Dynamic Ray Shooting and Shortest Paths in Planar Subdivisions via Balanced Geodesic Triangulations[*]

Michael T. Goodrich[†]
Dept. of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
`goodrich@cs.jhu.edu`

Roberto Tamassia[‡]
Dept. of Computer Science
Brown University
Providence, RI 02912–1910
`rt@cs.brown.edu`

Abstract

We give new methods for maintaining a data structure that supports ray shooting and shortest path queries in a dynamically-changing connected planar subdivision $\mathcal{S}$. Our approach is based on a new dynamic method for maintaining a balanced decomposition of a simple polygon via geodesic triangles. We maintain such triangulations by viewing their dual trees as balanced trees. We show that rotations in these trees can be implemented via a simple "diagonal swapping" operation performed on the corresponding geodesic triangles, and that edge insertion and deletion can be implemented on these trees using operations akin to the standard *split* and *splice* operations. We also maintain a dynamic point location structure on the geodesic triangulation, so that we may implement ray shooting queries by first locating the ray's endpoint and then walking along the ray from geodesic triangle to geodesic triangle until we hit the boundary of some region of $\mathcal{S}$. The shortest path between two points in the same region is obtained by locating the two points and then walking from geodesic triangle to geodesic triangle either following a boundary or taking a shortcut through a common tangent. Our data structure uses $O(n)$ space and supports queries and updates in $O(\log^2 n)$ worst-case time, where n is the current size of $\mathcal{S}$. It outperforms the previous best data structure for this problem by a $\log n$ factor in all the complexity measures (space, query times, and update times).

Keywords: Computational geometry, ray shooting, shortest path, point location, planar subdivision, polygon triangulation, dynamic data structure, on-line algorithm.

[*]This research was announced in preliminary form in the *9th ACM Symp. on Computational Geometry*, San Diego, May 19–21, 1993.

[†]Work by this author was supported in part by the National Science Foundation under grants CCR-9003299 and CCR-9300079, and by the NSF and DARPA under grant CCR-8908092.

[‡]Work by this author was supported in part by the National Science Foundation under grants CCR-9007851 and CCR-9423847, by the U.S. Army Research Office under grants DAAL03-91-G-0035 and 34990-MA-MUR, and by by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-91-J-4052, ARPA order 8225.

1 Introduction

An exciting trend in algorithmic research has been to show how one can efficiently maintain various properties of a collection of geometric objects while updating that structure in a dynamic fashion (e.g., see the survey on dynamic algorithms in computational geometry given by Chiang and Tamassia [6]). The main objective in this research is to design space-efficient data structures that can, for a particular collection of geometric objects, quickly process each operation to be performed, be it an insertion or deletion of a geometric object into the collection, or a query requesting combinatoric or geometric information about the collection. Examples of such problems include dynamic convex hull maintenance, dynamic point location, and dynamic range searching.

Let L be a set of line segments in the plane, and let $\mathcal{S}$ be the subdivision of the plane defined by $\mathbb{R}^2 \backslash L$. Suppose further that $\mathcal{S}$ is a *connected planar* subdivision, i.e., when joined at overlapping endpoints, the segments of L form a connected planar graph, which would occur, for example, if $\mathcal{S}$ were a Voronoi diagram or Delaunay triangulation (e.g., see [8, 21, 22]). Note that $\mathcal{S}$ has at least one unbounded region, and, without loss of generality we can assume that there is just one such external region, for otherwise we can "clip" all the unbounded regions by a large-enough bounding box. Moreover, by viewing each line segment in L as actually being two segments—a left segment and a right segment—then we can view each face of $\mathcal{S}$ as a simple polygon. Indeed, we can view $\mathcal{S}$ as being the union of a set of simple polygons. We assume a standard representation for the subdivision $\mathcal{S}$, such as doubly-connected edge lists [22], with this double-sided view. This representation allows us to identify, for any vertex v, the segment incident upon v, it allows us to identify for any (double) segment e, the face(s) on the two sides of e, and it allows us to identify, for any face f, a counterclockwise listing of the vertices and segments that form f's boundary.

There are several useful queries that one may wish to perform on $\mathcal{S}$, including the following:

- *point-location query*: given a point p in the plane, return the region in $\mathcal{S}$ containing p (or an $O(1)$ representation of the region containing p),
- *ray-shooting query*: given a ray $\vec{r}$, determine the first segment in L intersected by $\vec{r}$,
- *shortest-path query*: given two points p and q belonging to the same region in $\mathcal{S}$, find the shortest polygonal chain joining p and q that does not cross any segment of L.

In addition to being useful in computer graphics applications (e.g., see [23]), ray-shooting queries are in some sense generalizations of point-location queries, for point locations can easily be implemented using ray-shooting queries.

The specific dynamic computational geometry problem we address in this paper is to maintain a connected planar subdivision $\mathcal{S}$ subject to insertion and deletion of vertices and edges, and to ray shooting and shortest path queries. From now on, we denote with n the current size of $\mathcal{S}$.

1.1 Previous work

In the static setting, there are several optimal techniques for efficiently performing shortest-path and ray-shooting queries [1, 3, 4, 11, 12, 19], even in parallel [10, 15]. In particular, the data structures of Chazelle and Guibas [3] and of Guibas and Hershberger [11, 14] support respectively ray-shooting and shortest path queries in simple polygons in $O(\log n)$ time using $O(n)$ space. More recently, Chazelle *et al.* [2] give an elegant scheme for building a static ray shooting data structure that uses $O(n)$ space and answers ray shooting queries in $O(\log^2 n)$ time. Their method is based upon a decomposition of a simple polygon into "geodesic triangles" so as to allow a simple "walk-through" strategy for answering ray-shooting queries. They also show how to apply some more-sophisticated data structuring techniques to achieve an $O(\log n)$ query time using only a constant factor more space. Hershberger and Suri [16] further show how to achieve an $O(\log n)$ query time using nothing more than this walk-through strategy in an $O(n)$-sized triangulation of the interior P, although they may possibly introduce triangulation vertices (called *Steiner points*) that are not vertices of P (and this is necessary in some cases).

In the dynamic setting, the best result to date for connected subdivisions is the data structure of Chiang, Preparata, and Tamassia [5], which uses $O(n \log n)$ space and supports ray-shooting queries, shortest path queries, and insertion and deletion of vertices and edges in $O(\log^3 n)$ time (amortized for vertex updates). We also mention that the hidden-surface removal algorithms by Reif and Sen [23] is based on a polylog-time dynamic ray-shooting technique for monotone subdivisions.

1.2 Our results

In this paper we present a dynamic data structure for connected subdivisions that supports ray-shooting and shortest-path queries. The repertory of update operations includes insertion and deletion of vertices and edges. This repertory is *complete* for connected subdivisions, in that any connected planar subdivision $\mathcal{S}$ can be constructed "from scratch" using only these operations (with each intermediate subdivision being a connected planar subdivision). The space requirement for our structure is $O(n)$, and the worst-case running time for all operations (queries and updates) is $O(\log^2 n)$. Our data structure outperforms the previous best data structure [5] for this problem by a $\log n$ factor in all the complexity measures (space, query times, and update times). It is also conceptually simple, as it is based on dynamically maintaining a "balanced" geodesic triangulation of each region in the subdivision so as to answer ray-shooting queries by a simple walk-through strategy.

2 Geodesic Triangulations

A *geodesic path* (or *shortest path*) between two points p and q inside a simple polygon P is the shortest path joining p and q that does not go outside P. We denote such a path as $\pi(p, q)$. Given three vertices u, v, and w of a simple polygon P, which occur in counterclockwise order around P, the *geodesic triangle* $\tilde{\triangle}uvw$ they determine is the union of the paths $\pi(u,v)$, $\pi(v,w)$, and $\pi(w,u)$. (See Figure 1.) Let τ be such a geodesic triangle $\tilde{\triangle}uvw$. In general, τ will consist of a simple polygon made up of three concave chains and three polygonal chains emanating away from the three vertices where the concave chains are joined (see Figure 1). We refer to the inner polygonal region as the *deltoid* region of τ, due to its resemblance to the well-known quartic curve [17], and we refer to the three chains emanating out from the deltoid region as *tails*. (These definitions differ somewhat from those of Chazelle *et al.* [2], for their "geodesic triangle" is what we are calling a "deltoid region," and our "geodesic triangle" is something they refer to as a "kite.") Note that a geodesic triangle may actually be just a path (e.g., if $\pi(w,u) = \pi(u,v) \cup \pi(v,w)$), in which case it would have an empty deltoid region and one empty tail. Let $P[u,v]$ denote the counterclockwise-oriented subchain of P from a vertex u to a vertex v. Note that, by the Jordan curve theorem, we also have the following (see Figure 1):

Observation 2.1: *Let π_u, π_v, and π_w, respectively denote the tails of geodesic triangle $\tilde{\triangle}uvw$ incident upon u, v, and w (which occur counterclockwise around polygon P). There are no vertices of $P[v,w]$ in π_u, no vertices of $P[w,u]$ in π_v, and no vertices of $P[u,v]$ in π_w.*

A *geodesic triangulation* of a simple polygon P is a decomposition of P's interior into geodesic triangles whose boundaries do not cross. (See Figure 2.a.) Two geodesic triangles may have a non-empty intersection, however, if portions of their respective boundaries overlap.

A geodesic triangulation is combinatorially and topologically like a triangulation of a convex polygon. Hence, it immediately induces a degree-3 tree T, where each node in T corresponds to a geodesic triangle and we join the node corresponding to $\tilde{\triangle}uvw$ with the node corresponding to $\tilde{\triangle}xyz$ if they share two of their vertices (e.g., if $x = v$ and $z = w$). (See Figure 2.b.) The nodes of T corresponding to the geodesic triangles whose boundaries are intersected by some ray in P will always form a path in T. We say that a geodesic triangulation is *balanced* if the diameter of T is $O(\log |T|)$. As observed by Chazelle *et al.* [2], one can efficiently perform a ray-shooting query for a ray $\vec{r}$ by a simple "walk-through" strategy, where one first locates the geodesic triangle whose interior contains the starting point for $\vec{r}$ and one then iteratively traverses geodesic triangles along the direction $\vec{r}$ until one hits the boundary of P. The geodesic triangles traversed correspond to nodes that form a subset of nodes in a path of T; hence, this strategy crosses at most $O(\log |T|)$ geodesic triangles in a balanced geodesic triangulation.

Our approach is to maintain a geodesic triangulation of polygon P so that its dual tree T is a balanced binary tree—in particular, a red-black tree [7, 13, 20, 25]. Sleator, Tarjan, and Thurston [24] observe that, given a triangulation of a convex polygon P, then any two adjacent triangles $\triangle uvw$ and $\triangle wzu$ in this triangulation can be replaced by the triangles $\triangle vwz$ and $\triangle zuv$, and such a "diagonal swap" corresponds to a rotation in the tree dual to this triangulation. We extend this result to geodesic triangulations, and observe likewise that a rotation in T will correspond to swapping of diagonals determined by two adjacent geodesic triangles, i.e., it corresponds to replacing adjacent geodesic triangles $\tilde{\triangle}uvw$ and $\tilde{\triangle}wzu$ by

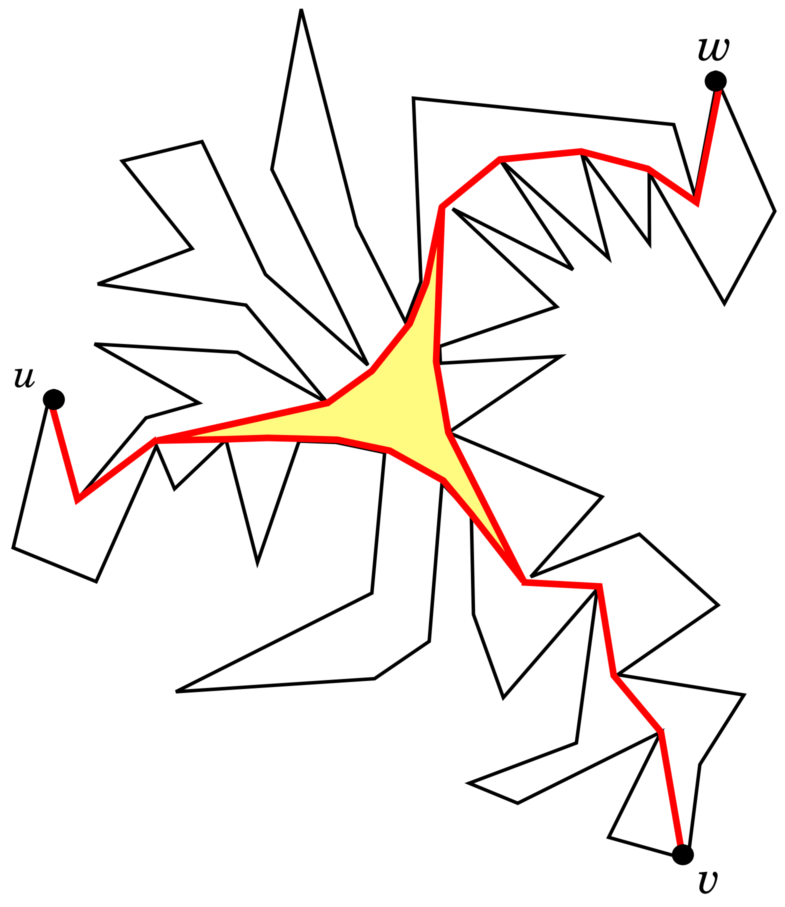

Figure 1: Geodesic triangle $\tilde{\triangle}uvw$. The deltoid region is shaded.

the geodesic triangles $\tilde{\triangle}vwz$ and $\tilde{\triangle}zuv$. We show that vertex insertion and deletion can be implemented by inserting and deleting edges and vertices in T, and that edge insertions and deletions can be performed using operations on T that are analogous to a sequence of *split* and *splice* operations. If we maintain geodesic paths in auxiliary structures, then we can perform each rotation and insertion in T in $O(\log n)$ time (using splits and splices on the geodesic paths involved in the rotation). We therefore achieve a running time for queries and updates that is $O(\log^2 n)$ in the worst case.

3 Red-Black Trees

Since our structure is built using the red-black tree data structure as a schematic, let us begin by reviewing this structure and showing how simple list updates can be performed using it. We use the formulation of Tarjan [25]. For any node v in a rooted tree T, let $p(v)$ denote the parent of v in T. Likewise, for any tree T, let $r(T)$ denote the root node of T. A *red-black tree* is a rooted binary search tree T whose nodes are assigned integer ranks that obey the following constraints:

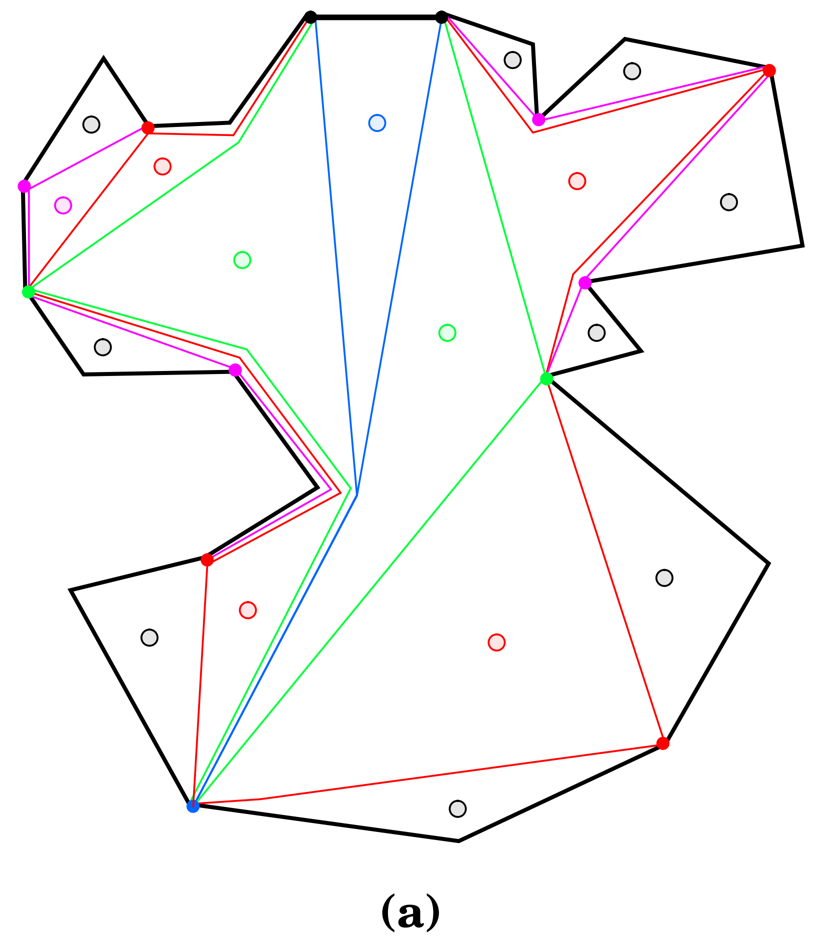
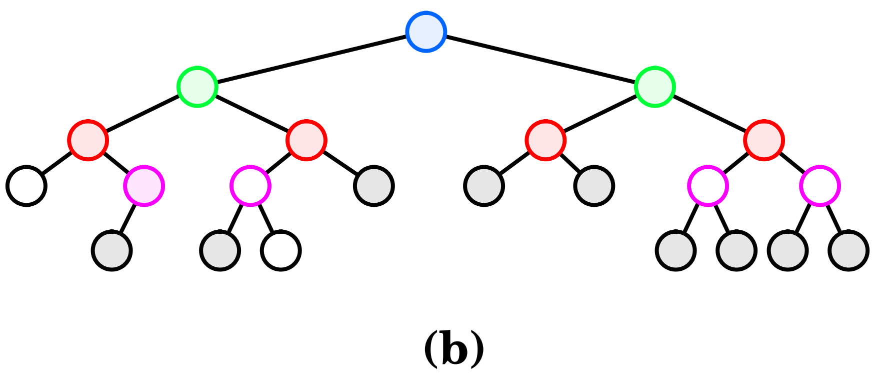
Figure 2: (a) Geodesic triangulation of a polygon. (b) Dual tree associated with the geodesic triangulation, where the white-filled nodes denote geodesic triangles with an empty deltoid region.

1. If v has a *nil* child pointer, then $rank(v) = 1$ and v's *nil* child pointer is viewed as pointing to a node with rank 0.

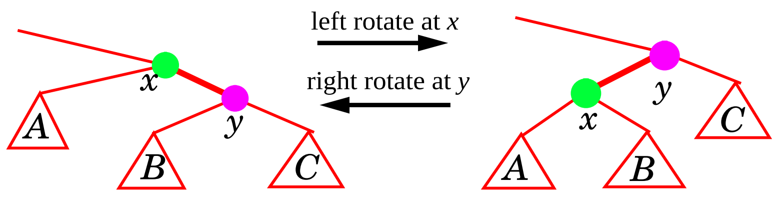

Figure 3: Illustrating left and right rotations.

2. If v is a node with a parent, then $rank(v) \leq rank(p(v)) \leq rank(v) + 1$.

3. If v is a node with a grandparent, then $rank(v) < rank(p(p(v)))$.

A node v is called *black* if $rank(p(v)) = rank(v) + 1$ or v is the root; v is *red* otherwise (i.e., if $rank(p(v)) = rank(v)$). Let n be the number of nodes of T. It is easy to see that $rank(v)$ is proportional to the logarithm of the number of descendents of v, so that $rank(r(T)) = O(\log n)$.

3.1 Tree updates

Given a node v in T, we recall that in a *split*(v) operation one divides binary tree T into binary trees T_1, and T_2, where T_1 contains the nodes of T with in-order rank smaller than v, and T_2 contains the nodes with larger in-order rank (we don't actually maintain in-order labels, however; we just use this ordering notion to describe the relative positions of nodes in red-black trees). A *splice*(T_1, v, T_2) operation is the inverse of a *split* operation. Tarjan [25] shows that red-black trees support the *split* and *splice* operations in $O(\text{rot}(n) \log n)$ time, where $\text{rot}(n)$ denotes the time complexity of performing a rotation in T. Recall the definition of a *left rotation* at a node x in T, where we let A denote the subtree rooted at x's right child, we let y denote x's right child, we let B and C respectively denote the subtrees rooted at y's left and right child, and we transform T so that x has A and B as subtrees rooted at its left and right child, with x now being y's left child (and x's old parent now being y's parent) so as to have C as the subtree rooted at y's right child. A *right rotation* at a node y is defined symmetrically (we assume that binary trees are always oriented, so that the notions of "left child" and "right child" are well defined). (See Figure 3.) Tarjan's methods are based upon using left and right rotations and various pointer manipulations to update red-black trees subject to split and splice operations. Note that in the standard red-black tree setting $\text{rot}(n)$ is $O(1)$, but this will not be the case in our application, where auxiliary data structures need to be updated after each rotation.

In our use of red-black trees, we must assume that each internal node has degree 3; thus, let us assume that the root of a red-black tree T actually has a parent, which is a degree-one "dummy node." In addition, we desire that our tree-modification operations be based strictly on the use of tree rotations, and not use the more general pointer changing as is used in the standard implementations [7, 13, 20, 25]. Fortunately, such implementations are easy to come by, as we show next.

3.2 Non-destructive tree updates

In this subsection we describe how to perform all red-black tree update operations using rotations only. We begin with the splice operation. Tarjan's method [25] for performing a *splice* of trees T_1 and T_2 depends upon the relative ranks of T_1 and T_2. If they are the same rank, then one simply creates a new node z, setting $r(T_1)$ and $r(T_2)$ as its left and right children, respectively. Otherwise (without loss of generality assume that $rank(r(T_1)) > rank(r(T_2))$), one searches down the rightmost path in T_1 from $r(T_1)$ to find a node v whose rank equals $rank(r(T_2))$. Then one creates a new node z replacing v in T_1, setting z's left child to v and z's right child to $r(T_2)$. One then sets $rank(z) = rank(v)+1$ and proceeds back up T_1 to perform any necessary rank increases and rotations needed to keep the tree balanced. The total time required is $O(\text{rot}(n) + |rank(r(T_1)) - rank(r(T_2))|) = O(\text{rot}(n) + \log n)$, and it results in a tree of rank at most $\max\{rank(r(T_1)), rank(r(T_2))\} + 1$.

In our implementation we assume the node z is already created and that T_1 and T_2 are initially the children of z. If $rank(T_1) = rank(T_2)$, then we are done before we start. Otherwise (again suppose $rank(T_1) > rank(T_2)$), we perform a sequence of right rotates at the parent of $r(T_2)$ until we reach a point where $r(T_1)$ and its sibling have the same rank. We then complete the procedure as in Tarjan's implementation [25]. Clearly, the total time needed is $O(\text{rot}(n)(|rank(T_1) - rank(T_2)| + 1)) = O(\text{rot}(n) \log n)$.

Likewise, let us describe a *non-destructive* version of a *split* of tree T at a node $v \in T$, which returns a tree whose left subtree is a red-black tree for the elements left of v in T, and whose right subtree is rooted at a node s, where s's left child is v and s's right subtree is a red-black tree for the elements right of v in T (see Figure 4). Recall that a *left* (resp., *right*) fringe node for a leaf-to-root path π is a node that is a left (resp., right) child of a node on π but is itself not on π. We perform a nondestructive split on T by performing a series of rotations to move v up to its final position in T. Any time a pair of nodes on the left fringe (resp., right fringe) of the path from the root of T to v become siblings during this series of rotations, we perform a non-destructive *splice* of their respective subtrees (see Figure 4.b), as described above. This allows us to replace each splice in Tarjan's implementation [25] of split with a non-destructive splice.

The analysis for our implementation of this operation follows by a simple adaptation of Tarjan's analysis. By the properties of rank in a red-black tree, the sum of the running times for performing the splices of subtrees to the left (resp., right) of v telescope so as to sum to $O(\text{rot}(n) \log n)$. Thus, the total time for performing a non-destructive split is $O(\text{rot}(n) \log n)$.

Finally, we must contend with the fact that the root of our red-black tree implementations has a "dummy node" parent and the leaf nodes represent objects that belong to a circular order (not a strict linear order). For any leaf node v, we define an operation *evert*(v) on a red-black tree T that creates v as the new dummy node parent of $r(T)$. In particular, *evert*(v) is implemented as follows (see Figure 5):

1. Perform a split at v, letting T_1 (resp., T_2) be the red-black tree storing the leaves of T with in-order rank smaller (resp., larger) than the in-order rank of v in T.

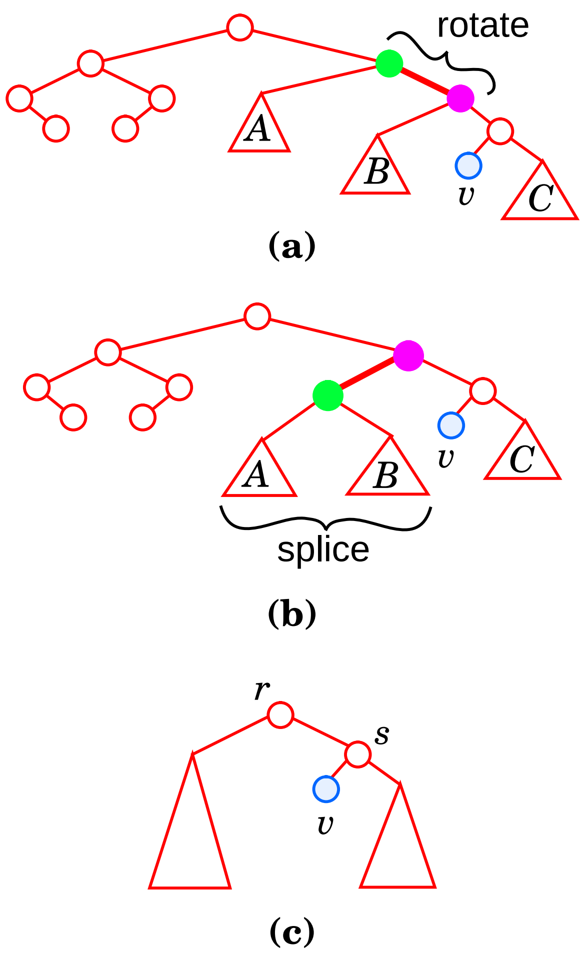

Figure 4: Schematic illustration of a nondestructive split in a red-black tree: (a–b) intermediate rotations; (c) the final tree.

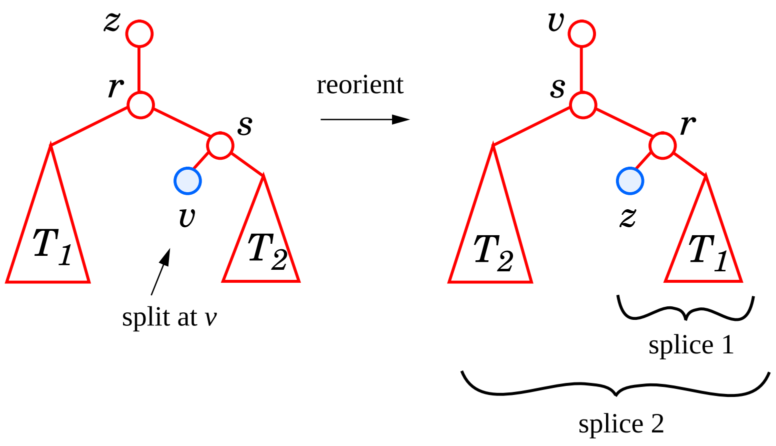

Figure 5: Illustrating the implementation of the evert operation.

2. Let z be the current dummy node parent of $r = r(T)$, and let s denote the parent of v (so s is the right child of r). Reorient T so that T now has v as its dummy node root-parent, with $s = r(T)$ being the child of v, so that s has $r(T_2)$ as its left child and r as its right child, with r having z as its left child and $r(T_1)$ as its right child.

3. Perform a splice of z and T_1, resulting in a red-black tree, T_1' rooted at r.

4. Perform a splice of T_2 and T_1'.

This clearly can be implemented in $O(\text{rot}(n) \log n)$ time.

We are now ready to describe our data structure.

4 The Ray-Shooting Data Structure

Let $\mathcal{S}$ be a connected subdivision, represented using some dynamically-updatable representation of an embedded planar graph. This can be done, for example, by a simple modification of the doubly-connected edge lists structure [22], where we store the edges of each face f in a red-black tree $B(f)$ ordered around f. This representation allows us to split a face f in $O(\log n)$ time or, alternatively, to splice two faces f_1 and f_2 along a removed common edge in $O(\log n)$ time. In this section we describe our data structure for performing ray shooting queries in $\mathcal{S}$.

4.1 The primary structure

As mentioned in the introduction, the main component of our data structure for $\mathcal{S}$ is a geodesic triangulation of each region of $\mathcal{S}$ (see Fig. 2.a). With each region P of $\mathcal{S}$, we also store the dual tree T of the geodesic triangulation (see Fig. 2.b). Each internal node μ in T corresponds to a geodesic triangle and we join the node corresponding to $\tilde{\triangle}uvw$ with the node corresponding to $\tilde{\triangle}xyz$ if they share two of their vertices (e.g., if $x = v$ and $z = w$). At each such μ we store pointers to the vertices u, v, and w of μ's associated geodesic triangle. Each leaf corresponds to an edge of P and is joined to the (parent) geodesic triangle that has this edge on its boundary. In particular, if one of the edges of a geodesic triangle τ is also an edge of P, then we say that τ is a *border* triangle, and, for each such border triangle τ, we add an adjacency in T from the node associated with τ to a (leaf) node associated with the edge of P on τ. Thus, the counterclockwise orientation of the edges around P determines the left-to-right (in-order) orientation of the leaves of T. In addition, we distinguish some border triangle ρ in P as the *root* triangle, so as to root T at the node associated with ρ. We associate with this "dummy parent" node a pointer to the record in the doubly-connected edge list structure repsenting P in $\mathcal{S}$. The main idea of our primary structure, then, is to maintain this rooted tree T as a red-black tree [7, 13, 20, 25], ignoring the (dummy) leaf node associated with ρ.

4.2 The secondary point location structure

As a secondary data structure we maintain a dynamic point location data structure on the deltoid regions determined by the geodesic triangulations of all the faces in $\mathcal{S}$. In particular, we use the structure of Goodrich and Tamassia [9], which uses $O(n)$ space, supports point location queries in $O(\log^2 n)$ time, edge insertion and deletion in $O(\log n)$ time, and vertex insertion and deletion in $O(\log n)$ time as well. The only caveat to using this structure is that it requires each face in the subdivision to be monotone (say, with respect to the x-axis). That is, it requires the underlying subdivision to be *monotone*. Of course, a deltoid region need not be monotone. Nevertheless, we have

Lemma 4.1: *The geodesic triangulation of a connected subdivision can be refined to a monotone subdivision by inserting at most one edge in each deltoid region.*

Proof: The deltoid region consists of three concave chains. If two of these chains were not monotone, then (since these two chains must be incident on the same vertex) the third chain could not be concave and still define a closed region with the other two chains. Since this third chain is concave, it can be divided into two monotone chains by splitting at some vertex v. Therefore, by connecting v to one of the other two chains we decompose this deltoid region into two monotone polygons. Doing this for each deltoid region, then, refines $\mathcal{S}$ into a monotone subdivision $\mathcal{S}'$. $\square$

Thus, our secondary structure consists of the dynamic point location of Goodrich and Tamassia [9] built upon the union of the deltoid regions in all the geodesic triangles in $\mathcal{S}$, together with at most one edge per deltoid region so as to make each face in the resulting subdivision $\mathcal{S}'$ monotone with respect to the x-axis.

4.3 The tertiary deltoid structures

The final component of our data structure is a tertiary structure built for the deltoid regions. In particular, for each deltoid region δ, we maintain each of the three concave chains for δ in a balanced tree structure (e.g., a red-black tree [7, 13, 20, 25]). Each internal node in such a tree corresponds to a subchain of a concave chain and stores the length of the associated subchain. In fact, let us assume for the remainder of this paper that an augmented balanced binary tree, called *chain tree*, will be used to represent any polygonal chain, where the leaves are associated with the edges, and the internal nodes with the vertices of the chain. Each node also corresponds to a subchain and stores its length. It should be clear that this information can be updated in $O(1)$ time per rotation, so that splitting or splicing two chain trees takes logarithmic time. With this representation, it is possible to find the two tangents from a point to a convex chain and the four common tangents between two convex chains in logarithmic time [22].

We store a double-link between the root of each tertiary tree t and the node μ in T associated with the geodesic triangle with deltoid region δ that has the edges of t as one of its concave chains. In addition, for any edge e stored in a chain tree t representing a chain on deltoid region δ, if e is not an edge of $\mathcal{S}$ (i.e., it was added to form the geodesic triangulation), then we store a pointer from e's record in t to e's record in the tertiary chain tree representing the deltoid region on the other side of e (i.e., the side not in the interior of δ).

Our entire data structure, $\mathcal{D}$, then, consists of the primary geodesic triangulation structures, the secondary point location structure, and the tertiary deltoid structures.

Lemma 4.2: *The data structure $\mathcal{D}$ requires $O(n)$ space.*

Proof: The primary structure requires only $O(n)$ space, since it stores $O(1)$ amount of information for each geodesic triangle, and a geodesic triangulation is topologically equivalent to a triangulation of a convex polygon. The secondary structure requires $O(n)$ space, since the total number of edges in the subdivision $\mathcal{S}$ is $O(n)$ and we are building the data structure of Goodrich and Tamassia [9] on a subdivision that is a subgraph of a triangulation of $\mathcal{S}$. This also implies that the total number of edges defined by all the deltoid regions is $O(n)$, implying that the total space used by all our tertiary structures is also $O(n)$. $\square$

4.4 Ray shooting

Suppose we have data structure $\mathcal{D}$ for our connected subdivision $\mathcal{S}$, and let $\vec{r}$ be a query ray for which we wish to perform a ray shooting query. See Figure 6. We begin by performing a point location for the origin p of $\vec{r}$ using the secondary point location structure. This takes time $O(\log^2 n)$ [9] and identifies a deltoid region δ containing p. By then following parent pointers up in a tertiary chain tree from any edge in δ we can identify the node μ in T that is associated with a geodesic triangle having δ as its deltoid region.

This sets up a generic *local ray shoot*, where we are given a pointer to a node μ in T representing a geodesic triangle with deltoid region δ and a ray $\vec{r}$ whose endpoint is inside δ (possibly even on the boundary of δ), and we wish to locate the edge e of δ that $\vec{r}$ hits first. We can identify this edge by using the tertiary structures for the convex chains of δ to determine, in $O(\log n)$ time, the first edge e on the boundary of δ where $\vec{r}$ exits (for it amounts to a simple binary search). If e is an edge of $\mathcal{S}$, then we are done, for we have located the edge of $\mathcal{S}$ that $\vec{r}$ hits. If e is not an edge of $\mathcal{S}$, on the other hand, then we follow the pointer from the current record for e to the record for e in the adjacent deltoid region, δ'. By then following parent pointers up this chain tree we can identify the node in T that has this face as its deltoid region. This, of course, sets up another instance of a local ray shoot; hence, we can now recurse on δ'.

Each local ray shoot test requires $O(\log n)$ time, as does the extra computation needed to set up the next ray shoot test, if needed. Since this query traverses a subset of nodes along a path in T (e.g., see Figure 6), the total time to perform such a ray shooting query is at most $O(\log^2 n)$. Therefore, we have

Lemma 4.3: *A ray-shooting query in $\mathcal{D}$ takes $O(\log^2 n)$ time.*

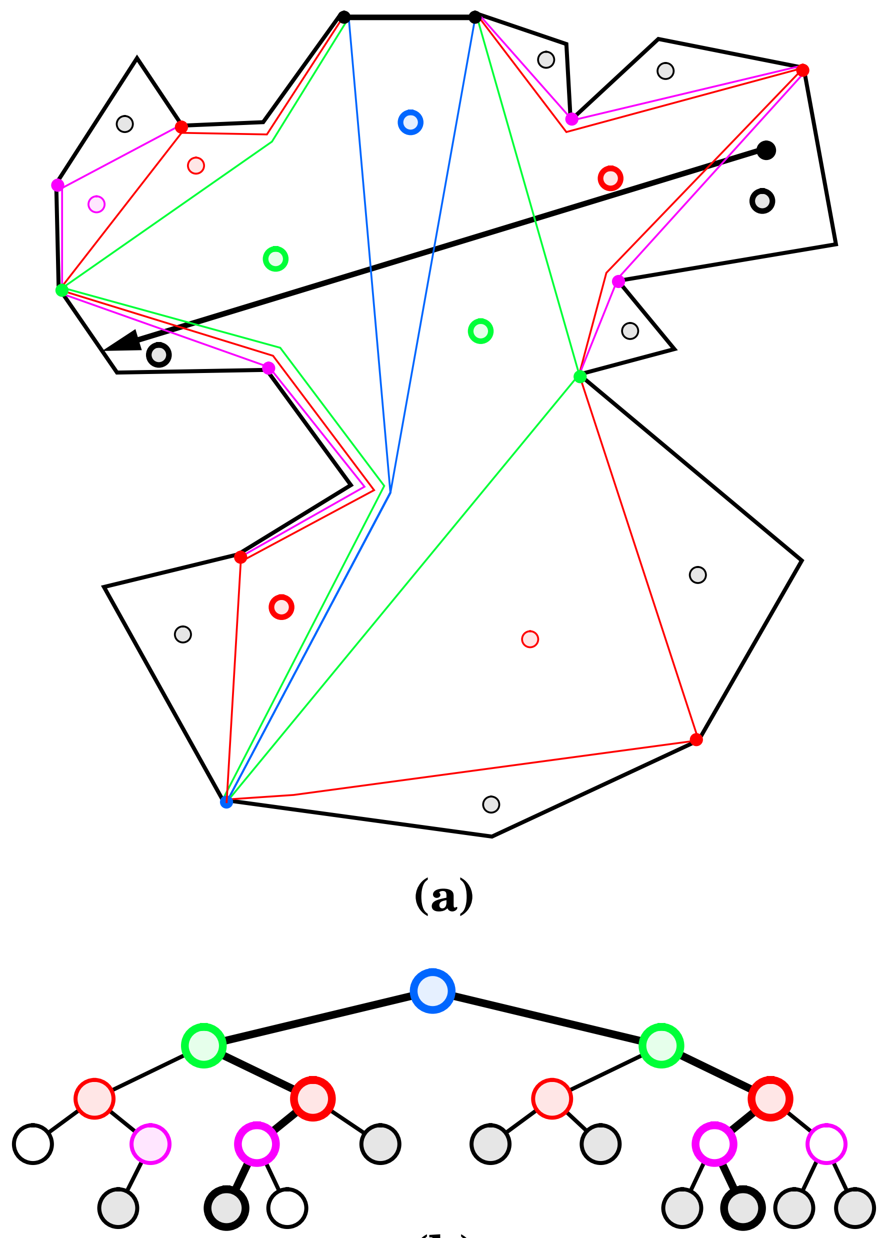

Figure 6: Illustration of a ray shooting query: (a) geodesic triangulation; (b) path in the dual tree visited during the execution of the query algorithm

5 Dynamic Balanced Geodesic Triangulations

In this section we show how to maintain the data structure $\mathcal{D}$ while performing edge insertion and deletion as well as vertex insertion and deletion. In particular, we define the following update operations on a connected subdivision $\mathcal{S}$:

InsertEdge$(e, v, w, R; R_1, R_2)$: Insert edge $e = (v, w)$ into region R such that R is partitioned into two regions R_1 and R_2.

RemoveEdge$(e, v, w, R_1, R_2; R)$: Remove edge $e = (v, w)$ and merge the regions R_1 and R_2 formerly on the two sides of e into a new region R.

InsertVertex$(v, e; e_1, e_2)$: Split the edge $e = (u, w)$ into two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ by inserting vertex v along e.

RemoveVertex$(v, e_1, e_2; e)$: Let v be a vertex with degree two such that its incident edges $e_1 = (u, v)$ and $e_2 = (v, w)$, are on the same straight line. Remove v and merge e_1 and e_2 into a single edge $e = (u, w)$.

AttachVertex$(v, e; w)$: Insert edge $e = (v, w)$ and degree-one vertex w inside some region R, where v is a vertex of R.

DetachVertex(v, e): Remove a degree-one vertex v and edge e incident on v.

The above repertory of operations is complete for connected subdivisions. Also, *AttachVertex* and *DetachVertex* can be simulated by a ray shooting query followed by a sequence of $O(1)$ *InsertVertex*, *RemoveVertex*, *InsertEdge*, and *RemoveEdge* operations [5]. For example, to perform *AttachVertex* at a vertex v in $\mathcal{S}$ to attach a node w along edge $e = (v, w)$, we could perform a ray shooting query to identify the point p on the edge f of $\mathcal{S}$ first hit by the ray emanating from v in the direction towards w. We could then perform an *InsertVertex* of p on f, an *InsertEdge* for (v, p), an *InsertVertex* for w on (v, p), and a *RemoveEdge* for the edge (w, p). Thus, we will not discuss further the implementation of operations *AttachVertex* and *DetachVertex*.

The only restrictions we place on these operations is that they should be applied in a way that does not violate the planarity or connectivity of $\mathcal{S}$. In the subsections that follow we describe how we can implement each of the above update operations (other than *AttachVertex* and *DetachVertex*).

5.1 Rotations

Before we describe our implementations for these operations, however, we must describe how we implement rotations in our primary data structure, the red-black tree T, since the red-black tree update operations of (non-destructive) *splice*, *split*, and *evert* are all built upon rotations.

The important observation is that a rotation in T corresponds to a swap of diagonals in two adjacent geodesic triangles (see Figures 7 and 8). We can determine the edges involved in such a diagonal swap by querying the tertiary structures associated with the deltoid regions of the associated geodesic triangles. If these deltoid regions share an edge, then we must compute $O(1)$ common supporting or cross tangents so as to determine the new diagonal edge (see Figure 8). This can easily be done in $O(\log n)$ time using a well-known

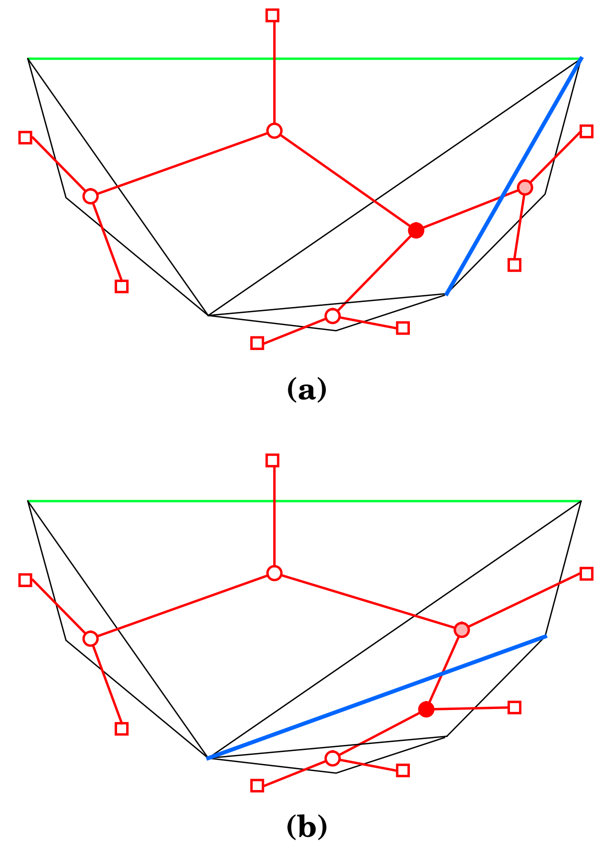

Figure 7: A swap of diagonals in a triangulation of a convex polygon and the corresponding rotation in the dual tree: (a) before the swap; (b) after the swap.

binary search approach (e.g., see Preparata and Shamos [22]). If these deltoid regions do not share an edge, then the diagonal swap simply involves identifying the deltoid regions with their new geodesic triangles. In any case the geodesic triangulation is modified with $O(1)$ *InsertEdge*/*RemoveEdge* operations, and the boundaries of the geodesic triangles are modified by $O(1)$ split/splice operations (see Figure 8). Thus, a rotation in T requires $O(\log n)$ total time, i.e., $\mathrm{rot}(n)$ is $O(\log n)$.

5.2 Vertex insertion and deletion

Operations $\mathit{InsertVertex}(v, e; e_1, e_2)$ and $\mathit{RemoveVertex}(v, e_1, e_2; e)$ correspond to the insertion/deletion of a node in the dual trees associated with the regions that share edge e. The geodesic triangulation is modified by two *InsertVertex*/*RemoveVertex* operations. The boundaries of the geodesic triangles are modified by two insertions/deletions, and, if the edge e is the dual to the dummy parent of the root of the dual tree, a split/splice operation.

Specifically, let us examine the computations required to implement the operation $\mathit{InsertVertex}(v, e; e_1, e_2)$, for the implementation of $\mathit{RemoveVertex}(v, e_1, e_2; e)$ is symmetric. In order to maintain a geodesic triangulation we let the adding of vertex v on edge $e = (u, w)$

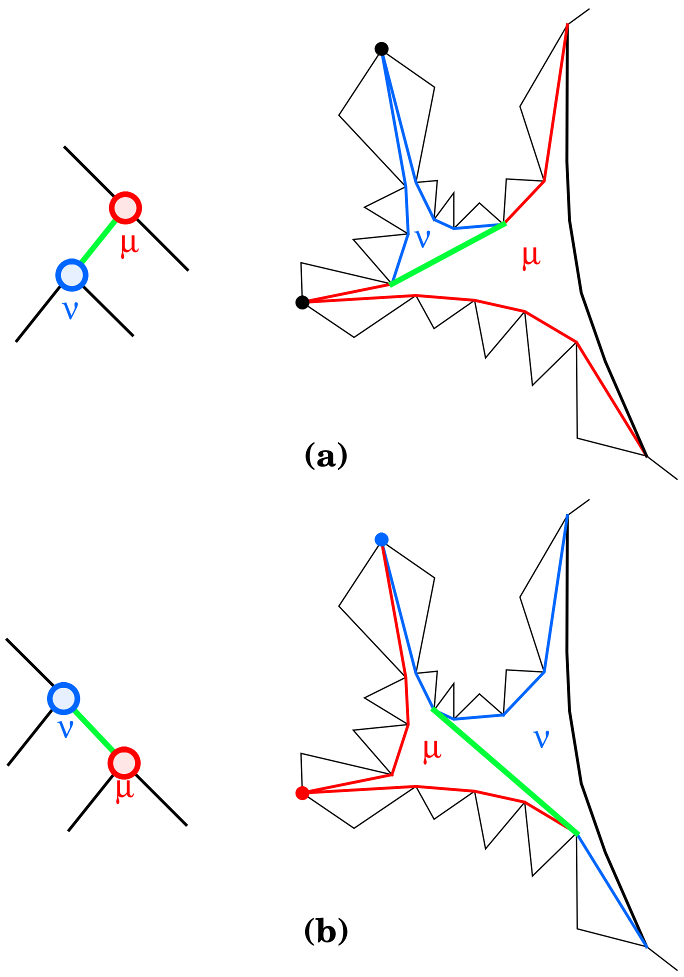

Figure 8: Update of a geodesic triangulation after a rotation in the dual tree: (a) before the rotation; (b) after the rotation.

create geodesic triangles $\tilde{\triangle}uvw$ and $\tilde{w}vu$, respectively, in the two polygonal regions incident on edge e. Since these geodesic triangle insertions are similar, let us concentrate on the insertion of $\tilde{\triangle}uvw$. Its insertion in polygonal region P is equivalent to a leaf-node insertion in the dual tree T for P. If e is not dual to the dummy parent of the root of T, then we can then peform the rotations (and occompanying diagonal swaps) needed to keep T as a balanced red-black tree. If, on the other hand, e is the dual edge for the dummy parent of the root of T, then we let (v, w) become the new dual to the dummy node parent of T, and we perform the rotations (and diagonal swaps) needed to perform a splice of the single node tree dual the edge (u, v) and the tree T. In either case, we can rebalance the dual tree for the geodesic triangulation of P using $O(\log n)$ rotations/diagonal swaps, each of which requires at most $O(\log n)$ time.

Lemma 5.1: *Operations InsertVertex and RemoveVertex each take $O(\log^2 n)$ time.*

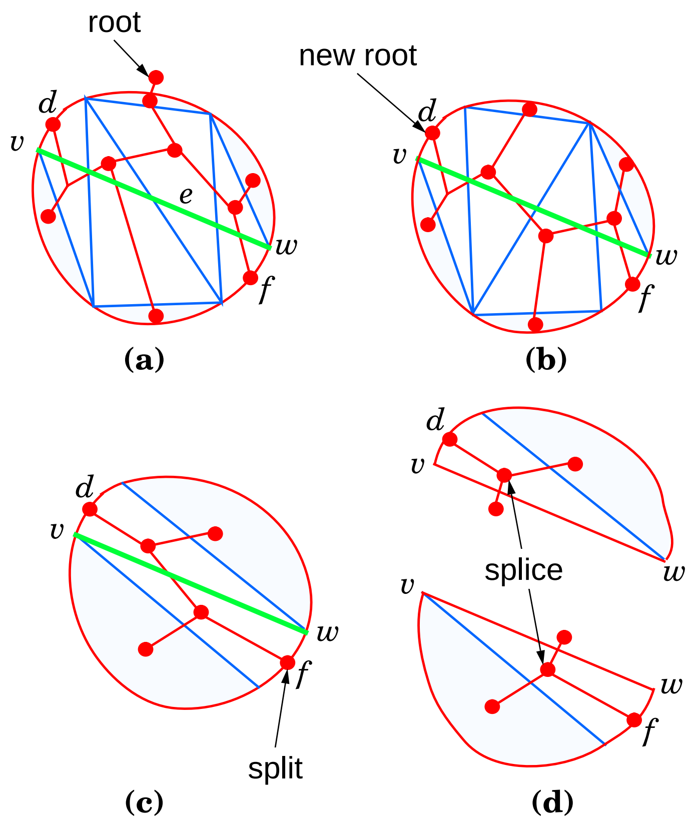

Figure 9: Schematic illustration of operation $\mathit{InsertEdge}(e, v, w, R; R_1, R_2)$: (a) initial geodesic triangulation and dual tree; (b) eversion; (c) split; (d) splice.

5.3 Edge insertion and deletion

Let us next consider edge insertion and deletion, and begin our discussion with the insertion case. We begin by noting that the insertion of an edge e in polygonal region R of $\mathcal{S}$ could intersect $O(\log n)$ edges of our geodesic triangulation of R. Nevertheless, the operation $\mathit{InsertEdge}(e, v, w, R; R_1, R_2)$ can be implemented in $O(\log^2 n)$ time as follows (see Figure 9). Let d and f be edges of R such that d is incident to v and f is incident to w, with d and f being on opposite sides of e (i.e., d and f will be separated after e is inserted).

We begin our implementation of the insertion of e by everting the tree T at the leaf for d, resulting in a geodesic triangulation of R corresponding to a red-black tree T' rooted at d. We then perform a non-destructive split on the dual tree T' at f so that the edge e is the diagonal between the geodesic triangles corresponding to the parent and grandparent of f, respectively, which gives us a new dual tree T''.

We may then insert the edge e, cutting T'' at the edge dual to e. This results in two new regions R_1 and R_2 with corresponding dual trees T_1 and T_2. Notice that the root of T_1 (resp., T_2) has as one of its children the root of a red-black tree and as its other child the node d (resp., f). We complete the construction, then, by performing a *splice* on the two children of the root of T_1 and the root of T_2, respectively.

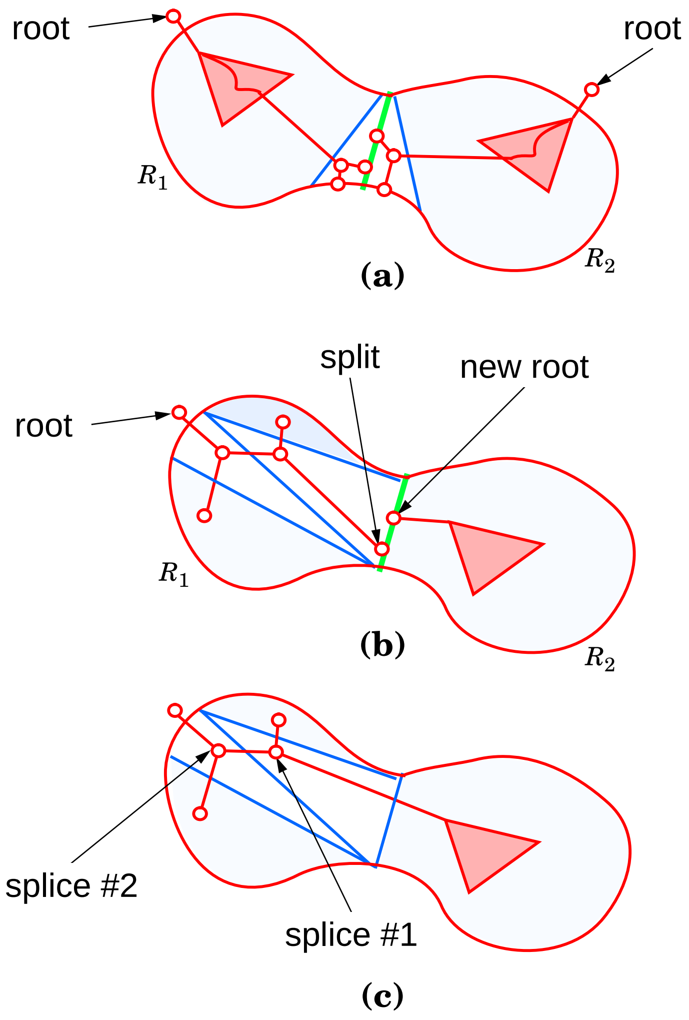

Figure 10: Schematic illustration of operation $\mathit{RemoveEdge}(e, v, w, R; R_1, R_2)$: (a) initial geodesic triangulations and dual trees; (b) eversion and split; (c) splices.

Note that this construction requires that we perform $O(1)$ *evert*, *split*, and *splice* operations on the dual trees for R_1 and R_2. Each red-black tree rotation required to implement these operations in T requires $O(\log n)$ time using the tertiary chain structures. Thus, this edge insertion can be implemented in $O(\log^2 n)$ time.

Let us therefore next consider the operation $\mathit{RemoveEdge}(e, v, w, R_1, R_2; R)$ (see Figure 10). Let T_1 and T_2 be the dual trees for the geodesic triangulations of R_1 and R_2, respectively. We begin by performing an *evert* operation on T_2 to make the leaf corresponding to e become the root for this new tree T_2' in R_2. We then perform a non-destructive *split* on T_1 at the leaf in T_1 corresponding to e, which gives us a new tree T_1'. We then conceptually merge R_1 and R_2 by replacing the leaf for e in T_1' with the root of T_2'. That is, if we let r denote the root of T_2', then we replace the leaf for e by r.

We complete the construction by performing a *splice* at the (new) parent for r, and then another *splice* at the grandparent of r. This gives us a balanced tree for the entire region

R. Notice that the implementation of this operation required $O(1)$ *evert*, *split*, and *splice* operations. Thus, it too can be implemented in $O(\log^2 n)$ time.

Lemma 5.2: *Operations InsertEdge and RemoveEdge each take $O(\log^2 n)$ time.*

6 Shortest Path Queries

In this section we show how to extend our approach so as to efficiently answer shortest path queries in $\mathcal{S}$. In this case we are given two query point p and q and we wish to determine the shortest path between p and q that does not cross any edges of $\mathcal{S}$. We may assume, without loss of generality, that p and q belong to the same region in $\mathcal{S}$, since we can test if this is not the case in $O(\log^2 n)$ time by point location [9]. So, suppose we are given two query points p and q in a region P of $\mathcal{S}$, and we wish to perform a shortest path query for the pair (p, q). We consider two variations of this query: reporting the length of the path, and reporting all the edges of the path.

In order to support shortest path queries, we extend our data structure so as to store tails of geodesic triangles. Specifically, we modify our data structure, so that for each node μ of tree T, in addition to information already stored at μ, we also store at μ a (possibly null) chain tree representing the tail of μ's geodesic triangle not shared by the geodesic triangle stored at μ's parent. By Observation 2.1, this amounts to a simple application of a space-saving technique pioneered by Lee and Preparata [18], where one stores shared edges at the highest node in a tree where they appear. This remains an $O(n)$-sized data structure, as can be established by a simple modification of the proof of Lemma 4.2. In addition,

Lemma 6.1: *Rotations in T can be performed in $O(\log n)$ time.*

Proof: The method for now performing rotations in T is identical to our earlier implementation, except that now we must maintain for each μ in T, a chain tree representing the tail of μ's geodesic triangle not shared by the geodesic triangle stored at μ's parent. In order to maintain this invariant during a rotation we may need to perform a few additional splits and splices, but they still take just $O(\log n)$ time. This is due to the fact that in the diagonal swap of two geodesic triangles t and t' whose dual nodes are involved in a rotation, the tails of the resulting geodesic triangles can be decomposed into $O(1)$ portions of tails and deltoid chains from t and t'. □

Having established that rotations in T can still be implemented in $O(\log n)$ time even in this augmented structure, we immediately have that all the update operations described above run in the same time bounds as before. So, we have only to describe how we perform a shortest path query for two given points p and q.

If p and q are vertices of R, the geodesic path algorithm is as follows: First, we evert T so that the dummy node of T is associated with an edge incident on p. This takes $O(\log^2 n)$ time. Next, perform a non-destructive split at a leaf μ_q of T incident upon q to bring μ_q to be the grandchild of the root of T so that the geodesic path from p to q is the diagonal separating the geodesic triangle for μ_q from the geodesic triangle for $p(\mu_q)$ (see Figure 9, as this is very similar to our operation for edge insertion with $v = p$ and $w = q$). Now, the shortest path between p and q is a diagonal in the geodesic triangulation for R. In fact, it is a diagonal defining the boundary of the root geodesic triangle in T; hence, the length of the entire geodesic path and its k edges can be retrieved in time $O(1)$ and $O(k)$, respectively, from the chain trees for this geodesic triangle and its one non-trivial tail. Finally, after we have answered the query, we undo the above rotations to reset the data structure to its original state. The overall time complexity is $O(\log^2 n)$, plus $O(k)$ if the path is reported in addition to its length.

If p and q are not vertices of R, we "attach" them to the boundary of R by means of two horizontal ray-shootings followed by two *AttachVertex* operations, which takes $O(\log^2 n)$ time, and we apply the previous method.

Lemma 6.2: *A shortest-path query takes $O(\log^2 n)$ time to report the length of the path, plus $O(k)$ time to report the k edges of the path.*

By combining Lemmas 4.2, 4.3, 5.1, 5.2, 6.1, and 6.2, we summarize our results in the following theorem:

Theorem 6.3: *Let $\mathcal{S}$ be a planar connected subdivision with n vertices. There is an $O(n)$-space fully dynamic data structure for $\mathcal{S}$ that supports point-location, ray-shooting, and shortest-path queries in $O(\log^2 n)$ time, and operations InsertVertex, RemoveVertex, InsertEdge, RemoveEdge, AttachVertex, and DetachVertex in $O(\log^2 n)$ time, all bounds being worst-case.*

7 Conclusion

We have given a simple and efficient scheme for dynamically maintaining a connected subdivision $\mathcal{S}$ subject to ray shooting and shortest path queries. Our method is based on maintaining geodesic triangulations of each polygonal region in $\mathcal{S}$ through the use of an elegant duality between diagonal swaps between adjacent geodesic triangles and rotations in red-black trees. Since we implement each rotation in $O(\log n)$ time, this results in worst-case running times of $O(\log^2 n)$ for queries and updates.

Hershberger and Suri [16] have recently showed that one can triangulate the interior of a simple polygon using additional interior points so that any ray intersects $O(\log n)$ triangles. Applying our approach to this method would not improve the running time of updates, however, since an edge insertion would still require changing $O(\log n)$ edges, and we would still require a dynamic point location structure. Thus, this would still require $O(\log^2 n)$ time. Therefore, this still leaves the important open question of whether one can achieve $o(\log^2 n)$ time for both updates and ray shooting queries in a dynamic connected subdivision.

Acknowledgement

We would like to thank Günter Rote and the anonymous referees for several useful comments.

References

[1] P. K. Agarwal and M. Sharir. Applications of a new partitioning scheme. In *Proc. 2nd Workshop Algorithms Data Struct.*, volume 519 of *Lecture Notes in Computer Science*, pages 379–391. Springer-Verlag, 1991.

[2] B. Chazelle, H. Edelsbrunner, M. Grigni, L. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12:54–68, 1994.

[3] B. Chazelle and L. J. Guibas. Visibility and intersection problems in plane geometry. *Discrete Comput. Geom.*, 4:551–581, 1989.

[4] S. W. Cheng and R. Janardan. Algorithms for ray-shooting and intersection searching. *J. Algorithms*, 13:670–692, 1992.

[5] Y.-J. Chiang, F. P. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 44–53, 1993.

[6] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80(9):1412–1434, September 1992.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[8] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, Heidelberg, West Germany, 1987.

[9] M. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd Annu. ACM Sympos. Theory Comput.*, pages 523–533, 1991.

[10] M. T. Goodrich, M. Ghouse, and J. Bright. Generalized sweep methods for parallel computational geometry. In *Proc. 2nd ACM Sympos. Parallel Algorithms Architect.*, pages 280–289, 1990.

[11] L. J. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. *J. Comput. Syst. Sci.*, 39:126–152, 1989.

[12] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[13] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes in Computer Science, pages 8–21, 1978.

[14] J. Hershberger. A new data structure for shortest path queries in a simple polygon. *Inform. Process. Lett.*, 38:231–235, 1991.

[15] J. Hershberger. Optimal parallel algorithms for triangulated simple polygons. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 33–42, 1992.

[16] J. Hershberger and S. Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 54–63, 1993.

[17] J. D. Lawrence. *A Catalog of Special Plane Curves.* Dover Publications, New York, NY, 1972.

[18] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[19] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14:393–410, 1984.

[20] K. Mehlhorn. *Sorting and Searching*, volume 1 of *Data Structures and Algorithms.* Springer-Verlag, Heidelberg, West Germany, 1984.

[21] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[22] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[23] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithm and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

[24] D. D. Sleator, R. E. Tarjan, and W. P. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *J. Amer. Math. Soc.*, 1:647–682, 1988.

[25] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* Society for Industrial Applied Mathematics, 1983.