

# Multiple-Source Shortest Paths in Embedded Graphs\*

Sergio Cabello<sup>†</sup>

Erin W. Chambers<sup>‡</sup>

Jeff Erickson<sup>§</sup>

February 1, 2012

## Abstract

Let  $G$  be a directed graph with  $n$  vertices and non-negative weights in its directed edges, embedded on a surface of genus  $g$ , and let  $f$  be an arbitrary face of  $G$ . We describe an algorithm to preprocess the graph in  $O(gn \log n)$  time, so that the shortest-path distance from any vertex on the boundary of  $f$  to any other vertex in  $G$  can be retrieved in  $O(\log n)$  time. Our result directly generalizes the  $O(n \log n)$ -time algorithm of Klein [Multiple-source shortest paths in planar graphs. In *Proc. 16th Ann. ACM-SIAM Symp. Discrete Algorithms*, 2005] for multiple-source shortest paths in planar graphs. Intuitively, our preprocessing algorithm maintains a shortest-path tree as its source point moves *continuously* around the boundary of  $f$ . As an application of our algorithm, we describe algorithms to compute a shortest non-contractible or non-separating cycle in embedded, undirected graphs in  $O(g^2n \log n)$  time.

**Keywords:** computational topology, topological graph theory, parametric shortest paths, dynamic data structures

---

\*A preliminary version of this work, without the third author's contributions, was presented at the 18th Annual ACM-SIAM Symposium on Discrete Algorithms [8].

<sup>†</sup>Department of Mathematics, IMFM, and Department of Mathematics, FMF, University of Ljubljana, Slovenia, [sergio.cabello@fmf.uni-lj.si](mailto:sergio.cabello@fmf.uni-lj.si). Research partially supported by the European Community Sixth Framework Programme under a Marie Curie Intra-European Fellowship, and by the Slovenian Research Agency, projects P1-0297, J1-7218 and J1-4106.

<sup>‡</sup>Department of Mathematics and Computer Science, Saint Louis University, [echambe5@slu.edu](mailto:echambe5@slu.edu). Portions of this work were done while the author was affiliated with the University of Illinois, Urbana-Champaign. Research partially supported by an NSF Graduate Research Fellowship and by NSF grants DMS-0528086 and CCF-1054779.

<sup>§</sup>Department of Computer Science, University of Illinois, Urbana-Champaign, [jeffe@cs.uiuc.edu](mailto:jeffe@cs.uiuc.edu). Research partially supported by NSF grants DMS-0528086 and CCF-0915519.

# 1 Introduction

Let  $G$  be a directed graph with  $n$  vertices and non-negative weights in the directed edges, embedded on a surface of genus  $g$ , and let  $f$  be an arbitrary face of  $G$ . In this paper, we describe an algorithm to preprocess the graph in  $O(gn \log n)$  time and space, so that later the shortest-path distance from any vertex on the boundary of  $f$  to any other vertex in  $G$  can be retrieved in  $O(\log n)$  time. Our preprocessing algorithm constructs an implicit representation of all shortest path trees rooted at vertices of  $f$ . Storing all these shortest path trees explicitly would require  $\Theta(n^2)$  space in the worst case.

Euler's formula implies that any graph with  $n$  vertices embedded in a surface of genus  $g$  has  $O(g+n)$  edges. Our problem can be solved by solving the all-pairs shortest path problem in  $G$  in  $O(gn + n^2 \log n)$  time using Dijkstra's algorithm [15] with Fibonacci heaps [29]. When  $g = O(1)$ , the running time can be reduced to  $O(n^2)$  using the faster shortest-path algorithm of Henzinger *et al.* [36]; see also [53]. Our algorithm improves both of these running times if and only if  $g = o(n)$ . Thus, for the rest of the paper, we assume  $g = o(n)$ , which implies that  $G$  has  $O(n)$  directed edges.

Our results generalize (and were inspired by) an algorithm of Klein [42] that solves the corresponding multiple-source shortest path problem in *planar* graphs in  $O(n \log n)$  time. Other noteworthy results on multiple-source shortest paths for planar graphs include Frederickson's all-pairs shortest paths representation [28], Lipton and Tarjan's planar separator theorem [44], and Schmidt's  $O(n \log n)$  algorithm that supports distance queries for specific subsets of vertices on a grid [47].

Let  $v_0, v_1, \dots, v_k$  be the sequence of vertices around the specified face  $f$ . Klein's algorithm begins with a shortest path tree  $T$  rooted at  $v_0$  and then proceeds in  $k$  phases; at the end of the  $i$ th phase,  $T$  is the shortest path rooted at  $v_i$ . At the beginning of the  $i$ th phase, the algorithm deletes the directed edge in  $T$  leading into  $v_i$ , adds the directed edge from  $v_i$  to  $v_{i-1}$ , and declares  $v_i$  to be the new root. The algorithm then transforms  $T$  back into a shortest path tree through a series of *pivot* operations; each pivot changes the predecessor of some vertex, replacing its previous incoming edge with a new directed edge, just as in the classical shortest-path algorithms of Dijkstra [15] and Bellman and Ford [26]. Specifically, at each iteration, Klein's algorithm pivots the *leafmost unrelaxed directed edge* into  $T$ . This characterization exploits the classical observation, originally by von Staudt [56], that for any spanning tree  $T$  of any planar graph  $G$ , the edges not in  $T$  comprise a spanning tree of the dual graph  $G^*$ . Klein describes how to find the leafmost unrelaxed directed edge and pivot it into  $T$  in  $O(\log n)$  time, using a dynamic tree data structure [49, 52]. He also proves that each directed edge pivots into  $T$  at most a constant number of times; the  $O(n \log n)$  running time follows immediately. To allow for future shortest-path queries, Klein maintains  $T$  in a *persistent* data structure [16].

The main obstacle to extend Klein's algorithm to higher-genus graphs is that the complement of a spanning tree is no longer a dual spanning tree, so there is no 'leafmost' unrelaxed directed edge. Our algorithm follows Klein's basic approach, but uses a different strategy for choosing which directed edges to pivot. Conceptually, we move the root of  $T$  *continuously* around the boundary of  $f$  and maintain the correct shortest path tree at all times. Even though the source vertex moves continuously, the combinatorial structure of the shortest path tree changes only at certain critical events, when one or more directed edges pivot into the tree. Our approach can be seen both as an example of the *kinetic data structure* paradigm proposed by Basch, Guibas, and Hershberger [2, 33] and as a special case of the *parametric shortest path* problem introduced by Karp and Orlin [38, 57].

A key observation in our analysis is that if we move the source vertex along a single edge of  $G$ , any directed edge of  $G$  pivots into the shortest-path tree at most once; thus, the number of pivots is equal to the symmetric difference between the initial and final shortest path trees. Moreover, as the source vertex moves around the boundary of  $f$ , each directed edge pivots into  $T$  at most  $O(g)$  times. Our algorithm finds and executes each pivot in  $O(g \log n)$  time, using a complex data structure composed of  $O(g)$  dynamic trees [49, 52]. These two bounds immediately imply that we can move the source completely

around  $f$  in  $O(g^2 n \log n)$  time; a more refined approach improves this time bound to  $O(g n \log n)$ . We describe our algorithm in the simpler setting of planar graphs in Section 3 and in full generality in Section 4.

In Section 5, as applications of our shortest-path algorithm, we describe algorithms to compute a shortest non-contractible cycle and a shortest non-separating cycle in a combinatorial surface in  $O(g^2 n \log n)$  time. Thomassen developed the first algorithm to compute shortest non-contractible or non-separating cycles, by exploiting the so-called *3-path condition* in  $O(n^3)$  time [55]; see also Mohar and Thomassen [45, Sect. 4.3]. Erickson and Har-Peled described a faster algorithm that runs in  $O(n^2 \log n)$  time [22]. Cabello and Mohar [10] gave an algorithm which runs in time  $g^{O(g)} n^{3/2} \log n$ , the first algorithm for this problem to have a running time which was bounded by a function of the genus. Cabello [7] later improved the running time using separators to  $g^{O(g)} n^{4/3}$ . Finally, for *orientable* surfaces Kutz [43] developed an algorithm with a running time of  $g^{O(g)} n \log n$  which computed a finite portion of the universal cover and then did standard shortest path computations in the resulting planar graph. Our algorithm has a better dependence on the genus and, for nonorientable surfaces of bounded genus, it is the first one to achieve a running time of  $O(n \log n)$ . Since the preliminary version [8] of this work appeared, the results have been used in [3, 5, 21, 23, 24, 25, 27, 37, 39].

## 2 Background

### 2.1 Surfaces, Embeddings, and Duality

**Surfaces.** A *surface* (alternatively, a *topological 2-manifold with boundary*) is a Hausdorff topological space in which every point has an open neighborhood homeomorphic to either the plane  $\mathbb{R}^2$  or the closed upper half-plane. The points that do not have neighborhoods homeomorphic to  $\mathbb{R}^2$  constitute the *boundary* of the surface. A *cycle* in a surface  $\Sigma$  is (the image of) a continuous map  $\gamma: S^1 \rightarrow \Sigma$ ; a cycle is *simple* if this map is injective. The *genus* of a surface  $\Sigma$  is the maximum number of simple disjoint cycles  $\gamma_1, \gamma_2, \dots, \gamma_g$  such that the subspace  $\Sigma \setminus (\gamma_1 \cup \dots \cup \gamma_g)$  is connected. A surface is *orientable* if it contains no subspace homeomorphic to the Möbius band. We consider only compact, connected surfaces in this paper; up to homeomorphism, such surfaces are uniquely determined by genus, orientability, and the number of boundaries. When dealing with an orientable surface (which will be the majority of the time), we also fix an orientation on the surface, so that terms like ‘left’, ‘right’, ‘clockwise’, and ‘counterclockwise’ are well-defined.

**Curves and Homotopy.** A *path* is (the image of) a continuous map  $p: [0, 1] \rightarrow \Sigma$ ; it is *simple* if this map is injective. The *endpoints* of a path  $p$  are the points  $p(0)$  and  $p(1)$ . A *loop* is a path  $p$  with  $p(0) = p(1)$ . An *arc* is a path whose endpoints are on the boundary of  $\Sigma$ . The term *curve* refers to a cycle, path, or arc. If two paths  $p$  and  $q$  satisfy  $p(1) = q(0)$ , their *concatenation*  $p \cdot q$  is the path defined by  $(p \cdot q)(t) = p(t/2)$ , if  $t \leq 1/2$ , and  $(p \cdot q)(t) = q(2t - 1)$ , if  $t > 1/2$ .

A *homotopy* between two paths  $p$  and  $q$  is a continuous map  $H: [0, 1] \times [0, 1] \rightarrow \Sigma$  such that  $H(0, \cdot) = p$ ,  $H(1, \cdot) = q$ ,  $H(\cdot, 0) = p(0) = q(0)$ , and  $H(\cdot, 1) = p(1) = q(1)$ . A homotopy between two arcs  $\alpha$  and  $\beta$  is a continuous map  $H: [0, 1] \times [0, 1] \rightarrow \Sigma$  such that  $H(0, \cdot) = \alpha$ ,  $H(1, \cdot) = \beta$ ,  $H(\cdot, 0)$  is always in the same boundary component, and  $H(\cdot, 1)$  is always in the same boundary component. A homotopy between two cycles  $\gamma$  and  $\beta$  is a continuous map  $H: [0, 1] \times S^1 \rightarrow \Sigma$  such that  $H(0, \cdot) = \gamma$  and  $H(1, \cdot) = \beta$ . Two curves  $\beta, \gamma$  are *homotopic* when there is some homotopy between them, and we denote it by  $\beta \sim \gamma$ . Being homotopic is an equivalence relation. Up to homotopy, each boundary component of  $\Sigma$  admits two parameterizations as simple cycles, one being the reverse of the other. We say that a cycle is homotopic to a boundary component of  $\Sigma$  when it is homotopic to any of these two parameterizations. If  $\gamma$  is a

simple cycle homotopic to a boundary  $\delta$  of  $\Sigma$ , and  $\gamma$  and  $\delta$  are disjoint, then  $\Sigma \setminus \gamma$  has two connected components, one of them a topological annulus with  $\gamma$  and  $\delta$  as boundaries.

A curve is **contractible** if it is homotopic to a constant map. In particular, a contractible arc must have its endpoints in the same boundary component  $\delta$  and, after contracting  $\delta$  to a point, the obtained loop must be contractible. A simple curve is **separating** if  $\Sigma \setminus \gamma$  has two connected components. (This concept is related to that of  $\mathbb{Z}_2$ -homology, but we will not use homology in this paper.) A simple cycle  $\gamma$  is contractible if and only if it bounds a disk, that is,  $\Sigma \setminus \gamma$  has two connected components, one of them being a topological disk.

**Darts and Embeddings.** Let  $G = (V, \vec{E})$  be a directed graph. Following Borradaile and Klein [4, 6], we refer to the directed edges in  $\vec{E}$  as the **darts** of  $G$ . (The term *arc* has been used for something else above.) Each dart connects two vertices, called its **tail** and its **head**; we will write  $u \rightarrow v$  to denote a dart with tail  $u$  and head  $v$ . Each dart  $u \rightarrow v$  has a unique **reversal**, defined by swapping its endpoints; thus, the reversal of  $u \rightarrow v$  is  $v \rightarrow u$ . We will assume that whenever  $G$  contains a dart  $u \rightarrow v$ , then it also contains its reversal  $v \rightarrow u$ . This can be enforced adding any missing darts with a large enough weight, like for example twice the sum of the weights of all original darts. (In our presentation we need the weights to be finite.)

Each dart  $u \rightarrow v$  defines an associated (undirected) edge  $uv$ , which does not carry any orientation. An edge  $uv$  represents the pair of darts  $\{u \rightarrow v, v \rightarrow u\}$ . For any directed graph  $G = (V, \vec{E})$ , we define its associated undirected graph  $G_{\text{un}} = (V, E)$  by replacing each dart with the associated edge. In most cases, there is no need to distinguish between  $G$  and  $G_{\text{un}}$  and we will use  $G$  to denote any of them.

Informally, an **embedding** of a directed graph  $G$  on a surface  $\Sigma$  is an embedding of its associated undirected graph: vertices are mapped to distinct points and edges are mapped to simple paths on  $\Sigma$  that connect the corresponding vertices. The darts  $u \rightarrow v$  and  $v \rightarrow u$  are embedded using the path of  $uv$ , but with different orientation. A **face** of an embedding is a maximal connected subset of  $\Sigma$  that does not intersect the image of any edge or vertex. An embedding is **cellular** (or *2-cell* [45]) if every face is an open topological disk; in particular, if the surface has boundary, every boundary cycle must be covered by edges of the graph. Any cellular embedding can be represented combinatorially by a rotation system and a signature. A **rotation system** is a permutation  $\pi$  of the darts of  $G$ , where  $\pi(\vec{e})$  is the dart that appears immediately after  $\vec{e}$  in the circular ordering of darts leaving  $\text{tail}(\vec{e})$ . A **signature** is a mapping  $i: E(G_{\text{un}}) \rightarrow \{0, 1\}$  assigning a bit to each edge that indicates whether the cyclic ordering at its endpoints are in the “same” direction or the “opposite” direction.

When the surface is orientable, the signature is not needed, and we can assume that the rotation system gives a counterclockwise ordering of the darts emanating from each vertex. Every dart in an embedded graph  $G$  separates two (possibly equal) faces. For orientable surfaces we may talk of the **left shore** and **right shore**, respectively denoted  $\text{left}(\vec{e})$  and  $\text{right}(\vec{e})$ . Reversing any dart swaps its shores:  $\text{left}(v \rightarrow u) = \text{right}(u \rightarrow v)$  and  $\text{right}(v \rightarrow u) = \text{left}(u \rightarrow v)$ .

If  $G$  is embedded on an orientable surface of genus  $g$ , Euler’s formula  $|V| - |E| + |F| = 2 - 2g$  implies that  $G$  has at most  $3n - 6 + 6g$  edges and at most  $2n - 4 + 4g$  faces, with equality if every face of the embedding is a triangle. If the surface is non-orientable, then Euler’s formula  $|V| - |E| + |F| = 2 - g$  implies that  $G$  has at most  $3n - 6 + 3g$  edges and at most  $2n - 4 + 2g$  faces. We consider only graphs and surfaces with  $g = o(n)$ , so in either case the overall complexity of any embedding is  $O(n)$ .

Our algorithms require an explicit cellular embedding as part of the input. If no embedding is provided, it is easy to construct an embedding of any given graph on *some* surface, but the genus of this surface may be  $\Omega(n)$  even when the minimum possible genus is constant [12]. Determining whether a given graph  $G$  can be embedded on a surface of genus  $g$  is NP-complete [54], although there is an

embedding algorithm that runs in  $g^{O(g)}n \log n$  time [40]. Moreover, for any fixed  $\varepsilon > 0$ , approximating the genus of an  $n$ -vertex graph within a factor of  $O(n^{1-\varepsilon})$  is NP-hard [12].

**Combinatorial Surfaces.** A *combinatorial surface*  $M$  is a surface  $\Sigma$  together with an *undirected* multigraph  $G$  with non-negative edge weights embedded cellularly on  $\Sigma$ . This concept, which we will use in Section 5, was introduced by Colin de Verdière [13] and it is equivalent to cellular embeddings of graphs. The combinatorial surface model is dual to the cross-metric surface model; see [14] for more discussion on this. The *complexity* of a combinatorial surface is the size of the multigraph that describes it.

In combinatorial surfaces, all curves are restricted to lie on  $G$ , possibly with repeated edges or vertices. Thus, a curve is a walk in  $G$ . We say a curve is *simple* if it can be infinitesimally perturbed off of  $G$  to be simple in  $\Sigma$ ; note that this allows a simple curve to follow the same edge multiple times. Similarly, two curves are *non-crossing* if they can be infinitesimally perturbed to non-crossing curves. The *multiplicity* of a curve is the maximum number of times that the same edge appears in the curve. The *length* of a curve  $\gamma$ , denoted by  $|\gamma|$ , is the sum of the edge weights of its edges, where each edge weight is counted as many times as the corresponding edge appears in the curve.

**Duality.** The *dual* of a surface-embedded graph  $G$  is another graph  $G^*$  embedded on the same surface, whose vertices correspond to faces of  $G$  and vice versa. Two vertices in  $G^*$  are joined by an edge if and only if the corresponding faces of  $G$  are separated by an edge of  $G$ ; thus, every edge  $e$  in  $G$  has a corresponding dual edge  $e^*$  in  $G^*$ . Similarly, every vertex  $v$  of  $G$  corresponds to a face  $v^*$  of  $G^*$ , and every face  $f$  of  $G$  corresponds to a vertex  $f^*$  of  $G^*$ . Each dual edge  $e^*$  is embedded so that it crosses the corresponding primal edge  $e$  exactly once and intersects no other primal edge. Duality is an involution—the dual of  $G^*$  is isomorphic to the original graph  $G$ . We will use duality only in orientable surfaces. In this case we will use a duality for darts defined by  $\vec{e}^* := (\text{right}(\vec{e}))^* \rightarrow (\text{left}(\vec{e}))^*$ . (For darts this is not an involution because  $((u \rightarrow v)^*)^* = v \rightarrow u$ .)

For any subgraph  $H$  of  $G$ , let  $H^*$  denote the corresponding subgraph of  $G^*$ .

To simplify notation, we will consistently use the letters  $s, u, v, w, x, y, z$  to denote vertices in the primal graph  $G$ , and the letters  $a, b, c, d$  to denote vertices in the dual graph  $G^*$ . Thus,  $a^*$  will always denote a face of  $G$ . (However,  $e$  is always an edge of  $G$ ,  $f$  is always a face of  $G$ , and  $g$  is always the genus of the underlying surface.)

**Tree and Cotrees.** Let  $G$  be an undirected graph embedded on a surface of genus  $g$  without boundary. A *spanning tree* of  $G$  is just a tree formed by some subset of the edges of  $G$  that includes every vertex of  $G$ . If  $C^*$  is a tree contained in the dual graph  $G^*$ , we call the corresponding primal subgraph  $C$  a *cotree* of  $G$ ; if moreover  $C^*$  is a spanning tree of  $G^*$ , we call  $C$  a *spanning cotree*. A *tree-cotree decomposition* of  $G$  is a partition of  $G$  into three edge-disjoint subgraphs: a spanning tree  $T$ , a spanning cotree  $C$ , and the leftover edges  $L = G \setminus (T \cup C)$  [18]. For any spanning tree  $T$ , we can complete a tree-cotree decomposition by choosing any spanning tree  $C^*$  of the dual subgraph  $(G \setminus T)^*$ . Euler's formula implies that  $|L| = 2g$ ; in particular, when the underlying surface is the sphere,  $L$  is empty.

## 2.2 Shortest-Path Trees

**Definitions.** Let  $G = (V, \vec{E})$  be a directed graph, and let  $w: \vec{E} \rightarrow \mathbb{R}^+$  be a non-negative weight function on the edges. To shorten some expressions we define for every edge  $uv$  the weight

$$\widehat{w}(uv) := w(u \rightarrow v) + w(v \rightarrow u).$$

A **shortest-path tree** is a spanning tree of  $G$  that contains shortest paths from a **source** vertex  $s$  to every other vertex of  $G$ . Shortest-path trees are typically represented by storing two values at every vertex:  $\mathit{dist}(v)$  is the shortest-path distance from  $s$  to  $v$ , and  $\mathit{pred}(v)$  is the predecessor of  $v$  in the shortest path from  $s$  to  $v$ , or equivalently, the parent of  $v$  if we regard the tree as rooted at  $s$ . In particular,  $\mathit{dist}(s) = 0$  and  $\mathit{pred}(s) = \text{NULL}$ . We define the **slack** of any dart  $u \rightarrow v$  as follows:

$$\mathit{slack}(u \rightarrow v) := \mathit{dist}(u) + w(u \rightarrow v) - \mathit{dist}(v).$$

So for any edge  $uv$ , we have

$$\mathit{slack}(v \rightarrow u) + \mathit{slack}(u \rightarrow v) = w(u \rightarrow v) + w(v \rightarrow u) = \widehat{w}(uv).$$

A dart is **tense** if its slack is negative, and an undirected edge is tense if either of its darts is tense. A classical result of Ford [26], exploited by almost all shortest-path algorithms, states that a collection of distances and predecessors describes a shortest-path tree if and only if there are no tense edges,  $\mathit{dist}(s) = 0$ , and  $\mathit{slack}(\mathit{pred}(v) \rightarrow v) = 0$  for every vertex  $v \neq s$ .

It is helpful to view the graph  $G$  as a continuous space, so that distances and shortest paths between points in the interior of edges are also well-defined. Specifically, each edge  $uv$  is homeomorphic to an interval  $[0, 1]$ ; each value  $\lambda \in [0, 1]$  represents a point  $s_\lambda$  on  $uv$  such that the distance from  $s_\lambda$  to  $u$  is  $\lambda w(v \rightarrow u)$ , the distance from  $u$  to  $s_\lambda$  is  $\lambda w(u \rightarrow v)$ , the distance from  $s_\lambda$  to  $v$  is  $(1 - \lambda)w(u \rightarrow v)$ , and the distance from  $v$  to  $s_\lambda$  is  $(1 - \lambda)w(v \rightarrow u)$ . In particular,  $s_0 = u$  and  $s_1 = v$ .

For simplicity we will assume that shortest paths are unique, and the union of all shortest paths from a common source vertex forms a tree. This assumption may be enforced by adding random infinitesimal weights to each edge; using the isolation lemma of [46], this yields shortest paths with high probability.

**Parametric Shortest Paths.** Our algorithm can be viewed as a special case of the *parametric shortest path* problem introduced by Karp and Orlin [38]; see also [19, 20, 32, 57]. Suppose we are given a graph whose dart weights are linear functions of a parameter  $\lambda$ ; specifically, let  $w_\lambda(\vec{e}) = w(\vec{e}) + \lambda \cdot w'(\vec{e})$ , for given functions  $w, w' : \vec{E} \rightarrow \mathbb{R}$ . (In Karp and Orlin's original formulation,  $w'(\vec{e}) \in \{0, 1\}$  for every dart  $\vec{e}$ .) Let  $T_\lambda$  denote the shortest-path tree rooted at some fixed source vertex  $s$  with respect to the weights  $w_\lambda$ . The parametric shortest-path problem asks us to compute  $T_\lambda$  for all  $\lambda$  in a certain range.

To solve this problem, Karp and Orlin [38] propose maintaining  $T_\lambda$  while *continuously* increasing the parameter  $\lambda$ . Although the shortest-path distances vary continuously as a function of  $\lambda$ , the combinatorial structure of  $T_\lambda$  changes only at certain *critical values* of  $\lambda$ . A critical value occurs when the slack of some non-tree dart  $y \rightarrow z$  decreases to zero; thus,  $y \rightarrow z$  is just about to become tense. At this critical value, the dart  $y \rightarrow z$  **pivots** into the shortest-path tree, replacing some other dart  $x \rightarrow z$ ; in other words,  $y$  becomes the new predecessor of  $z$ .<sup>1</sup>

The running time of any parametric shortest-path algorithm clearly depends on two factors: (1) the amortized time required to identify the next tense dart and pivot it into the tree and (2) the number of pivots. We also assume genericity in the sense that at most one pivot occurs at a time, which is again enforced by our assumption of unique shortest paths using infinitesimal perturbations and the isolation lemma [46].

Karp and Orlin's parametric shortest-path algorithm is an early prototypical example of the *kinetic data structure* paradigm of Basch, Guibas, and Hershberger [2, 33]. Their algorithms (and ours) can also be interpreted as a special case of the parametric objective simplex algorithm of Gass and Saaty [30, 41]; similar approaches have been applied to several other classical parametric optimization problems [1, 17, 34].

---

<sup>1</sup>If  $z$  is an ancestor of  $y$ , this pivot introduces a cycle into  $T_\lambda$ ; for all larger values of  $\lambda$ , the total weight of this cycle is negative, and the shortest-path tree  $T_\lambda$  is not well-defined. In our algorithm, darts will *always* have non-negative weight, so this condition never arises.

## 2.3 Dynamic Forest Data Structures

Our algorithms require dynamic forest data structures that implicitly maintain dart or vertex values under edge insertions, edge deletions, and updates to the values in certain substructures. We require two different data structures, which we call the *primal tree structure* and the *dual forest structure*, for reasons that will become clear later.

**Primal tree structure.** Our first data structure maintains a dynamic rooted forest with real values associated with the vertices; in our application, these values are shortest-path distances. The data structure supports the following operations:

- **CREATE( $val$ ):** Return a new tree containing a single vertex with value  $val$ .
- **CUT( $e$ ):** Remove the edge  $e$  from the tree  $T$  that contains it. The subtree that contains the root of  $T$  keeps that node as its root; the other subtree takes as root the endpoint of  $e$  that it contains.
- **LINK( $u, v$ ):** Add the edge  $uv$  to the forest; the root of the subtree containing  $u$  becomes the root of the new larger tree. This operation assumes that  $u$  and  $v$  are initially in different trees.
- **GETNODEVALUE( $v$ ):** Return the value associated with vertex  $v$ .
- **ADDSUBTREE( $\Delta, v$ ):** Add the real value  $\Delta$  to the value of every vertex in the subtree rooted at  $v$ .

We emphasize that the operations  $\text{LINK}(u, v)$  and  $\text{LINK}(v, u)$  give rise to the same undirected tree, but with different roots. Several dynamic forest data structures, including Euler-tour trees [35, 51] and self-adjusting top trees [52], support each of these operations in  $O(\log n)$  amortized time, where  $n$  is the number of vertices in the forest.

**Dual forest structure.** Our second data structure maintains a dynamic undirected, unrooted forest, with two values  $val(u \rightarrow v)$  and  $val(v \rightarrow u)$  associated with every edge  $uv$ ; that is, we associate separate values with each *dart* of the forest. In our application, we will have an orientable surface, each tree in the dynamic forest is a subgraph of the dual graph  $G^*$ , and the value associated with any dart in the forest is equal to the slack of the corresponding primal dart. The data structure supports the following operations:

- **CREATE():** Return a new one-vertex tree.
- **CUT( $uv$ ):** Remove the edge  $uv$  from the forest.
- **LINK( $u \rightarrow v, \alpha, \beta$ ):** Add the edge  $uv$  to the forest and set  $val(u \rightarrow v) = \alpha$  and  $val(v \rightarrow u) = \beta$ . This operation assumes that  $u$  and  $v$  are initially in different trees. The operations  $\text{LINK}(u, v, \alpha, \beta)$  and  $\text{LINK}(v, u, \beta, \alpha)$  yield identical results.
- **GETDARTVALUE( $u \rightarrow v$ ):** Returns the value  $val(u \rightarrow v)$ . This operation assumes that  $uv$  is an edge in the forest.
- **ADDPATH( $\Delta, u, v$ ):** For each dart  $x \rightarrow y$  on the (unique) directed path from  $u$  to  $v$ , add  $\Delta$  to  $val(x \rightarrow y)$  and subtract  $\Delta$  from  $val(y \rightarrow x)$ . This operation assumes that  $u$  and  $v$  lie in the same tree.
- **MINPATH( $u, v$ ):** Return a dart  $x \rightarrow y$  on the (unique) directed path from  $u$  to  $v$ , such that  $val(x \rightarrow y)$  is minimized. This operation assumes that  $u$  and  $v$  lie in the same tree.

- **JUNCTION( $t, u, v$ )**: Return the unique node that lies on the paths from  $t$  to  $u$ , from  $u$  to  $v$ , and from  $v$  to  $t$  in the forest. This operation assumes that the nodes  $t$ ,  $u$ , and  $v$  lie in the same component of the forest.

Several dynamic forest data structures, including link-cut trees [49] and self-adjusting top trees [52], can be adapted to support each of these operations in  $O(\log n)$  amortized time. In their original formulations, these data structures maintain only a single value at each undirected edge. However, Goldberg, Grigoriadis, and Tarjan [31, 50] describe the necessary modifications to link-cut trees [49] to support values on directed edges, specifically for use in network-simplex algorithms, of which our algorithm is an example. Similar modifications can be applied to any other dynamic tree structure.

The only operation we require that is not part of the standard dynamic-tree literature is **JUNCTION**. Here we describe how to implement this operation using link-cut trees; similar methods can be used with more recent dynamic tree data structures. Internally, link-cut trees actually represent each component of the forest as a *rooted* tree. Link-cut trees support two additional operations:

- **EVERT( $v$ )**: Make vertex  $v$  the root of the tree that contains it.
- **LCA( $u, v$ )**: Return the least common ancestor of vertices  $u$  and  $v$ . This operation assumes  $u$  and  $v$  lie in the same tree [49, p. 387].

We can implement **JUNCTION( $t, u, v$ )** simply by calling **EVERT( $t$ )** and then returning **LCA( $u, v$ )**.

### 3 Algorithm for Planar Graphs

Before describing our algorithm in full generality, we first consider the special case  $g = 0$ . Given a planar graph  $G$  and one of its faces  $f$ , our algorithm computes an implicit representation of the shortest path tree rooted at every vertex on the boundary of  $f$ .

Our high-level approach is similar to Klein’s algorithm [42]. We begin by computing a shortest-path tree  $T$  rooted at an arbitrary vertex on the boundary of  $f$ . We maintain  $T$  and the complementary dual spanning tree  $C^* = (G \setminus T)^*$  in appropriate dynamic forest data structures. Then, for each edge  $uv$  in order around the boundary of  $f$ , we modify the shortest path tree rooted at  $u$  until it becomes the shortest path tree rooted at  $v$ . However, our algorithm uses a different pivoting strategy to update the tree. Klein’s algorithm moves the source directly from  $u$  to  $v$  and then performs a sequence of pivots to repair the tree; our algorithm maintains a parametric shortest-path tree as the source point moves *continuously* from  $u$  to  $v$ .

#### 3.1 Moving Along an Edge

Consider a single edge  $uv$  in  $G$ . Suppose we have already computed the shortest path tree  $T_u$  rooted at  $u$ . We transform  $T_u$  into the shortest-path tree  $T_v$  rooted at  $v$  using a parametric shortest path algorithm as follows. First, we insert a new vertex  $s$  in the interior of  $uv$ , bisecting it into two edges  $su$  and  $sv$  with the parametric weights

$$\begin{aligned} w_\lambda(u \rightarrow s) &:= \lambda w(u \rightarrow v), & w_\lambda(s \rightarrow v) &:= (1 - \lambda)w(u \rightarrow v). \\ w_\lambda(s \rightarrow u) &:= \lambda w(v \rightarrow u), & w_\lambda(v \rightarrow s) &:= (1 - \lambda)w(v \rightarrow u). \end{aligned}$$

Every other dart  $x \rightarrow y$  has constant parametric weight  $w_\lambda(x \rightarrow y) := w(x \rightarrow y)$ . We then maintain the shortest-path tree  $T_\lambda$  rooted at  $s$ , with respect to the weight function  $w_\lambda$ , as the parameter  $\lambda$  increases

continuously from 0 to 1. The initial shortest-path tree  $T_0$  is equal to  $T_u$ , and the final tree  $T_1$  is equal to  $T_v$ .

For any vertex  $x$  and any parameter value  $\lambda \in [0, 1]$ , let  $\mathbf{dist}_\lambda(\mathbf{x})$  denote the distance from  $s$  to  $x$  with respect to the weight function  $w_\lambda$ . We color each vertex  $x$  **red** if  $\mathbf{dist}_\lambda(x)$  is an increasing function of  $\lambda$ , or **blue** if  $\mathbf{dist}_\lambda(x)$  is a decreasing function of  $\lambda$ .<sup>2</sup> Every vertex except  $s$  is either red or blue. If  $su$  is an edge in  $T_\lambda$ , then every vertex in the subtree rooted at  $u$  is red; symmetrically, if  $sv$  is an edge in  $T_\lambda$ , then every vertex in the subtree rooted at  $v$  is blue.  $T_0$  may contain only red vertices, and  $T_1$  may contain only blue vertices, but there is always an interval of values of  $\lambda$  such that  $T_\lambda$  contains vertices of both colors. For example it holds  $w_\lambda(s \rightarrow u) = w_\lambda(s \rightarrow v)$  when  $\lambda = w(u \rightarrow v) / \widehat{w}(uv)$ . We color an edge of  $G$  **green** if it has one red endpoint and one blue endpoint.

Similarly, let  $\mathbf{slack}_\lambda(\mathbf{x} \rightarrow \mathbf{y})$  denote the slack of  $x \rightarrow y$  with respect to the weight function  $w_\lambda$ :

$$\mathbf{slack}_\lambda(x \rightarrow y) := \mathbf{dist}_\lambda(x) + w_\lambda(x \rightarrow y) - \mathbf{dist}_\lambda(y).$$

We call a dart  $x \rightarrow y$  **active** if  $\mathbf{slack}_\lambda(x \rightarrow y)$  is a decreasing function of  $\lambda$ ; only active darts can become tense.

The following lemma applies to *arbitrary* graphs, not just planar graphs or even graphs on surfaces.

**Lemma 3.1.** *The following claims hold for all real  $\lambda \in [0, 1]$ .*

- (a) *If  $su \notin T_\lambda$ , there are no active darts.*
- (b) *If  $sv \notin T_\lambda$ , the only active dart is  $s \rightarrow v$ .*
- (c) *Otherwise,  $x \rightarrow y$  is active if and only if  $x$  is blue and  $y$  is red.*

**Proof:** Consider an arbitrary dart  $x \rightarrow y$ . If  $x$  is blue and  $y$  is red then, for some constants  $\sigma_0, \sigma_1, \sigma_2 \geq 0$ ,

$$\begin{aligned} \mathbf{slack}_\lambda(x \rightarrow y) &= \mathbf{dist}_\lambda(x) + w_\lambda(x \rightarrow y) - \mathbf{dist}_\lambda(y) \\ &= w_\lambda(s \rightarrow v) + \sigma_1 + w(x \rightarrow y) - w_\lambda(s \rightarrow u) - \sigma_2 \\ &= (1 - \lambda)w(u \rightarrow v) + \sigma_1 + w(x \rightarrow y) - \lambda w(v \rightarrow u) - \sigma_2 \\ &= \sigma_0 - \lambda \widehat{w}(uv), \end{aligned}$$

and therefore  $x \rightarrow y$  is active. Symmetrically, if  $x$  is red and  $y$  is blue, then  $\mathbf{slack}_\lambda(x \rightarrow y)$  is an increasing function of  $\lambda$ , and so  $x \rightarrow y$  is not active. Finally, if  $x$  and  $y$  are both red or both blue, then  $\mathbf{slack}_\lambda(x \rightarrow y)$  is constant, so  $x \rightarrow y$  is not active.

If  $su$  is not an edge in  $T_\lambda$ , then  $sv \in T_\lambda$ , and thus every vertex except  $s$  is blue. In this case,  $\mathbf{slack}_\lambda(u \rightarrow s)$  is constant and  $\mathbf{slack}_\lambda(s \rightarrow u)$  is increasing with  $\lambda$ , so no dart is active.

Similarly, if  $sv$  is not an edge in  $T_\lambda$ , then  $su \in T_\lambda$ , and thus every vertex except  $s$  is red. In this case,  $\mathbf{slack}_\lambda(v \rightarrow s)$  is constant and  $\mathbf{slack}_\lambda(s \rightarrow v)$  is decreasing with  $\lambda$ , so  $s \rightarrow v$  is the only active dart.  $\square$

The proof of Lemma 3.1 has two immediate corollaries, which also apply to arbitrary graphs:

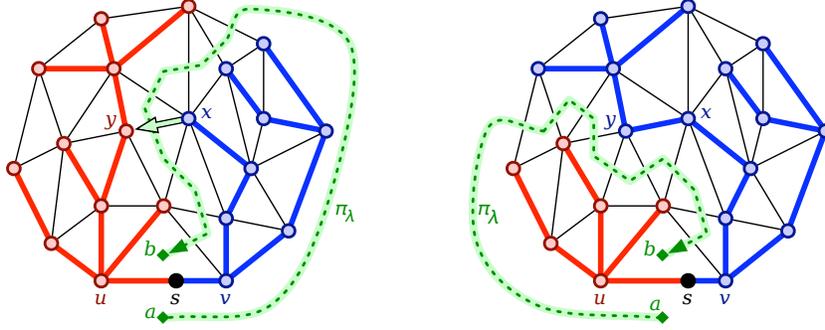
**Corollary 3.2.** *Fix a real value  $\lambda$  such that  $T_\lambda$  contains both  $su$  and  $sv$ . As  $\lambda$  increases, the next dart to become tense (if any) is the active dart  $x \rightarrow y$  such that  $\mathbf{slack}_\lambda(x \rightarrow y)$  is minimized.*

**Proof:** If  $x \rightarrow y$  is active, then  $\mathbf{slack}_\lambda(x \rightarrow y) = \sigma_0 - \lambda \widehat{w}(uv)$  for some constant  $\sigma_0 \geq 0$ . Thus, the slack of every active dart decreases at exactly the same rate.  $\square$

**Corollary 3.3.** *As  $\lambda$  increases continuously from 0 to 1, the number of pivots in  $T_\lambda$  is equal to the number of edges in  $T_u \setminus T_v$ .*

**Proof:** Just before any dart  $x \rightarrow y$  pivots into the shortest-path tree,  $x$  is blue and  $y$  is red; after the pivot, both  $x$  and  $y$  are blue. Thus, any dart that pivots into  $T_\lambda$  never pivots out again.  $\square$

<sup>2</sup>Readers familiar with astronomy may recall that visible light from objects moving away from the earth is shifted toward red, and visible light from objects moving toward the earth is shifted toward blue.



**Figure 1.** A single pivot in a planar shortest-path tree. Thick (red and blue) lines indicate the shortest-path tree  $T_\lambda$ ; the dotted (green) path is  $\pi_\lambda$ ; the hollow arrow indicates the pivoting dart  $x \rightarrow y$ .

### 3.2 Pivoting Quickly

To identify the next pivot quickly when  $G$  is planar, we maintain both the shortest-path tree  $T_\lambda$  and its complementary spanning cotree  $C_\lambda = G \setminus T_\lambda$  in dynamic tree structures. Specifically, we store  $T_\lambda$  in a primal tree structure, where the value of any node  $v$  is  $\text{dist}_\lambda(v)$ , and we store  $C_\lambda^*$  in a dual forest structure, where the value associated with any dual dart is the slack of the corresponding primal dart:  $\text{val}((x \rightarrow y)^*) := \text{slack}_\lambda(x \rightarrow y)$ .

To simplify notation related to the dart  $u \rightarrow v$ , let  $\mathbf{a}$  be the face  $\text{tail}((u \rightarrow v)^*) = \text{right}(u \rightarrow v)^*$ , let  $\mathbf{b}$  be the face  $\text{head}((u \rightarrow v)^*) = \text{left}(u \rightarrow v)^*$ , and let  $\pi_\lambda$  denote the unique directed path from  $\mathbf{a}$  to  $\mathbf{b}$  in the in the dual spanning tree  $C_\lambda^*$ . See Figure 1.

**Lemma 3.4.** *If  $T_\lambda$  contains both  $su$  and  $sv$ , then the next dart to become tense (if any) is the dart with minimum slack whose dual lies in the directed path  $\pi_\lambda$ .*

**Proof:** Lemma 3.1 and Corollary 3.2 imply that the next dart to become tense is the active dart  $x \rightarrow y$  with minimum slack. Thus, it suffices to prove that a dart is active if and only if its dual lies in  $\pi_\lambda$ .

The coloring of the vertices of  $G$  extends by duality to a coloring of the faces of  $G^*$ . Because the red and blue vertices of  $G$  define subtrees of  $T_\lambda$ , the union of the red dual faces and the union of the blue dual faces are both connected. Thus, both of these unions are topological disks whose common boundary is a cycle in  $G^*$ , composed of the edge  $ab$  and the unique (undirected) path from  $\mathbf{a}$  to  $\mathbf{b}$  in  $C_\lambda^*$ . Any active dart must cross this cycle in the opposite direction as the dart  $u \rightarrow v$ .  $\square$

**Lemma 3.5.** *We can perform the next pivot into  $T_\lambda$  in  $O(\log n)$  amortized time.*

**Proof:** Our algorithms for finding the next dart to pivot and executing the pivot are shown in Figure 2. Each algorithm performs a constant number of dynamic forest operations, plus a constant amount of additional work.

Under most circumstances, calling  $\text{MINPATH}(a, b)$  gives us (the dual of) the next dart  $x \rightarrow y$  to pivot into the tree, but there are several boundary cases. Two such cases are described by Lemma 3.1(a) and (b); another arises when there is too much slack for  $x \rightarrow y$  to pivot before  $\lambda$  reaches 1. To detect this latter case, we compute the value  $\lambda'$  that would make the slack of  $x \rightarrow y$  zero and check whether it is below 1. All these cases are handled by our algorithm  $\text{FINDNEXTPIVOT}$ .

If  $\text{FINDNEXTPIVOT}$  returns a dart  $x \rightarrow y$ , we perform the necessary data structure updates by calling  $\text{PIVOT}(x \rightarrow y)$ . If  $\Delta$  is the current slack of  $x \rightarrow y$ , the parameter  $\lambda$  must increase by  $\Delta / \widehat{w}(uv)$  before  $x \rightarrow y$  actually pivots into  $T_\lambda$ . We increase the distances of the red vertices by  $\Delta_R = \Delta w(v \rightarrow u) / \widehat{w}(uv)$  by calling  $\text{ADDSUBTREE}(\Delta_R, u)$ , decrease the distances at the blue vertices by  $\Delta_B = \Delta w(u \rightarrow v) / \widehat{w}(uv)$  by

<pre> <b>FINDNEXTPIVOT:</b> if <math>\text{pred}(v) \neq s</math> then return <math>s \rightarrow v</math> if <math>\text{pred}(u) \neq s</math> then return NULL <math>(x \rightarrow y)^* \leftarrow \text{MINPATH}(a, b)</math> <math>\Delta \leftarrow \text{GETDARTVALUE}((x \rightarrow y)^*)</math> <math>\lambda' \leftarrow \lambda + \Delta / \widehat{w}(uv)</math> if <math>\lambda' &lt; 1</math>   return <math>x \rightarrow y</math> else   return NULL </pre>	<pre> <b>PIVOT(<math>x \rightarrow y</math>):</b> <math>\Delta \leftarrow \text{GETDARTVALUE}((x \rightarrow y)^*)</math> <math>\lambda' \leftarrow \lambda + \Delta / \widehat{w}(uv)</math> <math>\Delta_B \leftarrow \Delta w(u \rightarrow v) / \widehat{w}(uv)</math> <math>\Delta_R \leftarrow \Delta w(v \rightarrow u) / \widehat{w}(uv)</math> if <math>\text{pred}(u) = s</math> then <math>\text{ADDSUBTREE}(\Delta_R, u)</math> if <math>\text{pred}(v) = s</math> then <math>\text{ADDSUBTREE}(-\Delta_B, v)</math> <math>\text{ADDPATH}(-\Delta, a, b)</math> <math>z \leftarrow \text{pred}(y)</math>; <math>\text{pred}(y) \leftarrow x</math> <math>\text{CUT}(zy)</math>; <math>\text{LINK}(x, y)</math> <math>\text{CUT}((xy)^*)</math>; <math>\text{LINK}((z \rightarrow y)^*, 0, \widehat{w}(yz))</math> </pre>
--	--

**Figure 2.** Algorithms to find and execute the next pivot when  $G$  is planar.

calling  $\text{ADDSUBTREE}(-\Delta_B, v)$ , and adjust the slacks of all the active darts and their reversals by calling  $\text{ADDPATH}(-\Delta, a, b)$ . Finally, we fix the underlying tree-cotree decomposition using two CUT and two LINK operations.  $\square$

When there are no edges left to pivot into the tree  $T_\lambda$ , the algorithm FINDNEXTPIVOT returns NULL. However, we still have to "slide" the source  $s$  by  $1 - \lambda$  to reach  $v$ . Thus, the distances in the primal tree have to be updated by calling  $\text{ADDSUBTREE}(-(1 - \lambda)w(u \rightarrow v) / \widehat{w}(uv), v)$  and, if  $\text{pred}(u) \neq s$ , also  $\text{ADDSUBTREE}((1 - \lambda)w(v \rightarrow u) / \widehat{w}(uv), u)$ .

**Theorem 3.6.** *Let  $G$  be a directed plane graph with  $n$  vertices, and let  $s$  be any vertex in  $G$ . After  $O(n \log n)$  preprocessing time, we can maintain a representation of the shortest-path tree  $T_s$  that supports the following operations:*

- Given any vertex  $v$ , return the shortest-path distance from  $s$  to  $v$  in  $O(\log n)$  time.
- Given any vertex  $v$ , return the last edge on the shortest path from  $s$  to  $v$  in  $O(1)$  time.
- For any edge  $sv$ , change the source vertex from  $s$  to  $v$  in  $O(k \log n)$  amortized time, where  $k$  is the number of edges in  $T_s \setminus T_v$ .

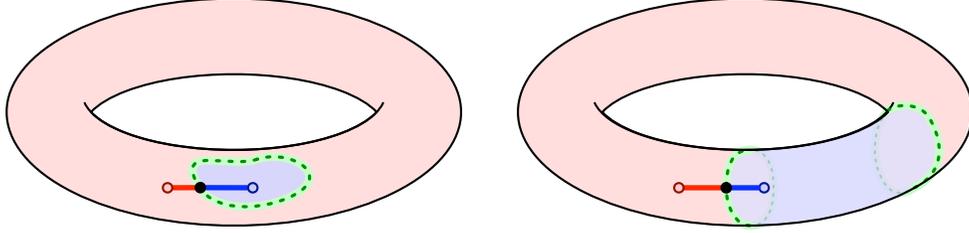
A nearly identical algorithm updates the shortest-path tree after changing the weight of any single edge, or deleting an edge, or inserting an edge (provided the graph remains planar), in  $O(k \log n)$  amortized time, where  $k$  is the number of edges added to or deleted from the shortest-path tree.

### 3.3 Moving Around a Face

Finally, we bound the running time of our algorithm as the source vertex moves all the way around the boundary of a given face  $f$ . Klein [42] noted that if one maintains the so-called *leftmost shortest path tree*, each edge enters and leaves the shortest path tree at most a constant number of times. (For graphs with unique shortest paths, the leftmost shortest path tree is the unique shortest path tree.) This property also holds for our algorithm.

**Lemma 3.7.** *As the source point  $s$  moves around the boundary of  $f$ , each dart of  $G$  pivots into (or out of) the shortest path tree rooted at  $s$  at most once.*

**Proof:** Without loss of generality, assume  $f$  is the outer face of  $G$ . Let  $T$  be any spanning tree  $T$  of  $G$ , let  $e$  be any edge of  $T$ , and let  $S$  be either component of  $T \setminus e$ . An easy inductive proof implies that the subtree  $S$  contains either no vertices of  $f$ , every vertex of  $f$ , or a contiguous interval of vertices of  $f$ .



**Figure 3.** On a higher-genus surface, the boundary between the red and blue dual faces may consist of multiple cycles.

Consider an arbitrary dart  $x \rightarrow y$ . Let  $A$  be the set of vertices of  $f$  whose shortest-path trees contain  $x \rightarrow y$ . Equivalently,  $A$  is the set of vertices of  $f$  that lie in the subtree rooted at  $x$  in the shortest path tree rooted at  $y$ . Thus, either  $A$  is empty,  $A$  is the complete set of vertices in  $f$ , or  $A$  is a contiguous interval of vertices of  $\partial f$ . In the first case,  $x \rightarrow y$  is never in the shortest path tree; in the second case,  $x \rightarrow y$  is always in the shortest path tree; in the third case,  $x \rightarrow y$  pivots in exactly once and pivots out exactly once.  $\square$

Like Klein [42], we can assume the input graph  $G$  graph has bounded degree. This assumption allows us to maintain the primal dynamic tree structure and the predecessor pointers at every node in a persistent data structure [16], so that we can access any version of the shortest-path tree in  $O(\log n)$  time. (It is not necessary to make the dual forest structure persistent.) The resulting persistent data structure requires  $O(\log n)$  space per pivot, or  $O(n \log n)$  space overall. We conclude:

**Theorem 3.8.** *Let  $G$  be a directed plane graph with  $n$  vertices, and let  $f$  be any face of  $G$ . In  $O(n \log n)$  time and space, we can construct a data structure that supports the following operations:*

- *Given any vertex  $u$  on the boundary of  $f$  and any other vertex  $v$ , return the shortest-path distance from  $u$  to  $v$  in  $O(\log n)$  time.*
- *Given any vertex  $u$  on the boundary of  $f$  and any other vertex  $v$ , return the shortest path from  $u$  to  $v$  in  $O(\log n + k)$  time, where  $k$  is the number of edges in the path.*

## 4 Algorithm for Higher-Genus Graphs

Our algorithm for planar graphs does not immediately extend to graphs of higher genus, primarily because the complement of a spanning tree is no longer a cotree, and therefore the active darts can have much more complex structure than a single dual path. Even when the red and blue subtrees are separated by a single dual cycle at one value of  $\lambda$ , a single pivot can split the red-blue boundary into multiple dual cycles; see Figure 3. Also, the analysis of the planar algorithm (implicitly) relies on the Jordan Curve Theorem, which does not hold on surfaces of higher genus.

We first restrict our attention to *orientable* surfaces. In Section 4.4 we show how to reduce the case of non-orientable surfaces to orientable ones.

### 4.1 Maintaining a Grove

Let  $G$  be a cellularly embedded graph on an orientable surface  $\Sigma$  of genus  $g > 0$ . Without loss of generality, we assume that  $G$  is a triangulation, so that every vertex of the dual graph  $G^*$  has degree 3. Let  $F$  be a spanning forest of  $G$  with  $\kappa$  components (that is, a spanning tree of  $G$  minus  $\kappa - 1$  edges) for some constant  $\kappa$ . (In our application, we always have  $1 \leq \kappa \leq 3$ .) Euler's formula implies that the complementary dual subgraph  $X = (G \setminus F)^*$  is a tree plus  $2g + \kappa - 1$  extra edges. Following Erickson

and Har-Peled [22], we refer to  $X$  as a **cut graph**; removing  $X$  cuts the underlying surface  $\Sigma$  into  $\kappa$  topological disks.

Our parametric shortest-path algorithm requires us to quickly identify active darts in this cut graph, and to maintain it as edges are inserted and deleted from  $F$ . To support these operations quickly, we decompose  $X$  into  $O(g)$  edge-disjoint subtrees as follows.

Let  $\bar{X}$  be the subgraph of  $X$  obtained by repeatedly removing vertices of degree 1 until no more remain; the subgraph of removed edges is a forest, which we denote  $H$  (for ‘hair’). Equivalently, a dual edge  $e^*$  is in  $\bar{X}$  if and only if the endpoints of the primal edge  $e$  lie in different trees in  $F$ , or if both endpoints are in the same tree, but adding  $e$  to that tree creates a non-contractible cycle.

Euler’s formula implies that  $\bar{X}$  consists of  $6g + 3\kappa - 3 = O(g)$  paths  $\pi_1, \pi_2, \pi_3, \dots$ , which meet at  $4g + 2\kappa - 2 = O(g)$  vertices of degree 3 [22, Lemma 4.2]. We refer to each path  $\pi_i$  as a **cut path**, and each degree-3 vertex a **branch point**. For each index  $i$ , let  $C_i$  denote the union of  $\pi_i$  with all trees in  $H$  that share a vertex with  $\pi_i$ . We call each cut path  $\pi_i$  the **anchor path** and its endpoints  $a_i$  and  $b_i$  the **anchor vertices** of the corresponding subtree  $C_i$ . We call the set of  $O(g)$  edge-disjoint subtrees  $\{C_1, C_2, C_3, \dots\}$  a **grove**.<sup>3</sup>

We maintain the grove by storing the subtrees  $C_i$  in the dual forest data structure described in Section 2.3. We maintain three separate copies of each branch point in this data structure, one for each subtree  $C_i$  that contains it. We also separately record (the correct copies of) the anchor vertices of each subtree  $C_i$ . Our grove data structure supports two restructuring operations:

- **GROVELINK( $u \rightarrow v, \alpha, \beta$ )**: Add edge  $uv$  to the cut graph, and set  $val(u \rightarrow v) = \alpha$  and  $val(v \rightarrow u) = \beta$ . This operation assumes that  $u$  and  $v$  are not in the same subtree  $C_i$ .
- **GROVECUT( $uv$ )**: Remove edge  $uv$  from the cut graph.

To insert an edge  $uv$  into  $X$ , we first determine the subtrees  $C_i$  and  $C_j$  that contain  $u$  and  $v$ , respectively. We then find the vertices  $\hat{a} = \text{JUNCTION}(u, a_i, b_i)$  and  $\hat{b} = \text{JUNCTION}(v, a_j, b_j)$ . Next we insert the edge into the dynamic forest by calling  $\text{LINK}(u \rightarrow v, \alpha, \beta)$ , which merges the two subtrees  $C_i$  and  $C_j$  into a single subtree  $\hat{C}$ . We split  $\hat{C}$  into five smaller subtrees by splitting  $\hat{a}$  and  $\hat{b}$  into three copies, each carrying one outgoing edge, using a constant number of **CUT** and **LINK** operations. Finally, we store the anchor vertices of each of the five new subtrees; in particular  $\hat{a}$  and  $\hat{b}$  are the anchor vertices for the subtree containing  $uv$ . See Figure 4. The entire procedure takes  $O(\log n)$  amortized time.

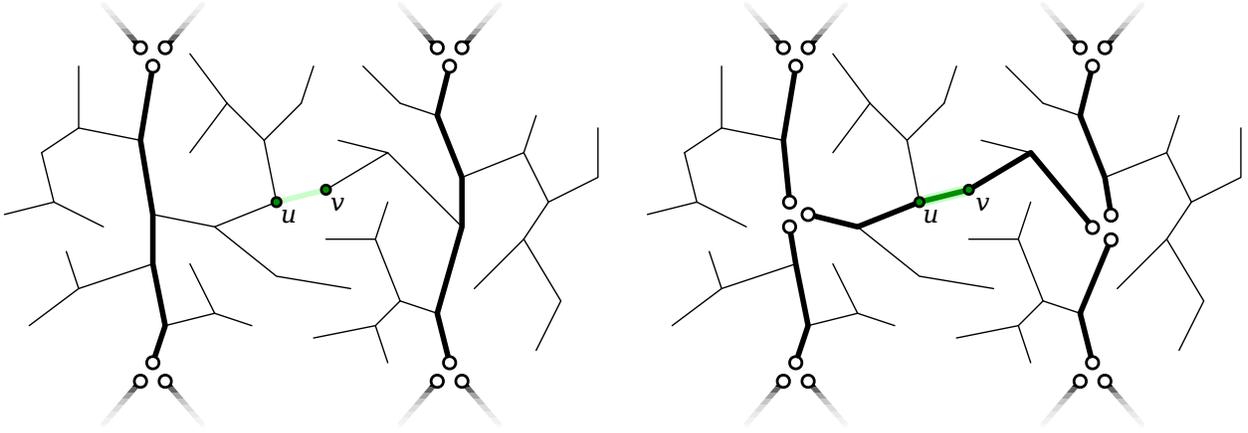
To remove the edge  $uv$ , we follow the insertion procedure in reverse. We begin by determining which tree  $C_i$  that contains  $uv$ . We then merge the three copies of the anchors  $a_i$  and  $b_i$  into single vertices, thereby merging five trees into a single tree  $\hat{C}$ . We remove the edge from the dynamic forest by calling  $\text{CUT}(uv)$ , splitting  $\hat{C}$  into two subtrees, one containing  $u$  and the other containing  $v$ . Finally, we update the anchor vertices of these two subtrees. Again, the entire procedure takes  $O(\log n)$  amortized time.

## 4.2 Pivoting Quickly

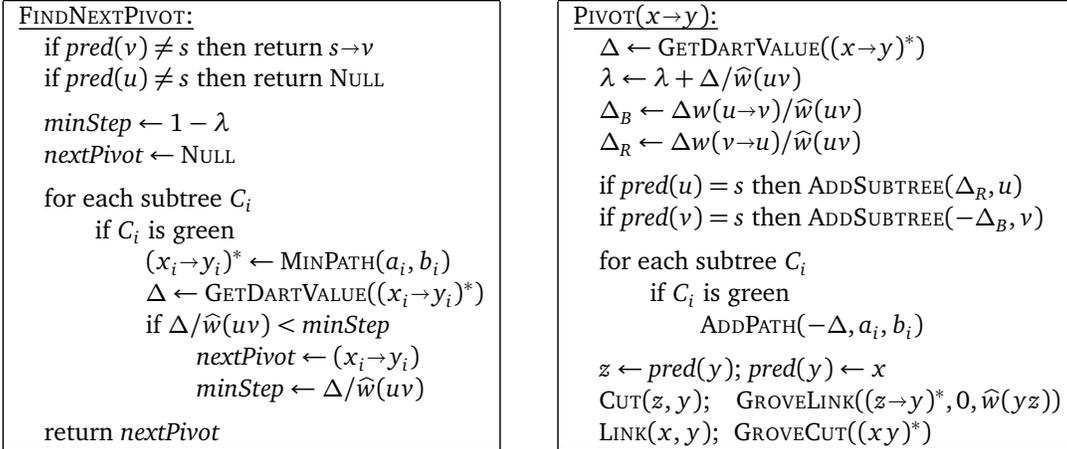
Now we describe how to efficiently move the source of a shortest-path tree in  $G$  along a single edge  $uv$ , using the parametric shortest path approach described in Section 3.1. We cannot apply the planar algorithm directly, because the set of active darts is no longer dual to a directed path in  $G^*$ . However, the grove structure we just described lets us quickly identify a superset of the active darts.

Suppose  $su$  and  $sv$  are both edges in the current shortest-path tree  $T_\lambda$ . Let  $R$  and  $B$  be the induced red and blue subtrees of  $T_\lambda$ , and let  $X = (G \setminus (R \cup B))^*$ . We maintain the cut graph  $X$  in a grove, as described above. We also associate a color with each subtree  $C_i$  in the grove according to the colors of

<sup>3</sup>The Oxford English Dictionary defines *grove* as ‘‘A small wood; a group of trees affording shade or forming avenues or walks, occurring naturally or planted for a special purpose.’’



**Figure 4.** Maintaining a grove. Left to right: Inserting an edge. Right to left: Deleting an edge.



**Figure 5.** Naive algorithms to find and execute the next pivot when  $G$  has positive genus.

primal edges that cross the anchor path  $\pi_i$ . If the crossing edges have two red endpoints, we color  $C_i$  *red*; if the crossing edges have two blue endpoints, we color  $C_i$  *blue*; if the crossing edges have one red endpoint and one blue endpoint, we color  $C_i$  *green*. Observe that the green edges of  $G$  are precisely the edges that cross green anchor paths.

We direct the green anchor paths so that they contain the duals of active darts; that is, for any dart  $x \rightarrow y$  where  $x$  is blue and  $y$  is red, the dual dart  $(x \rightarrow y)^*$  lies on some directed green anchor path. For each directed green anchor path  $\pi_i$ , let  $a_i$  denote its start vertex and  $b_i$  its end vertex. The following lemma is now almost immediate.

**Lemma 4.1.** *We can perform the next pivot into  $T_\lambda$  in  $O(g \log n)$  amortized time.*

**Proof:** The modified algorithms to find the next edge to become tense and pivot it into the shortest-path tree are shown in Figure 5. There are only a few differences from the planar algorithms. First, we must examine every directed green anchor path to find the next edge to pivot. Second, we must call  $\text{ADDPATH}$  on every green anchor path to update the slacks of the green edges. Finally, we must call  $\text{GROVECUT}$  and  $\text{GROVELINK}$  to repair the grove data structure. These algorithms perform  $O(g)$  dynamic forest operations, plus constant additional work, so their amortized running time is  $O(g \log n)$ .  $\square$

With a little more effort, we can improve the running time slightly. In addition to our other data structures, we maintain a priority queue of the green subtrees in the grove, where the priority of any subtree  $C_i$  is the minimum slack among all darts whose duals lie in  $\pi_i$ . Because the grove never contains more than  $O(g)$  subtrees, each priority queue operation takes  $O(\log g)$  time.

To find the next dart to pivot, instead of looping over every subtree in the grove, we simply extract the minimum element from the priority queue. Within each call to GROVECUT or GROVELINK, we perform a constant number of priority queue operations as subtrees are created and destroyed.

Finally, instead of calling ADDPATH once for each green subtree in PIVOT, we maintain a global offset for each green subtree in its priority queue record. This allows us to call ADDPATH just once on each green subtree, just before it is removed from the priority queue—when the subtree changes color, when the subtree is destroyed by GROVECUT or GROVELINK, or when the moving source point  $s$  reaches  $v$ . Thus, crudely, the number of calls to ADDPATH is less than the number of priority queue operations.

If we ignore priority queue operations and calls to ADDPATH, the remainder of FINDNEXTPIVOT and PIVOT requires  $O(\log n)$  amortized time. Thus, to complete our analysis, we need only an upper bound on the number of priority queue operations.

A single pivot can change the colors of several subtrees in the grove. When a subtree changes from red to green, we must insert it into the priority queue; when it changes from red to blue or from green to blue, we must delete it. To detect these color changes quickly, we separately maintain a **reduced cut graph**  $\tilde{X}$ , an abstract graph with a vertex for every branch point of  $X$  and an edge  $\tilde{e}_i$  for every cut path  $\pi_i$ , with a cellular embedding on  $\Sigma$  consistent with the embedding of  $\bar{X}$ . Each face of the reduced cut graph corresponds to a component of the primal forest  $F$ . We can easily update the reduced cut graph in  $O(1)$  time within each call to GROVECUT or GROVELINK.

Just after removing the old edge  $z \rightarrow y$  from the shortest-path tree  $T_\lambda$  and adding its dual to the grove, color the subtree rooted at  $y$  **purple**. Now the reduced cut graph  $\tilde{X}$  has exactly three faces: one red, one blue, and one purple. Edges  $\tilde{e}_i$  on the boundary of the purple face correspond to subtrees  $C_i$  that are changing color; specifically:

- If  $\tilde{e}_i$  also bounds the red face, the corresponding subtree  $C_i$  is changing from red to green; we insert it into the priority queue.
- If  $\tilde{e}_i$  also bounds the blue face, the corresponding subtree  $C_i$  is changing from green to blue; we delete it from the priority queue.
- If  $\tilde{e}_i$  has the purple face on both sides, the corresponding subtree  $C_i$  is changing directly from red to blue; we ignore it. (However, for purposes of analysis, we pretend to execute two priority queue operations.)

During a single pivot, the time to update  $\tilde{X}$  and traverse the purple face is less than the time spent maintaining the priority queue, so we can ignore it.

Now recall that the grove always contains  $O(g)$  subtrees. Let  $n_{\text{red}}$  and  $n_{\text{blue}}$  denote the number of red and blue subtrees, respectively. The number of priority queue operations for a single pivot is a constant (less than 20) plus the *decrease* in  $n_{\text{red}}$  plus the *increase* in  $n_{\text{blue}}$ . Over the entire parametric shortest path algorithm, the total decrease in  $n_{\text{red}}$  and the total increase in  $n_{\text{red}}$  is at most  $O(g)$ .

Thus, if moving the source point across  $uv$  requires  $k$  pivots, the total number of priority queue operations is  $O(g+k)$ . Because we perform at most one call to ADDPATH for each priority queue operation, the number of calls to ADDPATH is also at most  $O(g+k)$ . We conclude:

**Theorem 4.2.** *Let  $G$  be a directed graph with  $n$  vertices, cellularly embedded on an orientable surface of genus  $g$ , and let  $s$  be any vertex in  $G$ . After  $O(n \log n)$  preprocessing time, we can maintain a representation of the shortest-path tree  $T_s$  that supports the following operations:*

- Given any vertex  $v$ , return the shortest-path distance from  $s$  to  $v$  in  $O(\log n)$  time.
- Given any vertex  $v$ , return the last edge on the shortest path from  $s$  to  $v$  in  $O(1)$  time.
- For any edge  $sv$ , change the source vertex from  $s$  to  $v$  in  $O((g+k)\log n)$  amortized time, where  $k$  is the number of edges in  $T_s \setminus T_v$ .

Again, nearly identical algorithms update the shortest-path tree after changing the weight of any single edge, or deleting an edge, or inserting an edge (provided the graph remains embedded), in  $O((g+k)\log n)$  amortized time, where  $k$  is the number of edges added to or deleted from the shortest-path tree.

### 4.3 Moving Around a Face

We next bound the running time of our algorithm as the source vertex moves all the way around the boundary of a given face  $f$ . We first show a bound on the number of times that an edge can enter or leave the shortest path tree during the entire traversal.

**Lemma 4.3.** *As the source point  $s$  moves around the boundary of  $f$ , each dart of  $G$  pivots into (or out of) the shortest path tree rooted at  $s$  at most  $O(g)$  times.*

**Proof:** Fix an arbitrary dart  $x \rightarrow y$ . Let  $A$  be the set of points  $s$  (not just vertices) on the boundary of  $f$  such that the shortest path tree rooted at  $s$  contains the dart  $x \rightarrow y$ .  $A$  is the union of a set of disjoint, maximal paths  $A_1, \dots, A_r$  on the boundary of  $f$ . Dart  $x \rightarrow y$  enters the shortest path tree exactly  $r$  times, once at the initial endpoint of each interval  $A_i$ .

Fix a point  $v_i$  in each component  $A_i$ , and let  $p_i$  be the shortest path from  $v_i$  to  $y$ . By construction,  $p_i$  uses the dart  $x \rightarrow y$ , and for all  $i \neq j$ , paths  $p_i$  and  $p_j$  do not cross.

Let  $\Sigma/f$  be the surface obtained by contracting the face  $f$  to a point. Suppose  $p_i$  and  $p_j$  are homotopic in  $\Sigma/f$ . Then in  $\Sigma$ , there is a subwalk  $f'$  of  $f$  that together with  $p_i$  and  $p_j$  bound a disk (and thus a planar graph). This implies that the shortest path to  $y$  from every vertex of  $f'$  must contain the dart  $x \rightarrow y$ , because of Lemma 3.7. It follows that  $v_i$  and  $v_j$  lie in the same connected component of  $A$ , which is only possible if  $i$  and  $j$  must be equal. We conclude that the paths  $p_1, p_2, \dots, p_r$  are pairwise non-homotopic.

Finally, a double-counting argument using Euler's formula implies that the maximum possible number of pairwise non-crossing, non-homotopic paths in  $\Sigma/f$  is  $O(g)$  [11, Lemma 2.1]. Thus,  $r = O(g)$ , and the proof is complete.  $\square$

Recall from Theorem 4.2 that the time to move the shortest path tree across a single edge is  $O((g+k)\log n)$ , where  $k$  is the number of pivots. The previous lemma implies that the total number of pivots is  $O(gn)$ , and the total number of edges in  $f$  is trivially  $O(n)$ . Thus, the total running time to move the source of the shortest path tree along the boundary of a face is  $O(gn \log n)$ .

Finally, in order to make our primal data structure persistent, we require the graph to have bounded degree; however, our grove data structure requires the graph to be a triangulation. We can escape this seeming contradiction as follows. Given an *arbitrary* embedded graph  $G$ , we first replace every high-degree vertex with a small tree of degree-3 vertices, each consisting of darts with infinitesimal weight, and then we triangulate each face with edges whose darts have large enough weight. The edges whose darts have large weight never participate in any shortest path, so the maximum degree of any node in the shortest-path path tree is always at most 3, and so the primal tree structure can be made persistent efficiently. We conclude:

**Theorem 4.4.** *Let  $G$  be an directed graph with  $n$  vertices, cellularly embedded in an orientable surface of genus  $g$ , and let  $f$  be any face of  $G$ . In  $O(gn \log n)$  time and space, we can construct a data structure that supports the following operations:*

- *Given any vertex  $u$  on the boundary of  $f$  and any other vertex  $v$ , return the shortest-path distance from  $u$  to  $v$  in  $O(\log n)$  time.*
- *Given any vertex  $u$  on the boundary of  $f$  and any other vertex  $v$ , return the shortest path from  $u$  to  $v$  in  $O(\log n + k)$  time, where  $k$  is the number of edges in the path.*

#### 4.4 Non-Orientable Surfaces

We can extend Theorem 4.4 to non-orientable surfaces working in the *orientable double cover* of the surface, which is an orientable covering space of  $\Sigma$ . (Our construction is essentially the same as used in [9, 10], but is presented here for completeness.)

Recall that a cellular graph embedding of a graph  $G$  on a non-orientable surface consists of a rotation system, encoding the cyclic sequence of outgoing darts from each vertex, and a signature, consisting of a bit  $i(e)$  for each edge  $e$ . The signature indicates whether the cyclic sequences at its endpoints are in the “same” direction or the “opposite” direction.

The **orientable double cover**  $D$  of  $G$  is an embedded graph constructed as follows. For each vertex  $v$  in the original graph, create two vertices  $(v, 0)$  and  $(v, 1)$  in  $D$ . For each dart  $u \rightarrow v$  in the original graph, create two darts  $(u \rightarrow v, 0) = (u, 0) \rightarrow (v, i(uv))$  and  $(u \rightarrow v, 1) = (u, 1) \rightarrow (v, 1 - i(uv))$  in  $D$ . The rotation system at  $(v, 0)$  and  $(v, 1)$  is just a copy of the rotation system of  $v$ . That is, if  $\pi_G$  is the original rotation system and  $\pi_D$  the rotation system in  $D$ , then for any dart  $u \rightarrow v$  of  $G$  it holds  $\pi_D(u \rightarrow v, j) = (\pi_G(u \rightarrow v), j)$ . Finally, darts  $(u \rightarrow v, 0)$  and  $(u \rightarrow v, 1)$  have the same dart weight and the same signature bit as  $u \rightarrow v$ . This cover is orientable because, for any cycle, the sum of the signatures of its edges is 0 modulo 2. Note that this cover  $D$  has twice as many edges as  $G$  and can be constructed in linear time.

Any walk  $(u_0, j_0)(u_1, j_1)(u_2, j_2), \dots, (u_t, j_t)$  in  $D$  naturally **projects** to the walk  $u_0 u_1 u_2 \dots u_t$  in  $G$ . A **lift** of a walk  $u_0 u_1 u_2 \dots u_t$  in  $G$  is a walk  $(u_0, j_0)(u_1, j_1)(u_2, j_2), \dots, (u_t, j_t)$  in  $D$ . That is, the projection of a lift recovers the original walk. A lift is uniquely determined by the value  $j_0$ . Indeed, since  $(u_\ell, j_\ell)(u_{\ell+1}, j_{\ell+1})$  must be an edge of  $D$ , we have

$$j_{\ell+1} = j_\ell + i(u_\ell u_{\ell+1}) = j_0 + \sum_{\ell' \leq \ell} i(u_{\ell'} u_{\ell'+1}) \quad (\text{modulo } 2).$$

This implies that a shortest path from  $u$  to  $v$  in  $G$  corresponds to either the shortest path from  $(u, j)$  to  $(v, 0)$  or the shortest path from  $(u, j)$  to  $(v, 1)$ , where  $j \in \{0, 1\}$  is arbitrary. Therefore, the distance in  $G$  from  $u$  to  $v$  is the minimum between the distance from  $(u, j)$  to  $(v, 0)$  and the distance from  $(u, j)$  to  $(v, 1)$ .

It is easy to see that a cycle in  $D$  is a facial cycle of  $D$  if and only if it projects to a facial cycle of  $G$ . It follows that  $D$  has twice as many faces as  $G$ . Euler’s formula then implies that the genus of  $D$  is  $g - 1$ .

We can now extend Theorem 4.4 to non-orientable surfaces. Given the graph  $G$  and a face  $f$ , we construct the orientable double cover  $D$ , choose one of the lifts  $f'$  of  $f$  in the double cover, and store the data structure of Theorem 4.4 for  $D$  and  $f'$ . This construction takes  $O(gn \log n)$  time because  $D$  has size  $O(n)$  and smaller genus. Whenever we want to compute a distance in  $G$  from a vertex  $u$ , on the boundary of  $f$ , to a vertex  $v$ , we take the copy  $(u, j)$  of  $s$  on the boundary of  $f'$ , query for the distances in  $D$  from  $(u, j)$  to  $(v, 0)$  and from  $(u, j)$  to  $(v, 1)$ , and return the smallest. This takes  $O(\log n)$  time. The shortest path from  $s$  to  $v$  can also be recovered using the projection of the shortest path in  $D$  from  $(u, j)$  to  $(v, 0)$  or  $(v, 1)$ , whichever is closer. We summarize:

**Corollary 4.5.** *Theorem 4.4 also holds for non-orientable surfaces.*

## 5 Computing Shortest Non-Separating and Non-Contractible Cycles

Let  $\Sigma$  be a combinatorial surface with complexity  $n$  and genus  $g$ ; this surface may or may not be orientable. In this section, we describe algorithms to find the shortest non-separating and shortest non-contractible cycles in  $\Sigma$ . Our use of the technique for maintaining shortest-path trees is condensed in the following lemma.

**Lemma 5.1.** *Let  $\alpha$  be an arbitrary simple cycle or arc in  $\Sigma$ . A shortest cycle crossing  $\alpha$  exactly once can be obtained in  $O(gn \log n)$  time.*

**Proof:** Consider the surface obtained by cutting  $\Sigma$  along  $\alpha$ : each vertex  $v$  in  $\alpha$  gives rise to two vertices  $v'$  and  $v''$ , and two boundary arcs or cycles  $\alpha'$  and  $\alpha''$ . Let  $\Sigma'$  be the surface obtained by gluing disks to the boundary components that contain  $\alpha'$  and  $\alpha''$ . (If  $\alpha$  is an arc or a 1-sided cycle, then  $\alpha'$  and  $\alpha''$  can be contained in a single boundary.) A cycle in  $\Sigma$  that crosses  $\alpha$  once at a point  $v$  becomes a path in  $\Sigma'$  between  $v'$  and  $v''$ . Thus, a shortest cycle that crosses  $\alpha$  once at  $v$  is a shortest path that connects  $v'$  to  $v''$  in  $\Sigma'$ , and vice versa. All the points  $v'$  with  $v \in \alpha$  belong to a face of  $\Sigma'$ . Thus, Theorem 4.4 implies that we can find a closest pair  $(v'_0, v''_0)$  in  $O(gn \log n)$  time. Computing the shortest path from  $v'_0$  to  $v''_0$  gives the desired path.  $\square$

For any simple arc or cycle  $\alpha$ , let  $Cross(\alpha)$  denote the set of cycles that cross  $\alpha$  exactly once. If  $\alpha$  is separating, then  $Cross(\alpha)$  is empty, because every cycle crosses  $\alpha$  an even number of times. Also, every cycle in  $Cross(\alpha)$  is non-contractible, because contractible cycles are also separating, and therefore any cycle must cross them an even number of times.

### 5.1 Shortest Non-Separating Cycle

Consider a surface  $\Sigma$ . It suffices to consider surfaces without boundary, because any shortest non-separating cycle in  $\Sigma$  is also non-separating in the surface obtained by attaching disks to the boundaries.

Cabello and Mohar [10] describe how to construct in  $O(gn \log n)$  time a set  $S$  of  $O(g)$  simple cycle such that the shortest cycle in  $\bigcup_{\ell \in S} Cross(\ell)$  is a shortest non-separating cycle. This set is constructed by fixing a shortest path tree  $T_x$  from any vertex  $x \in G$ , fixing a tree-cotree decomposition  $(T_x, C_x, L_x)$ , and then setting  $S$  to be the set of cycles formed by adding each of the edges  $e \in L_x$  to  $T_x$ . Cabello and Mohar use  $\mathbb{Z}_2$  homology and the fact that the cycles are formed from 2 shortest paths to prove that the shortest non-separating cycle must cross some cycle of  $S$  exactly once [10].

Once we have the set  $S$ , we compute the shortest non-separating cycle by applying Lemma 5.1 once for each simple cycle in  $S$ , and taking the globally shortest cycle.

**Theorem 5.2.** *Let  $\Sigma$  be a surface, orientable or not, possibly with boundary, of complexity  $n$  and genus  $g$ . We can find a shortest non-separating cycle in  $\Sigma$  in  $O(g^2n \log n)$  time.*

### 5.2 Shortest Non-Contractible Cycle

The main technique for non-contractible cycles is to find a curve that intersects a shortest non-contractible cycle at most once and whose removal decreases either the genus or the number of boundaries of  $\Sigma$ . Our main tool to prove that such a curve exists is the following exchange argument.

**Lemma 5.3.** *Let  $\Sigma$  be a combinatorial surface and let  $\ell_x$  be a shortest non-contractible loop with given basepoint  $x \in \Sigma$ . There is a shortest non-contractible cycle in  $\Sigma$  that crosses  $\ell_x$  at most once.*

**Proof:** Let  $C$  be any non-contractible cycle that crosses  $\ell_x$  at least twice, at points  $y$  and  $z$ . Let  $\gamma_1$  and  $\gamma_2$  be the two subpaths of  $C$  from  $y$  to  $z$ , and let  $\beta_1$  and  $\beta_2$  be the subpaths of  $\ell_x$  from  $y$  to  $z$  so that  $\beta_1$  contains  $x$ . To simplify notation, we do not differentiate between a path and its reverse. We consider two cases, and in each case find a non-contractible cycle  $\tilde{C}$  that is no longer than  $C$  and crosses  $\ell_x$  fewer times than  $C$ . This implies that some shortest non-contractible cycle crosses  $\ell_x$  at most once.

First, suppose  $\beta_2 \sim \gamma_1$ ; the case  $\beta_2 \sim \gamma_2$  is symmetric. The loop  $\beta_1 \cdot \gamma_1$  passes through  $x$ , and it is non-contractible because  $(\beta_1 \cdot \gamma_1) \sim (\beta_1 \cdot \beta_2) = \ell_x$ . We then have  $|\beta_2| \leq |\gamma_1|$  because  $|\ell_x| \leq |\beta_1| + |\gamma_1|$ . The cycle  $\tilde{C} = \gamma_2 \cdot \beta_2$  is non-contractible because  $(\gamma_2 \cdot \beta_2) \sim (\gamma_2 \cdot \gamma_1) = C$ . The cycle  $\tilde{C}$  also crosses  $\ell_x$  at least two fewer times than  $C$ , and  $|\tilde{C}| = |\gamma_2| + |\beta_2| \leq |\gamma_2| + |\gamma_1| \leq |C|$ .

Now suppose that  $\beta_2$  is not homotopic to  $\gamma_1$  or  $\gamma_2$ . In this case, the cycles  $\beta_2 \cdot \gamma_1$  and  $\beta_2 \cdot \gamma_2$  are non-contractible. The cycle  $\beta_1 \cdot \gamma_1$  or the cycle  $\beta_1 \cdot \gamma_2$  is non-contractible (perhaps both); otherwise  $\gamma_1 \sim \beta_1 \sim \gamma_2$ , which would imply that  $C = \gamma_1 \cdot \gamma_2$  is contractible. Let us suppose that  $\beta_1 \cdot \gamma_1$  is non-contractible; the other case is symmetric. Since  $\beta_1 \cdot \gamma_1$  is a non-contractible loop through  $x$ , we have  $|\beta_1| + |\beta_2| = |\ell_x| \leq |\beta_1| + |\gamma_1|$ , and so  $|\beta_2| \leq |\gamma_1|$ . The cycle  $\tilde{C} = \beta_2 \cdot \gamma_2$  is non-contractible because  $\beta_2 \approx \gamma_2$ . Cycle  $\tilde{C}$  also crosses  $\ell_x$  two fewer times than  $C$ , and  $|\tilde{C}| = |\beta_2| + |\gamma_2| \leq |\gamma_1| + |\gamma_2| = |C|$ .  $\square$

The following lemma discusses what happens with simple non-contractible cycles when pasting a disk into a boundary of the surface. It should be noted that any shortest non-contractible cycle is always simple.

**Lemma 5.4.** *Let  $\Sigma$  be a surface with boundary and let  $\delta$  be one of its boundary components. Let  $\Sigma'$  be the surface obtained by pasting a disk to  $\delta$ . A non-contractible simple cycle in  $\Sigma$  is either non-contractible in  $\Sigma'$  or homotopic to  $\delta$  in  $\Sigma$ .*

**Proof:** Consider a non-contractible simple cycle  $C$  in  $\Sigma$ . Let  $D_\delta$  be the disk that is attached to  $\Sigma$  to obtain  $\Sigma'$ . If  $C$  is contractible in  $\Sigma'$ , then  $C$  bounds a disk  $D_C$  in  $\Sigma'$ . The disk  $D_C$  must contain  $D_\delta$ , because otherwise,  $C$  would also bound a disk in  $\Sigma$ , implying that  $C$  is contractible in  $\Sigma$ .  $D_C \setminus D_\delta$  is an annulus in  $\Sigma$  with boundary cycles  $C$  and  $\delta$ . It follows that  $C$  and  $\delta$  are homotopic in  $\Sigma$ .  $\square$

We also have the following results regarding arcs in a surface with boundary.

**Lemma 5.5.** *Let  $\Sigma$  be a surface with boundary, let  $\delta$  be one of its boundary cycles, and let  $\alpha$  be a shortest non-contractible arc with endpoints in  $\delta$ . There is a shortest non-contractible cycle in  $\Sigma$  that is either homotopic to  $\delta$  or that crosses  $\alpha$  at most once.*

**Proof:** Let  $C$  be a shortest non-contractible cycle in  $\Sigma$ . Assume  $C \approx \delta$ , since otherwise the proof is complete. Lemma 5.4 implies that  $C$  is a shortest non-contractible cycle in  $\Sigma'$ , the surface obtained by pasting a disk  $D$  at  $\delta$ . In the surface  $\Sigma'$ , place a new vertex  $p$  in  $D$  and connect it through edges with weight  $L$  to each vertex of  $\delta$ . Consider the loop  $\ell$  with basepoint  $p$  that follows the edge  $p\alpha(0)$ , the arc  $\alpha$ , and the edge  $\alpha(1)p$ . If we choose  $L$  large enough,  $\ell$  is a shortest non-contractible loop in  $\Sigma'$  through  $p$ , so Lemma 5.3(a) implies that a shortest non-contractible cycle  $C'$  in  $\Sigma'$  crosses  $\ell$  at most once. We must have  $|C'| = |C|$ , and if  $L$  is large enough,  $C'$  must avoid the disk  $D$ . It follows that  $C'$  is a shortest non-contractible cycle in  $\Sigma$  that crosses  $\alpha$  at most once.  $\square$

**Lemma 5.6.** *Let  $\Sigma$  be a surface with at least two boundary components, and let  $\alpha$  be a shortest arc connecting two different boundaries of  $\Sigma$ . There is a shortest non-contractible cycle in  $\Sigma$  that crosses  $\alpha$  at most once.*

**Proof:** Let  $C$  be a shortest non-contractible cycle in  $\Sigma$  that crosses  $\alpha$  the minimum number of times. We claim that  $C$  crosses  $\alpha$  at most once. Assume for the purpose of contradiction that  $C$  crosses  $\alpha$  at least twice, and let  $y$  and  $z$  be two such crossings. Let  $\gamma_1$  and  $\gamma_2$  be the two subpaths of  $C$  from  $y$  to  $z$ , let  $\tilde{\alpha}$  be the subpath of  $\alpha$  from  $y$  to  $z$ . Since  $\alpha$  is a shortest arc connecting the two specified boundaries, we have  $|\tilde{\alpha}| \leq |\gamma_1|$  and  $|\tilde{\alpha}| \leq |\gamma_2|$ . If  $\tilde{\alpha} \approx \gamma_1$ , then we have a contradiction: the cycle  $\tilde{\alpha} \cdot \gamma_1$  is non-contractible, crosses  $\alpha$  fewer times than  $C$  does, and  $|\tilde{\alpha}| + |\gamma_1| \leq |\gamma_2| + |\gamma_1| = |C|$ . Similarly, we must have  $\tilde{\alpha} \sim \gamma_2$ . But then  $\gamma_1 \sim \gamma_2$ , which implies that  $C$  is contractible, which is a contradiction.  $\square$

The next lemma summarizes the algorithmic tools that we will use.

**Lemma 5.7.** *Let  $\Sigma$  be a surface of complexity  $n$ .*

- (a) *Given a basepoint  $x$ , we can find a shortest non-contractible loop with basepoint  $x$  in  $O(n \log n)$  time. This loop has multiplicity 2.*
- (b) *Given a boundary  $\delta$ , we can find in  $O(n \log n)$  time a shortest cycle homotopic to  $\delta$ .*
- (c) *Given a boundary  $\delta$ , we can find in  $O(n \log n)$  time a shortest non-contractible arc with endpoints in  $\delta$ . This arc has multiplicity at most 2 and is edge-disjoint from  $\delta$ .*
- (d) *If  $\Sigma$  has exactly two boundaries, we can find in  $O(n \log n)$  time a shortest arc with endpoints in both boundaries. This arc has multiplicity 1 and is edge-disjoint from the boundary of  $\Sigma$ .*

**Proof:** (a) See Erickson and Har-Peled [22, Lemma 5.2].

(b) See Cabello et al [9].

(c) Contract  $\delta$  to a point  $p$  and construct a shortest non-contractible loop with basepoint  $p$  as in part (a). This is the desired arc in  $\Sigma$ .

(d) Select boundaries  $\delta$  and  $\delta'$  of  $\Sigma$ , contract them to points  $p$  and  $p'$ , and construct in  $O(n \log n)$  time the shortest path from  $p$  to  $p'$ .  $\square$

**Lemma 5.8.** *Let  $\Sigma$  be a combinatorial surface with complexity  $n$ , genus  $g$ , and  $b$  boundaries. Let  $\Sigma'$  be the surface obtained by pasting a disk to each boundary of  $\Sigma$ . We can find a shortest non-contractible cycle in  $\Sigma$  in  $O(bn \log n)$  time plus the time used to find a shortest non-contractible cycle in  $\Sigma'$ .*

**Proof:** Number the boundaries  $\delta_1, \delta_2, \dots, \delta_b$ , and set  $\Sigma_0 = \Sigma$ . For  $i \geq 1$ , let  $\Sigma_i$  be the surface obtained from  $\Sigma_{i-1}$  by pasting a disk to  $\delta_i$ . In particular,  $\Sigma_b = \Sigma'$ . For each  $i$ , let  $C_i$  be a shortest cycle homotopic to  $\delta_i$  in  $\Sigma_{i-1}$ . We can compute each cycle  $C_i$  in  $O(n \log n)$  time.

Lemma 5.4 implies that a shortest non-contractible cycle in  $\Sigma_{i-1}$  is either  $C_i$  or a shortest non-contractible cycle in  $\Sigma_{i-1}$ . Thus, the shortest non-contractible cycle in  $\Sigma$  is either the shortest among  $C_1, \dots, C_b$  or a shortest non-contractible cycle in  $\Sigma'$ .  $\square$

**Theorem 5.9.** *Let  $\Sigma$  be a combinatorial surface with complexity  $n$ , genus  $g$ , and  $b$  boundaries. We can find a shortest non-contractible cycle of  $\Sigma$  in  $O((g^2 + b)n \log n)$  time.*

**Proof:** We apply Lemma 5.8 to  $\Sigma$ . We spend  $O(bn \log n)$  time, and have reduced the problem to find a shortest non-contractible cycle in a surface  $\Sigma'$  of genus  $g$  and no boundary. We next give an iterative algorithm that reduces either the genus or the number of boundaries of the subproblems in each iteration. Through the algorithm we use a variable *minCycle* to store the shortest non-contractible cycle found so far. In each iteration we find one new non-contractible cycle  $\gamma$  whose length is compared against

the length of  $minCycle$ , and  $minCycle$  is updated if needed. The algorithm stops when each remaining component is a topological disk.

We distinguish three cases.  $\Sigma'$  denotes the surface in the subproblem. Through the algorithm the surface  $\Sigma'$  has genus at most  $g$  and each component of  $\Sigma'$  has at most two boundaries.

- (a) If  $\Sigma'$  is a surface without boundary, we choose a point  $x \in \Sigma'$  and find a shortest non-contractible loop  $\ell_x$  through  $x$  (Lemma 5.7(a)). Lemma 5.3 implies that there is a shortest non-contractible cycle in  $\Sigma'$  crossing  $\ell_x$  at most once. We compute the shortest cycle  $\gamma$  in  $Cross(\ell_x)$  using Lemma 5.1 and update  $minCycle$  with  $\gamma$ , if it is shorter. Then we set  $\Sigma' = \Sigma' \setminus \ell_x$ . Note that if  $\ell_x$  is separating, then  $\Sigma'$  has two connected components and  $Cross(\ell_x) = \emptyset$ . If  $\Sigma'$  has complexity  $m$ , we spend  $O(gm \log m)$  time in this iteration.
- (b) If  $\Sigma'$  has a component with non-zero genus and exactly one boundary  $\delta$ , we find a shortest non-contractible arc  $\alpha$  with endpoints in  $\delta$  (Lemma 5.7(c)). Lemma 5.5 implies that there is a shortest non-contractible cycle in  $\Sigma'$  that either crosses  $\alpha$  at most once or is homotopic to  $\delta$ . We compute the shortest cycle in  $Cross(\alpha)$  and the shortest cycle homotopic to  $\delta$  in  $O(gn \log n)$  time (Lemmas 5.1 and 5.7(b)). The shortest of these two cycles is compared against  $minCycle$ , and  $minCycle$  is updated if necessary. We then set  $\Sigma' = \Sigma' \setminus \alpha$ . If  $\Sigma'$  has complexity  $m$ , we spend  $O(gm \log m)$  time in this iteration.
- (c) If  $\Sigma'$  has one component with exactly two boundaries, we find a shortest arc  $\alpha$  connecting them (Lemma 5.7(d)). Lemma 5.6 implies that there is a shortest non-contractible cycle  $\ell$  in  $\Sigma'$  crossing  $\alpha$  at most once. We compute the shortest cycle  $\gamma$  in  $Cross(\alpha)$  using Lemma 5.1, and update  $minCycle$  with  $\gamma$ , if it is shorter. Then we replace  $\Sigma'$  with  $\Sigma' \setminus \alpha$ . If  $\Sigma'$  has complexity  $m$ , we spend  $O(gm \log m)$  time in this iteration.

This finishes the description of the algorithm. The algorithm has  $O(g)$  iterations; starting with no boundary, we iterate in case (a) once and then cases (b) or (c) at most  $2g$  times. The arcs and loops that are used to cut, which are obtained from Lemma 5.7, have multiplicity at most two and are edge-disjoint from the boundary. Thus any subsurface  $\Sigma'$  considered in a subproblem has at most four copies of an edge of  $\Sigma$ . This means that any surface  $\Sigma'$  has complexity  $O(n)$  and in each iteration we spend  $O(gn \log n)$  time.  $\square$

## References

- [1] Pankaj K. Agarwal, D. Eppstein, L. J. Guibas, and M. Henzinger. Parametric and kinetic minimum spanning trees. In *Proc. 39th Ann. IEEE Sympos. Found. Comput. Sci.*, pages 596–605, 1998.
- [2] Julian Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *J. Algorithms*, 31(1):1–28, 1999.
- [3] MohammadHossein Bateni, MohammadTaghi Hajiaghayi, and Dániel Marx. Approximation schemes for Steiner forest on planar graphs and graphs of bounded treewidth. *J. ACM*, 58:21, 2011.
- [4] Glencora Borradaile. *Exploiting Planarity for Network Flow and Connectivity Problems*. PhD thesis, Brown University, May 2008.
- [5] Glencora Borradaile, Erik D. Demaine, and Siamak Tazari. Polynomial-time approximation schemes for subset-connectivity problems in bounded-genus graphs. In *Proc. 26th Int. Symp. Theoretical Aspects Comput. Sci.*, pages 171–182. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.
- [6] Glencora Borradaile and Philip Klein. An  $O(n \log n)$  algorithm for maximum  $st$ -flow in a directed planar graph. *J. ACM*, 56(2): 9:1–30, 2009.
- [7] Sergio Cabello. Many distances in planar graphs. *Algorithmica*, 62(1-2):361–381, 2012.
- [8] Sergio Cabello and Erin W. Chambers. Multiple source shortest paths in a genus  $g$  graph. In *Proc. 18th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 89–97, 2007.
- [9] Sergio Cabello, Matt DeVos, Jeff Erickson, and Bojan Mohar. Finding one tight cycle. *ACM Transactions on Algorithms*, 6(4), 2010.
- [10] Sergio Cabello and Bojan Mohar. Finding shortest non-separating and non-contractible cycles for topologically embedded graphs. *Discrete Comput. Geom.*, 37(2):213–235, 2007.
- [11] Erin W. Chambers, Éric Colin de Verdière, Jeff Erickson, Francis Lazarus, and Kim Whittlesey. Splitting (complicated) surfaces is hard. *Comput. Geom. Theory Appl.*, 41(1–2):94–110, 2008.
- [12] Jianer Chen, Saroja P Kanchi, and Arkady Kanevsky. A note on approximating graph genus. *Inform. Proc. Lett.*, 61(6):317–322, 1997.
- [13] Éric Colin de Verdière. *Raccourcissement de courbes et décomposition de surfaces [Shortening of Curves and Decomposition of Surfaces]*. PhD thesis, University of Paris 7, December 2003.
- [14] Éric Colin de Verdière and Jeff Erickson. Tightening non-simple paths and cycles on surfaces. *SIAM J. Comput.*, 39(8):3784–3813, 2010.
- [15] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [16] James R. Driscoll, Niel Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. System Sci.*, 38(1):86–123, 1989.
- [17] Mark J. Eisner and Dennis G. Severance. Mathematical techniques for efficient record segmentation in large shared databases. *J. ACM*, 23(4):619–635, 1976.

- [18] David Eppstein. Dynamic generators of topologically embedded graphs. In *Proc. 14th Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 599–608, 2003.
- [19] David Eppstein and Kevin A. Wortman. Optimal embedding into star metrics. In *Proc. 11th Workshop on Algorithms and Data Structures*, volume 5664 of *Lecture Notes in Comput. Sci.*, pages 290–301, 2009.
- [20] Jeff Erickson. Parametric shortest paths and maximum flows in planar graphs. In *Proc. 21st Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 794–804, 2010.
- [21] Jeff Erickson. Shortest non-trivial cycles in directed surface graphs. In *Proc. 27th Ann. Symp. Comput. Geom.*, pages 236–243, 2011.
- [22] Jeff Erickson and Sarel Har-Peled. Optimally cutting a surface into a disk. *Discrete Comput. Geom.*, 31:37–59, 2004.
- [23] Jeff Erickson and Amir Nayyeri. Computing replacement paths in surface graphs. In *Proc. 22nd Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 1347–1354, 2011.
- [24] Jeff Erickson and Amir Nayyeri. Minimum cuts and shortest non-separating cycles via homology covers. In *Proc. 22nd Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 1166–1176, 2011.
- [25] Jeff Erickson and Pratik Worah. Computing the shortest essential cycle. *Discrete Comput. Geom.*, 44(4):912–930, 2010.
- [26] Lester R. Ford. Network flow theory. Paper P-923, The RAND Corporation, Santa Monica, California, August 14, 1956. Cited in [48].
- [27] Kyle Fox. Faster shortest non-contractible cycles in directed surface graphs. *CoRR*, abs/1111.6990, 2011.
- [28] Greg N. Frederickson. Fast algorithms for shortest paths in planar graphs with applications. *SIAM J. Comput.*, 16(6):1004–1004, 1987.
- [29] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.
- [30] S. Gass and T. Saaty. The computational algorithm for the parametric objective function. *Naval Research Logistics Quarterly*, 2:39–45, 1955. Cited in [41].
- [31] Andrew V. Goldberg, Michael D. Grigoriadis, and Robert E. Tarjan. Use of dynamic trees in a network simplex algorithm for the maximum flow problem. *Math. Program.*, 50:277–290, 1991.
- [32] Andrew V. Goldberg and Robert E. Tarjan. Finding minimum-cost circulations by cancelling negative cycles. *J. ACM*, 36(4):873–886, 1989.
- [33] Leonidas J. Guibas. Kinetic data structures — a state of the art report. In P. K. Agarwal, L. E. Kavradi, and M. Mason, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209. A. K. Peters, Wellesley, MA, 1998.
- [34] Dan Gusfield. Parametric combinatorial computing and a problem of program module distribution. *J. ACM*, 30(3):551–563, 1983.

- [35] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.
- [36] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.
- [37] Ken-ichi Kawarabayashi and Mikkel Thorup. The minimum  $k$ -way cut of bounded size is fixed-parameter tractable. In *Proc. 52nd Ann. IEEE Sympos. Found. Comput. Sci.*, pages 160–169, 2011.
- [38] Richard M. Karp and James B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Appl. Math.*, 3:37–45, 1981.
- [39] Ken-Ichi Kawarabayashi, Philip N. Klein, and Christian Sommer. Linear-space approximate distance oracles for planar, bounded-genus and minor-free graphs. In *Proc. 38th Int. Colloquium Conf. on Automata, Languages and Programming - Volume Part I*, volume 6755 of *Lecture Notes in Comput. Sci.*, pages 135–146. Springer-Verlag, 2011.
- [40] Ken-ichi Kawarabayashi, Bojan Mohar, and Bruce Reed. A simpler linear time algorithm for embedding graphs into an arbitrary surface and the genus of graphs of bounded tree-width. In *Proc. 49th IEEE Symp. Found. Comput. Sci.*, pages 771–780, 2008.
- [41] Victor Klee and Peter Kleinschmidt. Geometry of the Gass-Saaty parametric cost LP algorithm. *Discrete Comput. Geom.*, 5(1):13–26, 1990.
- [42] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proc. 16th Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 146–155, 2005.
- [43] Martin Kutz. Computing shortest non-trivial cycles on orientable surfaces of bounded genus in almost linear time. In *Proc. 22nd Ann. Symp. Computational Geometry*, pages 430–438, 2006.
- [44] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. *SIAM J. Applied Math.*, 36(2):177–189, 1979.
- [45] Bojan Mohar and Carsten Thomassen. *Graphs on Surfaces*. Johns Hopkins University Press, Baltimore, 2001.
- [46] Ketan Mulmuley, Umesh Vazirani, and Vijay Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.
- [47] Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM J. Comput.*, 27(4):972–992, 1998.
- [48] Alexander Schrijver. On the history of combinatorial optimization (till 1960). In K. Aardal, G.L. Nemhauser, and R. Weismantel, editors, *Handbook of Discrete Optimization*, pages 1–68. Elsevier, 2005.
- [49] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.
- [50] Robert E. Tarjan. Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem. *Math. Oper. Res.*, 16(2):272–291, 1991.
- [51] Robert E. Tarjan. Dynamic trees via Euler tours, applied to the network simplex algorithm. *Mathematical Programming: Series A and B*, 78:169–177, 1997.

- [52] Robert E. Tarjan and Renato F. Werneck. Self-adjusting top trees. In *Proc. 16th Ann. ACM-SIAM Symp. Discrete Algorithms*, pages 813–822, 2005.
- [53] Siamak Tazari and Matthias Müller-Hannemann. Shortest paths in linear time on minor-closed graph classes, with an application to steiner tree approximation. *Discrete Applied Mathematics*, 157(4):673–684, 2009.
- [54] Carsten Thomassen. The graph genus problem is NP-complete. *J. Algorithms*, 10(4):568–576, 1989.
- [55] Carsten Thomassen. Embeddings of graphs with no short noncontractible cycles. *J. Comb. Theory Ser. B*, 48(2):155–177, 1990.
- [56] Karl Georg Christian von Staudt. *Geometrie der Lage*. Verlag von Bauer and Rapse (Julius Merz), Nürnberg, 1847.
- [57] Neal E. Young, Robert E. Tarjan, and James B. Orlin. Faster parametric shortest path and minimum balance algorithms. *Networks*, 21(2):205–221, 1991.