SINGLE-FACE NON-CROSSING SHORTEST PATHS IN PLANAR GRAPHS

BY

ALEX STEIGER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Jeff Erickson

Abstract

We consider the following problem: Given an n-vertex undirected planar-embedded graph with a simple boundary cycle, non-negative edge lengths, and k pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ specified on the boundary, find non-crossing shortest paths connecting all pairs of terminals (if any such paths exist). We present an algorithm to find such paths in $O(n \log\log k)$ time which improves upon the previous best runtime of $O(n \log k)$ by Takahashi, Suzuki, and Nishizeki [*Algorithmica* 1996].

To my parents, Ron and Jami Steiger.

Acknowledgments

I first and foremost want to thank my adviser, Professor Jeff Erickson, who introduced me to computational topology. This thesis could not have been completed without his insight and guidance during the research process. I thank Jeff for his support and patience while I prepared this thesis, and for our lengthy discussions about various aspects of academia, particularly those about teaching.

I also want to thank Professor Chandra Chekuri who has been a mentor to me since I was an undergraduate. I appreciate our many conversations about how to prepare for and succeed throughout the rest of my academic career.

I want to thank Tracey Hoy at the College of Lake County for her high standards, encouragement, and recommendation to tutor at the college's Math Center where I gained my first teaching experience. I strive to be as effective and inspiring as an educator as she was for me.

I would like to thank my friends and colleagues in the theory group for providing a collaborative environment and healthy distractions from work.

I am thankful to the staff in the advising office for their help, friendship, and off-the-beaten-path restaurant recommendations.

Lastly, I want to thank my family for their endless love and support, especially my parents, Ron and Jami Steiger, to whom this thesis is dedicated.

Table of Contents

Chapter 1

Introduction

In this thesis, we consider following problem: Given an n-vertex undirected planar-embedded graph with a simple boundary cycle, non-negative edge lengths, and k pairs of terminals specified on the boundary, find non-crossing shortest paths connecting all pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ (if such paths exist). Before this work, Takahashi, Suzuki, and Nishizeki [1] gave an algorithm to solve this problem which can be implemented to run in $O(n \log k)$ time using the linear time shortest path algorithm of Henzinger *et al*. [2]. Over twenty years earlier, Reif [3] gave an algorithm to find such paths when the terminals are ordered $s_1, s_2, \ldots, s_k, t_k, \ldots, t_1$ clockwise around the boundary, which can be implemented to run in $O(n \log k)$ time using the algorithm of Henzinger *et al*. Italiano, Nussbaum, Sankowski, and Wulff-Nilsen [4] recently gave a new algorithm for this case that runs in $O(n \log \log n)$ time, which we note is independent of k.

Our main contribution is an algorithm for the general case that runs in $O(n \log \log k)$ time found in chapter 9. Leading up to this result, we provide a description and analysis of Takahashi *et al*.'s algorithm for the general case in chapter 7, and an overview of the algorithm by Italiano *et al*. in chapter 8 that we modify to run in $O(n \log \log k)$ time.

Erickson and Nayerri [5] considered a generalization of this problem: Given an n-vertex undirected planar-embedded graph G with k pairs of terminals specified on h faces, find non-crossing *walks* connecting all k pairs of terminals of minimum total length (if such walks exist). They give an algorithm to solve this problem in $2^{O(h^2)} n \log k$ time by using the $O(n \log k)$-time algorithm by Takahashi *et al*. as a blackbox. Our $O(n \log \log k)$-time algorithm directly implies their algorithm can be implemented to run in $2^{O(h^2)} n \log \log k$ time.

1.1 Definitions

We use the notation $\boldsymbol{V(G)}$ and $\boldsymbol{E(G)}$ to refer to the vertices and edges of a graph G. We use the notation $\textbf{height}(\boldsymbol{T})$ to denote the height of a rooted tree T, and $\textbf{diam}(\boldsymbol{G})$ to denote the diameter of a graph G. A ***rotation system*** is a combinatorial representation of an embedded graph and records the clockwise ordering of incident edges around each vertex. In this work, we assume ***planar-embedded*** graphs are represented by a rotation system. A ***walk*** is an ordered alternating sequence of adjacent vertices and edges where the first and last elements are vertices. A ***path*** is a walk where each vertex appears at most once. We refer to the first and last vertices of a path to be the ***source*** and

destination vertices respectively. We refer to a path with source u and destination v as a ***u-to-v path***. The ***reversal*** of a u-to-v path P is the reversal of its underlying sequence of vertices and edges, a v-to-u path. For a u-to-v path P_1 and v-to-w path P_2, we denote the concatenation of paths P_1 and P_2, a u-to-w path, by $\boldsymbol{P_1 \cdot P_2}$. The ***boundary*** of a planar-embedded graph G, denoted by ∂G, is the subgraph induced by the edges of G incident to the outer (or infinite) face. The ***boundary cycle*** of a planar-embedded graph G is the (potentially non-simple) cycle obtained by traversing ∂G in clockwise order. The boundary cycle is simple if and only if ∂G is a cycle graph. An ***articulation point*** (or ***cut vertex***) v of a connected graph G is a vertex such that $G - v$ is disconnected.

1.2 Cutting planar graphs

We define an operation to ***cut*** a simple u-to-v path $P = u, \ldots, v$ in a planar-embedded graph G for any $u, v \in \partial G$ as follows. First, we create two copies of P, $P_L = u_L, \ldots, v_L$ and $P_R = u_R, \ldots, v_R$. For each edge wx where $w \in P$ and $x \notin P$, we add the edge $w_L x$ if wx emanates left of P and otherwise add the edge $w_R x$. We order the edges incident to the vertices of P_L and P_R so that they are consistent with the orderings induced by the edges that emanate left and right of P in G respectively, then remove P from the graph. This operation takes time linear in the number of edges incident to the vertices in P before the incision. We denote the planar-embedded graph resulting from cutting along a path P in a planar-embedded graph G by $\boldsymbol{G \not\downarrow P}$, notation that we borrow from Cabello and Mohar [6]. We use the notation G_L and G_R to refer to the components containing P_L and P_R respectively in $G \not\downarrow P$. We note that the bounded faces of $G \not\downarrow P$ are the bounded faces of G and each bounded face of G is in exactly one component of $G \not\downarrow P$.

1.3 r-divisions

Let G be an n-vertex planar-embedded graph, and D be a partitioning of its edges. A subgraph induced by a partition of edges in D is a ***piece*** (also called ***regions*** in some literature). The ***multiplicity*** of a vertex is equal to the number of pieces that contain in it with respect to D. A vertex with multiplicity greater than one is a ***boundary vertex***. A ***hole*** of a piece P is a face of P that is not a face of G. We say D is an ***r-division*** if D has $O(n/r)$ partitions (so that it induces $O(n/r)$ pieces), and each piece contains $O(r)$ vertices and $O(\sqrt{r})$ boundary vertices. An r-division where each piece has $O(1)$ holes is referred to as an ***r-division with few holes***. Italiano *et al.* [4] give an algorithm to compute an r-division with few holes in $O(n \log r + \frac{n}{\sqrt{r}} \log n)$ time. Two years later, Klein, Mozes, and Sommer [7] gave an $O(n)$-time algorithm to compute such an r-division.

1.4 Dense distance graphs

Let G be an n-vertex planar-embedded graph and let D be an r-division of G. Consider a piece P of D. The complete graph over the boundary vertices in P where the length of each edge uv is equal to the shortest distance between u and v via only edges in P is referred to as the ***distance clique*** for P. Since pieces may not be connected, edge lengths may be infinite. The union of the distance cliques for each piece is called the ***dense distance graph*** (DDG), originally defined by Fakcharoenphol and Rao [8]. Since each distance clique has $O(\sqrt{r}\sqrt{r}) = O(r)$ edges and there are $O(n/r)$ pieces, the size of the DDG is $O(n)$. Klein [9] describes how to construct a data structure for a given n-vertex planar-embedded graph and one of its faces f such that the length of the shortest paths between a vertex on f and any other vertex can be reported in $O(\log n)$ time. The data structure can also report the edges of shortest paths between such vertices in $O(\log\log\Delta)$ time per edge where Δ is the maximum degree of the vertices in the path. Given an r-division with few holes, Klein's data structure can be constructed for each of the $O(1)$ holes of a piece P in $O(\sqrt{r}\log\sqrt{r}) = O(\sqrt{r}\log r)$ time. Then the lengths of the $O(r)$ edges of the distance clique for P can be obtained in $O(r\log r)$ time. Since there are $O(n/r)$ pieces, it takes $O(\frac{n}{r}\cdot r\log r) = O(n\log r)$ time overall to compute all edge lengths and construct the DDG.

1.5 Fast Dijkstra for dense distance graphs

The DDG is useful for a number of reasons. For our purposes, we rely on the fact that for any two boundary vertices u and v, any shortest u-to-v path in the DDG corresponds to a shortest u-to-v path in the underlying graph and vice-versa. For an n-vertex planar-embedded graph and r-division of it, Fakcharoenphol and Rao [8] described a variant of Dijkstra's algorithm [10] to compute shortest paths between two vertices in the DDG that exploits the planarity of the pieces and runs in $O(\frac{n}{\sqrt{r}}\log^2 n)$ time. This runtime is near linear in the sum of the multiplicity of the boundary vertices, $O(n/\sqrt{r})$, and is sublinear in n for sufficiently large r, such as when $r = \log^6 n$. Mozes, Nussbaum, and Weimann [11] recently described two faster implementations of their algorithm which run in either $O(\frac{n}{\sqrt{r}}\log^2 r)$ time with $O(n\log r)$ preprocessing, or $O(\frac{n}{\sqrt{r}}\log^4 r)$ time with $O(\frac{n}{\sqrt{r}}\log^2 r)$ preprocessing.

1.6 Incomparable vertices

Let T be a rooted tree. Two nodes u and v in T are called ***incomparable*** if u is not an ancestor of v and vice-versa. We call a subset of nodes I in T ***pairwise incomparable*** if all pairs of distinct nodes in I are incomparable.

1.7 Heavy-path decomposition

We describe the ***heavy-path decomposition*** of rooted trees as follows, first defined by Sleator and Tarjan [12]. Let T be an n-vertex rooted tree. Denote by T_u the subtree of T rooted at vertex u. An edge to a child v of u is ***heavy*** if $|V(T_v)| > |V(T_u)|/2$ and ***light*** otherwise. Then at most one edge incident to u in T_u may be heavy. The heavy edges of T can be identified in $O(n)$ time using a simple recursive algorithm to mark each vertex with the size of the subtree rooted at it. A maximal connected subtree of T consisting of only heavy edges is a ***heavy path***. It follows that each vertex in a heavy path must be in different levels of T since no two edges to distinct children of a vertex are heavy. Let the ***heavy path tree*** T_H of T be the tree resulting from contracting each heavy path in T. Now consider any path from the root vertex to a leaf in T_H. Walking from the root to the leaf, at each vertex u, the size of the subtree rooted at u is at most half the size of its parents subtree, so the length of the path is at most $\log n$. Thus, $\text{height}(T_H) = O(\log n)$.

Chapter 2

The Decision Problem

In this chapter we show how to determine if non-crossing paths connecting all k pairs of terminals exists in a given n-vertex planar-embedded graph in $O(n+k)$ time.

Let G be an n-vertex undirected planar-embedded graph with k pairs of terminals $\{(s_1,t_1),(s_2,t_2),\ldots,(s_k,t_k)\}$ specified on ∂G where no terminal is an articulation point, and no shortest path between a pair of terminals contains another terminal. We define the ***connection (multi)graph*** C of such a graph G as follows. Recall that we define the boundary cycle to be the *clockwise* traversal of ∂G. Let B be the cycle graph whose vertices are the terminals of G and edges are of the form uv where v is the first terminal to succeed terminal u in the boundary cycle of G. If all vertices in ∂G are in at least one pair of terminals, then $B = \partial G$. Let D be the set of edges composed of edges whose endpoints are s_i and t_i for each $i \in \{1,2,\ldots,k\}$. We define the connection graph C for G and its pairs of terminals to be $B + D$. We construct a combinatorial embedding C_π of C as follows. For each terminal $u \in B$, we order its incident edges in the same clockwise order as its incident vertices appear in the boundary cycle of G starting from u. See an example in (a) and (b) of figure 2.1.

It is easy to see that C_π is a planar embedding if and only if there exists no pairs of terminals (s_i,t_i) and (s_j,t_j) that appear as s_i,s_j,t_i,t_j in clockwise order around ∂G. We use this fact in the proof of the following lemma.

Lemma 2.0.1. *C_π is a planar embedding of C if and only if there exists non-crossing shortest paths in G connecting all pairs of terminals.*

Proof. The backwards direction is straightforward. Consider the geometric embedding of C where the edges $uv \in B$ follow the image of the corresponding u-to-v path in ∂G, and the edges in D follow along the paths images of the paths in the solution. This geometric embedding is planar and consistent with the combinatorial embedding C_π.

Now assume C_π is a planar embedding. Consider an arbitrary edge $s_it_i \in D$. Let P_i be a shortest s_i-to-t_i path in G. Since C_π is a planar embedding, from the observation above, for all other pairs of terminals (s_j,t_j) where $j \neq i$, both of the terminals are on the same left or right side of P_i in G. Thus, there exists a shortest s_j-to-t_j path P_j in G that does not cross P_i. For sake of contradiction, suppose not. Let P_j be a shortest s_j-to-t_j path that crosses P_i, and let a and b be the first and last vertices of P_j where it crosses P_i. Then the s_j-to-a subpath P_1 of P_j, the a-to-b subpath P_2 of P_i, and the b-to-t_j subpath P_3 of P_j are shortest paths. It follows that $P_1 \cdot P_2 \cdot P_3$ is a shortest s_j-to-t_j path that does not

cross P_i in G. It follows that the lengths of the shortest paths in the component of $G \not\propto P_i$ that contains them are equal to the lengths of the shortest paths in G. We note that no terminals can become articulation points by this procedure since no shortest path between any pair of terminals contains another terminal. It follows that the connection graphs for the components of $G \not\propto P_i$ are well-defined and exactly the components of $C \not\propto s_i t_i$. See figure 2.1 for an example. By induction, repeating this process on the left and right components of $G \not\propto P_i$ paired with the respective components of $C \not\propto s_i t_i$ results in finding paths which trivially project back to non-crossing shortest paths in G for all pairs of terminals. □

This lemma easily leads to an algorithm to check if a solution exists. We construct the embedding C_π of C by ordering the incident edges around each vertex, then count its number of faces f. Euler's formula implies that $f = 2 - c + (c + k) = 2 - k$ if and only if C_π is a planar embedding of C where c is the number of distinct terminals in ∂G. Constructing C_π and counting its faces can be done in $O(n + k)$ time [13].

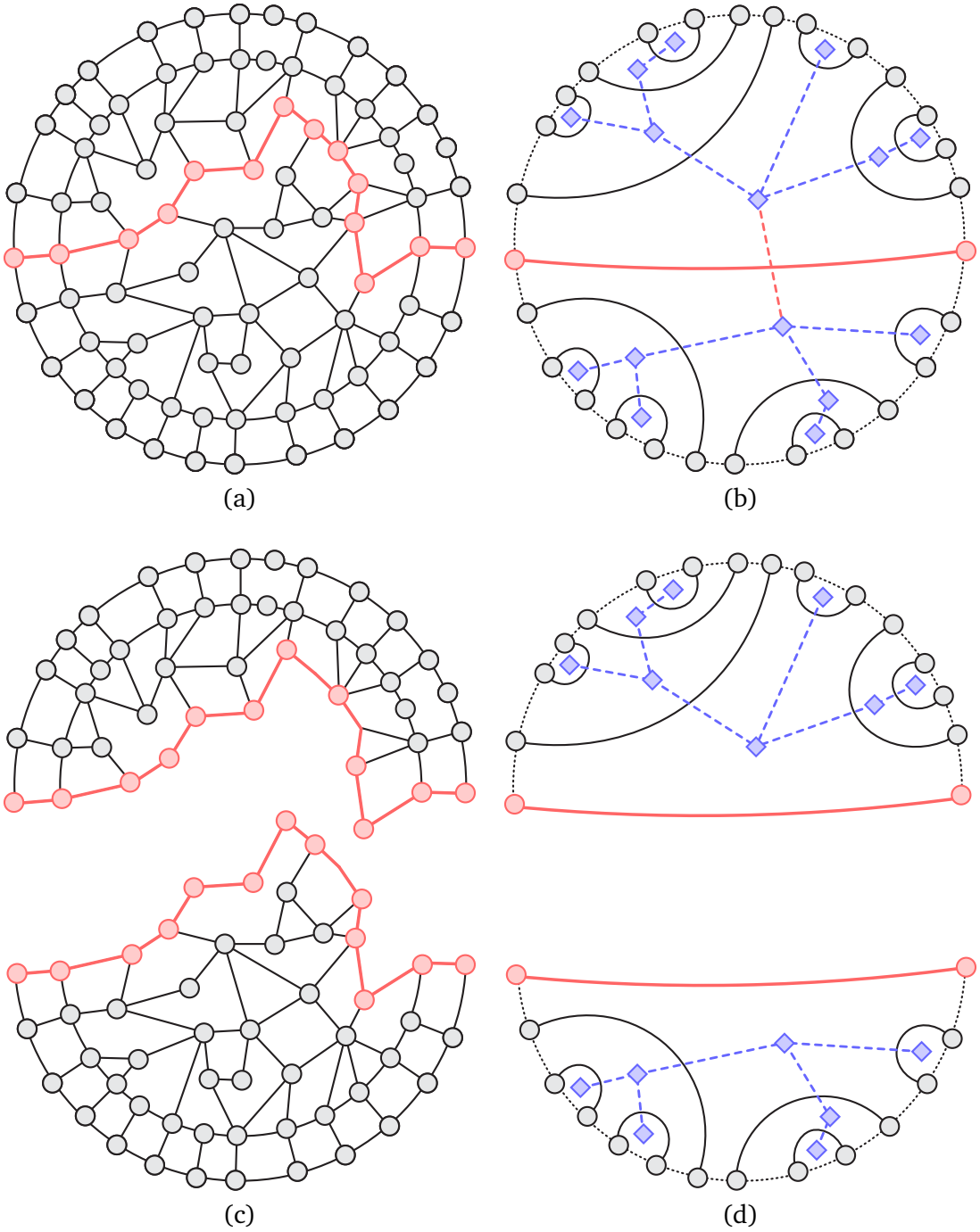

Figure 2.1: (a): A graph G (in black) and shortest path P_i between a pair of terminals (s_i, t_i) (in red). (b): The connection graph C for G and its weak dual tree (in blue). (c): $G \not\propto P_i$. (d): The connection graphs for the components of $C \not\propto P$ and their weak dual trees (in blue). We refer to the weak dual tree of a connection graph as a *terminal tree* which we formally introduce and discuss in chapter 5.

Chapter 3

Problem Formulation

An instance of this problem consists of an undirected simple planar-embedded graph G with a simple boundary ∂G, non-negative edge lengths, and k pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ specified on ∂G. We assume non-crossing shortest paths connecting all pairs of terminals exists since we can determine otherwise in linear time. Under this assumption, Euler's formula implies $k = O(n)$ since the corresponding connection graph has $O(n)$ vertices and $O(n + k)$ edges.

Without loss of generality, we further assume any shortest path connecting a pair of terminals contains terminals only at its endpoints and each vertex on ∂G is in exactly one pair of terminals. Both of these properties can be enforced by connecting a new vertex embeddded in the outer face to each terminal with zero length edges for each pair containing them, pairing these new vertices accordingly, then adding edges with infinite length between the newly added vertices. This introduces exactly $2k$ new vertices and $3k$ edges and takes $O(n + k) = O(n)$ time overall. This modified graph maintains the other assumed properties. See figure 3.1 for an example.

Lastly, we assume an ordering over the terminals around ∂G. For any edge $s_i t_i$ incident to any leaf in the terminal tree T, s_i and t_i are neighbors in ∂G, and thus we can assume the terminals are labeled so that s_1 immediately succeeds t_1 in clockwise order around ∂G. Then we can label the terminals so that for all $i > 1$, s_i precedes s_{i+1} and t_i in the boundary cycle starting from s_1. Lastly, we assume each terminal u is indexed with its position in the s_1-to-t_1 path clockwise around ∂G, which we denote by $\text{idx}(u)$. This implies $\text{idx}(s_1) = 1$, $\text{idx}(s_2) = 2$, and $\text{idx}(t_1) = 2k$. For convenience, we use the notation $[i, j]$ for integers $i, j \leq 2k$ to be the usual interval of integers from i to j when $i \leq j$, and $[j, 2k] \cup [1, i]$ when $i > j$. Then $[i, j]$ is the subset of indices of terminals located on the boundary cycle from the terminal with index i to the terminal with index j, inclusive.

Some algorithms presented later will assume additional properties about the input graph to simplify their analysis. These are described at the beginning of their respective chapters.

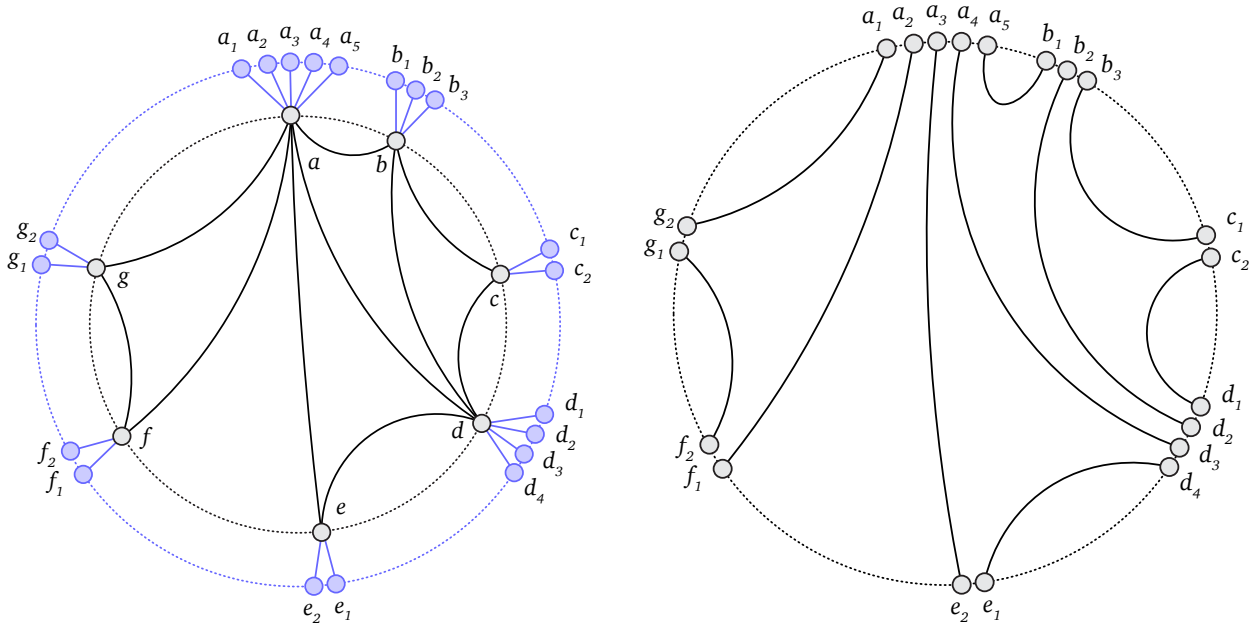

Figure 3.1: On the left, a connection graph (in black) for ten pairs of non-distinct terminals in an underlying graph G. For each terminal in G, a new terminal (in blue) is added for each pair it is contained in, i.e. for each incident solid edge in the connection graph. The new boundary consists of infinite length edges (in blue). On the right, the connection graph for the modified graph and new pairs of terminals.

Chapter 4

Output Representation

Let G be an n-vertex undirected planar-embedded graph with k pairs of terminals specified on ∂G. The number of edges in any shortest path in G may be $\Omega(n)$, and any two shortest paths may share any number of edges, even if they are non-crossing. Thus, the number of edges in any solution to our problem may be $\Omega(nk)$, so explicitly constructing and outputting the solution takes longer than we can afford. Instead, we construct an implicit representation of the solution from which a represented path between any pair of terminals can be reported in $O(\log \Delta)$ time per edge. When the graph has constant maximum vertex degree, we output a path in time proportional to its number of edges.

The representation proposed by Takahashi *et al.* [1] is stated to be a collection of tree subgraphs of the input graph wherein for each pair of terminals there is a tree containing both terminals and the unique path between them is a shortest path. They state that each edge of the input graph is in at most two trees, so the total complexity of the forest is $O(n)$. However, it is unclear from the description of their algorithm how their subgraphs are constructed while maintaining these properties. Our description of Takahashi *et al.*'s algorithm in chapter 7 is designed to construct the implicit representation described by Polishchuk and Mitchell [14] who study a variation of this problem in a geometric setting: a single subgraph of the input graph (that may contain cycles) where each edge is marked with a constant amount of additional information. This representation uses $O(n)$ space overall. We describe it, how it represents the paths, and how to construct it in the following.

Let $\mathcal{P}$ be a collection of non-crossing shortest paths $\mathcal{P}$ connecting all k pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ on the boundary of a planar-embedded graph G. Let $\mathcal{P}'$ be the set of paths in $\mathcal{P}$ and their reversals, and let R be the union of the paths in $\mathcal{P}$ (without multiplicity). For each ordered pair of adjacent vertices u and v in R, we define $\mathcal{P}'[u, v]$ to contain the paths $P \in \mathcal{P}$ where v immediately succeeds u in P. Finally, let $R[u, v]$ be the subgraph of R that is the union of all paths in $\mathcal{P}'[u, v]$. We define $(u, v)^-$ and $(u, v)^+$ to be the indices of the closest terminals to v in clockwise and counterclockwise order from v around the boundary of its containing component in $R[u, v] - u$. The pair of indices $(u, v)^-$ and $(u, v)^+$ is the constant amount of additional information we store for each ordering of the endpoints of each edge uv in R.

The significance of these terminal indices is the following. Consider any ordered pair of adjacent vertices u and v in R. For any shortest s_i-to-t_i path P_i implicitly represented in R that contains u, v immediately succeeds u in P if and only if $\text{idx}(t_i) \in [(u, v)^-, (u, v)^+]$. This fact follows directly from the fact that the paths implicitly represented are non-crossing. See figure 4.1 for an example.

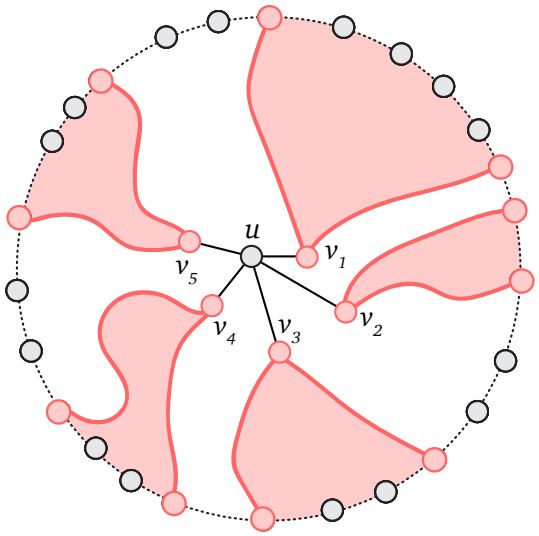

Figure 4.1: A vertex u and its incident edges uv_i in the implicit representation R of the non-crossing shortest paths connecting all pairs of terminals. The components of subgraphs $R_{(u,v_i)} - u$ containing each v_i are contained in the shaded regions for each i. The red terminals in each shaded region are those with indices $(u, v_i)^-$ and $(u, v_i)^+$.

It is easy to use these indices to report the edges of the s_i-to-t_i path P_i represented for any i. Starting at terminal s_i, find the adjacent vertex u such that $\text{idx}(t_i) \in \left[(s_i, u)^-, (s_i, u)^+\right]$. There must be exactly one such vertex u, and that edge $s_i u$ must be the first edge of P_i. Repeating this process until t_i is reached reveals each edge of P_i. This operation can be implemented using binary search to take $O(\log \Delta)$ time where Δ is the maximum vertex degree in the underlying graph, and thus reporting a path P with $|P|$ edges takes $O(|P| \log \Delta)$ time.

It is also easy to maintain these indices as the s_i-to-t_i paths as they are found for each i during the algorithms presented. We update the implicit representation to include a shortest s_i-to-t_i path P_i that does not cross any path already represented in R as follows. We traverse P_i from s_i to t_i. If a vertex or edge in P_i does not exist in R, we add it. For each vertex u and its immediate successor v in P_i, we check if $\text{idx}(t_i) \in \left[(u, v)^-, (u, v)^+\right]$ and if $\text{idx}(s_i) \in \left[(v, u)^-, (v, u)^+\right]$. If either or both inequalities do not hold, we update the stored indices accordingly. This process takes constant time per edge in P_i.

Chapter 5

A Simple $O(nk)$-time Algorithm

The proof in chapter 2 that a solution exists when C_π is a planar emebdding leads to a straightforward algorithm for finding a solution. However, before we analyze its runtime, we formalize it below in a form that the $O(n \log k)$-time and $O(n \log\log k)$-time algorithms will follow. The main differences from the previous description are that we formulate it in a recursive divide-and-conquer fashion and its parameters. We input a connected planar-embedded graph G and the weak dual graph T of its corresponding connection graph instead of the connection graph itself. In fact, since when C is planar-embedded, then it is outerplanar where $\partial C = \partial G$, and thus T is a tree. We call the weak dual tree of a connection graph the ***terminal tree*** since its edges are exactly those that represent the pairs of terminals, $s_i t_i$ for each pair of terminals (s_i, t_i). Although the algorithm presented in this chapter does not use the tree structure of T beyond its encoding of the pairs, it is used in the faster algorithms presented later. It is significant to observe that the weak dual trees of the components of $C \not\downarrow s_i t_i$ for any edge $s_i t_i \in T$ are the components of $T - s_i t_i$. Thus, we can quickly obtain the corresponding terminal trees for the recursive calls by simply removing the edge corresponding to the pair of terminals that a shortest path between was found and cut along. We note that the terminal tree closely resembles the *genealogy tree* defined by Takahashi *et al*. used to represent the pairs of terminals in [1].

```
SIMPLE(G, T):
  if T is a single node, terminate

  let s_i t_i be an arbitrary edge in T
  find a shortest s_i-to-t_i path P_i in G
  let G_L and G_R be the left and right components of G ↓ P_i,
      and let T_L and T_R be their corresponding terminal trees
  output P_i
  call SIMPLE(G_L, T_L) and SIMPLE(G_R, T_R)
```

SIMPLE(G, T) outputs non-crossing shortest paths in the given n-vertex graph G between k pairs of terminals represented by the given terminal tree T.

As per the proof of correctness from the previous section, what remains is to analyze its runtime. Finding a shortest path in a planar graph can be done in time linear in its number of vertices using the shortest path algorithm by Henzinger *et al*. [2]. Cutting along the path P_i also takes linear time in the number of vertices in the graph. Lastly, outputting the path takes time proportional to the

number of vertices in P_i, which is bounded by the number of vertices in the graph. Since the graph input to each recursive call is a subgraph of the initial graph, the amount of work done in each call is $O(n)$, giving an overall runtime of $O(nk)$.

We emphasize that this algorithm is correct for *any* choice of the pairs of terminals in each call, and thus the order in which the paths are found and cut along is independent of the correctness. The main ideas behind the algorithms in the following chapters are to identify an ordering of the pairs of terminals (edges of the initial terminal tree) and implicitly represent the paths found so that the overall runtime is improved.

Chapter 6

Takahashi *et al.*'s $O(n)$-time Algorithm for Star Terminal Trees

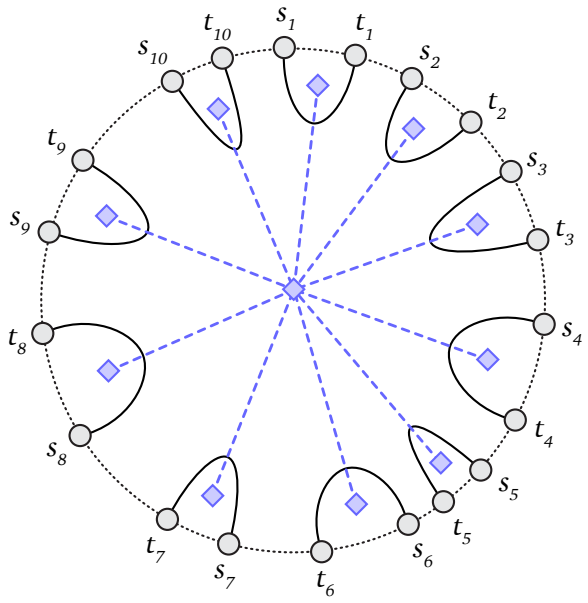

Figure 6.1: A star terminal tree (in blue) and its corresponding connection graph (in black).

In this section we present the linear time algorithm of Takahashi *et al.* [1] for the case when the terminal tree is a star. This is used as a subroutine in their $O(n \log k)$-time algorithm that we present in the next chapter.

We begin by observing that if the terminal tree T is a star, then the terminals can be relabeled so that they appear as $s_1, t_1, s_2, t_2, \ldots, s_k, t_k$ in clockwise order around ∂G. Takahashi *et al.* prove the following lemma which we rewrite using our notation and terminology.

Lemma 6.0.1. *Let S be a shortest path tree in G rooted at s_1. For each $i > 1$, denote by S_i the unique s_i-to-s_{i+1} path in S. Then for each edge $s_i t_i \in T$, there exists a shortest s_i-to-t_i path P_i in G that does not cross S_i, i.e. is contained in $G \not\curvearrowleft S_i$.*

Using this lemma, they describe the following linear time algorithm to find non-crossing shortest paths connecting all pairs of terminals in this case. They find a shortest path tree S rooted at s_1 using the linear time algorithm of Henzinger *et al.* [2]. Then they cut along S_i for each i, and use the algorithm of Henzinger *et al.* in each resulting component to find a shortest s_i-to-t_i path for each

i. The paths between the terminals are non-crossing in G. This is summarized in the pseudocode below.

```
TSN-STAR(G, T):
  label the terminals s_1, t_1, s_2, t_2, ..., s_k, t_k around ∂G
  let S be a shortest path tree rooted at s_1
  let S_i be the unique s_i-to-s_{i+1} path in S for each i

  for each component G_i in (...((G ✂ S_1) ✂ S_2)...) ✂ S_k:
      find a shortest s_i-to-t_i path P_i in G_i
      output P_i
```

TSN-STAR(G, T) outputs non-crossing shortest paths in the given n-vertex graph G between k pairs of terminals represented by the given path terminal tree T.

To analyze the runtime, we observe that each edge in G is contained in either zero or two paths S_i and S_{i+1} for some i, so explicitly cutting along S_i for each i takes $O(n)$ time overall. This further implies that the total number of vertices across the subgraphs is $O(n)$ since each edge of G is contained in at most two resulting components, and thus the sum of the work to find all paths P_i using the algorithm of Henzinger *et al.* is $O(n)$. Furthermore, the total complexity of the paths is $O(n)$ since each edge in G is contained in at most two paths in the solution so we can afford to explicitly output the paths. The theorem below summarizes this result.

Theorem 6.0.2. *Let G be an n-vertex undirected planar-embedded graph with a simple boundary cycle, non-negative edge lengths, and k pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ specified on ∂G. Let T be the corresponding terminal tree whose edges represent the pairs of terminals. If T is a star, then non-crossing shortest paths connecting all pairs of terminals can be found and output in $O(n)$ time.*

Chapter 7

Takahashi *et al.*'s $O(n \log k)$-time Algorithm for Arbitrary Terminal Trees

In this section we will show how Takahashi *et al.* [1] use the algorithm of the previous section to speed up the simple $O(nk)$ time algorithm for the general case to run $O(n \log k)$ time. First, we describe the additional properties of G and its pairs of terminals that we assume for purposes of analysis and convenience, and how to obtain them. We assume that G has constant maximum vertex degree, and that all bounded faces are triangles. To obtain the former, we replace each vertex with a binary tree gadget with zero length edges so that all vertices have degree three (see figure 7.1). To obtain the latter, we then triangulate each bounded face of G with edges of infinite length by "zig-zagging" which triples the degree of each vertex as worst (see figure 7.2). These transformations take $O(n)$ time and increase the size of the graph by $O(n)$ vertices and edges. It is significant to note that a path P' in the modified graph may have $\Omega(\log \Delta)$ times as many edges in its corresponding path P in the original graph. This will affect the amount of time it takes to report a path from the implicit representation R of the paths found by a factor of $\log \Delta$ in the worst case.

Next we establish the following lemma based on the observations of Takahashi *et al.*

Lemma 7.0.1. *Let C be the connection graph and T the corresponding terminal tree T rooted at some node r. Let I be a subset of pairwise incomparable nodes in T when rooted at r, and let C' be the connection graph induced by the pairs of terminals represented by the parent edges of the nodes of I in T. Then the terminal tree T' corresponding to C' is a star.*

Proof. Consider iteratively deleting the edges of $T - E(T')$ from C to obtain C'. This is equivalent to iteratively contracting those edges in T to obtain T'. After the each contraction the root node does not change and the nodes in I remain pairwise incomparable. It follows that the nodes in I are pairwise incomparable in T', and thus it is a star. □

We use this fact to find non-crossing shortest paths for potentially many pairs of terminal in each recursive call while taking time linear in the number of the input graph's vertices by using the algorithm from the previous chapter, TSN-Star.

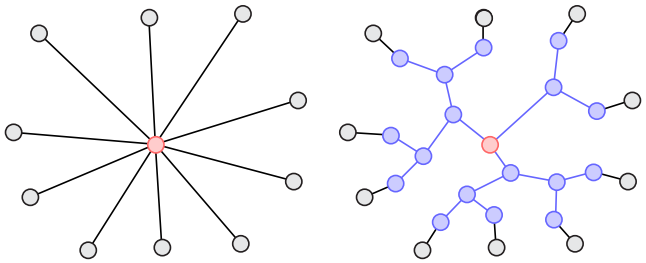

Figure 7.1: On the left, a vertex u of arbitrary degree (in red) and its neighbors (in black) in a planar-embedded graph G. On the right, the red vertex has been replaced with a binary tree gadget (in blue) with edges with zero length so that u has degree three. The black edges are the original edges of G.

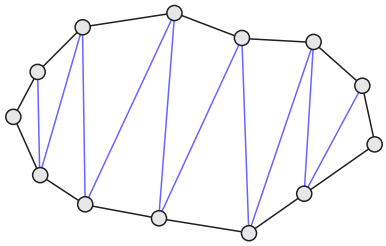

Figure 7.2: A bounded face (in black) triangulated with blue edges by "zig-zagging" between its vertices.

TSN-ARBITRARY(G, T):
  if T is a single node, terminate

  pick a node r in T and subset I of pairwise incomparable nodes in T rooted at r
  let T_I be the star terminal tree induced by the parent edges of the nodes in I

  call TSN-STAR(G, T_I)
  let P_i be the shortest s_i-to-t_i path found for each edge $s_i t_i \in T_I$

  for each component G_i of $(\ldots((G \not\leftarrow P_1) \not\leftarrow P_2)\ldots) \not\leftarrow P_{|I|}$:
    let T_i be the terminal tree for G_i, i.e. the corresponding subtree of $T - E(T_I)$
    add P_i to the implicit representation R
    contract all maximal paths of degree-two articulation points in G_i
      by replacing each path with one edge with equal total length
    call TSN-ARBITRARY(G_i, T_i)

TSN-ARBITRARY(G, T) finds and cuts along non-crossing shortest paths in the given n-vertex graph G between k pairs of terminals represented by the given terminal tree T, and adds them to a globally accessible implicit representation R. When the initial call is made, R is initialized as the empty subgraph of the input graph.

Except for the contraction of the degree-two articulation points in the resulting components as paths are found and cut along and the order in which the pairs of terminals are chosen, the

behavior of this algorithm is equivalent to that of the simple $O(nk)$-time algorithm. We will first give two choices for r and I which bound the recursion depth of this algorithm, then discuss how the contractions ensure that the amount of work in each call is $O(n)$ while maintaining correctness of the paths found in future calls that contain edges corresponding to contracted vertices.

The first choice of r and I is a slight generalization of the one used by Takahashi *et al*. in their description of this algorithm. We let r be the (rough) midpoint of a diameter-defining path in T, then let I be the nodes at the median depth of T when rooted at r. Clearly these nodes are pairwise incomparable. It follows that the subtrees of T resulting from deleting the edges of T_I each have diameter at most half that of T and thus the recursion depth is $O(\log \operatorname{diam}(T))$.

As an alternative, we include a second choice of r and I. Let r be a center node of T and I be its incident edges. Then the terminal trees of the resulting components after cutting along paths connecting each pair of terminals represented by the edges incident to r each contain at most half the nodes in T, so the recursion depth is $O(\log k)$.

Since T has k edges, $\operatorname{diam}(T)$ is at most k, and thus the recursion depth using either choice of r and I is $O(\log k)$.

What remains is to discuss the implications of contracting all degree-two articulation points. First, we note that the length of any shortest path connecting a pair of terminals in the subgraphs after the contractions remain the same as before. This allows us to determine the lengths of the paths represented in R once the algorithm terminates in only constant time as opposed to retrieving the entire path from R to determine its length.

Observe that any articulation point v must have resulted from its two incident edges uv and vw being cut along in at least two shortest paths, and so they were added to the implicit representation R and the indices stored on each edge were updated. Then the index of each terminal in the subgraph is contained in exactly one of $\left[\operatorname{idx}(v,u)^-, \operatorname{idx}(v,u)^+\right]$ or $\left[\operatorname{idx}(v,w)^-, \operatorname{idx}(v,w)^+\right]$. Suppose we change our algorithm not to contract any vertices. It follows that any shortest s_i-to-t_i path P_i to be found in a future recursive call on this subgraph that contains v must also contain edges uv and vw, and the indices stored on those edges will not need to be updated when P_i is added to R. See figure 7.3 for an example. Thus, when an edge resulting from a contraction is contained in a shortest path in some call, we simply ignore it, as the indices of the corresponding contracted edges of the input graph do not need to be updated.

Now we show that contracting the degree-two articulation points ensures that the number of vertices across all subgraphs of each level of recursion is $O(n)$. Fix a level of recursion, and let G' be a subgraph input to a call in that level. The bounded faces of the initial graph G are triangles, thus so are the bounded faces of G'. It follows that the average degree over each vertex in 2-edge-connected components containing at least three vertices is at least $7/3$. We are ensured that all vertices that compose trivial 2-edge-connected components containing only themselves must have degree at least three, so we conclude that the average degree over all vertices in G' is at least $7/3$. Euler's formula implies that the number of vertices in G' equals $(2f-2)/(\delta-2)$ where f is its number of bounded faces and δ is the average vertex degree. It follows that the number of vertices in G' is strictly less

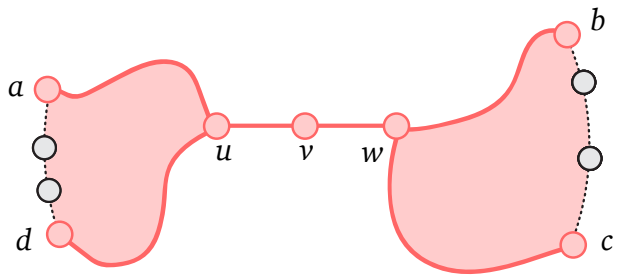

Figure 7.3: A degree-two articulation point v and its incident edges uv and vw in a subgraph G' resulting from cutting along the red a-to-b and c-to-d paths. In this example, $(v,u)^- = (w,v)^- = \text{idx}(d)$, $(v,u)^+ = (w,v)^+ = \text{idx}(a)$, $(v,w)^- = (u,v)^- = \text{idx}(b)$, and $(v,w)^+ = (u,v)^+ = \text{idx}(c)$. The indices of all terminals in G' are such that either $\text{idx}(u) \in [(v,u)^-, (v,u)^+]$ or $\text{idx}(u) \in [(v,w)^-, (v,w)^+]$.

than six times its number of bounded faces. Euler's formula also implies that the number of bounded faces in G is $O(n)$, and our cutting operation ensures that the bounded faces of the initial graph G are found in exactly one subgraph in each level of recursion. Thus, the number of vertices across all subgraphs in this level of recursion is $O(n)$.

It follows that the amount of work done in each level of recursion is $O(n)$ since r and I can be found in linear time for either choice described, and then the paths can be found and cut along for each pair of terminals in I using TSN-Star. The recursion depth is $O(\log k)$ using either method of picking r and I, and thus we conclude that this algorithm takes $O(n \log k)$ time overall. The following theorem summarizes this result.

Theorem 7.0.2. *Let G be an n-vertex undirected planar-embedded graph with a simple boundary cycle, non-negative edge lengths, and k pairs of terminals $\{(s_1,t_1),(s_2,t_2),\ldots,(s_k,t_k)\}$ specified on ∂G. Let T be the corresponding terminal tree whose edges represent the pairs of terminals. Then an implicit representation R of non-crossing shortest paths connecting all pairs of terminals can be constructed in $O(n \log \text{diam}(T)) = O(n \log k)$ time and using $O(n)$ space. The length of any path P represented in R can be reported in constant time, and the path itself can be reported in $O(|P| \log \Delta)$ time where Δ is the maximum degree of the vertices in P.*

Chapter 8

Italiano *et al.*'s $O(n \log\log k)$-time Algorithm for Path Terminal Trees

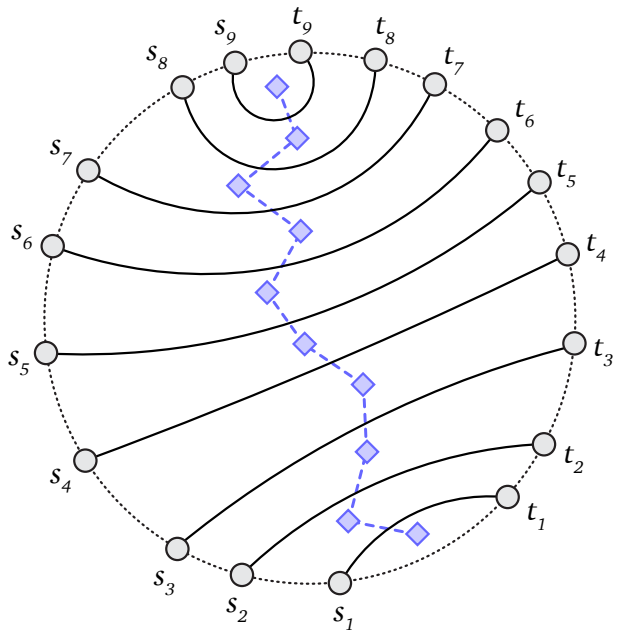

Figure 8.1: A path terminal tree (in blue) and its corresponding connection graph (in black).

In this section we give an overview of Italiano *et al.*'s algorithm [4] and slight modifications to improve the runtime from $O(n \log\log n)$ to $O(n \log\log k)$.

We begin by observing that if the terminal tree T is a path, then the terminals can be relabeled so that they appear as $s_1, s_2, \ldots, s_k, t_k, \ldots, t_2, t_1$ in clockwise order around ∂G. Additionally, since the implementation of FR-Dijkstra by Mozes *et al.* [11] requires that the maximum vertex degree in the input graph has constant degree three, we ensure this in $O(n)$ time by replacing each vertex with a binary tree gadget as we did for TSN-Arbitrary. Italiano *et al.* also assume the input graph has constant maximum vertex degree for purposes of analysis.

The main idea of their two-phase algorithm is to construct an r-division for a suitable r then find and cut along shortest paths found using FR-Dijkstra in the dense distance graph (DDG). The first phase is essentially the same as the previous divide-and-conquer algorithms but modified to work in the DDG. They ensure that when the first phase terminates, each resulting subgraph contains at most $O(r)$ pairs of terminals and the total number of vertices across all the subgraphs

is $O(n)$. By replacing their r-division construction with that of Klein *et al.* [7] and their use of Fakcharoenphol and Rao's FR-Dijkstra [8] with the implementation by Mozes *et al.* [11], we show their first phase can be done in $O(n \log\log k)$ time overall when choosing $r = \log^{10} k$. It follows that running TSN-Arbitrary in each subgraph with its corresponding terminal tree of size $O(\log^{10} k)$ takes $O(n \log\log^{10} k) = O(n \log\log k)$ time so that the entire algorithm takes only $O(n \log\log k)$ time. The majority of this section will be in describing their algorithm and emphasizing the nuances of working in the DDG. Since the global behavior is the same as iteratively finding and cutting along paths as in our $O(nk)$ simple algorithm for the general case, its correctness is implied directly from the correctness of the latter.

```
INSW-Path(G, T):
  let r = log^10 k
  construct an r-division with few holes of G
  for each multiple i of r in {1, 2, ..., k}:
      mark s_i and t_i as boundary terminals
  let T' be the path terminal tree induced by the pairs of boundary terminals
  construct the DDG H of G

  call INSW-DDG-Path(H, T')
  let P_i be the shortest s_i-to-t_i path in H found for each edge s_i t_i ∈ T'
  reveal and cut along the corresponding paths P'_i in G for each P_i
  add each path P'_i to the implicit representation R

  for each resulting subgraph G_i and path terminal tree T_i with O(r) edges:
      call TSN-Arbitrary(G_i, T_i)
```

```
INSW-DDG-Path(H, T):
  if T is a single node, terminate

  let s_i t_i be a median edge of T
  find a shortest s_i-to-t_i path P_i in H via FR-Dijkstra
  implicitly cut H along P_i into H_L and H_R
  let T_L and T_R be the corresponding path terminal trees for H_L and H_R, resp.
  call INSW-DDG-Path(H_L, T_L) and INSW-DDG-Path(H_R, T_R)
```

INSW-Path(G, T) finds and cuts along shortest paths in the given n-vertex graph G between k pairs of terminals represented by the given path terminal tree T, then adds them to a globally accessible implicit representation R. When the initial call is made, R is initialized as the empty subgraph of the input graph. INSW-DDG-Path(H, T) returns a representation of non-crossing shortest paths in the given DDG H connecting all pairs of terminals represented by a given path terminal tree T from which the corresponding paths in the underlying graph can be easily obtained.

To construct the r-division with few holes, we use the recent linear time algorithm of Klein *et al.* [7] instead of the slower construction by Italiano *et al.* [4]. The latter would take $O(n \log\log n)$

time for our choice of r, which is longer than we can afford. Constructing the DDG H of G takes $O(n \log r) = O(n \log\log k)$ time; we give the details of this in section 1.4. However, in between constructing the r-division and the DDG, we artificially mark a particular subset of terminals as boundary vertices, henceforth referred to as ***boundary terminals*** for convenience. This ensures that the number of terminals on ∂G between consecutive boundary terminals is $O(r)$, and so the terminal trees for each resulting subgraph after finding and cutting along paths between pairs of boundary terminals contain only $O(r)$ pairs of terminals. We do this by artificially labeling s_i and t_i for each i that is a multiple of r as boundary vertices if they are not already. Those that are artificially marked are contained in exactly one piece. To see that this does not invalidate the number of boundary vertices in each piece P, note each component of $P \cap \partial G$ is a line graph, and that the degree-one vertices in each component in $P \cap \partial G$ are boundary vertices. Thus, there are $O(\sqrt{r})$ components in $P \cap \partial G$. Furthermore, each component contains $O(r)$ vertices, so it follows that $O(1)$ boundary vertices are artificially marked in each component of $P \cap \partial G$, and thus $O(\sqrt{r})$ are marked in each piece overall.

What remains is to discuss how Italiano *et al*. apply the divide-and-conquer approach of finding and cutting along shortest paths to work in the DDG H instead of the underlying graph G. This part of the first phase is outlined in our pseudocode for INSW-DDG-Path. Let T' be the path terminal tree induced by the pairs of boundary terminals, and let $s_i t_i$ be a median edge of it. First, they find a shortest s_i-to-t_i path P_i in H using FR-Dijkstra that corresponds to a shortest s_i-to-t_i path P_i in G. Then they recurse on the subgraphs of H for the left and right components of $G \not\curvearrowleft P_i$ and their corresponding path terminal trees. However, explicitly rebuilding the DDGs for the components takes longer than can be afforded. Instead, they show how to implicitly represent H and its subgraphs so that they can be "cut" in time proportional to their number of vertices by maintaining cyclic orderings of the boundary vertices around the holes of each piece. Furthermore, they show how to ensure that the number of boundary vertices across the (implicitly represented) subgraphs of H in each level of recursion is $O(n/\sqrt{r})$ so that the amount of work done in any level is proportional to one FR-Dijkstra call in H. Their technique is similar to that of our contraction of degree-two articulation points in TSN-Arbitrary. When this recursive procedure terminates and non-crossing shortest paths in H between all pairs of boundary terminals have been found, they still need to obtain and cut along the corresponding paths in the underlying initial graph G. They describe how to do this while ensuring that the resulting subgraphs of G have $O(n)$ vertices overall in only $O(n)$ time. It is easy to modify their algorithm also add these paths in G to our implicit representation R. For the full details of each of these steps, we refer the reader to [4].

By replacing their use of the $O(\frac{n}{\sqrt{r}} \log^2 n)$-time implementation of FR-Dijkstra by Fakcharoenphol and Rao [8] with the recent $O(\frac{n}{\sqrt{r}} \log^4 r)$-time implementation by Mozes *et al*. [11], the total amount of work done in each level of recursion for our choice of $r = \log^{10} k$ is $O(\frac{n \log^4 \log k}{\log^5 k}) = O(n/\log k)$. There are $O(\log k)$ levels since the length of the path terminal tree is roughly halved in each call, so the overall time to complete the first phase is $O(n)$ after the $O(n \log\log k)$-time construction of the DDG H.

We conclude that the first phase takes $O(n \log\log k)$ time overall, including the time to build the r-division, construct the DDG, and then finding and cutting along shortest paths connecting the pairs of boundary terminals. Since the resulting subgraphs are ensured to have $O(n)$ total vertices, the second phase takes $O(n \log\log k)$, and thus the case where the terminal tree is a path can be done in $O(n \log\log k)$ time overall. This result is summarized in the theorem below.

Theorem 8.0.1. *Let G be an n-vertex undirected planar graph with a simple boundary cycle, non-negative edge lengths, and k pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ specified on ∂G. Let T be the corresponding terminal tree whose edges represent the pairs of terminals. If T is a path, then an implicit representation R of non-crossing shortest paths connecting all pairs of terminals can be constructed in $O(n \log\log k)$ time and using $O(n)$ space. The length of any path P represented in R can be reported in constant time, and the path itself can be reported in $O(|P| \log \Delta)$ time where Δ is the maximum degree of the vertices in P.*

Chapter 9

Our $O(n \log\log k)$-time Algorithm for Arbitrary Terminal Trees

In this section we combine the diameter-based $O(n \log \text{diam}(T))$-time algorithm by Takahashi *et al.* [1] for the general case and the $O(n \log\log k)$-time algorithm by Italiano *et al.* [4] for the path case to obtain an $O(n \log\log k)$ time algorithm for the general case. Our algorithm has two phases. The first phase identifies a subset of edges in T for which non-crossing shortest paths can be found, cut along, and added to the implicit representation so that the terminal trees corresponding to the resulting subgraphs are paths. Then the fast algorithm for the path case is used to find the remaining paths in each subgraph.

```
FASTER-ARBITRARY(G, T):
  let T_H be the heavy path tree of T rooted at an arbitrary node
  call TSN-ARBITRARY(G, T_H)
  for each resulting subgraph G_i and corresponding path terminal tree T_i:
      call INSW-PATH(G_i, T_i)
```

FASTER-ARBITRARY(G, T) finds and cuts along non-crossing shortest paths in the given n-vertex graph G between k pairs of terminals represented by the given terminal tree T, and adds them to a globally accessible implicit representation R. When the initial call is made, R is initialized as the empty subgraph of the input graph.

Specifically, we take the heavy-path decomposition of T rooted at an arbitrary node to construct its heavy path tree T_H. Let L be the light edges of T. T_H is the terminal tree induced by the edges in L since it can be obtained by contracting the edges of $T - L$ which is equivalent to deleting them from the connection graph corresponding to T. In the first phase, we find and cut along shortest paths connecting each pair of terminals represented in T_H using TSN-ARBITRARY. When the algorithm terminates, the terminal trees corresponding to each resulting subgraph are the components of $T - L$. These components are the heavy paths of T, so our second phase consists of using INSW-PATH to find and cut along shortest paths connecting all remaining pairs of terminals in their respective subgraphs. These two phases essentially enforce an order in which the paths are to be found and cut along for the simple $O(nk)$-time algorithm, and thus its correctness follows directly from the correctness of the blackboxed algorithms.

Since $\text{height}(T_H) = O(\log \text{height}(T))$, the first phase takes $O(n \log \text{diam}(T_H)) = O(n \log\log \text{diam}(T))$ $= O(n \log\log k)$ time using TSN-ARBITRARY. The algorithm ensures that the number of vertices across the resulting subgraphs is $O(n)$, and clearly each resulting path terminal tree has at most k

edges. It follows that the second phase takes $O(n \log\log k)$ total time to find the remaining paths by calling INSW-PATH. The following theorem summarizes this result.

Theorem 9.0.1. *Let G be an n-vertex undirected planar graph with a simple boundary cycle, non-negative edge lengths, and k pairs of terminals $\{(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)\}$ specified on ∂G. Let T be the corresponding terminal tree whose edges represent the pairs of terminals. Then an implicit representation R of non-crossing shortest paths connecting all pairs of terminals can be constructed in $O(n \log \operatorname{diam} T) = O(n \log\log k)$ time and using $O(n)$ space. The length of any path P represented in R can be reported in constant time, and the path itself can be reported in $O(|P| \log \Delta)$ time where Δ is the maximum degree of the vertices in P.*

References

[1] J. Takahashi, H. Suzuki, and T. Nishizeki, "Shortest noncrossing paths in plane graphs," *Algorithmica*, vol. 16, no. 3, pp. 339–357, 1996.

[2] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian, "Faster shortest-path algorithms for planar graphs," *Journal of computer and system sciences*, vol. 55, no. 1, pp. 3–23, 1997.

[3] J. H. Reif, "Minimum s-t cut of a planar undirected network in $O(n \log^2(n))$ time," *SIAM Journal on Computing*, vol. 12, no. 1, pp. 71–81, 1983.

[4] G. F. Italiano, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen, "Improved algorithms for min cut and max flow in undirected planar graphs," in *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, 2011, pp. 313–322.

[5] J. Erickson and A. Nayyeri, "Shortest non-crossing walks in the plane," in *Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2011, pp. 297–308.

[6] S. Cabello and B. Mohar, "Finding shortest non-separating and non-contractible cycles for topologically embedded graphs," *Discrete & Computational Geometry*, vol. 37, no. 2, pp. 213–235, 2007.

[7] P. N. Klein, S. Mozes, and C. Sommer, "Structured recursive separator decompositions for planar graphs in linear time," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. ACM, 2013, pp. 505–514.

[8] J. Fakcharoenphol and S. Rao, "Planar graphs, negative weight edges, shortest paths, and near linear time," *Journal of computer and system sciences*, vol. 72, no. 5, pp. 868–889, 2006.

[9] P. N. Klein, "Multiple-source shortest paths in planar graphs," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 146–155.

[10] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[11] S. Mozes, Y. Nussbaum, and O. Weimann, "Faster shortest paths in dense distance graphs, with applications," *CoRR*, vol. abs/1404.0977, 2014. [Online]. Available: http://arxiv.org/abs/1404.0977

[12] D. D. Sleator and R. E. Tarjan, "A data structure for dynamic trees," *Journal of computer and system sciences*, vol. 26, no. 3, pp. 362–391, jun 1983.

[13] B. Mohar and C. Thomassen, *Graphs on Surfaces*. The Johns Hopkins University Press, 2001, vol. 2.

[14] V. Polishchuk and J. S. Mitchell, "Thick non-crossing paths and minimum-cost flows in polygonal domains," in *Proceedings of the twenty-third annual symposium on Computational geometry*. ACM, 2007, pp. 56–65.