applies Dijkstra's algorithm to the resulting graph gives a time bound of $O(mn + n^2 \log n)$. Recently, this bound was improved to $O(mn + n^2 \log \log n)$ [4]. Roditty and Zwick [10] present a randomized algorithm with $\tilde{O}(m\sqrt{n})$ running time for unweighted, directed graphs. See also [2].

For planar directed graphs, an $O(n \log^3 n)$ time recursive algorithm is given in [3]. Klein, Mozes, and Weimann [7] show how recursion can be avoided and improve the time bound to $O(n \log^2 n)$ using linear space.

Our contribution is to improve the time bound of [7] for planar graphs by giving a linear space algorithm with $O(n \log n)$ running time. Our result is obtained by an adaptation of the $O(n \log n)$ time multiple-source shortest path algorithm of Klein [6] to the RPP.

The organization of the paper is as follows. In Section 2, we give some definitions, introduce some notation, and present some basic results that will prove useful later on. A large part of this section is taken from [6]. In Section 3, we show how the RPP can be split into two simpler sub-problems. Before presenting our algorithm, we show how to efficiently solve a problem related to the RPP in Section 4. The ideas introduced here will be a stepping stone towards obtaining our main result. We then give the algorithm for the first sub-problem in Section 5 and bound its time and space requirements in Section 6. In Section 7, we present an efficient algorithm for the other sub-problem. Finally, we make some concluding remarks in Section 8.

2 Definitions, Notation, and Toolbox

Let $G = (V, E)$ be a graph with non-negative edge lengths. For an edge $e \in E$, let $l_G(e)$ denote its length in G. For vertices, $u, v \in V$, $d_G(u, v)$ is the length of a shortest path in G from u to v w.r.t. l_G. If there is no such path, $d_G(u, v) = \infty$. We let V_G resp. E_G denote V resp. E.

We can assume w.l.o.g. that all shortest paths considered are simple. For a simple path $P = v_1 \to \cdots \to v_m$ in a directed graph G, $l_G(P) = \sum_{i=1}^{m-1} l_G(v_i, v_{i+1})$ denotes its length and for $1 \leq i \leq j \leq m$, $P[v_i, v_j]$ is the subpath $v_i \to \cdots \to v_j$. We let f_P be the flow in G assigning values to edges and reverses of edges of G as follows: for each edge (u, v) of P, $f_P(u, v) = -f_P(v, u) = 1$ and for all other edges/reverses of edges, f_P is zero.

Let T be a spanning tree in a directed graph $G = (V, E)$ and let T be rooted at a vertex s. For a $v \in V$, $T[v]$ is the simple path from s to v in T. We say that edge $(u, v) \in E$ is *relaxed (w.r.t. T)* if

*Department of Computer Science, University of Copenhagen.

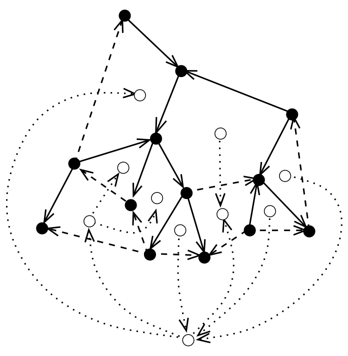


Figure 1: The dual (white vertices and dotted edges) of a spanning tree (solid edges) in a plane graph (black vertices and solid and dashed edges).

$d_T(s,u)+l_G(u,v) \geq d_T(s,v)$ and that (u,v) is *unrelaxed (w.r.t. T)* otherwise. Observe that all edges of E_T are relaxed. For an unrelaxed edge (u,v), removing the edge in T ending in v and reconnecting T by adding (u,v) is called *relaxing* (u,v). It is well-known that if all edges of E are relaxed w.r.t. T, T is a shortest path tree in G with source s.

Assume in the following that G is plane. Then since T is a spanning tree of G, the edges of $E \setminus E_T$ define a spanning tree T^* in the dual of G rooted at the external face of G (see [6]) and T^* contains all unrelaxed edges of G, where we identify each edge of G with its corresponding edge in the dual of G. We call T^* the *dual of T (in G)*, see Figure 1. For each edge e in T, the corresponding dual edge is directed from the face to the left of e (in the direction of e) to the face to the right of e. For an edge (u,v) in T^*, we define $l_{T^*}(u,v) = d_T(s,u) + l_G(u,v) - d_T(s,v)$. A *leafmost unrelaxed edge (in G w.r.t. T)* is an unrelaxed edge in G (and hence it belongs to T^*) w.r.t. T none of whose proper descendant edges in (the undirected version of) T^* are unrelaxed in G w.r.t. T.

Given a plane directed graph $G = (V,E)$, let G_∞ be a plane graph obtained by adding a vertex v_∞ to the interior of the external face of G and an edge from v_∞ to each vertex on the external face of G. For edges (u,v), (v,x), and (v,y) in G_∞, we say that (v,x) is *left (right) of (v,y) w.r.t. (u,v)* if (v,x) occurs strictly between (v,y) and (u,v) in counter-clockwise (clockwise) order.

Given an edge (v,y) on a simple path P in G, we say that an edge (v,x) *emanates left (right) from P* if either there is an edge (u,v) preceding (v,y) on P and (v,x) is left (right) of (v,y) w.r.t. (u,v) or if v is the first vertex of P and belongs to the external face of G and (v,x) is left (right) of (v,y) w.r.t. the edge from v_∞ to v in G_∞.

Given another simple path Q in G and a vertex $u \in V_P \cap V_Q$, we say that Q *leaves P from the left (right) at u* if there is an edge (u,v) of Q starting in u which emanates left (right) from P. And we say that Q *enters P from the left (right) at u* if there is an edge (v,u) of Q ending in u such that the reverse edge (u,v) emanates left (right) from P.

If both P and Q start in the same vertex s and end in the same vertex t, we say that Q is *left (right)* of P if the edges of positive flow in $f_Q - f_P$ define counter-clockwise (clockwise) cycles only (this definition is by Weihe [11] and is specialized in [6]).

For two spanning trees T_1 and T_2 in G, T_1 is *left* of T_2 if for all $v \in V$, path $T_1[v]$ is left of $T_2[v]$. If T is a shortest path tree in G with source s, we call it a *right-most* shortest path tree if every other shortest path tree in G with source s is left of T.

An s-rooted spanning tree T in G is *right-short* if the following holds for all $v \in V$: if P is a simple path in G from s to v that is right of $T[v]$ and $l_G(P) \leq l_G(T[v])$ then $P = T[v]$. A right-most shortest path tree is right-short [6]. The following result from [6] will prove useful.

Lemma 2.1. *Relaxing a leafmost unrelaxed edge in an r-rooted right-short spanning tree T in G yields an r-rooted right-short spanning tree in G that is left of T.*

We will need two dynamic tree data structures which represent and maintain, respectively, a rooted spanning tree T and its dual T^* and which support the following operations:

`replace`(e,e')**:** replaces edge e by edge e'.

`sum`(x)**:** returns the sum of lengths of edges from the root to vertex x.

`find`$()$**:** returns a leafmost unrelaxed edge

`change`(x,Δ)**:** for each edge e on the path between x and the root, the length of e is increased by the real number Δ if e points towards the root and decreased by Δ otherwise.

Top trees [1] support the above in logarithmic time per operation, see [6]. Note that the `change`-operation can be extended to subpaths of the path between x and the root by applying the operation twice.

3 Simplifying the Problem

In the following, let $G = (V,E)$ be an n-vertex plane directed graph with non-negative edge lengths and let $P = (v_0 = s) \to v_1 \to \cdots \to v_{m-1} \to (v_m = t)$ be a shortest path in G from a vertex s to a vertex t. For $i = 1,\ldots,m$, let e_i denote the edge (v_{i-1}, v_i). By transforming G if necessary, we may assume that s belongs to the external face of G. Since we are only

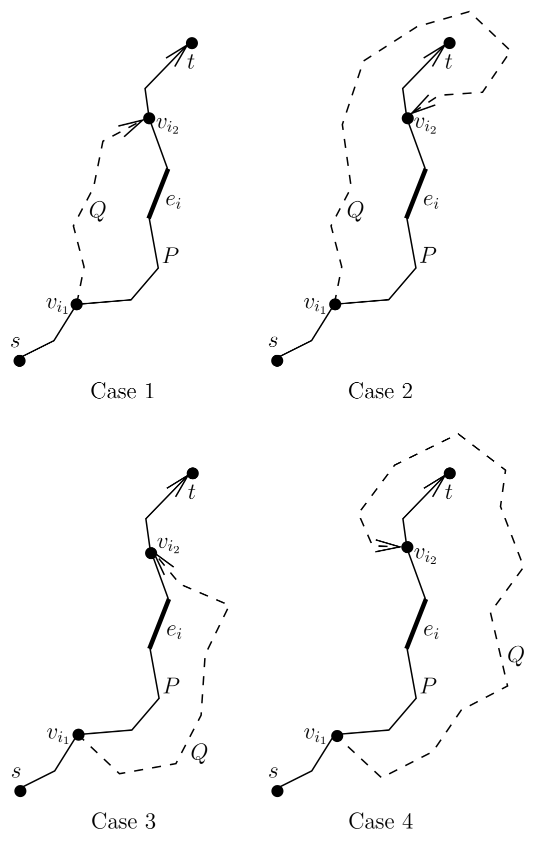


Figure 2: The four possible cases for shortest path Q.

interested in shortest paths ending in t, we may assume that this vertex has no outgoing edges.

Let $e_i \in E_P$ and let us analyze the structure of a shortest path Q in G from s to t avoiding e_i. Since P is a shortest path in G, Q can be chosen such that it has a decomposition $Q = Q_1Q_2Q_3$ where $Q_1 = P[s, v_{i_1}]$, $Q_3 = P[v_{i_2}, t]$ for some $0 \leq i_1 < i \leq i_2 \leq m$, and Q_2 is a path in G from v_{i_1} to v_{i_2} containing no vertices of P except v_{i_1} and v_{i_2}.

There are now four possible cases (see Figure 2):

Case 1: Q leaves P from the left at v_{i_1} and enters P from the left at v_{i_2}

Case 2: Q leaves P from the left at v_{i_1} and enters P from the right at v_{i_2}

Case 3: Q leaves P from the right at v_{i_1} and enters P from the right at v_{i_2}

Case 4: Q leaves P from the right at v_{i_1} and enters P from the left at v_{i_2}

Our algorithm for the RPP consists of four phases where in phase p, $p = 1, 2, 3, 4$, shortest paths of the form Q above are restricted to having the structure in case p. After these phases, we have four distance values for each edge e_i. The minimum of these four values is then the length of a shortest path in G from s to t that avoids e_i.

In the following, we consider each phase separately. Due to symmetry, we may restrict our attention to phases 1 and 2. We start with phase 1 and consider phase 2 in Section 7.

We remove from G edges (u, v), $v \neq t$, for which either (u, v) or (v, u) emanates right from P since these edges will not be needed in phase 1. Note that G may contain more than one connected component. If so, we remove all components except the one containing P. Now, P belongs to the external face of G and is part of a counter-clockwise walk of that face.

By adding edges to interior faces of G while keeping G planar, we may assume that for each $v \in V$, there is a path in G from s to v sharing no edges with P. We pick the lengths of these new edges sufficiently large so that finite shortest path distances will not decrease. With this modification of G, there is a shortest path tree in G rooted at s avoiding any given set of edges of P. Furthermore, we can ensure that these edges are avoided by increasing their lengths by a sufficiently large value (M_+ defined below). Note that P remains on the external face of G after these edges have been added.

Clearly, phase 1 corresponds to solving the RPP for the modified graph G. The idea is to use a dynamic tree data structure to maintain a shortest path tree in G that initially avoids e_m, then e_{m-1}, then e_{m-2}, and so on until a shortest path tree avoiding e_1 is obtained. During this process, the distances in G from s to t in the intermediate trees are computed. This is similar to the idea behind the multiple-source shortest path algorithm of Klein [6]. Indeed, we rely heavily on many of the results from that paper.

4 Solving a Related Problem

To simplify the presentation of our algorithm, we first consider a related problem. In this section, we show how to solve this problem in $O(n \log n)$ time. The ideas involved will prove useful in Sections 5, 6, and 7 where we present the RPP-algorithm.

The problem we consider is the following: for $i = 0, \ldots, m-1$, compute the length of a shortest path in G from s to t avoiding every edge on $P[v_i, t]$. We call it the *Forbidden Suffix Paths Problem (FSPP)*.

In the following, let $M_+ < \infty$ be a value such that any simple path in G has length strictly less than M_+. Pick, say, $M_+ = 1 + \sum_{e \in E} l_G(e)$.

We now present an $O(n \log n)$-time algorithm for the FSPP. Pseudo-code is given in Figure 3.

```
1.  compute a rightmost shortest path tree $T$ in $G$
    with source $s$
2.  for $i = m, \ldots, 1$
3.     $l_G(e_i) := l_G(e_i) + M_+$
4.     while there exists an unrelaxed edge
5.        relax a leafmost unrelaxed edge
6.     output $d_T(s,t)$
```

Figure 3: The FSPP algorithm.

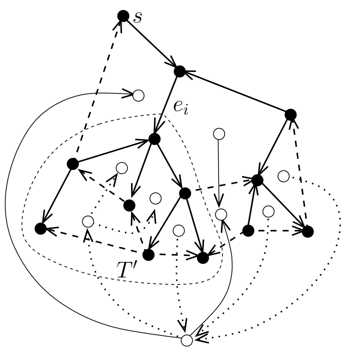


Figure 4: When the length of edge e_i changes in T then the edges of T^* whose lengths change form a single path (solid edges).

Theorem 4.1. *The algorithm in Figure 3 solves the FSPP for G and can be implemented to run in $O(n \log n)$ time with $O(n)$ space requirement.*

Proof. Clearly, the algorithm solves the FSPP for G by outputting the desired distances in reverse order.

We maintain T and its dual T^* using top trees supporting the operations of Section 2.

Note that edge lengths change during the course of the algorithm. Instead of explicitly making these changes in the underlying graph G, we choose an implementation maintaining the correct edge lengths in T and its dual T^*. This works since $E_T \cup E_{T^*} = E$ so we still keep track of the lengths of all edges in G.

Maintaining edge lengths in T: In line 3, we increase $l_T(e_i)$ by M_+ if $e_i \in T$. Otherwise, we do nothing.

Now, suppose a new edge (u,v) is about to be inserted into T in line 5. The algorithm needs to compute $l_G(u,v)$, the length of the edge to be inserted into T. Edge (u,v) is not already in T so $(u,v) \in T^*$. Thus, its length in T^* can be obtained in $O(\log n)$ time since $l_{T^*}(u,v) = |\texttt{sum}(v) - \texttt{sum}(u)|$ where the sum-operation is applied to T^* . We can then use the following formula to determine $l_T(u,v)$ in $O(\log n)$ time:

$$\begin{aligned} l_T(u,v) = l_G(u,v) &= l_{T^*}(u,v) + d_T(s,v) - d_T(s,u) \\ &= l_{T^*}(u,v) + \texttt{sum}(v) - \texttt{sum}(u), \end{aligned}$$

where the sum-operation is applied to T. We used the definition $l_{T^*}(u,v) = d_T(s,u) + l_G(u,v) - d_T(s,v)$.

Maintaining edge lengths in T^*: In line 3, either $e_i \in T$ or $e_i \notin T$. If $e_i \notin T$ then no distances from s in T change. Since $l_{T^*}(u,v) = d_T(s,u) + l_G(u,v) - d_T(s,v)$ for all edges $(u,v) \in T^*$, no edge lengths in T^* change except for $l_{T^*}(e_i)$ which is increased by M_+. This update can be performed in $O(\log n)$ time with two `change`-operations.

If on the other hand $e_i \in T$ then $d_T(s,u)$ increases by M_+ for all vertices u in the subtree T' of T rooted at the vertex of e_i furthest from s. For all other vertices u of T, $d_T(s,u)$ does not change. Thus, the edges of T^* whose lengths change are those between vertices of T' and vertices not in T'. These edges form a single path in (the undirected version of) T^* (see Figure 4 and [6]) and so they can be updated with at most two `change`-operations. This takes $O(\log n)$ time.

When an edge is inserted into T^* in line 5, we can compute its length in $O(\log n)$ time using ideas similar to those above for T.

Right-shortness invariant: We have now given an implementation where each execution of line 3 takes $O(\log n)$ time. As observed in [6], each relaxation can be performed within the same time bound and so can line 6 if we use the $\texttt{sum}(t)$-operation on T.

What remains therefore is to give an $O(n)$ bound on the total number of relaxations. Before doing this, we will need the following invariant: each tree generated by the algorithm is right-short.

To show this invariant, first observe that the initial tree is a rightmost shortest path tree and thus right-short. To see that line 3 preserves right-shortness, consider iteration i and let l_1 resp. l_2 be length function l_G just before resp. after line 3 is executed. Assume that T is right-short w.r.t. l_1 and let $v \in V$ be given. We need to show that $l_2(Q) > l_2(T[v])$ for any simple path $Q \neq T[v]$ in G from s to v that is right of $T[v]$.

So let Q be such a path. Suppose first that $T[v]$ does not contain e_i. If $e_i \in Q$ then $l_2(Q) \geq M_+ > l_2(T[v])$ where the last inequality follows from the assumption that there is a path in G from s to v avoiding every edge of P. So assume $e_i \notin Q$. Then $l_2(Q) = l_1(Q) > l_1(T[v]) = l_2(T[v])$ by right-shortness of T w.r.t. l_1.

Now, suppose that $e_i \in T[v]$. Then also $e_i \in Q$ since e_i is part of a counter-clockwise walk of the external face of G and Q is simple and right of $T[v]$. Then $l_2(Q) = l_1(Q) + M_+ > l_1(T[v]) + M_+ = l_2(T[v])$ by right-shortness of T w.r.t. l_1.

We conclude that line 3 preserves right-shortness and Lemma 2.1 implies that lines 4 and 5 also preserve

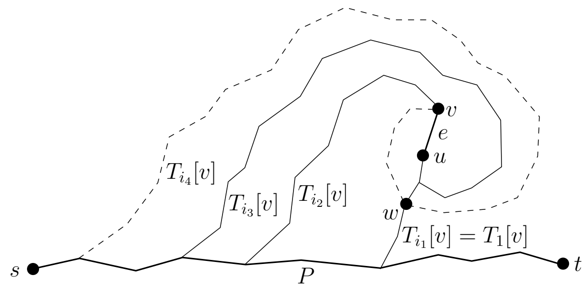


Figure 5: Path $T_{i_4}[v]$ must leave $T_1[v]$ from the left at a vertex w after entering this path from the right. But this contradicts the right-shortness of $T_{i_4}[v]$.

right-shortness. This shows the invariant.

Bounding the number of relaxations: We are now ready to give the $O(n)$ bound on the total number of relaxations. Consider any edge $e = (u, v)$ of G. Let k be the total number of trees generated by the algorithm and let T_i be the ith tree, $i = 1, \ldots, k$. We will show that the set of indices of the trees containing e is a consecutive subsequence of the cycle $(1 \ldots k)$. This will imply that e is relaxed at most once, giving the desired linear bound on the total number of relaxations (the same idea is used in [6]).

Consider four trees $T_{i_1}, T_{i_2}, T_{i_3}, T_{i_4}$, where $1 \leq i_1 < i_2 < i_3 < i_4 \leq k$. Assume for the sake of contradiction that either $e \in T_{i_1}, T_{i_3}$ and $e \notin T_{i_2}, T_{i_4}$ or that $e \in T_{i_2}, T_{i_4}$ and $e \notin T_{i_1}, T_{i_3}$. We will only consider the first case. The second case is similar. We may assume that $i_1 = 1$. The situation is shown in Figure 5.

By the right-shortness invariant and Lemma 2.1, $T_{i_{j'}}[v]$ is to the left of $T_{i_j}[v]$ for $1 \leq j < j' \leq 4$. Since $T_{i_2}[v]$ is to the left of $T_{i_1}[v]$, since $T_{i_3}[v]$ is to the left of $T_{i_2}[v]$, and since $e \notin T_{i_2}[v]$, $T_{i_3}[v]$ must leave $T_{i_1}[v]$ from the left and enter $T_{i_1}[v]$ from the right. Since $e \in T_{i_3}[v]$, $T_{i_3}[v]$ does not enter $T_{i_1}[v]$ from the right at v.

Since $e \notin T_{i_4}[v]$ and since $T_{i_4}[v]$ is to the left of $T_{i_3}[v]$, we must have that $T_{i_4}[v]$ leaves $T_{i_1}[v]$ from the left at some vertex w after entering this path from the right. Tree $T_{i_1} = T_1$ is a shortest path tree in the original graph so the subpath P_1 of $T_{i_1}[v]$ from w to v is no longer than the subpath P_4 of $T_{i_4}[v]$ from w to v in the graph containing T_{i_4}. This follows from the observation that the two subpaths do not contain edges of P. But since P_1 is to the right of P_4, $T_{i_4}[v]$ cannot be right-short, contradicting the invariant.

We conclude that e is relaxed at most once. Hence, the total number of relaxations performed by the algorithm is $O(n)$ and the $O(n \log n)$ time bound follows. Space is clearly linear.

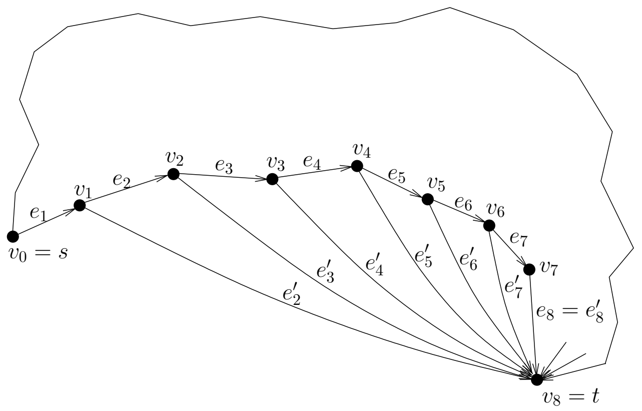


Figure 6: Graph G' is obtained from G by adding edges e'_i for $i = 2, \ldots, m-1$, here shown for an instance with $m = 8$.

5 The Algorithm

In this section, we present our algorithm for phase 1 of the RPP. Define graph $G' = (V', E')$ where $V' = V$ and E' is obtained from E by adding edge $e'_i = (v_{i-1}, t)$ of length $l_{G'}(e'_i) = d_G(v_{i-1}, t) + M_+$ for $i = 2, \ldots, m-1$. We also define $e'_m = e_m$. Note that G' is planar with new interior faces defined by triangles $e_i e'_i e'_{i+1}$ for $i = 2, \ldots, m-1$, see Figure 6. When convenient, we regard G as a subgraph of G'.

Pseudo-code of our algorithm is shown in Figure 7. Notice the similarity with the FSPP algorithm in Figure 3.

THEOREM 5.1. *The algorithm in Figure 7 solves the RPP for G.*

Proof. First, observe that in any iteration i, $l_{G'}(e_j) \geq M_+$ for $j = i, \ldots, m$ and $l_{G'}(e'_j) \geq M_+$ for $j = 2, \ldots, m$ when line 3 has just been executed. When line 6 is reached, $d_T(s, v_j)$ is thus the length of a shortest path in G from s to v_j that avoids edges $e_i, \ldots, e_m$, for $j = 1, \ldots, m$. In particular, in lines 9 and 11, $d_T(s, t)$ is the length of a shortest path in G from s to t that avoids edges $e_i, \ldots, e_m$.

Since $i + 1 > m$ if and only if we are in iteration $i = m$, the correct value for that iteration is thus output in line 11.

Now, assume that we are in iteration $i < m$ so that lines 6 to 10 are executed instead of line 11. In line 8, $l_{G'}(e'_j) = d_G(v_{j-1}, t) + M_+ - M_+ = l_G(P[v_{j-1}, t])$ for $j = i+1, \ldots, m$. Hence, $d_T(s, v_{j-1}) + l_{G'}(e'_j)$ is the length of a shortest path in G from s to t that avoids edges $e_i, \ldots, e_{j-1}$ and uses edges $e_j, \ldots, e_m$.

There is a shortest path Q in G from s to t avoiding e_i which can be decomposed into $Q_1 Q_2 Q_3$, where $Q_1 = P[s, v_{i_1}]$ and $Q_3 = P[v_{i_2}, t]$ for some

```
1.  compute a rightmost shortest path tree $T$ in $G'$
    with source $s$
2.  for $i = m, \ldots, 1$
3.    $l_{G'}(e_i) := l_{G'}(e_i) + M_+$
4.    while there exists an unrelaxed edge
5.      relax a leafmost unrelaxed edge
6.    if $i + 1 \leq m$
7.      for $j = i+1, \ldots, m$, $l_{G'}(e'_j) := l_{G'}(e'_j) - M_+$
8.      compute
        $d = \min_{j=i+1,\ldots,m}\{d_T(s, v_{j-1}) + l_{G'}(e'_j)\}$
9.      output $\min\{d_T(s,t), d\}$
10.     for $j = i+1, \ldots, m$, $l_{G'}(e'_j) := l_{G'}(e'_j) + M_+$
11.   else output $d_T(s,t)$
```

Figure 7: RPP algorithm for phase 1.

$0 \leq i_1 < i \leq i_2 \leq m$, and where Q_2 is a shortest path in G from v_{i_1} to v_{i_2} sharing no vertices with P except v_{i_1} and v_{i_2}. Combining this with the above observations, it follows that in line 9, $\min\{d_T(s,t), d\}$ is the length of a shortest path in G from s to t that avoids e_i. This shows the correctness of the algorithm.

6 Bounding Time and Space

We now give an implementation of the algorithm in Figure 7 and show that it has $O(n \log n)$ running time and $O(n)$ space requirement.

We maintain T, its dual T^* (in G'), and their edge lengths (and not the edge lengths in G' explicitly) as in the proof of Theorem 4.1. Since only edges of G are relaxed in line 5 (all other edges of G' have length at least M_+), we can use arguments similar to those in that proof to conclude that the total number of relaxations performed in line 5 is $O(n)$. Each execution of lines 9 and 11 takes $O(\log n)$ time with the `sum`-operation of Section 2.

We claim that each execution of lines 3, 7, and 10 also takes logarithmic time. From the proof of Theorem 4.1, this is true for line 3 so let us consider line 7 (line 10 is handled in a similar way).

Since $l_{G'}(e'_j) \geq M_+$ for $j = i+1, \ldots, m$ when this line is reached, none of these edges belong to T so no edge lengths in T are affected in this line. It follows that $e'_{i+1}, \ldots, e'_m$ all belong to T^* and they all need to decrease in length by M_+. Since these edges are on the same simple path in T^*, we can make this update with the `change`-operation in T^* in $O(\log n)$ time.

The above shows that if we ignore line 8, the algorithm can be implemented to run in $O(n \log n)$ time. In the following, we therefore focus on the problem of efficiently computing the value d.

```
8.1.  remove from $T$ the edge $e$ ending in $t$ and
      reconnect by adding $e'_{i+1}$
8.2   for all edges $e' \in E \setminus \{e_m\}$ adjacent to $t$,
      $l_{G'}(e') := l_{G'}(e') + M_+$
8.3.  while there exists an unrelaxed edge
8.4.    relax a leafmost unrelaxed edge
8.5.  let $e'_j$ be the edge of $T$ ending in $t$
8.6.  let $d = d_T(s,t)$
8.7.  if $i + 1 \leq j - 1$
8.8.    for $j' = i+1, \ldots, j-1$,
        $l_{G'}(e'_{j'}) := l_{G'}(e'_{j'}) + M_+$
8.9   for all edges $e' \in E \setminus \{e_m\}$ adjacent to $t$,
      $l_{G'}(e') := l_{G'}(e') - M_+$
8.10. remove $e'_j$ from $T$ and reconnect by adding the
      edge $e$ from line 8.1
```

Figure 8: Sub-routine of RPP algorithm computing the value d in iteration i.

The idea is to relax leafmost unrelaxed edges as in line 5 while ensuring that they all belong to $\{e'_{i+1}, \ldots, e'_m\}$. To guarantee that only $O(n)$ edges are relaxed throughout the course of the algorithm, we increase, in each iteration, the length of certain edges by M_+ so that they will not be relaxed again. We will show that these edges can be assumed not to belong to T in subsequent iterations which ensures that the algorithm remains correct.

Line 8 is expanded to the sub-routine in Figure 8. We now prove its correctness.

Looking at lines 4 to 7 in Figure 7, we see that just before line 8.1 is executed, all edges of $E' \setminus \{e'_{i+1}, \ldots, e'_m\}$ are relaxed. Hence, just after the execution of this line, only edges adjacent to t in G' can be unrelaxed. Line 8.2 has the effect that no edges of $E \setminus \{e_m\}$ adjacent to t are relaxed during the sub-routine. Combining this with the observation that in line 8.2, all edges in $\{e'_2, \ldots, e'_i\}$ have length at least M_+ whereas all edges in $\{e'_{i+1}, \ldots, e'_m\}$ have length strictly less than M_+ and $e'_{i+1} \in T$, it follows that when line 8.2 has just been executed, all unrelaxed edges of G' belong to $\{e'_{i+2}, \ldots, e'_m\}$ during the while-loop in lines 8.3 and 8.4.

Line 8.10 therefore ensures that T is the same at the beginning and end of the sub-routine. And line 8.9 ensures that this also holds for edge lengths of E' except those changed in line 8.8.

The above observations imply that if line 8.8 is omitted, the correct value of d is computed in line 8.6 since in that line, edges $\{e'_{i+1}, \ldots, e'_m\}$ are all relaxed w.r.t. T so $d_T(s,t) = \min\{d_T(s, v_{j-1}) + l_{G'}(e'_j) | j = i+1, \ldots, m\}$. Theorem 5.1 then implies that the entire algorithm is correct. Lemma 6.2 below shows that this

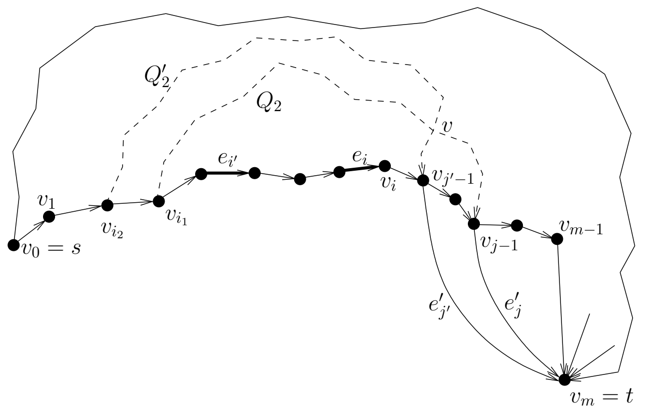


Figure 9: In the proof of Lemma 6.2, paths Q_2 and Q'_2 must share a vertex v.

is also true if we include line 8.8. First, we need the following result.

Lemma 6.1. *Just before an edge $e'_j \in \{e'_{i+2}, \ldots, e'_m\}$ is relaxed in the sub-routine in Figure 8, all edges in $\{e'_{i+1}, \ldots, e'_{j-1}\}$ are relaxed.*

Proof. Since initially, $e'_{i+1} \in T$ and since leafmost edges are relaxed, the lemma follows.

Lemma 6.2. *Let $J = \{i+1, \ldots, j-1\}$ be the set of indices j' in line 8.8 of iteration i in Figures 7 and 8 and let $1 \leq i' < i$. Let G_i resp. $G_{i'}$ be the graph G' just after line 7 has been executed in iteration i resp. i', but with the length of edge $e'_{j'}$ redefined as $d_G(v_{j'-1}, t)$ for all $j' \in J$. Then there is a shortest path in $G_{i'}$ from s to t avoiding each such edge.*

Proof. Let Q' be a shortest path in $G_{i'}$ from s to t and suppose it contains $e'_{j'}$ for some $j' \in J$. Let Q be the path from s to t in T in line 8.5 of iteration i. Then Q is a shortest path in G_i from s to t avoiding e_i and containing e'_j.

Since the restriction of T to E is right-short, Q can be decomposed into Q_1Q_2, where Q_1 is a subpath $P[s, v_{i_1}]$ of P and Q_2 is a path from v_{i_1} to t containing no vertices of P except v_{i_1} and v_{j-1} and containing e'_j as the last edge, see Figure 9.

We may also assume that Q' can be decomposed into $Q'_1Q'_2$, where Q'_1 is a subpath $P[s, v_{i_2}]$ of P and Q'_2 is a path from v_{i_2} to t containing no vertices of P except v_{i_2} and $v_{j'-1}$ and containing $e'_{j'}$ as the last edge.

We claim that $i_1 \geq i_2$, as shown in Figure 9. For suppose $i_1 < i_2$. Note that $i' < i$ and $i + 1 \leq j' < j$. When traversing P from s to t, we thus encounter v_{i_1}, v_{i_2}, $e_{i'}$, e_i, $v_{j'-1}$, and v_{j-1} in that order. Hence, Q and Q' both avoid e_i and $e_{i'}$ so these paths must be of equal length in G_i. We know that the algorithm relaxes e'_j in line 8.4 of iteration i. By Lemma 6.1, just before this event occurs, $e'_{j'}$ must be relaxed. But since $l_{G_i}(Q) = l_{G_i}(Q')$, e'_j must be relaxed as well at this point in time, a contradiction.

It follows that when traversing P from s to t, we encounter v_{i_2}, v_{i_1}, e_i, $v_{j'-1}$, and v_{j-1} in that order. Due to planarity, this is only possible if Q_2 and Q'_2 share a vertex v, see Figure 9. Then $Q_2[v,t]$ and $Q'_2[v,t]$ have the same length in $G_{i'}$, implying that $Q'[s,v]Q[v,t]$ is a shortest path from s to t in $G_{i'}$. Since this path avoids $e'_{j'}$ for all $j' \in J$, the lemma follows.

Theorem 6.1. *The algorithm in Figures 7 and 8 solves the RPP for G and can be implemented to run in $O(n \log n)$ time using $O(n)$ space.*

Proof. We have already argued that the algorithm is correct when line 8.8 in Figure 8 is omitted. And Lemma 6.2 states that even if an edge in line 8.8 was not increased in length by M_+, the algorithm would not find a shorter path from s to t in subsequent iterations. This shows that the full algorithm is correct.

Proving the time bound is split into two parts: first, we ignore the time for relaxations and give an $O(n \log n)$ time bound for the remaining algorithm. Then we show that the total number of relaxations is $O(n)$. Since each relaxation can be performed in logarithmic time with our top tree data structure, the claim will follow.

We now consider the first part. If we exclude line 8, we have previously argued that the remaining lines can be executed in a total of $O(n \log n)$ time. So let us consider the sub-routine in Figure 8.

In line 8.1, we need to update the graph structure of T and T^* as well as edge lengths in these trees. The former can be accomplished with the `replace`-operation. As for the latter, we need to compute the length of the new edge e'_{i+1} in T. This can be achieved in $O(\log n)$ time as in the proof of Theorem 4.1. Similarly, we can compute the length of the new edge e in T^* in $O(\log n)$ time.

For every other edge (u,t) adjacent to t in G', (u,t) belongs to T^*. Its old length (i.e., before the update) in T^* is $d_T(s,u) + l_{G'}(u,t) - (d_T(s,v) + l_{G'}(e))$, where $e = (v,t)$. Its new length is $d_T(s,u) + l_{G'}(u,t) - (d_T(s,v_i) + l_{G'}(e'_{i+1}))$. Hence, the length of (u,t) in T^* should increase by $\Delta = d_T(s,v) + l_{G'}(e) - (d_T(s,v_i) + l_{G'}(e'_{i+1}))$.

Since Δ is independent of the choice of (u,t), the lengths in T^* of all edges adjacent to t except e and e'_{i+1} should increase by Δ. Since the set of these edges form at most three simple paths in T^*, a constant number of `change`-operations suffice to make this update.

We have shown that line 8.1 can be executed in $O(\log n)$ time. A similar argument shows the same bound for line 8.10.

Lines 8.2, 8.8, and 8.9 can also be executed in $O(\log n)$ time (since none of the edges in those three lines belong to T, the argument for line 7 applies). And line 8.6 also takes logarithmic time since $d_T(s,t)$ can be obtained as $\texttt{sum}(t)$ in T.

Now, let us focus on the second part of the proof: giving an $O(n)$ bound on the total number of relaxations. Previously, we showed this for line 5 so we only need to consider the relaxations performed in line 8.4.

We first observe that when the length of an edge is increased in line 8.8, it will never again drop below M_+.

Now, consider some iteration i and suppose k_i edges are relaxed in line 8.4. Since leafmost unrelaxed edges are relaxed and since the edge e'_{i+1} initially belonging to T has length $l_{G'}(e'_{i+1}) < M_+$, there must be at least k_i edges in $\{e'_{i+2}, \ldots, e'_j\}$ of length strictly less than M_+ in line 8.5. Since the lengths of edges $e'_{i+2}, \ldots, e'_{j-1}$ are increased by M_+ in line 8.8, at least $k_i - 1$ of these lengths are increased from a value below to a value equal to or above M_+.

By the above observation, we can use a charging scheme to obtain the following bound on the total number of relaxations in line 8.4 over all m iterations of the for-loop in lines 2 to 11:

$$\begin{aligned}\sum_{i=1}^{m} k_i &= m + \sum_{i=1}^{m}(k_i - 1) \leq m + |\{e'_2, \ldots, e'_m\}| \\ &= 2m - 1 < 2n.\end{aligned}$$

It follows that the algorithm can be implemented to run in $O(n \log n)$ time. It is easy to see that space requirement is linear.

7 Phase 2

Above, we showed how to solve the phase corresponding to case 1 in Section 3. We now consider phase 2, i.e. we restrict our attention to shortest paths Q in G from s to t avoiding an edge of P with Q leaving P from the left and entering P from the right, see Figure 2.

The algorithm for phase 2 is similar to that of Section 5. The main difference lies in the modification of G and the construction of $G' = (V', E')$.

We modify G essentially by making an incision in G from s to t along P (see Figure 10) and removing edges not needed in phase 2.

More formally, we start by removing path $P \setminus \{t\}$ and its incident edges. Then two copies, $\overleftarrow{P} = (\overleftarrow{v_0} = \overleftarrow{s}) \to \overleftarrow{v_1} \to \cdots \to (\overleftarrow{v_m} = t)$ and $\overrightarrow{P} = (\overrightarrow{v_0} = \overrightarrow{s}) \to \overrightarrow{v_1} \to \cdots \to (\overrightarrow{v_m} = t)$, of P are inserted. These paths share only the last vertex t.

For $i = 0, \ldots, m-1$, we add to G the edge $(\overleftarrow{v_i}, u)$ for each edge (v_i, u) emanating left from P at v in the original graph G. Similarly, we add to G the edge $(u, \overrightarrow{v_i})$ for each edge (u, v_i) in G entering P from the right in the original graph G. The lengths of the inserted edges are identical to those in the original graph. Note that $\overleftarrow{P}$ and $\overrightarrow{P}$ belong to the external face of G.

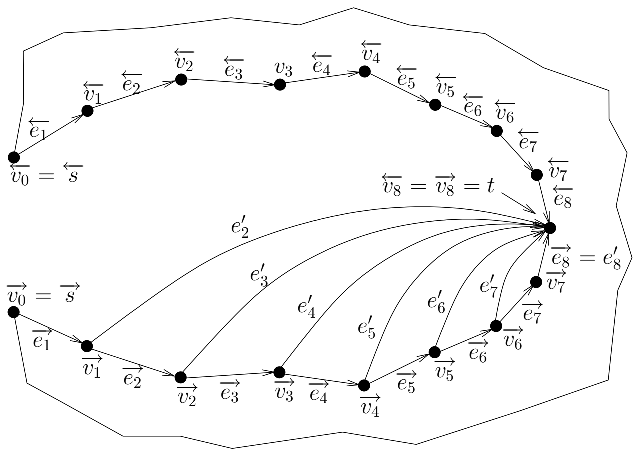


Figure 10: Graph G', obtained from G in phase 2 by adding edges e'_i for $i = 2, \ldots, m-1$, here shown for an instance with $m = 8$.

As in phase 1, we add edges to interior faces of G while keeping G planar such that for each $v \in V$, there is a path in G from $\overleftarrow{s}$ to v sharing no edges with $\overleftarrow{P} \cup \overrightarrow{P}$. We pick the lengths of these new edges sufficiently large so that finite shortest path distances will not decrease. We can perform this modification without having edges entering $\overleftarrow{P}$ or leaving $\overrightarrow{P}$. With this modification of G, there is a shortest path tree in G' rooted at $\overleftarrow{s}$ avoiding any given set of edges of $\overleftarrow{P} \cup \overrightarrow{P}$. And we can ensure that these edges are avoided by increasing their lengths by M_+. Note that $\overleftarrow{P} \cup \overrightarrow{P}$ remains on the external face of G after these edges have been added.

For $i = 1, \ldots, m$, let $\overleftarrow{e_i} = (\overleftarrow{v_{i-1}}, \overleftarrow{v_i})$ and $\overrightarrow{e_i} = (\overrightarrow{v_{i-1}}, \overrightarrow{v_i})$. Phase 2 corresponds to solving the following problem on the modified graph G: for $i = 1, \ldots, m$, compute the length of a shortest path in G from $\overleftarrow{s}$ to t that avoids edges $\overleftarrow{e_i}$ and $\overrightarrow{e_i}$. We will refer to this problem as the RPP for G.

Graph $G' = (V', E')$ is obtained from G by adding, for $i = 2, \ldots, m-1$, the edge $(\overrightarrow{v_i}, t)$ of length $d_G(v_i, t) + M_+$, where M_+ is defined as in Section 4. We let e'_i denote this edge, see Figure 10. We also define $e'_m = \overrightarrow{e_m}$. Note that G' is planar and of size $O(n)$. In G', we set the length $l_{G'}(\overrightarrow{e_i})$ of $\overrightarrow{e_i}$ equal to M_+.

The algorithm for phase 2 is shown in Figures 11 and 12. We now prove that it is correct and has the desired time and space bounds.

Using ideas from Section 6, it follows that the

```
1.  compute a rightmost shortest path tree $T$ in $G'$ with source $\overleftarrow{s}$
2.  for $i = m, \ldots, 1$
3.    $l_{G'}(\overleftarrow{e_i}) := l_{G'}(\overleftarrow{e_i}) + M_+$
4.    while there exists an unrelaxed edge
5.      relax a leafmost unrelaxed edge
6.    if $i + 1 \leq m$
7.      for $j = i+1, \ldots, m$, $l_{G'}(e'_j) := l_{G'}(e'_j) - M_+$
8.      compute $d = \min_{j=i+1,\ldots,m}\{d_T(s, \overrightarrow{v_{j-1}}) + l_{G'}(e'_j)\}$
9.      output $\min\{d_T(s,t), d\}$
10.     for $j = i+1, \ldots, m$, $l_{G'}(e'_j) := l_{G'}(e'_j) + M_+$
11.   else output $d_T(s,t)$
```

Figure 11: RPP algorithm for phase 2.

```
8.1.  remove from $T$ the edge $e$ ending in $t$ and reconnect by adding $e'_m$
8.2   for all edges $e' \in E \setminus \{\overrightarrow{e_m}\}$ adjacent to $t$, $l_{G'}(e') := l_{G'}(e') + M_+$
8.3.  while there exists an unrelaxed edge
8.4.    relax a leafmost unrelaxed edge
8.5.  let $e'_j$ be the edge of $T$ ending in $t$
8.6.  let $d = d_T(s,t)$
8.7.  if $j + 1 \leq m$
8.8.    for $j' = j+1, \ldots, m$, $l_{G'}(e'_{j'}) := l_{G'}(e'_{j'}) + M_+$
8.9   for all edges $e' \in E \setminus \{e_m\}$ adjacent to $t$, $l_{G'}(e') := l_{G'}(e') - M_+$
8.10. remove $e'_j$ from $T$ and reconnect by adding the edge $e$ from line 8.1
```

Figure 12: Sub-routine of RPP algorithm for phase 2.

entire algorithm is correct if line 8.8 is omitted. And Lemma 7.2 below shows that this also holds with this line included.

LEMMA 7.1. *Just before an edge $e'_j \in \{e'_{i+1}, \ldots, e'_{m-1}\}$ is relaxed in the sub-routine in Figure 12, all edges in $\{e'_{j+1}, \ldots, e'_m\}$ are relaxed.*

Proof. Since initially, $e'_m \in T$ and since leafmost edges are relaxed, the lemma follows.

LEMMA 7.2. *Let $J = \{j+1, \ldots, m\}$ be the set of indices j' in line 8.8 of iteration i in Figures 11 and 12 and let $1 \leq i' < i$. Let G_i resp. $G_{i'}$ be the graph G' just after line 7 has been executed in iteration i resp. i', but with the length of edge $e'_{j'}$ redefined as $d_G(v_{j'-1}, t)$ for all $j' \in J$. Then there is a shortest path in $G_{i'}$ from $\overleftarrow{s}$ to t avoiding each such edge.*

We omit the proof since it is similar to that of Lemma 6.2. We can now bound the time and space used for phase 2.

THEOREM 7.1. *The algorithm in Figure 11 and Figure 12 can be implemented to run in $O(n \log n)$ time using $O(n)$ space.*

Again, we omit the proof since it is similar to that of Theorem 6.1.

By symmetry, phases 3 and 4 can be executed within the same bounds as phases 1 and 2. This gives us the following result.

THEOREM 7.2. *For a directed planar n-vertex graph with non-negative edge-lengths, the replacement paths problem can be solved in $O(n \log n)$ time with $O(n)$ space.*

8 Concluding Remarks

Given an n-vertex planar directed graph G with non-negative edge lengths and given a shortest path P in G from a vertex s to a vertex t, we presented a linear-space algorithm that computes, for each edge $e \in P$, the length of a shortest path in G from s to t that avoids e. Running time is $O(n \log n)$, improving on a bound of $O(n \log^2 n)$ by Klein, Mozes, and Weimann.

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. *Maintaining information in fully-dynamic trees with top trees.* ArXiv cs.DS/0310065.

[2] C. Demetrescu, M. Thorup, R. Alam Chaudhury, and V. Ramachandran. *Oracles for distances avoiding a link-failure.* Preliminary version of this paper appears in Proc. of 13th SODA, pages 838–843, 2002.

[3] Y. Emek, D. Peleg, and L. Roditty. *A Near-Linear Time Algorithm for Computing Replacement Paths in Planar Directed Graphs.* In SODA'08: Proceedings of the Nineteenth Annual ACM-SIAM symposium on Discrete Algorithms, pages 428–435, Philadelphia, PA, USA, 2008, Society for Industrial and Applied Mathematics.

[4] Z. Gotthilf and M. Lewenstein. *Improved algorithms for the k simple shortest paths and the replacement paths problems.* Information Processing Letters, Vol. 109, 7/2009, Pages 352–355.

[5] J. Hershberger, S. Suri, and A. Bhosle. *On the difficulty of some shortest path problems.* In Proc. of the 20th STACS, pages 343–354, 2003.

[6] P. N. Klein. *Multiple-source shortest paths in planar graphs.* Proceedings, 16th ACM-SIAM Symposium on Discrete Algorithms, 2005, pp. 146–155.

[7] P. N. Klein, S. Mozes, and O. Weimann. *Shortest Paths in Directed Planar Graphs with Negative Lengths: a*

Linear-Space $O(n \log^2 n)$-Time Algorithm. Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms, p. 236–245, 2009.

[8] K. Malik, A. K. Mittal, and S. K. Gupta. *The k most vital arcs in the shortest path problem.* Operations Research Letters, 8(4):223–227, 1989.

[9] E. Nardelli, G. Proietti, and P. Widmayer. *A faster computation of the most vital edge of a shortest path.* Information Processing Letters, 79 (2): 81–85, 2001.

[10] L. Roditty and U. Zwick. *Replacement paths and k simple shortest paths in unweighted directed graphs.* In Proc. Automata, Languages and Programming, 32nd International Colloquium, 249–260, 2005.

[11] K. Weihe. *Maximum (s,t)-flows in planar networks in $O(|V| \log |V|)$ time.* JCSS 55 (1997), pp. 454–476.

Available online at www.sciencedirect.com
ScienceDirect
Computational Geometry 40 (2008) 1–13
Computational Geometry Theory and Applications
www.elsevier.com/locate/comgeo

Steiner hull algorithm for the uniform orientation metrics

Christian Wulff-Nilsen

Department of Computer Science, University of Copenhagen, DK-2100 Copenhagen, Denmark

Received 26 September 2005; received in revised form 4 October 2007; accepted 5 October 2007

Available online 22 October 2007

Communicated by R. Seidel

Abstract

Given a set Z of $n < \infty$ points in the plane and an integer $\lambda \geqslant 2$, we consider the problem of finding a λ-Steiner hull of Z, i.e., a region containing every Steiner minimal tree for Z in the λ-metric. We define a λ-Steiner hull $\lambda\text{-SH}(Z)$ of Z as a set obtained by a maximal sequence of removals of certain open wedge-shaped regions from an initial hull followed by a simplification of its boundary. A perhaps surprising result is presented, namely that a Euclidean MST for Z can be used to decompose the problem of finding $\lambda\text{-SH}(Z)$ into subproblems. Each of these can then be solved recursively using linear searches combined with a sweep line approach. Using this result, we present an algorithm computing $\lambda\text{-SH}(Z)$. This algorithm has $O(\lambda n \log n)$ running time and $O(\lambda n)$ space requirement which is optimal for constant λ. We prove that $\lambda\text{-SH}(Z)$ is independent of the order of removals of the open wedge-shaped regions.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Computational geometry; Uniform orientation metric; Steiner tree problem; Steiner hull; Minimum spanning tree

1. Introduction

The classical *Steiner tree problem* is the problem of computing a *Steiner minimal tree* (SMT), i.e., a tree of minimum Euclidean length, spanning a given set of points in the plane [3]. It is distinguished from the minimum spanning tree problem in that new points may be added to shorten the tree. This makes the problem much harder—in fact, it has been shown to be NP-hard.

Steiner minimal trees are useful for routing in VLSI design [4]. Here, an important objective is to interconnect a set of pins on a chip using minimum total wire length. Due to manufacturing limitations however, the orientation of wires have typically been restricted to horizontal and vertical only, making the L_1-metric more suitable for measuring the cost of a network.

More recently, routing using an arbitrary number of uniformly distributed wire orientations has become feasible. For this reason, the *uniform orientation metric* has received some attention in recent years.

This metric is defined as follows. Given an integer $\lambda \geqslant 2$, the set of *uniform orientations* or *λ-orientations* is the set of angles $i\omega$, $i = 0, \ldots, \lambda - 1$, where $\omega = \pi/\lambda$. A line segment, half-line, or line l is said to be *uniformly oriented* if the angle between l and the x-axis is a uniform orientation. The *λ-distance* d_λ between two points is the length of a

E-mail address: koolooz@diku.dk.

shortest path of uniformly oriented line segments between the points and we refer to d_λ as the λ-*metric* or the *uniform orientation metric*. Note that the 2-metric is the L_1-metric.

A λ-tree is a tree in the plane such that all edges consist of uniformly oriented line segments. The *Steiner tree problem in the uniform orientation metric* (USTP) is to find a λ-tree of minimal length spanning a finite set Z of points or *terminals* in the plane. We refer to such a tree as a *Steiner minimal λ-tree* (λ-SMT) for Z. Additional *Steiner points* may be incorporated to shorten the tree. Like the Euclidean Steiner tree problem, the USTP is NP-hard [2].

In the Euclidean metric, a *Steiner hull* of a given set Z of terminals is a subset of the plane containing every SMT for Z. The convex hull $\mathrm{CH}(Z)$ of Z is an example of a Steiner hull of Z. Having a tight Steiner hull can make the computation of an SMT easier since the number of feasible topologies is reduced as the number of terminals on the boundary of a Steiner hull increases. Furthermore, a non-simple Steiner hull results in the decomposition of an SMT into SMTs for smaller terminal subsets.

Winter [7] presented an $O(n \log n)$ time algorithm for computing a Steiner hull of n terminals. The algorithm starts with $\mathrm{CH}(Z)$ and then iteratively removes certain open wedge-shaped regions to obtain smaller and smaller Steiner hulls.

In this paper, we consider Steiner hulls for the λ-metric. We define a λ-*Steiner hull* of Z to be a subset of the plane known to contain every λ-SMT for Z.

We will address the problem of efficiently finding a tight λ-Steiner hull of Z. We consider a type of λ-Steiner hull, referred to as $\lambda\text{-SH}(Z)$, which in many ways is similar to that presented in [7] for the Euclidean metric.

We will show that this λ-Steiner hull can be constructed in $O(\lambda n \log n)$ time using $O(\lambda n)$ space and prove that this is optimal under the assumption that λ is a constant. This assumption seems reasonable since in VLSI design, λ is typically much smaller than n (to the author's knowledge, λ-values of 2 and 4 are probably the most widely used today).

The paper is organized as follows. In Section 2, we make various definitions and some simple observations. In Section 3, we prove that a certain set $\lambda\text{-SH}'(Z)$, from which $\lambda\text{-SH}(Z)$ is easily derived, is a λ-Steiner hull of Z. Letting n equal the number of terminals, we then present a naive $O((\lambda n)^3)$ time algorithm computing this set. In Section 4, we show how a Euclidean MST can be used to decompose the problem of finding $\lambda\text{-SH}'(Z)$ into smaller problems each of which can be solved recursively. The results of Section 5 enable us to efficiently check if a region of our partially constructed λ-Steiner hull can be removed. To do this we use a sweep line algorithm for preprocessing. This improves running time to $O((\lambda n)^2)$. In Section 6, we show how to construct $\lambda\text{-SH}(Z)$ by performing linear searches "in parallel" at each level of the recursion. In Section 7, we show that $\lambda\text{-SH}(Z)$ can be found in time $O(\lambda n \log n)$ using $O(\lambda n)$ space. We show that this is optimal for constant λ. In Section 8, we prove that $\lambda\text{-SH}(Z)$ does not depend on the order of removals of open wedge-shaped regions. Finally, we make some concluding remarks in Section 9.

2. Definitions and basic properties

Since we will be dealing with different types of points, we will reserve the letter z for terminals, s for Steiner points, u, v, and w for vertices (terminals and Steiner points), and other letters for regular points.

Let p and q be two points in the plane. If pq is uniformly oriented, there is a unique shortest path from p to q in the λ-metric, namely the line segment pq. Otherwise, the set of shortest paths from p to q in the λ-metric constitutes a parallelogram $prqr'$. The shortest paths prq and $pr'q$ from p to q are called the *critical paths* from p to q and r and r' are called *corner points* of the critical paths.

The λ-*lune of p and q* denoted $L_\lambda(p,q)$ is defined as the set $L_\lambda(p,q) = \{s \in \mathbb{R}^2 \mid d_\lambda(s,p) < d_\lambda(p,q) \wedge d_\lambda(s,q) < d_\lambda(p,q)\}$, see Fig. 1.

If a, b, and c are three distinct points in the plane then we define $\angle abc$ as the smaller non-negative angle between line segments ba and bc.

Let l be a half-line emanating from a point p and let l_x be the horizontal half-line emanating from p and lying to the right of p. Then we say that l has *direction* $\theta \in [0, 2\pi[$ if the counter-clockwise angle from l_x to l equals θ.

Given a simple polygon P, we define a *clockwise walk of P* to be a walk of the boundary of P such that the interior of P is to the right during the walk. For a tree T embedded in the plane, consider inflating its edges. An outer walk of T is then called *clockwise* if the "interior" of T is to the right during the walk.

For any subset X of $\mathbb{R}^2$ we let X° denote the interior of X. We shall assume that all subsets of the plane considered in this paper are closed unless otherwise stated.

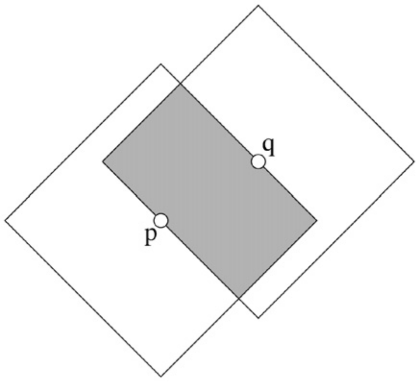


Fig. 1. The λ-lune $L_\lambda(p,q)$ (dark area) of points p and q, here shown for $\lambda = 2$.

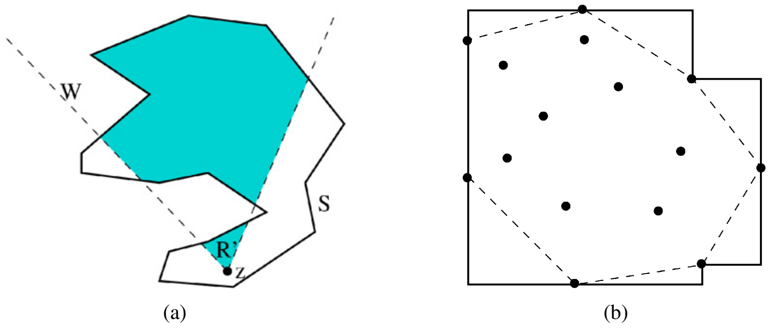


Fig. 2. (a) $W \cap S$ is the union of simple polygons. (b) λ-CH(Z), here shown for $\lambda = 2$. The dashed polygon shows CH(Z).

Suppose that l_a and l_b are uniformly oriented half-lines emanating from a common point p. Then we let $W(l_a, l_b)$ denote the open wedge-shaped region of points hit when sweeping a halfline emanating from p counter-clockwise from l_a to l_b. Halfline l_a is called the *right leg* and l_b is called the *left leg* of $W(l_a, l_b)$. Let θ denote the counter-clockwise angle from l_a to l_b. If $\theta = \lfloor 2\lambda/3 \rfloor \omega$ then $W(l_a, l_b)$ is called a λ-*wedge* (*of* p) and if $\theta = \omega$, $W(l_a, l_b)$ is called a λ-*cone* (*of* p). If $p_a \neq p$ is a point on l_a and $p_b \neq p$ is a point on l_b then we define $W(p, p_a, p_b) = W(l_a, l_b)$.

The λ-Steiner hulls that we will consider in this paper are constructed by iteratively removing regions bounded by λ-wedges from an initial hull. We need to make sure that each such region does not contain any part of any λ-SMT for Z. In particular it should not contain any terminals.

This motivates the following definition. Let S be a simple polygon and let W be a λ-wedge of a terminal $z \in S$. Then $W \cap S$ is a union of regions bounded by simple polygons. One of these regions, say R', contains z on its boundary, see Fig. 2(a). Suppose that $R = W^\circ \cap R'$ is non-empty and contains no terminals of Z. Then W is called *safe* (*in* S), R is called a *safe region* (*of* S) and the removal of R from S is called a *safe removal* (*from* S). We say that R is *bounded by* W.

We refer to a subpath of the boundary of S connecting two consecutive terminals as a *boundary subpath* (*of* S). If R is a safe region of S then the part I of the boundary of S intersecting R is free of terminals and thus I is fully contained in a boundary subpath p. Let z_1 be the terminal of the safe λ-wedge bounding R and let z_2 and z_3 be the end terminals of p. Then we say that R is *bounded* by z_1, z_2, and z_3 and we refer to z_1 as the *base terminal* of R.

Let $z_0, \ldots, z_{r-1}$ be a cyclic ordering of the terminals on the boundary of the convex hull CH(Z) of Z. For $i = 0, \ldots, r-1$, let P_i be the parallelogram consisting of all the shortest paths between z_i and z_{i+1} in the λ-metric (indices are modulo r). The λ-*convex hull* λ-CH(Z) of Z is then defined as $\lambda\text{-CH}(Z) = \text{CH}(Z) \cup \bigcup_{i=0}^{r-1} P_i$, see Fig. 2(b).

In the following, we will let λ-SH$'(Z)$ denote a set obtained by a maximal sequence of safe removals from the initial hull λ-CH(Z).

Note that for each set S obtained in such a maximal sequence, all concave angles of the boundary of S are at terminals. This implies that all safe regions removed are convex. Also note that the line segments bounding these regions are all uniformly oriented.

K

We obtain λ-SH(Z) from λ-SH′(Z) by replacing each boundary subpath of λ-SH′(Z) by a critical path between the two terminals defining the endpoints of that boundary subpath; the critical paths are chosen such that all corner points are right turns in a clockwise walk of λ-SH(Z). If the line segment l between the two terminals is uniformly oriented, the boundary subpath is replaced by l.

As we shall see, λ-SH(Z) is a λ-Steiner hull and it is independent of the chosen maximal sequence of safe removals.

3. λ-Steiner hull

For now, let us consider λ-SH′(Z). We will return to λ-SH(Z) in Section 6.

In this section, we prove that λ-SH′(Z) is a λ-Steiner hull of Z. We do this by showing that λ-CH(Z) is a λ-Steiner hull of Z and that each safe removal does not cut off any part of any λ-SMT T for Z. The former is shown in Lemma 4 below. To show the latter we will show that after a safe removal,

(1) no terminal is cut off;
(2) no Steiner point of T is cut off;
(3) no part of any edge of T is cut off.

The first part follows by definition of a safe λ-wedge. The second part is shown in Lemma 2 below and the third part in Lemma 3. We need the following result.

Lemma 1. *Let (u, v) be any edge of a λ-SMT. No vertex of the λ-SMT can lie in the λ-lune $L_\lambda(u, v)$.*

Proof. If w is a vertex in $L_\lambda(u, v)$, we may assume that the λ-SMT contains a path from w to u not containing v. Since $d_\lambda(w, v) < d_\lambda(u, v)$, the λ-SMT can be shortened by deleting (u, v) and adding (w, v), a contradiction. □

Lemma 2. *Let S be a λ-Steiner hull of terminal set Z and let S' be the set obtained by a safe removal from S. Then all Steiner points of any λ-SMT for Z belong to S'.*

Proof. Let W be a safe λ-wedge of a terminal z and let T be a λ-SMT for Z. Suppose for the sake of contradiction that the safe region R of S bounded by W contains a Steiner point s of T. Pick s such that its Euclidean distance to z is maximized over all Steiner points of T contained in R. Since the angle between the legs of W is $\lfloor 2\lambda/3\rfloor\omega$ and since the angle between Steiner tree edges of s is at most $(\lfloor 2\lambda/3\rfloor + 1)\omega$ [1] there exists an edge (s, v) in T such that $v \in W$. Since S is a λ-Steiner hull of Z, (s, v) is fully contained in S and since $s \in R$, we have $v \in R$.

By the choice of s, v must be a terminal. But this contradicts the assumption that R is a safe region. □

Lemma 3. *Let S be a λ-Steiner hull of terminal set Z and let S' be the set obtained by a safe removal from S. Then all edges of any λ-SMT for Z are fully contained in S'.*

Proof. Let $W(l_1, l_2)$ be a safe λ-wedge of a terminal z and let T be a λ-SMT for Z. We claim that the safe region R bounded by $W(l_1, l_2)$ does not intersect any edge of T.

Assume, for the sake of contradiction, that (u, v) is an edge of T intersecting R. By Lemma 2, (u, v) must cross $W(l_1, l_2)^\circ$. Without loss of generality, assume that u belongs to the halfplane of the line through l_1 not containing l_2, see Fig. 3.

Suppose that the line segment from z to u makes angle θ_u with the x-axis, that the line segment from z to v makes angle θ_v with the x-axis, that l_1 makes angle θ_1 with the x-axis, and that l_2 makes angle θ_2 with the x-axis. By rotating about z by a multiple of ω if necessary, we may assume that $0 \leqslant \theta_u < \omega$ and we have the inequalities $\theta_u \leqslant \theta_1 < \theta_2 \leqslant \theta_v$.

Let C_u be the set of points having λ-distance at most $d_\lambda(u, z)$ to u. We assume that $\theta_u > 0$. The case $\theta_u = 0$ is handled similarly. Since (u, v) crosses $W(l_1, l_2)^\circ$ we have $\theta_v < \pi + \omega$. The intersection of C_u and the λ-cone of u containing z is a triangle $\triangle uab$ and line segment ab makes angle $\lceil\frac{\theta_u}{\omega}\rceil\omega + \frac{\pi-\omega}{2}$ with the x axis. Since

$$\left\lceil\frac{\theta_u}{\omega}\right\rceil\omega + \frac{\pi-\omega}{2} < \theta_2 \leqslant \theta_v < \pi + \omega = \left\lceil\frac{\theta_u}{\omega}\right\rceil\omega + \pi < \left\lceil\frac{\theta_u}{\omega}\right\rceil\omega + \frac{\pi-\omega}{2} + \pi,$$

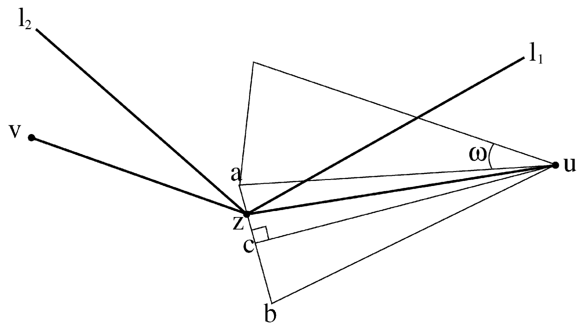


Fig. 3. The situation in the proof of Lemma 3.

and since $z \in ab$, v must belong to the halfplane of the line through ab not containing u. Since C_u is convex, $v \notin C_u$ implying that $d_\lambda(u,v) > d_\lambda(u,z)$. Symmetrically, $d_\lambda(u,v) > d_\lambda(v,z)$. Hence, z belongs to $L_\lambda(u,v)$ contradicting Lemma 1. □

By applying Lemmas 2 and 3 to a λ-Steiner hull containing λ-CH(Z) (pick say the entire plane) it is easy to show the following.

Lemma 4. *λ-CH(Z) is a λ-Steiner hull of Z.*

We have now shown the main result of this section.

Theorem 5. *λ-SH$'(Z)$ is a λ-Steiner hull of Z.*

A naive way of computing λ-SH$'(Z)$ is as follows. First we initialize $S = \lambda$-CH(Z). Then for each terminal in S and each λ-wedge W of z, we check if W bounds a safe region by computing the simple polygon R' of $W \cap S$ containing z and checking each terminal for inclusion in $R = R' \cap W^\circ$. If R contains no terminals, we set $S := S \setminus R$ and repeat the algorithm on S.

Recalling that a safe region is convex with a boundary consisting of uniformly oriented line segments, it can be determined whether a region is safe in $O(\lambda n)$ time. Since a terminal can be a base terminal $O(\lambda)$ times throughout the course of the algorithm and since there are $O(\lambda n)$ candidate safe regions in each iteration, it follows that the above algorithm can be implemented to run in $O((\lambda n)^3)$ time using $O(n)$ space. We will show how to find λ-SH$'(Z)$ more efficiently.

4. MST regions

Let Z be a terminal set. In the following, let M denote a fixed Euclidean MST for Z. The boundary subpaths of λ-CH(Z) together with the edges of M partition λ-CH(Z) into faces or *MST regions*.

We will show that computing λ-SH$'(Z)$ can be restricted to each MST region. The following lemma will prove useful.

Lemma 6. *$M \subseteq \lambda$-SH$'(Z)$.*

Proof. We show that if S is any partially constructed λ-SH$'(Z)$ then S contains M. The proof is by induction on the number $r \geqslant 0$ of safe regions removed. Since $M \subseteq$ CH$(Z) \subseteq \lambda$-CH(Z), this holds when $r = 0$.

Now suppose that after the removal of r safe regions, $M \subseteq S$. For the sake of contradiction, suppose there is a safe region R such that $M \nsubseteq S \setminus R$. Let z be the base terminal of R and suppose that, looking from z, r_1 respectively r_2 is the first point of intersection between the boundary of S and the left respectively right half-line of the safe λ-wedge bounding R.

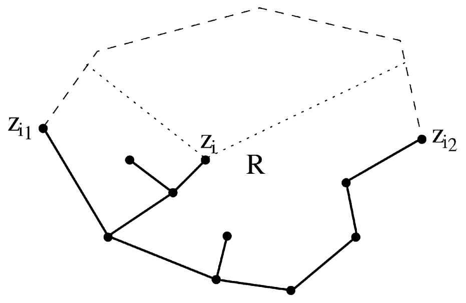


Fig. 4. Removing a safe region splits a subregion into two smaller subregions.

Since R contains no terminals, there must be an edge e of M crossing R. By the induction hypothesis, e must cross zr_1 in a point p and zr_2 in a point q. Let z_1, z_2 be end terminals of e such that z_1p and qz_2 are not contained in R.

If we remove e from M we split M into two components one containing say z_1 and z and the other containing z_2. Since $\angle pzq \geqslant \frac{\pi}{2}$, we also have $\angle z_1zz_2 \geqslant \frac{\pi}{2}$, implying that $|e| = |z_1z_2| > |z_2z|$. Thus, by adding edge (z_2, z), a new tree M' spanning Z is obtained and $|M'| < |M|$, a contradiction. Thus $M \subseteq S \setminus R$. $\square$

Consider a clockwise walk of M visiting the terminals of an MST region R_{MST} in the order $z_0, \dots, z_m$. A terminal may appear several times in this list since it may be visited more than once. Now consider a safe region R of λ-CH(Z) bounded by z_0, z_m and a base terminal z. By Lemma 6, R is fully contained in R_{MST}, hence $z = z_i$ for some $i \in \{0, \dots, m\}$. The removal of R separates R_{MST} into a subregion containing the terminals $z_0, \dots, z_i$ and a subregion containing the terminals $z_i, \dots, z_m$. Generalizing, we have

Theorem 7. *Consider a subregion induced by terminals $z_{i_1}, \dots, z_{i_2}$. If a safe region is bounded by z_{i_1}, z_{i_2}, and some base terminal z_i then $z_i \in \{z_{i_1}, \dots, z_{i_2}\}$. The removal of this safe region partitions the subregion into two smaller subregions, one containing $z_{i_1}, \dots, z_i$ and one containing $z_i, \dots, z_{i_2}$ (Fig.* 4).

Theorem 7 yields a recursive algorithm that removes safe regions from an MST region. Unfortunately, since we do not yet have a strategy for searching for base terminals, this result alone does not improve the $O((\lambda n)^3)$ asymptotic running time of our brute-force algorithm from Section 3. However, in Section 6 we shall present a clever strategy for finding base terminals.

5. Finding safe regions

Let M and R_{MST} be defined as in the preceding section. In this section we will show that, given a λ-wedge of a terminal of (a subregion of) R_{MST}, we can determine whether this λ-wedge bounds a safe region in constant time with $O(\lambda n \log n)$ preprocessing time. The idea is to use the fact that terminals in R_{MST} are all on the same path in the MST. Thus, instead of checking each terminal for inclusion in a candidate safe region, we simply check if the path crosses the boundary of that region. To do this efficiently, we will need the following definitions.

Let $z_k \notin \{z_0, z_m\}$ be a terminal of R_{MST}. Let $d \in \{0, \dots, 2\lambda - 1\}$ and let l be the half-line emanating from z_k with direction $d\omega$. If $e = (z_i, z_{i+1})$ is an edge of R_{MST} we say that *e is d-visible from z_k* if l avoids edges and terminals of R_{MST} before intersecting e in its interior looking from z_k, see Fig. 5. If $z_j \neq z_k$ is a terminal of R_{MST} we say that *z_j is d-visible from z_k* if l avoids edges and terminals of R_{MST} before intersecting z_j looking from z_k.

Let R be a subregion of R_{MST} induced by terminals $z_{i_1}, \dots, z_{i_2}$ and suppose that $z_k \in R$. To simplify the analysis, we assume that $z_k \notin \{z_{i_1}, z_{i_2}\}$; the case $z_k \in \{z_{i_1}, z_{i_2}\}$ is handled in a similar way. We make the following simple observations.

Any edge of R (i.e., an edge (z_i, z_j) with $i_1 \leqslant i, j \leqslant i_2$) has an oppositely directed edge in R if and only if it does not belong to the boundary of R, see Fig. 6. We say that an edge respectively terminal on the boundary of R *bounds* R. If z_k has an ingoing edge e in R that bounds R, this edge is unique and we refer to the endpoint of e opposite z_k as *in*(z_k). We define *out*(z_k) similarly.

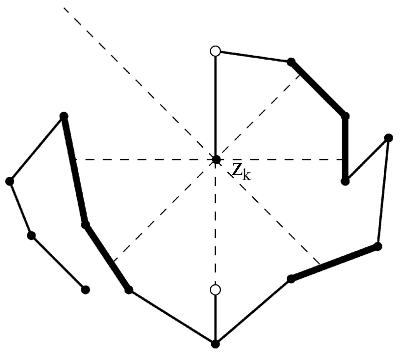


Fig. 5. Edges and terminals d-visible from z_k for $d = 0, \ldots, 7$ and $\lambda = 4$. Bold edges and white terminals are d-visible from z_k.

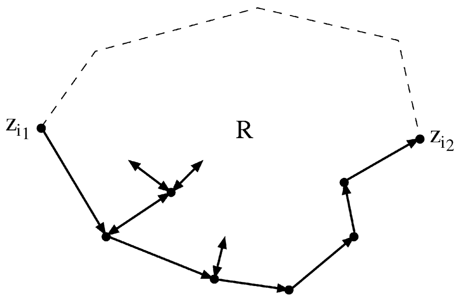


Fig. 6. An edge bounds R if and only if it has no oppositely directed edge. The boundary subpath between z_{i_1} and z_{i_2} is shown as dashed line segments.

As above, let l be the half-line emanating from z_k with direction $d\omega$. Let $e = (z_i, z_{i+1})$ be an edge of R which is d-visible from z_k and let l' be the line through e. Imagine walking along l starting at z_k. Then we cross e from the outside of R if and only if e bounds R and z_k belongs to the right halfplane of l' looking from z_i to z_{i+1}. If a terminal $z_j \notin \{z_{i_1}, z_{i_2}\}$ of R is d-visible from z_k then we cross z_j from the outside of R if and only if $in(z_j)$ and $out(z_j)$ exist and z_k belongs to $W(z_j, in(z_j), out(z_j))^\circ$.

Theorem 9 below relates the above definitions to safe regions. We need the following lemma.

Lemma 8. *With the above definitions, suppose that z_k is a base terminal of a safe region of R. If an edge e of R is d-visible from z_k then halfline l crosses e from the outside of R looking from z_k. If a terminal $z_j \notin \{z_{i_1}, z_{i_2}\}$ in R is d-visible from z_k then l crosses z_j from the outside of R looking from z_k.*

Proof. We only show the first part of the lemma. The second part is shown similarly. Let e be an edge of R which is d-visible from z_k. Suppose for the sake of contradiction that l does not cross e from the outside of R looking from z_k. Since l bounds a safe region with base terminal z_k, l must intersect the boundary subpath P between z_{i_1} and z_{i_2} (at least) twice since we leave R and then enter R again when moving from z_k to e. Let p respectively q be the first respectively second such intersection when looking from z_k. Let $p_0 = z_{i_1}$, $p_{r+1} = z_{i_2}$ and let $p_1, \ldots, p_r$ be the corner points of P when moving from z_{i_1} to z_{i_2}. Pick r_1 and r_2 such that p belongs to $p_{r_1}p_{r_1+1}$ and such that q belongs to $p_{r_2}p_{r_2+1}$.

Since the only concave angles of the boundary of the current λ-Steiner hull are at terminals, we must have the situation depicted in Fig. 7. Since no edges or terminals of R_{MST} belong to $P \setminus \{z_{i_1}, z_{i_2}\}$, it follows that R_{MST} is contained in polygon $S = pp_{r_1+1}p_{r_1+2}\cdots p_{r_2}qp$ and thus isolated from the rest of the MST M, a contradiction. □

We are now ready for the main result of this section. Theorem 9 below gives necessary and sufficient conditions for two half-lines to bound a safe region. The theorem assumes that terminals are in *general position*, which in this setting means that no two terminals are on the same uniformly oriented line. See Appendix A for details about how this restriction can be removed.

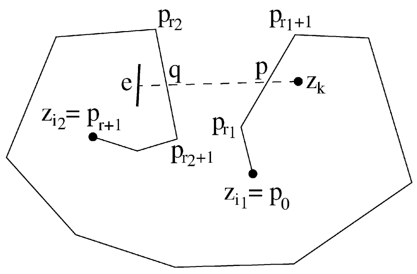


Fig. 7. The situation leading to a contraction in the proof of Lemma 8.

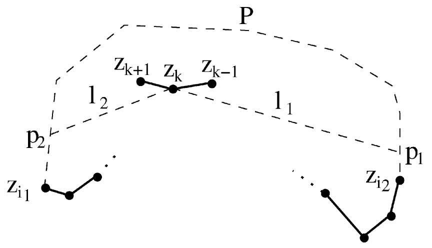


Fig. 8. The impossible situation in the proof of Theorem 9.

Theorem 9. *Let R and $z_k \neq z_{i_1}, z_{i_2}$ be defined as above. Let l_1 and l_2 be half-lines emanating from z_k and having directions $d_1\omega$ and $d_2\omega$ such that $W(l_2, l_1)$ is a λ-wedge. Then the following two statements are equivalent.*

(1) *$W(l_2, l_1) \subseteq W(z_k, z_{k+1}, z_{k-1})$ and for $m = 1, 2$, if an edge e in R is d_m-visible from z_k then l_m crosses e from the outside of R looking from z_k.*
(2) *$W(l_2, l_1)$ is a safe λ-wedge in R.*

Proof. Assume that (1) is satisfied. For $m = 1, 2$, let p_m be the first intersection point between $l_m \setminus \{z_k\}$ and an edge of R, a terminal of R, or the boundary subpath P between z_{i_1} and z_{i_2}. The existence of p_m follows from the assumption $W(l_2, l_1) \subseteq W(z_k, z_{k+1}, z_{k-1})$.

Since terminals are in general position, p_m cannot be a terminal. If p_m belonged to the interior of an edge $e = (z_j, z_{j+1})$ of R then e would be d_m-visible from z_k implying that l_m would cross e from the outside of R looking from z_k. But $z_k p_m \subseteq R$. For $m = 1, 2$ we conclude that p_m belongs to P.

We also need to check that, when moving from z_{i_1} to z_{i_2} along P, we first pass p_1 and then p_2. If we assume the opposite (see Fig. 8) then, since edges (z_{k-1}, z_k) and (z_k, z_{k+1}) are consecutive in the path of edges from z_{i_1} to z_{i_2} in R and terminals are in general position, there can be no path of edges connecting z_k and z_{i_1}, a contradiction.

The above shows that no edges cross the boundary of the candidate safe region R' bounded by $W(l_2, l_1)$. Thus, R' contains no terminals and so $(1) \Rightarrow (2)$. The other implication follows from Lemma 8. □

We find d-visible edges and terminals in R_{MST} using an $O(n \log n)$ time and $O(n)$ space sweep line algorithm for each of the 2λ values of d. The algorithm is straightforward and so we will not discuss it further.

Using Theorem 9 we can now determine in $O(1)$ time whether a terminal is a base terminal in (a subregion of) R_{MST} once d-visible edges and terminals in R_{MST} have been found. This improves the running time of our algorithm to $O((\lambda n)^2)$.

We will now turn our attention to λ-SH(Z) and show how to compute this set using only $O(\lambda n \log n)$ time and $O(\lambda n)$ space.

6. Parallel linear searches

In this section, we will describe an efficient way of searching through the terminals of a (subregion of) R_{MST}.

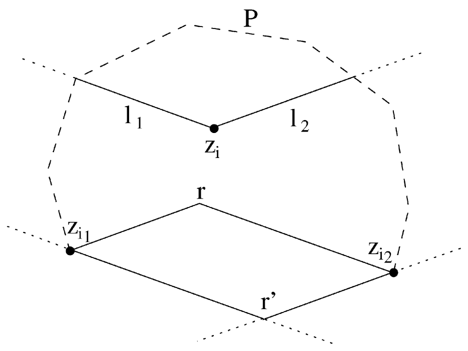


Fig. 9. The regions considered in the proof of Theorem 10.

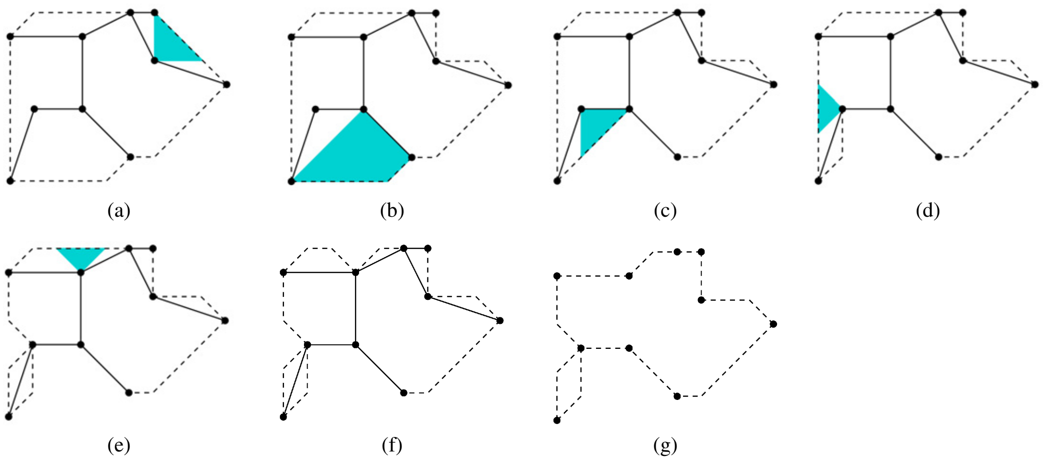


Fig. 10. Illustrating the algorithm for $\lambda = 4$. (a) Initial λ-Steiner hull λ-CH(Z). (b)–(f) Successive removals of safe regions. (g) λ-SH(Z). MST edges respectively boundary subpaths shown as solid respectively dashed line segments. Note that boundary subpaths are not maintained in the actual algorithm.

Recall that, in order to compute λ-SH(Z), we do not need the boundary subpaths of λ-SH$'(Z)$ but only the terminals on the boundary of λ-SH$'(Z)$ and the order in which they occur. Theorem 9 shows that we do not need to maintain the boundary subpaths throughout the course of the algorithm.

With the above in mind and using the recursive algorithm of Section 4, the overall algorithm that removes safe regions in R_{MST} is as follows. For any subregion R induced by terminals $z_{i_1}, \ldots, z_{i_2}$ we find a base terminal in R and recursively remove safe regions in the two new subregions. If no base terminal is found we terminate.

We check for base terminals in the order $z_{i_1+1}, z_{i_2-1}, z_{i_1+2}, z_{i_2-2}, \ldots$. In effect, we perform two linear searches in parallel, one visiting terminals in the order $z_{i_1+1}, z_{i_1+2}, \ldots$ and one visiting the terminals in the order $z_{i_2-1}, z_{i_2-2}, \ldots$. The idea is that a long search time is compensated for by an even (good) split of the subregion (or termination if no base terminal exists) whereas an uneven (bad) split is compensated for by a short search time.

Note that we no longer check if z_{i_1} and z_{i_2} are base terminals since safe regions with either of these terminals as base terminals would only affect the boundary subpaths.

Fig. 10 illustrates the various steps of the algorithm on an instance consisting of ten terminals. Before proving time and space bounds for this algorithm, we need the following theorem which shows that λ-SH(Z) is in fact a λ-Steiner hull of Z.

Theorem 10. *The set λ-SH(Z) is a λ-Steiner hull of Z.*

Proof. Let R be a subregion of an MST region in λ-SH$'(Z)$ and let $z_{i_1}, \ldots, z_{i_2}$ be the terminals of R. Let $z_{i_1} r z_{i_2}$ be the critical path from z_{i_1} to z_{i_2} making a right turn at r and let $z_{i_1} r' z_{i_2}$ be the critical path from z_{i_1} to z_{i_2} making a left turn at r'. Let P be the boundary subpath in λ-SH$'(Z)$ between z_{i_1} and z_{i_2}.

We claim that no terminal belongs to the interior of the bounded region R' bounded by P and $z_{i_1}r'z_{i_2}$. For suppose that z_i is such a terminal. Let l_1 respectively l_2 be the half-line emanating from z_i having the same direction as the half-line emanating from r' and intersecting z_{i_1} respectively z_{i_2}, see Fig. 9.

We may assume that z_i is the only terminal in $W(l_2, l_1) \cap R'$. But then $W(l_2, l_1)$ contains a safe region with base terminal z_i, contradicting the fact that no safe regions can be removed from λ-SH(Z). We conclude that the interior of R' contains no terminals.

Now let R'' be the bounded region bounded by P and $z_{i_1}rz_{i_2}$. We will show that the interior of R'' contains no part of any λ-SMT for Z. By the above, the interior of R'' contains no terminals of Z and by using a similar argument as in Lemma 2, it follows that R'' contains no Steiner points of any λ-SMT for Z. It is then easy to see that no edges of any λ-SMT for Z intersect the interior of R''.

Applying the above to each subregion of λ-SH$'(Z)$ shows the theorem. □

7. Running time and space requirement

In this section, we show that our algorithm computing λ-SH(Z) has worst-case running time $O(\lambda n \log n)$ and $O(\lambda n)$ space requirement where n is the number of terminals. We show that, regarding λ as constant, this is optimal.

Theorem 11. *The algorithm presented above has $O(\lambda n \log n)$ worst-case running time.*

Proof. We can find CH(Z), M, and the MST regions of M in $O(n \log n)$ time. Consider any MST region R and let r be the number of terminals (with repetitions) on the subpath in R induced by the clockwise walk of M. We need to show that it takes $O(\lambda r \log r)$ time to remove safe regions from R.

Since we make 2λ calls to the sweep line algorithm, the total time spent on this is $O(\lambda r \log r)$. Now let $t(k)$ denote the highest number of terminals checked in any subregion R' of R containing exactly k terminals. Here we also count terminals checked in recursive calls to subregions of R'. We claim that

$$t(k) \leqslant (2k-3)\lg k - 1. \tag{1}$$

We show (1) by induction on $k \geqslant 2$. The base case is trivial since then we perform no checks. Now let $k > 2$ and assume that (1) holds for all values smaller than k. To show (1) for k, suppose first that R' contains no base terminals. Then we search through $k-2$ terminals before terminating, i.e., we search through $k-2$ terminals.

Now suppose instead that we find a base terminal in R' after having checked i terminals. Since we search in parallel from both ends of the path in R', we split R' into one subregion containing $\lfloor (i+1)/2 \rfloor + 1$ terminals and one subregion containing $k - \lfloor (i+1)/2 \rfloor$ terminals.

By the above,

$$t(k) \leqslant \max\Bigl\{k-2, \max_{i=1,\ldots,k-2}\bigl\{t\bigl(\lfloor (i+1)/2 \rfloor + 1\bigr) + t\bigl(k - \lfloor (i+1)/2 \rfloor\bigr) + i\bigr\}\Bigr\}.$$

Using the induction hypothesis, it can be shown that the right-hand side is at most $(2k-3)\lg k - 1$. This completes the induction. Thus, in all parallel linear searches we check at most $O(r \log r)$ terminals for a given direction. Clearly the time to check the first set of statements in the generalized version of Theorem 9 is at most a constant times the maximum possible degree of any node in an MST. It is well known that this degree is six [6] and since we need to check $O(\lambda)$ directions, the total time spent on removing safe regions in R is $O(\lambda r \log r)$. □

Theorem 12. *The algorithm presented above has $O(\lambda n)$ space requirement.*

Proof. MST M and CH(Z) require $O(n)$ storage. Since the (clockwise) walk of M has length $O(n)$ we can represent all paths of terminals encountered in the algorithm using a total of $O(n)$ space. Each terminal has $O(\lambda)$ d-visible terminals and edges. The space requirement for all calls to the sweep line algorithm is $O(n)$. Clearly, we can represent λ-SH(Z) using $O(n)$ space. This shows that the entire algorithm uses $O(\lambda n)$ space. □

Theorem 13. *For constant λ, the algorithm presented above is optimal.*

Proof. Clearly, any algorithm that computes λ-SH(Z) must use $\Omega(n)$ space. The terminals of Z on the boundary of the initial λ-Steiner hull λ-CH(Z) of the algorithm are exactly the terminals belonging to the boundary of CH(Z). Since these terminals remain on the boundary of the partially constructed λ-Steiner hull throughout the course of the algorithm, the boundary of λ-SH(Z) must also contain all terminals belonging to the boundary of CH(Z).

Clearly, the boundary of λ-SH(Z) is a simple polygon with $O(n)$ vertices. Since the convex hull of the vertices on a simple polygon with $O(n)$ vertices can be determined in $O(n)$ time [5], it follows that any algorithm computing λ-SH(Z) uses $\Omega(n \log n)$ time. □

8. Uniqueness of λ-SH(Z)

In this section, we show that λ-SH(Z) is uniquely defined in the sense that it does not depend on the chosen maximal sequence of safe removals from λ-CH(Z). The uniqueness proof is quite similar to that in [7] for the Steiner hull in the Euclidean metric.

We let $C(z_{i_1}, z_{i_2})$ denote the path of terminals encountered when walking along the boundary of λ-SH(Z) starting in z_{i_1} and ending in z_{i_2} where z_{i_1} and z_{i_2} belong to the same MST region. We need the following two lemmas.

Lemma 14. *Let R be a subregion induced by terminals $z_{i_1}, \ldots, z_{i_2}$. If z_k is a base terminal in R then $z_k \in C(z_{i_1}, z_{i_2})$.*

Proof. Suppose the lemma does not hold. Then there exists a maximal sequence of safe removals from R such that λ-SH(Z) contains a subregion R' in R induced by terminals $z_{j_1}, \ldots, z_{j_2}$ where $j_1 < k < j_2$. Let S be the safe region in R bounded by z_{i_1}, z_{i_2}, and z_k and let W be the λ-wedge bounding S. Then W is safe in R', a contradiction. □

If k is the smallest index such that z_k is a base terminal in subregion R then the removal of the corresponding safe region in R is called *canonical*. A maximal sequence of safe removals from R is said to be *canonical* if all its safe removals are canonical.

Lemma 15. *If λ-SH(Z) is obtained by some maximal sequence of safe removals from λ-CH(Z) then the same polygon can be obtained by a canonical sequence.*

The proof of Lemma 15 is in Appendix A. We now present the main result of this section.

Theorem 16. *λ-SH(Z) does not depend on the chosen maximal sequence of safe removals from λ-CH(Z).*

Proof. This follows from Lemma 15 and the fact that every canonical sequence of safe removals from λ-CH(Z) yields the same λ-SH(Z). □

9. Concluding remarks

In this paper, we defined a region λ-SH(Z) known to contain every λ-SMT for Z. Letting $n = |Z|$, we presented an $O(\lambda n \log n)$ time and $O(\lambda n)$ space algorithm that computes this set by removing open wedge-shaped regions from an initial hull. We proved that our algorithm is optimal in both time and space for constant λ and showed that λ-SH(Z) is independent of the order of removals of open wedge-shaped regions.

A possible improvement to the algorithm would be to flip suitable critical paths of λ-SH(Z). This would yield a smaller hull (which would not contain every λ-SMT but at least one) but it would not increase the number of terminals on the boundary of the hull. However, it would restrict the feasible locations of Steiner points further thus possibly making it easier to compute a λ-SMT for Z.

Acknowledgements

I would like to thank Pawel Winter for his comments and remarks.

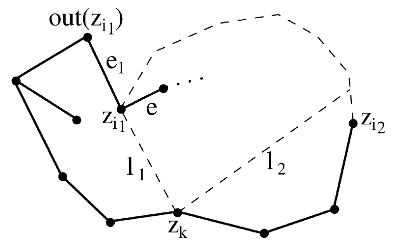


Fig. A.1. The situation in the proof of the generalized version of Theorem 9.

Appendix A. Theorem 9 generalized

In Theorem 9 we assumed that terminals were in general position. In this section we show how to remove this restriction. So suppose that the terminals of Z have arbitrary locations. We claim that, with the definitions in Theorem 9, the following two sets of statements are equivalent.

(1) z_1 is not d_2-visible from z_k and z_2 is not d_1-visible from z_k.
For $m = 1, 2$, if an edge e in R is d_m-visible from z_k then l_m crosses e from the outside of R looking from z_k.
For $m = 1, 2$, if a terminal $z_j \notin \{z_{i_1}, z_{i_2}\}$ in R is d_m-visible from z_k then l_m crosses z_j from the outside of R looking from z_k.
If z_{i_1} is d_1-visible from z_k then $W(z_{i_1}, z_k, out(z_{i_1}))^\circ$ contains no edges ending in z_{i_1}.
If z_{i_2} is d_2-visible from z_k then $W(z_{i_2}, in(z_{i_2}), z_k)^\circ$ contains no edges ending in z_{i_2}.
(2) $W(l_2, l_1)$ is a safe λ-wedge in R.

Assume that (1) is satisfied. To show (2) we only consider the cases not covered by Theorem 9. Letting p_m be as in the proof of Theorem 9, suppose for the sake of contradiction that p_m is a terminal z_j. The third statement in (1) implies that $z_j \in \{z_{i_1}, z_{i_2}\}$. If $z_j = z_{i_1}$ then l_1 must intersect z_{i_1} and z_{i_1} is d_1-visible from z_k.

Let e_1 be the edge directed from z_{i_1} to $out(z_{i_1})$. The right side of e_1 looking from z_{i_1} belongs to the outside of R. Since the interior of $z_k z_{i_1}$ contains no terminals then, as shown in Fig. A.1, e must intersect the interior of $W(z_{i_1}, z_k, out(z_{i_1}))$, contradicting (1). We show $z_j \neq z_{i_2}$ similarly.

We can exclude the situation in Fig. 8 since z_1 is not d_2-visible from z_k and z_2 is not d_1-visible from z_k. Thus, (1) $\Rightarrow$ (2).

Now suppose that (2) holds. Since l_1 is the left leg of W, l_1 cannot intersect z_{i_1} and l_2 cannot intersect z_{i_2}. For $m = 1, 2$ we have, by Lemma 8, that if a terminal $z_j \notin \{z_{i_1}, z_{i_2}\}$ in R is d_m-visible from z_k then l_m crosses z_j from the outside of R looking from z_k.

Finally suppose that say z_{i_1} is d_1-visible from z_k. No edges of R can cross the safe region in R bounded by $W(l_2, l_1)$. In particular, no edges in R with endpoint in z_{i_1} can cross this safe region. It follows that $W(z_{i_1}, z_k, out(z_{i_1}))^\circ$ contains no edges with endpoint in z_{i_1}. This shows (2) $\Rightarrow$ (1).

Appendix B. Proof of Lemma 15

Let S be any maximal sequence of safe removals from λ-CH(Z) and suppose that the first safe removal in S of some safe region R, bounded by z_{i_1}, z_{i_2}, and some base terminal, is not canonical. We will show that we can substitute R by the canonical removal of a safe region R' bounded by z_{i_1}, z_{i_2}, and a base terminal z_k, followed by an appropriate maximal sequence of safe removals such that λ-SH$'(Z)$ remains the same. By repeating this procedure a sufficient number of times, the lemma follows.

By Lemma 14, $z_k \in C(z_{i_1}, z_{i_2})$ when applying sequence S. Let S' denote the subsequence of S beginning with the removal of R and ending with z_k being added to the boundary.

Let S'' denote the subsequence of S' consisting of the safe removals of regions bounded by terminals of the form z_{j_1}, z_{j_2}, and some base terminal where $i_1 \leqslant j_1 < k < j_2 \leqslant i_2$. The sequence S'' followed by the sequence $S' \setminus S''$ yields the same boundary as S'. Hence, we may assume that S'' is a prefix of S'.

We start by removing R'. This splits our MST region into a subregion R_1 containing $z_{i_1}, \ldots, z_k$ and a subregion R_2 containing $z_k, \ldots, z_{i_2}$. Suppose $C(z_{i_1}, z_k)$ has an intermediate terminal and let z_h be the successor of z_{i_1} in $C(z_{i_1}, z_k)$. There is a safe region $\bar{R}$ bounded by z_{i_1}, z_j, and base terminal z_h for some $j \in \{k+1, \ldots, i_2\}$. If W is the λ-wedge bounding $\bar{R}$ then W is safe in R_1. We remove the corresponding safe region from R_1 and repeat the procedure on $C(z_h, z_k)$ if it has intermediate terminals.

We can apply the same procedure to $C(z_k, z_{i_2})$. Thus, we have modified our sequence S into another sequence starting with a canonical removal without affecting the resulting λ-SH$'(Z)$.

References

[1] M. Brazil, D.A. Thomas, J.F. Weng, Minimum networks in uniform orientation metrics, SIAM Journal on Computing 30 (5) (2000) 1579–1593.

[2] M. Brazil, D.A. Thomas, J.F. Weng, M. Zachariasen, Canonical forms and algorithms for steiner trees in uniform orientation metrics, Algorithmica 44 (2006) 281–300.

[3] F.K. Hwang, D.S. Richards, P. Winter, The Steiner Tree Problem, North-Holland, 1992.

[4] A.B. Kahng, G. Robins, On Optimal Interconnections for VLSI, Kluwer Publishers, 1995.

[5] D.T. Lee, On finding convex hull of a simple polygon, International Journal of Computing and Information Sciences 12 (1983) 87–98.

[6] G. Robins, J.S. Salowe, On the maximum degree of minimum spanning trees, in: Symposium on Computational Geometry, 1994, pp. 250–258.

[7] P. Winter, Optimal Steiner hull algorithm, Computational Geometry 23 (2002) 163–169.

Wiener Index and Diameter of a Planar Graph in Subquadratic Time

Christian Wulff-Nilsen*

Abstract

We solve two open problems by proving the existence of subquadratic time algorithms for computing the Wiener index, defined as the sum of all-pairs shortest path distances, and the diameter, defined as the maximum distance between any vertex pair, of an unweighted planar graph. We do this by exhibiting algorithms with $O(n^2 \log\log n/\log n)$ running time and $O(n)$ space requirement where n is the number of vertices of the graph.

1 Introduction

A *molecular topological index* is a value obtained from the graph structure of a molecule such that this value (hopefully) correlates with physical and/or chemical properties of the molecule. Perhaps the most studied molecular topological index is the so called *Wiener index*, a generalization of a definition given by Wiener in 1947 [8]. The Wiener index of a graph (weighted or unweighted) is defined as the sum of distances between all pairs of vertices of the graph. Computing the Wiener index is essentially equivalent to computing the average distance between vertex pairs.

The Wiener index can clearly be computed in the amount of time it takes to compute APSP distances. For special types of graphs, faster algorithms are known. Linear time algorithms are known for cactii [9] and benzenoid systems [2] and recently Cabello and Knauer [1] gave near-linear time algorithms for graphs of bounded treewidth. More specifically, they showed that the Wiener index of an n-vertex graph of treewidth $k \geq 3$ can be found in $O(n \log^{k-1} n)$ time. All bounds hold for graphs with arbitrary non-negative edge weights.

For planar graphs, the Wiener index can be found in quadratic time using the algorithm of Frederickson [4]. One of the main open problems in this context concerns the existence of a subquadratic time algorithm for such graphs.

Another important quantity is the *diameter* of a graph, defined as the maximum distance between any vertex pair. With Frederickson's algorithm, the diameter of a planar graph can be found in quadratic time but it is open whether a subquadratic time algorithm exists (Problem 6.2 in [3]).

*Department of Computer Science, University of Copenhagen, koolooz@diku.dk

In this paper, we solve both of these open problems for planar unweighted graphs. We do this by exhibiting algorithms with running time $O(n^2 \log\log n/\log n)$ and space requirement $O(n)$ where n is the number of vertices.

The organization of the paper is as follows. In Section 2, we give various definitions and introduce some notation. We mention a result by Frederickson [4] in Section 3, a result that allows us to divide a planar graph into regions with some nice properties. In Section 4, we rely on this result to obtain our subquadratic time algorithm for computing the Wiener index of a planar graph. In Section 5, we show how a simple modification gives an algorithm with the same time bound for computing the diameter of a planar graph. Finally, we make some concluding remarks in Section 6.

2 Definitions and Notation

In all the following, $G = (V, E)$ is an unweighted planar graph with n vertices. For $u, v \in V$, we let $d_G(u, v)$ denote the length of a shortest path in G between u and v.

Given a subgraph H of G, we let V_H denote its vertex set.

Given subsets $U_1, U_2 \subseteq V$, we define

$$\sum(U_1, U_2) = \sum_{u \in U_1} \sum_{v \in U_2} d_G(u, v).$$

We omit G in the notation but this should not cause any confusion. For a vertex u and a subset U of V, we write $\sum(U, u)$ as a shorthand for $\sum(U, \{u\})$.

We let $\sum G$ denote the sum of all-pairs shortest path distances in G, i.e. $\sum G = \frac{1}{2}\sum(V, V)$, and we refer to it as the *Wiener index* of G.

A *region* of G is a subset R of vertices of G. A *boundary vertex* of R is a vertex in R which is adjacent in G to a vertex in $V \setminus R$. Vertices of R that are not boundary vertices are called *interior vertices* (of R).

We let log denote the base 2 logarithm.

3 r-division of a Planar Graph

By applying the separator theorem of Lipton and Tarjan [6] recursively to a given planar graph, Frederickson [4] obtained the following result which we state as a lemma.

Lemma 1 *Given a parameter $r \in (0, n)$ (which may depend on n), an n-vertex planar graph can be divided into $\Theta(n/r)$ regions each of which contains at most r vertices and $O(\sqrt{r})$ boundary vertices. Furthermore, each interior vertex is contained in exactly one region. Finding such a division can be done in $O(n \log n)$ time.*

For parameter r, we refer to the division in Lemma 1 as an *r-division* (of the graph). If $R_1, \ldots, R_k \subseteq V$ are the regions obtained, we denote the r-division by the tuple $(R_1, \ldots, R_k)$.

Finding an r-division for a suitable value of r is essential in our algorithms.

4 Wiener Index of a Planar Graph

In this section, we show how to compute the Wiener index $\sum G$ of G in $O(n^2 \log\log n/\log n)$ time. We will assume that G is connected since otherwise the problem is trivial.

The first step of our algorithm is to compute an r-division $(R_1, \ldots, R_k)$ of G for some parameter r which we specify later. For now, just regard r as some function of n. We will show how to compute $\sum G$ in $O(n^2/\sqrt{r} + nr^{O(\sqrt{r})})$ time. From this and from a suitable choice of r, the first result of the paper will follow.

In the following, let B be the set of boundary vertices over all regions $R_1, \ldots, R_k$. We precompute shortest path distances from each vertex in B to all vertices in V. Since $|B| = O(n/\sqrt{r})$, this can be done in $O(n^2/\sqrt{r})$ time using the linear time SSSP algorithm in [5] for each vertex in B. From these distances, we obtain values $\sum(B, V)$ and $\sum(B, B)$ in $O(n^2/\sqrt{r})$ time.

Observe that $\sum(B, V) - \frac{1}{2}\sum(B, B)$ is the sum of all shortest path distances in G between vertex pairs (u, v) for which either u or v (or both) is a boundary vertex. Since, by Lemma 1, each interior vertex belongs to exactly one region, we can thus obtain $\sum G$ as the sum

$$\sum(B, V) - \frac{1}{2}\sum(B, B) +$$
$$\frac{1}{2}\sum_{i=1}^{k}\sum(R_i \setminus B, V \setminus (R_i \cup B)) + \sum(R_i \setminus B, R_i \setminus B).$$

Let R be one of the regions $R_1, \ldots, R_k$. In the following, we focus on the problem of computing $\sum(R \setminus B, V \setminus (R \cup B))$ and $\sum(R \setminus B, R \setminus B)$. If we can show that these two quantities can be computed in $O(n\sqrt{r} + r^{O(\sqrt{r})})$ time, it will follow that $\sum G$ can be computed in $O(n^2/\sqrt{r} + nr^{O(\sqrt{r})})$ time since $k = \Theta(n/r)$.

Let us start with the easy part, that of computing $\sum(R \setminus B, R \setminus B)$. We do this by computing shortest path distances in G between each pair of vertices in R. To do this efficiently, we take the subgraph of G induced by R and add to it an edge between each pair of boundary vertices of R; the length of this edge is equal to the distance in G between those two vertices (we do not add an edge between a pair of boundary vertices already connected by an edge). We then run an APSP algorithm like Floyd-Warshall on the resulting graph.

Since shortest path distances from boundary vertices of R to all vertices in G (and in particular to all boundary vertices in R) have been precomputed, it follows that we can compute shortest path distances in G between each pair of vertices in R in $O(r^3)$ time. Hence, we can compute $\sum(R \setminus B, R \setminus B)$ in $O(r^3)$ time.

Now, to compute $\sum(R \setminus B, V \setminus (R \cup B))$, let $C_1, \ldots, C_s$ be the connected components of the subgraph of G induced by R. Then

$$\sum(R \setminus B, V \setminus (R \cup B)) =$$
$$\sum_{i=1}^{s}\sum(V_{C_i} \setminus B, V \setminus (R \cup B)). \qquad (1)$$

Let C be one of these connected components, let $n_C = |V_C|$, and let $p_1, \ldots, p_t$ be the boundary vertices of R belonging to C. In the following, we show how to compute $\sum(V_C \setminus B, V \setminus (R \cup B))$ in $O(nt + n_C^{O(t)})$ time. It will then follow that the left-hand side of (1) can be computed in $O(n\sqrt{r} + r^{O(\sqrt{r})})$ time since each of the $O(\sqrt{r})$ boundary vertices of R belongs to exactly one connected component.

To compute $\sum(V_C \setminus B, V \setminus (R \cup B))$, the basic idea is the following. Given some vertex $u \in V \setminus (R \cup B)$, suppose we have precomputed, for each boundary vertex p_i, $i = 1, \ldots, t$,

1. the number $n_{i,u}$ of vertices v in $V_C \setminus B$ for which i is the smallest j such that $d_G(u, v) = d_G(u, p_j) + d_C(p_j, v)$,
2. the sum $D_{i,u}$ of distances in C from p_i to each of these vertices in $V_C \setminus B$.

Then

$$\sum(V_C \setminus B, u) = \sum_{i=1}^{t} n_{i,u} d_G(u, p_i) + D_{i,u}, \qquad (2)$$

see Figure 1.

Given these precomputations, we can thus obtain $\sum(V_C \setminus B, u)$ in $O(t)$ time and from this it follows that $\sum(V_C \setminus B, V \setminus (R \cup B))$ can be computed in $O(nt)$ time.

In order to perform the above precomputations efficiently we need the following key observation.

Lemma 2 *Let $u \in V \setminus (R \cup B)$ and let $i \in \{1, \ldots, t\}$ be given. Then $n_{i,u}$ and $D_{i,u}$ are completely determined by shortest path distances in C and values $d_G(u, p_j) - d_G(u, p_1)$ for $j = 1, \ldots, t$.*

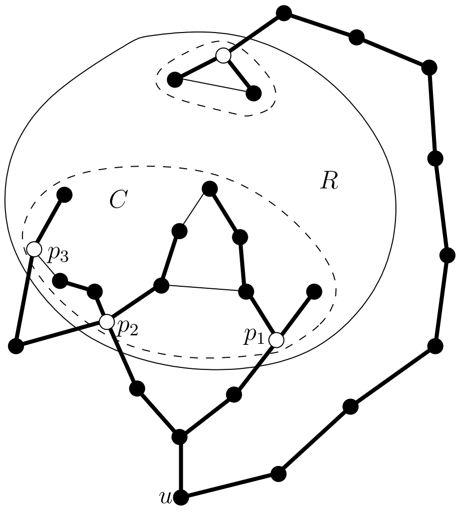


Figure 1: Graph instance for which component C has $t = 3$ boundary vertices p_1, p_2, and p_3 of R and where $(n_{1,u}, D_{1,u}) = (4,7)$, $(n_{2,u}, D_{2,u}) = (4,6)$, and $(n_{3,u}, D_{3,u}) = (1,1)$. Also $\sum(V_C \setminus B, u) = \sum_{i=1}^{t} n_{i,u} d_G(u, p_i) + D_{i,u} = 43$.

Proof. Let $v \in V_C \setminus B$. Any path from u to v in G must contain at least one of the boundary vertices $p_1, \ldots, p_t$. Hence, the following two conditions are equivalent:

1. i is the smallest j such that $d_G(u,v) = d_G(u,p_j) + d_C(p_j,v)$,
2. $d_G(u,p_i) - d_G(u,p_j) \le d_C(v,p_j) - d_C(v,p_i)$ holds for $1 \le j \le t$ with strict inequality for $1 \le j < i$.

Since $d_G(u,p_i) - d_G(u,p_j) = (d_G(u,p_i) - d_G(u,p_1)) - (d_G(u,p_j) - d_G(u,p_1))$, the above shows that $n_{i,u}$ depends only on shortest path distances in C and values $d_G(u,p_j) - d_G(u,p_1)$, $j = 1, \ldots, t$. Clearly, this also holds for $D_{i,u}$. □

Before proceeding, let us define a map $\phi : V \setminus (R \cup B) \to \mathbb{Z}^t$ by

$$\phi(u)[j] = d_G(u,p_j) - d_G(u,p_1),$$

for $j = 1, \ldots, t$. Let p be a point in $\phi(V \setminus (R \cup B))$ and let u be a vertex in $V \setminus (R \cup B)$ such that $\phi(u) = p$. Associate with p values $n_p(i)$ and $D_p(i)$ for $i = 1, \ldots, t$, defined by

$$n_p(i) = n_{i,u},$$
$$D_p(i) = D_{i,u}.$$

By Lemma 2, this is well-defined since $n_p(i)$ and $D_p(i)$ do not depend on the choice of $u \in \phi^{-1}(\{p\})$.

The strategy now is to precompute n_p- and D_p-values for each $p \in \phi(V \setminus (R \cup B))$ and then, for each $u \in V \setminus (R \cup B)$, compute $p = \phi(u)$ and obtain, for $i = 1, \ldots, t$, $n_{u,i}$ and $D_{u,i}$ as the precomputed values $n_p(i)$ and $D_p(i)$, respectively. Function ϕ will act in a way similar to a hash function in that it maps a key (a vertex u) into a hash (the point $p = \phi(u)$) to obtain a value ($n_p(i)$ and $D_p(i)$ for $i = 1, \ldots, t$).

For this strategy to work well, we need the following lemma which shows that the number of points in $\phi(V \setminus (R \cup B))$ is small compared to $V \setminus (R \cup B)$ and hence that we only need to compute a small number of n_p- and D_p-values.

Lemma 3 $\phi(V \setminus (R \cup B)) \subseteq \{-(n_C - 1), \ldots, n_C - 1\}^t$.

Proof. Let $u \in V \setminus (R \cup B)$ and let $j \in \{1, \ldots, t\}$ be given. Since C is connected there is a simple path in C from p_1 to p_j and this path consists of at most $n_C - 1$ edges. Thus, $d_G(p_1,p_j) \le d_C(p_1,p_j) \le n_C - 1$ and the triangle inequality implies

$$-(n_C - 1) \le d_G(u,p_j) - d_G(u,p_1) \le n_C - 1.$$

This shows the lemma since $\phi(u)[j] = d_G(u,p_j) - d_G(u,p_1)$. □

We are now ready to describe how to compute $\sum(V_C \setminus B, V \setminus (R \cup B))$. We first initialize a t-dimensional table T with an entry $T[p]$ for each point $p \in \{-(n_C - 1), \ldots, n_C - 1\}^t$. Associated with $T[p]$ are two t-dimensional vectors to hold values $(n_p(1), \ldots, n_p(t))$ and $(D_p(1), \ldots, D_p(t))$. Initially, all entries of T are unmarked. The initialization step takes a total of $O(t(2n_C - 1)^t) = O(n_C^{O(t)})$ time.

For each $u \in V \setminus (R \cup B)$, we compute point $p = \phi(u)$ in $O(t)$ time (this is possible since SSSP distances in G have been precomputed for each boundary vertex). Assume first that entry $T[p]$ is unmarked. Then we mark it and compute the n_p- and D_p-values and store them in the vectors associated with $T[p]$. By Lemma 2, computing and storing these values can clearly be done in time polynomial in n_C (a weak analysis but it suffices). From these values, we compute $\sum(V_C \setminus B, u)$ in $O(t)$ time.

If $T[p]$ is already marked then we do not compute n_p- and D_p-values. Instead we perform a lookup in T at entry $T[p]$ to obtain $\sum(V_C \setminus B, u)$ in $O(t)$ time.

Clearly, we compute n_p- and D_p-values at most once for each $p \in \{-(n_C - 1), \ldots, n_C - 1\}^t$. It follows that the above algorithm computes $\sum(V_C \setminus B, V \setminus (R \cup B))$ in $O(nt + n_C^{O(t)})$ time.

From the above and from (1), we get the following result.

Lemma 4 *For each region R, values $\sum(R \setminus B, V \setminus (R \cup B))$ and $\sum(R \setminus B, R \setminus B)$ can be computed in $O(n\sqrt{r} + r^{O(\sqrt{r})})$ time assuming shortest path distances from each boundary vertex of R to each vertex in G have been precomputed.*

We are now ready for our first result.

Theorem 5 *The Wiener index $\sum G$ of an unweighted planar n-vertex graph G can be computed in $O(n^2 \log\log n/\log n)$ time and $O(n)$ space.*

Proof. Computing shortest path distances from each boundary vertex to each vertex in G can be done in $O(n^2/\sqrt{r})$ time.

Applying Lemma 4 to each of the $\Theta(n/r)$ regions in the r-division of G gives us $\sum G$ in $O(n^2/\sqrt{r}+nr^{c'\sqrt{r}})$ time for some constant c'.

We pick $r = (c\log n/\log\log n)^2$ where $c > 0$ is some constant (to be specified). For $n > 2^c$ we have $\log n > c$, and for $n > 4$ we have $\log\log n > 1$. Thus, for $n > \max\{2^c, 4\}$,

$$\begin{aligned} r^{c'\sqrt{r}} &< (c\log n)^{2c'c\log n/\log\log n} \\ &< (\log n)^{4c'c\log n/\log\log n} \\ &= n^{4c'c}, \end{aligned}$$

since $(\log n)^{\log n/\log\log n} = n$. It follows that if we choose $c < 1/(4c')$, we have $r^{c'\sqrt{r}} = O(n^{\epsilon})$, where $\epsilon < 1$. With this choice of r, the total running time of the algorithm is

$$\begin{aligned} O(n^2/\sqrt{r} + nr^{c'\sqrt{r}}) &= O(n^2\log\log n/\log n + n^{1+\epsilon}) \\ &= O(n^2\log\log n/\log n), \end{aligned}$$

as requested.

Simple modifications of the algorithm described above allows us to obtain linear space requirement without affecting running time. Due to space constraints, we omit the details. □

5 Diameter of a Planar Graph

The following theorem shows that we can obtain a similar time bound for computing the diameter of an unweighted planar graph.

Theorem 6 *The diameter of an n-vertex planar graph with non-negative edge weights can be computed in $O(n^2\log\log n/\log n)$ time and $O(n)$ space.*

Proof. We can apply ideas similar to those above for the Wiener index problem. The only essential difference is that instead of variables $n_{i,u}$ and $D_{i,u}$ (see Section 4) we introduce variables $L_{i,u}$ for each i and u that keep track of the longest distance in C from p_i to the set of vertices specified in the definition of $n_{i,u}$. We can then use the identity

$$\max(V_C \setminus B, u) = \max_{1\le i\le t}\{d_G(u,p_i) + L_{i,u}\}$$

instead of (2). □

Note that our two algorithms do not rely on planarity except when computing an r-division and SSSP distances in linear time. Our results can therefore be extended to the larger class of unweighted subgraph-closed $\sqrt{n}$-separable graphs for which separators can be found efficiently [5].

6 Conclusion

We solved two open problems, the existence of subquadratic time algorithms for computing the Wiener index and diameter of an unweighted planar graph. We did this by exhibiting $O(n^2\log\log n/\log n)$ time algorithms where n is the number of vertices. Both algorithms have linear space requirement.

It remains open whether "truly" subquadratic time algorithms exist, i.e. algorithms with $O(n^c)$ running time for constant $c < 2$.

In a forthcoming paper, we extend our results to planar graphs with arbitrary non-negative edge weights. The new ideas we introduce in that paper allow us to solve also the stretch factor problem for planar graphs in subquadratic time. In another paper, we show how similar ideas give a faster algorithm for computing shortest path distances between $k = O(n^2)$ pairs of vertices in a planar graph for large k.

References

[1] S. Cabello and C. Knauer. *Algorithms for graphs of bounded treewidth via orthogonal range searching.* Manuscript, Berlin, 2007.

[2] V. Chepoi and S. Klavžar. *The Wiener index and the Szeged index of benzenoid systems in linear time.* J. Chem. Inf. Comput. Sci., 37:752–755, 1997.

[3] F. R. K. Chung. *Diameters of Graphs: Old Problems and New Results.* Congressus Numerantium, 60:295–317, 1987.

[4] G. N. Frederickson. *Fast algorithms for shortest paths in planar graphs, with applications.* SIAM J. Comput., 16 (1987), pp. 1004–1022.

[5] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. *Faster Shortest-Path Algorithms for Planar Graphs.* Journal of Computer and System Sciences volume 55, issue 1, August 1997, pages 3–23.

[6] R. J. Lipton and R. E. Tarjan. *A Separator Theorem for Planar Graphs.* STAN-CS-77-627, October 1977.

[7] B. Mohar and T. Pisanski. *How to compute the Wiener index of a graph.* J. Math. Chem., pages 267–277, 1988.

[8] H. Wiener. *Structural determination of paraffin boiling points.* J. Amer. Chem. Sot., 69:17–20, 1947.

[9] B. Zmazek and J. Žerovnik. *Computing the weighted Wiener and Szeged number on weighted cactus graphs in linear time.* Croatica Chemica Acta, 76:137–143, 2003.

Wiener Index, Diameter, and Stretch Factor of a Weighted Planar Digraph in Subquadratic Time

Christian Wulff-Nilsen [*]

Abstract

We consider the following three open problems: can the Wiener index (sum of all-pairs shortest path distances) and diameter of a planar digraph with arbitrary real edge weights and with no cycles of negative weight, and the stretch factor of a plane undirected geometric graph (maximum over all pairs of distinct vertices of the ratio between the graph distance and the Euclidean distance between the two vertices) be computed in subquadratic time? We present a theorem that allows us to solve all three problems by giving $O(n^2(\log\log n)^4/\log n)$ worst-case time algorithms for the Wiener index and diameter problems and an $O(n^2(\log\log n)^4/\log n)$ expected time algorithm for the stretch factor problem, where n is the size of the graph. Another corollary of the theorem is that an oracle for exact distance queries in a planar digraph with arbitrary real edge weights and no negative weight cycles can be constructed in subquadratic time. More generally, for a parameter $S = O(n^{1/5}/\log^{8/5} n)$, exact distance queries can be answered in $O(S\log^4 S/\log n)$ time per query with $O(n^2/S)$ preprocessing time, improving on previous results when $\log^4 S = o(\log n)$.

[*]Department of Computer Science, University of Copenhagen, `koolooz@diku.dk`, `http://www.diku.dk/~koolooz/`

1 Introduction

We consider the problems of computing the Wiener index (sum of all-pairs shortest path (APSP) distances) and diameter (maximum distance between any vertex pair) of a digraph with real edge weights and no negative weight cycles, and the problem of computing the stretch factor of an undirected geometric graph (maximum over all pairs of distinct vertices of the ratio between the graph distance and the Euclidean distance between the two vertices). For planar graphs, all three problems can be solved in quadratic time by applying Frederickson's APSP algorithm [6] but it is open whether subquadratic time algorithms exist (see [2], [3], and [1], respectively).

In this paper, we solve these three open problems. More precisely, we present algorithms for the Wiener index and diameter problems with $O(n^2(\log\log n)^4/\log n)$ worst-case running time and an algorithm for the stretch factor problem with $O(n^2(\log\log n)^4/\log n)$ expected running time, where n is the size of the graph. All three results are derived from a new theorem and it is our hope that this theorem can be applied to other planar graph problems.

From this main theorem, the following result also follows easily for an n-vertex planar digraph with arbitrary real edge weights and with no cycles of negative weight: for a parameter $S = O(n^{1/5}/\log^{8/5} n)$, distance queries can be answered in $O(S\log^4 S/\log n)$ time per query with $O(n^2/S)$ preprocessing time. Djidjev [4] gets $O(S)$ query time with $O(n^2/S)$ preprocessing time for $S = O(\sqrt{n})$. Our result is better for small S, i.e., when $\log^4 S = o(\log n)$. And in particular, we obtain an oracle for exact distance queries with $O(n^2(\log\log n)^4/\log n) = o(n^2)$ preprocessing time; the previous best result was to precompute all-pairs shortest path distances in quadratic time.

To obtain our results, we use ideas from [10] of obtaining a so-called r-division of G and performing heavy preprocessing in each region of the r-division to speed up computations in the main algorithm. However, [10] only applies to unweighted, undirected planar graphs so many new ideas are also needed.

The organization of the paper is as follows. In Section 2, we introduce some notation and give some basic definitions and results. In Section 3, we describe a generic algorithm which we will later apply to all the problems we consider. The description of the algorithm is split into two subsections. In Subsection 3.1, we describe the preprocessing step and in Subsection 3.2, we describe the main algorithm. We also bound the running time of the entire algorithm. The results are summed up in our main theorem, Theorem 2. In Section 4, we apply this theorem to the problems described above to obtain the desired time bounds. Finally, we make some concluding remarks in Section 5.

2 Definitions, Notation, and Basic Results

In this section, we introduce some notation and definitions and present some basic results.

For a digraph $G = (V, E)$, we define $V_G = V$ and $E_G = E$. Suppose G has real edge weights and no negative weight cycles. For $u, v \in V$, we let $d_G(u, v)$ denote the length of a shortest path in G from u to v. If no such path exists, we define $d_G(u, v) = \infty$.

In the rest of this section, let $G = (V, E)$ be a triangulated n-vertex planar digraph with a planar embedding. For a subset S of the plane, the *restriction* of G to S is defined as the intersection between G and S when regarding G as a point set.

A Jordan curve C partitions the plane into a bounded region, called the *interior* of C, and an unbounded region, called the *exterior* of C. We let $\mathit{Int}(C)$ resp. $\mathit{Ext}(C)$ denote the restriction of G to the interior resp. exterior of C, and we let $\overline{\mathit{Int}}(C)$ resp. $\overline{\mathit{Ext}}(C)$ denote the restriction of G to the closure of the interior resp. exterior of C. We omit G in these definitions but this should not cause any confusion.

Define a *region* R (of G) to be the subgraph of G induced by a subset of V. In G, the vertices of V_R that are adjacent to vertices in $V \setminus V_R$ are called *boundary vertices* (of R) and the set of boundary vertices of R is called the *boundary* of R which we denote by ∂R. Vertices of V_R that are not boundary vertices of R are called *interior vertices* (of R).

Let $r \in (0, n)$ be a parameter. Fakcharoenphol and Rao [5] showed how to recursively apply the cycle separator theorem of Miller [9] such that in $O(n \log n)$ time, (the plane embedding of) G is divided into $O(n/r)$ regions satisfying:

1. each region contains at most r vertices and $O(\sqrt{r})$ boundary vertices,
2. no two regions share interior vertices,
3. each region has a boundary contained in $O(1)$ faces, defined by simple cycles.

We refer to such a division as an *r-division* of G. The bounded faces of a region are its *holes*. We may assume that all vertices of faces containing the boundary of a region are boundary vertices of that region. Furthermore, we make the assumption that for each region R in an r-division, R is contained in $\overline{Int}(C)$ for one of the cycles C in the boundary of R. This can always be achieved by adding a new cycle if needed. Cycle C is the *external face* of R.

Consider an r-division of G consisting of regions $R_1, \ldots, R_p$. For each region R_i, let $\mathcal{C}_{R_i}$ denote the set of faces defining the boundary of R_i. Note that $|\mathcal{C}_{R_i}| = O(1)$.

Given a region R_i and a face $C \in \mathcal{C}_{R_i}$, define $U_{R_i,C}$ as the set of vertices of G belonging to $Int(C)$ resp. $Ext(C)$ if C is a hole resp. the external face of R_i.

Lemma 1. *With the above definitions, let $i \in \{1, \ldots, p\}$. For any $C \in \mathcal{C}_{R_i}$, any $u \in U_{R_i,C}$, and any $v \in V_{R_i}$, every shortest path in G from u to v contains a boundary vertex of R_i belonging to C. Furthermore, $\cup_{C \in \mathcal{C}_{R_i}} U_{R_i,C}$ is the set of vertices of V not belonging to R_i.*

3 A Generic Algorithm

In this section, $G = (V, E)$ denotes an n-vertex planar digraph with non-negative edge weights. We assume that G is triangulated (if not, triangulate it with oppositely directed infinite weight edges). We compute a planar embedding of G and identify G with this embedding. From this we obtain an r-division of G for some parameter r which we leave unspecified for now.

We will assume that single-source shortest path (SSSP) distances in G with each of the $O(n/\sqrt{r})$ boundary vertices as sources have been precomputed. Using the algorithm of [7], this takes $O(n\frac{n}{\sqrt{r}})$ time. For each boundary vertex p, we also precompute $d_G(u, p)$ for all $u \in V$. By reversing edges of G, we can again apply the algorithm of [7] to obtain these distances in $O(n\frac{n}{\sqrt{r}})$ time. We refer to all these distances as *boundary vertex distances* (in G).

Let us start by describing the problem that the generic algorithm should solve. In the following, let R be one of the regions in the r-division of G. Let C be one of the cycles in $\mathcal{C}_R$ and let $p_1, \ldots, p_t$ be the boundary vertices of R belonging to C. Assume that in a simple walk of C, $p_1, \ldots, p_t$ are visited in that order. Let $U = U_{R,C}$.

By Lemma 1, for any $u \in U$ and $v \in V_R$, $d_G(u, v) = d_G(u, p_i) + d_G(p_i, v)$ for some $p_i \in C$. We make the assumption that $d_G(u, v) = d_G(u, p_i) + d_R(p_i, v)$. This does not hold in general but can be satisfied as follows. For each cycle $C' \in \mathcal{C}_R \setminus \{C\}$, add to R an edge (u, v) for each pair of boundary vertices u and v of R belonging to C'. The weight of this edge is equal to the length of a shortest path in G from u to v among paths with all interior vertices belonging to $U_{R,C'}$. If no such path exists, the edge is omitted. Note that the complexity of the problem does not increase with the addition of these edges.

Now, the problem is to find, for each $u \in U$, a colouring of the vertices of R using t colours which we denote by indices $1, \ldots, t$. For $i = 1, \ldots, t$, a vertex $v \in R$ should be assigned colour i if $d_G(u, v) = d_G(u, p_i) + d_R(p_i, v)$. Ties may be resolved in any way. We refer to such a colouring as a *u-colouring* (of R).

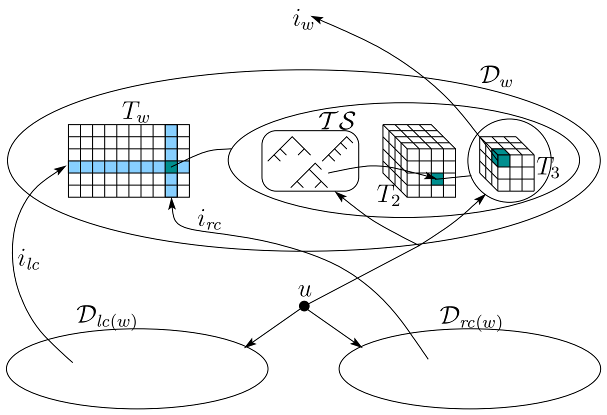


Figure 1: When queried with u, $\mathcal{D}_w$ returns an integer i_w representing an $\mathcal{I}_w$-colouring of R w.r.t. u. Here, $\mathcal{TS}$, T_2, and T_3 denote a tree set, a level 2-table, and a level 3-table, respectively.

3.1 Preprocessing Step

The main part of the generic algorithm is preceeded by a preprocessing step which we describe in this subsection. In the following, assume that R, C, U, and $p_1, \ldots, p_t$ are defined as above. We make the assumption that C is a hole. Hence, R is contained in $\overline{Ext}(C)$. The case where C is the external face is dealt with in a similar way.

The main algorithm should find a u-colouring of R w.r.t. each $u \in U$. The preprocessing step does not depend on u but to make it more clear how this step can speed up computations in the main algorithm, we assume in the following that we are given some unspecified $u \in U$.

Let $\mathcal{I}_t = \{1, \ldots, t\}$. For non-empty subset $\mathcal{I} \subseteq \mathcal{I}_t$, define an *$\mathcal{I}$-colouring of R* as a colouring of the vertices of R with colours in $\mathcal{I}$. We extend this definition to subgraphs of R and to subsets of vertices of R. When less specific, we call an $\mathcal{I}$-colouring of R an $|\mathcal{I}|$-colouring of R. When convenient, we will regard an $\mathcal{I}$-colouring of a set A of vertices as a map $c : A \rightarrow \mathcal{I}$.

An *$\mathcal{I}$-colouring of R w.r.t. u* is an $\mathcal{I}$-colouring of R where a vertex $v \in R$ is given colour i only if $d_G(u, p_i) + d_R(p_i, v) \leq d_G(u, p_j) + d_R(p_j, v)$ for all $j \in \mathcal{I}$, with ties resolved in any way. Given a subset A of vertices of R, an $\mathcal{I}$-colouring of A w.r.t. u is defined as an $\mathcal{I}$-colouring of A which can be extended to an $\mathcal{I}$-colouring of R w.r.t. u.

For an index set $\mathcal{I} = \{i_1, \ldots, i_m\}$ with $i_1 < i_2 < \ldots < i_m$, define the *lower subset* of $\mathcal{I}$ as the set $\{i_1, \ldots, i_{\lceil m/2 \rceil}\}$ and define the *upper subset* of $\mathcal{I}$ as the set $\{i_{\lceil m/2 \rceil + 1}, \ldots, i_m\}$.

The generic algorithm should find an $\mathcal{I}_t$-colouring of R w.r.t. u. The idea is to obtain this colouring recursively from an $\mathcal{I}_1$-colouring and an $\mathcal{I}_2$-colouring of R w.r.t. u, where $\mathcal{I}_1$ ($\mathcal{I}_2$) is the lower (upper) subset of $\mathcal{I}_t$.

In the preprocessing step of the algorithm, a balanced binary tree $\mathcal{T}$ is constructed that reflects this recursion. We refer to this tree as the *main tree*. Associated with each vertex w of $\mathcal{T}$ is a subset $\mathcal{I}_w$ of $\mathcal{I}_t$.

Main tree $\mathcal{T}$ and these subsets are defined as follows. The root r of $\mathcal{T}$ is associated with $\mathcal{I}_r = \mathcal{I}_t$. For each vertex w of $\mathcal{T}$, if $|\mathcal{I}_w| = 1$ then w is a leaf of $\mathcal{T}$. Otherwise, w has a left child which is associated with the lower subset of $\mathcal{I}_w$ and has a right child which is associated with the upper subset of $\mathcal{I}_w$. We let $\mathcal{T}_w$ denote the subtree of $\mathcal{T}$ rooted at w.

For each non-leaf vertex w of $\mathcal{T}$, we let $lc(w)$ resp. $rc(w)$ denote the left resp. right child of w. With each such w we associate a data structure called $\mathcal{D}_w$. When queried with u in the main algorithm, $\mathcal{D}_w$ returns an integer that identifies an $\mathcal{I}_w$-colouring of R w.r.t. u.

Data structure $\mathcal{D}_w$ is illustrated in Figure 1. It consists of a two-dimensional table T_w called the *level* 1*-table* (of w). Associated with each entry $T_w(i, j)$ of T_w is a data structure called a *tree set*. It represents a set of binary trees and there is a one-to-one map from this set into a higher-dimensional table, also associated with entry $T_w(i, j)$. We call this table a *level* 2*-table*.

With each entry of this table, a higher-dimensional *level* 3-*table* is associated.

Each level 3-table entry associated with $\mathcal{D}_w$ corresponds to an $\mathcal{I}_w$-colouring of R. When $\mathcal{D}_w$ is queried with u in the main algorithm, this vertex will be mapped to a level 3-table entry that corresponds to an $\mathcal{I}_w$-colouring of R w.r.t. u. A unique number is assigned to each level 3-table entry and if u is mapped to entry number x then x is the key value that uniquely identifies that $\mathcal{I}_w$-colouring of R w.r.t. u. It is this key value that $\mathcal{D}_w$ returns when queried with u.

In the preprocessing step, the actual colourings associated with level 3-entries will be needed. These colourings will be computed bottom-up in main tree $\mathcal{T}$. In the following, we consider a non-leaf vertex w of $\mathcal{T}$ and describe the components of data structure $\mathcal{D}_w$ associated with w. We assume that colourings of level 3-table entries associated with descendants of w in $\mathcal{T}$ have already been computed.

3.1.1 Level 1-Table

The index of each row of level 1-table T_w is an integer representing an $\mathcal{I}_{lc(w)}$-colouring of R w.r.t. some vertex and the index of each column is an integer representing an $\mathcal{I}_{rc(w)}$-colouring of R w.r.t. some vertex. There is a row resp. column for each level 3-table entry associated with $\mathcal{D}_{lc(w)}$ resp. $\mathcal{D}_{rc(w)}$ and each entry of T_w is associated with the two colourings corresponding to the entry's row and column, respectively.

In the main algorithm, data structures $\mathcal{D}_{lc(w)}$ and $\mathcal{D}_{rc(w)}$ are recursively queried with vertex u, giving integers i_{lc} and i_{rc} identifying, respectively, an $\mathcal{I}_{lc(w)}$-colouring and an $\mathcal{I}_{rc(w)}$-colouring of R w.r.t. u. From these, we obtain entry (i_{lc}, i_{rc}) in T_w, see Figure 1.

In case $lc(w)$ is a leaf of $\mathcal{T}$ then there is only one $\mathcal{I}_{lc(w)}$-colouring of R w.r.t. u so we define $i_{lc} = 1$ and T_w has only one row. And similarly, if $rc(w)$ is a leaf of $\mathcal{T}$ then $i_{rc} = 1$ and T_w has only one column.

In the following, we describe the tree set and the level 2-table and level 3-tables associated with the entry (i_{lc}, i_{rc}) of T_w that u is mapped to in the main algorithm.

3.1.2 Tree Set

The tree set data structure is essentially a compact representation of a certain set of binary trees and has a recursive definition. It is either a *leaf* or a *non-leaf*. If it is a leaf it represents one binary tree which itself is a leaf.

If it is a non-leaf it consists of a pair of arrays and each entry of these arrays represents a root of a set of binary trees. An entry corresponding to a root r points to two recursively defined tree sets representing, respectively, left and right subtrees of a binary tree with root r. Taking all combinations of left and right subtrees that those two tree sets represent gives all binary trees with root r.

Let $\mathcal{TS}$ be the tree set associated with the entry (i_{lc}, i_{rc}) of T_w that we consider and let $\mathcal{A}_i$ and $\mathcal{A}_j$ be the two arrays representing the roots of binary trees of $\mathcal{TS}$. The entries of $\mathcal{A}_i$ correspond to vertices of a shortest path P_i in R from boundary vertex p_i to a suitable vertex $v \in R$ where i is the colour of v in the $\mathcal{I}_{lc(w)}$-colouring of R w.r.t. u, see Figure 2. Similarly, the entries of $\mathcal{A}_j$ correspond to vertices of a shortest path P_j in R from boundary vertex p_j to v where j is the colour of v in the $\mathcal{I}_{rc(w)}$-colouring of R w.r.t. u. This is well-defined since the two colourings are independent of which u-vertex is mapped to entry (i_{lc}, i_{rc}) in T_w. Vertex v is chosen such that i resp. j is neither the first nor last index of $\mathcal{I}_{lc(w)}$ resp. $\mathcal{I}_{rc(w)}$.

For each vertex $v' \in P_i$, two shortest paths in R are associated: the subpath $P_i(v')$ of P_i from p_i to v' and a shortest path $P_i'(v')$ from $p_{j(v')}$ to v' where $j(v')$ is the colour of v' in the $\mathcal{I}_{rc(w)}$-colouring of R w.r.t. u. Path $P_i'(v')$ is chosen such that it does not cross $P_i(v')$, i.e., $P_i(v') \cup P_i'(v')$ contains no cycles when edges are considered to be undirected.

The union of $P_i(v')$ and $P_i'(v')$ partitions R into two smaller regions, R_1 and R_2, both containing $P_i(v')$ and $P_i'(v')$. Boundary vertices p_i and $p_{j(v')}$ partition C into two subpaths

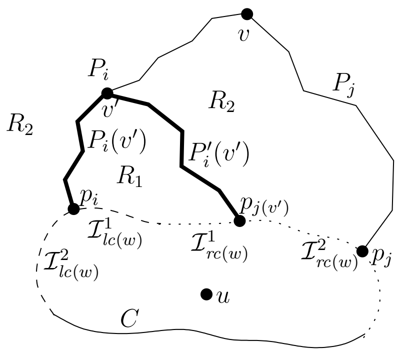


Figure 2: Associated with each vertex $v' \in P_i$ are two paths, $P_i(v')$ and $P'_i(v')$. For this example, the dashed resp. dotted curve illustrates the portion of C containing boundary vertices with indices in $\mathcal{I}_{lc(w)}$ resp. $\mathcal{I}_{rc(w)}$.

which induce a partition of $\mathcal{I}_{lc(w)}$ into two smaller index sets, $\mathcal{I}^1_{lc(w)}$ and $\mathcal{I}^2_{lc(w)}$, and induce a partition of $\mathcal{I}_{rc(w)}$ into index sets, $\mathcal{I}^1_{rc(w)}$ and $\mathcal{I}^2_{rc(w)}$, see Figure 2.

Two shortest paths are similarly associated with each vertex of P_j. We omit the definition since it is symmetric to the above.

The main algorithm will search in $\mathcal{A}_i$ (or in $\mathcal{A}_j$) to pick vertex v' in such a way that for $m = 1, 2$, each vertex of R_m can be assigned a colour from $\mathcal{I}^m_{lc(w)} \cup \mathcal{I}^m_{rc(w)}$ in an $\mathcal{I}_w$-colouring of R w.r.t. u. In other words, the problem of colouring vertices of R is divided into two subproblems.

In the preprocessing step, we need to consider all possible choices of v'. For each such choice, we recurse on the two subproblems to obtain the two tree sets that the v'-entry of $\mathcal{A}_i$ (or $\mathcal{A}_j$) points to. It may be necessary to redefine the given $\mathcal{I}_{lc(w)}$- and $\mathcal{I}_{rc(w)}$-colourings obtained from $\mathcal{D}_{lc(w)}$ and $\mathcal{D}_{rc(w)}$, respectively, to ensure that the colour pair in $\mathcal{I}_{lc(w)} \times I_{rc(w)}$ for each vertex of R_m belongs to $\mathcal{I}^m_{lc(w)} \times \mathcal{I}^m_{rc(w)}$, $m = 1, 2$. We omit the details but it can be shown that these redefinitions depend only on the choice of indices i and $j(v')$. Hence, these computations can be performed in the preprocessing step instead of the main algorithm.

The recursion stops when there is no choice of v satisfying the index requirement above. Each leaf of $\mathcal{TS}$ then has an associated subregion of R and two colourings of this subregion with colours from subsets of $\mathcal{I}_{lc(w)}$ and $\mathcal{I}_{rc(w)}$, respectively.

The main algorithm will search in $\mathcal{A}_i$ if $d_G(u, p_i) + d_R(p_i, v) \geq d_G(u, p_j) + d_R(p_j, v)$ and in $\mathcal{A}_j$ otherwise. If the search is in $\mathcal{A}_i$ then v' is picked such that it is the first vertex of P_i such that $d_G(u, p_i) + d_R(p_i, v') \geq d_G(u, p_{j(v')}) + d_R(p_{j(v')}, v')$ (if the search is in $\mathcal{A}_j$, v' is picked similarly in P_j). Binary search is applied to find this vertex. It can be shown that with this choice of v', we get the above division of our colouring problem into two subproblems.

Observe that with each choice of u, a certain binary tree of $\mathcal{TS}$ is traversed in the main algorithm. We refer to this traversal as a *u-traversal* of $\mathcal{TS}$. Lemma 3 below shows that $\mathcal{I}_w$-colourings of R w.r.t. u-vertices traversing the same binary tree are in some sense related. The following lemma will also be needed.

Lemma 2. *Each binary tree represented by $\mathcal{TS}$ has $O(|\mathcal{I}_w|)$ vertices.*

Proof. (Sketch) It is easy to see that $|\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}| + |\mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}| = |\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}| + 2$. And since v was chosen such that i resp. j is neither the first nor last index of $\mathcal{I}_{lc(w)}$ resp. $\mathcal{I}_{rc(w)}$, we have $|\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}|, |\mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}| < |\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}|$. This implies that the number of leaves in a tree represented by $\mathcal{TS}$ is at most $l(|\mathcal{I}_w|)$, where $l : \mathbb{N} \to \mathbb{N}$ is defined by

$$l(k) = \begin{cases} \max\{l(k_1) + l(k_2) | k_1 + k_2 = k + 2, k_1, k_2 < k\} & \text{if } k > 4 \\ 1 & \text{if } k \leq 4, \end{cases}$$

It can be shown by induction on $k \geq 3$ that $l(k) \leq k - 2$ which implies the lemma. ☐

Lemma 3. *Let U' be a subset of vertices of U all traversing the same binary tree in $\mathcal{TS}$. Then there are $O(|\mathcal{I}_w|)$ sets $V_1, \ldots, V_m$ of vertices whose union is V_R such that for $k = 1, \ldots, m$ and for each $u' \in U'$ there is a 2-colouring of V_k which is an $\mathcal{I}_w$-colouring of V_k w.r.t. u'.*

Proof. (Sketch) Let $u \in U'$ and let $A_1, \ldots, A_p$ be the sets of vertices of regions associated with leaves of the binary tree traversed by u. From the construction of $\mathcal{TS}$ and the definition of u-traversal, it follows that $\cup_{k=1}^{p} A_k = V_R$.

Let $k \in \{1, \ldots, p\}$ be given and let l_k denote the leaf with which the region with vertex set A_k is associated. Let $c : A_k \to \mathcal{I}$ and $c' : A_k \to I'$ be the colourings of A_k associated with l_k, where $\mathcal{I} \subseteq \mathcal{I}_{lc(w)}$ and $\mathcal{I}' \subseteq \mathcal{I}_{rc(w)}$. From the way the v'-vertices are chosen by the main algorithm, it follows that for each $x \in A_k$, either $c(x)$ or $c'(x)$ is a colour of x in an $\mathcal{I}_w$-colouring of R w.r.t. u. It can be shown that the total number of distinct colour pairs for vertices in regions associated with leaves of a tree T of $\mathcal{TS}$ is $O(|\mathcal{I}_w|)$. The lemma then follows by letting $V_1, \ldots, V_m$ be maximal sets each containing vertices with identical colour pairs. $\square$

3.1.3 Level 2-Table

We now describe the level 2-tables associated with $\mathcal{D}_w$. Recall that there is a tree set associated with each entry of the level 1-table T_w. For each such tree set $\mathcal{TS}$ there is a one-to-one map ϕ from the set of binary trees that $\mathcal{TS}$ represents to entries of a level 2-table. In the following, we define this map and table.

Let $c \in \mathbb{N}$ be a constant such that each tree represented by $\mathcal{TS}$ has at most $c|\mathcal{I}_w|$ non-leaf vertices. Such a constant exists by Lemma 2. Let $n_R = O(r)$ be the number of vertices of R. Arbitrarily assign a unique number in $\{0, \ldots, n_R - 1\}$ to each of these vertices.

Consider a tree T represented by $\mathcal{TS}$. Its root corresponds to an entry in one of the two arrays at the top-level of $\mathcal{TS}$. This entry is uniquely defined by the choice of array and the choice of v'-entry. The array is uniquely determined by the value of a single bit and v is uniquely determined by its number in the above assignment. It follows that the root of T is uniquely defined by a pair in $\{0, 1\} \times \{0, \ldots, n_R - 1\}$ which we may regard as a pair in $\{0, \ldots, n_R - 1\}^2$.

Applying the above recursively to the subtrees of T, it follows that T is uniquely defined by a vector in $\{0, \ldots, n_R - 1\}^{2c|\mathcal{I}_w|}$. We define $\phi(T)$ to be this vector and with this definition, ϕ is one-to-one. The level 2-table T_2 associated with $\mathcal{TS}$ represents the set $\{0, \ldots, n_R - 1\}^{2c|\mathcal{I}_w|}$.

3.1.4 Level 3-Table

Let U' be a set of vertices of U that traverse the same tree in $\mathcal{TS}$. Then $\phi(U') = \{\overline{v}\}$ for some vector $\overline{v}$ corresponding to an entry in T_2.

Let $V_1, \ldots, V_m$ be defined as in Lemma 3. Then the same lemma implies that for each $u' \in U'$ it is enough to specify 2-colourings of $V_1, \ldots, V_m$ to specify an $\mathcal{I}_w$-colouring of R w.r.t. u'. The following lemma shows that each of these 2-colourings can be specified by an integer in $\{0, \ldots, n_R\}$ and that this integer can be computed efficiently.

Lemma 4. *Let i and j be distinct indices in $\mathcal{I}_w$. With the above definitions, there is a map $f : U' \to \{0, \ldots, n_R\}$ and $\{i, j\}$-colourings $c_0, \ldots, c_{n_R}$ of R such that for each $u \in U'$, $c_{f(u)}$ is an $\{i, j\}$-colouring of R w.r.t. u. Assuming $d_G(u, p_i)$ and $d_G(u, p_j)$ are given, $f(u)$ can be computed in $O(\log r)$ time for any $u \in U'$ with preprocessing time polynomial in r.*

The $O(\log r)$ time bound is obtained by using binary search in the $O(r)$ possible $\{i, j\}$-colourings of R. We omit the details.

We associate with T_2-entry $\overline{v}$ a level 3-table T_3. This table represents the set $\{0, \ldots, n_R\}^m$, where m is the number of sets in Lemma 3. This lemma and Lemma 4 show how u-vertices can be mapped in the main algorithm to entries in T_3 such that if two vertices of U are mapped to the same entry then they induce identical colourings of R. In the preprocessing step, the proper colourings are computed for each level 3-table entry. It is easy to see that each such colouring can be computed in time polynomial in r.

We enumerate all entries of level 3-tables associated with vertex w of main tree $\mathcal{T}$ with integers $1, \ldots, m_w$, where m_w is the total number of such entries. In the main algorithm, when a query vertex $u \in U$ is mapped to level 3-table entry with integer k then k is the integer returned. By the above, this value defines an $\mathcal{I}_w$-colouring of R w.r.t. u.

It can be shown that the number of level 3-table entries associated with main tree $\mathcal{T}$ is $O(r^{O(\sqrt{r}\log r)})$. Since the colouring associated with each entry can be computed in time polynomial in r, we get the following result.

Theorem 1. *The number of level* 3*-table entries associated with main tree $\mathcal{T}$ and the total time spent in the preprocessing step are $O(r^{O(\sqrt{r}\log r)})$, assuming boundary vertex distances are given.*

3.2 Main Algorithm

We are now ready to describe the main algorithm. It is applied to each $u \in U$. Let u be one such vertex. We start at the root r of $\mathcal{T}$ and recursively find two integers, i_{lc} and i_{rc}, representing, respectively, an $\mathcal{I}_{lc(r)}$-colouring and an $\mathcal{I}_{rc(r)}$-colouring of R w.r.t. u. Using these integers as indices, we obtain the tree set $\mathcal{TS}$ associated with entry (i_{lc}, i_{rc}) in level 1-table T_r.

We perform a u-traversal in $\mathcal{TS}$ which gives us a binary tree and we map this tree to a vector defining an entry in the level 2-table associated with $\mathcal{TS}$. Let T_3 be the level 3-table associated with this entry.

Finally, we use the algorithm implicit in Lemma 4 to obtain the correct entry of T_3. The integer associated with this entry is then returned as that representing a u-colouring of R.

Let us bound the time it takes to map a u-vertex to a level 3-table entry. We have precomputed the main tree $\mathcal{T}$ with associated data structures and boundary vertex distances in G. At a vertex w of $\mathcal{T}$, it takes $O(|\mathcal{I}_w|\log r)$ time to perform a u-traversal. This follows easily from Lemma 2 and the fact that binary search is applied to arrays of tree sets. Lemma 4 then implies that the total time spent at vertex w is $O(|\mathcal{I}_w|\log r)$ (ignoring preprocessing which is polynomial in r). Summing over all w, this is $O(t\log^2 r)$, which gives the following result.

Lemma 5. *Given main tree $\mathcal{T}$ with associated data structures and given boundary vertex distances in G, the main algorithm obtains the integer representing a u-colouring of R in time $O(t\log^2 r)$ for any $u \in U$ with preprocessing time polynomial in r.*

We can now present our main theorem in which we fix r, state our generic algorithm, and bound its running time.

Theorem 2. *With the above definitions, suppose $r = (c\log n/(\log\log n)^2)^2$ where c is a constant. If c is sufficiently small then there is a constant $\epsilon < 1$, an integer $N = O(n^\epsilon)$, a map $f : U \to \{1, \ldots, N\}$, and $\mathcal{I}_t$-colourings $c_1, \ldots, c_N$ of R where $c_{f(u)}$ is a u-colouring of R for all $u \in U$. The integers $f(u)$ for $u \in U$ and colourings $c_1, \ldots, c_N$ can be computed in a total of $O(n^\epsilon + |U|\log n)$ time with $O(n^2(\log\log n)^2/\log n)$ preprocessing time (independent of U and R).*

Proof. (Sketch) We start by precomputing boundary vertex distances in G in a total of $O(n^2/\sqrt{r})$ time.

In the main algorithm, vertices of U are mapped to integers in the range $\{1, \ldots, N\}$, where N is the total number of level 3-table entries. By Theorem 1 and Lemma 5 this takes a total of $O(r^{c'\sqrt{r}\log r} + |U|\sqrt{r}\log^2 r)$ time for some constant c'. So to show one part of the theorem, that integers $f(u)$ for $u \in U$ can be computed in a total of $O(n^\epsilon + |U|\log n)$ time for some constant $\epsilon < 1$, we will pick constant c such that $\sqrt{r}\log^2 r = O(\log n)$, $r^{c'\sqrt{r}\log r} = O(n^\epsilon)$, and $n^2/\sqrt{r} = O(n^2(\log\log n)^2/\log n)$. This will also show $N = O(n^\epsilon)$.

With the choice of r, it can be shown that for sufficiently large n, $r^{c'\sqrt{r}\log r} \leq n^{16c'c}$. By setting $c < 1/(16c')$, we have $r^{c'\sqrt{r}\log r} = O(n^\epsilon)$ with $\epsilon < 1$. Also, $\sqrt{r}\log^2 r = O(\log n)$ and $n^2/\sqrt{r} = O(n^2(\log\log n)^2/\log n)$, as requested.

To bound the time to compute the colourings, we recall that each colouring can be obtained in time polynomial in r. Thus, the total time to compute all colourings is $O(Nr^{O(1)}) = O(r^{O(\sqrt{r}\log r)})$. This is $O(n^\epsilon)$ for c sufficiently small. □

4 Applications of the Generic Algorithm

In this section, we apply the generic algorithm to obtain subquadratic time algorithms for computing the Wiener index and diameter of a planar digraph with real edge weights, and the stretch factor of a plane undirected geometric graph. We also present our algorithm for distance queries.

We start with the Wiener index problem for a planar digraph $G = (V, E)$ with real edge weights. For $V_1, V_2 \subseteq V$, define $\sum(V_1, V_2) = \sum_{v_1 \in V_1}\sum_{v_2 \in V_2} d_G(v_1, v_2)$. We extend this definition to subgraphs of G by summing over their vertex sets. Then $\frac{1}{2}\sum(V, V)$ is the *Wiener index* of G.

Theorem 3. *The Wiener index of an n-vertex planar digraph with real edge weights and no negative weight cycles can be computed in $O(n^2(\log\log n)^4/\log n)$ time.*

Proof. Let G be an n-vertex planar digraph. We only consider the case where G has non-negative edge weights. Arbitrary real edge weights are dealt with in the appendix. We start by triangulating G and computing an r-division $\mathcal{R}$ of G in $O(n\log n)$ time with $r = (c\log n/(\log\log n)^2)^2$ for constant c satisfying Theorem 2.

By Lemma 1,

$$\sum G = \frac{1}{2}\sum_{R \in \mathcal{R}}\left(\sum(R, R) + \sum_{C \in \mathcal{C}_R}\sum(U_{R,C}, R)\right).$$

Let $R \in \mathcal{R}$ be given. We will show how to compute $\sum(R, R)$ and $\sum_{C \in \mathcal{C}_R}\sum(U_{R,C}, R)$ in $O(|\mathcal{C}_R|n^\epsilon + n\log n)$ time for some constant $\epsilon < 1$. Since each cycle occurs in at most two regions and since there are $O(n/r)$ regions it will follow from this that $\sum G$ can be computed in time

$$O\left(\frac{n}{r}n^\epsilon + \frac{n}{r}n\log n\right) = O\left(\frac{n^{1+\epsilon}}{r} + \frac{n^2\log n}{(\log n/(\log\log n)^2)^2}\right) = O(n^2(\log\log n)^4/\log n).$$

To compute $\sum(R, R)$, let R' be the graph obtained from R by adding an edge between each pair of boundary vertices of R. The weight of each edge is equal to the distance in G from the start to the end vertex of the edge. We compute APSP distances in R' and obtain $\sum(R, R)$ by adding up all these distances. The time it takes to add edges and compute their weights is $O(r)$ time, given the precomputed boundary vertex distances. It then takes $O(r^3)$ time to compute APSP distances by using an algorithm like Floyd-Warshall. Thus, $\sum(R, R)$ can be computed in time polylogarithmic in n.

What remains is to show that $\sum_{C \in \mathcal{C}_R}\sum(U_{R,C}, R)$ can be computed in $O(|\mathcal{C}_R|n^\epsilon + n\log n)$ time for some constant $\epsilon < 1$ with $O(n^2(\log\log n)^2/\log n)$ preprocessing time. So let $C \in \mathcal{C}_R$ be given and let $p_1, \dots, p_t$ be the boundary vertices of R belonging to C.

By Theorem 2, there is a constant $\epsilon' < 1$, an integer $N = O(n^{\epsilon'})$, a map $f : U_{R,C} \to \{1, \dots, N\}$, and $\mathcal{I}_t$-colourings $c_1, \dots, c_N$ of R such that $c_{f(u)}$ is a u-colouring of R for all $u \in U_{R,C}$. The integers $f(u)$ for $u \in U_{R,C}$ and colourings $c_1, \dots, c_N$ can be computed in a total of $O(n^{\epsilon'} + |U_{R,C}|\log n)$ time with $O(n^2(\log\log n)^2/\log n)$ preprocessing time.

Let $M \in \{1, \dots, N\}$. For the colouring c_M of R corresponding to M, we compute $\sum(\{p_i\}, V_{i,M})$ for $i = 1, \dots, t$, where $V_{i,M}$ is the set of vertices of R with colour i. We also compute the number $|V_{i,M}|$ of vertices in $V_{i,M}$. This can clearly be done in time polynomial in r which is poly-logarithmic in n. Over all i and M, this is $\tilde{O}(n^{\epsilon'}) = O(n^{\epsilon})$ time for some constant $\epsilon < 1$.

Now, for a $u \in U_{R,C}$, let $M_u = f(u)$. Then

$$\sum(\{u\}, R) = \sum_{i=1}^{t} d_G(u, p_i)|V_{i,M_u}| + \sum(\{p_i\}, V_{i,M_u}). \tag{1}$$

Given the above precomputations, it follows that $\sum(\{u\}, R)$ can be computed in $O(t)$ time. Hence, $\sum(U_{R,C}, R)$ can be computed in $O(n^{\epsilon} + |U_{R,C}|t) = O(n^{\epsilon} + |U_{R,C}|\sqrt{r})$ time with $O(n^2(\log\log n)^2/\log n)$ preprocessing time.

Adding this up over all $C \in \mathcal{C}_R$ and using the fact that $\sum_{C\in\mathcal{C}_R} |U_{R,C}| \leq n$, it follows that $\sum_{C\in\mathcal{C}_R} \sum(U_{R,C}, R)$ can be computed in

$$O(|\mathcal{C}_R|n^{\epsilon} + n\sqrt{r}) = O(|\mathcal{C}_R|n^{\epsilon} + n\log n/(\log\log n)^2)$$

time in addition to the $O(|\mathcal{C}_R|n^{\epsilon'} + n\log n)$ time spent in Theorem 2 and $O(n^2(\log\log n)^2/\log n)$ preprocessing time. □

Next, the diameter of a planar graph.

Theorem 4. *The diameter of an n-vertex planar digraph with real edge weights and no negative weight cycles can be computed in $O(n^2(\log\log n)^4/\log n)$ time.*

Proof. The proof is similar to that of Theorem 3. The only essential difference is that we compute $\max_{v\in V_{i,M_u}} d_G(p_i, v)$ instead of $\sum(\{p_i\}, V_{i,M_u})$ and use the identity

$$\max_{v\in V_R} d_G(u, v) = \max\left\{d_G(u, p_i) + \max_{v\in V_{i,M_u}} d_G(p_i, v) | i = 1, \ldots, t\right\}$$

instead of (1). □

Let $G = (V, E)$ be a plane undirected geometric graph. For $V_1, V_2 \subseteq V$, let $\delta_G(V_1, V_2) = \max\{d_G(v_1, v_2)/|v_1v_2|_2 | v_1 \in V_1, v_2 \in V_2, v_1 \neq v_2\}$. Extend this definition to subgraphs of G by taking the maximum over their vertex sets. the *stretch factor* of G is $\delta_G(V, V)$.

Theorem 5. *The stretch factor of an n-vertex plane undirected geometric graph can be computed in $O(n^2(\log\log n)^4/\log n)$ expected time.*

Proof. Let G be an n-vertex plane undirected geometric graph. We first compute an r-division of a triangulation of G (the triangulation is not of the embedding of G) with $r = (c\log n/(\log\log n)^2)^2$ for constant c satisfying Theorem 3 and compute boundary vertex distances.

Let R be a region in this r-division. We will show how to compute $\delta_G(U_{R,C}, R)$ for all $C \in \mathcal{C}_R$ using a total of $O(|\mathcal{C}_R|n^{\epsilon} + n\log n)$ expected time for some constant $\epsilon < 1$ with $O(n^2(\log\log n)^2/\log n)$ preprocessing time. It will follow from this that the stretch factor of G can be computed in $O(n^2(\log\log n)^4/\log n)$ expected time.

Let $C \in \mathcal{C}_R$ and let $p_1, \ldots, p_t$ be the boundary vertices of R belonging to C. By Theorem 2, there is a constant $\epsilon' < 1$, an integer $N = O(n^{\epsilon'})$, a map $f : U_{R,C} \to \{1, \ldots, N\}$, and $\mathcal{I}_t$-colourings $c_1, \ldots, c_N$ of R such that $c_{f(u)}$ is a u-colouring of R for all $u \in U_{R,C}$. The integers $f(u)$ for $u \in U_{R,C}$ and colourings $c_1, \ldots, c_N$ can be computed in a total of $O(n^{\epsilon'} + |U_{R,C}|\log n)$ time with $O(n^2(\log\log n)^2/\log n)$ preprocessing time.

For $M = 1, \ldots, N$, define the *group* U_M as the set of $u \in U_{R,C}$ such that $f(u) = M$. All vertices in the same group induce identical colourings of R.

Consider one such group U_M and let $i \in \{1, \ldots, t\}$ be given. Lift each vertex $v \in V_{i,M}$ to height $d_G(p_i, v)$ on the z-axis, where $V_{i,M}$ is the set of vertices of R with colour i in the colouring

c_M associated with group U_M. Furthermore, lower each vertex u of U_M to height $-d_G(u, p_i)$. Each lifting/lowering of a vertex takes constant time given the precomputed boundary vertex distances. The sets $V_{i,M}$ over all i and M can be computed in $\tilde{O}(n^{\epsilon'}) = O(n^{\epsilon})$ time for some $\epsilon > 0$.

Observe that the height difference between any lowered vertex u and any lifted vertex v is equal to $d_G(u, v)$. Now, arbitrarily divide U_M into $O(|U_M|/\sqrt{r})$ subsets each containing $O(\sqrt{r})$ vertices. Applying the algorithm of [1] to the (lowered) vertices in each of these subsets and to the (lifted) vertices in $V_{i,M}$ gives $\delta_G(U_M, V_{i,M})$ in expected time

$$O\left(\frac{|U_M|}{\sqrt{r}}(\sqrt{r} + |V_{i,M}|)\log(\sqrt{r} + |V_{i,M}|)\right) = O\left(|U_M|\left(1 + \frac{|V_{i,M}|}{\sqrt{r}}\right)\log r\right).$$

Summing over all i, we see that $\delta_G(U_M, R)$ can be found in expected time

$$O(|U_M|(t + r/\sqrt{r})\log r) = O(|U_M|\sqrt{r}\log r).$$

Hence, $\delta_G(U_{R,C}, R)$ can be found in $O(|U_{R,C}|\sqrt{r}\log r)$ expected time. Over all $C \in \mathcal{C}_R$, this is $O(n\sqrt{r}\log r) = O(n\log n/\log\log n)$. This shows the theorem. $\square$

We strongly believe that by using parametric search as in [1], it is possible to compute the stretch factor of G in $O(n^2(\log\log n)^{O(1)}/\log n)$ *worst-case* time.

Finally, we present our algorithm to answer distance queries.

Theorem 6. *Let $S = O(n^{1/5}/\log^{8/5} n)$ be a parameter. With $O(n^2/S)$ preprocessing time, exact distance queries in an n-vertex planar digraph with arbitrary real edge weights and no negative weight cycles can be answered in $O(S\log^4 S/\log n)$ time per query.*

The proof can be obtained from a slightly more general version of Theorem 2. Details can be found in the appendix. For $\log^4 S = o(\log n)$, Theorem 6 is an improvement over the result in [4]. By setting $S = \log^2 n/(\log\log n)^4$, we obtain an oracle for exact distance queries with subquadratic preprocessing time.

Corollary 1. *With $O(n^2(\log\log n)^4/\log n)$ preprocessing time, exact distance queries in an n-vertex planar digraph with arbitrary real edge weights and no negative weight cycles can be answered in constant time per query.*

5 Concluding Remarks

We showed how to compute the Wiener index and diameter of an n-vertex planar digraph with real edge edge weights and no negative weight cycles in $O(n^2(\log\log n)^4/\log n)$ worst-case time and the stretch factor of an n-vertex plane undirected geometric graph in $O(n^2(\log\log n)^4/\log n)$ expected time. Previously, it was open whether any of these three problems could be solved in subquadratic time.

We also gave an algorithm that answers distance queries in an n-vertex planar digraph with real edge weights and no negative cycles in $O(S\log^4 S/\log n)$ time per query with $O(n^2/S)$ preprocessing time for $S = O(n^{1/5}/\log^{8/5} n)$, improving on previous results when $\log^4 S = o(\log n)$. In particular, we obtained an oracle for exact distance queries with subquadratic preprocessing time.

All our results are obtained by applying the same generic algorithm. We hope that this algorithm may yield faster algorithms for other planar graph problems.

We pose the following questions: can we obtain "truly" subquadratic running time, i.e. $O(n^c)$ time for some constant $c < 2$? Can our results be generalized to a larger class of graphs such as the class of subgraph-closed $\sqrt{n}$-separable digraphs with real edge weights? All our algorithms have space requirement only bounded by running time. Is it possible to obtain, say, linear space requirement for the Wiener index and diameter problems as for unweighted undirected planar graphs [10]? What about lower bounds on running time?

References

[1] P. K. Agarwal, R. Klein, C. Knauer, S. Langerman, P. Morin, M. Sharir, and M. Soss. Computing the Detour and Spanning Ratio of Paths, Trees and Cycles in 2D and 3D. Discrete and Computational Geometry, 39 (1): 17–37 (2008).

[2] S. Cabello and C. Knauer. Algorithms for graphs of bounded treewidth via orthogonal range searching. Manuscript, Berlin, 2007.

[3] F. R. K. Chung. Diameters of Graphs: Old Problems and New Results. Congressus Numerantium, 60:295–317, 1987.

[4] H. Djidjev. Efficient algorithms for shortest path problems on planar digraphs. In WG'96, volume 1197 of LNCS, pages 151–165, Springer, 1997.

[5] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. Available from the authors' webpages. Preliminary version in FOCS'01.

[6] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. SIAM J. Comput., 16 (1987), pp. 1004–1022.

[7] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster Shortest-Path Algorithms for Planar Graphs. Journal of Computer and System Sciences volume 55, issue 1, August 1997, pages 3–23.

[8] P. N. Klein, S. Mozes, and O. Weimann. Shortest Paths in Directed Planar Graphs with Negative Lengths: a Linear-Space $O(n \log^2 n)$-Time Algorithm. Proc. 19th Ann. ACM-SIAM Symp. Discrete Algorithms, p. 236–245, 2009.

[9] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. J. Comput. Syst. Sci., 32:265–279, 1986.

[10] C. Wulff-Nilsen. Wiener Index and Diameter of a Planar Graph in Subquadratic Time. Proc. 25th European Workshop on Computational Geometry, Brussels, 2009, p. 25–28.

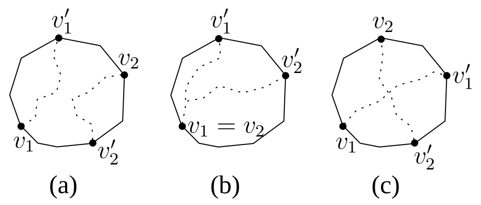


Figure 3: In (a) and (b), paths represented by abstract edges $e_1 = (v_1, v_1')$ and $e_2 = (v_2, v_2')$ can be chosen such that they do not cross. In (c) they have to cross.

Appendix

Abstract Edges

Recall that we added edges to R in the beginning of Section 3 to ensure that for any $u \in U$ and any vertex $v \in R$, $d_G(u,v) = d_G(u,p_i) + d_R(p_i,v)$ for some $p_i \in C$. We shall refer to these additional edges as *abstract edges* since we do not specify any embedding of them. Using the algorithm of [7], we can compute the length of all abstract edges within the desired time bound.

When convenient, we regard an abstract edge between two boundary vertices of R belonging to a cycle C' as a shortest path between the vertices having all its interior vertices in $U_{R,C'}$. By definition, the weight of the abstract edge is equal to the length of this path. The following lemma will prove useful as it allows us to obtain information about such paths without explicitly knowing them (see Figure 3).

Lemma 6. *Let $C' \in \mathcal{C}_R \setminus \{C\}$ and let $e_1 = (v_1, v_1')$ and $e_2 = (v_2, v_2')$ be abstract edges, where $v_1, v_1', v_2, v_2' \in C'$. Then the paths represented by e_1 and e_2 may be chosen such that they do not cross if and only if there is a cyclic walk of C' that visits v_1, v_1', v_2, and v_2' in one of the following two orders:*

1. *v_1, v_1', v_2, v_2' (where possibly $v_1' = v_2$ or $v_1 = v_2'$),*
2. *v_1, v_1', v_2', v_2 (where possibly $v_1' = v_2'$ or $v_1 = v_2$).*

Proof. This is an easy consequence of planarity and the fact that the interior vertices of the shortest paths represented by e_1 and e_2 are all contained in $\overline{Int}(C')$ or all contained in $\overline{Ext}(C')$. $\square$

Subdividing the colour problem

In Section 3.1.2, we claimed that the choice of v' gave the desired division of our colouring problem into two subproblems. We now fill in the details.

Recall the requirement that v' had to satisfy in order to be chosen. Clearly, this vertex always exists. For symmetry reasons, let us restrict our attention to the case where it is found by a search in $\mathcal{A}_i$. Then the following conditions must be satisfied:

1. $d_G(u,p_i) + d_R(p_i,v') \geq d_G(u,p_{j(v')}) + d_R(p_{j(v')},v')$ and
2. $d_G(u,p_i) + d_R(p_i,v'') < d_G(u,p_k) + d_R(p_k,v'')$ for any vertex $v'' \in V_{P_i(v')} \setminus \{v'\}$ and any $k \in \mathcal{I}_{rc(w)}$.

That v' gives the desired division of our colouring problem is shown in the following lemma.

Lemma 7. *Suppose v' satisfies the above. Then for $m = 1, 2$, there is an $\mathcal{I}_w$-colouring of R_m w.r.t. u where each vertex of R_m is assigned a colour from $\mathcal{I}^m_{lc(w)} \cup \mathcal{I}^m_{rc(w)}$.*

M

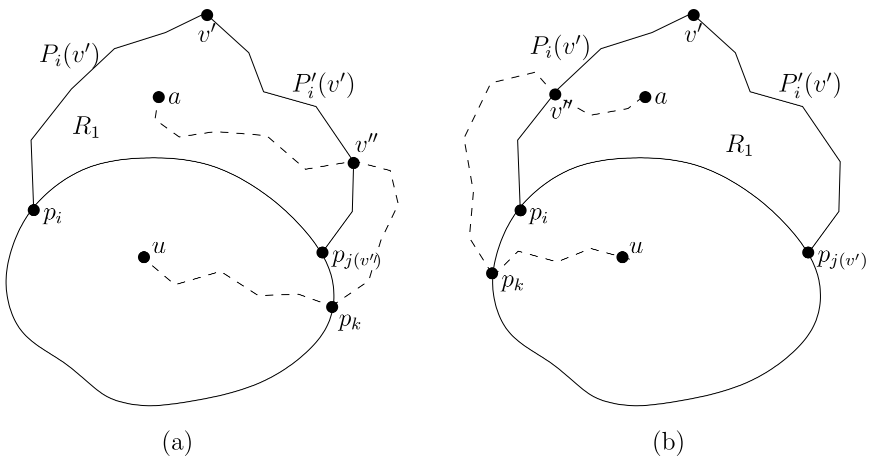


Figure 4: A shortest path in R from p_k to a intersects either $P'_i(v')$ or $P_i(v') \setminus \{v'\}$ in some vertex v''.

Proof. By symmetry, it suffices to show the lemma for $m = 1$. So let a be a vertex in R_1. We will show that in an $\mathcal{I}_w$-colouring of R_1 w.r.t. u, a can be assigned a colour in $\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}$.

Pick $k \in \mathcal{I}_w$ such that $d_G(u, p_k) + d_R(p_k, a) \leq d_G(u, p_{k'}) + d_R(p_{k'}, a)$ for all $k' \in \mathcal{I}_w$. Suppose that $k \notin \mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}$ (otherwise, the lemma holds). Then a shortest path in R from p_k to a must contain some vertex $v'' \in P_i(v') \cup P'_i(v')$. Assume first that $v'' \in P'_i(v')$, see Figure 4(a). Since the search for v' was in $\mathcal{A}_i$, we have $d_G(u, p_i) + d_R(p_i, v') \geq d_G(u, p_{j(v')}) + d_R(p_{j(v')}, v')$, implying that any vertex on $P'_i(v')$, and in particular v'', can be assigned colour $j(v')$ in an $\mathcal{I}_w$-colouring of R_1 w.r.t. u. This implies that a can be assigned colour $j(v') \in \mathcal{I}^1_{rc(w)}$ in such a colouring, as requested.

Now, suppose $v'' \in P_i(v') \setminus \{v'\}$, see Figure 4(b). Either $k \in \mathcal{I}_{lc(w)}$ or $k \in \mathcal{I}_{rc(w)}$. If $k \in \mathcal{I}_{lc(w)}$ then we may assign colour $i \in \mathcal{I}^1_{lc(w)}$ to a in an $\mathcal{I}_w$-colouring so assume that $k \in \mathcal{I}_{rc(w)}$. Then $d_G(u, p_i) + d_R(p_i, v'') < d_G(u, p_k) + d_R(p_k, v'')$, again since the search was in $\mathcal{A}_i$. By the choice of k and the triangle inequality,

$$\begin{aligned} d_G(u, p_k) + d_R(p_k, v'') + d_R(v'', a) &= d_G(u, p_k) + d_R(p_k, a) \\ &\leq d_G(u, p_i) + d_R(p_i, a) \\ &\leq d_G(u, p_i) + d_R(p_i, v'') + d_R(v'', a), \end{aligned}$$

implying that $d_G(u, p_k) + d_R(p_k, v'') \leq d_G(u, p_i) + d_R(p_i, v'')$. But this contradicts the inequality above. Hence, the lemma holds in all cases. □

By applying Lemma 7 recursively at each node of $\mathcal{TS}$, we can obtain the desired decomposition of the colouring problem. But for this to work, we also need to address the following problem which we briefly mentioned in the main paper. In the preprocessing step, let c_{lc} and c_{rc} be the $\mathcal{I}_{lc(w)}$- and $\mathcal{I}_{rc(w)}$-colourings that are obtained recursively from data structures $\mathcal{D}_{lc(w)}$ and $\mathcal{D}_{rc(w)}$, respectively. Since an $\mathcal{I}_w$-colouring of R w.r.t. u need not be unique, it may happen that the desired $\mathcal{I}_w$-colouring in Lemma 7 cannot be obtained from colourings c_{lc} and c_{rc}.

In this case, these two colourings need to be modified as follows. For $m = 1, 2$ and any vertex $a \in R_m$, if $c_{lc}(a) \notin \mathcal{I}^m_{lc(w)}$, redefine $c_{lc}(a) := i$. And if $c_{rc}(a) \notin \mathcal{I}^m_{rc(w)}$, redefine $c_{rc}(a) := j(v')$. Note that these redefinitions do not depend on u so as we claimed previously, they can be performed in the preprocessing step instead of the main algorithm.

Now, clearly $c_{lc}(a) \in \mathcal{I}^m_{lc(w)}$ and $c_{rc}(a) \in \mathcal{I}^m_{rc(w)}$ for each vertex $a \in R_m$, $m = 1, 2$. And it follows easily from the proof of Lemma 7 that for each vertex $a \in R_m$, either $c_{lc}(a)$ or $c_{rc}(a)$ is the colour of a in an $\mathcal{I}_w$-colouring of R w.r.t. u. This gives the desired division of the colouring problem into two subproblems.

Running time of the algorithm is not affected since each colouring can be modified in $O(r)$ time, giving a total additional time of $O(r^{\sqrt{r}\log r})$ in the preprocessing step.

Two more details are needed. First, in Figure 2, the index set $\mathcal{I}_w$ associated with region R consists of indices which are consecutive in cycle C. This is not the case for regions at deeper levels of recursion (for instance, the indices associated with R_2 in Figure 2 are not consecutive in C) but all our arguments still carry through in this case.

Finally, the recursively defined regions should not share vertices except those on shortest paths bounding the regions (for instance, regions R_1 and R_2 share only vertices of $P_i(v') \cup P_i'(v')$). This can always be achieved since shortest paths may be chosen such that they do not cross. For instance, when dividing region R_1 into two smaller regions, the two shortest paths defining this division can be chosen such that they do not cross $P_i(v') \cup P_i'(v')$.

Proof of Lemma 2

To prove the lemma, we need the following result.

Lemma 8. *In the above tree set construction,* $|\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}| + |\mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}| = |\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}| + 2$ *and* $|\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}|, |\mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}| < |\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}|$.

Proof. The first part follows from the observation that $\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)} = \mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)} \cup \mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}$ and that i and $j(v')$ are the only indices of $\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}$ shared by $\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}$ and $\mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}$.

To show the second part, suppose vertex v' corresponds to an entry in array $\mathcal{A}_i$. Set $\mathcal{I}^1_{lc(w)}$ contains an index k which is either the first or the last index of $\mathcal{I}_{lc(w)}$. By the choice of v, i is neither the first nor last index of $\mathcal{I}_{lc(w)}$ so $k \neq i$. The only index of $\mathcal{I}_{lc(w)}$ shared by $\mathcal{I}^1_{lc(w)}$ and $\mathcal{I}^2_{lc(w)}$ is i so $k \notin \mathcal{I}^2_{lc(w)}$. Hence, $k \notin \mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}$, implying that $|\mathcal{I}^2_{lc(w)} \cup \mathcal{I}^2_{rc(w)}| < |\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}|$. A similar argument shows that $|\mathcal{I}^1_{lc(w)} \cup \mathcal{I}^1_{rc(w)}| < |\mathcal{I}_{lc(w)} \cup \mathcal{I}_{rc(w)}|$. The inequalities also follow if v' corresponds to an entry in $\mathcal{A}_j$. □

We can now prove Lemma 2. Consider a tree represented by $\mathcal{TS}$. Lemma 8 implies that the number of leaves in this tree is at most $l(|\mathcal{I}_w|)$, where $l : \mathbb{N} \to \mathbb{N}$ is defined by

$$l(k) = \begin{cases} \max\{l(k_1) + l(k_2) | k_1 + k_2 = k + 2, k_1, k_2 < k\} & \text{if } k > 4 \\ 1 & \text{if } k \leq 4, \end{cases}$$

where we used the fact that there is no choice for v when $|\mathcal{I}_w| \leq 4$ in the algorithm that constructs $\mathcal{TS}$. Since the number of non-leaf nodes is one less than the number of leaves, the lemma will follow if we can show that $l(k) \leq k - 2$ for $k \geq 3$.

The proof is by induction on $k \geq 3$. If $k = 3$, we have $l(k) = 1 = k - 2$ by definition so assume that $k > 3$ and that the induction hypothesis holds for smaller values than k. Let k_1, k_2 be given where $k_1 + k_2 = k + 2$ and $k_1, k_2 < k$. Then $l(k_1) + l(k_2) \leq k_1 + k_2 - 4 = k - 2$. Hence,

$$l(k) = \max\{l(k_1) + l(k_2) | k_1 + k_2 = k + 2, k_1, k_2 < k\} \leq k - 2,$$

as requested.

Proof of Lemma 3

We need the following result.

Lemma 9. *Given a binary tree T of $\mathcal{TS}$, the total number of distinct colour pairs associated with leaves of T is $O(|\mathcal{I}_w|)$.*

Proof. Let T be a tree of $\mathcal{TS}$. For any of its non-leaf vertices, if there are k indices associated with this vertex then the total number of indices associated with its two children is $k+2$ by Lemma 8. Hence, if m is the number of non-leaf vertices of T then the total number of indices associated with leaves of T is $2m+|\mathcal{I}_w|$ which is $O(|\mathcal{I}_w|)$ by Lemma 2.

Let $\mathcal{I} \subseteq \mathcal{I}_{lc(w)}$ and $\mathcal{I}' \subseteq \mathcal{I}_{rc(w)}$ be the index sets and let A be the subset of vertices associated with a leaf of T. By the above, the lemma will follow if we can show that the number of distinct colour pairs in $\mathcal{I} \times \mathcal{I}'$ of vertices in A is $O(|\mathcal{I} \cup \mathcal{I}'|)$.

Since we are in a leaf of T, we have that for any such colour pair (i, j), either i is the first or last index of $\mathcal{I}$ or j is the first or last index of $\mathcal{I}'$. This implies that the number of distinct colour pairs is at most $2|\mathcal{I} \cup \mathcal{I}'|$, as requested. $\square$

We are now ready to prove Lemma 3. Let $u \in U'$. Define $A_1, \ldots, A_p$ as the subsets of vertices associated with leaves of the binary tree traversed by u. From the construction of $\mathcal{TS}$ and the definition of u-traversal, it follows that $\cup_{k=1}^{p} A_k$ is the set of vertices of R.

Let $k \in \{1, \ldots, p\}$ be given and let l_k denote the leaf with which A_k is associated. Let $c : A_k \to \mathcal{I}$ and $c' : A_k \to I'$ be the colourings of A_k associated with l_k, where $\mathcal{I} \subseteq \mathcal{I}_{lc(w)}$ and $\mathcal{I}' \subseteq \mathcal{I}_{rc(w)}$.

From the way we choose v'-vertices, it follows that for each $x \in A_k$, either $c(x)$ or $c'(x)$ is a colour of x in an $\mathcal{I}_w$-colouring of R w.r.t. u. The lemma follows from Lemma 9 by letting $V_1, \ldots, V_m$ be maximal sets each containing vertices with identical colour pairs.

Proof of Lemma 4

Assume first that $D = d_R(p_i, p_j) < \infty$. For any $u \in U'$, the triangle inequality implies that $|d_G(u, p_i) - d_G(u, p_j)| \le D$. Observe that if $d_G(u, p_i) - d_G(u, p_j) = -D$ then there is an $\{i, j\}$-colouring of R w.r.t. u where all vertices of R have colour c_i. And if $d_G(u, p_i) - d_G(u, p_j) = D$ then there is an $\{i, j\}$-colouring where all vertices of R have colour c_j.

Consider adding a new vertex u to G and edges $e_i = (u, p_i)$ and $e_j = (u, p_j)$. Set the weights of e_i and e_j such that $x = -D$, where $x = d_G(u, p_i) - d_G(u, p_j)$. Now, consider adjusting the weights such that x is increased continuously from $-D$ to D.

Initially, all vertices of R have colour c_i in the $\{i, j\}$-colouring of R w.r.t. u. There are event points in $[-D, D]$, where the colouring changes. Such changes occur exactly when $d_G(u, p_i) + d_R(p_i, v) = d_G(u, p_j) + d_R(p_j, v)$, i.e. when $x = d_R(p_j, v) - d_R(p_i, v)$ for some vertex $v \in R$.

Since there are n_R vertices in R, we have shown that there are at most $n_R + 1$ distinct $\{i, j\}$-colourings and that each colouring corresponds to an interval between two consecutive event points in $[-D, D]$. By ordering the event points, we can apply binary search to find, in $O(\log r)$ time, the interval corresponding to an $\{i, j\}$-colouring of R w.r.t. a $u \in U'$, assuming we are given $d_G(u, p_i)$ and $d_G(u, p_j)$. This shows the lemma when $D < \infty$.

Now, assume that $D = \infty$ and let $u \in U'$. Then for any vertex $v \in R$, if $d_R(p_i, v) < \infty$ then v can be assigned colour c_i in an $\{i, j\}$-colouring of R w.r.t. u. And if $d_R(p_j, v) < \infty$ then v can be assigned colour c_j in an $\{i, j\}$-colouring of R w.r.t. u. Finally, if $d_R(p_i, v) = d_R(p_j, v) = \infty$ then v can be assigned either of the two colours c_i and c_j in an $\{i, j\}$-colouring of R w.r.t. u. This shows the lemma when $D = \infty$.

Proof of Theorem 1

Let us bound the time for the preprocessing step and the size of the data structure obtained. We will assume that boundary vertex distances in G have been precomputed, and that for each region R' in the r-division of G and for each cycle $C \in \mathcal{C}_{R'}$, SSSP distances in $\overline{Int}(C)$ (or in $\overline{Ext}(C)$ if $R' \subseteq \overline{Int}(C)$) have been precomputed for each boundary vertex of R' belonging to C. The latter precomputation allows us to efficiently compute the lengths of all abstract edges of

R'. The total time to compute all boundary vertex distances and lengths of abstract edges over all regions is $O(n^2/\sqrt{r})$ if the linear time SSSP algorithm of [7] is applied.

From the description and analysis of the preprocessing step, it is easy to see that it has running time at most a factor polynomial in r larger than the number of level 3-table entries, given the above precomputations (here we also use Lemma 6). We will show that the main tree $\mathcal{T}$ and its associated data structures contain a total of $O(r^{O(\sqrt{r}\log r)})$ level 3-table entries. This will imply that the total running time of the preprocessing step is $O(r^{O(\sqrt{r}\log r)})$ in addition to the $O(n^2/\sqrt{r})$ time above.

Let w be a vertex of $\mathcal{T}$. We prove by induction on the height ≥ 0 of subtree $\mathcal{T}_w$ that the total number of level 3-table entries associated with w is at most $r^{c_1|\mathcal{I}_w|\log(|\mathcal{I}_w|)}$ for some constant c_1. Since $\mathcal{T}$ has $O(\sqrt{r})$ vertices, this will show our claim.

If the height is zero then $\mathcal{T}_w$ is a leaf. Since a leaf has no associated data structure, our claim trivially holds in this case.

Now, suppose the height is at least one and that the induction hypothesis holds for smaller heights. Since the level 1-table T_w associated with w has a row resp. column for each level 3-table entry associated with $lc(w)$ resp. $rc(w)$, it follows from the induction hypothesis that T_w has at most $r^{c_1\lceil|\mathcal{I}_w|/2\rceil\log(\lceil|\mathcal{I}_w|/2\rceil)}$ rows and at most $r^{c_1(|\mathcal{I}_w|-\lceil|\mathcal{I}_w|/2\rceil)\log(|\mathcal{I}_w|-\lceil|\mathcal{I}_w|/2\rceil)}$ columns.

Consider some entry of T_w. The associated level-2 table has at most $r^{c_2|\mathcal{I}_w|}$ entries for some constant c_2. The number of level 3-table entries associated with each entry of that level-2 table is at most $r^{c_3|\mathcal{I}_w|}$ for some constant c_3. Hence, the total number of level 3-table entries associated with a single entry of T_w is at most $r^{c_4|\mathcal{I}_w|}$ where $c_4 = c_2c_3$.

Since the height is at least one, we have $|\mathcal{I}_w| > 1$, implying that $\lceil|\mathcal{I}_w|/2\rceil < \frac{3}{4}|\mathcal{I}_w|$. From this and the above, it follows that the total number of level 3-table entries associated with w is at most

$$\begin{aligned} r^{c_1(\lceil|\mathcal{I}_w|/2\rceil+|\mathcal{I}_w|-\lceil|\mathcal{I}_w|/2\rceil)\log(\lceil|\mathcal{I}_w|/2\rceil)+c_4|\mathcal{I}_w|} &= r^{c_1|\mathcal{I}_w|\log(\lceil|\mathcal{I}_w|/2\rceil)+c_4|\mathcal{I}_w|} \\ &< r^{c_1|\mathcal{I}_w|\log(\frac{3}{4}|\mathcal{I}_w|)+c_4|\mathcal{I}_w|} \\ &= r^{c_1|\mathcal{I}_w|\log(|\mathcal{I}_w|)+c_1\log(\frac{3}{4})|\mathcal{I}_w|+c_4|\mathcal{I}_w|} \end{aligned}$$

Thus, if we choose c_1 sufficiently large, i.e. such that $c_1\log(\frac{3}{4}) \leq -c_4$ then the total number of level 3-table entries associated with w is at most $r^{c_1|\mathcal{I}_w|\log(|\mathcal{I}_w|)}$, as requested.

Proof of Lemma 5

Let $u \in U$ be given. Let w be any non-leaf vertex of main tree $\mathcal{T}$ and assume that we are given the two integers i_{lc} and i_{rc} representing, respectively, an $\mathcal{I}_{lc(w)}$-colouring and an $\mathcal{I}_{rc(w)}$-colouring of R w.r.t. u. We will show that we can obtain the integer representing an $\mathcal{I}_w$-colouring of R w.r.t. u in $O(|\mathcal{I}_w|\log r)$ time. This will imply the lemma since $\mathcal{T}$ has height $O(\log r)$, since the sum of the sizes of index sets associated with vertices of the same depth in $\mathcal{T}$ is $O(|\mathcal{I}_w|)$, and since $|\mathcal{I}_w| = t$ when w is the root of $\mathcal{T}$.

We can obtain the tree set $\mathcal{TS}$ associated with entry (i_{lc}, i_{rc}) of the level 1-table of w in constant time. It suffices to show that the binary tree of $\mathcal{TS}$ traversed by u can be found in $O(|\mathcal{I}_w|\log r)$ time. For suppose this tree is given. Then a depth-first traversal of it gives the vector mapping the tree to an entry of the associated level 2-table in $O(|\mathcal{I}_w|)$ time by Lemma 2. And given this entry, we can find the entry in the appropriate level 3-table in $O(|\mathcal{I}_w|\log r)$ time with preprocessing time polynomial in r by Lemmas 3 and 4.

To show that the tree of $\mathcal{TS}$ traversed by u can be found in $O(|\mathcal{I}_w|\log r)$ time we will show that the vertex v' can be found with binary search in one of the two arrays $\mathcal{A}_i$ and $\mathcal{A}_j$. This will show our claim since the tree traversed by u has $O(|\mathcal{I}_w|)$ vertices by Lemma 2.

Deciding which of the two arrays $\mathcal{A}_i$ and $\mathcal{A}_j$ should be searched can be done in constant time since this involves determining whether $d_G(u,p_i)+d_R(p_i,v)\geq d_G(u,p_j)+d_R(p_j,v)$ and since boundary vertex distances have been precomputed.

Suppose w.l.o.g. that the algorithm decides to search in $\mathcal{A}_i$. Then $d_G(u,p_i)+d_R(p_i,v)\geq d_G(u,p_j)+d_R(p_j,v)$. Vertex v' should be picked such that it is first on P_i satisfying $d_G(u,p_i)+d_R(p_i,v')\geq d_G(u,p_{j(v')})+d_R(p_{j(v')},v')$.

Suppose for some vertex v'' on P_i, $d_G(u,p_i)+d_R(p_i,v'')<d_G(u,p_{j(v'')})+d_R(p_{j(v'')},v'')$, where $j(v'')$ is the colour of v'' in the $\mathcal{I}_{rc(w)}$-colouring of R w.r.t. u. Then this inequality holds when replacing v'' with any vertex preceding v'' in P_i since P_i is a shortest path in R.

Thus, we can find v' as follows. Start with v'' as the middle vertex of P_i. If $d_G(u,p_i)+d_R(p_i,v'')\geq d_G(u,p_{j(v'')})+d_R(p_{j(v'')},v'')$ then v' cannot be any of the vertices succeeding v'' in P_i since it is the first vertex satisfying $d_G(u,p_i)+d_R(p_i,v')\geq d_G(u,p_{j(v')})+d_R(p_{j(v')},v')$. Hence, we can exclude half the vertices. On the other hand, if $d_G(u,p_i)+d_R(p_i,v'')<d_G(u,p_{j(v'')})+d_R(p_{j(v'')},v'')$ then the above shows that v' cannot be any of the vertices preceding v'' in P_i and again we can exclude half the vertices.

It follows that binary search can be applied to find v'. Since boundary vertex distances have been precomputed and since $\mathcal{A}_i$ and $\mathcal{A}_j$ have $O(r)$ entries, it follows that it takes $O(\log r)$ time to find v'. A similar argument can be applied to searches in deeper levels of tree set $\mathcal{TS}$. From the discussion above, this suffices to prove the lemma.

Proof of Theorem 2

We start by precomputing boundary vertex distances in G in a total of $O(n^2/\sqrt{r})$ time.

In the main algorithm, vertices of U are mapped to integers in the range $\{1,\ldots,N\}$, where N is the total number of level 3-table entries. By Theorem 1 and Lemma 5 this takes a total of

$$O(r^{c'\sqrt{r}\log r}+|U|\sqrt{r}\log^2 r)$$

time for some constant c'. So to show one part of the theorem, that integers $f(u)$ for $u\in U$ can be computed in a total of $O(n^\epsilon+|U|\log n)$ time for some constant $\epsilon<1$, we will pick constant c such that $\sqrt{r}\log^2 r=O(\log n)$, $r^{c'\sqrt{r}\log r}=O(n^\epsilon)$, and $n^2/\sqrt{r}=O(n^2(\log\log n)^2/\log n)$. This will also show $N=O(n^\epsilon)$.

For sufficiently large n,

$$\begin{aligned}
r^{c'\sqrt{r}\log r} &= (c\log n/(\log\log n)^2)^{2c'(c\log n/(\log\log n)^2)\log((c\log n/(\log\log n)^2)^2)}\\
&\leq (c\log n)^{(4c'c\log n/(\log\log n)^2)\log(c\log n)}\\
&\leq (c\log n)^{(8c'c\log n/(\log\log n)^2)\log\log n}\\
&\leq (\log n)^{16c'c\log n/\log\log n}\\
&= 2^{16c'c\log n}\\
&= n^{16c'c}.
\end{aligned}$$

By setting $c<1/(16c')$, we have $r^{c'\sqrt{r}\log r}=O(n^\epsilon)$ with $\epsilon<1$. Also,

$$\sqrt{r}\log^2 r=O((\log n/(\log\log n)^2)(\log\log n)^2)=O(\log n).$$

And finally, $n^2/\sqrt{r}=O(n^2(\log\log n)^2/\log n)$, as requested.

To bound the time to compute the colourings, we observe that each colouring can be obtained in time polynomial in r. Thus, the total time to compute all colourings is $O(Nr^{O(1)})=O(r^{O(\sqrt{r}\log r)})$. This is $O(n^\epsilon)$ for c sufficiently small.

Proof of Corollary 1

Let G be defined as above. We start by applying Theorem 2 to G. Total running time for this is $O(n^2(\log\log n)^2/\log n + \frac{n}{r}n\log n) = O(n^2(\log\log n)^4/\log n)$.

We may assume that distances in G from each boundary vertex to all vertices of G and distances in G between each pair of vertices belonging to the same region have been precomputed. We may also assume that each vertex of G is associated with a region containing that vertex.

Consider one of the regions R in the r-division. We may assume that colourings $c_1, \ldots, c_N$ of R are stored in arrays such that for any $u \in V \setminus V_R$, the colour of each vertex of R w.r.t. the u-colouring $c_{f(u)}$ of R can be found in constant time.

Now, suppose we need to answer a query for distance $d_G(u,v)$. We find a region R containing v in constant time. We then obtain the colour i of v in the colouring $c_{f(u)}$ of R, also in constant time. Since $c_{f(u)}$ is a u-colouring of R, we have $d_G(u,v) = d_G(u,p_i) + d_R(p_i,v)$ so we can find $d_G(u,v)$ in constant time with the above precomputations.

Proof of Theorem 6

We can generalize the result in Corollary 1 by obtaining a tradeoff between running time and space requirement. The idea is to apply a slightly more general version of Theorem 2.

First, we compute an r-division of G with r unspecified for now. In $O(n^2/\sqrt{r} + r^3)$ time, we precompute distances in G from each boundary vertex to all vertices of G and distances in G between each pair of vertices belonging to the same region. We may assume that each vertex of G is associated with a region containing that vertex.

For each region R, we do as follows. For each choice of U and C (defined as in Section 3), we partition the boundary vertices of R belonging to C into $O(m)$ groups, each containing $O(t/m)$ vertices, where t is the number of boundary vertices of R. For each group, we apply Theorem 2 but still with r unspecified and with the restriction that shortest paths are only allowed to use boundary vertices in that group.

Now, suppose we need to answer a query for distance $d_G(u,v)$. We find a region R containing v in constant time. In the proof of Corollary 1, we obtained in constant time a colour i of v such that $d_G(u,v) = d_G(u,p_i) + d_R(p_i,v)$. Now, we instead obtain $O(m)$ colours, one for each group. For one of these colours i, $d_G(u,v) = d_G(u,p_i) + d_R(p_i,v)$ so we can find the distance in G from u to v in $O(m)$ time.

From the above and from the proof of Theorem 2, it follows easily that the total preprocessing time is

$$O\left(nr^2 + \frac{n^2\log^2 r}{\sqrt{r}} + nr^{\frac{c\sqrt{r}\log r}{m}}\right)$$

for some constant c.

Setting $m = 2c\sqrt{r}\log^2 r/\log n$, we get a preprocessing time of

$$\begin{aligned} O\left(nr^2 + \frac{n^2\log^2 r}{\sqrt{r}} + n2^{\frac{c\sqrt{r}\log^2 r}{m}}\right) &= O\left(nr^2 + \frac{n^2\log^2 r}{\sqrt{r}} + n2^{\frac{1}{2}\log n}\right) \\ &= O\left(nr^2 + \frac{n^2\log^2 r}{\sqrt{r}} + n^{3/2}\right) \end{aligned}$$

which is $O(nr^2 + \frac{n^2\log^2 r}{\sqrt{r}})$ since $r = O(n)$. When $r = O(n^{2/5}\log^{4/5} n)$, we get $O(\frac{n^2\log^2 r}{\sqrt{r}})$ preprocessing time and query time is $O(m) = O(\sqrt{r}\log^2 r/\log n)$. Or if we let $S = \sqrt{r}/\log^2 r = O(n^{1/5}/\log^{8/5} n)$, we get $O(n^2/S)$ preprocessing time and $O(S\log^4 S/\log n)$ query time, as requested.

Allowing arbitrary real weight edges

So far, we have only considered non-negative edge weights. We now show how Theorems 3, 4, and 6, and Corollary 1 can be generalized to graphs with arbitrary real edge weights and no negative weight cycles.

Theorem 3

First, Theorem 3. Let $w : E \to \mathbb{R}$ be the weight function for graph G and let $\tilde{w}$ be the *reduced weight function* $\tilde{w} : E \to \mathbb{R}$, defined by

$$\tilde{w}(u, v) = d_G(x, u) + w(u, v) - d_G(x, v)$$

for each edge $(u, v) \in E$, where x is any (fixed) vertex of G and d_G is the shortest path distance function for G induced by w. Clearly, G has non-negative edge weights w.r.t. $\tilde{w}$ and it is well-known that shortest paths are preserved when switching from w to $\tilde{w}$. Let $\tilde{d}_G$ be the shortest path distance function induced by $\tilde{w}$. Note that for any vertices $u, v \in V$, $\tilde{d}_G(u, v) = d_G(x, u) + d_G(u, v) - d_G(x, v)$ (telescoping sum).

For subsets $V_1, V_2 \subseteq V$, define $\sum(V_1, V_2) = \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} d_G(v_1, v_2)$ as before and define $\tilde{\sum}(V_1, V_2) = \sum_{v_1 \in V_1} \sum_{v_2 \in V_2} \tilde{d}_G(v_1, v_2)$

We start by precomputing boundary vertex distances in (G, w) with source x. This can be done in $O(n \log^2 n)$ time with the algorithm in [8]. Next, we precompute boundary vertex distances in $(G, \tilde{w})$ in $O(n^2/\sqrt{r}) = O(n^2(\log\log n)^2/\log n)$ time. From these distances and from SSSP distances in (G, w) with source x, we can obtain boundary vertex distances in (G, w) in $O(n^2/\sqrt{r}) = O(n^2(\log\log n)^2/\log n)$ time.

Now, apply Theorem 2 to $(G, \tilde{w})$. Going through the proof of Theorem 3, we see that what needs to be shown is that for each region R, $\sum(R, R)$ can be computed in time polynomial in r and that $\sum(\{u\}, R)$ can be computed in $O(t)$ time for each $C \in \mathcal{C}_R$ and each $u \in U_{R,C}$ with $O(n^2(\log\log n)^2/\log n)$ preprocessing time.

The former is clear since for each edge $e \in R$, $w(e)$ can be obtained from $\tilde{w}(e)$ in constant time. This follows from the assumption that SSSP distances in (G, w) with source x have been precomputed. So let us focus on computing $\sum(\{u\}, R)$.

As observed in the paper,

$$\sum(\{u\}, V_R) = \sum_{i=1}^{t} d_G(u, p_i)|V_{i,M_u}| + \sum(\{p_i\}, V_{i,M_u}),$$

where we also use the fact that shortest paths remain unchanged when switching from w to $\tilde{w}$, implying that V_{i,M_u} remains the same.

Since boundary vertex distances in (G, w) have been precomputed, $\sum_{i=1}^{t} d_G(u, p_i)|V_{i,M_u}|$ can be computed in the desired $O(t)$ time bound. So let us consider $\sum_{i=1}^{t} \sum(\{p_i\}, V_{i,M_u})$. We have

$$\begin{aligned}\sum(\{p_i\}, V_{i,M_u}) &= \sum_{v \in V_{i,M_u}} d_G(p_i, v) \\ &= \left(\sum_{v \in V_{i,M_u}} (d_G(x, p_i) + d_G(p_i, v) - d_G(x, v)) \right) - |V_{i,M_u}| d_G(x, p_i) + \sum_{v \in V_{i,M_u}} d_G(x, v) \\ &= \tilde{\sum}(\{p_i\}, V_{i,M_u}) - |V_{i,M_u}| d_G(x, p_i) + \sum(\{x\}, V_{i,M_u}).\end{aligned}$$

Value $|V_{i,M_u}| d_G(x, p_i)$ can be obtained in constant time and so can $\sum(\{x\}, V_{i,M_u})$ if we precompute this sum for each V_{i,M_u}-subset (here, we use the fact that SSSP distances in (G, w) with source x have been precomputed). We can also precompute $\tilde{\sum}(\{p_i\}, V_{i,M_u})$ as described in the main paper (since we have now applied Theorem 2 to $(G, \tilde{w})$).

It follows that $\sum(\{u\}, V_R)$ can be computed in $O(t)$ time with $O(n^2(\log\log n)^2/\log n)$ preprocessing time, as requested.

Theorem 4

For the proof of Theorem 4 to be generalized to arbitrary real edge weights, we need to show that

$$\max_{v\in V_R} d_G(u,v) = \max\left\{d_G(u,p_i) + \max_{v\in V_{i,M_u}} d_G(p_i,v) | i = 1,\ldots,t\right\}$$

can be computed in $O(t)$ time with $O(n^2(\log\log n)^2/\log n)$ preprocessing time (the rest of the proof is similar to that of Theorem 3). And this reduces to showing that $\max_{v\in V_{i,M_u}} d_G(p_i,v)$ can be computed in $O(t)$ time over all i with $O(n^2(\log\log n)^2/\log n)$ preprocessing time.

We use the same preprocessing as above for Theorem 3. For set V_{i,M_u}, we have distances $\tilde{d}_G(p_i,v)$ for each $v \in V_{i,M_u}$ and from these we can obtain $d_G(p_i,v)$ in a total of $O(|V_{i,M_u}|)$ additional time. By picking the largest such distance $d_G(p_i,v)$ in the preprocessing step, we can obtain $\max_{v\in V_{i,M_u}} d_G(p_i,v)$ in constant time so that $\max_{v\in V_R} d_G(u,v)$ can be computed in $O(t)$ time, as requested. The total preprocessing time is not increased since the total size of all V_{i,M_u}-sets is $O(r^{\sqrt{r}\log r})$.

Theorem 6 and Corollary 1

It suffices to consider Theorem 6. The extension is trivial: precompute SSSP distances in G with source x (defined above). Now, to answer distance query $d_G(u,v)$, we first obtain $\tilde{d}_G(u,v)$ using the algorithm for graphs with non-negative edge weights. We then compute $d_G(u,v)$ in $O(1)$ additional time using the formula

$$d_G(u,v) = \tilde{d}_G(u,v) + d_G(x,v) - d_G(x,u).$$