Multiple Source Shortest Paths in a Genus g Graph

Sergio Cabello* Erin W. Chambers†

Abstract

We give an $O(g^2 n \log n)$ algorithm to represent the shortest path tree from all the vertices on a single specified face f in a genus g graph. From this representation, any query distance from a vertex in f can be obtained in $O(\log n)$ time. The algorithm uses a kinetic data structure, where the source of the tree iteratively moves across edges in f. In addition, we give applications using these shortest path trees in order to compute the shortest non-contractible cycle and the shortest non-separating cycle embedded on an orientable 2-manifold in $O(g^3 n \log n)$ time.

1 Introduction

A shortest path tree (sp-tree) in a graph is a tree containing shortest paths from some vertex to all other vertices. Sp-trees are a fundamental tool on graphs, and have applications for flows, distance queries, connectivity, and many other problems.

In the planar graph setting, there are many results for multiple source shortest paths. The result of primary interest for the purposes of this paper is Klein [9], who addressed the problem of maintaining the sp-tree in a plane graph as the source of the tree moves along some face in the graph. He gave an algorithm that represented the sp-tree for all vertices on a common face in the graph in $O(n \log n)$ time. Other noteworthy results on multiple source shortest paths include Frederickson's all-pairs shortest paths representation [6], Lipton and Tarjan's planar separator theorem [11], and Schmidt's $O(n \log n)$ algorithm which supports distance queries for specific subsets of vertices on a grid [13].

In this paper, we give an algorithm to maintain the sp-tree as the source of the tree moves around a face on a genus g graph, that is, a graph embedded in an orientable surface of genus g. This result extends Klein's result on plane graphs, and the running time to compute the sp-tree for every vertex on a given face becomes $O(g^2 n \log n)$.

In Klein's algorithm, the source of the shortest path tree is moved to a neighbor of a vertex, and then the sp-tree is updated edge by edge based on a set of candidate edges which should now be in the sp-tree. The next edge from the set is chosen based on what Klein refers to as the leafmost unrelaxed edge, which exploits the property that the edges not present in the sp-tree form a tree in the dual graph.

A main obstacle to extend this idea to genus g graphs is that the structure of the edges not present in the sp-tree is more complex, and in particular the concept of leafmost unrelaxed edge does not carry on. This makes harder to find out which edges have to be added and removed from the sp-tree, as well as in which order these operations should be done. Our approach consists of looking at the scenario like a kinetic sp-tree, where we maintain the sp-tree as the source moves continuously along an edge. Edges to add and remove from the sp-tree are discrete events which we compute during the course of this movement. (For more background on kinetic data structures in general, see [7].)

As applications, we use our algorithm to find in an embedded graph a shortest non-contractible in $O((g + b)g^2 n \log n)$ time, and a shortest non-separating cycle in $O(g^3 n \log n)$ time. Here b is the number of boundaries of the surface. The best previous results for this problem were $O(n^2 \log n)$ by Erickson and Har-Peled [5], $O(g^{3/2} n^{3/2} \log n)$ by Cabello and Mohar [1], and $O(g^{O(g)} n \log n)$ by Kutz [10]. Our approach improves them for a large range of values of g.

The main tool we use is the self-adjusting top tree [16]. A review of this data structure along with necessary topological background is provided in Section 2. Section 3 describe the maintenance of the shortest path tree when the root moves along an edge. First, we present the planar case to illustrate the necessary concepts, and then give the algorithm on a graph of genus g. Section 4 gives the analysis of the time used when the root moves along a face. Section 5 gives the algorithms to compute the shortest non-separating and non-contractible cycles on an embedded graph.

*Department of Mathematics, IMFM, and Department of Mathematics, FMF, University of Ljubljana, Slovenia, sergio.cabello@fmf.uni-lj.si. Partially supported by the European Community Sixth Framework Programme under a Marie Curie Intra-European Fellowship, and by the Slovenian Research Agency, project J1-7218.

†Department of Computer Science, University of Illinois, erinwolf@uiuc.edu. Research partially supported by an NSF Graduate Research Fellowship and by NSF grant DMS-0528086.

2 Background

2.1 Self-adjusting top trees. For our algorithm, we require a dynamic forest data structure supporting edge insertions and deletions. In addition, each vertex of the forest will maintain a number which is its distance to some vertex in the graph. Updates to this distance data must be supported on both subtrees and paths within the tree.

The data structure used here is the self adjusting top tree [16]. Operations used in our application are `join`, `cut`, and `expose`. We briefly describe these operations.

The operations $\texttt{join}(u, v)$ and $\texttt{cut}(u, v)$ respectively add an edge between u and v in different components of the forest and delete the edge between u and v. The operation $\texttt{expose}(u, v)$ makes the path between u and v in the forest (if it exists) the top-level path in the data structure, allowing quick updates and searches in this path. All of these operations can be supported in $O(\log n)$ amortized time.

2.2 Graphs. Throughout the paper, we work on a weighted, embedded graph $G = (V, E)$, where V is the vertex set and E is the edge set. The weight of an edge uv is denoted $w(uv)$. The *dual graph* of G, written G^*, consists of the graph given by making every face of G a vertex, and adding edges between adjacent faces. For $e \in E$, we use e^* to denote the edge in the dual graph which goes between the two faces that e borders.

As in Klein's paper, we define $d_T : V \to \mathbb{R}$ as the distance in a rooted directed tree T from the source of the tree to a given vertex. (We omit T when the tree is clear from context.) Define the *tension* of an edge $\overrightarrow{uv}$ as $t(\overrightarrow{uv}) = d(v) - w(\overrightarrow{uv}) - d(u)$ (for more examples, see [15]). We say the edge $\overrightarrow{uv}$ is *tense* if $t(\overrightarrow{uv}) > 0$. In other words, if $\overrightarrow{uv}$ is tense, there is a shorter path to v which uses a path in the tree to v plus the edge $\overrightarrow{uv}$. It is a simple exercise to verify if a tree T on G leaves no tense edges in $(E \setminus T)$, then T is a sp-tree.

For simplicity, *we assume that all shortest paths in G are unique.* As pointed out in [5], this condition can be obtained with high probability: the Isolation Lemma of Mulmuley, Vazirani, and Vazirani [12] implies that adding an infinitesimal weight $\varepsilon \cdot r(e)$ to each edge e, where $r(e)$ is chosen from $[1, 2, \ldots, n^2]$ uniformly at random, makes all shortest paths unique with probability at least $1 - 1/n$.

2.3 Topological background. We briefly present some concepts and definitions from algebraic and combinatorial topology. For more detailed background, see also [14] and [17].

A *surface*, or 2-manifold with boundary, is a topological Hausdorff space where each point has a neighborhood homeomorphic to either the plane (if the point is *interior*) or the closed half-plane (if the point is on the *boundary*). Surfaces considered here are connected and compact, and are orientable unless otherwise specified. Any connected compact orientable surface is homeomorphic to a sphere with g handles and b open disks removed. We say g is the *genus* of the surface and b is the number of *boundaries*.

Let M be a surface. A *curve* or *path* on M is a continuous map $p : [0, 1] \to M$. The *endpoints* are denoted $p(0)$ and $p(1)$. A loop with basepoint x is a path p with $x = p(0) = p(1)$. An *arc* α is a path whose endpoints are in the boundary. A *cycle* is a continuous map $\gamma : S^1 \to M$, where S^1 is the unit circle. A curve is *simple* if it is one-to-one. Two cycles are disjoint if the do not intersect, and two paths are disjoint if they intersect only at their endpoints.

A *homotopy* between two paths p and q, with $p(0) = q(0)$ and $p(1) = q(1)$, is a continuous map $H : [0, 1] \times [0, 1] \to M$ 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 cycles γ and β is a continuous map $H : [0, 1] \times S^1$ such that $H(0, \cdot) = \gamma$ and $H(1, \cdot) = \beta$. A cycle is *contractible* if it is homotopic to a constant cycle. An arc whose endpoints are in the same boundary component δ is non-contractible if after contracting δ to a point, the obtained loop is non-contractible. A simple cycle or arc is *separating* if $M \setminus \gamma$ has two connected components.

For the topological results in this paper, we work on the *combinatorial surface* model. This model has been commonly used for most related results [1], [2], [3], [5]. A combinatorial surface is an abstract surface M together with a weighted undirected graph G embedded on M such that each open face is a disk. Allowed paths are walks in G, and the length of a path is the sum of the weights of the edges on the path (with multiplicity). The multiplicity of a path is the maximum number of times that an edge appears in it. The complexity of the combinatorial surface, usually denoted by n, is the number of vertices, edges, and faces in the embedding of G.

In a combinatorial surface, a curve or cycle in M is simple if it can be infinitesimally perturbed to a simple curve in M. Any arrangement of curves in the surface can be seen as an embedded graph [3], which basically handles this by increasing the number of vertices and edges appropriately to obtain a graph G' where the curves are edge-disjoint. This allows us to assume that, in the surface, all the curves considered are generically embedded.

3 Updating the sp-tree

Suppose that we are given the sp-tree rooted at some vertex u. We wish update the tree so that it becomes the shortest path rooted at v, a neighbor of u. View the sp-tree as a kinetic data structure, in which the root s slides continuously from u to v along the edge $e = \overrightarrow{uv}$. Any vertex whose shortest path to s goes through v immediately before reaching s is called a *blue* vertex. All other vertices are colored *red*. Note that in the sp-tree rooted at s, the vertices of each color form a subtree of the sp-tree, since any vertex's shortest path to s must go through either u or v. The blue vertices are in some sense already in the final sp-tree, since their shortest paths to s do not change as s moves closer to v. Eventually, when s arrives at v, the sp-tree rooted at v is obtained.

As s slides across e a distance Δ, the distance from s to every red vertex increases by Δ, and the distance from s to every blue vertex decreases by Δ. When an edge $\overrightarrow{xy}$ is about to become tense (i.e. if $t(\overrightarrow{xy})$ was negative and just became $= 0$), then $d(y) = d(x)+w(\overrightarrow{xy})$ for some red vertex y and blue vertex x. This means that as s continues to move along e, the path from s to y that goes through x is to be shorter than the previous shortest path.

When an edge becomes tense, we have an *event* in the sp-tree. In an event, the edge immediately preceding y along its shortest path to s should be deleted, and the edge $\overrightarrow{xy}$ is added to the sp-tree instead. Additionally, y and all the vertices in its subtree should be recolored to blue, since the shortest path to s now passes through v last.

The kinetic property we exploit here is that we have no extraneous events. Each time an edge becomes tense, we must do exactly one cut and one join in the sp-tree. Using appropriate data structures, a running time of $O(\log n)$ per update to the sp-tree is obtained. The main issue becomes detecting what edges come in and out in the tree.

In the algorithm, we use two types of data structures. The first holds the sp-tree T, and the second maintains a subgraph of the dual graph $(G \setminus T)^*$. The main difference between the algorithms for the planar case and the genus g case is the dual structure. In a planar graph, the duals of edges not contained in the sp-tree form the so-called *cotree*, a spanning tree of the dual graph. However, in a genus g graph, there are $O(g)$ extra edges that ruin this cotree structure and make the algorithm more complicated.

In the primal structure, we have a directed forest that is a subgraph of the current sp-tree. Each vertex implicitly maintains its distance to s. Note that all of the following operations can be implemented in $O(\log n)$ (worst-case) time using Euler-Tour trees [8].

- `GetValue`(v) returns $d(v)$.
- `Cut`(e') removes the specified edge e' from the current forest.
- `Join`(e') adds the edge e' to the forest. (This operation assumes the forest remains acyclic after adding e.)
- `AddSubtree`(Δ, x) adds the value Δ to every vertex in the subtree rooted at vertex x.

In the dual graph, we need a different set of operations. These differ in the planar and non-planar cases, and will be described in more detail in the appropriate sections.

3.1 Planar graphs. We begin with the algorithm in the planar case. This algorithm is similar to Klein's [9] in that the same tree and cotree decomposition is used, and the basic idea of relaxing edges is the same. However, we relax edges in our kinetic data structure one by one as the source moves along an edge, instead of changing the root and then relaxing edges iteratively from a set of tense edges as Klein does.

Our algorithm begins with a sp-tree T rooted at a vertex $s = u$, and moves the source s of the sp-tree T across a fixed edge $e = \overrightarrow{uv}$. In the dual graph, each edge xy^* dual to an edge xy that is not in T stores $t(\overrightarrow{xy})$ and $t(\overrightarrow{yx})$. This set of edges in $(G \setminus T)^*$ forms a spanning tree. We use the following operations, each of which can be performed in $O(\log n)$ amortized time using self adjusting top trees [16].

- `Cut`(e) cuts the edge e from the tree, and returns two new separate trees.
- `Join`(e) adds the edge e, joining two trees into a single tree.
- `AddPath`(Δ, π) adds $\pm\Delta$ to the tension of each of the edges in the path π. (The value $+\Delta$ is added to the tension of the edge oriented from red to blue vertex, and $-\Delta$ is added to the orientation from blue to red.)
- `MaxPath`(π) finds the edge in the path π that has maximum tension (where the tension is oriented from a blue vertex to a red vertex).

Assume that e is in the sp-tree. Any (primal) edge whose tension is changing must have one blue endpoint and one red endpoint, since any edge with monochromatic endpoints has constant tension (its endpoints

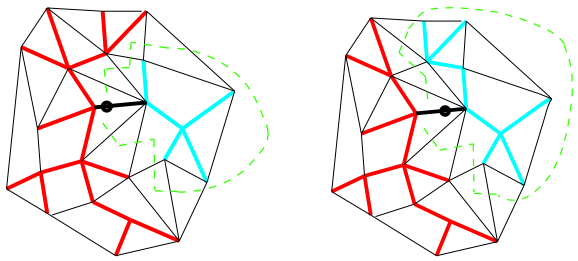

Figure 1. The thick lines in the graph are the sp-tree rooted and the source, shown moving along the edge e. As s moves closer to the target, an edge along the green path in the dual (shown in dotted lines) becomes tense and is inserted.

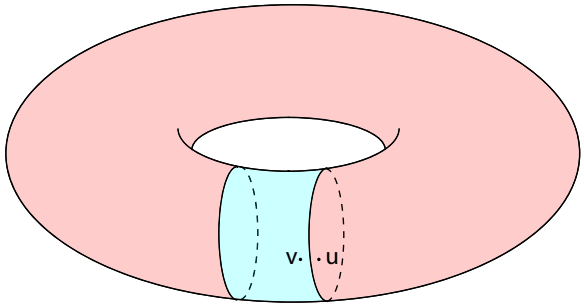

Figure 2. As the blue tree expands, the set of green edges could intersect itself and separate into multiple components. Here, the blue subtree (rooted at v) wrapped around the torus, and the set of green edges is shown as two disconnected cycles in the dual which separate the red subtree from the blue subtree.

change at the same rate). This means that the set of edges whose tension is changing corresponds to the unique *path* in T^* that goes between the endpoints of e^*. Call this set of edges *green*, and let the dual path consisting of all the green edges be π^*.

If the edge e is not in the sp-tree, all vertices are red. At some stage e becomes tense, since the distance from s to v along e is going to zero as s slides along e. At this stage, the edge e enters the sp-tree, and the subtree rooted at v is immediately colored blue.

Once e is in the sp-tree, the algorithm iterates over the following steps until either every vertex is blue or s reaches v. See Figure 1. We call `MaxPath` on the two endpoints of e^* to find the first edge $\overrightarrow{xy}$ that has $t(\overrightarrow{xy}) > 0$ as u slides towards v. Let Δ be the amount the root needs to slide for $\overrightarrow{xy}$ to become tense.

Next, move the source s a distance Δ along the edge e and update the distances in the primal tree and tension in the dual tree. `AddPath`$(\pi^*, 2\Delta)$ updates the tensions of the green edges. In the primal tree, `AddSubtree`(u, Δ) adds Δ to every distance for vertices in the red subtree, simulating the root sliding along the edge e by a distance of Δ. Similarly, `AddSubtree`$(v, -\Delta)$ updates the distance from s to the blue vertices.

Now we must actually update the sp-tree to reflect the new edge being added. We first `Cut`(wy) where $\overrightarrow{wy}$ is the last edge in the path in T from s to y. Then `Join`$(\overrightarrow{xy})$ reconnects s to y via the new shorter path. This reconnects the sp-tree, and recolors y and its subtree blue.

To update the dual spanning tree, we call `Cut`(xy^*) to remove the dual edge whose tension is no longer changing, and then `Join`(wy^*) to reconnect the dual tree and update the structure.

Each edge change in the tree calls a constant number of operations taking $O(\log n)$ time. This concludes the description of the algorithm to move the source along a single edge. We summarize in the following lemma:

Lemma 3.1. *Sp-trees in planar graphs can be represented in such a way that the sp-tree rooted at u can be changed to the sp-tree rooted at a neighbor of u in $O(k \log n)$ time, where k is the number of edges entering or leaving the trees. In this representation, a shortest path distance from the root can be computed in $O(\log n)$ time.* □

3.2 Graphs of genus g. The algorithm in the primal case does not immediately extend to genus g graphs, since the dual graph $(G \setminus T)^*$ is no longer a tree. Without this tree structure, the set of green edges can become more complicated than a path in the dual, and finding tense edges is harder. For example, consider the case when the graph is on the torus. Initially, the blue subtree is divided from the red subtree by a cycle of green edges. However, as the blue subtree grows, it is possible for the blue to meet at other places, and the green boundary splits into different connected components. See Figure 2.

Our algorithm for graphs of genus g is identical to the planar algorithm on the primal data structure. However, the dual structure is necessarily more complicated, since we no longer have a tree in the dual. (Throughout the algorithm, we consider the edge $(uv)^*$ as part of $(G \setminus T)^*$ to simplify the algorithm.) For each dual edge xy^* corresponding to an edge xy that is not in T, we maintain the tenseness of $\overrightarrow{xy}$ and $\overrightarrow{yx}$.

Decompose the edges of $(G \setminus T)^* \cup (uv)^*$ as follows. Iteratively delete all edges of degree 1. Denote the deleted edges as F^*, and note that F^* is a forest in G^*. After the edges F^* have been deleted, we have a set of paths, called *cut paths*, that meet at vertices of degree ≥ 3. Denote these cut paths as $P^* = \{\pi_1, \pi_2, \ldots, \pi_k\}$, where each π_i is a path between two vertices of degree ≥ 3. Euler's formula implies that $k = O(g)$ (see [5, Lemma 4.2]).

The set of green edges, or primal edges whose endpoints are not monochromatic, are again those edges that could potentially become tense. All the green edges are dual to edges in P^*. To see this, note that P^* bounds two topological disks, one containing all the blue vertices and the other containing all the red vertices. For any edge $xy^* \in F^*$, the edge xy cannot have endpoints of different colors, since its endpoints are in the same face. Therefore, P^* must contain the dual of every green edge. Moreover, any path $\pi_i \in P^*$ has all the edges dual to either monochromatic edges or to green edges.

Each path π_i, along with any components of F^* which are rooted along π_i, is put into a self adjusting top tree T_i, where π_i is the top level path. We also maintain two linked lists of pointers representing the current boundary of the red and blue faces. For each π_i on the boundary of a face, a pointer in the linked list for that face points to the corresponding tree T_i. These pointers appear in the linked list in the same order that the paths appear along the boundary. Note that it is possible for a path to bound the same region on both sides; the linked list will simply contain two pointers to the same tree. See Figure 3.

The operations on the dual structure are:

1. `Cut`(e) and `Join`(e) are identical to the planar operations.
2. `AddBoundary`(Δ) adds $\pm\Delta$ to the tension of every edge that borders both the red and the blue faces. (Again, we add $+\Delta$ to the orientation going from red to blue, and $-\Delta$ to the orientation from blue to red.)
3. `MinBoundary` returns the edge with maximum tenseness among edges that border both faces..

To implement `MinBoundary`, we search both linked lists in $O(g)$ time to find the cut paths that border both regions; this is the candidate list of green edges. We can find the edge in each cut path with highest tension in $O(\log n)$ time by querying the appropriate T_i with `MinPath`. Thus, it takes $O(g \log n)$ time to find the edge $\overrightarrow{xy}$ that becomes tense first.

To implement `AddBoundary`, we again search both linked lists in order to find the cut paths that contain green edges. These are the only trees that need to be updated, since edges with monochromatic endpoints have constant tension. Each of the self adjusting top trees can update tensions in $O(\log n)$ time using `AddPath`, giving a bound of $O(g \log n)$ time for updating all of the cut paths.

Now we describe the algorithm. We first call `MinBoundary` to find the first tense edge $\overrightarrow{xy}$. Let Δ be the amount that s must move along e in order for $\overrightarrow{xy}$ to become tense.

In the primal sp-tree, we can update the distances using `AddSubtree`(Δ) to u's subtree (the red vertices), and `AddSubtree`$(-\Delta)$ to v's subtree (the blue vertices). We then call `AddBoundary` with a value of 2Δ. We also need to update the sp-tree. This is done using `Cut`$(\overrightarrow{wy})$ to cut y out of the red tree, where $\overrightarrow{wy}$ is the red edge that connects y to the s in the sp-tree. Then `Join`$(\overrightarrow{xy})$ reconnects y and its subtree to the source s.

We also have to update the dual structure. The dual edge $\overrightarrow{yw}^*$ is now a green edge, since w is still red and y is now blue. The endpoints of $\overrightarrow{wy}^*$ are either in F^* or are directly on some path π_j, since we know that wy was in the sp-tree. Find the unique (and possibly empty, if the vertex is on some π_j) path in F^* connecting each endpoint of $\overrightarrow{wy}^*$ to the paths P^* in $O(\log n)$ by simply splaying up the self adjusting top tree that they belong to until reaching the top level path. Denote these (possibly empty) paths as π' and π''.

We call `expose` to make π and π' the top level in their respective trees, and then we combine them into one path π using `Join`$(\overrightarrow{wy}^*)$. We now insert π into the boundaries of the two faces in our linked lists. In addition, the path π_i that $\overrightarrow{xy}$ cut across is no longer a boundary, so we must update the tree for that component and remove it from the boundary lists. It is also possible that π intersects one or two paths π_j; if so, those trees must be subdivided at the appropriate location using `Cut`.

We update the red face's linked list first by cutting (if necessary) the paths that the endpoints of π subdivided. We then remove all of the pointers in the linked list that were between those endpoints, since they gave the boundary of the subtree rooted at y that is now blue and therefore it is no longer a boundary. Note that this automatically removes the old path π_i from the boundary, since π necessarily either blocks or divides that path.

For the blue face's linked list, we must add in any of the paths π_k that were removed from the red list. We also add in the self adjusting top tree for π, since it now borders the blue face.

Altogether, updating the linked lists can be done in $O(g)$ time, since we traverse each list at most twice while cutting out and adding in links to the appropriate cut paths.

This finishes the description of the algorithm when moving along one edge. We summarize in the following lemma:

Lemma 3.2. *Sp-trees in graphs on a surface of genus g can be represented in such a way that the sp-tree rooted at u can be changed to the sp-tree rooted at a neighbor*

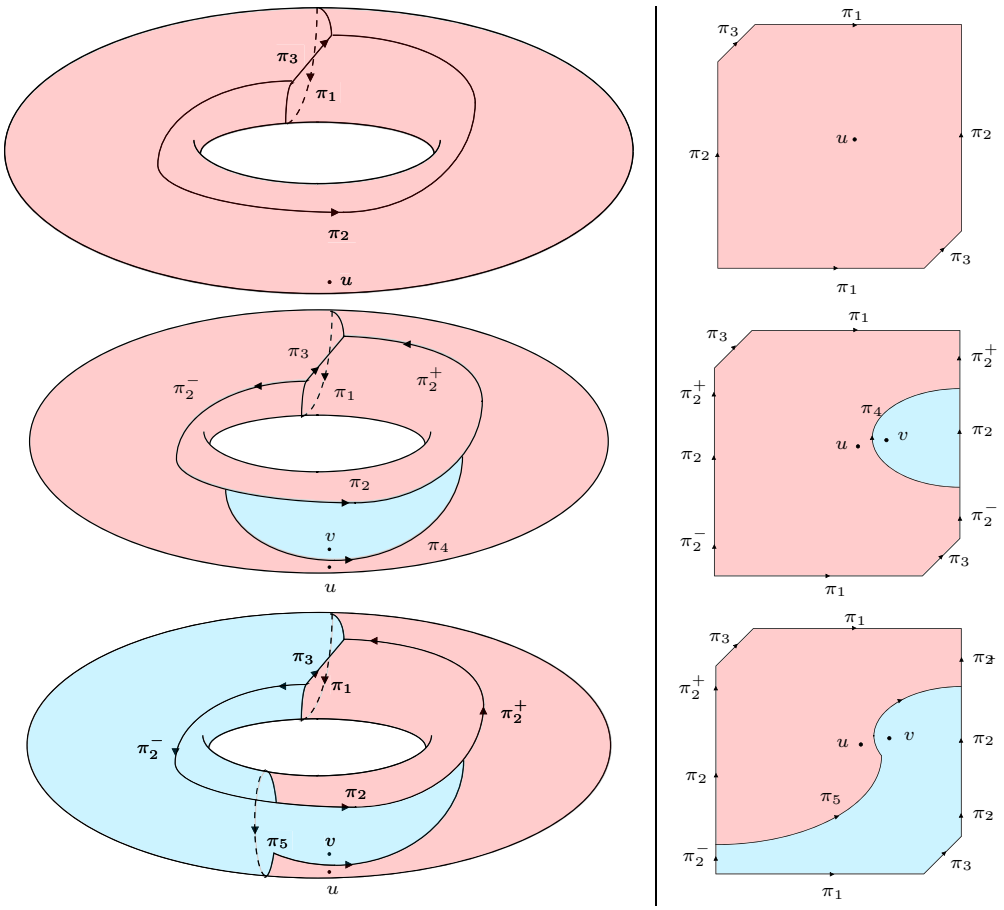

Figure 3. An example of the algorithm progressing. On the left, the alterations are shown on the surface on the torus, and on the right, they are shown on the polygonal schema obtained by cutting along the initial cut locus of u. As new edges become tense, the set of cut paths alters, but always separates the blue subtree from the red subtree.

of u in $O(kg \log n)$ time, where k is the number of edges entering or leaving the trees. In this representation, a shortest path distance from the root can be computed in $O(\log n)$ time. □

4 Moving the root along a face

4.1 Planar graphs. Consider the case when G is a plane graph, and we want to maintain the sp-tree as the root moves along a face f. Klein [9] noted two key facts. First, each edge can enter and leave the tree a constant number of times. For graphs with unique shortest paths, this can be seen via a relatively simply crossing argument. (Klein shows this also in general if one maintains a left-most sp-tree.) Also, using standard techniques, the graph can be assumed to have bounded degree, and then one can use persistence [4] to store and search any previous versions of the sp-tree. (This increases the required space to $O(n \log n)$.) With these two observations and Lemma 3.1, the following result is obtained.

Theorem 4.1 (Klein [9]). *Let G be a plane graph with n vertices, and let be f a face of G. With $O(n \log n)$ preprocessing time, a shortest path distance from any vertex on f to any other vertex can be found in $O(\log n)$ time.*

4.2 Graphs of genus g. Let G be a graph embedded in a surface of genus g. We are interested on maintaining the sp-tree as the root moves along a face f. We first show a bound on the number of times that an edge can enter or leave the sp-tree.

Lemma 4.1. *As the root of the sp-tree moves along f, each edge enters and leaves the sp-tree $O(g)$ times.*

Proof: Let e be an edge in G, and let N be the surface obtained by contracting e to a point e^* and f to a point f^*. Consider all the shortest paths from vertices in f to any of the endpoints of e. In N, these shortest paths define $O(g)$ distinct homotopy classes of paths with endpoints e^* and f^*. This follows from [2,

Lemma 2.1], since we can combine pairs of pairwise non-homotopic paths to get a set of pairwise non-crossing, non-homotopic loops with basepoint f^*.

If we consider all the shortest paths from vertices on f to an endpoint of an edge e, we know that $e = uv$ will enter or leave the sp-tree when the shortest path from $w \in f$ to v uses e and the shortest path from the neighbor of w on f does not use e. In a homotopy class, the edge e cannot switch from being in and out of the shortest paths more than constant number of times, since a homotopy class of paths forms a planar graph.

Note that the homotopy classes cannot interleave, that is, all the shortest paths that have the same homotopy class in N appear consecutively as we walk along f. Indeed, any two paths in the same homotopy class, together with a piece f' of f and e define a topological disk, and it cannot be that a path in a different homotopy class has an endpoint in the piece f'.

Since there are at most $O(g)$ homotopy classes, each of which occurs as a block, and in each class, an edge comes in and out $O(1)$ times, the result follows. □

Since we take $O(g \log n)$ time each time we update the tree, this gives $O(g^2 n \log n)$ time total to update all the trees over the course of the algorithm. Again, we can use standard techniques to convert to a graph with bounded degree, and use persistence [4] to store and search any previous versions of the sp-tree. Regarding the space bounds, during the process, we need $O(gn)$ space to maintain the (dual) structures. However, for answering distance queries, we only need to store the primal structure, and therefore, the final data structure uses $O(gn \log n)$ space. With these two observations and Lemma 3.2, we obtain our main result.

Theorem 4.2. *Let G be a graph with n vertices embedded in a surface of genus g, and let be f a face of G. With $O(g^2 n \log n)$ preprocessing time, a shortest path distance from any vertex on f to any other vertex can be found in $O(\log n)$ time.*

5 Computing shortest non-separating and non-contractible cycles

In this section we consider the problems of finding a shortest non-separating and a shortest non-contractible cycle in an orientable combinatorial surface M. The use of our technique on maintaining shortest path trees is condensed in the following lemma.

Lemma 5.1. *Let α be a simple cycle or arc in M, and let $Cross_1(\alpha)$ be the set of cycles that cross α exactly once. A shortest cycle in $Cross_1(\alpha)$ can be obtained in $O(g^2 n \log n)$ time.*

Proof: Consider the surface obtained by cutting M along α: each vertex v in α gives rise to two vertices v', v'', and two boundary arcs or cycles α', α''. Let N be the surface obtained by gluing topological disks to the boundaries that contain α' and α''. (If α is an arc, then α' and α'' are contained in a single boundary.) A cycle in M that crosses α once at a point v becomes a path in N connecting v' to v''. Thus, a shortest cycle that crosses α once at v is a shortest path that connects v' to v'' in N, and vice versa. Since all the points v', with $v \in \alpha$, belong to a face of N, we can use Theorem 4.2 to find in $O(g^2 n \log n)$ time a closest pair (v'_0, v''_0). Computing the shortest path from v'_0 to v''_0 gives the result. □

Note that if α is separating, then $Cross_1(\alpha) = \emptyset$ because any cycle crosses α an even number of times. We also use that *all* the cycles in $Cross_1(\alpha)$ are non-contractible. This is due to the fact that a contractible cycle C is a separating cycle, and any cycle or arc must cross C an even number of times.

5.1 Shortest non-separating cycle. A shortest non-separating cycle in M is also non-separating in the surface obtained by attaching disks to the boundaries. Therefore, we only need to consider surfaces without boundary.

Cabello and Mohar [1] have shown how to construct in $O(gn \log n)$ time a set S of $O(g)$ simple loops with the property that a shortest cycle in $\bigcup_{\ell \in S} Cross_1(\ell)$ is a shortest non-separating cycle. Since S consists of $O(g)$ loops, we can apply the previous lemma to $Cross_1(\ell)$ for each ℓ in S and take the globally shortest cycle. We summarize:

Theorem 5.1. *Let M be an orientable surface, possibly with boundary, of complexity n and genus g. We can find a shortest non-separating cycle in $O(g^3 n \log n)$ time.*

5.2 Shortest non-contractible cycle. The main approach is to find a curve whose removal decreases the genus or number of boundaries of the surface, and moreover, it has the property that there is a shortest non-contractible cycle intersecting it at most once. The main tool to prove the following results is an exchange argument. We omit their proof.

Lemma 5.2. *Let M be an orientable surface.*

(a) *Let ℓ_x be a shortest non-contractible loop with given basepoint $x \in M$. There is a shortest non-contractible cycle in M that crosses ℓ_x at most once.*

(b) *Let δ be a boundary cycle in M and let α be a shortest non-contractible arc with endpoints in δ. There is a shortest non-contractible cycle in M that is homotopic to δ, or that crosses α at most once.*

(c) *Let α be a shortest arc connecting two different specified boundaries of M. There is a shortest non-contractible cycle in M that crosses α at most once.*

The next result summarizes the time needed to find the curves described in the previous lemma.

Lemma 5.3. *Let M be an orientable combinatorial surface of complexity n.*

(a) *Given a basepoint x, we can find in $O(n \log n)$ time a shortest non-contractible loop with basepoint x and multiplicity two.*

(b) *Given a boundary δ, we can find in $O((g+b)n \log n)$ time a shortest cycle homotopic to δ.*

(c) *Given a boundary δ, we can find in $O(n \log n)$ time a shortest non-contractible arc with endpoints in δ, multiplicity two, and edge-disjoint from δ.*

(d) *If M has two or more boundaries, we can find in $O(n \log n)$ time a shortest arc with endpoints in different boundaries, multiplicity one, and edge-disjoint from all boundaries of M.*

Proof: (a) See Erickson and Har-Peled [5, Lemma 5.2].

(b) Construct a cross-metric surface N using a cylinder with a boundary ∂ having 3 edges and another boundary ∂' that gets glued to δ. Assign infinite length to each edge not in ∂'. Since N has complexity $O(n)$ and ∂ consists of three edges, an algorithm of Colin de Verdière and Erickson [3, Theorem 6.1] finds in $O((g + b)n \log n)$ time a shortest cycle homotopic to ∂ in N. This is the desired cycle in M.

(c) Contract δ to a point p and construct a shortest non-contractible loop with basepoint p as in item (a). This is the desired arc in M.

(d) Select a boundary δ of M, contract it to a point p, and construct in $O(n \log n)$ time a shortest path tree from p. Let q be the closest point to p among vertices v in the boundary of M. The shortest path from p to q in M is the shortest arc connecting the boundary δ to any other boundary, and it does not use any edge from any boundary.

□

Theorem 5.2. *Let M be an orientable surface of complexity n, genus g, and b boundaries. We can find a shortest non-separating cycle of M in $O((g+b)g^2 n \log n)$ time.*

Proof: We give a recursive algorithm that reduces either the genus or the number of boundaries of the subproblems. The recursion stops when the subproblem is a topological disk. We distinguish three cases; N denotes the surface in the subproblem and m its complexity. The genus of N is bounded by g.

(a) If N is a surface without boundary, we choose a point $x \in N$ and find a shortest non-contractible loop ℓ_x through x. Because of Lemma 5.2(a), there is a shortest non-contractible cycle in N that crosses ℓ_x at most once. Consider the surface $N' = N \backslash \ell_x$, and return a shortest cycle among: the cycles $Cross_1(\ell_x)$ and the shortest non-contractible cycle in N'. Note that if ℓ_x is separating, then N' has two connected components and $Cross_1(\ell_x) = \emptyset$. From Lemmas 5.1 and 5.3(a), it follows that we spend $O(g^2 m \log m)$ time in N, plus the time used for N'.

(b) If N has exactly one boundary δ, we find a shortest non-contractible arc α with endpoints in δ. Because of Lemma 5.2(b), there is a shortest non-contractible cycle in N that crosses α at most once, or it is homotopic to δ. Consider the surface $N' = N \setminus \ell_x$, and return a shortest cycle among: the cycles $Cross_1(\ell_x)$, a shortest cycle homotopic to δ, and a shortest non-contractible cycle in N'. From Lemmas 5.1 and 5.3(b-c), it follows that we spend $O(g^2 m \log m)$ time in N, plus the time used for N'.

(c) If N has two or more boundaries, we find a shortest arc α connecting two different boundaries. Because of Lemma 5.2(c), there is a shortest non-contractible cycle in N that crosses α at most once. Consider the surface $N' = N \setminus \ell_x$, and return a shortest cycle among: the cycles $Cross_1(\ell_x)$ and a shortest non-contractible in N'. From Lemmas 5.1 and 5.3(d), it follows that we spend $O(g^2 m \log m)$ time in N, plus the time used for N'.

This finishes the description of the algorithm. Note that at most $O(g + b)$ recursive subproblems are considered: starting from a surface M, we encounter $O(b+1)$ times cases (a) or (c), until we first obtain pieces with one boundary, and then we repeatedly encounter $O(g)$ cases (b-c) that maintain each component with one or two boundaries and decrease their genus. The arcs and loops that are used for cutting can be obtained from

Lemma 5.3, which have multiplicity at most two and are edge-disjoint from the boundary. It follows that any surface considered in a subproblem has at most four copies of an edge of M. Thus, $m = O(n)$, and in each piece we spend $O(g^2 n \log n)$ time. □

References

[1] S. Cabello and B. Mohar. Finding shortest non-separating and non-contractible cycles for topologically embedded graphs. In G. S. Brodal and S Leonardi, editors, *Algorithms - ESA 2005, 13th Annual European Symposium, Palma de Mallorca, Spain, October 3-6, 2005, Proceedings*, volume 3669 of *LNCS*, pages 131–142. Springer, 2005.

[2] E. W. Chambers, É. Colin de Verdière, J. Erickson, F. Lazarus, and K. Whittlesey. Splitting (complicated) surfaces is hard. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 421–429, New York, NY, USA, 2006. ACM Press.

[3] É. Colin de Verdière and J. Erickson. Tightening non-simple paths and cycles on surfaces. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 192–201, New York, NY, USA, 2006. ACM Press.

[4] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM Press.

[5] J. Erickson and S. Har-Peled. Optimally cutting a surface into a disk. *Discrete and Comput. Geometry*, 31(1):37–59, 2004.

[6] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1987.

[7] L. Guibas. Kinetic data structures: A state of the art report, 1998.

[8] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999.

[9] P. N. Klein. Multiple-source shortest paths in planar graphs. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 146–155, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[10] M. Kutz. Computing shortest non-trivial cycles on orientable surfaces of bounded genus in almost linear time. In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 430–438, New York, NY, USA, 2006. ACM Press.

[11] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

[12] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–114, 1987.

[13] J. 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.

[14] J. Stillwell. *Classical Topology and Combinatorial Group Theory*. Springer-Verlag, New York, 1993.

[15] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, Philadelphia, 1983.

[16] R. E. Tarjan and R. F. Werneck. Self-adjusting top trees. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 813–822, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[17] A. Zomorodian. *Topology for Computing*. Cambridge University Press, 2005.